Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Dynamic test planning: a study in an industrial context

Gabriella Carrozza[1], Roberto Pietrantuono[2], Stefano Russo[2,3]

[1] SESM s.c.ar.l., A Finmeccanica Company, Via Circumvallazione Esterna di Napoli, 80014 Napoli, Italy
e-mail: gcarrozza@sesm.it

[2] DIETI, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy
e-mail: roberto.pietrantuono@unina.it

[3] Critiware spin off, Incubatore Incipit, Complesso Univ. di Monte S. Angelo, Via Cinthia, 80126, Napoli, Italy
e-mail: stefano.russo@unina.it

The date of receipt and acceptance will be inserted by the editor

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

We present a method to dynamically allocate testing resources to software components minimizing the estimated number of residual defects and/or the estimated residual defect density. We discuss the application to a real-world critical system in the homeland security domain. We describe a support tool aimed at easing industrial technology transfer by hiding to practitioners the mathematical details of the method application.

1 Introduction

1.1 *Context and motivation*

In many software systems, testing is the predominant factor of cost. Engineers are constantly under pressure, asked for delivering high-quality products within stringent times and with underestimated resources. For test manager, one crucial problem is to use these resources in the best possible way. The decision on how to effectively distribute the resources available for testing, or how to optimize the budget for a target quality level is of primary concern.

In mission-critical large-scale systems this need is particularly felt, since a lot of effort is devoted to testing compared to design, and the benefit of a proper upfront planning is greater. These systems are often developed in a highly modular way, adopting a strong component-based approach to foster reuse and a build-by-integration approach. However, in many cases, testing processes do not exploit the advantage coming from component-based development adequately. Indeed, it would be desirable for test managers to preliminary allocate testing resources at the beginning, by considering the trade-off between testing effort and achievable quality.

The common industrial practice disregards such an important step. We believe that the main reasons are the lack of simple and tool supported methods, as well as the lack of evidence of success of the proposed approaches into real industrial contexts. There exist indeed mathematically sound methods, but with no proven appropriateness to the specific needs raising from real contexts. Thus, very generic criteria are typically applied in the practice, such as allocating resources driven by requirements (e.g., testing a component until all requirements have been tested at least once), or driven by the size (more testing to bigger modules). Sometimes, intuition drives testing choices: based on experience, a tester may deem one component more "critical" than another, therefore deserving more testing. As there may be relevant differences among components in terms of quality - 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 [1]. Moreover, a component can be newly developed, or it may be a reused unit that already underwent a functional testing phase, hence with a higher testing maturity. These differences intuitively call for a tailored engineering approach, in which more testing resources are spent where there is actually a greater need (i.e., poorer quality).

The work described in this article is developed within an industrial context where the need of such an approach is particularly felt. It is a result of the industry-academia collaboration among Federico II University, SELEX ES and SESM. SELEX ES is a large company manufacturing critical systems in domains such as air and maritime

traffic control, and homeland security. SESM acts as research center supporting the "Software Engineering" unit in the software verification and validation (V&V) activities, and as facilitator of technology transfer.

Despite the great effort of SELEX ES into V&V strategies to ensure high quality software level, as well as into forefront development methodologies and tools, the increasing complexity of the produced systems is demanding additional and strong commitment to identify new means for staying on the cutting edge of V&V techniques. Test planning is one of these key concerns. Indeed, for large-scale critical systems, an accurate planning is difficult, because of the high variability from the planning time to the testing completion. Testing a single component may in fact require months; during this period changes related to the testing process, the environment, the personnel, and the technology often affect the result in terms of defect detection, and are likely to invalidate any assumption made at planning time. Moreover, the diversity of tested units, of their development teams, and of testing teams as well, is a further factor of variability. This is even harder to handle for integration testing, at subsystem or system level, for which components heterogeneity always put testers in a tight spot. Such a context has led us to develop a systematic approach to support test planning in a dynamic way, with expected benefits in terms of effort/quality trade-off.

1.2 Contribution

In this paper, we describe a method to dynamically allocate testing resources to software components, so as to minimize the estimated number of residual defects, and/or the estimated residual defect density, given a fixed testing budget. The method grounds upon software reliability growth models (SRGMs) [2], used at component-level to monitor the testing progress of each component. From these, an estimate is obtained of the quality achievable for a component in relation to the testing effort devoted to it. Then, by iteratively solving an optimization problem, the next testing effort is directed towards the component that contributes the most to reduce the residual number of defects (density) in the overall system, thus improving the final trade-off between effort spent and residual defectiveness.

The proposed solution we have implemented is, unlike existing ones: *i)* dynamic, namely able of using testing data as they become available, exploiting them to adjust performance online, and robust with respect to variations during testing and volatility of planning time's assumptions; *ii)* simple in its application, and with as few assumptions as possible on the testing process; *iii)* ready-to-use, supported by an automatic tool. As result, we formulated the method, provided an algorithm, implemented it in a tool called *effecT!*© (effective Testing), and evaluated everything on the testing process of a SELEX ES system. The system subject to experiments is a critical application in the domain of *homeland security*; its aim is the management of port, maritime, and coastal surveillance. The results show how applying a systematic method to effort scheduling can bring advantages even in presence of variable contexts. This is pushing the company to pursue this direction to replace an intuition-driven test resource allocation method with a quantitative, quality-driven, and tool-supported one.

In the following, the related work in testing resource allocation is first surveyed (Section 2). Then, the proposed method is presented (Section 3). Section 4 reports the results of the study we conducted at SESM to evaluate the approach. Section 5 describes the implemented support tool. Section 6 concludes the paper.

2 Related work

The problem of identifying those parts of a system that should receive more resources during testing has been addressed by many researchers.

Typically, testing aims to detect as many defects as possible. Hence, most criteria try to allocate a greater effort to software modules expected to contain more faults. To this aim, much research focused on *fault-proneness models*, i.e., models able to estimate the defectiveness of software modules based on process or product metrics.

Less frequently, the testing goal is to improve reliability[1] [3]. In this case, effort allocation criteria do not look for modules with more faults, but with a greater impact on overall reliability. Many of such methods in the literature use Software Reliability Growth Models.

The method we present lies in between: it aims at improving the defect detection ability but using SRGMs.

2.1 Fault-proneness models for static defect prediction

Fault/defect[2] prediction usually adopts a"white-box" approach to rate the expected defectiveness of software modules. It starts from a sample of modules with known values for software metrics and known number of experienced defects; the sample is used to build defects predictive models, adopting software metrics as predictors. There is no explicit allocation scheme, but testers are implicitly suggested to put more effort on modules with higher expected defectiveness.

The many works in this literature differ in the types of metrics and data mining techniques. Commonly used metrics are: function/method-level metrics (e.g., cyclomatic complexity), class-level metrics (e.g., number of weighted methods per class, coupling between classes, depth of inheritance), source file metrics (e.g., Lines of Code or of Commented Code per file) [4].

[1] Note that detecting more faults does not imply improving reliability: this requires removing the faults occurring more frequently.

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

In [7], a set of 11 metrics (including McCabe complexity, Lines of Code, Halstead's metrics [5]) is used with regression trees to predict modules more prone to contain faults. Object-oriented (O-O) metrics were proposed in [6] as predictors of faults density; the survey of eight empirical studies in [8] shows that O-O metrics are significantly correlated to faults. Basili *et al.* [9] focused on validating O-O metrics for fault prediction.

Other studies investigated design metrics able to predict modules more prone to failures [10]. In [11], authors adopt logistic regression to relate software measures and fault-proneness for classes of homogeneous products. In [12], authors mined metrics to predict the number of post-release faults in five large Microsoft's projects. They adopted the statistical technique of Principal Component Analysis (PCA) to transform the original set of metrics into a set of uncorrelated variables, to avoid the problem of redundant features (multicollinearity). Subsequent studies confirmed feasibility and effectiveness of fault prediction using public-domain datasets from real-world projects, as the NASA Metrics Data Program, and using regression and classification models [13,14].

In many cases, common metrics provide good prediction results also across several different products. Recent studies focused also on transferring prediction models across different projects and companies [15]. However, it is difficult to claim that a given regression model or a set of models is general enough to be used even with very different products, as discussed in [12,16].

The drawbacks of this approach are: *i)* the need of having of an extensive knowledge base, from which the empirical relation between metrics and defects can be derived; *ii)* the need of the product's source code, in order to extract metrics; *iii)* the lack of formulation of an explicit allocation scheme; these methods do not usually predict the number of defects, but the probability of a module of being defective and/or the ranking of more/less critical modules, thus supporting a "relative" testing effort allocation, not an absolute one; *iv)* metrics-based prediction is "static", in that it does not take into account additional testing data that could adjust the prediction based on online observations; this limits the approach whenever the predicting and the predicted modules' contexts are heterogeneous.

2.2 SRGMs for dynamic resource allocation

Reliability analysis can be conducted by modeling, by measurements, and by hybrid approaches [17–19]. Software Reliability Growth Models (SRGM) are useful means to reliability analysis during testing. They describe how reliability grows as software is improved during testing by faults detection and removal. SRGMs are usually calibrated using failure data collected during testing, namely fitting inter-failure times, and observing the variation of the failure intensity (number of failures per time unit) with testing time. The shape of the intensity curve distinguishes the wide variety of SRGMs [2,20–24].

SRGMs are used to answer questions such as "how long to test a software", or "how many faults are likely to remain"; this system-level usage is the most common.

Fewer works use SRGMs at component-level, to optimize effort distribution among system components while satisfying a reliability target. Among these, Yamada *et al.* [25] formulated two variants of the problem of optimal effort allocation in module testing, assuming the same reliability growth model for all the involved modules. In this and later papers, SRGMs include in the formulation what is known as Testing Effort Function (TEF). In fact, the time dimension for assessing reliability growth can be expressed as calendar time, CPU execution time, number of test-runs, or similar measures: but, in general, the testing effort does not vary linearly with time, and the TEF describes this non-linear relation [26,27].

Lyu et *al.* [28] target the same problem, proposing an optimization model with the cost function based on well-known growth models. They include the use of a coverage factor for each component, to take into account the possibility that a failure in a component could be tolerated. Lyu and Huang formulate the same problem with very little variations [29]. Cost, along with testing effort function, is considered also in later works by the same authors, e.g. in [30]. The authors in [31] also try to allocate optimal testing times to components, with an SRGM limited to the Hyper-Geometric (S-shaped) model. The work in [32] considers also the software architecture implicitly, by taking into account the utilization of each component with a factor assumed to be known. Finally, in our previous work [33], we formulated a testing time allocation problem subject to a minimum reliability constraint, in which we merged the idea of SRGM-based allocation with an architectural model of the system, expressed through a discrete-time Markov chain, so as to explicitly account for components' usage.

The usage of SRGMs implies some assumptions on the process, that are easily violated in practice. These include: perfect repair, immediate repair, independent inter-failure times, no duplicate defect reports, no change to the code during testing, equal probability to find a failure across time units (e.g., also during holidays and vacations) [35], [36]. These two last works argue that SRGMs give good results even when data partly violate the model's assumptions.

The method we presented uses SRGMs, but it has several differences with respect to SRGM-based past works: *i)* the goal is to minimize the *expected number of residual defects* or the *expected residual defect density*, not reliability; this avoids the assumption of conducting testing according to an operational profile[3]; *ii)* the approach uses several different SRGMs and exploits feedback from

[3] Improving operational reliability is more desirable, but it requires the knowledge of the operational profile, seldom available.

ongoing testing, to adjust the allocation online, rather than assuming one model a priori. SRGMs are chosen dynamically, selecting the one that best fits the actual testing data. This greatly reduces the negative impact of the SRGMs' assumptions on result; *iii)* most previous studies validate models through numerical examples outside industrial contexts. These studies do not consider the impact of assumptions violation in real industrial environments. It is our opinion that the lack of success stories is one of the causes of the scarce adoption of quantitative test planning methods. By conceiving and experimenting the method into the SELEX ES industrial process, the impact of assumptions violation is, in a sense, included in the evaluation, thus providing evidence that the approach can work in real settings.

3 The allocation method

3.1 Background

The method aims at minimizing the number of residual defects in the code (*defect-based allocation*) or the residual defect density (*defect density-based allocation*) for a testing budget given in terms of *man-weeks*.

We consider the most common class of SRGMs, those that describe the failing process as a non-homogeneous Poisson process (NHPP). These are characterized by the parameter of the stochastic process, $\lambda(t)$, indicating the failure intensity, and by the *mean value function* (*mvf*), *m(t)*, that is the expectation of the cumulative number of defects detected at time t [22]:

$$N(t)\text{: } m(t) = E[N(t)];\ \frac{dm(t)}{dt} = \lambda(t).$$

These provide indication on how testing is proceeding, namely on how many defects are being detected over time, and how many defects are expected to be found at a certain testing time t. The different types of SRGMs can be described by their mean value function, that appears in this form $m(t) = aF(t)$, where a is the expected number of total defects, and $F(t)$ is a distribution function that can take several forms depending on the failure occurrence process.

Many models have been proposed in the literature, and several tools have been developed to deal with parameterization and fitting of models (such as SMERFS, SoRel, PISRAT, and CASRE). For our purpose, we consider the list reported in Table 1 because of their wide spread in the literature and of their ability to capture several different potential behaviours of the testing process. In particular, we use the model proposed by Goel and Okumoto [20], which describes the failing process by an exponential *mvf* distribution, as it is one of the most successful and popular models for reliability growth analysis. The Delayed S-Shaped curve [21], also very popular, has been proposed in order to capture the possible increasing/decreasing behaviour of the failure rate during the testing process. With similar purposes, the logistic-based distributions (namely, the log-logistic [22] and the truncated logistic [24]) describe the processes in which the initial phase of testing is characterized by a slow increase because of the gradual improvement of testers skills in the initial learning phase, and because of defects being mutually dependent (i.e., some defects are not detectable before some others are). We also consider the generalized version of the Goel-Okumoto model capturing the S-Shaped nature of software failure occurrence, wherein Goel simply proposed an additional parameter turning the exponential into a Weibull distribution [2]. Finally, the normal-based (log- and truncated-normal) SRGMs are considered as they demonstrated a noticeable ability to fit a wide variety of reliability growth scenarios and to software failure data collected in real software projects [23], [42].

Table 1. Software Reliability Growth Models

Model	*m(t) function*
Exponential [20]	$a \cdot (1 - e^{-bt})$
S-shaped [21]	$a \cdot [1 - (1 + gt)e^{-bt}]$
Weibull [2]	$a \cdot (1 - e^{-bt^{\gamma}})$
Log Logistic [22]	$a \cdot \frac{(\lambda t)^{\kappa}}{1+(\lambda t)^{\kappa}}$
Log Normal [23]*	$a \cdot \Phi(\frac{log(t)-\mu}{\sigma})$
Truncated Logistic [24]	$a \cdot \frac{(1-e^{-t/\kappa})}{(1+e^{-(t-\lambda)/\kappa})}$
Truncated Normal [42]*	$a \cdot \frac{\Phi((t-\mu)/\sigma)}{1-\Phi(-\mu/\sigma)}$
* Φ indicates the normal distribution	

Common assumptions of the listed SRGMs
Perfect repair, immediate repair, independent inter-failure times, no duplicate defect reports, no change to the code during testing, equal probability to find a failure across time units (e.g., also during holidays and vacations).

Table 1 reports the models along with the corresponding expression of the mean value function (*mvf*); the estimated number of defects is always the *mvf*'s first parameter, a, while the quantity in parenthesis is $F(t)$. In the formulation of the approach, we considered that, in practice, there is no model to fit all the situations. Thus, we fit testing data with all the considered SRGMs, by using the EM algorithm [40], and then choose the best one as explained in the next Section.

3.2 Method description

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

will thus receive a *testing effort* equal to W_i *man-weeks*[4]. The key idea is to use SRGMs to predict the detection ability of each component's testing process iteratively, and based on that, to allocate resources to components where testing will have the highest detection power. The method is based on the following main steps:

1. *Initialization.* Testing starts at time t_0, when there may be no (previous) data available on testing of components to build any initial SRGM[5]. Without any additional information, which could help to prioritize testing efforts at this stage, the initial allocation is done uniformly to all components, and the testing starts.
2. *Start-up check.* In this initial phase, at each time units (our time unit is the *week*), the method checks if the optimal allocation procedure can be applied with the available defect data. Specifically, we try to fit defect data of each component with every SRGM listed in Table 1 by the EM algorithm [40], and perform a goodness of fit (GoF) test by means of the one-sample Kolmogorov-Smirnov (KS) test (with 90% confidence level) for comparison of samples with reference probability distribution. If the test is satisfied for at least one SRGM, it means that defect data sample can be said, with 90% of confidence, to come from that SRGMs distribution. In this case, the component is ready for the subsequent step (it is said to be statistically valid). In general we will have more SRGMs that fit one component, and will keep track of them for the next steps. This start-up check can be automatically repeated at each time unit from the beginning, or performed when the tester is confident that there are enough data for each component: in the practice, as rule of thumb, we observed that after no more than 20% of the total testing time there is at least one valid SRGM for every component[6]. Thus, we advise to start checking from about 10-15% of the initially allotted testing time on.
   As a guard, we conceived the possibility to skip to the next step also with only a subset of statistically valid components; in such a case (e.g., when there are components with very few and/or highly irregular data), the optimal allocation will apply only to that subset.
3. *SRGM Selection.* Given a number N of components with associated a set of statistically valid SRGMs, we select the best SRGM for each component by applying a goodness-of-fit measure based on the *Akaike Information Criterion* (AIC). In particular, for each SRGM satisfying the KS test, the AIC value is computed as: $AIC = -2log[L] + 2\varphi$, where φ is the degree of freedom for the SRGM, i.e. the number of free parameters, and L is the maximized value of the likelihood function for the estimated model. The model with the lowest AIC value is preferred, denoting the minimal information loss that we incur by selecting that model. The AIC is also successfully used in [41]. If, from the previous step, there is some component with no statistically valid SRGM, these are excluded from the optimal allocation strategy only for that iteration. These components will receive an amount of resources proportionally to their current detection rate[7].
4. *Optimization.* Each component is thus represented by a potentially different SRGM, characterized by its mean value function (*mvf*) as described in Table 1. The *mvf* is used to estimate the number of total defects in the software, as well as the cumulative number of defects detected after a given testing effort. The total number of defects is estimated by the parameter a of the SRGMs *mvf* (cf. with Table 1); we denote it as EST_i (i.e., number of estimated defects for the component i). The number of defects expected at a given effort value is computed by the *mvf* (denoted as *m(.)*). Their difference represents the prediction of residual defects after a given testing effort value, and are the basis of the objective functions to minimize. Depending on the goal (defect or density minimization), one of the following optimization problems is solved:

$$\begin{aligned} min! \;\; E[Defects] = \textstyle\sum_{i=1}^{N}(EST_i - m(W_i^* + W_i)) \\ s.t. \textstyle\sum W_i \leq B^* \end{aligned} \tag{1}$$

$$\begin{aligned} min! \quad E[Density] = \textstyle\sum_{i=1}^{N} \frac{EST_i - m(W_i^* + W_i)}{SIZE_i} \\ s.t. \textstyle\sum W_i \leq B^* \end{aligned} \tag{2}$$

where: N is the number of components; EST_i is the number of expected defects in the i-th component; W_i is the testing effort to allocate to the i-th component; W_i^* is the testing effort already allocated to the i-th component; $m(W_i^* + W_i)$ is the (estimated) number of defects that would be removed if component i receives an effort of $(W_i^* + W_i)$; $SIZE_i$ is the

[4] We assume that the number of man-weeks worked per week is fixed; namely, the assigned resources are spent uniformly across the weeks.

[5] This is the worst case; it may happen indeed that past data exist, and can be used as starting point to build a model.

[6] Authors in [35] indicate a time of 25% to have an SRGM with a definitive accuracy deviation of 20%; however in our method we practically need much less time, since we do not select a definitive model now, but we will iteratively select the best model (and thus improve the initial accuracy deviation) as testing time proceeds.

[7] We assure them to receive a certain amount of resources, in order to not stop completely their testing, so as to have more data for it in the next iteration. This amount will be the same as the previous iteration but diminished by a factor $\alpha \in [0, 1]$. The latter is computed as: $\alpha_j = [DR_j - min_i(DR)]/[max_i(DR) - min_i(DR)]$, where DR_j is the current detection rate (defect/weeks) of the component j and the index i spans across components. Thus, if C_j had the best detection rate, it receives the same amount as the previous cycle; otherwise, it is penalized.

size of component i measured in *KLoC*, used to compute the defect density; B^* is the residual budget at the current iteration[8].

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

The simplified algorithm follows.

The dynamic allocation algorithm.
Input: *budget*, *nComp*, SIZE, *reallocStep*, *startCheck*;

```
//upper case variables are arrays; lower case are scalar
1. i=1; t=0; ready=false
2. while i ≤ nComp do
3.     W*_i = W_i = budget/nComp;//Uniform Allocation
4. end while
5. // start testing
6. while !ready do
7.     // do testing
8.     t++;
9.     if (t ≥ startCheck)
10.          D-DATA = readData();// read defect data
11.          SRGM-MATRIX = fitData(D-DATA);
12.          ready=checkSRGM(SRGM-MATRIX);
13.    end if
14. end while
15. toAllocate = updateBudget(budget, t); //for the next step
16. while (toAllocate > 0) do
17.    SRGM = selectSRGM(SRGM-MATRIX);
18.    W = solveOpt(SRGM,D-DATA,SIZE,W*,toAllocate);
19.    //compute actual effort employed
20.    r = computeResidualEffort(W, t);
21.    if (r !=0) W' = allocateResidualEffort(W, r);
22.    else W'=W;
23.    // do testing with the allocated W* man-weeks
24.    t = t+ reallocStep; //for the next step
25.    toAllocate = updateBudget(budget, t); //for the next step
26.    if (toAllocate > 0)
27.          D-DATA = readData();
28.          SRGM-MATRIX= fitData(D-DATA);
29.    end if
30. end while
31. end
```

The algorithm starts with a uniform allocation of the budget to components. It takes as input parameters: the budget to allocate, the number of components, their size, the reallocation step, namely how often to recompute the allocation, and the time at which to begin the start-up check. Initially, the allocation is done uniformly (lines 2-4); then, during testing, the start-up check is performed to decide when there are enough data for the next step (lines 6-14). In particular, the *readData* function gets defect data for each component into the D-DATA matrix (line 10); the *fitData* function (line 11) iteratively fits data with all SRGMs, performing the KS test, and provides the SRGM-MATRIX with valid SRGMs per component; the *checkSRGM* function receives this matrix and sets *ready* to true when all components have at least one valid SRGM[9]. When there are enough data, the step 3, 4, and 5 of the method start; namely, the residual resources (*toAllocate*) are re-allocated optimally and iteratively updated (we enter the *while* loop, line 16).
The SRGM-MATRIX is given to input to the *selectSRGM* function (line 17), that computes the AIC values and outputs the best SRGM for each component (hence, SRGM is a vector). These are then given as input to the *solveOpt* function (line 18), which solves one of the two optimization problems formulated above, depending on the tester's choice. This returns the optimal allocation vector W[10].
The function *computeResidualEffort* at line 20 calculates the resources (in man-weeks) remaining after the allocation: for instance, if a component receives 1 man-week, and the next update is in 4 weeks, there are unspent resources that could be potentially re-employed.
The function *allocateResidualEffort* is in charge of reallocating those resources to the other components, proportionally to the W vector of already allocated efforts. The final allocation is given by the W' vector, which considers the optimal allocation plus the possible residual efforts redistributed. After testing with those resources, the remaining budget is updated for the next allocation step (line 25). If there are still resources to allocate, the new defect data are read and used to build the SRGM-MATRIX for the next iteration. The procedure ends when resources are no more available.

Note that, in general, the best frequency of reallocation (i.e., the re-allocation step) is dependent on the

[8] B^* also takes into account the possible resources already allocated to non-statistically valid components (see note 7)

[9] This function also manages the case in which the tester wants to go on with even a subset of statistically valid components; in such a case, the function sets *ready* to true when at least x components are valid, with x decided by the tester (omitted for simplicity).

[10] It also applies the detection rate proportional allocation described above (note 6), in case there is some component with no valid SRGM

specific case under study and on defect data occurrence pattern. The ideal case would be to reallocate whenever defect occurrence trend is noticeably and stably changing in some of the components under test. In the practice we observe that, for datasets/systems resembling the one experimented hereafter, a minimal re-allocation step of 1 month (i.e., 4 weeks) is required, in order to have enough time to observe stable changes in the detection rates while avoiding too frequent reallocations. Alternatively, the allocation can be done not necessarily at a fixed frequency, but each time the tester wants to reallocate the effort (i.e., the update rate can be variable too). By a slight variant of the algorithm (i.e., modifying the *updateBudget* function and removing the *reallocStep*), tester can decide when to reallocate resources by observing, online, the detection rate trends, visually revealing changes in one or more components.

4 Evaluation

4.1 Context

We show the application of the method, describing the experiment conducted within the mentioned industry-academia collaboration.

SELEX ES systems are typically built with a component-based development approach. Components are named CSCIs (Computer Software Configuration Items), according to [34]. Each component is first tested white-box, then it undergoes a black-box functional test that aims at marking it as “ready-to-use”. This functional testing step is performed by the Software Verification unit, when a major release (known as build) of a component is delivered: if this gets through, the CSCI is “qualified” internally and it is ready to be integrated with other CSCIs. The Software Verification Unit is also in charge of verifying the integrated system (or even system of systems) before starting the acceptance testing of the final solution. The case study we present is a system for homeland security, in charge of managing the port, maritime, and coastal surveillance. More in detail, the system is made up of five CSCIs, whose characteristics are summarized in Table 2 as far as they are of interest for this paper. CSCIs names, as well as further details on the system, are omitted for confidentiality reasons.

We point out the following system/process features:

- CSCIs perform different missions, from low-level drivers to Web applications and GUIs;
- CSCIs are written in various programming languages (C, C++, JAVA);
- CSCIs have been developed by different teams;
- some CSCIs were outsourced, hence some teams were external to SELEX ES;
- CSCIs development times vary from a few weeks to 12 months;
- CSCIs defects/bugs have been stored in several ways, using different tools (e.g., Mantis bug tracker or unstructured xls files) and logging different information;
- neither developers/testers nor project managers were aware of this study, so as to avoid possible bias.

4.2 Procedure

CSCIs have been tested between 2009 and 2012, using a total amount of testing resources of 326 man-weeks, detecting in total 1,119 bugs. On this dataset, we compare various allocation schemes to figure out if, and to what extent, the proposed method outperforms the others.

The comparison is with a *uniform allocation* (same number of weeks for all CSCIs), and a *size-based allocation* (a common rule-of-thumb approach), proportional to the size of CSCIs (we used LoC as size measure). Overall, we compare four schemes: *i) uniform*; *ii) size-based*; *iii) defect-based*; *iv) defect density-based.* We compare the results in terms of total number of defects that would have been detected by allocating effort according to each scheme.

Thus, we use the same data for all the cases, so as to avoid the bias that could be introduced by using different testing techniques, different testers, technologies, environment, and in general different testing processes. In fact, the objective of the experiment is to figure out which allocation scheme performs better given the same underlying testing process.

Detected defects for each CSCI are in Figure 1(a)-1(e). The trends show that a method should avoid to allocate man-weeks to CSCIs in which no, or few, defects will be detected, saving those resources for a CSCI testing with a higher detection rate (e.g., the first weeks of CSCI 1 and 2). To be sure to have enough data for each CSCI, we assume an initial budget of 150 man-weeks of testing[11] to distribute among CSCIs. In both schemes defect- and density-based schemes of our method, the “update” step is set to 4 weeks, namely the allocation is recomputed each month. Moreover, the first allocation, carried out uniformly, is kept until the 8-*th* week, since it has been the lower limit to have statistically significant SRGMs.

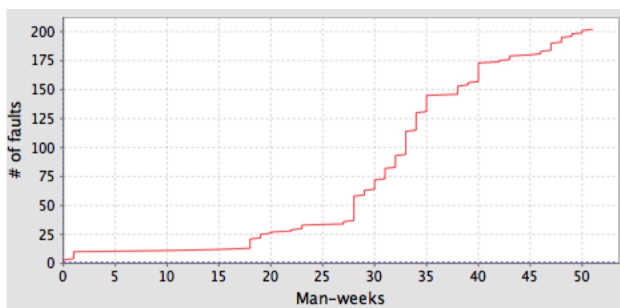
4.3 Results

Table 3 shows the results obtained by the uniform allocation. From the initial budget $B = 150$ man-weeks, the uniform scheme simply assigns 30 man-weeks per CSCI. The number of defects detected is observed after 8 weeks, and then every 4 weeks, in order to have a comparison with the proposed method (there is no dynamic

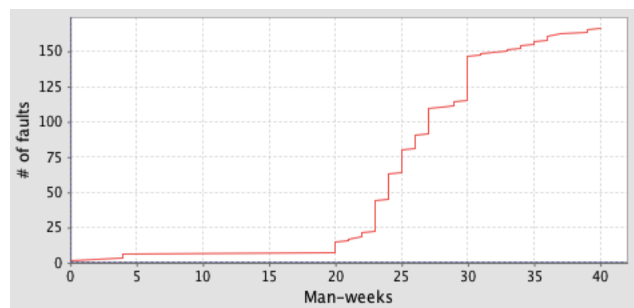
[11] We assume to have a budget lower than 326 man-weeks, so as to avoid to allocate a number of man-weeks to a CSCI greater than the amount actually available (e.g., CSCI 1 has been tested with 32 man-weeks, if the number of man-weeks allocated by the scheme is greater than 32, the experiment may be invalid).

Table 2. Features of the analyzed CSCIs components

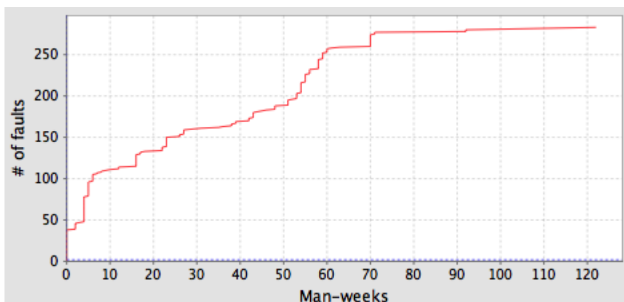
CSCI	LoC	Development Time (months)	Description
C1	39059	11	The CSCI has the primary goal of assuring interoperability during Expeditionary Warfare operations between operative/strategic and tactical systems for Network-Centric Operations support.
C2	55154	6	The CSCI is in charge of managing anomalies, alarms and smart agents associated with maritime track
C3	22208	10	It is in charge of managing presentation layer components
C4	59535	12	It manages standard-compliant messages and related standard communication protocols
C5	34700	10,5	It is responsible for verification and validation of messages of application-level protocols, and their correct sending/publication/receiving/representation



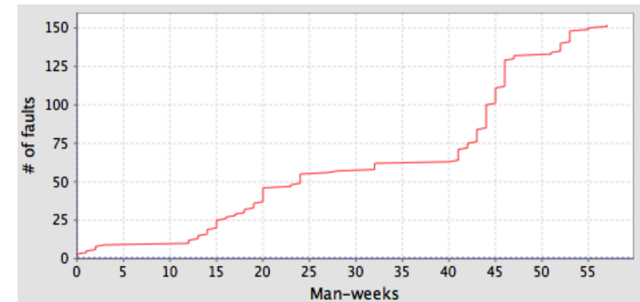
(a) Detected defects for CSCI 1



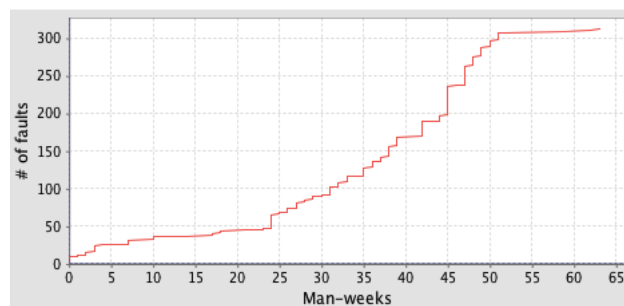
(b) Detected defects for CSCI 2



(c) Detected defects for CSCI 3



(d) Detected defects for CSCI 4



(e) Detected defects for CSCI 5

Fig. 1. Cumulative number of detected defects for each component

allocation update at the monitored points, since this is a static scheme). After 8 weeks a high number of defects is detected for CSCI 3, where more than 100 defects are detected very soon. Resources are however predefined, and the same amount of effort (i.e., 30 man-weeks per each) is preserved for each CSCI. With time, the number of defects gets to 488. This is definitely a good number, especially thanks to CSCI 2, which has an abrupt increase only after 24 weeks; before that time, the number of defects detected was 354.

The size-based scheme allocation is reported in Table 4, again monitored every 4 weeks, after the initial 8 weeks. In this scheme, we envisaged a re-employment of unspent resources: as may be seen from the first row of Table 4, there are three cases in which the number of man-weeks is less than 30; thus, when those efforts have been spent (e.g., for CSCI 3 after 15 weeks, assuming one man), residual man-weeks are re-employed proportionally to the size of CSCIs. In this case, results of the first 15 weeks are the same as the uniform case; at week 16, CSCI 3 exhausts its resources and we no longer observe defects detected. It similarly happens later on for CSCIs 1 and 5; on the other hand, the man-weeks allocated to C2 and C4, namely 38 and 41 man-weeks, make them achieve 162 and 128 detected defects, many more than the corresponding ones in the uniform allocation. The second approach ends up with 523 detected faults, 35 more than the previous one. Accounting for the size improved the final result.

Our dynamic defect-based and density-based allocations (Tables 5 and 6, respectively) reserved initially 8 man-weeks per CSCI, with updates each 4 weeks. After the first 8 weeks, a new solution is computed each 4 weeks, with available man-weeks re-allocated to each CSCI. To this aim, a new SRGM is built at each update and for each CSCI, by considering the defect data collected up to that week (if no defect is detected, the SRGM remains unchanged). This SRGM provides the estimate of the total number of defects contained in a CSCI (i.e., the EST_i parameter of the objective function), and is used to obtain the prediction of defects that will be detected under the proposed solution - namely, by assigning the resources to the CSCI as suggested by the solution (i.e., the $m(W^* + W)$ parameter of the objective function). As more defects are found, the SRGM changes, and thus the total estimate and the predicted number of detected defects will change too.

Results are reported in Table 5. Each row reports, in the right part, the number of residual man-weeks to allocate after x weeks, and the computation of the new allocation replacing the previous one; in the left part, each row reports, for each CSCI: the number of defects actually detected after x weeks (denoted with DET), the EST value, and the $m(W^* + W)$ value under the found solution. The difference between EST and $m(W * + W)$ is the prediction of residual defects.

The allocation starts with 30 man-weeks per CSCI; after the first 8 weeks, based on detected defects, the allocation changes, and the method suggests placing more resources on C3 and C5, where detection is working better. Let us observe the results after 12 weeks. Both the 8-*th* and 12-*nd* week results have the same number of detected defects than the previous approaches. This is because after 12 weeks there is no CSCI without resources. However, at this time the allocation is recomputed based on SRGMs built with data from week 0 to week 12, and there are CSCIs that will receive no effort (C2) or a low effort (C1) in the next weeks, for the bad detection of their testing. On the other hand, C4 exhibits a detection trend from week 8 to 12 with an abrupt increase, which makes the SRGM estimate 1105 defects. Accordingly, 33.05 man-weeks are allocated to C4, predicted as sufficient to detect all the defects; the residual resources are distributed between C3 and C5. After 20 weeks, the defects detection trend in C5 makes the SRGM predict a significantly higher EST value than the others (at week 20, it detected 68-40=20 defects in 4 weeks): with this rate, the predicted $m(W^* + W)$ value is also high, and thus all the resources are devoted to it to minimize the predicted residual defects. Note that, at week 20, the detection rate of C3 is even better (namely 159-132= 27 defects in 4 weeks); however the shape of the detection trend is smoother when approaching to week 20, leading the SRGM to estimate lower values for EST, and thus for $m(W^* + W)$. The algorithm finds more convenient to place all the resources on C5 as, according to the prediction, C3 has lower improvement margin (EST is estimated at 171.63, and the current detected defects are DET = 159; thus only EST-DET = 12.63 defects could be detected; whereas, for C5, there is much room for improvement: $EST - DET$=287.8). This is a good choice, since C5's detected defects go from 68 to 312 in 8 weeks. At week 24, it detected 239-68=171 defects in 4 weeks.

After 28 weeks, the detection rate of C5 becomes slower, as it is going toward the saturation. The SRGM estimates a total of 316.43 defects being present in the CSCI, and the room for improvement is therefore only $316.43 - 312 = 4.43$ defects. Contrarily, the SRGM of C3 and C4 (which are the same since the last detection, namely, since week 20) tell that the residual defects for C3 and C4 are 12.62 and 5.69 respectively. The tool redistributes the residual 10 man-weeks to these two CSCIs, managing to detect further 7 defects with that effort. The final result is 552 defects detected (i.e., 13.11% more than uniform, 5.55% more than size-based), with an estimated defect density of 0.5869 defects/KLoC (computed as $\sum_i (EST_i - Detected_i)/SIZE_i$).

Finally, the density-based allocation (Table 6) exhibits results very similar to the defect-based one; also in this case C1 and C2 receive no more resources after 20 weeks, and the allocations are slightly more balanced because of the smoothing effect of the CSCI sizes. The

Table 3. Results of the uniform allocation

	C1	C2	C3	C4	C5		
Allocated man-weeks	30	30	30	30	30		
Detected Defects per CSCI						**Total Det. Defects**	*Residual man-weeks*
after 8 weeks	10	6	107	9	31	**163**	*110*
after 12 weeks	11	6	112	46	36	**211**	*90*
after 16 weeks	12	6	129	55	37	**239**	*70*
after 20 weeks	27	15	133	57	44	**276**	*50*
after 24 weeks	33	44	150	62	65	**354**	*30*
after 28 weeks	51	110	159	62	85	**467**	*10*
after 30 weeks	**63**	**114**	**160**	**62**	**89**	**488**	*0*

Table 4. Results of the size-based allocation

	C1	C2	C3	C4	C5		
Allocated man-weeks	27.46	38.78	15.61	41.87	26.29		
Detected Defects per CSCI						**Total Det. Defects**	*Residual man-weeks*
after 8 weeks	10	6	107	9	31	**163**	*110*
after 12 weeks	11	6	112	46	36	**211**	*90*
after 16 weeks	12	6	114	55	37	**224**	*70*
after 20 weeks	27	15	114	57	44	**257**	*50*
after 24 weeks	39	54	114	76	65	**348**	*30*
after 28 weeks	51	148	114	94	68	**475**	*10*
after 30 weeks	**51**	**162**	**114**	**128**	**68**	**523**	*0*

detected defects are the same of the previous scheme up to week 24 (with the exception of C3, where the effect of the reemployment of resources makes it detect one additional defects at week 16). After week 24, the results slightly change because of the different allocation between C3, C4, and C5. The final result is similar to the defect-based allocation, namely 547 defects (12.09% and 4.58% more than uniform and size-based, respectively), but with a better estimated density, 0.5572 defects/KLoC, which was the pursued objective.

Figure 2 summarizes the total number of defects found by the various approaches as time (weeks) proceeds; this provides an easy comparison of methods' detection ability over time. The histogram bars have the same value for the first 12 weeks. Then, the dynamic schemes outperform the others.

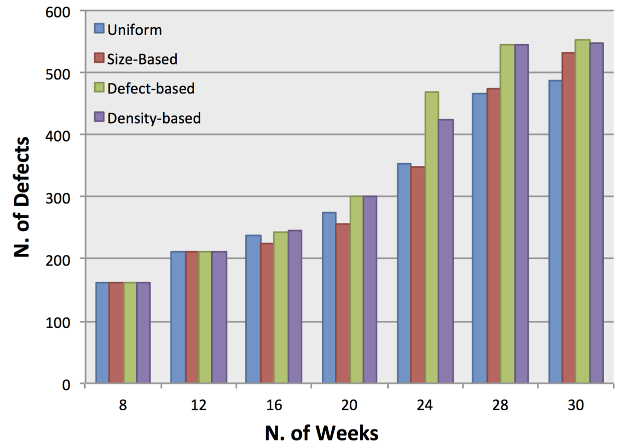


Fig. 2. Number of detected defects over testing time

4.4 Remarks

Assumptions like perfect and immediate debugging, inter-failure times independence, unchanged code, equal testing quality over time, no differences among testing teams, and similar ones, are typically violated in real contexts. Our SRGM-based method showed to be effective in a real context, in spite of possible assumptions' violations. The mentioned study presented in [35] reported that SRGMs provided good results after about 25% of total testing time, with prediction accuracy deviating only by 20% even with partially violated assumptions. In our case, for the extent of assumptions' violations, the model fitted with 25% of time was statistically valid but turned out to be not the definitive ones; in the remaining 75%, the selection of the best SRGM changed several times, because of data variability[12]. For instance, there are components like C1 and C2 where testing gave no result for months (ten/fifteen weeks) and abruptly improved after week 20: defects revealed in the later weeks clearly show that this is not because of the components' greater quality, but more likely because of a lack of good testing before week 20. This may depend on several reasons, related to human, technical, environmental, or technological factors

[12] The models that more often fitted data have been: *exponential*, especially in the beginning, *truncated logistic*, and *truncated normal*.

Table 5. Results of the defect-based allocation

Iteration *time in weeks*	Number of Detected Defects (*DET*) Estimated defects (*EST*) Predicted detections (*m(W*+W)*)					**Total Detected Defects**	**Man-weeks to (Re-)allocate**	Reallocation of man-weeks (*#of allocated man-weeks*)				
	C1	**C2**	**C3**	**C4**	**C5**			**C1**	**C2**	**C3**	**C4**	**C5**
0 weeks (t_0)												
DET	0	0	0	0	0	0	*150*	30	30	30	30	30
8 weeks												
DET	10	6	107	9	31	**163**	*110*	5.93	5.93	52.41	5.99	39.72
EST	11.00	6.02	128.67	9.01	35.12							
m(W+W)*	10.99	6.00	128.66	9.00	35.11							
12 weeks												
DET	11	6	112	46	36	**211**	*90*	1.41	0	26.69	33.06	28.83
EST	11.00	6.02	118.97	1105.58	39.04							
m(W+W)*	10.99	6.00	118.97	1105.5	39.00							
16 weeks												
DET	11	6	132	55	40	**244**	*70*	0	0	38.45	1.69	29.86
EST	11.00	6.02	140.92	55.53	42.52							
m(W+W)*	10.99	6.00	140.90	55.52	42.50							
20 weeks												
DET	11	6	159	57	68	**301**	*50*	0	0	0	0	50
EST	11.00	6.02	171.62	62.69	355.80							
m(W+W)*	10.99	6.00	159.08	57.00	169.09							
24 weeks												
DET	11	6	159	57	239	**469**	*30*	0	0	0	0	30
EST	11.00	6.02	171.62	62.69	432.94							
m(W+W)*	10.99	6.00	159.08	57.00	331.13							
28 weeks												
DET	11	6	159	57	312	**545**	*10*	0	0	7.84	2.15	0
EST	11.00	6.02	171.62	62.69	316.43							
m(W+W)*	10.99	6.00	166.02	61.10	316.26							
30 weeks												
DET	11	6	161	62	312	**552**	*0*	-	-	-	-	-

changing over time; in any case, it is a clear example of violation of SRGM assumptions. Thanks to the periodic re-computation of SRGMs, the method resulted to be a robust solution to such noisy variations. In fact, in this specific study, the strange behaviour of CSCI 1 and 2 favored the static approaches, which had a sufficient amount of man-weeks to intercept the increase of detected defects in both CSCIs (approximately 20 man-weeks). If the same schemes were used with 100 man-weeks (or with more regular data for C1 and C2), the static methods did not intercept the increasing points, detecting many fewer defects with respect to the dynamic case. Notwithstanding this advantageous situation for the static scheme, the dynamic methods inevitably outperformed it, because they allowed to save precious resources in the first weeks to be reallocated elsewhere, pursuing globally more efficient allocations toward other CSCIs.

5 The support tool

The allocation schemes are implemented in the *effecT!*© test planning tool by the academic spin off company *Critiware*. The tool consists of an *Analyzer* and a *Planner* (Figure 3). The *Analyzer* allows to evaluate the effectiveness of the testing and defect fixing processes through a series of reports showing the detected defects, their type and severity, the assigned priority, the V&V techniques which detected them, the reproducibility features, the time-to-fix distribution, the defect opening *vs.* closing curves. We focus here on the *Planner*, which is responsible for providing testers with a means to plan the allocation of testing effort optimally[13]. Both blocks exploit the underlying database (DB), containing information on testing sessions (and on the fixing process) performed on each CSCI. This includes data on detected defects, their detection time, their severity, priority, and

[13] The *Analyzer* is left out from the explanation, since it does not regard the test planning issue presented here.

Table 6. Results of the defect density-based allocation

Iteration *time in weeks*	Number of Detected Defects (*DET*) Estimated defects (*EST*) Predicted detections (*m(W*+W)*)					**Total Detected Defects**	**Man-weeks to (Re-)allocate**	Reallocation of man-weeks (*#of allocated man-weeks)*				
	C1	**C2**	**C3**	**C4**	**C5**			**C1**	**C2**	**C3**	**C4**	**C5**
0 weeks (t_0)												
DET	0	0	0	0	0	0	*150*	30	30	30	30	30
8 weeks												
DET	10	6	107	9	31	**163**	*110*	17.72	17.72	37.94	17.72	18.89
EST	11.00	6.02	128.67	9.01	35.12							
m(W+W)*	11.00	6.01	128.66	9.00	35.11							
12 weeks												
DET	11	6	112	46	36	**211**	*90*	0.15	0.14	34.91	31.33	23.44
EST	11.00	6.02	118.97	1105.58	39.04							
m(W+W)*	11.00	6.01	118.97	1105.5	39.00							
16 weeks												
DET	11	6	133	55	40	**245**	*70*	0.87	0.84	38.74	0.85	28.69
EST	11.00	6.02	140.36	55.53	42.52							
m(W+W)*	11.00	6.01	140.34	55.52	42.50							
20 weeks												
DET	11	6	159	57	68	**301**	*50*	0	0	1.33	0	48.66
EST	11.00	6.02	171.62	62.75	355.80							
m(W+W)*	11.00	6.01	161.76	55.98	166.62							
24 weeks												
DET	11	6	160	57	189	**423**	*30*	0	0	0	0	30
EST	11.00	6.02	168.57	62.69	405.72							
m(W+W)*	11.00	6.01	160.01	55.98	329.67							
28 weeks												
DET	11	6	160	57	312	**546**	*10*	0	0	2.65	0	7.35
EST	11.00	6.02	168.57	62.75	337.03							
m(W+W)*	11.00	6.01	161.86	55.98	325.47							
30weeks												
DET	11	6	161	57	312	**547**	*0*	-	-	-	-	-

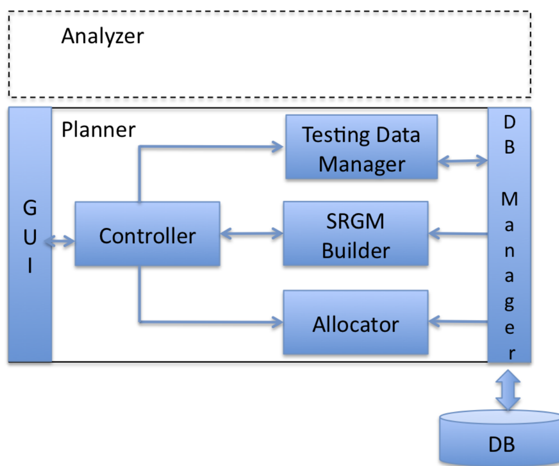


Fig. 3. Architecture of the *effecT!*© support tool

data on the testing session itself, such as the start/end date, the testing techniques used, the V&V phase.

The *Planner* comprises the following modules:

– **Testing Data Manager**. The module manages information provided by the tester via the GUI, regarding *i)* the name of the software components to which tester wants to allocate resources, *ii)* the budget to allocate (in man-weeks or man-months), and *iii)* the desired re-allocation step[14]. If components are reused modules, a testing session was performed in the past; in this case data about detected defects are retrieved from the DB. Alternatively, the module can import these data from an external *.csv* file. If they are new CSCIs, for which no testing data are available yet, the tool provides a *defect tracking* functionality: the tester inserts data, through the GUI, about each new defect detected during testing, and stores them in the DB.

[14] The re-allocation step input can also be empty, giving the possibility to the tester of requiring the re-allocation at any time s/he wants by just pressing a button.

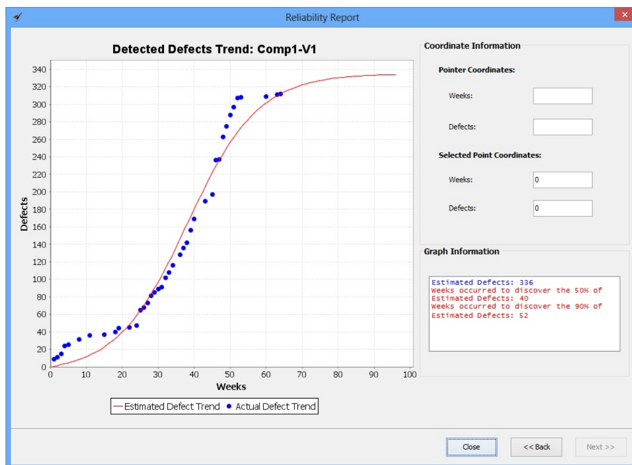


Fig. 4. Screenshots of the *effecT!*© tool

- **SRGM Builder**. This module uses defect data produced by the previous module about CSCI testing sessions, and iteratively fits them with a set of 7 SRGMs. The Builder shows all SRGMs to the tester, along with the goodness-of-fit test result and the AIC value, and suggests the best fitting model[15].
- **Allocator**. This module implements the algorithm for the optimization problems (Eq. 1 and 2), taking the SRGM for each CSCI as input, and the defect data. It outputs the vector of values suggesting the effort to be devoted to each CSCI.
- **Controller**. This module coordinates all the operations: its core is the algorithm presented in Section 3. Thus, it receives commands from the GUI, queries the previous modules (i.e., at each cycle, it retrieves data from **Testing Data Manager**, fits them with SRGMs by the **SRGM Builder**, and recalls the allocator to distribute the residual effort), and provides results back to the tester.

A screenshots of *effecT!*© is in Figure 4. It shows an output of the *SRGM builder*, namely an SRGM obtained from data of a CSCI.

6 Conclusions

In the current industrial practice, the task of allocating testing resources to components of a large software system is often left to the sensibility of test/project managers, who typically decide on the basis of their experience or intuition. Simple yet sound engineering methods are highly desirable to guide their choices in a systematic and measurable way. We have presented a method to compute dynamically the resources to allocate for the testing of software components, so as to minimize the number of residual defects and/or the estimated residual product defect density.

The method stems from the actual need of a large system integration company. It has been applied with reference to a real-world critical software system for homeland security, whose coarse-grain components are developed and tested over several months. The results report on the improvement of the testing process outcome, given a predefined amount of testing resources.

The spread of any new method in the industrial practice requires proper knowledge and technology transfer means; for this reason, we have realized and described the architecture of a tester support tool. This eases the application of the proposed method, hiding the tester from the complex mathematical details. Future work will address accounting for other historical data, such as defect severity, or context-related information, such process metrics or metrics related to human factors.

7 Acknowledgement

This work has been partially supported by MIUR under project SVEVIA (PON02_00485_3487758) of the public-private laboratory COSMIC (PON02_00669) and by the European Commission in the context of the FP7 project ICEBERG, Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 324356. The work of Dr. Pietrantuono is supported by the project Embedded Systems in Critical Domains (CUP B25B09000100007) in the framework of POR Campania FSE 2007-2013.

References

1. D. Cotroneo, R. Pietrantuono, S. Russo: Testing techniques selection based on ODC fault types and software metrics. Journal of Systems and Software, 86 (6), 1613-1637 (2013)
2. A. L. Goel: Software Reliability Models: Assumptions, Limitations and Applicability. IEEE Transactions on Software Engineering, SE-11(12), 1411-1423 (1985)
3. D. Cotroneo, R. Pietrantuono, S. Russo: Combining operational and debug testing for improving reliability. IEEE Transactions on Reliability, 62 (2), 408-423 (2013)
4. C. Catal, B.M. Diri: A systematic review of software fault prediction studies. Expert Systems with Applications, 36 (4), 7346-7354 (2009)
5. M. Halstead: Elements of Software Science, Elsevier Science, New York (1977)
6. S.R. Chidamber, C.F. Kemerer: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), 476-493 (1994)
7. S.S. Gokhale, M.R. Lyu: Regression Tree Modeling for the Prediction of Software Quality. In: Proc. 3rd ISSAT (1997)
8. R. Subramanyam, M.S. Krishnan: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering, 29(4), 297-310 (2003)

[15] The best fitting SRGM is chosen by default.

9. V.R. Basili, L.C. Briand, W.L. Melo: A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, 22(10), 751-761 (1996)
10. N. Ohlsson, H. Alberg: Predicting fault-prone software modules in telephone switches. IEEE Transactions on Software Engineering. 22(12), 886-894 (1996).
11. G. Denaro, M. Pezzè: An Empirical Evaluation of Fault-proneness Models. In: Proc. 24th Int. Conference on Software Engineering (ICSE), pp. 241-251 (2002)
12. N. Nagappan, T. Ball, A. Zeller: Mining Metrics to Predict Component Failures. In: Proc. 28th Int. Conference on Software Engineering (ICSE), pp. 452-461 (2006)
13. T. Ostrand, E. Weyuker, R. Bell: Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering, 31(4), 340-355 (2005)
14. T. Menzies, J. Greenwald, A. Frank: Data Mining Static Code Attributes to Learn Defect Predictors. IEEE Transactions on Software Engineering, 33 (1), 2-13 (2007)
15. J. Nam, S. Jialin Pan, S. Kim: Transfer Defect Learning. In: Proc. 35th Int. Conference on Software Engineering (ICSE), pp. 382-391 (2013)
16. T. Zimmermann *et al.*: Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In: Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Eng., pp. 91-100 (2009)
17. J.B. Dugan: Automated Analysis of Phase-Mission Reliability. IEEE Transactions on Reliability, 40, 45-52 (1991).
18. M.R. Garzia: Assessing the Reliability of Windows Servers. In: Proc. of IEEE Dependable Systems and Networks conference (2002).
19. R. Pietrantuono, S. Russo, K.S. Trivedi: Online Monitoring of Software System Reliability. In: Proc. of the European Dependable Computing Conference (EDCC), 209-218 (2010).
20. A.L. Goel, K. Okumoto: Time-dependent error-detection rate model for software reliability and other performance measures. IEEE Transactions on Reliability, R-28(3), 206-211 (1979)
21. S. Yamada, M. Ohba, S. Osaki: S-Shaped Reliability Growth Modeling for Software Error Detection. IEEE Transactions on Reliability, R-32(5), 475-485 (1983)
22. S.S. Gokhale, K.S. Trivedi: Log-logistic software reliability growth model. In: Proc. 3rd Int. High-Assurance Systems Engineering Symposium, pp. 34-41 (1998)
23. R.E. Mullen: The lognormal distribution of software failure rates: application to software reliability growth modeling. In: Proc. 9th Int. Symposium on Software Reliability Engineering (ISSRE), pp. 134-142 (1998)
24. H. Okamura, T. Dohi, S. Osaki: EM algorithms for logistic software reliability models. In: Proc. 22nd IASTED Int. Conference on Software Engineering, pp. 263-268 (2004)
25. S. Yamada, T. Ichimori, M. Nishiwaki: Optimal Allocation Policies for Testing-Resource Based on a Software Reliability Growth Model. Int. Journal of Mathematical and Computer Modeling. 22(10-12), 295-301 (1995)
26. C. Huang, S. Kuo, M.R. Lyu: An Assessment of Testing-Effort Dependent Software Reliability Growth Models. IEEE Transactions on Reliability, 56 (2), 198-211 (2007)
27. S. Yamada, H. Ohtera, and H. Narihisa: Software reliability growth models with testing effort. IEEE Transactions on Reliability, R-35, 19-23 (1986)
28. M.R. Lyu, S. Rangarajan, A.P.A. van Moorsel: Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development. IEEE Transactions on Reliability, 51 (2), 336-347 (2002)
29. C.Y. Huang, J.H. Lo, S.Y. Kuo, M.R. Lyu: Optimal Allocation of Testing Resources for Modular Software Systems. In: Proc. 13th Int. Symposium on Software Reliability Engineering (ISSRE), pp. 129-138 (2002)
30. C.Y. Huang, J.H. 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. R.H. Hou, S.Y. Kuo, Y.P. Chang: Efficient allocation of testing resources for software module testing based on the hyper-geometric distribution software reliability growth model. In: Proc. 7th Int. Symposium on Software Reliability Engineering (ISSRE), pp. 289-298 (1996)
32. W. Everett: Software Component Reliability Analysis. In: Proc. Symposium on Application-specific Systems and Software Eng. and Techn. (ASSET), pp. 204–211 (1999)
33. R. Pietrantuono, S. Russo, K.S. Trivedi: Software Reliability and Testing Time Allocation: An Architecture-Based Approach. IEEE Transactions on Software Engineering, 36 (3), 323-337 (2010)
34. U. S. Department of Defense, MIL-STD-498. Overview and Tailoring Guidebook, 1996. [Online]. Available at: www.abelia.com/498pdf/498GBOT.PDF
35. V. Almering, M. Van Genuchten, G. Cloudt, P.J.M. Sonnemans: Using Software Reliability Growth Models in Practice. IEEE Software, 24 (6), 82-88 (2007)
36. C. Stringfellow, A. Amschler Andrews: An Empirical Method for Selecting Software Reliability Growth Models. Empirical Software Engineering, 7 (4), 319-343 (2002)
37. W. Farr: Handbook of Software Reliability Engineering, M.R. Lyu (Ed.), chapter: Software Reliability Modeling Survey, pp. 71-117. McGraw-Hill, New York, NY (1996)
38. J.D. Musa, K. Okumoto: A logarithmic Poisson execution time model for software reliability measurement. In Proc. 7th Int. Conference on Software Engineering (ICSE), pp. 230-238 (1984)
39. B. Zachariah, R.N. Rattihalli: Failure Size Proportional Models and an Analysis of Failure Detection Abilities of Software Testing Strategies. IEEE Transactions on Reliability, 56 (2), 246-253 (2007)
40. H. Okamura, Y. Watanabe, T. Dohi: An iterative scheme for maximum likelihood estimation in software reliability modeling. In: Proc. 14th Int. Symposium on Software Reliab. Eng. (ISSRE). IEEE CS Press, pp. 246256 (2003).
41. K. Ohishi, H. Okamura, T. Dohi: Gompertz software reliability model: Estimation algorithm and empirical validation. Journal of Systems and Software, 82 (3), 535-543 (2009)
42. H. Okamura, T. Dohi, S. Osaki: Software reliability growth model with normal distribution and its parameter estimation. In: Proc. Int. Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE), pp. 411-416 (2011)