Information and Software Technology xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A software quality framework for large-scale mission-critical systems engineering

Gabriella Carrozza[a], Roberto Pietrantuono[*,b], Stefano Russo[b]

[a] Accenture Operations, Piazzale dell'Industria 40, Rome 00144, Italy
[b] Università degli Studi di Napoli Federico II, Via Claudio 21, Naples 80125, Italy

ARTICLE INFO

Keywords:
Software quality
Decision support systems
Mission-critical systems
Software testing
Software reliability
Automatic static analysis
Testing resource allocation
Defect prediction
Fault prediction

ABSTRACT

*Context:*In the industry of large-scale mission-critical systems, software is a pivotal asset and a key business driver. Production and maintenance costs of systems in domains like air/naval traffic control or homeland security are largely dependent on the quality of software, and there are numerous examples where poor software quality is blamed for major business failures. Because of the size, the complexity and the nature of systems and engineering processes in this industry, there is a strong need yet a slow shift toward innovation in software quality management.

*Objective:*We present SVEVIA, a framework for software quality assessment and strategic decisions support for large-scale mission-critical systems engineering, and its application in a three years long industry-academy cooperation.

*Method:*We started with the analysis of the industrial software quality management processes, and identified the key challenges toward a satisfying quality-cost-time trade-off. We defined new methods for product/process quality assessment, prediction, planning and optimization. We experimented them on the industrial partner systems and processes. They finally conflated in the SVEVIA framework.

*Results:*SVEVIA was integrated into the industrial process, and tested with hundreds of software (sub)systems. More than 20 millions of lines of code – deployed in about 20 sites in Italy and UK – have come under the new quality measurement and improvement chain. The framework proved its ability to support systematic management of software quality and key decisions for productivity improvement.

*Conclusion:*SVEVIA supports team leaders and managers coping with software quality in mission-critical industries, yielding figures and projections about quality and productivity trends for a prompt and informed decision-making.

1. Introduction

Modern systems for the management of critical infrastructures – e.g. air/maritime traffic control, power grids, homeland security – heavily ground on software. As numerous functions are software-implemented, software has become a dominant part, and inevitably put in charge of much of their quality, production cost and time-to-market. Software quality is reported to worth more than $500 billion per year worldwide, and quality excellence practices yield a ROI of about 15$ per 1$ spent [1]. At the same time, experience shows that poor software quality has been responsible for severe business and safety disasters [2].

In the large-scale mission-critical systems industry there is a discrepancy between the acknowledgment of the primary role of software and the engineering practices adopted for software quality. This is due to technical, organizational and cultural reasons. The size and complexity of systems make difficult and expensive the application of best practices for software quality. Differently from other sectors where software is involved, industries in this field are typically big, with many employees and outsourcing companies. Innovations in software quality practices are hard to introduce, because of start-up and re-engineering costs (training and process changes), as well as because of skepticism of managers, who are often anchored to consolidated industrial processes where software is seen just as an intangible "add-on" to the concrete system. In such contexts, the generic perception of the importance of software quality needs to be supported in quantitative ways, integrated into wider-scope *system engineering* processes.

We present the results of a three-year industry-academia partnership focused on innovating software processes in systems engineering, in the framework of the COSMIC public–private laboratory between Finmeccanica and Federico II University. The partner company

* Corresponding author.
E-mail addresses: gabriella.carrozza@accenture.com (G. Carrozza), roberto.pietrantuono@unina.it, stefano.russo@unina.it (R. Pietrantuono).

https://doi.org/10.1016/j.infsof.2018.05.009
Received 28 September 2017; Received in revised form 5 April 2018; Accepted 22 May 2018
0950-5849/ © 2018 Elsevier B.V. All rights reserved.

produces systems in domains like air and naval traffic control and homeland security, whose software part has thousands of requirements and millions of lines of code. The work resulted in the SVEVIA framework for strategic decisions support in software quality management and productivity improvement.

SVEVIA provides team leaders and managers with quantitative views about trends of software quality and productivity. It considers three key dimensions: *software **quality**; development and quality assurance **cost**; **time**to market*. Trade-offs among them are quantified and related to management decisions, as:

- Prediction of the best time to release products;
- Identification of software quality assurance activities (e.g., static analysis, code inspection, testing) providing best returns;
- Optimal planning and distribution of efforts to quality assurance activities to attain quality goals while minimizing risks and/or delivery time;
- Identification of software components and of production phases mostly impacting quality and schedule;
- Assessment of performance of external suppliers against quality targets.

SVEVIA supports software quality management (SQM) decisions through:

- **Measurement and estimation**, aimed to monitor/gauge the *effectiveness* and *efficiency* of activities by traditional as well as newly defined *key performance indicators* (KPI) for quality and productivity;
- **Prediction** of quality-cost-time trends, based on advanced mathematical models, for quantitative comparison of alternative strategic decisions;
- **Resources optimization**, through models for optimal distribution of efforts in quality assurance activities, given user-specified constraints (e.g., budget) and objectives (e.g., maximize the expected software quality).

We describe the framework and the experiments with tens of large-scale systems, accounting for about 900 components and around 20 millions of lines of code.

The paper is organized as follows. Section 2 introduces the reference systems engineering context and the requirements for the framework. Sections 3 and 4 present the SVEVIA architecture and services. The results of the on-field experiments in the years 2013–2015 are presented in Section 5. Lessons learnt and hints for replicating the framework in similar domains are discussed in Section 6. Related work is discussed in Section 7. Section 8 provides conclusions.

2. Industrial domain and SQM requirements

2.1. The systems engineering industrial context

Most companies in the mission-critical systems market are manufacturers or system integrators (not software houses) of what are often categorized software-intensive systems, including sensors and electronics such as radars and video surveillance equipments. They usually adopt the *V-model* engineering process and the MIL-STD-498 standard for software artifacts and documentation [3].

The industrial partner in this study is a large industry of this kind. Software engineering teams account for about 1000 employees in several plants in Italy and UK, organized in *capabilities* or *sub-capabilities* (e.g., “Surveillance Data Processing”, “Human Machine Interfaces”), and in *units*. Fig. 1 shows the *component-based* V-model process adopted. Components are autonomously deliverable entities known as *Computer Software Configuration Items* (**CSCI**). A CSCI encompasses up to hundreds of thousands of Logical Lines of Code (LLOC); its lifecycle is managed by a Unit. Several CSCIs form a system, managed under a *Project*. CSCIs and Projects have a many-to-many relation.

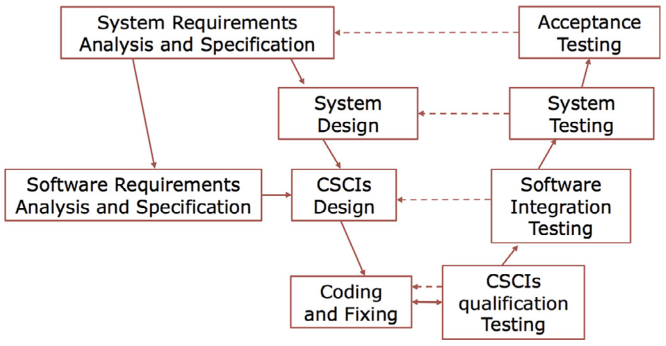

System Requirements Analysis and Specification
Acceptance Testing
System Design
System Testing
Software Requirements Analysis and Specification
CSCIs Design
Software Integration Testing
Coding and Fixing
CSCIs qualification Testing

Fig. 1. The industrial V-model lifecycle.

The main phases and related MIL-STD-498 artifacts are:

- *System requirements analysis and specification*. Output documents are: System/Subsystems Specification (**SSS**); Interface Requirements Specification (**IRS**), with external system interfaces and data model;
- *System design*. Output documents: System Subsystem Design Description (**SSDD**), containing the high-level architecture of the envisaged solution, and the allocation of requirements to subsystems;
- *Software requirements analysis and specification*. Output documents: Software Requirements Specification (**SRS**) for each identified CSCI. Each SRS is complemented by an Interface Control Document (**ICD**) specifying the CSCI interfaces and the related data model;
- *CSCI design*. Output documents: Software Design Description (**SDD**) with the internal design of a CSCI, and the allocation of software requirements to its subcomponents. Each SDD is related to a CSCI and accompanied by an Interface Design Document (**IDD**) that specifies the CSCI internal interfaces and exchanged data;
- *Coding and fixing*: the CSCI source code is produced, with continuous feedback from unit testing to iteratively fix detected defects before releasing the CSCI to the integration testing stage.

The testing phases and main related artifacts are:

- *CSCI unit (or qualification) testing*, producing unit test plan (**STP**) and design (**STD**) documents, running tests, and producing a Software Test Report (**STR**);
- *Software integration testing*, yielding Software Integration Test Description (**SITD**), running tests, and producing the related report (**SITR**);
- *System testing and acceptance testing*, specifying the Acceptance Test Plan (**ATP**), running tests, and producing the *in-Factory* and *on-Site Acceptance Test* report (**FAT** and **SAT**, respectively).

2.2. Software quality decision support needs

The managerial perspective is to deliver products with a target quality level under given cost and time constraints. This demands for a shift from a decision-making process using basic measurement data, yet often driven by intuition and experience in this industrial domain, to a process supported by engineering techniques and tools leveraging advanced prediction and optimization algorithms.

In the reference domain, the component-based approach yields a highly modular design (centered around CSCIs), and most of the effort in quality assurance is spent in *Verification and Validation* activities (V&V) on the right branch of the V-model. Indeed, for the considered systems, V&V costs can be as high as 50% of total cost. Rather than replacing quality tools and practices in use, with additional cost and arguable usefulness, there is the need to leverage them in defining new methods and techniques to systematically support strategic decisions in the management of software quality and productivity.

It is important to note that the engineering of a CSCI typically takes many months or even years, and involves several company plants; moreover, CSCIs may be partially outsourced to external suppliers. Hence, there is high heterogeneity of engineers' skills, coding styles, and employed tools. The general requirement for a decision support framework is to provide quality assessment, prediction and optimization algorithms so as to timely find the tradeoff among *quality, cost* and *schedule* – known as the *iron triangle* factors [4]. The success of *projects* depends heavily on the goodness and timeliness of related decisions.

2.3. Framework requirements

The three main quality assurance areas investigated are: *quality measurement and assessment; V&V decisions support; productivity* management. They should provide engineers with a clear reading of current quality and cost figures, as well as with support for planning quality assurance activities. This high-level objective is to be achieved with *minimal impact on the existing processes*.

Based on the potential impact on quality, managers demand for decision support for the following **engineering phases**:

- coding, integration, system and acceptance test;
- corrective maintenance (pre- and post-release bug fixing).

The following capabilities are required:

- **monitoring and reporting**, to control "what is happening" with the key quality and productivity indicators;
- **detective analysis**, to understand "why something happened" (e.g., change in a site's performance);
- **prediction**, to forecast "what is likely to happen" in terms of quality and productivity trends;
- **optimization**, to suggest "which actions to implement", quantifying benefits on quality/cost/time.

In a large organization, two further crucial requirements are:

- **low intrusiveness** in quality monitoring practices;
- leveraging the many quality-related **legacy tools**.

As for the former aspect, innovations need to be introduced with minimal or no changes to existing practices. Developers should not be required to perform additional tasks; this is due to cost, organizational and human-related concerns – quality measurements are usually perceived as a form of control. For code quality, for instance, companies do typically have prescriptive policies for programmers in the form of coding rules, and static analysis is periodically performed automatically for computing quality metrics without involving them.

As for the second aspect, a decision support framework needs to exploit the tools already in use for gathering quality data. **Data sources** include tools for:

- automated code analysis and review (e.g., Parasoft©);
- pre- and post-release bugs management; tools in use are spreadsheets, open source and commercial issue trackers (MantisBT,[1] Bugzilla,[2] Jira).[3]

Capabilities and units in company plants use various tools, with different terminologies and semantics for similar data. Information has to be inferred from the available data, to save past investments and to avoid additional training. The framework has also to be decoupled from tools and independent of the various data formats. **Interoperability** of tools can be pursued through adapters.

In summary, the framework has to abstract the information collected by each team and tool, requiring minimal data for feeding models. Its instantiation requires a bottom-up black-box approach, where the object of the evaluation (i.e., which quality aspects), the type of information to gather/infer (which data), the way to gather/infer them (which metrics and procedures), and the way to interpret and use results come out from the analysis of the context.

Finally, the analyses offered by the framework are required to be highly **usable**, meaning that: *(i)* different views should be provided to different users, since the interest of a project leader is not the same as the top manager; *(ii)* advanced mathematical models have to be transparent to end users.

2.4. Software quality dimensions

The dimensions that project managers usually wish to control are quality, cost and schedule. SVEVIA analyzes the sources of software quality to predict and control the effects on cost and time. While cost and schedule are quantified in terms of effort (e.g., *man-days*) and *calendar time*, respectively, quality is assessed through several metrics, as it has multiple facets. According to the ISO/IEC 25010 standard [5], quality attributes of a software product are:[4]

- *External*: properties that the final users can experience. They are related to the dynamic behavior of the software in its usage context. Examples are: reliability, usability, portability, security, performance efficiency.
- *Internal*: properties of software artifacts of interest for producers. They are static properties, and their measurement does not require software execution. Examples are: requirements size and completeness; design modularity, degree of reuse; code features like McCabe complexity, lines of code, fan-in and fan-out, degree of compliance to programming rules.

External attributes represent the desired characteristics of the product, and the process is meant to provide a high level of such quality attributes at low cost. However, they are not easy to measure. A relevant metric is the degree of defectiveness, considered a measure of (non-)quality of artifacts and of activities. It is by far the most implemented quality metric, since it is easy to measure and close to user-perceived quality. Common indicators for defectiveness are number, density and rate of defects. In addition, for process analysis purposes, defects can be characterized by a set of attributes, including source, type, injection phase, trigger, detection phase, closing time [6,7]. SVEVIA exploits defect-based metrics also to estimate and predict *reliability*, an external attribute of major concern for mission-critical systems.

Internal quality impacts defectiveness, thus external quality. Its measurement is useful not only for understanding the product quality at intermediate stages, but also for external quality prediction (e.g., in terms of expected residual defects) and consequent planning of improvement actions. Given the focus on V&V, the framework stresses the measurement of internal quality attributes of the source code, by checking compliance to company programming rules, such as *"avoid use before initialization", "avoid null pointer de-referencing"*.

In summary, SVEVIA considers as key drivers for strategic decisions about testing and product release: code complexity and compliance to coding policies; defectiveness; reliability.

[1] http://www.mantisbt.org.

[2] http://www.bugzilla.org/about.

[3] http://www.atlassian.com/software/jira.

[4] This standard introduces also "in-use" attributes; the distinction between external and in-use quality is neglected for the purposes of this work.

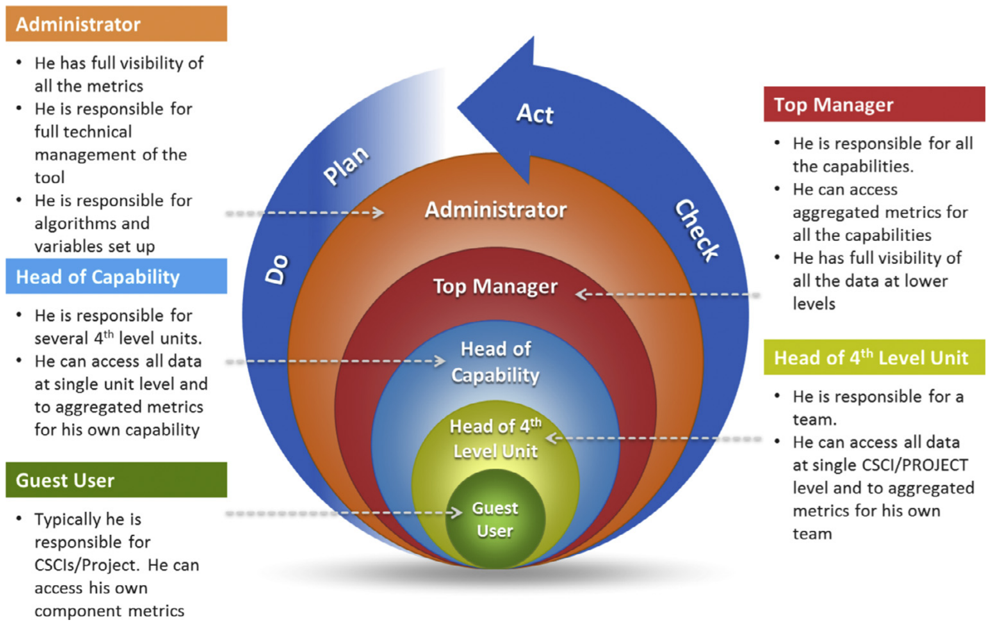

Administrator
He has full visibility of all the metrics
He is responsible for full technical management of the tool
He is responsible for algorithms and variables set up
Head of Capability
He is responsible for several 4th level units.
He can access all data at single unit level and to aggregated metrics for his own capability
Guest User
Typically he is responsible for CSCIs/Project. He can access his own component metrics
Act
Plan
Do
Check
Administrator
Top Manager
Head of Capability
Head of 4th Level Unit
Guest User
Top Manager
He is responsible for all the capabilities.
He can access aggregated metrics for all the capabilities
He has full visibility of all the data at lower levels
Head of 4th Level Unit
He is responsible for a team.
He can access all data at single CSCI/PROJECT level and to aggregated metrics for his own team

Fig. 2. Actors of the SVEVIA framework.

3. The SVEVIA framework

The main functionalities of SVEVIA are grouped in three main areas (besides Administration) regarding: software **quality assessment, decision support** and **productivity management**. These are intended to provide engineers with a clean reading of current quality, cost and time figures, as well as to support decisions for efficient planning and execution of quality assurance activities. The human roles interacting with SVEVIA are shown in Fig. 2. The framework displays customized views according to the user role. For instance, a top manager is provided with high-level indicators of productivity and quality of software assets, with the chance of navigating into single Projects or Capabilities, while the Head of a Unit with lower-level details of CSCIs.

The estimation, prediction and optimization algorithms are conceived to use a minimal amount of input information. Specifically, three sources of information are considered related to the quality-cost-time of production phases: *(i)* the software *code; (ii)* the *history of software-related defects* detected during testing or operation; *(iii)* the records of *produced artifacts and efforts spent per activity*. The history of defects is the real pivot, as it summarizes the progress of perceived (non-)quality of the final products and of software-related processes. SVEVIA is integrated – transparently to the end user – with the company existing tools which manage such information: it is able to get data from two code analysis tools, from several heterogeneous issue trackers adopted across all the company, and from the development activities recording tool.

Fig. 3 shows the high-level architecture, designed according to the Model-View-Controller (MVC) pattern. The model part retrieves data from external tools through an adaptation layer, which copes with the heterogeneity of data sources and formats. The controller has three components, *quality assessment, decision support* and *productivity management* (besides administration), which use three further components embedding the core algorithms: *estimation, prediction, optimization*. The view presents results via a web interface.

4. SVEVIA services

4.1. Quality assessment

SVEVIA focuses on the three mentioned engineering phases: **coding, testing**, and **maintenance**. Coding applies to CSCIs – the building blocks of systems. Testing concerns individual CSCIs, their integration, and final system validation. Maintenance persists beyond the system delivery, throughout its operational phase. Quality assessment targets phases, *Units* and *Projects*.

4.1.1. Code quality

CSCIs are scrutinized through static code analysis. This checks if an extensive set of rules is respected by programmers, to spot errors (such as buffer overflows and null pointer dereferences) that can escape compilers' detection and testing, and to verify compliance to industry standards like MISRA [8].

SVEVIA considers several rules for the C/C++, C# and Java programming languages, categorized as *bug detective* (BD) and *coding rules* (CR). BD are rules that, if not met, are likely to lead to software defects; CR refer to programming style guidelines. Macrogroups of rules are listed in Table 1. Rules are assigned a *severity* level from 1 to 5 (most to least critical), and are grouped by quality attribute they refer to: *Security, Reliability, Performance, Maintainability*.

Rule infringements are checked periodically by means of Automatic Static Analysis (ASA) tools. SVEVIA parses their results, and builds numerical and graphical indicators for code quality monitoring and reporting, providing feedback to team leaders and enforcing control over CSCIs. These include:

- *Basic statistics*, e.g., mean and standard deviation of BD and CR *rule violations* (a.k.a. *infringements)*, BD and CR density (number of infringements per 1K or 10K LOC);
- *Count of infringements* by CSCI, rule, severity, site (where the CSCI is produced), unit and sub-unit. These suggest: *(i)* what are the most violated rules and their severity; *(ii)* what are the differences across CSCIs, across internal/outsourced development teams, Units and plants, and which ones remarkably deviate from the average quality;
- *Temporal trend* of BD and CR density in the last 8 trimesters, checked against a target threshold, current *distance from the target* CR and BD infringements density, and *prediction* of time to achieve the target;
- *Code complexity* metrics, such as McCabe cyclomatic complexity and number of LOC, which are known to be related to code quality [9].
- *Monetary risk* associated to a CSCI, estimated as average effort to remove a rule weighted by number of violations and by rule severity. This gives a measure of **technical debt**, meant as cost needed to fix infringements (by their number, type, and average removal

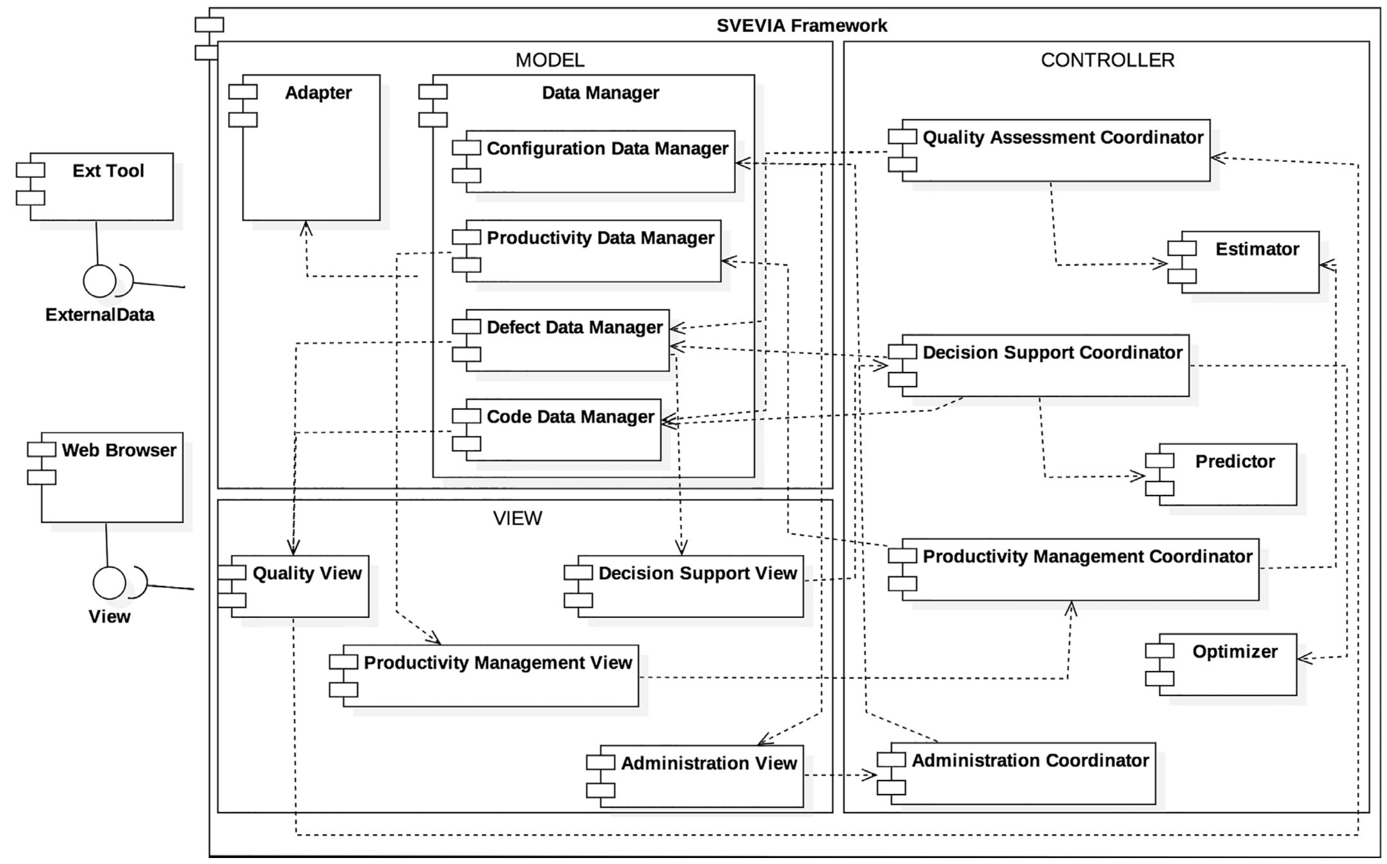


Fig. 3. Architecture of the SVEVIA framework.

cost) and to *sanitize* the code;

Examples of basic decisions supported by the analysis of current status and trend over time are:

(i)prioritize internal coding rules to enforce;
(ii)train programmers based on most violated rules;
(iii)inspect sites or units with anomalous quality patterns;
(iv)increase the effort to lower the trend of violations;
(v)reduce heterogeneity of quality across CSCIs, units, sites;
(vi)reduce complexity of CSCIs.

More complex decisions are supported by prediction and optimization algorithms; they are described later in Section 4.2.

Fig. 4 shows one SVEVIA *code quality view* for one of the capabilities of a test project – a system for Air Traffic Control (ATC).[5] Synthetic informations on violations' density are on top. The graphs indicate an improvement of two key indicators (CR and BD violations) over time (trimesters) with respect to the starting temporal reference. The navigation into sub-capability and CSCI figures allows engineers to spot problems at finer level. The view provides figures about:

- rules with highest number of violations;
- rules with highest violation density, grouped by top three severity levels;
- distance of CSCIs from the target violations;
- tree map of the CSCIs with more infringements;
- violation density, broken down by company site;
- CSCIs with highest monetary risk.

In the example, we can note that: the general trend of violations is decreasing across trimesters; reliability rules are the most critical ones, and the first six BD rules have more than 100 violations; the worst 20 CSCIs exhibit a high share of violations, and some of these are really critical (e.g., one CSCI is 900% distant from the target with a monetary risk of one order of magnitude greater than the fourth in the list).

4.1.2. *Testing process quality*

Code quality properties are essential, yet they do not have a *direct* impact on user-perceived quality. Even if unlikely, poor code quality, e.g. due to high complexity or high number of infringements, does not imply a high number of operational failures. This is because *(i)* infringements are not necessarily actual software defects, and *(ii)* defects do not always lead to operational failures [10]. *Testing* is where real failure-causing defects are discovered; if not corrected, they are likely to appear in operation. Hence, the end quality depends heavily on testing and debugging [11].

Common testing metrics include: coverage of requirements or code; percentage of tests exposing failures; number of wrong tests (failed due to mistakes); effort spent to create and execute tests. Besides these, SVEVIA offers innovative metrics to assess testing **effectiveness** and **efficiency**. Since the ultimate goal of testing is to reveal defects (not just to achieve high coverage), effectiveness refers to how much testing is able to expose defects *with respect to total defectiveness*; efficiency relates effectiveness to the effort required. However, the total defectiveness is unknown. SVEVIA provides an algorithm for estimating of total number of defects expected to be found by testing (*ExpDefect*).

The estimation algorithm is based on **software reliability growth models** (SRGMs), a widely used class of models in software reliability engineering [12]. SRGMs fit inter-failure times from test data in order to estimate the next time to failure on the basis of the observed trend. They are commonly used to estimate the reliability growth over testing time, the mean time between failures [13] or the number of expected remaining defects [14].

[5] Data are anonymized for confidentiality reasons.

Table 1
Macrogroups of coding rules.

	Macrogroup	#Rules	Example
BD-G1	Possible bugs	11	Avoid conditions that always evaluate to the same value
BD-G2	Resources	4	Ensure resources are freed
BD-G3	Security	7	Protect against integer overflow/ underflow from tainted data
BD-G4	Threads and synchr.	3	Do not abandon unreleased locks
BD-G5	Other rules[a]	13	Ensure resources are deallocated
CR-G1	Formatting rules	5	Each variable shall be declared in a separate declaration
CR-G2	Metrics rules	1	Source lines shall be kept to a length of 120 characters
CR-G3	Coding convention	6	Avoid magic numbers
CR-G4	MISRA rules	34	Use parentheses unless all operators in the expression
CR-G5	Naming convention	1	Names of parameters in declaration and definition should be identical
CR-G6	Initialization rules	3	Do not assume that members are initializ- ed in any special order in constructors
CR-G7	Optimization rules	1	Avoid inline constructors and destructors
CR-G8	Object oriented Programming rules	5	Avoid declaring virtual functions inline
CR-G9	Memory and resource	6	Declare a copy constructor for classes with dynamically allocated memory
CR-G10	Comments rules	1	Each source file shall contain an header detailing the owner and information about the version and release of the file
CR-G11	Exceptions rules	2	A class type exception shall always be caught by reference
CR-G12	Other rules[a]	67	Limit the maximum length of a line

[a] For brevity, we grouped in BD-G5 and CR-G12 other rules referring to Java, numerous yet less relevant as most of the test project code is in C/C++.

In SVEVIA they are mainly used to estimate the expected number of (residual) defects at the end of testing, the effort still needed to detect them (useful for scheduling the best release time), and for resource allocation optimization as part of the decision support function (Section 4.2). There exist many such models, varying for the shape of the data fitting function [15]. Given a set of testing data, SVEVIA fits them with a set of models, then selecting the best fitting one among a set of eight models. Specifically, the models implemented in the current version of the framework are: *exponential; S-shaped; Weibull; log logistic; log normal; truncated logistic; truncated extreme-value max; truncated extreme-value min*. Table 2 reports the models with their parameters. The SVEVIA algorithm applies all the models to the testing dataset, determines the SRGM's parameter values by means of the Expectation–Maximization (EM) algorithm developed by Okamura et *al.* [16], and then computes the Akaike information criteria (AIC) for each SRGM, selecting the one with the best AIC value [17]. The number of total expected defects as estimated by the selected SRGM (paramter '*a*' in all the cases, Table 2) is taken as *ExpDefect* metric. Based on it, effectiveness is estimated by *Test Maturity*%:

$$TM\% = \frac{Defects}{ExpDefects}\cdot 100. \tag{1}$$

TM% is the percentage of defects found over all expected defects, thus yielding the current state of testing with respect to the SRGM-estimated expectation. Note that the numerator is an exact number, while denominator is a statistical estimate; this makes the metric a statistical estimate. Two efficiency metrics are derived. The first one is the *Test efficiency*%:

$$TE\% = \frac{TM\%}{TestEffort} = \frac{DetRate}{ExpDefects}. \tag{2}$$

TE% is the percentage of achieved effectiveness relatively to the testing effort spent (*TestEffort*). Testing effort is expressed in man-days, although other choices are possible in the framework (man-hours, man-weeks, man-months). Note that this metric also represents the *defect detection rate* (*DetRate*) normalized over the expected defects. *DetRate* is defined as the number of defects detected per unit of testing effort – $DetRate = \frac{Defects}{TestEffort}$.

The second efficiency metric is for relative comparisons of CSCIs (or Projects). Indeed, *DetRate* and *TE%* indicate the actual efficiency of testing, but they do not allow a fair comparison of CSCIs, because the number of defects does not vary linearly with the testing effort (testing is usually more efficient initially, when it exposes more failures per time unit). SVEVIA introduces a 'relative' efficiency measure: given a test maturity value, it tells which testing process exhibits the best (normalized) rate. For instance, considering *TM*% = 90%, SVEVIA compares the efficiency of testing teams in achieving that level. Defining the number of effort units (man-days) to detect *x*% of total estimated defects ($TestEffort_x\%$), its normalization over the number of estimated defects is a relative (in)efficiency measure (the higher the value, the lower the efficiency):

$$RTE\% = \frac{TestEffort_x\%}{ExpDefects}. \tag{3}$$

Overall, SVEVIA provides, for each CSCIs and/or Project:

- Basic testing KPIs, including percentage of failure-exposing tests, number of wrong tests, effort spent to create and execute the test suite. They report about the goodness of the test suite observed up to current time;
- SRGMs-based metrics (e.g., expected total defects, reliability, or failure intensity) and graphs, describing the expected testing trend per CSCI/project. From these, testing *effectiveness* and *efficiency* at current/future time is estimated/predicted according to the mentioned metrics;[6]
- Reports of defects detected per CSCI/project split by categories, such as defect state, defect severity (a team may work well at finding critical defects than trivial ones), as well as a map of the most critical CSCIs containing the 80% of defects in a project.

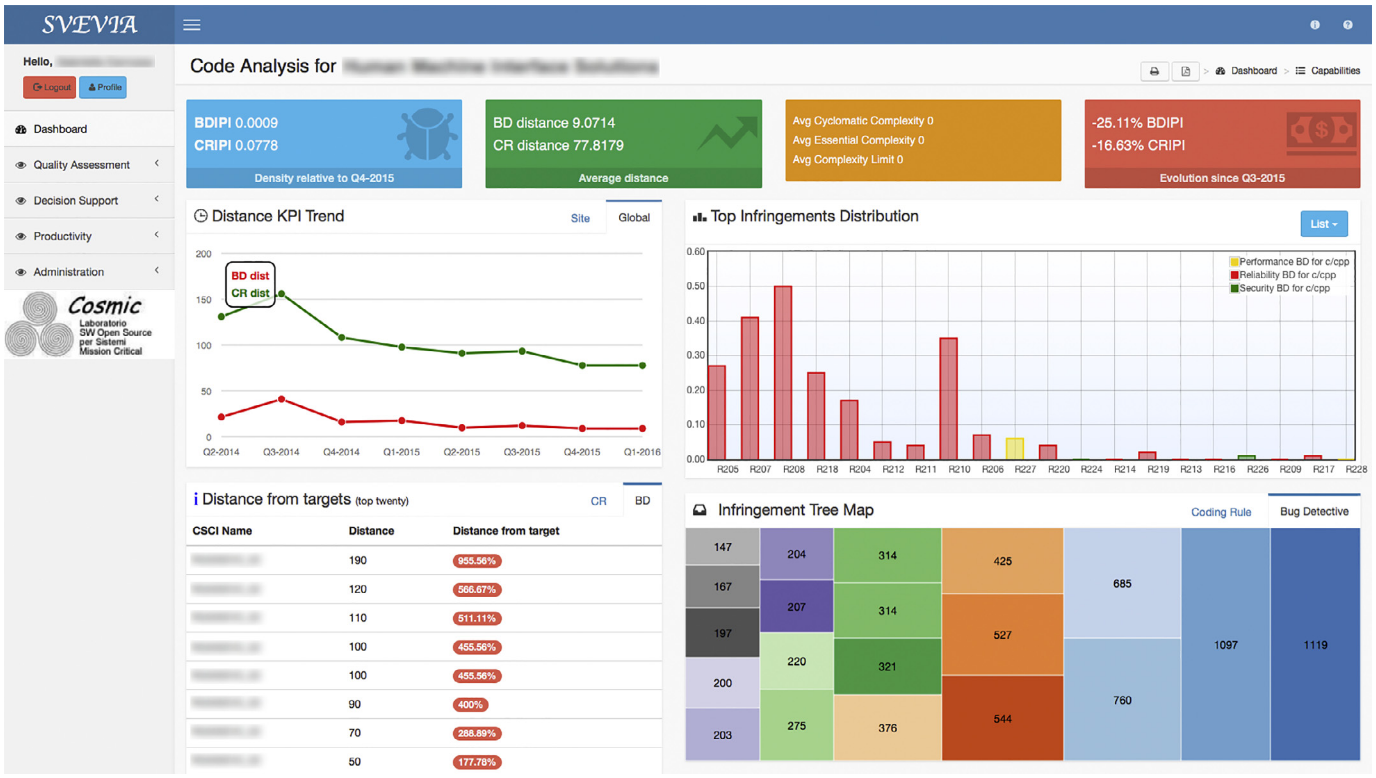
The statistics on average and range of variability across CSCIs/ Projects (computed as 95% mean confidence interval) are useful to detect discrepancies among engineering teams. This perspective allows assessing effectiveness and efficiency for single CSCIs or Projects and the variability across them, so as to spot bottlenecks or best-in-class elements.

4.1.3. Debugging process quality

For large-scale industrial systems, the management of defect correction is crucial for quality-effort-time tradeoff. Corrections should be as prompt as possible, and the way they are scheduled and performed should not give rise to bottlenecks or further imperfections. SVEVIA computes the following metrics for *effectiveness* and *(in)efficiency* assessment of the correction process [23]:

- Reduction of defectiveness (*Fixing Maturity*) achieved by the *debugging team*, as ratio of closed over total bugs:

[6] Note that basic KPIs are based on known metrics, while the defined effectiveness and efficiency are based on unknown (i.e., estimated) metrics. Being based on an estimate of total defects, they are less accurate but more informative about the goodness of testing.



Distance from targets (top twenty) CR BD

CSCI Name	Distance	Distance from target
[redacted]	190	955.56%
[redacted]	120	566.67%
[redacted]	110	511.11%
[redacted]	100	455.56%
[redacted]	100	455.56%
[redacted]	90	400%
[redacted]	70	288.89%
[redacted]	50	177.78%

Fig. 4. A code quality assessment view in SVEVIA.

Table 2
Software reliability growth models.

Model	*m(t) function*
Exponential [18]	$a\cdot(1-e^{-bt})$
S-shaped [19]	$a\cdot[1-(1+bt)e^{-bt}]$
Weibull [12]	$a\cdot(1-e^{-bt^{\kappa}})$
Log logistic [20]	$a\cdot\frac{(\lambda t)^{\kappa}}{1+(\lambda t)^{\kappa}}$
Log normal [21]*	$a\cdot\Phi\left(\frac{\log(t)-\mu}{\sigma}\right)$
Truncated logistic [22]	$a\cdot\frac{(1-e^{-t/\kappa})}{(1+e^{-(t-\lambda)/\kappa})}$
Truncated extreme-value max [17]	$a\cdot\left(1-\frac{exp\left(-e^{\frac{(t+\lambda)}{\kappa}}\right)}{exp(-e^{\lambda/\kappa})}\right)$
Truncated extreme-value min [17]	$a\cdot\left(1-\frac{1-exp\left(-e^{\frac{-(t-\lambda)}{\kappa}}\right)}{1-exp(-e^{\lambda/\kappa})}\right)$

*Φ indicates the normal distribution. a is the total # of expected defects; t is the independent variable. b and λ are rate/scale parameters; κ is the shape parameter.

$$FM=\frac{ClosedDefects}{TotalDefects};\tag{4}$$

- Mean and median time to repair defects ($MTTR_{\mu}$, $MTTR_{Mdn}$), as measure of temporal (in)efficiency. The median is relevant because the distribution of times to repair a defect is usually non-normal [24]. MTTR conveys direct information on the debugging time. The reciprocal are measures of actual fixing time efficiency (*FtE*):

$$FtE=\frac{1}{MTTR};\tag{5}$$

- Mean cost to repair defects (MCTR) – based on hourly cost of debuggers, varying among company sites – and, consequently, cost-efficiency to repair defects (*FcE*):

$$FcE=\frac{1}{MCTR}.\tag{6}$$

In addition, the framework supports assessment of the *internal quality* of the debugging process, in terms of these properties:

- *continuity* of fixing actions over time;
- *homogeneity* of fixing actions across defects;
- distribution over scales of *priority* and *severity*;
- percentage of *re-openings*.

As for continuity and homogeneity, SVEVIA provides graphs of *closed vs opened* defects trend over time, and the empirical distribution of debugging times (TTR). The graphs show if the correction follows the detection (basic statistics – *min, max, mean, std* – are computed on the closed-opened differences over time to quantify the continuity). The TTR distribution emphasizes the variability of repair times: the debugging process is homogeneous when most defects have a TTR close to the average, and most of the TTR variance is due to many defects with short TTR. These features are quantified by two well-known distribution metrics: *kurtosis*[7] and *skewness*.[8] The breakdown by *priority* and *severity* shows whether the MTTR is consistent with defect priorities and categories set by testers. Inconsistencies raise warnings about proper scheduling of corrections (which bugs are removed first). Finally, the number of times a bug gets re-opened indicates the rework it

[7] A high kurtosis is good, meaning that variance is mostly due to few peaks.
[8] A positive skew denotes that peaks are in the left side of the distribution, i.e., many defects have a short TTR.

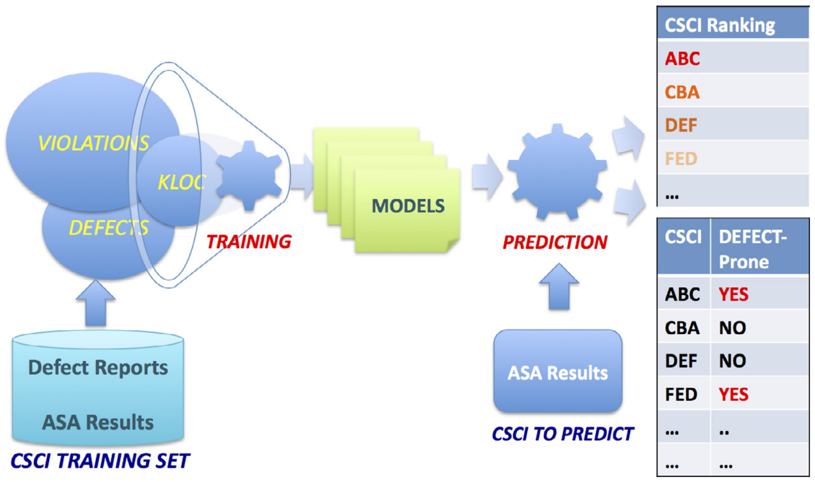


Fig. 5. Defect prediction process in SVEVIA.

underwent. This evidences the amount of non-perfect fixes, which can increase the likelihood of introducing regression bugs. Values of all the above metrics are provided also per CSCI and Project, to spot differences among them.

4.2. Decision support

Product and process metrics are a valuable basis for decision-making. They support the judgment by engineers who can quantify trade-offs between quality, cost and time for coding, testing and debugging. Some planning decisions, however, demand for more advanced support. The framework implements sophisticated prediction and optimization algorithms, in a user transparent way. They concern test prioritization, code sanitization and test optimization.

4.2.1. Inspection/CSCI test prioritization

Static analysis results are used to provide feedback to drive code inspection/testing. A correlation has been demonstrated in the literature between infringements of coding rules and defects [25,26]. SVEVIA exploits infringements data to predict CSCIs defectiveness. The prediction process is shown in Fig. 5. The input is the number of rule infringements of a CSCI. The output is two-fold: *(i)* a binary indication of whether the CSCI is defect-prone or not; *(ii)* a relative ranking of CSCIs based on defect-proneness. The binary information is provided with higher confidence, but it is less accurate and useful. Ranking quantifies the relative degree of defectiveness of CSCIs. Managers can prioritize the most critical components for code inspection and unit testing, before delivering them to the next stages.

Several combinations of algorithms and metrics are tried during training, in order to identify the model with the best prediction power. Currently, six algorithms are used (*Decision Trees, Bayesian Network, Naive Bayes, Logistic Regression* for binary classification and *Linear Regression, Support Vector Machine* for ranking), and two metrics (number of violations and defects, violations/defects density per KLOC). In the binary classification scheme, three criteria are tried to label a CSCI as defective:

- a CSCI is considered defective if the density is greater than the normal statistical upper cut-off (UCO);[9]
- if the density is higher than the 90th percentile of the defect density distribution over CSCIs used for training (useful due to the strong non-normality of distribution);
- if the density is higher than the 3rd quartile of the defect density distribution over CSCIs used for training.

Following the scheme in Fig. 5, the steps are:

1. Models are trained with samples of data about CSCIs produced in the past. Training is run by a well-known method, i.e., through 10-fold cross-validation, repeated 100 times per classifier.
2. Performance of each run of cross-validation is assessed, for binary classifiers, by means of the *Balance*[10] indicator, commonly adopted for defect prediction [27–30] wherein the dataset is unbalanced [31,32]. For ranking models, the Fault Percentile Average (FPA) is used.[11]The algorithm with the best *Balance*/FPA (for binary classification or ranking, respectively) is selected as predictor;
3. The algorithm selected and trained with available data – where defect (density) is known – is used for prediction for CSCIs with yet unknown defect (density). The output is the list of CSCIs deemed as dangerous (either with a binary label – defect-prone or not – or by a ranking).
4. The most impacting rules are selected. Attributes are ranked by their contribution to the gain, using the *Information Gain* algorithm [34].

The output of this framework service are:

- the *list of risky CSCIs*; developer should focus upon them for improving quality (both quantitatively, in terms of relative effort spent, and qualitatively, as more complex techniques should be used for risky CSCIs);
- the *list of highest prediction-impacting rules*. This serves e.g. for assessing programmers' training needs, and is used by the next algorithm for code sanitization optimization.

4.2.2. Code sanitization

With the huge amount of LoC and number of CSCIs involved, it is important to minimize the effort spent in code sanitization (correction of coding rule infringements) to attain a target objective. SVEVIA

[9] $UCO = \mu + [(z_{\alpha/2}]*\sigma/\sqrt{(N)}$, where μ is the mean defect density of CSCIs, σ its standard deviation, N the number of CSCIs, $z_{\alpha/2}$ the upper $\alpha/2$-quantile of the standard normal distribution, α the desired significance level ($\alpha = 0.05$ in our case) [26].

[10] Balance computation is based on *true/false positives* (TPs/FPs), and *true/false negatives* (TNs/FNs) (i.e., CSCIs are TPs if they are correctly classified, FNs otherwise; non-defective CSCIs are TNs if correctly classified, FPs otherwise), as: $Bal = 100 - \frac{\sqrt{(0-PF)^2 + (100-PD)^2}}{\sqrt{2}}$, where PD (*Probability of Detection*) is $\frac{TP}{TP+FN}\cdot 100\%$, PF (*Probability of False Alarms*) is $\frac{FP}{TN+FP}\cdot 100\%$. It represents the trade-off between a high PD and a low PF, being based on the Euclidean distance from the ideal objective PD = 100% and PF = 0%.

[11] Given k modules, it is the average of the proportions of actual defects in the top m ($m = 1, ..., k$) modules to the whole defects [33].

implements three variants of an optimization model targeting trade-off between the desired quality goal in terms of rules infringements and cost required to attain that quality. The user specifies the quality target in three possible ways (for BD and CR rules):

- **Target Average** (**T-A**): the mean number of violations across CSCIs has to be under a specified target $\bar{v}_{max}$;
- **Target Average and Standard Deviation** (**T-A & T-STD**): this requires the standard deviation to be under a user-specified target std_{max};
- **Target Number** (**T-N**): the number *CN* of CSCIs which have to have less violations than a threshold v_{max}.

The algorithm computes: *(i)* the minimal effort (man-hours) to attain the target; *(ii)* how such effort should be allocated to each CSCI and *(iii)* to each type of rules among the violated ones, based on their severity and expected mean removal effort. The solution focuses on CSCI and rules more likely leading to a reduction of cost (both actual cost and monetary risk).

Let v_i denote the total number of violations of type i, and $v_{i,\ j}$ the number of violation of type i in the jth CSCI. A violation not removed can result in a software defect; on the other hand, its removal has a cost. The violation removal effort at coding stage is lower than the effort to remove a defect in later stages. We assume each type of violation (e.g., a rule) being characterized by a pair $< RE_i, w_i >$, where:

- RE_i (*Removal Effort*) is the effort (e.g., in man-hours) per unit needed to remove one violation of the ith type;
- w_i is a [0,1] weight rating the type of violation i with respect to the expected impact that its non-removal has on the system quality.

RE_i values are established by querying company historical data about the removal of violations of the ith type. Weights are derived by one of two ways: they may be assigned by experts judgment (company engineers), who gives more importance to specific types of violation deemed more critical (based on rules severity); alternatively, the list of rules more correlated to defects (an output of the previous prediction algorithm) is used.[12] Weights represent the likelihood that a violation of a given type results in a defect. Let us denote with E_d the effort per unit to remove a defect in the system/acceptance testing phase; this is estimated by querying the repository tracking the man-hours devoted to defect removal activities; clearly, $E_d >> RE_i$. Then, the *Non-Removal Effort* (*NRE*) is defined as the expected effort per unit incurred if the violation of type i is not removed: it is assessed as $NRE_i = w_i{*}E_d$. *NRE* is a measure of the *monetary risk* incurred by leaving the violation in the code, given by the unitary effort for a defect removal weighted by the impact of violation i.

It would be extremely expensive to eliminate all violations from all CSCIs. The algorithms find the optimal trade-off between the number and the type of violations to remove, and the risk of not removing them. Formally, denoting by $x_{i,\ j}$ the number of violations of the ith type that the algorithm proposes to eliminate from CSCI j, the cost for CSCI j of removing $\Sigma_i x_{i,\ j}$ violations plus the expected cost of not removing them (the residual monetary risk) is:

$$C_j = \sum_{i=1}^{m} x_{i,j}\cdot RE_i + \sum_{i=1}^{m} [(v_{i,j} - x_{i,j})\cdot NRE_i]. \tag{7}$$

Thus, the basic optimization model is:

$$\text{Min! } C = \sum_{j=1}^{n} C_j = \sum_{j}^{n} [\sum_{i=1}^{m} (x_{i,j}{*}RE_i) + \sum_{i=1}^{m} [(v_{i,j} - x_{i,j}){*}(w_i{*}E_d)]] \tag{8}$$

[12] Based on the average rank obtained over a 10-fold cross-validation, rules are rated by their importance; weights are taken as the min–max normalization of the average rank value (i.e., $w_i = [r_i - min(r_i)]/[max(r_i) - min(r_i)]$ with r_i being the average rank value.

subject to:

where $j = 1...n$ are the CSCIs, $x_{i,\ j}$ the decision variables, and STD denotes the standard deviation. The model is solved by a sequential trustregion algorithm with a linear approximations approach [35]. The solution of the model provides a bi-dimensional matrix reporting the amount of violations engineers have to remove for each type (column) and for each CSCI (row), as well as the *estimated cost of violation removal* (i.e., the first addend of the objective function) and the associated *"residual" monetary risk*, namely expected technical debt, after removal (second addend). These are used to meet the desired quality objectives at minimal cost. It can happen that no solution is available to meet two constraints together in the T-A & T-STD model (i.e., average and standard deviation together); in such a case, the framework relaxes the standard deviation constraint incrementally, warning the user of this choice, until the target on the average is met. Fig. 6 shows an optimization plan for CSCIs of a test project. The minimum target set for CR and BD violation density is 50 and 10, respectively, and referring just to the top 20 rules; the initial situation is 93.03 and 15.31 violation density in the two cases. The algorithm suggests allocating a minimum budget of 890 h to reach the target, which would yield a final value of 48.17 and 8.25 for CR and BD violations (under both targets) over the top 20 rules. It also provides a matrix, in the bottom part of the view, with the list of violations to remove for each rule and for each CSCI in order to achieve the target with that budget.

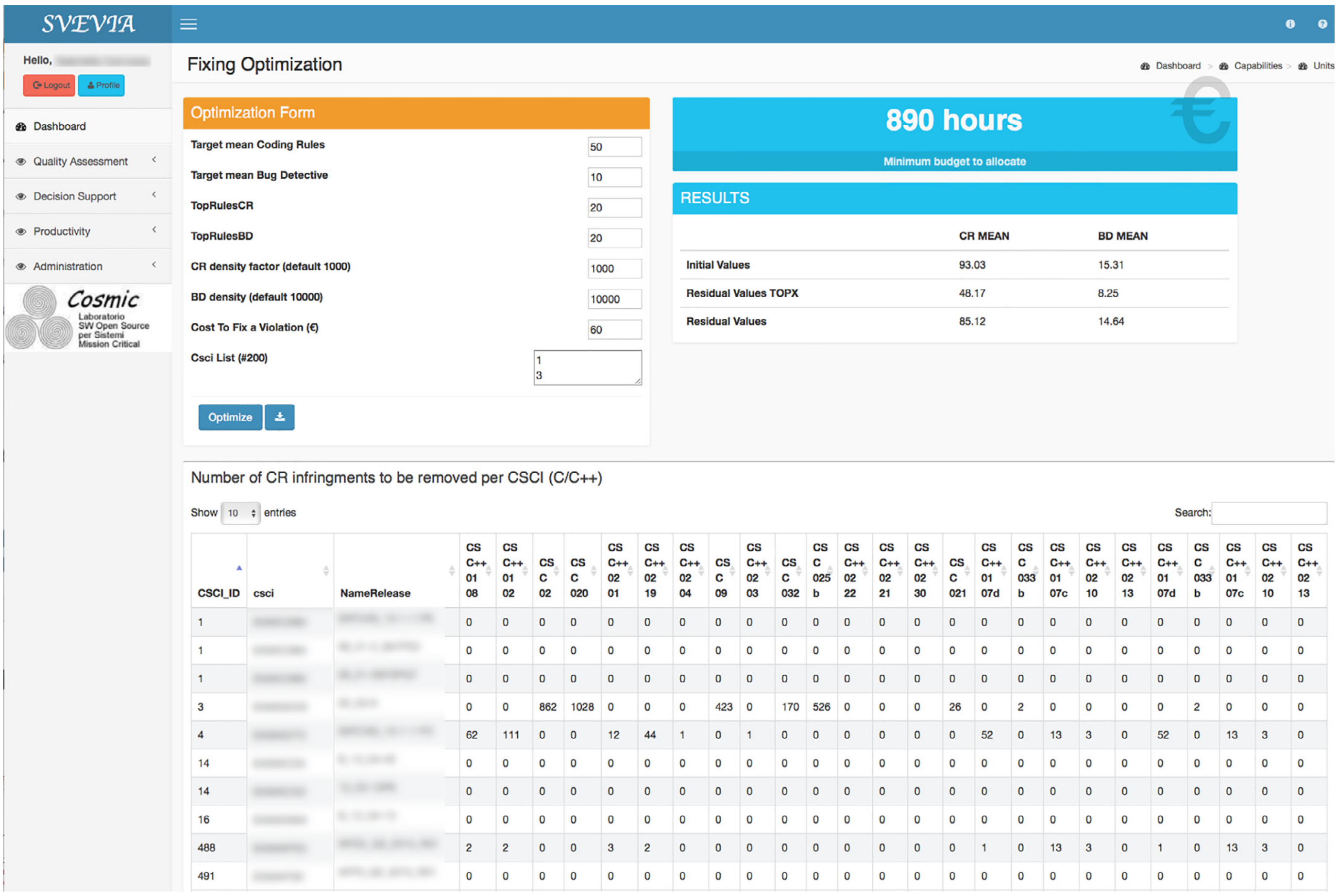
4.2.3. Test optimization

Optimization-based planning algorithms for efficiently allocating resources are a powerful means for decision support [36]. SVEVIA exploits optimization models for the testing resource allocation problem. Specifically, it supports strategic decisions on how to effectively distribute the testing resources available to each CSCI's team. The optimization tool suggests the test effort distribution under defined objectives and constraints. It builds analytical models of testing trends of each CSCI by means of SRGMs; these are then used to get the expectations of detectable defects, and the time and effort that will be spent. The relation between testing time t and testing effort (cost) W is expressed by a testing effort functions (TEF) embedded in the SRGM [37,38], allowing an accurate characterization of testing time-cost-quality relation. SVEVIA implements flexible optimization models, allowing to specify various objectives and constraints. The general form of the multi-objective implemented models is:

$$\begin{aligned}
&min! \;\; E\,[ResidualDefects] = \textstyle\sum_{i=1}^{N} (EST_i - m_i(W_i^* + W_i))\\
&min! \;\; E\,[TestingTime] = \min_{i=1\cdots N} (t_i)\\
&min! \;\; E\,[Cost] = \textstyle\sum_{i=1}^{N} C_1\cdot m_i(W_i^* + W_i) + C_2\cdot(EST_i - m_i(W_i^* + W_i))\\
&\qquad\qquad + C_3 m_i(W_i^* + W_i)dt\\
&s.\ t. \textstyle\sum W_i \le B^*
\end{aligned} \tag{9}$$

where: N is the number of CSCIs; EST_i is the number of expected defects in the ith CSCI estimated by SRGMs; W_i is the test effort to allocate to the ith component; W_i^* is the effort already spent on the ith CSCI; $m_i(W_i^* + W_i)$ is the number of defects that would be removed if component i receives an effort of $(W_i^* + W_i)$ – it is the *mean value function* (*mvf*) of the SRGM selected for the ith CSCI; t_i is the testing time; C_1 is the cost to correct a bug during testing; C_2 is the cost of correcting a residual bug in operation (typically $C_2 > C_1$ [39]); C_3 is the cost testing per effort unit *W; B** is the residual budget. Cost parameters C_1, C_2, C_3 are estimated based on historical data.

SVEVIA implements the *mvf* of the eight SRGMs listed in Table 2. If the user assumes a testing time t varying linearly with testing effort W, then no TEF is needed and the *testing effort* allocation corresponds to a *testing time* allocation: $m_i(W_i^* + W_i) = m_i(t_i^* + t_i)$. Alternatively, the framework implements a TEF. The TEF chosen is the most common one,



Fixing Optimization

Optimization Form

Target mean Coding Rules	50
Target mean Bug Detective	10
TopRulesCR	20
TopRulesBD	20
CR density factor (default 1000)	1000
BD density (default 10000)	10000
Cost To Fix a Violation (€)	60
Csci List (#200)	1 3

890 hours

Minimum budget to allocate

RESULTS

	CR MEAN	BD MEAN
Initial Values	93.03	15.31
Residual Values TOPX	48.17	8.25
Residual Values	85.12	14.64

Number of CR infringments to be removed per CSCI (C/C++)

CSCI_ID	csci	NameRelease	CS C++ 01 08	CS C++ 01 02	CS C 02	CS C 020	CS C++ 02 01	CS C++ 02 19	CS C++ 02 04	CS C 09	CS C++ 02 03	CS C 032	CS C 025 b	CS C++ 02 22	CS C++ 02 21	CS C++ 02 30	CS C 021	CS C++ 01 07d	CS C 033 b	CS C++ 01 07c	CS C++ 02 10	CS C++ 02 13	CS C++ 01 07d	CS C 033 b	CS C++ 01 07c	CS C++ 02 10	CS C++ 02 13
1	[redacted]	[redacted]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	[redacted]	[redacted]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	[redacted]	[redacted]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	[redacted]	[redacted]	0	0	862	1028	0	0	0	423	0	170	526	0	0	0	26	0	2	0	0	0	0	2	0	0	0
4	[redacted]	[redacted]	62	111	0	0	12	44	1	0	1	0	0	0	0	0	0	52	0	13	3	0	52	0	13	3	0
14	[redacted]	[redacted]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	[redacted]	[redacted]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	[redacted]	[redacted]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
488	[redacted]	[redacted]	2	2	0	0	3	2	0	0	0	0	0	0	0	0	0	1	0	13	3	0	1	0	13	3	0
491	[redacted]	[redacted]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 6. Code sanitization view for a test project.

that is the logistic TEF [37,38,40]. It represents the usual trend of testing effort by this equation:

$$W(t) = \frac{\mathscr{B}}{\sqrt[h]{1 + \mathscr{A} exp[-\alpha ht]}} \tag{10}$$

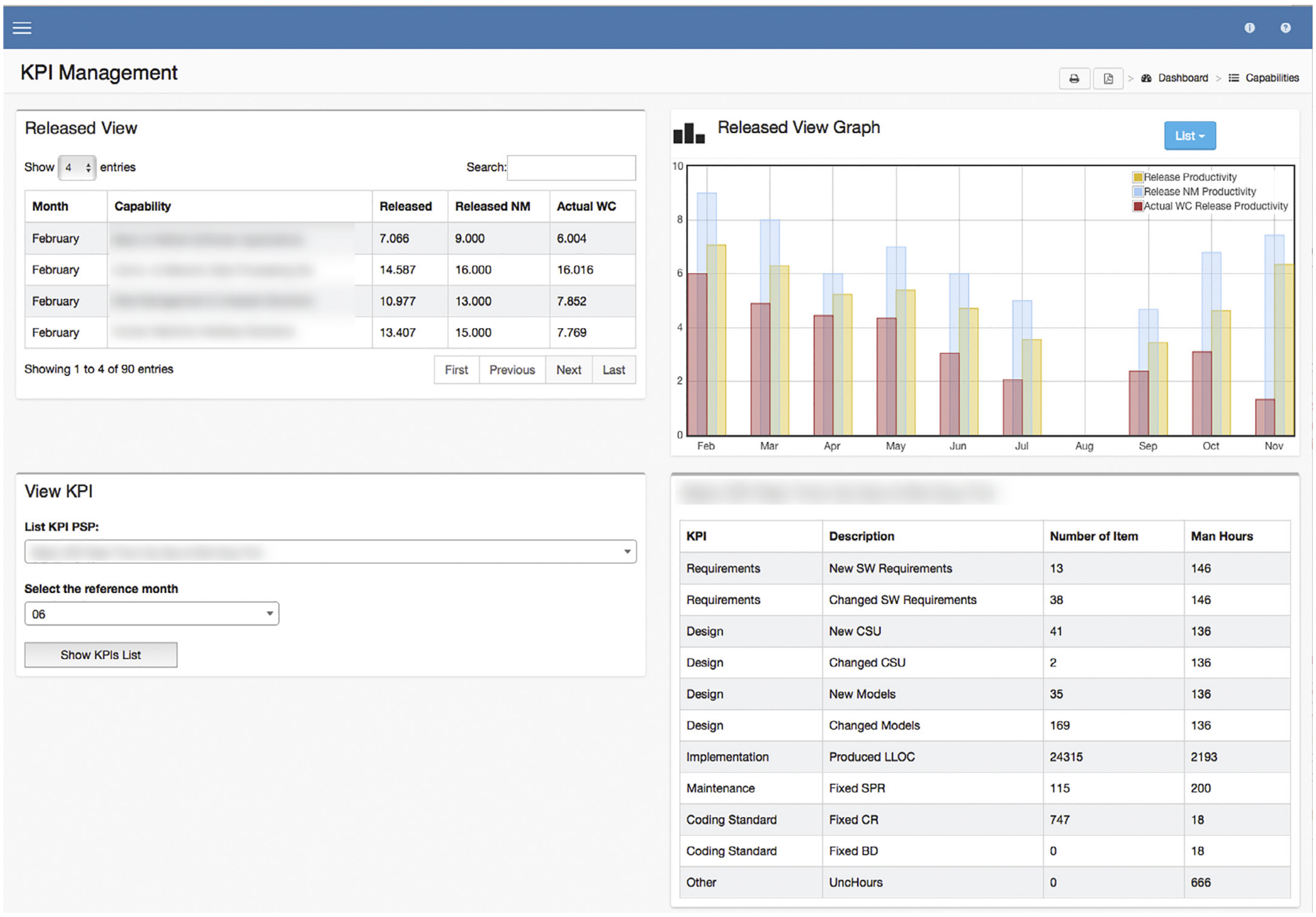
where $\mathscr{B}$ is the total amount of testing effort to be consumed; α is the consumption rate of testing-effort expenditures; $\mathscr{A}$ is a constant; h is a structuring index (a large value models well-structured software development processes); and $W(t)$ is the testing effort consumption at time t. The latter is used in lieu of testing time t in the *mvf* expressions in Table 2 to implement TEF-aware SRGMs. In such a case, testing time is obtained by the inverse of the TEF:

$$t = \left(-\frac{1}{\alpha * h} \cdot ln \left(\frac{\left(\frac{\mathscr{B}}{W}\right)^h - 1}{\mathscr{A}} \right) \right). \tag{11}$$

The model can be specialized for various goals. Managers may wish to compute *the effort allocation to CSCIs which maximizes the delivered quality under a given budget*. In this case, just the former objective function is considered [41] and the effort is a constraint. Similarly, in order to *determine the allocation which minimizes the time (or cost) for achieving a target quality level*, the quality objective is changed in a constraint, and the second (or third) objective function is considered [42]. More in general, multi-objective approaches consider a set of solutions optimizing trade-offs between testing cost, achieved quality in terms of expected residual defects, and testing time [43] and solutions are obtained by a well-known Multi-objective Evolutionary Algorithm (MOEA), NSGA-II [44], as default. Three further MOEAs are implemented: IBEA [45], MOCELL [46], PAES [47]. The provided output is the list of Pareto front solutions with *Hypervolume, Inverted Generational Distance* (IGD) and *Spread* as quality indicators The test manager can select a unique solution from the set of Pareto front solutions by a loss function depending on the importance given to these objectives – i.e., the solution minimizing the loss function is provided. Importance can be expressed by the user by means of [0,1] weights given to the (normalized) objectives. Denoting the three values of a given allocation solution $\mathbf{X}$ as $\mathbf{Y}(\mathbf{X}) = \{y_{1,\ x}, y_{2,\ x}, y_{3,\ x}\}$ for the three objective functions, we normalize them in [0,1] over the entire Pareto front: $y'_{i,x} = \frac{y_{i,x} - min_x(y_{i,x})}{max_x(y_{i,x}) - min_x(y_{i,x})}$, with $i = 1, 2, 3$. The chosen solution $\mathbf{X}^*$ is the one with the minimum *loss function* value: $L(\mathbf{Y}'(\mathbf{X})) = \sum_{i=1}^{3} w_i{\cdot}y'_{i,x}$, with w_i being the weights assigned to each objective. For instance, a balanced tradeoff is obtained with 0.33 given to all three objectives. The framework is extensible by adding models reflecting further needs about constraints and objectives, and the uncertainty of input parameters [48].

4.3. Productivity

Productivity is measured by a set of KPIs about the items produced by the teams, adjusted by a quality factor. This gives rise to what we call **quality-aware productivity**. KPIs are basically given by *produced items over employed effort*. Items cover each phase, including: number of new and changed requirements; new and changed CSU; new and changed models; produced LLOC; new and changed test cases; new and changed test runs; closed SPR (software problem reports); fixed CR and BD. The adjustment is required because the plain productivity accounts only for how much has been produced, without any regard to quality. For instance, spending 10 man-weeks for one KLoC containing 5 defects



Released View

Show 4 entries Search:

Month	Capability	Released	Released NM	Actual WC
February	[redacted]	7.066	9.000	6.004
February	[redacted]	14.587	16.000	16.016
February	[redacted]	10.977	13.000	7.852
February	[redacted]	13.407	15.000	7.769

Showing 1 to 4 of 90 entries

View KPI

List KPI PSP: [redacted]

Select the reference month: 06

[redacted]

KPI	Description	Number of Item	Man Hours
Requirements	New SW Requirements	13	146
Requirements	Changed SW Requirements	38	146
Design	New CSU	41	136
Design	Changed CSU	2	136
Design	New Models	35	136
Design	Changed Models	169	136
Implementation	Produced LLOC	24315	2193
Maintenance	Fixed SPR	115	200
Coding Standard	Fixed CR	747	18
Coding Standard	Fixed BD	0	18
Other	UncHours	0	666

Fig. 7. Productivity assessment view for a test project.

is not the same as 10 man-weeks/KLoC with 8 defects.

For coding, we consider the produced LLOC over effort adjusted by a quality factor varying between 0 and 1:

$$QA = \frac{LLOC}{Effort} \cdot \frac{1}{ED + 1} \tag{12}$$

where *ED* is the estimated defect density (*ExpDefect*, obtained by the SRGM, over *LLOC*). The metric penalizes the productivity as defect density increases, ideally achieving zero under infinite defects, while we preserve the plain productivity under an ideally perfect (zero-defects) product. The higher the value, the higher the actual productivity is, accounting for both the raw productivity and the quality of what produced. Values of both raw and quality-aware productivity are monitored, for each CSCI team, unit or capability in the organization.

Fig. 7 shows an example for unit involved in a test project for ATC. In that case, the productivity decreases up to July, and then starts increasing in September, and so does the quality-aware productivity (red bar in the graph). However, in the last month (November), the plain productivity increases while the quality-aware productivity decreases with respect to the previous month, denoting that more LLOC are being produced but with relatively more defects inside. This raises a warning on the unit being monitored. Details about each KPI at the bottom of the view can help further spotting the problem.

5. On-field experience

In 2013 the industrial partner started a process for quality assessment based on static analysis, comprising the following steps: *(i)* definition of coding standards policies; *(ii)* ASA tools scouting and selection; *(iii)* infrastructure setup, tools deployment and staff training; *(iv)* analysis running on regular basis, quarterly and at product releases.

In this process, SVEVIA provided an important contribution through both quality assessment and decision support models. The first step was to exploit ASA data for quality assessment (through new metrics such as monetary risk) and efficient code improvement (through the optimal code sanitization). Then, the defect management process was addressed, wherein SVEVIA coped with the high heterogeneity of data gathered (by means of the capability to import data from the many bug trackers in use). Defect data were exploited for quality assessment of coding (from the expected defectiveness point of view), of testing and of bug fixing processes, as well as for the planning of optimal release time, and for the optimal allocation of testing efforts. Finally, the focus was shifted on the definition of a plan for exploiting static analysis and defect data together, so as to *(i)* use violations and historical defect data to predict future defectiveness of CSCIs, and *(ii)* to use of violations and defects to "weigh" the actual productivity (i.e., quality-adjusted productivity).

All the introduced models were validated during the project on selected CSCIs and included in the framework. After validation, the framework entered regular use and it was gradually extended to the whole software division, accounting, at the end of the project, for about 900 CSCIs, with benefits on quality (e.g., in terms of reduction of violations) and productivity. We hereafter first report experimental results obtained on the three decision support models, which were recognized as a key aid by both company managers and developers (Sections 5.2 and 5.3). Then, high-level achievements across the 2013–2015 period are summarized (Section 5.4).

Table 3
Balance comparison among classifiers for BD. Legend: UCO = upper cut-off, DEN-UCO = , 90 = 90th percentile, DEN-90 = , Q = 3rd quartile, DEN-Q =.

Algorithm	UCO	DEN-UCO	90	DEN-90	Q	DEN-Q
DT	29.10	29.23	**75.83**	29.29	**89.40**	29.00
BNet	29.24	29.29	29.24	29.29	**86.04**	29.29
Log	29.24	39.78	**75.09**	52.14	**69.50**	44.73
NB	28.85	18.15	**52.14**	44.15	48.60	43.33

5.1. Prediction model

To test the prediction ability of defective CSCIs (i.e., the model described in Section 4.2.1), we considered violations and pre-release defect reports observed on a subset of 29 CSCI on the last 3 months of 2013 and the first 3 months of 2014. The model trained by these data is then used to predict defectiveness on a set of 156 CSCI (11.6 Millions of LoC) and 167 coding rules as predictors. All the six configurations were used, differing in the metrics used (absolute numbers or density) and in the criterion to label defect data (by UCO, 90th percentile, 3rd quartile), thus obtaining six datasets. On these datasets, we first compared the classifiers (*Decision Trees, Bayesian Networks, Logistic Regression and Naive Bayes*) through 10-fold cross-validation repeated 100 times per classifier, and measured performance by the balance metric. Results are in Tables 3 and 4, for the model built with the BD rules and the CR rules, respectively. The tables highlight in boldface the columns in which the classifier in the corresponding row provided the statistically (i.e., with 95% of confidence) highest values. The *Nemenyi* test is used for multiple classifiers statistical comparison [49]. Underlined there is the highest among these, denoting the best classifier/dataset pair.

Decision Tree and *Bayes Net* classifiers performed remarkably better than *Logisitc* and *Naive Bayes*. Among these combinations, we selected, for the BD model, the *Decision Tree* on the *quartile* dataset (i.e., when the third quartile criterion is used for establishing defectiveness), which has: $PD = 85.70\%$, $PF = 4.50\%$, $Bal = 89.40\%$; whereas, for the CR model, we selected the *Decision Tree* on the 90th percentile dataset, having: $PD = 100.00\%$, $PF = 3.80\%$, $Bal = 97.31\%$. *Information Gain* attribute ranking algorithm was then applied. The rules turned out to be more important for the prediction point of view are, for the BD model: *(i) avoid dereferencing before checking for null*); *(ii) Avoid overflow due to reading a not zero terminated string*, and *(iii) avoid conditions that always evaluate to the same value*. In the CR model, they are: *(i) each variable shall be declared in a separate declaration statement; (ii) Multiple variable declarations shall not be allowed on the same line*; and *(iii) Source lines shall be kept to a length of 120 characters or less*.

The selected models was then applied to the dataset of 156 CSCIs. The BD model predicted 18 CSCI (i.e., *11.54%*) as being defective; the CR model predicted 16 CSCI (i.e., *10.25%*) as being defective; 7 CSCIs were predicted to be defective by both models. Testing of all the CSCIs was then monitored (by looking at the defects in the issue tracking systems) and, after one year from the prediction, these 7 CSCIs turned out to be among the first 8 most defective CSCIs. This result allowed managers having confidence in the prediction models to be used for prioritizing test activities. As consequence, the models were improved and integrated in the framework; currently they are trained on a set of 763 CSCIs and their training can be repeated periodically by means of a back-end SVEVIA functionality. On the same dataset, we then also trained the two ranking algorithms (*Linear Regression, Support Vector Machine for regression*) and obtained a model for ranking whose performance is currently measured by an FPA = 87%, meaning that, given k CSCIs, the average of the proportions of actual defects in the top m ($m = 1, \ldots, k$) CSCIs to the whole defects is 0.87.

Table 4
Balance comparison among classifiers for CR. Legend: UCO = upper cut-off, DEN-UCO = , 90 = 90th percentile, DEN-90 = , Q = 3rd quartile, DEN-Q =.

Algorithm	UCO	DEN-UCO	90	DEN-90	Q	DEN-Q
DT	29.28	45.00	**97.31**	28.82	**89.39**	64.04
BNet	29.28	29.28	**76.30**	29.28	**86.05**	45.94
Log	29.28	41.55	51.59	49.14	**68.99**	34.16
NB	29.24	27.77	**75.83**	27.99	48.60	52.11

Table 5
Results of the allocation for BD and CR rules compared to the random solution. Legend: A: % Gain on total # of violations to remove; B: percentage reduction of **total cost**; C: percentage reduction of avg. number of violations (*reduction w.r.t. initial average*); D: reduction of the standard deviation of # violations (*reduction w.r.t. initial standard deviation*); E: number (and %) of CSCI with $v < v_{max}$.

Model	A	B	C	D	E
T-A	+16.1%	−32.9%	−14.9% (*−55.8%*)	−33.1% (*−57.6%*)	**122** *(78.2%)*
T-A & T-STD	+16.1%	−29.1%	−14.9% (*−55.8%*)	−59.6% (*−75.4%*)	**110** *(70.5%)*
T-N	+16.8%	−25.1%	−15.6% (*−56.1%*)	−57.6% (*−73.1%*)	**110** *(70.5%)*
(a) Model applied to BD violations					
Model	**A**	**B**	**C**	**D**	**E**
T-A	+78.4%	−42.2%	−15.9% (*−30.2%*)	−14.5% (*−28.8%*)	**117** *(75.0%)*
T-A & T-STD	+22.0%	−2.8%	−4.5% (*−20.6%*)	−21.6% (*−34.7%*)	**110** *(70.5%)*
T-N	+22.6%	−2.3%	−4.6% (*−20.7%*)	−22.8% (*−35.8%*)	**110** *(70.5%)*
(b) Model applied to CR violations					

5.2. Optimal code sanitization

This section reports the results of applying static analysis and optimal code sanitisation (i.e., the model described in Section 4.2.2) to the same set of 156 CSCIs and 167 rules as above. For validation purposes, we compared the three model variants against each other and against a *Random* (**R**) strategy. The latter roughly reflected the state of the practice, where each CSCI manager independently decided how many violations to remove for its CSCI, arbitrarily choosing the type of violation. We forced the random strategy to consume all the available budget so as to have a fair comparison with the other strategies. Since the solution provided by the random allocation may be different from run to run, the random strategy is repeated 50 times and taking the average solution in terms of total cost, so as to have statistically meaningful results.

Results are on a budget of maximum 800 man-hours for BD and 800 man-hours for CR treatments and the following quality objectives for the three models: *(i)* average number of violations ($\bar{v}_{max}$) reduced by at least 40% for BD rules (25% for CR rules); *(ii)* standard deviation std_{max} reduced by at least 75% along with an average reduced of at least 30% for BD rules (std_{max} reduced by 30% and average reduction by 20% for CR rules); *(iii)* at least the 70% of CSCIs with a number of violations less than 50% of the pre-optimization average for BD rules (25% for CR rules). Given the same budget, the comparison is on the following relative metrics with respect to the random case: the percentage gain of number of violations suggested to remove; the percentage reduction of the total estimated cost (TOT_{Cost}, namely, the sum of cost of removal and of non-removal, as described by the objective function); the percentage reduction of the average number of violations across CSCIs after applying the solution; the percentage reduction of the standard deviation of the number of violations across CSCIs; the number of CSCI with $\Sigma_i v_{i,\,j} < v_{max}$.

Table 5 reports the percentage gain or reduction with respect to random case,[13] and the reduction achieved with respect to the initial average and standard deviation, indicating to what extent targets have been met. They show that:

- the T-A model meets its target on the average number of violations; the number of violations suggested to remove is 16% higher than the random solution (in the BD case), and it is impressive in the CR case where a +78% is achieved. The total cost is reduced of 32% and 42% in the two cases. This solution allows removing much more violations than the other cases, and also the most cost-impacting ones;
- the T-A & T-STD model achieves its targets on average and standard deviation. The gain in terms of standard deviation is paid in terms of number of removed violations and total cost, where the gains are much more limited than the T-A model; it focuses on acting more on the few CSCIs with a high number of violations to meet the standard deviation target than on most impacting rules;
- the T-N model achieves its target too. Results are very similar to the standard deviation model.

The T-A model is able to select the best types of violations to remove, because its target is met early, and thus the residual effort is devoted to lower the non-removal effort value by selecting the most impacting types of violation. Of course, the choice of the target influences the output; thus the user can relax constraints on T-A & T-STD and T-N solutions to reduce the total expected cost. Conversely, whenever the targets are met with margin (e.g., in the T-A case), engineers can decide to spend less money and re-compute a solution with a lower available budget (e.g., 600 man-hours). They both represent ways in which company engineers were used to exploit the models during (and after) the project. The models most used by company engineers are the average model (as their primary objective was to reduce the average number of violations) and the T-N model in order to isolate problematic CSCIs. However, all the three models were included in the SVEVIA framework after this evaluation.

5.3. *Test planning*

Test optimization (i.e., the model described in Section 4.2.3) was evaluated retrospectively on the testing process of a case-study system for homeland security in charge of managing the port, maritime, and coastal surveillance. The system is made up of 5 CSCIs with size ranging from 22KLoC to 59KLoC and total time taken for development ranging from 6 to 12 months. These were tested between 2009 and 2012, using a total amount of testing resources of 326 man-weeks, and detecting in total 1119 bugs. The aim of the experiment was to figure out, after the testing completion, how many defects would have been detected by allocating the testing effort according to various competing schemes. Compared schemes were: *(i) uniform allocation* (same resources to all CSCIs), *(ii) size-based allocation* (a common rule-of-thumb approach), proportional to the size of CSCIs; *(iii)* two versions our allocation scheme, in a single-objective setting, named *defect-based* and *defect density-based*, where the objective to minimize are, respectively, the first objective of Eq. (9) and its variant with the number of defects over size (i.e. density) instead of absolute number of defects. Thus, we used 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. An initial budget of 150 man-weeks of testing to allocate to CSCIs is assumed. In both defect- and density-based schemes the "update" step was set to 4 weeks, namely the allocation to CSCIs was recomputed each month.

Fig. 8 summarizes the results in terms of total number of defects

[13] Absolute number of violations cannot be disclosed for confidentiality.

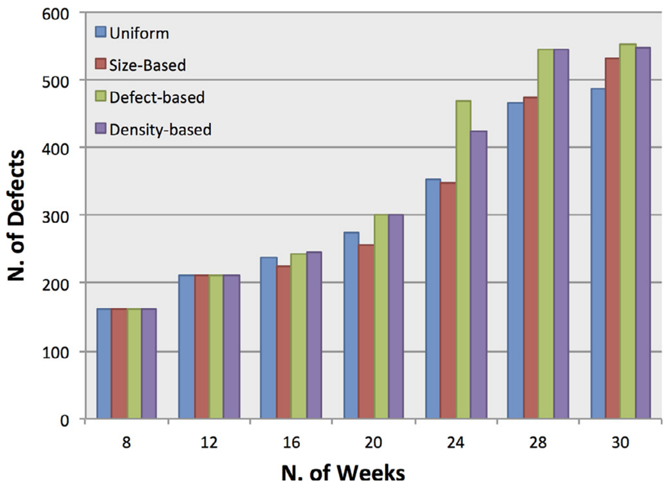


Fig. 8. Number of detected defects over testing time.

found by the various approaches as time (weeks) proceeds. The bars have the same value for the first 12 weeks; then, the dynamic schemes outperform the others. Detailed results on re-allocated weeks are in our previous work [41]. Besides the benefit of an optimized allocation, the evaluation shows that a *dynamic* allocation, with periodic updates, is important to be robust to violations of SRGM's assumptions (such as perfect or immediate debugging, inter-failure times independence, equal testing quality over time, no differences among testing teams). In the studied 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% of time, the selection of the best SRGM changed several times, because of data variability. For instance, testing of components C1 and C2 gave no result at all for months (ten/fifteen weeks) and then 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 changing over time; in any case, it is a clear example of violation of SRGM assumptions. The periodic re-computation of SRGMs makes the allocation method a robust solution to such variations. The functionality was integrated into SVEVIA in order to provide quantitative reasoning support for the management of testing resources, complemented by expert judgment about testing teams composition. It indeed also contributed to the increment of productivity observed in 2013–2015 in terms of total project delivery time and milestone hits, thanks to the improvement of the testing phase efficiency.

5.4. *High-level achievements*

After validation, the experimented models were incrementally added to the framework. Engineers started adopting the SVEVIA models to take quality improvement actions, for instance by exploiting the violation *removal plans* suggested by SVEVIA each 3 months; by analyzing the defect detection trends of integration/system testing activities and the V&V resource management with predictions made by SVEVIA (hence spotting problems with the testing process/teams preventively); by reasoning, at management-level, on quality-aware productivity trends as computed by SVEVIA. Users at different level (cf. with Fig. 2) started producing periodic reports/plans tailored for their profile and on data they could access – automatically managed by SVEVIA utilities – on all these aspects (code quality, testing and debugging and productivity measurements/prediction/optimization) and to use them as quantitative support for their decisions (e.g., during meetings). These practices entered (and contributed to) the overall structured processes about quality management created in those years.

Table 6
Company boost in quality management.

	2013	2015
Tracked CSCIs	1912	2498
Used bug trackers	20	7
Monitored projects	0	144
Analyzed LLOCs	24.5M	163.5

In December 2015 company's engineers assessed the SVEVIA contribution to the following high-level achievements:

- Implementation of structured processes for quality/productivity measurement, and for better defect management exploitation;
- Implementation of estimation, prediction and optimization algorithms for systematic quality/productivity assessment and improvement;
- Integration of tools already in use under a unique framework for software quality management.

The relevance of the SVEVIA support is highlighted by the following indicators in the years 2013–2015 (Table 6):

- The number of projects analyzed, as high as 144;
- The increase of tracked CSCIs, by 30.64% up to about 2500 CSCIs. The CSCIs measured by ASA and analyzed by SVEVIA are about 900, a number increased by 127% since 2013 (Fig. 9 shows the trend until 2015 3Q);
- The decrease in the number of different bug and issue trackers from 20 to 7, resulting in higher homogeneity of defect data;
- The boost in the LLOC analyzed in 2015, 6.7 times higher than in 2013.

SVEVIA has contributed to the following company achievements:

- **Software quality**: reduction of about 25% per year of the absolute number of rule infringements. Fig. 10 shows the trend of violation density (every 1KLOC for CR and 10KLoC for BD rules) in the time frame 2013 to 2015, denoting a total reduction of 33.5% for CR and 34.6% for BD rules.
- **Software productivity – time**: increase of milestone hits (i.e., projects delivered on time), where SVEVIA impact was assessed at about 90%.
- **Software productivity – outsourcing**: reduction of coding activities outsourced, where SVEVIA impact was assessed to be as high as 100%.

Currently, the framework is continuously fed with data of about 900 CSCIs belonging to tens of Projects of real-world large-scale mission-

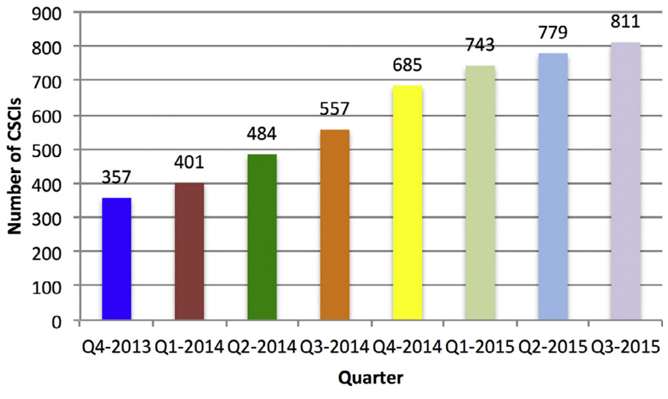


Fig. 9. CSCIs analyzed per trimester in 2013–2015.

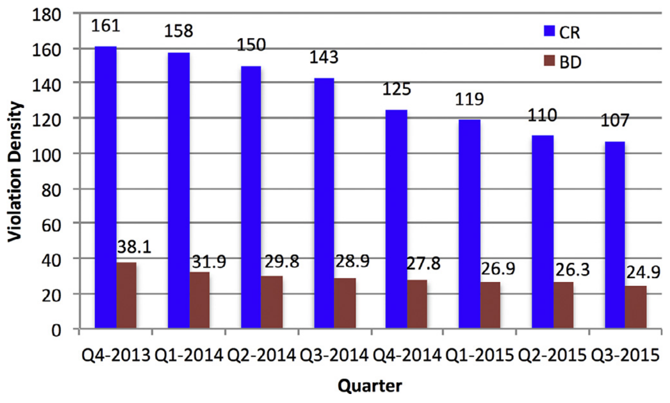


Fig. 10. Code quality improvement in years 2013–2015.

critical systems.

6. Lessons learnt

Today's large-scale mission-critical systems are software-intensive systems, typically manufactured by big companies, where the system engineering culture prevails over the software engineering one. Innovation in software quality management is not simply a matter of applying established techniques and adopting proper tools. It is a goal to be pursued in the medium-long term, yet the benefits need to be concrete, quantifiable and progressively visible. Support for strategic decision-making is crucial to this aim. Advanced software quality techniques, algorithms and methodologies are available in the scientific literature, but far from direct applicability in real industrial production environments. This is due to cost, skills, as well as organizational, cultural and human factors.

The approach followed by SVEVIA, and the key for its success, was bottom-up: rather than introducing sudden changes from the top, we experienced that starting from exploiting the information available almost 'for free' is by far more useful, as it requires low effort, investments and training, and very limited modifications to the way engineers use to work. We learned the following lessons:

- Tracking of engineering activities produces large amounts of data, which can be mined for extracting are highly valuable information to drive strategic decisions. Data-driven analysis techniques should be context-driven; they are best defined after careful identification of available data sources. Sometimes relevant information is hidden and needs to be made explicit.
- Quality and heterogeneity of data have to be carefully considered. In a large company, different units/teams use rather different approaches and tools, and collect data in many formats and with only apparently same semantics. One of the first functionalities developed in SVEVIA was an *import* of defect data and of code violations data from the various tools adopted across the company. While this may appear technically trivial, actually it required to analyze teams' practices and fill semantic gaps among similar data. *Domain knowledge* is essential: understanding differences among teams in various plants and countries is best accomplished by software engineers together with domain experts.
- Decision support is provided by sophisticated models and algorithms requiring data quality. Indeed, the more detailed and complete the information, the more types of analysis are enabled, and the more accurate their output. However, there is a trade-off between data quality and availability. The general principle we followed is to rely on data collectable with minimal impact on the workload of company engineers. The presented framework is adaptive with respect to the available data: modules can be added incrementally when more detailed data are made available.

Moreover, it may be important to provide stakeholders with incremental deliveries of small yet tangible results.

- The success of models is directly related to the *visibility* of results. For instance, the optimal code sanitization approach was the most successful functionality because of the immediate visible feedback: the measurements of violations regularly each 3 months favored a regular and clear feedback on the effect of optimization, hence boosting the adoption of this practice since the early phases of the project. Similarly, prediction was appreciated as another key functionality but only once observed its real “outcomes” (i.e., once observed that CSCI predicted as most defective were actually the most problematic ones), and not after the cross-validation of models (see Section 5.1). The effect of productivity of all the other activities (quality measurements, planning) was appreciated just later, because the cause-effect relation is less clear and explicit (hence the effect less visible) and comes out only after some time.
- The complexity of statistical and optimization models need to be made transparent to decision-makers. Moreover, separation of responsibilities is crucial to involve various managers in the company hierarchy. We used role profiling – the framework has a uniform interface, but differentiates the abstraction level of the information displayed depending on the role.
- Historical product/process data are very valuable for tailoring prediction and optimization techniques. As usual when introducing innovations in consolidated processes, they need to be non intrusive initially, yet they induce changes in practices over time. In the long term this results in quality data which can be mined for making explicit the knowledge traditionally implicit in the experience of company professionals.
- Setting up such a quality management framework encountered difficulties too. The initial cultural barrier and skepticism towards novelty in this type of industries (with consolidated and well-proven practices) was a first hurdle, but it was overcome quite soon thanks to a clear commitment at management level and to our bottom-up approach discussed above. The tailoring of scientific methods and techniques to real-world systems was a further hard engineering work, especially whenever the assumptions made by models were systematically violated and required more robust approaches (e.g., the dynamic test resources allocation). Academics and industrials worked constantly together on looking for good trade-offs between novelty of proposals and working prototypes. Validation on real systems was the key feedback to turn research ideas into innovative and concrete solutions. Finally, the integration on the existing process took several months: it involved the company organization at different sites on a global scale, engineers and developers from several departments, requiring changes in the process (e.g., new policies was released), in the tools (e.g., homogeneity of data collected by all the used issue trackers), and in the engineers' mentality (e.g., considering the central role of quality measurements as support for decision making and not as a control means).

7. Related work

There are many commercial or open source tools for software quality. Tools for accomplishing specific tasks such as code complexity metrics measurement, automatic static analysis, and bug tracking are sources of data leveraged by SVEVIA. The more detailed and fine-grained the data gathered, the more accurate the SVEVIA predictions and optimizations. On the other side of the spectrum, there are tools to provide project management services complementary to those for decision support offered by SVEVIA. Examples are: the mentioned Atlassian JIRA for software planning, tracking and release; RationalPlan[14] for resources allocation, workload analysis, progress tracking, costs estimation.

The tools more closely related to SVEVIA are those for product quality management (e.g., *SonarQube, IBM Rational Quality Manager* and *HP Enterprise Quality Center*) and for business intelligence (e.g., *JReport, SAP, Qlik, ORACLE, Tableau*). The formers support continuous monitoring, inspection and analysis of code quality, and offer reporting functionalities about phases of the software lifecycle – requirements management, project/process metrics, management of test cases, (non-optimal) test planning. To the best of our knowledge, none of such tools incorporates the analytical models peculiar of SVEVIA. While they feature management of resources, time, cost and quality, they offer neither prediction nor optimization services, able to quantitatively support managers in informed decision-making. Often, decision makers about software quality keep on relying on their own intuition and experience.

Business intelligence tools do provide advanced visualization and interactive dashboard reporting capabilities supporting quality-related decisions. Although several techniques they employ have some commonalities with those implemented in the SVEVIA framework (e.g., predictive modelling, data mining, reporting and visualization), their focus is not on software, but on aspects of business such as administration, customer relationship management, business and strategic planning, budgeting, operations and distributions, accounting and financials. SVEVIA techniques are grounded on software-specific models we developed for quality/cost/time prediction and optimization. They embrace several research areas in the software engineering field, including our own research work about reliability growth modeling [50], defect analysis [23], test planning [41], optimal resource allocation [42,51], defect prediction [52]. Unlike other tools, defect prediction in SVEVIA exploits both SRGMs and the so-called fault-proneness models. These approaches exploit different principles and use different data to train models, as the former predicts the number of remaining defects by observing the trend of inter-failure times during testing or operational phase, whereas the latter exploit code-level metrics (in our case, ASA violations) and a global history of pairs *metrics-defects information* to train machine learning models and predict which component is defective (by a binary or ranking outcome), rather than predicting the number of defects. SVEVIA provides both, as they can be exploited in different phases (e.g., the SRGM-based prediction is useful during testing or for release planning, the fault-proneness prediction is useful before testing to spot critical components) and by different actors, depending on the information available at a given stage.

SVEVIA uses SRGMs also for test planning. This functionality is embedded in a dynamic approach that can re-evaluate the best allocation on-demand, based on the available test data, in a multi-objective setting (to balance cost, time and reliability), and by comparing eight different models to give faithful predictions. These are all features not available in existing tools either for test planning and/or for defect prediction. Optimization for code sanitization is a further key novelty. Indeed, tools for static analysis, like *SonarQube* or *Cast*, offer facilities to suggest the violations to remove, based on the severity of violated rules. They do not employ, however, an optimization model to minimize a cost/technical-debt function able to tell how many violations and of which type should be removed from which single component in order to attain a desired quality target in terms of number of violations.

Finally, although existing tools do provide numerous metrics for assessing the quality of processes, SVEVIA exploits prediction to build more informative metrics. For instance, stating that testers have detected 10 defects in a component is not informative about the quality of their work nor about the quality of the component. There is a big difference if the component has 15 or 50 total defects: including the estimated number of defects (as we did with SRGMs-based estimation) is indeed more informative, despite the metric is a statistical estimate and not a deterministic one. Another example is the quality-aware productivity, where quality information is embedded in the productivity figure. The metrics in SVEVIA support a quality assessment of the three

[14] http://www.rationalplan.com.

key processes – coding, testing, bug fixing – more oriented to decision making.

In summary, the added value of SVEVIA services comes from:

- statistical techniques (time series analyses, stochastic modeling and multi-objective optimization) applied to software quality assessment/improvement and test planning;
- reliability growth models and fault-proneness models for prediction;
- code infringements and tests prioritization algorithms;
- innovative metrics and key performance indicators suited to support decisions related to software quality;
- an integrated framework where all the presented models and metrics about the three key processes are provided together in a single platform.

8. Conclusions

Large-scale software-intensive mission-critical systems are engineered in ecosystems centered around big industries or system integrators, and encompassing tool providers and external suppliers. Innovations in software quality management processes in these industries are constrained by cost, organizational, cultural and human factors. We have presented the SVEVIA framework for supporting strategic decisions concerning software quality and productivity assessment and improvement, and effective planning of resources usage. It features low intrusiveness in existing quality monitoring practices, by exploiting data from available heterogeneous sources, including automated static analysis and test management tools, and issue trackers.

Based on advanced mathematical models and algorithms, made transparent to decision-makers, the framework provides support for software quality estimation, for prediction of quality-cost-time trends, and for optimization of the allocation of resources to software verification and validation activities.

The results and the lessons learnt with an ATC pilot project and with three years of field data of tens of projects in various mission-critical system domains show the effectiveness at supporting managers in making thoughtful decisions concerning software quality and engineering processes.

Acknowledgments

The work by Pietrantuono and Russo has been supported by the PRIN 2015 project “GAUSS” funded by MIUR Grant no. n. 2015KWREMX. The work by G. Carrozza was performed while she was Head of Metrics and Reporting Unit at Selex ES.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.infsof.2018.05.009.

References

[1] C. Jones, O. Bonsignour, The Economics of Software Quality, Financial Times/Prentice Hall, 2011.
[2] R.L. Glass, Software Runaways: Monumental Software Disasters, Prentice Hall, 1998.
[3] US Department of Defense, MIL-STD-498, Overview and Tailoring Guidebook, 1996, available online http://www.abelia.com/498pdf/498GBOT.PDF.
[4] Association for Project Management, A History of the Association for Project Management, 2010, 1972–2010, https://www.apm.org.uk/sites/default/files/a-history-of-the-association-for-project-management_lr.pdf.
[5] ISO/IEC 25010 – Software Product Quality, 2011, http://iso25000.com/index.php/en/iso-25000-standards/iso-25010.
[6] R. Chillarege, et al., Orthogonal defect classification – a concept for in-process measurements, IEEE Trans. Softw. Eng. 18 (11) (1992) 943–956.
[7] L.O. Damm, L. Lundberg, C. Wohlin, Determining the improvement potential of a software development organization through fault analysis: a method and a case study, Proceedings of the Eleventh International Conference on Software Process Improvement, LNCS 3281 Springer, 2004, pp. 138–149.
[8] MISRA-C:2012 – Guidelines for the use of the C Language in Critical Systems, 2013, http://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/Default.aspx.
[9] C. Catal, B. Diri, A systematic review of software fault prediction studies, Expert Syst. Appl. 36 (4) (2009) 7346–7354.
[10] D. Cotroneo, R. Pietrantuono, S. Russo, K. Trivedi, How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation, J. Syst. Softw. 113 (2016) 27–43.
[11] M. Cinque, C. Gaiani, D. De Stradis, A. Pecchia, R. Pietrantuono, S. Russo, On the impact of debugging on software reliability growth analysis: a case study, Proceedings of the 2014 Computational Science and its Applications, LNCS 8583 Springer, 2014, pp. 461–475.
[12] A. Goel, Software reliability models: assumptions, limitations and applicability, IEEE Trans. Softw. Eng. SE-11 (1985) 1411–1423.
[13] L. Pham, H. Pham, Software reliability models with time-dependent hazard function based on Bayesian approach, IEEE Trans. Syst. Man Cybern. Part A Syst. Hum. 30 (1) (2000) 25–35.
[14] C.G. Bai, K.Y. Cai, Q.P. Hu, S.H. Ng, On the trend of remaining software defect estimation, IEEE Trans. Syst. Man Cybern. Part A Syst. Hum. 38 (5) (2008) 1129–1142.
[15] A. Kumar, Software reliability growth models, tools and data sets – a review, Proceedings of the Ninth ACM India Software Engineering Conference, (2016), pp. 80–88.
[16] H. Okamura, Y. Watanabe, T. Dohi, An iterative scheme for maximum likelihood estimation in software reliability modeling, Proceedings of the Fourteenth IEEE International Symposium on Software Reliability Engineering, (2003), pp. 246–256.
[17] K. Ohishi, H. Okamura, T. Dohi, Gompertz software reliability model: estimation algorithm and empirical validation, J. Syst. Softw. 82 (3) (2009) 535–543.
[18] A. Goel, K. Okumoto, Time-dependent error-detection rate model for software reliability and other performance measures, IEEE Trans. Reliab. R-28 (3) (1979) 206–211.
[19] S. Yamada, M. Ohba, S. Osaki, S-Shaped reliability growth modeling for software error detection, IEEE Trans. Reliab. R-32 (5) (1983) 475–484.
[20] S. Gokhale, K. Trivedi, Log-logistic software reliability growth model, Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, (1998), pp. 34–41.
[21] R. Mullen, The lognormal distribution of software failure rates: application to software reliability growth modeling, Proceedings of the Ninth IEEE International Symposium on Software Reliability Engineering, (1998), pp. 134–142.
[22] H. Okamura, T. Dohi, S. Osaki, EM algorithms for logistic software reliability models, Proceedings of the Twenty-Second IASTED International Conference on Software Engineering, (2004), pp. 263–268.
[23] G. Carrozza, R. Pietrantuono, S. Russo, Defect analysis in mission-critical software systems: a detailed investigation, J. Softw. Evol. Process 27 (1) (2015) 22–49.
[24] N. Schneidewind, Modelling the fault correction process, Proceedings of the Twelfth IEEE International Symposium on Software Reliability Engineering, (2001), pp. 185–190.
[25] R. Plosch, et al., On the relation between external software quality and static code analysis, Proceedings of the Thirty-Second Annual IEEE Software Engineering Workshop, (2008), pp. 169–174.
[26] N. Nagappan, T. Ball, Static analysis tools as early indicators of pre-release defect density, Proceedings of the Twenty-Seventh International Conference on Software Engineering, ACM, (2005), pp. 580–586.
[27] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–13.
[28] D. Cotroneo, R. Natella, R. Pietrantuono, Predicting aging-related bugs using software complexity metrics, Perform. Eval. 70 (3) (2013) 163–178.
[29] T. Ostrand, E. Weyuker, R. Bell, Predicting the location and number of faults in large software systems, IEEE Trans. Softw. Eng. 31 (4) (2005) 340–355.
[30] F. Frattini, R. Pietrantuono, S. Russo, Principles of Performance and Reliability Modeling and Evaluation, Reproducibility of Software Bugs - Basic Concepts and Automatic Classification, Springer Series in Reliability Engineering, Springer, 2016, pp. 551–565.
[31] C. Seiffert, T.M. Khoshgoftaar, J.V. Hulse, Improving software-quality predictions with data sampling and boosting, IEEE Trans. Syst. Man Cybern. Part A Syst. Hum. 39 (6) (2009) 1283–1294.
[32] D. Drown, T. Khoshgoftaar, N. Seliya, Evolutionary sampling and software quality modeling of high-assurance systems, IEEE Trans. Syst. Man Cybern. Part A Syst. Hum. 39 (5) (2009) 1097–1107.
[33] X. Yang, K. Tang, X. Yao, A learning-to-rank approach to software defect prediction, IEEE Trans. Reliab. 64 (1) (2015) 234–246.
[34] Y. Yang, J.O. Pedersen, A comparative study on feature selection in text categorization, ICML 97 (1997) 412–420.
[35] M.J.D. Powell, A Direct Search Optimization Method that Models the Objective and Constraint Functions by Linear Interpolation, Springer, Dordrecht, the Netherlands, 1994, pp. 51–67.
[36] X. Han, et al., Optimization-based decision support software for a team-in-the-loop experiment: asset package selection and planning, IEEE Trans. Syst. Man Cybern. Syst. 43 (2) (2013) 237–251.
[37] C.Y. Huang, S.Y. Kuo, M.R. Lyu, An assessment of testing-effort dependent software reliability growth models, IEEE Trans. Reliab. 56 (2) (2007) 198–211.
[38] C.Y. Huang, M.R. Lyu, Optimal release time for software systems considering cost, testing-effort, and test efficiency, IEEE Trans. Reliab. 54 (4) (2005) 583–591.
[39] B.W. Boehm, Software Engineering Economics, Prentice Hall, 1981.
[40] C. Huang, J. Lo, Optimal resource allocation for cost and reliability of modular software systems in the testing phase, J. Syst. Softw. 79 (5) (2006) 653–664.

[41] G. Carrozza, R. Pietrantuono, S. Russo, Dynamic test planning: a study in an industrial context, Int. J. Softw. Tools Technol. Trans. 16 (5) (2014) 593–607.
[42] R. Pietrantuono, S. Russo, K. Trivedi, Software reliability and testing time allocation: an architecture-based approach, IEEE Trans. Softw. Eng. 36 (3) (2010) 323–337.
[43] G. Zhang, Z. Su, M. Li, F. Yue, J. Jiang, X. Yao, Constraint handling in NSGA-II for solving optimal testing resource allocation problems, IEEE Trans. Reliab. 99 (2017) 1–20.
[44] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[45] E. Zitzler, S. Künzli, Indicator-Based Selection in Multiobjective Search, Springer, Berlin, Heidelberg, 2004, pp. 832–842.
[46] A.J. Nebro, J.J. Durillo, F. Luna, B. Dorronsoro, E. Alba, Design issues in a multiobjective cellular genetic algorithm, Proceedings of the 2007 Evolutionary Multi-Criterion Optimization, LNCS 4403 Springer, 2007, pp. 126–140.
[47] J. Knowles, D. Corne, The Pareto archived evolution strategy: a new baseline algorithm for pareto multiobjective optimisation, Proceedings of the 1999 IEEE Congress on Evolutionary Computation (CEC), 1 (1999), p. 105.
[48] R. Pietrantuono, P. Potena, A. Pecchia, D. Rodriguez, S. Russo, L. Fernández-Sanz, Multiobjective Testing Resource Allocation Under Uncertainty, IEEE Transactions on Evolutionary Computation 22 (3) (2018) 347–362, http://dx.doi.org/10.1109/TEVC.2017.2691060.
[49] J. Demšar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.
[50] M. Cinque, D. Cotroneo, A. Pecchia, R. Pietrantuono, S. Russo, Debugging-workflow-aware software reliability growth analysis, Softw Test Verif Reliab. 27 (2017) e1638. https://doi.org/10.1002/stvr.1638 .
[51] G. Carrozza, M. Cinque, U. Giordano, R. Pietrantuono, S. Russo, Prioritizing correction of static analysis infringements for cost-effective code sanitization, Proceedings of the Second IEEE/ACM International Workshop on Software Engineering Research and Industrial Practice, (2015), pp. 25–31.
[52] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, Analysis and prediction of mandelbugs in an industrial software system, Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation, (2013), pp. 262–271.