Multi-Objective Testing Resource Allocation under Uncertainty

Roberto Pietrantuono, Pasqualina Potena, Antonio Pecchia, Daniel Rodriguez, Stefano Russo, Luis Fernandez

Abstract—Testing resource allocation is the problem of planning the assignment of resources to testing activities of software components so as to achieve a target goal under given constraints. Existing methods build on Software Reliability Growth Models (SRGMs), aiming at maximizing reliability given time/cost constraints, or at minimizing cost given quality/time constraints. We formulate it as a multi-objective debug-aware and robust optimization problem under uncertainty of data, advancing the state-of-the-art in the following ways. Multi-objective optimization produces a set of solutions, allowing to evaluate alternative trade-offs among reliability, cost and release time. Debug awareness relaxes the traditional assumptions of SRGMs – in particular the very unrealistic immediate repair of detected faults – and incorporates the bug assignment activity. Robustness provides solutions valid in spite of a degree of uncertainty on input parameters. We show results with a real-world case study.

I. INTRODUCTION

A. Motivations

Testing is an essential activity to improve quality of software products, impacting their production cost and time-to-market. Engineers have to judiciously manage testing resources, finding a trade-off among quality, cost and release time.

The problem of testing resource allocation has been addressed in the software engineering literature mainly by formulating optimization models by means of Software Reliability Growth Models (SRGMs), able to describe the relation between test effort and reliability [27][35][46][59]. The way the optimization function and constraints are defined (e.g., cost minimization under reliability constraints, reliability maximization under cost/time constraints), and the SRGM modeling choices (e.g., multiple or single SRGM), have led to many models. While it is known that allocation choices impact jointly quality, cost and time, few proposals have addressed the problem in terms of multi-objective optimization [45][61][63]. Multi-objective models have no unique solution, and are valuable to generate the best set of alternatives, where trade-offs among contrasting objectives can be evaluated.

Existing single- and multi-objective models have several assumptions/limitations that undermine their practical applicability, often causing engineers to opt for easier – though non-quantitative – approaches. We identify major challenges in:

- **Impact of debugging.** Most allocation models maximize fault detection, without accounting for the fault correction process. The actual quality of a software product depends on the number of corrected faults: debug unaware optimization can be remarkably misleading [8]. Real debugging consists of various sub-activities, which have a severe impact especially for large systems [7][23][38][41]. Debugging times and bugs’ priorities and assignment are crucial in allocation decisions;

- **Data uncertainty.** Models rely on parameters (e.g., expected number of faults, fault detection rate, fault correction times) and on testing data (e.g., the operational profile) which are subject to non-negligible uncertainty. The quality-cost-schedule trade-off analysis may be strongly distorted by this uncertainty, as demonstrated by some work for single-objective optimization [28][30]. Even when testing parameters are estimated based on historical data, they only approximate real values;

- **Unrealistic assumptions.** SRGMs rely on several assumptions: independent inter-failure times, no code change during testing, perfect and immediate repair. Moreover, many techniques choose a SRGM a priori, regardless of its suitability to the available data. Testing in real contexts is likely to generate data that partially violate the assumptions and may escape an a priori selected model;

- **Time-effort non-linearity.** The relation between testing effort and time is generally non-linear [26][29]. This is considered almost only in single-objective models.

We believe these issues are at the root of the gap between research results and industry practice in testing resource allocation. We aim to fill it by the following contributions.

B. Contributions

We define a multi-objective, debug-aware, robust and adaptive formulation of the testing resource allocation problem.

Multi-objective denotes the ability to jointly consider: (i) quality – in terms of number of detected and corrected faults; (ii) cost – as expected cost of testing and faults correction; (iii) schedule – time to complete the testing activities.

Debug-awareness refers to the inclusion of debugging in the model. This encompasses the bug fixing time distributions (removing the assumption of immediate debugging), as well as the impact of bug assignment. We leverage the bug history to (i) estimate the ability of debuggers to correct faults, and (ii) assess the expected fixing time for the software components...
they work on\(^1\). Besides testing resources, the model determines the best way to assign debuggers to functionalities to maximize the number of corrected faults at minimum time and cost.

**Robustness** is the ability to produce solutions that are valid in spite of a given degree of uncertainty in the input parameters. We use Monte Carlo (MC) simulation to assess the robustness of a resource allocation solution under uncertainty. This approach allows eliciting and representing uncertainties as probability distributions, simulating the impact on the Pareto front of resource allocation solutions.

Finally, the proposed formulation is independent from any specific type of SRGM. **Adaptivity** refers to the selection of the SRGM best fitting each system component, based on the analysis of its bug history or of online data \([6]\). The method works with any Non-Homogeneous Poisson Process (NHPP) model. The main contributions are:

- Formulation of a multi-objective multiple-SRGM optimization model that addresses: (i) both the fault detection and correction processes; (ii) the testing time as function of the effort by means of testing effort functions (TEFs); (iii) the cost of testing and debugging. This allows exploring the trade-offs among typical testing objectives. Including TEFs in multi-objective formulations significantly improves resource allocation;
- The inclusion of the debuggers scheduling problem, which supports accurate testing resource allocation. This is especially relevant in large software projects;
- An approach to deal with the uncertainty of testing and debugging parameters that combines multi-objective evolutionary algorithms (MOEAs) and MC simulation.

The former are used in a wide spectrum of reliability-related optimization problems: resource management and task partition of grid systems, redundancy allocation, and reliability optimization \([63]\). MC methods are widely established for uncertainty analysis: examples are found in \([37],[51]\), where MC is used for handling parameter uncertainties in software architectures. Their combination allows reasoning in terms of “ranges” of potential solutions based on the ranges of input parameters;
- The empirical analysis with a real-world industrial case study. The proposed method is the result of an industry-academia collaboration in the ICEBERG EU-project\(^2\), investigating novel approaches to improve the understanding of software quality and its relation with cost.

### C. Organization

The paper is organized as follows: Section II presents background notions. In Section III we present an overview of the testing resource allocation framework. Section IV deals with uncertainty; Section V presents the optimization model at the core of the approach. Section VI describes the method used in the empirical study. The results are presented in Section VII. Section VIII discusses threats to validity. Section IX overviews related work, while conclusions are presented in Section X.

\(^1\)Here we refer to a component or module as an independently testable functionality. The terms are used as synonymous if not differently specified.

\(^2\)www.iceberg-sqa.eu.

## II. Background

### A. Modeling fault detection and correction through SRGMs

Software Reliability Growth Models (SRGMs) are well-known mathematical models of how reliability grows as software is improved by testing and debugging. They are built by fitting failure data collected during testing.

We consider the most common class of parametric SRGMs, modeling the process as a NHPP. They are characterized by the mean value function (mvf) \(m(t)\), which is the expectation of the cumulative number of defects \(N(t)\) detected by testing at time \(t\): \(m(t) = E[N(t)]\). The mvf is written as \(m(t) = \alpha F(t)\), where \(\alpha\) is the expected total number of faults, and \(F(t)\) is a distribution function whose form depends on the fault detection process \([19]\). The variety of SRGMs includes: the seminal model by Goel and Okumoto in 1979 \([18]\), describing fault detection by an exponential \(mvf\); the S-Shaped \([60]\) and log-logistic \([19]\) models, capturing increasing/decreasing behavior of the detection rate; the models derived from the statistical theory of extreme-value, based on the Gompertz SRGM \([42]\). Other models are available, accounting for needs emerged from real-world projects, e.g., non-negligible debugging times, imperfect debugging, multiple release points, nonlinear testing effort-time relation. A recent survey is in \([33]\).

Most SRGMs used in testing resource allocation models (e.g., \([25],[26],[27],[29],[35]\)) consider only the fault detection process, thus assuming the fault correction (i.e., debugging) being an immediate action. However, immediate debugging is very far from reality in today’s software systems: as software projects grow in size and complexity, the mean time to repair a fault is often very high, because of the complexity in managing the debugging workflow timely and correctly \([7],[23],[41]\). While many SRGMs refer exclusively to fault detection, we consider the combined modeling of fault detection and correction.

In the literature, debug-aware SRGMs represent fault correction as a process following detection with a time-dependent behaviour \([50],[57]\). Few proposals capture detection and correction together, wherein detection models are adjusted to consider several forms of the time-dependent fault correction \([31],[34]\). In this work, we leverage the framework proposed by Lo and Huang \([34]\), where the correction \(mvf\) is derived from the detection \(mvf\) considering the equations of Xie et al. \([57]\). The framework is briefly explained hereafter.

Let us first distinguish fault detection and correction by denoting with \(m_d(t)\) and \(m_e(t)\) the two mean value functions, respectively. The mean number of faults detected in the time interval \([t, t + \Delta t]\) is assumed to be proportional to the mean number of residual faults \([34],[50]\). Similarly, the mean number of faults corrected in \([t, t + \Delta t]\) is assumed to be proportional to the mean number of detected yet not corrected faults. This proportionality is expressed by the fault detection and fault correction rate per fault as functions of time, denoted with \(\lambda(t)\) and \(\mu(t)\). The following relations hold:

\[
\frac{dm_d(t)}{dt} = \lambda(t)(a - m_d(t)), \quad a > 0 \tag{1}
\]

\[
\frac{dm_e(t)}{dt} = \mu(t)(m_d(t) - m_e(t)) \tag{2}
\]
where $a$ is the initially estimated number of faults in the system, $(a - m_d(t))$ are the expected residual faults, and $(m_d(t) - m_r(t))$ are the yet uncorrected faults. In the case of constant detection rate per fault ($\lambda(t) = \beta$), $m_d(t)$ is the Goel-Okumoto exponential SRGM, $m_d(t) = a[1 - e^{-\beta t}]$. The function $m_c(t)$ is modeled in relation to the fault detection SRGM and the fault correction time. It can be shown that, using Equations (1) and (2), and defining $D(t) = \int_0^t \lambda(s)ds$ and $C(t) = \int_0^t \mu(s)ds$ (i.e., the cumulative detection and correction rate, respectively), the cumulative number of detected and corrected faults are [34]:

$$m_d(t) = a[1 - e^{-D(t)}]$$  (3)

$$m_c(t) = e^{-C(t)}\left(\int_0^t \mu(s)e^{C(s)m_d(s)}ds\right) = e^{-C(t)}\left(\int_0^t \mu(s)e^{C(s)}[1 - e^{-D(s)}]ds\right)$$  (4)

For instance, if the detection process follows an exponential SRGM with parameter $\beta$ ($\lambda(t) = \beta$), and the correction time is exponentially distributed with parameter $\gamma$ ($\mu(t) = \gamma$), then:

$$m_c(t) = a\left(1 + \frac{\gamma}{\beta - \gamma}e^{-\beta t} - \frac{\beta}{\beta - \gamma}e^{-\gamma t}\right)$$  (5)

In formulating the optimization model, we consider the so-computed fault correction for each functionality, in order to account for the real software quality increase occurring when a fault is actually corrected. The general expression in Equation (4) allows to avoid deciding a priori the SRGMs to adopt and/or the shape for the debugging time.

B. Modeling testing effort within SRGMs

The previous framework assumes the effort spent for testing is proportional to the testing time spent. This is, in general, not the case as the testing effort does not necessarily vary linearly with time. In the literature, this has been typically modeled by so-called Testing Effort Functions (TEFs), which describe how effort varies with time. When a TEF is considered, the previous fault detection model (Eq. 1) is adjusted as:

$$\frac{d m_d(t)}{dt} \times \frac{1}{y(t)} = \lambda(t)(a - m_d(t)), \quad a > 0$$  (6)

where $y(t)$ is the current testing-effort consumption at time $t$. The most common TEF, which was shown to well represent the usual trend of testing effort, is the logistic TEF [26], [25], [29] given by the following equation:

$$Y(t) = \frac{B}{\sqrt{1 + Ae^{\left[-\alpha/ht\right]}}}$$  (7)

where $B$ is the total amount of testing effort to be consumed; $\alpha$ is the consumption rate of testing-effort expenditures; $A$ is a constant; $h$ is a structuring index (a large value models well-structured software development processes); and $y(t) = \frac{dY(t)}{dt}$. When considering the TEF, $m_d(t)$ in Eq. (3) and, correspondingly, in Eq. (4) – is replaced by the solution of Eq. (6), which depends on the chosen SRGM and TEF. For instance, considering the exponential SRGM, i.e., $\lambda(t) = \beta$, and logistic TEF of Eq. (7), Equation (6) results in:

$$m_d(t) = a(1 - \exp[-(\beta Y(t) - Y(0))]), \quad a > 0$$  (8)

assuming $m_d(0) = 0$. The latter is replaced into Eq. 4 to obtain the effort-aware fault correction function, $m_c(t)$. We use this framework in the optimization model to describe the increase of quality as testing proceeds for each functionality.

III. OPTIMAL TESTING RESOURCE ALLOCATION

The objectives pursued by the allocation process are: (i) Fault Correction Objective (FCO), namely the expected number of corrected faults, to maximize; (ii) Testing Time Objective (TTO), namely the expected time to complete testing, to minimize; (iii) Testing/Debugging Cost Objective (TCO), namely the expected cost of testing and debugging, to minimize. The solution indicates the testing effort that must be devoted to each system functionality (or, equivalently, components), the binary assignment of debuggers to functionalities, and the hours each debuggers should spend on each functionalities. The phases of the process are now presented.

A. SRGM Construction

The first phase consists in inferring the functionality-level SRGMs in order to characterize the progression of testing activities. We do not assume any SRGM beforehand; the most suitable SRGM is inferred as follows:

- **Data Gathering.** Let $f$ denote one in a set of functionalities $F$. At the beginning of the optimization process ($t_0$), there are two possible cases: (i) historical testing data of $f$ are available (e.g., from another system that includes $f$, or also from testing of a previous version of $f$); (ii) no previous data exist. In the former case, the available fault correction times are used to fit SRGMs for functionality $f$, by using Equation 4. In the latter case, resources are initially allocated uniformly to functionalities: once testing starts, the incoming data are progressively used to fit SRGMs. The former case allows optimization before testing starts; however, it requires historical data. The latter case uses data gathered as testing proceeds, but the optimization can take place only when enough data are available. Dynamic allocation produces results more accurate and less sensitive to SRGMs assumptions violations, but it may be less useful if started late.

- **Parameter Estimation.** Data gathered for each functionality are fitted by means of all the SRGMs the tester wishes to try. Fitting of parameters for every SRGM is done via expectation-maximization (EM) [43].

- **SRGM selection.** For each functionality, the obtained SRGMs are compared to select the best fitting one. We adopt a typical goodness-of-fit measure, i.e., the Akaike Information Criterion (AIC), already used successfully for SRGM selection in [42]. The SRGM with the lowest AIC value is preferred out of the set, denoting that the fitting incurs into the minimal information loss.

As a result of this SRGM selection, each system functionality is assigned the best fitting SRGM.
uncertain parameters, A. Specification of Uncertain Parameters

Given a solution, the three objective functions (i.e., FCO, ACC, F mating) are accurately evaluated when not enough information is available. Parameters either inferred from available data or that cannot be addressed by the proposed uncertainty handling approach. The specification of the parameters is detailed in Section IV.

C. Robust Optimization

In the third phase the optimization model is processed by a model solver, which implements a multi-objective evolutionary algorithm (MOEA). The solver starts with an initial population of candidate solutions (Figure 1), known as individuals in the MOEA terminology. At each iteration, operators, such as crossover and mutation, are used to generate new individuals. The fitness of an individual is evaluated by handling uncertain parameters through MC simulation: the parameters are sampled multiple times, thus generating more fitness values for each individual (MC loop in Figure 1). We associate to each individual an interval of fitness values (rather than a point solution), and we compare individuals according to interval comparison criteria. The interval solution reflects the variability of the optimal solution depending on the variability of uncertain parameters (uncertainty handling for robust optimization is detailed in Section IV.) The most promising individuals are selected by the MOEA, producing a new set of solutions (new population in Figure 1), until the MOEA stopping criteria are satisfied.

IV. Solution Evaluation Under Uncertainty

Uncertainty is mainly dependent on the estimation of parameters either inferred from available data or that cannot be accurately evaluated when not enough information is available. Given a solution, the three objective functions (i.e., FCO, TTO, TCO) are evaluated by considering the uncertainty of parameters. In the following, we detail i) which are the uncertain parameters, ii) how individuals are compared to get robust solutions, and iii) the stopping criterion for MC runs.

A. Specification of Uncertain Parameters

The uncertain parameters are categorized as follows:

• Detection-specific parameters. They are related to the detection process of each functionality \( f \). These are the parameters of the detection rate per remaining fault function of the SRGM associated to \( f \). Such parameters are encompassed by the \( D(t) \) function in Equation 4, and their estimation depends on (past or current) failure data (so, they are affected by uncertainty).

• Correction-specific parameters. These are (i) the parameters of the correction rate per pending fault function, characterizing the debug-aware SRGM (\( C(t) \) parameters of Equation 4), and (ii) the average number of hours to fix a bug per functionality. Their estimation also depends on observed data (correction times).

• 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 operation.

The values of these parameters are treated as samples of either continuous or discrete probability distributions. Distributions can be inferred by means of different approaches [51], such as: (i) using the source of variations, in the cases when the source of uncertainty is known and can be estimated; (ii) empirically, when a considerable amount of data regarding the parameter behaviour are available; (iii) approximation as a uniform distribution if no information is available, and (iv) as a discrete distribution, when parameters are discrete-valued.

We use continuous uniform distributions (UD) for SRGM-related fault detection parameters. Since a SRGM is built by fitting historical data, the ranges of the uniform distributions are set with the 95% confidence interval bounds of the parameters estimate. A discrete distribution over the set of functionalities is used for the usage profile values. Call rates are estimated by historical data about the usage of each functionality, if available; otherwise they are specified by a domain expert, or they are assigned an even probability if no estimate is available. Finally, in line with the literature, we use the exponential distribution for the fault correction SRGM (with the average correction rate as parameter), as it has been shown to well represent the debugging process [50]. In this case, the average correction rate \( \mu(t) \) is constant, and is the reciprocal of the average number of hours to fix a bug (per functionality). The latter is estimated by querying data about bugs correction tracked in the company’s bug repository, as in [7][64][65], taking the median (or the mean, if the distribution is not skewed) of bug fixing times. If the information is not available, it should be assessed by a domain expert.

B. Robust Solution Evaluation

Samples are drawn from the above-mentioned distributions. They are used in the objective functions and constraints of the model to assess a candidate solution. The procedure is iterated \( N \) times – an iteration consisting of a MC run – until the desired accuracy is achieved. The output of a MC run leads to one possible fitness value of the candidate solution (i.e., the triple FCO, TTO and TCO).

Given the \( N \) fitness values of the candidate solution, the robust values for the objective functions can be derived by
using two methods [37]. The former method consists in deriving a Probability Density Function (PDF) for each objective function and taking the robust objective as the value at a given confidence. However, this approach is computationally expensive; furthermore, prospective probability distributions need to be specified a priori. The alternative method leverages non-parametric or distribution-free statistical procedures. For each candidate solution and each objective, the method assesses descriptive statistics (e.g. percentiles, mean, variance or confidence bound) from the observed sample (consisting of the $N$ MC runs). To capture the robustness of a candidate with different degree of tolerance, appropriate percentiles can be used as robust objectives. This method does not make any assumption on the probability distributions.

We adopt the non-parametric method. Several options are available regarding the descriptive statistics. A conservative solution is to select the lower/upper bound, namely the 5th or 95th percentiles, depending on whether the objective is to maximize or minimize, respectively. This approximates the bounds of 95% confidence interval. Others percentiles could be selected (e.g., the 50th). For instance, if the objective is to maximize (such as in the case of FCO), we consider the lower bound as a robust solution (namely the 5th percentile of observed values); whereas, for TTO and TCO the 95th percentile is taken as robust solution. In this way, the Pareto-front concept is enhanced to express the robustness of a solution with respect to uncertainty of parameters: as a result, the notion of dominance used by the MOEA is adjusted accordingly. Given the minimization of a vector function $f$ of a vector variable $x (x_k, \quad k = 1, \ldots, K)$, namely, minimization of $f(x) = (f_1(x), \ldots, f_N(x))$, subject to inequality and equality constraints $(g_j(x) \geq 0, \quad j = 1, \ldots, J \quad \text{and} \quad h_m(x) = 0, \quad m = 1, \ldots, M)$, let us denote with $\hat{f}(x) = (\hat{f}_1(x), \ldots, \hat{f}_N(x))$ the upper bound function vector, where $\hat{f}_n (n \in \{1, \ldots, N\})$ is the confidence upper bound of $f_n$ obtained from MC runs. Then, a solution vector $u = (u_1, \ldots, u_K)$ dominates a solution vector $v = (v_1, \ldots, v_K)$, denoted by $u \preceq v$ if $\hat{f}(u)$ is partially less than $\hat{f}(v)$, namely: $\forall n \in \{1, \ldots, N\}, \hat{f}_n(u) \leq \hat{f}_n(v) \land \exists n \in \{1, \ldots, N\}; \hat{f}_n(u) < \hat{f}_n(v)$. On this basis, the output Pareto front can also account for parameters uncertainty.

C. Stopping Criterion

We address the issue of selecting the number $N$ of MC runs (i.e., the sample size) that should be performed to have an accurate estimate for each candidate solution. We use a dynamic stopping criterion [37][48] to (i) monitor the accuracy of the value to estimate (e.g., number of faults corrected) and (ii) automatically stop the MC step when the number of runs is enough to meet a predefined error threshold.

For instance, let us consider the FCO objective and denote with $f_1$ the value of the objective after one MC run. Further runs of the MC simulation will likely provide different values of the objective, due to the parameters’ values taken at each run, i.e., $F = f_1, f_2, \ldots, f_N$. The goal is to establish the number of runs $N$ to obtain an estimate of the desired percentile of the set $F$, i.e., $\bar{f}_{perc}$. 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, $\bar{f}_{perc}$.
- As the number of runs increases beyond $k$, further estimates are obtained, considering the increasing number of observations, i.e.: $\bar{f}_{perc}$ from $f_1, \ldots, f_{k+1}$; $\bar{f}_{perc}$ from $f_1, \ldots, f_{k+2}$, and so on. The variation of the estimate is monitored over a sliding windows of size $k$. The last $k$ estimates are considered: $\{\bar{f}_{perc}; \bar{f}_{perc+1}; \ldots; \bar{f}_{perc+k}\}$.
- The statistical significance for the last $k$ estimates is:

$$e = \frac{z_{(1-\alpha)/2}}{\sqrt{k}} \sqrt{\frac{\bar{f}_{perc}^2 - \bar{f}_{perc}^2}{f_{perc}}}$$

where $e$ is the relative error, $\bar{f}_{perc}$ is the average of last $k$ estimates, $\bar{f}_{perc}$ is the mean-square of the last $k$ estimates, $z$ is the normal distribution evaluated at the desired significance level $\alpha$.

The relative error $e$ is checked against a predefined tolerance level (set to 0.01 in this study): MC runs are stopped when the error is below this level, as the desired accuracy is achieved. MC runs not satisfying some constraint of the model (e.g., because of values of uncertain parameters causing the constraint violation) are discarded and not counted as run.

V. OPTIMIZATION MODEL FORMULATION

The formulation aimed at minimizing the three objective functions (-FCO, TTO, TCO), under reliability and testing budget constraints, uses the notations listed in Table I.

A. Assumptions

The following assumptions are made, similarly as in related research [63][30][11][61][26].

- Functionalities are independently testable;
- Usual SRGMs assumptions, namely: fault detection and removal can be modeled as a NHPP; the mean number of faults detected in the $(t, t + \Delta t)$ is proportional to the mean number of remaining faults; interference times are independent. SRGMs have been demonstrated to provide accurate predictions even when these assumptions are partially violated [1];
- The relation between testing effort and testing time can be modeled by a TEF [29];
- Debugger manpower is available to independently fix bugs in system functionalities.

B. Model parameters

The main parameters given as input to the model are:

- The initial time $t_0$: it is the time the tester decides to run the algorithm. As described in Section III, $t_0$ can denote the beginning of testing (when historical data are used to build the SRGMs: $t_0 = 0$), or any time during testing (when online testing data are used; $t_0 > 0$). In the latter case, allocation can be run repeatedly during testing (i.e., dynamic allocation); we refer to $t_0$ as (re-)iteration time.
- $N_{d_{dc}}(t_0)$ is the number of faults detected and corrected in functionality $k$ at time $t_0$ ($N_{d_{dc}}(t_0) = 0$ if $t_0 = 0$).
- SRGMs for each functionality are characterized by $\lambda_k(t)$, and $\mu_k(t)$ (cf. with Equations 1 and 2), i.e., the detection
and correction rate per fault. They are related to $D(t)$ and $C(t)$, and consequently to $m_d(t)$ and $m_c(t)$ functions used in the objective functions.

- The $\delta_k$ parameter is the average number of hours required to fix a bug for functionality $k$. Recall that, being the correction time assumed exponential, the rate $\mu_k(t) = 1/\delta_k$ is estimated as $\mu_k = 1/\delta_k$.

- $\omega_k$ is the call rate, namely the probability that the $k$-th functionality will be invoked: $\omega_k \geq 0, \forall k = 1 \ldots K$, and: $\sum_{k=1}^{K} \omega_k = 1$. Note that parameters of $\lambda_k(t), \mu_k(t)$ and $\omega_k, \delta_k$ are the uncertain parameters (Section IV-A).

- $\alpha, h, B, A$ are the parameters of the logistic testing effort function (Equation 7), which is used to explain how testing effort varies in function of calendar time. They were discussed in Section II.

- $\gamma^*_d$ is the processing capacity of debugger $d$ with respect to functionality $k$. It represents the working rate of the debugger, expressed as average number of hours per day that debugger $d$ is allowed to work on functionality $k$.

- $C^*_1, C^*_2, C^*_3$ are the cost parameters used in the cost-related objective function (TCO). They are: (i) $C^*_1$, the cost per man-day to correct a bug during testing; (ii) $C^*_2$, the cost per man-day to correct a bug during operational use (typically $C^*_2 > C^*_1$ [5]); (iii) $C^*_3$, 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).

Note that SRGM parameters (parameters of $\lambda_k(t), \mu_k(t), \alpha, h$ and $A$) are estimated by fitting (historical or online) testing data as discussed in Section III; $\omega_k$ and $\delta_k$ are estimated by historical data, design information or expert judgment (cf. Section IV-A); debuggers capacity $\gamma^*_d$, cost parameters $C^*_i$ and the budget $B$ are provided as input by the tester.

C. Variables

The decision variables of the model are:

- $Y_{k} (1 \leq k \leq K)$ variables are used to suggest the amount of testing effort (in man-hours) to perform on functionality $k$; $Y_k$ depends on the calendar testing time $t_k$ devoted to test functionality $k$. The relationship between testing effort and time is modeled by the logistic TEF (Equation 7): the amount of spent testing time corresponds to $t_k = F^{-1}(Y_k)$ hours, where $F^{-1}$ is the inverse of the TEF. $x_{d}^{k} (1 \leq d \leq D, 1 \leq k \leq K)$ and $N_{d}^{k} (1 \leq d \leq D, 1 \leq k \leq K)$ variables are used to regulate the assignment of debugger $d$ to functionality $k$. Thus, the fault correction process is modeled as function of available debuggers. Specifically, $x_{d}^{k} = 1$ if debugger $d$ is scheduled on functionality $k$, and 0 otherwise. $N_{d}^{k}$ is the time (in hours) assigned to debugger $d$ to work on functionality $k$ in $(t_0,t_1]$. A solution consists of: a vector $Y$ of $Y_k$ values, with the optimal testing effort per functionality; a matrix $X$ of $x_{d}^{k}$ values, assigning debuggers to functionalities; a matrix $N$ of $N_{d}^{k}$ values, with number of debuggers’ hours per functionality.

D. Constraints

The most relevant constraints are the following ones:

1. $\sum_{d=1}^{D} N_{d}^{k} \geq \delta_k (m_d (t_0 + t_k) - m_d (t_0)) \quad \forall k = 1 \ldots K$
2. $N_{d}^{k} \leq \frac{t_k}{\gamma_d} \cdot x_{d}^{k} \quad \forall k = 1 \ldots K, \forall d = 1 \ldots D$
3. $\left\{ \begin{array}{ll} x_{d}^{k} = 1 & \text{iff Deb. d. to be assigned to k;} \forall k = 1 \ldots K, \forall d = 1 \ldots D \\
 x_{d}^{k} = 0 & \text{iff Deb. d. not to be assigned to k;} \forall k = 1 \ldots K, \forall d = 1 \ldots D 
\end{array} \right.$
4. $m_d (t_0 + t_k) - m_d (t_0) \leq \alpha_k - F_{dk} (t_0) \quad \forall k = 1 \ldots K$
5. $\sum_{k=1}^{K} Y_k \leq B \quad \forall k = 1 \ldots K$
6. $Y_k \leq B (1 - \prod_{d=1}^{D} (1 - \frac{x_{d}^{k}}{\gamma_d})) \quad \forall k = 1 \ldots K$
7. $\sum_{k=1}^{K} \omega_k \cdot \phi_k (t) \leq \phi^*$

They are to be interpreted as follows:

- For each functionality $k$, faults detected in the interval $(t_0,t_1]$ must be fixed. Equation 1 represents this constraint. The total time (in hours) 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 functionality $k$ is tested for a time $t_k$).

- The bug correction process is modeled as a function of (i) the amount of time (in hours) required to fix the bugs detected and (ii) the working time of debuggers. The waiting queues are modeled by a constraint on the capacity of debuggers. Equation 2 represents this constraint.

---

**TABLE I: Notations adopted in the formulation of the multi-objective optimization**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>Number of system functionalities</td>
<td>$k$</td>
<td>System functionality index</td>
</tr>
<tr>
<td>$a_k$</td>
<td>Expected number of initial faults in the functionality $k$</td>
<td>$\omega_k$</td>
<td>Probability the functionality $k$ will be invoked</td>
</tr>
<tr>
<td>$\delta_k$</td>
<td>Average number of hours required for fixing a bug of the functionality $k$</td>
<td>$t_0$</td>
<td>Time at which the resource allocation model is run</td>
</tr>
<tr>
<td>$R$</td>
<td>Minimum threshold given to the reliability on demand of the system</td>
<td>$Y_0$</td>
<td>Testing effort (measured in man-hours) already spent at time $t_0$</td>
</tr>
<tr>
<td>$F_{dk}(t_0)$</td>
<td>Number of faults of the functionality $k$ detected and corrected at $t_0$</td>
<td>$D$</td>
<td>Total number of debuggers</td>
</tr>
<tr>
<td>$x_{d}^{k}$</td>
<td>Debugger used/not used to fix bugs of the functionality $k$ (i.e., 1/0)</td>
<td>$d$</td>
<td>Debugger index</td>
</tr>
<tr>
<td>$N_{d}^{k}$</td>
<td>Time assigned to the debugger $d$ for testing the $k$-th functionality (hours)</td>
<td>$t_k$</td>
<td>Calendar testing time devoted to test the functionality $k$ (hours)</td>
</tr>
<tr>
<td>$Y_k(t)$</td>
<td>Cumulative testing effort devoted to functionality $k$ in $[0, t]$ (man-hours)</td>
<td>$\lambda_k(t)$</td>
<td>Fault detection rate per undetected fault for the functionality $k$</td>
</tr>
<tr>
<td>$B$</td>
<td>Total amount of testing-effort available for consumption (man-hours)</td>
<td>$A$</td>
<td>Constant parameter in the logistic TEF</td>
</tr>
<tr>
<td>$C^*_i$</td>
<td>Average cost per man-day to correct a bug during testing</td>
<td>$C^*_2$</td>
<td>Average cost per man-day to correct a bug in operational use</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Consumption rate of testing-effort expenditures in the logistic TEF</td>
<td>$\phi_k(t)$</td>
<td>Expected failure intensity function for $k$ at testing time $t$</td>
</tr>
<tr>
<td>$\phi^*$</td>
<td>Maximum threshold given to failure intensity of the system after test</td>
<td>$\mu_k(t)$</td>
<td>Fault correction rate per detected but uncorrected fault for $k$</td>
</tr>
</tbody>
</table>

For simplicity, we assume that this time, for a given functionality $k$ (i.e., $\delta_k$), is the same for each debugger $d$ working on that functionality.
For each functionality \( k \), the load of debugger \( d \) caused by the assignment of bugs is limited by a function of the processing capacity of debugger \( d \) (i.e., \( \gamma_d^k \)). \( N_d^k \) 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 to 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 \( \gamma_d^k > 0 \) and \( \delta_k > 0 \).

- Equation 3 represents (possible) constraints, which can be defined for debuggers that must 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, to solve incompatibilities or dependencies among debuggers and/or functionalities, due, for instance, to human factors or functionality characteristics, additional constraints can be added. For example, \( x_2^1 \leq x_3^2 \) means that if debugger 2 is scheduled on functionality 1, then debugger 3 must be scheduled on functionality 2.
- Equation 4 states that the expected number of faults detected in \( (t_0, t_k] \) (where \( t_k = F^{-1}(Y_k) \) is the time devoted to test functionality \( k \)) cannot be greater than the expected number of residual fault of functionality \( k \).
- Equation 5 is a constraint on the maximum effort that can be allocated. The sum of efforts cannot exceed the maximum available budget \( B \) (expressed in man-hours).
- If there are no available debuggers for functionality \( k \), then the effort \( Y_k \) allocated to it must be zero (as detected bugs would not be corrected). In other words, if functionality \( k \) receives a certain amount of testing effort, one or more debuggers must be assigned to functionality \( k \). Equation 6 represents this constraint.
- Equation 7 is the constraint on a maximum desired failure intensity at the end of testing \( T \). Failure intensity \( \phi_k(t_k) \) of a functionality \( k \) is estimated through its SRGM as the derivative of \( m_d(t) \). A maximum failure intensity threshold \( \phi^* \) is given as input. In an average case, like the one we assume, this constraint is expressed by Equation 7. In the worst case, all functionalities could be required to satisfy the failure intensity constraint, and the constraint would be: \( \max_{k=1 \ldots K} (\phi_k(t_k)) \leq \phi^* \). Notice that this constraint can be referred to as reliability constraint, since failure intensity at the end of testing is related to reliability: \( R(t|T) = \exp[-\phi(T) \cdot t] \), where \( T \) is the release time [46].

Further system-specific constraints could be introduced based on specific needs, but at the expense of higher complexity and less understandability.

### E. Multi-Objective Function

#### 1) Fault correction process’ Effectiveness Objective (FCO)

The objective is to maximize the predicted number of corrected faults at the end of testing. The prediction is carried out by the debug-aware SRGMs per functionality.

\[
FCO = \sum_{k=1}^{K} m_{c_k}(t_0 + t_k)
\]  

where (taking Equation 4 and \( Y_k = TEF(t_k) \)):

\[
m_{c_k}(t_0 + t_k) = e^{-c_k(t_0 + t_k)} \cdot \left( \int_{t_0}^{t_0 + t_k} a_k c(s)e^{C(s)}(1 - \exp[-(D_k(Y_k(s)))ds]) \right).
\]  

The expression can be instantiated for any detection and correction rates \( (D(t) \text{ and } C(t)) \) and TEF relating \( t \) to \( Y \). For instance, in the case of exponential detection and correction process \( (\lambda(t) = \beta, \mu(t) = \mu) \), it becomes:

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

The expected number of faults detected and corrected depends on: (i) the fault detection rate, related to the testing effort \( Y_k \) and time \( t_k \) (through the TEF); (ii) the availability of sufficient debuggers (hours), regulated by \( N_d^k \) and \( x_d^k \) variables, for the correction of detected faults at the rate expressed by \( C_k(t) \).

#### 2) Testing Time Objective (TTO)

The relationship between testing effort and time is typically modeled by the TEF. For a generic TEF \( F \), we could write: \( t_k = F^{-1}(Y_k) \). Assuming the TEF being modeled by the generalized logistic testing-effort [29] (Equation 7), testing time for functionality \( k \) is:

\[
t_k = \left( -\frac{1}{\alpha + \phi} \cdot \ln \left( \frac{\phi^*}{\phi} \left( \frac{1}{A} \right) \right) \right)
\]  

where parameters are as described in Section II.

Since functionalities are assumed to be tested independently, TTO is the time minimization for testing the \( K \) functionalities:

\[
TTO = \min_{k=1 \ldots K} t_k
\]  

#### 3) Testing-effort Cost Objective (TCO)

The third objective concerns with the minimization of cost, which is a measure related to the effort spent but that goes beyond it. In agreement with [25], [58], the cost of testing-effort expenditures during software development and testing, and the cost of correcting errors before and after release, can be expressed as:

\[
Cost_k(t) = C_1^* \cdot (\delta_k/24) \cdot m_{c_k}(t) + C_2^* \cdot (\delta_k/24) \cdot (m_{d_k}(\infty) - m_{c_k}(t)) + C_3^* \cdot (Y_k/24)
\]  

where: (i) \( C_1^* \cdot (\delta_k/24) \) is the cost to correct a bug during testing; (ii) \( C_2^* \cdot (\delta_k/24) \) is the cost to correct a bug in operational use (typically \( C_2^* > C_1^* [5] \)); and (iii) \( C_3^* \) is the cost of testing per unit testing-effort expenditure, expressed in cost of a man-day (for a tester). The TCO is the minimization of total cost over all the functionalities:

\[
TCO = \sum_{k=1}^{K} Cost_k(Y_k)
\]  

The three objectives to minimize are thus: \( \min(FCO, TTO, TCO) \), under the specified constraints.
VI. EXPERIMENTATION

We design an empirical study to experiment the method. The addressed research questions are presented, followed by the industrial case study description and the experimental setup.

A. Research questions

- RQ1. Multi-objective optimization. A multi-objective optimization problem is formulated and solved by various metaheuristics. We formulate these questions to assess the goodness of proposed solutions:
  - RQ1.1. Validation. How does the proposed approach perform compared to random search? This is a typical question performed as a preliminary “sanity check”, since any intelligent search technique is expected to outperform random search unless there is something wrong in the formulation [16].
  - RQ1.2 Comparison of metaheuristics. Which of the considered multi-objective evolutionary algorithms (MOEAs) yields the best solution? This question focuses on the comparison among common MOEAs according to performance metrics regarding the goodness of the provided Pareto solutions.

- RQ2. Uncertainty analysis. The proposed approach deals with parameter uncertainty: what is the effect of explicitly considering the uncertainty of parameters? This question aims at evaluating to what extent embedding the MC simulations into the search technique provides robust solutions and at what computational expense.

- RQ3. Sensitivity to debugging. The proposed approach considers the debugging process: how does the optimal solution vary when considering debugging in the model? Differently from existing allocation models, our model is based on the fault correction process. This question aims at evaluating solutions computed by considering the bug assignment activity against solutions not explicitly incorporating bug assignment, under various configurations.

- RQ4. Scalability: How does the performance of our approach change while varying problem size? With the selected case study, we show to what extent the process is fast enough to analyze a real testing effort allocation decision process whose size and complexity is similar to those of other published large-scale/industrial testing effort allocation problems (e.g., [6] and [63]).

B. Case study

We consider a real-world issue tracking system of a Customer Relationship Management (CRM) software of a multinational company operating in the healthcare sector. Data have been made available within the ICEBERG European Project by ASSIOMA.NET, an IT company involved in the development of the CRM. The system has a layered architecture, with a Front-end layer, a Backend layer and a Database, made interoperable through an Enterprise Application Integration (EAI) layer. It provides typical CRM functionalities: sales management, customer folder, agenda, inventory, supplying, payments, user profiling, and various reporting tools. Collected issue records span a period of 2 years and a half, from September 2012 to January 2014. A total of 612 software faults collected during testing are considered. Once faults are detected, they undergo a debugging process: when an issue is opened, it becomes new and it is enqued, waiting to be processed (published state); once an issue starts to be processed (in study), it is assigned (launched) to a developer and, once completed, it can be assigned to another developer for further processing, if needed. Then, the amendment is tested, delivered, and finally closed. Detection and correction times are tracked in the issue tracker The Time To Repair (TTR) a fault is the time to transit from the new state to the closed state. The TTR distribution is highly skewed in our data: medians of TTR values in lieu of means are considered to represent the average TTR per functionality.

C. Experiment settings

1) Setting for MOEAs Evaluation: We experiment four metaheuristics to obtain an allocation solution, namely: NSGA-II [12], IBEA [66], MOCELL [39], PAES [32]. To measure performance, we use two well-known indicators: the inverted generational distance (IGD) [53], and the spread [12], which reflect the two goals of a MOEA: 1) convergence to a Pareto-optimal set and 2) maintenance of diversity in solutions of the Pareto-optimal set. The IGD is computed as the average Euclidean distance between the set of solutions S, and the reference front RF [53] (the smaller its value the better the solution set). The latter is computed by considering the union of reference fronts of the approaches compared. The spread measures the extent of spread achieved among solutions: it is computed as a ratio accounting for the consecutive distances among solutions (and their error from the average) at numerator, related to the case where all solutions would lie on one point at denominator [12].

As evolutionary algorithms have a stochastic nature, we perform 30 independent runs for each algorithm. For each of them, IGD and spread are computed and compared by means of non-parametric test, since data are non-normal and heteroschedastic. We use the Friedman test for non-parametric ANOVA, since it does not make assumptions on normality of observations, homoscedasticity of variances, independence of data among compared samples, and it works well under balanced designs as ours. Then, to detect which algorithms are different, we use the Nemenyi test as post hoc, which is a test for pairwise comparisons after a non-parametric ANOVA particularly suitable when all groups are compared to each other (rather than comparing groups against a control group) [13], as in our case.

2The Shapiro-Wilk test rejected the normality hypothesis at p < .0001 in both cases, and the Levene’s test, used because less sensitive to normality, rejected Homoscedasticity hypothesis at p = 0.0074 and p < .0001 for spread and IGD, respectively.

3The test uses the “critical difference” (CD): two levels are significantly different if their average ranks differ by at least CD = qₐ√(k+1)/6N, where qₐ is based on the Studentized range statistic divided by √₂, and adjusted according to the number of comparisons; k is the number of levels; N is the sample size. As the family-wise error rate is controlled by considering qₐ, no other multiple comparison protection procedure is needed.
We measure the “effect size” to assess whether the difference among MOAEs is worthy of interest. We adopt the Vargha and Delaney test [52], as suggested in [3], using the \( \hat{A}_{12}(x,y) \) statistic. The latter can be interpreted as probability that the performance metric’s value of technique \( x \) will be greater than \( y \) – namely, the probability that a randomly selected observation from one sample is bigger than a randomly selected observation from the other sample [20]. Before estimating the effect size, there could be the need to transform the data, if the data being compared do not faithfully represent the “right” meaning of the comparison (e.g., because of unstated assumptions, as: “algorithms response times lower than 100 ms are imperceptible and should be considered equally good” [40]). However, this is not the case of IGD and spread, for which we do not detect this kind of assumptions.

Algorithms implementation and experimental settings are performed with jMetal, an object-oriented Java framework to develop and experiment MOEA\(^8\). Parameters of each algorithm are set as reported in Table II, ensuring the same maximum number of fitness evaluations for all the algorithms (25,000) and default values as provided by jMetal.

2) Specification of certain/uncertain parameters: As for parameters taken without uncertainty, the following values are set as input for the experimentation, after consulting the company domain expert. They refer to the allowed processing capacity of each developer (\( \gamma^k \)); the cost parameters (\( C^1_k, C^2_k, C^3_k \)); the parameters of the logistic TEF (\( \alpha, \beta, \) the structuring index; \( A, B \)). Values are: \( \gamma^k = 1/24 \) hours per day; \( C^1_k = 60 \) €; \( C^2_k = 80 \) €; \( C^3_k = 60 \) €; \( \alpha = 0.5 \) man-hours per hour; \( h=0.05; A = 0.8; B = 2500 \) man-hours. For uncertain parameters, data from the bug repository about fault detection and correction times in the previous version are used as historical basis to assess SRGMs and debugging parameters for each functionality (see Section IV-A). The detection rate \( \beta_k \) is sampled via a uniform distribution with ranges set to the 95% confidence interval of its SRGM estimate\(^9\); the correction rate is sampled via an exponential distribution whose parameter, \( \mu_k \), is obtained as \( \mu_k = 1/\delta_k \), with \( \delta_k \) being the TTR median of each functionality available from the bug tracker. Finally, the usage profile values (the last uncertain parameter, \( \omega_k \)) are specified by domain experts as values of a discrete distribution over the set of functionalities. All these values are in Table III.

A. Results for RQ1 (Validation and MOEA Comparison)

The first part of RQ1 establishes if the proposed strategy is worth with respect to a random search. The random search model just picks up solutions that satisfy the constraints, by a pseudorandom number generator, as implemented by the jMetal framework. With all 4 MOEAs the random search has been statistically worse than any MOEA algorithm for both quality indicators\(^10\), with a confidence greater than 99%. We do no longer consider it in the next research questions.

RQ1.2 is about MOEAs comparison. Figure 2 shows the notched box plots for both indicators (IGD and Spread). The Friedman test yielded a \( p-value = 3.42 \) E-6 for IGD and \( p-value = 1.08 \) E-6 for spread, rejecting the null hypothesis that all the MOEAs are statistically equivalent, for both indicators. Table IV reports the results of the Nemenyi test.

\(^8\)Metal is available at http://jmetal.sourceforge.net/.

\(^9\)For the purpose of experimentation, a single-parameter exponential SRGM is used, i.e., \( \lambda_k(t) = \beta_k \).

\(^10\)Since, in this case, we compare all the algorithms against one control algorithm (the Random Search), we adopt the Bonferroni-Dunn test as post-hoc, which is more powerful than the Nemenyi test in such a case [13].
is statistically equivalent to PAES and MOCELL (although, in terms of effect size, it is better than PAES and worse than MOCELL). Since all MOEAs are around 1 (meaning not a good diversity), and given the remarkable difference of IGD between IBEA and NSGA-II, we keep NSGA-II as reference scheme for the other research questions.

### B. Results for RQ2 (Uncertainty analysis)

To assess the effectiveness of considering uncertainty, we evaluate: i) the impact of the number of Monte Carlo runs on accuracy and computational time; ii) the accuracy of the dynamic stopping criterion. To this end, we perform a sensitivity analysis. For each experiment, the reliability threshold is set to 0.9, while the other parameters are as in Section VI-C.

**Impact of the Monte Carlo runs.** We run a set of experiments without using the dynamic stopping criterion to determine how many runs to perform, keeping the number of MC runs in each experiment fixed. To measure the accuracy, we use the Root Mean Square Error (RMSE), considering 10,000 as reasonable large number of MC runs to get an accurate value against which to compare. We use the 95th percentile for comparison, to be conservative under the minimization objectives (-FCO, TTO, TCO), as explained in Section IV-B.

Given an objective function, we first run the experiment under 10,000 MC runs per each solution evaluation. On each solution, the 95th percentile of the fitness value distribution is computed, and kept as fitness value of that solution to compare. We use the Root Mean Square Error (RMSE), computed for 10,000 MC runs, as reasonable large number of MC runs to get an accurate value against which to compare. We use the 95th percentile for comparison, to be conservative under the minimization objectives (-FCO, TTO, TCO), as explained in Section IV-B.

Then, the same procedure is applied for each experiment where a different number of MC runs is considered (e.g., 10 runs for the first experiment), obtaining the 30 solutions (Sol_i) to compare against Sol_fo. The RMSE, computed for each objective function, is given by:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{30} (\text{Sol}_f - \text{Sol}_i)^2}{30}}$$  \hspace{1cm} (16)

Figure 3 shows the results. Computational time is the mean across the 30 repetitions, which is the same in the three cases, since the three fitness values are output of the same experiment. The RMSE decreases with the number of MC runs while the mean computational time increases up to 585s. Good tradeoffs between RMSE and time are approximately at 200 MC runs (e.g., in the case of TTO and TCO, beyond 200 MC runs the accuracy improves a little, while the computational time still keeps on increasing linearly).

**Accuracy of the dynamic stopping criteria.** To check the accuracy of the stop criterion, we let the number of MC runs increase in each experiment (with default settings of Section VI-C). Specifically, in each of the 30 repetition, we compute the error as in Section IV-B by Equation 9, with a threshold of 0.01 under 95% of confidence, and on a sliding sample window of 10. We stop the experiment when the maximum number of MC runs needed to satisfy the error threshold in all the three objective functions is reached. The average of such values over 30 repetitions turned out to be 207. This number provides also a good trade-off between solution accuracy and computational time, as in the previous paragraph (at 207 MC runs, the RMSE of FCO is about 7 faults, 6.5 hours and almost 460 EUR for a computational overhead of about 220 seconds). In general, the number of MC runs should be chosen by looking at trade-offs between the error threshold, the computational time and solution accuracy it entails.

### C. Results for RQ3 (Sensitivity to debugging)

We analyze the impact of considering debugging on the solution. We compare the behavior of our debug-aware testing allocation model against a debug-unaware model, where the assumption of immediate repair is done. Specifically, the debug-unaware model considers m_c(t) = m_d(t) (i.e., correction times are the same as detection times), and ignores parameters related to debuggers assignment to functionality (e.g., δ, γ, N_d(t), x_d), but it still considers uncertainty.

The comparison is to figure out the difference in estimating the corrected faults, testing time and testing/debugging cost if the debugging process is neglected with respect to considering debugging in the formulation. The comparison is done again on sets of 30 solutions. Solutions are selected from the Pareto front by four criteria: assuming that tester is interested only in i) maximizing the number of corrected faults, regardless the testing time and testing/debugging cost (i.e., take the solution with the maximum FCO); ii) minimizing the testing time, regardless the other objectives (i.e., the solution with

---

Image sources: (a) FCO (#Faults) accuracy and time, (b) TTO (hours) accuracy and time, (c) TCO ($\varepsilon$) accuracy and time

---

Fig. 3: Impact of the MC runs on the accuracy and mean computational time
Fig. 4: Solutions according to the four different criteria, for each objective. Reliability constraint set at 0.9

minimum TTO); iii) minimizing the cost, regardless FCO and TCO; iv) assuming that tester wishes to optimize a balanced tradeoff between the three objectives. As for the latter, let us denote the set of the 3 fitness values of a given allocation solution \( X \) as \( Y(X) = \{y_1, y_2, y_3\} \), denoting, respectively, the FCO, TTO and TCO values of that solution. We normalize these values in \([0,1]\) over the entire Pareto front: 

\[
y^i_{\text{opt}} = \frac{y^i_{\text{opt}} - \min_{X}(y^i_{\text{opt}})}{\max_{X}(y^i_{\text{opt}}) - \min_{X}(y^i_{\text{opt}})}, \quad \text{with } i=1, 2, 3.
\]

The chosen solution \( X^* \) is the one with the minimum loss function value: 

\[
L(Y'(X)) = \sum_{i=1}^{3} w_i \cdot y^i_{\text{opt}},
\]

with \( w_i \) being the weights assigned to each objective. For a balanced tradeoff, we select these weights: -0.33 for FCO; 0.33 for TTO and TCO. Results are in Figure 4. The boxplots report the 30 solutions selected for each combination. The models provide comparable expected numbers of corrected faults, with little differences: however, the debug-unaware model gives estimates of testing time and cost considerably lower than the debug-aware model (especially cost estimates). The immediate debugging assumption makes the debug-unaware model heavily underestimate the time and cost. In reality, the correction of those faults has a non-zero time and cost; the debug-aware model accounts for this, and yields much higher estimates. The experiment is repeated by setting a minimum reliability constraint at 0.94 and 0.97 (set at 0.9 in Figure 4), observing that difference of faults and testing time seem slightly lower under more stringent constraints at 0.94 and 0.97, while the cost difference is still very high. With respect to the four criteria to select solutions, there are slight improvements in one objective at the expense of the others, when considering the corresponding single-objective criteria, but the relative difference among debug-aware and debug-unaware case is roughly invariant. Table V summarizes all results, reporting the absolute and percentage relative difference between the means in the two cases. In terms of expected number of corrected faults (first part of the Table), the worst difference is 4.07% (23 faults) in the solution minimizing cost. The worst difference in terms of testing time (second part of the Table) is observed again under cost-minimizing solution, and is 5.54% (4.25 hours) under a reliability constraint of 0.90. Differences of cost estimates (third part) are very high (always around 50%); the worst case is 59.95% (3,968 EUR) under time-minimizing solution with reliability constraint at 0.90. In the latter case, the difference between the estimates...
of corrected faults is negligible (0.09%), and 3.89% (2.86 h) in terms of testing time: by adopting the solution suggested by the debug-unaware model, the corrected faults is not expected to vary, but the actual calendar testing time is expected to be 2.86 hours longer and the testing/debugging cost 3.968 EUR more than the estimated ones. Accounting for debugging will thus help taking better-informed decisions.

**D. Results for RQ4 (Scalability)**

We run a set of experiments varying the problem size. Starting from parameters of Section VI-C, we generate three problem instances with number of functionalities 14, 19, and 25. Parameters of the added functionalities (e.g., SRGM parameters) are generated by random perturbations of 10% of existing functionalities’ parameters. For each configuration, we run the model for a number of debuggers spanning from 8 to 24, to further increase the size of the problem.

Figure 5 illustrates the performance depending on the functionality and debuggers set size. For a given number of debuggers, the model execution time increases with the increase of the number of functionalities, at approximately a linear pace in all the cases. The slope increases with the number of debuggers too. Thus, the algorithm is still able of managing the search space under these configurations, which are compatible with our industrial case study.

**VIII. THREATS TO VALIDITY**

**Construct validity.** The approach uses as input the fault detection and correction times (typically stored in bug reports) for SRGM fitting and for time-to-bug-fix estimation.

This implicitly assumes that historical information about bug reports is correct: namely, reporters can distinguish a bug from a feature request, correctly identify duplicate bug reports, and the average correction time can be faithfully approximated as bug closing (i.e., fault correction time) minus bug opening (i.e., fault detection time). These assumptions strongly depend on the quality of bug reports: many works are available discussing on this threat and how it can be mitigated (e.g., [21][22]). The cost parameters are provided by tester. They are assumed to be known within the company: this information is not always easily accessible, and more or less complex models could be needed to estimate it (e.g., COCOMO II [9]). This can be done without essentially changing the model structure, but with the side effect of increasing its complexity.

**Internal Validity.** As mentioned, the approach is subject to common SRGMs (with TEF) assumptions, which can bias the allocation accuracy. The impact of potentially different testing techniques on detected faults (hence on SRGMs) can also affect results [10]. Differently from previous work (e.g., [25][26][27][29][35]), our model mitigates them by enabling selection from multiple SRGMs with online data, accounting for the effect of such assumptions’ violations [6]. Additionally, accounting for uncertainty in SRGM parameters is exactly a way to counteract such a threat. As further threat, changing the MOEA strategies and their parameters can change the result. To limit it, we selected four MOEAs among commonly used ones in the literature with their default settings.

**External Validity.** Although the steps of the process are easily applicable to other contexts, we cannot claim that results are statistically generalizable. Nonetheless, since analyses on proprietary industrial systems are rare in the related literature, we believe that reported results provide an important contribution toward the practicability of such kind of models in industry. Indeed, the method formulation leveraged from useful feedbacks by our industrial partner, which allowed to tailor the testing allocation process on real industrial needs.

**Conclusion Validity.** We achieved statistically reliable results by randomly repeating runs 30 times and by using non-parametric statistical testing. Moreover, conclusions were enforced by running experiments under various configurations of the main parameters of interest.

**IX. RELATED WORK**

**Testing resource allocation.** Several papers with SRGM-based testing resource allocation [25][26][27][29][30][35][63]. Yamada et al. [59] formulated two variants of the problem of optimal effort allocation in module testing, assuming the same SRGM for all the involved modules. Lyu et al. [35][27] target the same problem, proposing an optimization model with cost function based on well-known SRGMs, including the use of a coverage factor for each component to account for fault tolerance. Cost, along with testing effort function, is considered also in later work by the same authors [25]. The authors in [24] allocate optimal testing times to components using the Hyper-Geometric (S-shaped) SRGM. The work in [30] uses a flexible SRGM with a testing effort model able to describe either exponential or S-shaped failure patterns. In our previous work [46], and later in [61], SRGM-based allocation
is merged with a software architectural model, expressed through a discrete-time Markov chain to explicitly account for components’ usage. In several of these papers, SRGMs include the TEF [25][26][29]. Most approaches are based on single-objective optimization. The few papers focusing on multi-objective optimization, e.g., maximizing reliability while minimizing testing cost and time [11][61][63], do not consider various aspects accounted for in our model: the debuggers assignment task, the uncertainty of parameters, the dynamic selection of multiple SRGMs.

**Test case minimization, selection, prioritization.** Search based techniques are used to support testing minimization [17], selection [47] and prioritization [54] (the paper [62] surveys each of the mentioned areas). It is worth noting that the approach proposed by our paper can be applied independently of the order tests are executed: in this respect, our resource allocation approach under uncertainty can support regression testing practices.

**Bug assignment.** Bug assignment has gained attention in recent years. In large software projects, it requires considerable contextual information about bugs and developers, and it is a time-consuming and tiresome process [23]. Research in the field of mining software repositories proposed: (i) developers’ expertise models based on previous bug reports [2] or source code contributions [36]; (ii) machine learning techniques to learn the kinds of reports each developer resolves [2]; (iii) preference elicitation methods to determine developer’s preferences for fixing certain types of bugs [4]; (iv) an auction-based multi-agent mechanism allowing developers to require bugs to triage, to then make decisions based on their preferences and expertise [23]. The application of search techniques to implement an efficient bug repair policy is largely unexplored. In [56], a genetic algorithm is designed for scheduling developers and testers to bug-finding tasks considering both human properties (skill set, skill level and availability) and bug characteristics (severity and priority).

**Uncertainty of parameters.** Although software testing is fraught with a not negligible uncertainty, this topic, in practice, is not commonly addressed. Research efforts have been spent to evaluate the quality attributes (e.g., reliability [55] and performance [51]) of software architectures under uncertain parameters, adopting, for example, a robust optimization approach [37], or a bayesian approach [14]. The robustness of an architectural model despite uncertainty via Monte Carlo method is used, e.g., in [51] and [37]. Also, fuzzy methods are adopted to represent uncertain parameters (e.g., [15]) of an alternative architecture. The fuzzy paradigm is also used in [44], wherein uncertainty in estimated parameters of SRGM is addressed in imperfect debugging environment.

The proposed process provides several novelties with respect to the state-of-the-art: (i) robust-to-uncertainty allocation solutions, (ii) multi-objective optimization addressing quality/cost/time, and (iii) encompassing the debugging process in the solution. Furthermore, most previous studies validate models through numerical examples, while we experimented the method within an industrial context. It is our opinion that the lack of case studies is one of the causes of the scarce adoption of quantitative test planning methods.

**X. Conclusions**

We presented a framework for optimal software testing resource allocation under uncertainty. It has been experimented on an industrial case study in the domain of health care systems. Various MOEAs have been compared, and a sensitivity analysis has been conducted to figure out the tradeoffs between accuracy of solution under uncertainty and computational time. The debugging process is taken into account, providing test managers with a more trustworthy prediction of faults expected to be removed, of needed testing time, and especially of required cost, enabling a more accurate decision-making. Assessed properties (solution optimality, robustness to uncertainty, estimates accuracy and scalability) are essential for practitioners who deal with testing of large software systems.

**ACKNOWLEDGMENT**

This work has been supported by the EU FP7 Marie S. Curie IAPP Project ICEBERG (Grant no. 324356).

**REFERENCES**


Roberto Pietrantuono, Ph.D., IEEE Senior Member, received the MS degrees in computer engineering in 2006, the PhD degree in computer and automation engineering in 2009 from the Federico II University of Naples, Italy. In 2011, he co-founder of the Critiware spin-off company (www.critiware.com), an innovative startup active in the field of quality assurance of critical software systems. He is currently research fellow at CINI, the inter-university consortium for informatics, collaborating on national and European projects (main projects were: ICEBERG - http://www.iceberg-sqa.eu/; SVEVIA - http://www.dieti.unina.it/index.php/it/didattica/progetti-di-formazione-pont/244-svevia; TENACE - www.dis.uniroma1.it/tenace/). The main interests are in the area of software quality, software dependability modelling and evaluation, and software V&V for large-scale critical systems. He published several articles in international top-level journals.

Pasqualina Potena is Senior Researcher at RISE SICS Västerås, Sweden. She received the degree in Computer Science from the University of L’Aquila and the Ph.D. degree in Sciences from the University “G. D’Annunzio” Chieti e Pescara (Italy). She was research fellow at the University of L’Aquila, Politecnico di Milano, and University of Bergamo. She also was Experienced Researcher at University of Alcalá (Spain) in the ICEBERG project funded by EU under Industry-Academia Partnerships and Pathways (IAPP) Marie Curie Program (grant 324356). She carries out research in the areas of Quality of Architectures, Architecture-based self-adaptation, and Software Testing of large scale industrial software systems. Her research interests include: non-functional properties (reliability, availability, performance, cost, ...), self-adaptive systems with uncertainties, optimization models, and Search Based Software Engineering (SBSE).

Antonio Pecchia received the B.S. (2005), M.S. (2008) and Ph.D. (2011) in Computer Engineering from the Federico II University of Naples, where he is lecturer in Advanced Computer Programming. He is a post-doc at CINI in European projects, and co-founder of the Critiware spin-off company (www.critiware.com). His research interests include data analytics, log-based failure analysis, dependable and secure distributed systems. He is a member of the IEEE.

Stefano Russo is Professor of Computer Engineering at the Federico II University of Naples, where he teaches Software Engineering and Distributed Systems, and leads the DEpendable Systems and Software Engineering Research Team (DESSERT, www.dessert.unina.it). He co-authored over 160 papers in the areas of software engineering, middleware technologies, mobile computing. He is Senior Member of IEEE.

Daniel Rodriguez, IEEE member, is currently an associate professor (tenured) at the Computer Science Department of the University of Alcalá, Madrid, Spain. In the past, he has been a lecturer at the University of Reading (2001-2006). He earned his degree in Computer Science at the University of the Basque Country (UPV/EHU) and PhD degree at the University of Reading, UK in 2003. His research interest include software engineering in general and the application of data mining and optimisation techniques to software engineering problems in particular. He is a member of IEEE and ACM associations.

Luis Fernández-Sanz is an associate professor at Dept. of Computer Science of Universidad de Alcalá (UAH). He earned a degree in Computing in 1989 at Universidad Politécnica de Madrid (UPM) and his Ph.D. in Computing with a special award at University of the Basque Country in 1997. With more than 20 years of research and teaching experience (at UPM, Universidad Europea de Madrid and UAH), he has also been engaged in the management of the main Spanish Computing Professionals association (ATI: www.ati.es) as vice-president and he has also served in the Board of Directors of CEPI (Council of European Professional Informatics Societies: www.cepis.org) from 2011 to 2013 and again from 2016. His general research interests are software quality and engineering, ICT accessibility and IT professionalism and education.

Pascualina Potena is Senior Researcher at RISE SICS Västerås, Sweden. She received the degree in Computer Science from the University of L’Aquila and the Ph.D. degree in Sciences from the University “G. D’Annunzio” Chieti e Pescara (Italy). She was research fellow at the University of L’Aquila, Politecnico di Milano, and University of Bergamo. She also was Experienced Researcher at University of Alcalá (Spain) in the ICEBERG project funded by EU under Industry-Academia Partnerships and Pathways (IAPP) Marie Curie Program (grant 324356). She carries out research in the areas of Quality of Architectures, Architecture-based self-adaptation, and Software Testing of large scale industrial software systems. Her research interests include: non-functional properties (reliability, availability, performance, cost, ...), self-adaptive systems with uncertainties, optimization models, and Search Based Software Engineering (SBSE).