# Prioritizing Correction of Static Analysis Infringements for Cost-Effective Code Sanitization

Gabriella Carrozza*, Marcello Cinque†‡, Ugo Giordano†, Roberto Pietrantuono†‡, Stefano Russo†‡
*SELEX ES, A Finmeccanica Company, Piazza Montegrappa 4, 00195, Roma, Italy. E-mail: gabriella.carrozza@selex-es.com
†DIETI, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy.
E-mail: {macinque, ugo.giordano, sterusso, roberto.pietrantuono}@unina.it
‡ Critiware s.r.l., Incipit, Via Cinthia, Complesso Univ. Monte S. Angelo, 80126, Napoli, Italy
E-mail: {marcello.cinque, stefano.russo, roberto.pietrantuono}@critiware.com

***Abstract*—Static analysis is a widely adopted technique in the industrial development of software systems. It allows to automatically check for code compliance with respect to predefined programming rules. When applied to large software systems, sanitizing the code in an efficient way requires a careful guidance, as a high number of (more or less relevant) rule infringements can result from the analysis.**

**We report the results of a static analysis study conducted on several industrial software systems developed by SELEX ES, a large manufacturer of software-intensive mission-critical systems. We analyzed results on a set of 156 software components developed in SELEX ES; based on them, we developed and experimented an approach to prioritize components and violated rules to correct for a cost-effective code sanitization. Results highlight the benefits that can be achieved in terms of quality targets and incurred cost.**

## I. Introduction

A common way to reduce software defectiveness is to improve activities related to the detection of defects before systems deployment. "Early detection" refers to the attempt of discovering defects at the earliest stage (i.e., in the same development phase in which they get injected) in order to avoid their later degeneration into operational failures and reduce fixing and maintenance costs. A powerful early detection technique, widely adopted for large-scale software systems, is static code analysis. It is based on the exploration of all possible execution paths in a program at compile time to check if some properties of interest (e.g., coding rules, programming conventions) are respected. Its application is very useful to identify programming errors (such as buffer overflows, null pointer dereferences, the use of uninitialized variables) that can escape both compilers' detection facilities and testing campaigns. The use of static analysis can be traced back to '90s, with many respectful software giants, like Microsoft and Nasa, massively using this technique with very good results [1], [2]. Results of static analysis can provide several insights into the quality of the code, indicating, for instance, which components have the greatest violation share and which kind of rules are more violated. A common problem with static analysis approaches is the overwhelming amount of information they provide, which makes it difficult to recognize the most important infringements to correct and the most critical components. This is especially relevant when dealing with large software projects, spanning hundreds of components and millions of lines of code, where the proper allocation of efforts for cleaning the code can have a significant impact both in terms of quality increase and cost reduction.

This paper presents the approach taken to face this problem in the context of an industry-academia collaboration between University of Napoli and SELEX ES[1]. The company massively adopts static analysis on its products for supporting code improvement and defect detection/removal. Considering the huge amount of code, judiciously prioritizing efforts for code sanitization is expected to lead to considerable cost savings. To this end, we applied static analysis on a set of 156 software components developed in SELEX, with reference to 167 coding rules; based on results, we defined and experimented a method for effort allocation, using its output as feedback to developers and tester as suggested improvement actions. Specifically, SELEX engineers have been provided with:

- statistics about rule violations across components, which are important indicators used to understand the current status and set the desired improvement goals;
- statistics about rules most often violated by programmers, which are useful for internal regulations (i.e., coding standard enforcement, tailored training) and to suggest rules that can be confidently neglected (thus, not computed in the future) because not often violated;
- a direct indication regarding which components and which rules should be prioritized for an effective code sanitization. With a so-high number of components, it is important for SELEX engineers to focus their effort to attain a desired quality objective at minimal cost. We define three variants of an optimization model depending on the desired quality goal, and investigated their performance in terms of cost reduction with respect to a random effort allocation resembling the currently adopted approach.

Results showed that, given pre-defined quality targets on the code (in terms of average number of rule violations, or of their

[1]SELEX ES, a Finmeccanica company, is an international leader in electronic and information solutions for defense, aerospace, security, surveillance, network management, information security and mission-essential services.

standard deviation, or of a desired percentile value) and a fixed budget, the greatest gain in quality achieved, in our experiment, is up to 15.99% of average number violation reduction, up to 59.61% in terms of standard deviation reduction, and up to 42.21% in terms of expected cost for residual violations[2]. Besides the effective results, we experienced that, in our long-lasting academia-industry partnership, key factors for the success of the collaboration were: the definition of a clear and short-term objective, the continuous feedback from one side to another, the aptitude of someone from the company for exploring new and non-trivial solutions to common problems, and the low intrusiveness of techniques proposed by the academic partner in the daily work of company's practitioners.

## II. Background and Related Work

### A. Automated Static Analysis Tools

Static analysis can be used to check if the code meets the expectations around security, dependability, performance, and maintainability [3]. It is able to provide a foundation for producing solid code, by exposing structural errors or preventing entire classes of defects. Automatic static analysis (ASA) tools identify specific types of anomalies (i.e., violation of properties) by pattern-based analysis, data and control flow analysis, path flow analysis, and so on. A common usage of ASA is to check the source code against patterns known to cause defects or, more generally, to penalize quality. This focuses on checking compliance to coding standard rules for preventing improper language usage, satisfying industry standards (e.g. JSF, MISRA etc.), and enforcing internal coding guidelines [4]. There is a wide range of tools, some open source and some commercial, that differ by the language type that are able to analyze. For example, *FindBugs*[3] and *PMD*[4] are two open source tools that analyze, respectively, the Java bytecode and source code. By static rule-based analysis, they identify potential problems related to programming errors as well as programming style issues. *PC-lint*© and *FlexeLint*©[5] check C/C++ source code for detecting defects, anomalies, non-portable constructs, inconsistencies and redundant code. Additionally, there are several tools that support multiple programming language, such as *SonarQube*[6]. None of the tools strictly encloses another, and each tool is likely to find different types of defects [5].

Although the usage of tools leads definitely to benefits, they have limitation when applied in practice. An important issue of ASA is the large number of false positives, warning about defects that the program does not actually contain. An extreme case is reported in [6], where authors report more than 96% of raised coding concerns not relating to any defect or refactoring modification. To overcome this issue, it is often possible to customize the tools by filtering out non-relevant rules, so as to reduce, at least in part, the number of false positives.

From a research perspective, there are two main streams regarding ASA and code quality: *i)* understanding whether (and which) ASA issues are real indicators of defects, or *ii)* whether ASA issues can be used as early indicators of defect-prone modules (e.g. software components, files, classes). Along the former stream, besides the mentioned problem of false positives (spotted in several papers, e.g., [6], [7], [8]), many studies observed that it is more effective to use a set of ASA issues to identify the most defect-prone components rather than an individual ASA issue. For instance, the authors in [9] found a positive correlation between *FlexeLint* issues and a large-scale telecommunications system; similarly it is done in [7], where issues densities from two ASA tools, *PREfix* and *PREfast*, are used successfully to predict defect-prone components, and in [10] which confirmed the correlation between ASA tools (*FindBugs* and *PMD*) issues and defect data Eclipse SDK. To the best of our knowledge, while there are several approaches to allocate effort for testing activity (e.g., [11], [12], [13]), the problem of allocating effort for code sanitization activity after the application of ASA has not been addressed; in the following, we focus on such a problem with reference to components and types of violated rules.

### B. Parasoft© tools for programming conventions analysis

To improve the quality of software, it is highly recommended to follow "programming conventions", which are guidelines that recommend about programming style, practices, and methods to follow when writing a program. Usually, these conventions cover file organization, indentation, comments, declarations, statements, programming principles, architectural best practices, and so on. In SELEX ES, the *Parasoft*© *test* tool (for C/C++ and Java) has been used to analyze statically the code and monitor compliance with standard rules. *Parasoft*© *test* is an ASA tool that includes technology for static code analysis, peer code review automation, unit test case generation and execution, and runtime error detection. It differentiates the coding rules into two main groups: *Bug Detective* (BD) and *Coding Rules* (CR). The first group includes those rules that, if not met, are more likely to lead to software defects; the rules of the second group refers to programming style guidelines. These two groups are in turn divided into sub-groups of rules, reported in Table I[7].

## III. Industrial Context and Data Source

SELEX ES is actually the outcome of a recent process through which three big companies have been merged into one, addressing an impressive variety of user application domains. It is the prime software company within the Finmeccanica group, being in charge of almost the 80% of the overall software solutions underlying the mother company's portfolio. The presented study results from an active collaboration

[2]Results in the paper are always reported as a percentage measure, for confidentiality reasons.

[3]http://findbugs.sourceforge.net/.

[4]http://pmd.sourceforge.net/

[5]http://www.gimpel.com/html/products.htm

[6]http://www.sonarqube.org/

[7]Note that we grouped together (in BD-G5 and CR-G12 denoted as "other rules") the rules referring to Java, which are numerous and less relevant for us, being most of the code in C/C++.

TABLE I: Macrogroups of coding rules.

| | Macrogroups of rules | Number of rules |
|---|---|---|
| BD-G1 | Possible Bugs | 11 |
| BD-G2 | Resources | 4 |
| BD-G3 | Security | 7 |
| BD-G4 | Threads & Synchronization | 3 |
| BD-G5 | Other rules | 13 |
| CR-G1 | Formatting Rules | 5 |
| CR-G2 | Metrics rules | 1 |
| CR-G3 | Coding Convention Rules | 6 |
| CR-G4 | MISRA Rules | 34 |
| CR-G5 | Naming Convention Rules | 1 |
| CR-G6 | Initialization Rules | 3 |
| CR-G7 | Optimization Rules | 1 |
| CR-G8 | Object Oriented Programming Rules | 5 |
| CR-G9 | Memory and Resource | 6 |
| CR-G10 | Comments Rules | 1 |
| CR-G11 | Exceptions Rules | 2 |
| CR-G12 | Other rules | 67 |

between University of Naples and SELEX ES on software quality topics, which started few years ago and is yielding relevant research results in the area of testing [14], [11], [15], [16], and defect analysis and process evaluation [17].
This study has been running close in touch with the SELEX ES SW Engineering function (hereafter SWEng), accounting for about 800 people distributed over more than 20 plants in Italy and UK.

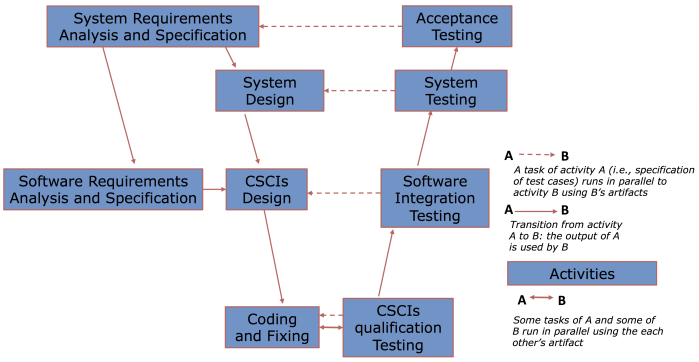


Fig. 1: Selex development lifecycle

For systems development, the SWEng department adopts, as reference development life-cycle process, a customization of the traditional V-model (Figure 1). In terms of artifacts and software documentation, the critical domains they address demand the application of a rigorous widely known military standard named MIL-STD-498 [18], which establishes uniform requirements for software development and documentation. According to [18], software projects are organized into configuration items (named CSCI, Computer Software Configuration Item) that represent the smallest software unit addressed in this paper. CSCIs we considered go from 100 to hundred of thousands of LLOC (Logical Lines of Code) and come from several industrial capabilities like "Basic and Middleware Applications", "Human Machine Interface", "Surveillance Data Processing". To cope with the variety of applications, the SWEng function has designed a uniform policies and processes framework encompassing phases from requirements analysis to software verification and system validation. Within this framework, the SWEng direction decided to introduce static analysis in the overall process. The real strength of such a move lies into the feeling that, apart from outcomes in terms of infringed coding rules, many other results in terms of software quality and cost reduction can be pursued. The study conducted to this aim has been run on a set of 156 CSCIs, amounting to a total of 11,66 Million of LoC. The source code has been made available for the analysis, and the output used for investigating the effectiveness of proposed models. The following steps are implemented: *i)* application of *Parasoft*© tool to CSCI source code; *ii)* output data normalization in terms of percentage of rule violations per CSCI; *iii)* inferring of basic percentage statistics on rule violations, considering separately the BD from CR groups; *iv)* definition and application of effort allocation models, suggesting which violations to remove to cost-effectively achieve a desired quality goal. The next subsections report the results for these steps.

## IV. Results

### A. BD and CR Rules Violations

Tables II and III report, respectively, the total BD and CR rule violations separated by group expressed in percentage terms. As for BD, rules of group 1 ("possible bugs") are clearly the most relevant ones accounting for 81% of total violations; for CR, the most relevant groups are the first 4 (about 75% of total violations) along with the last one ("other rules") accounting for about 20 % violations.

The distribution of rule violations across CSCIs is characterized by a high standard deviation; the coefficient of variation of BD rules amounts to 2.34 (namely, standard deviation more than the double of the mean), and 1.82 for CR rules. This suggests a high skew, hence there are few CSCIs with high number of violations – a first indication useful for an effective allocation of sanitization effort.

For a more useful characterization, we are required to take into account the different size of the CSCIs, thus focusing on densities. Specifically, we consider violations per each 10KLoC (for the BD rules), and per each 1 KLoC (for the CR rules). The former are normalized over 10KLoC as they are deemed more critical by SELEX engineers with respect to CR violations. Then, to filter out less relevant rules, we focus on the top-15 BD and CR rules, namely the 15 rules most frequently violated in terms of violations per 10KLoC and per 1KLoC for BD and CR rules, respectively. The top 15 BD rules belong to these groups: 8 to BD-G1 (including the top 5), namely "possible bugs", 1 to BD-G2 ("resources"), 1 to BD-G3 ("security"), 1 to BD-G4 ("threads synchronization"), and 4 to BD-G5 ("others"). The top 15 CR rules belong to groups as follows: 2 to CR-G1 ("formatting"), 1 to CR-G2 ("metrics rules"), 3 to CR-G3 ("coding conventions"), 5 to CR-G4 ("MISRA"), 1 to CR-G10 ("comments"), and 3 to CR-G12 ("others"). The type of violated rules provide hints on what

TABLE II: Percentage of group violations over total violations for Bug Detective rules.

| | BD-G1 | BD-G2 | BD-G3 | BD-G4 | BD-G5 |
|---|---|---|---|---|---|
| % of group violations | 81.07% | 8.57% | 2.72% | 1.41% | 6.23% |

TABLE III: Percentage of group violations over total violations for Coding rules.

| | CR-G1 | CR-G2 | CR-G3 | CR-G4 | CR-G5 | CR-G6 | CR-G7 | CR-G8 | CR-G9 | CR-G10 | CR-G11 | CR-G12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| % of group violations | 6.58% | 12.80% | 36.03% | 19.86% | 0.54% | 0.23% | 0.38% | 0.77% | 0.30% | 1.88% | 0.93% | 19.69% |

are the most common encountered problems, driving possible process-level actions (enforcing coding standard, driving training, etc.).

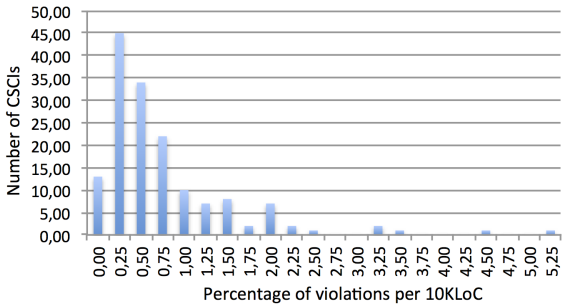


Fig. 2: Distributions of BD violations.

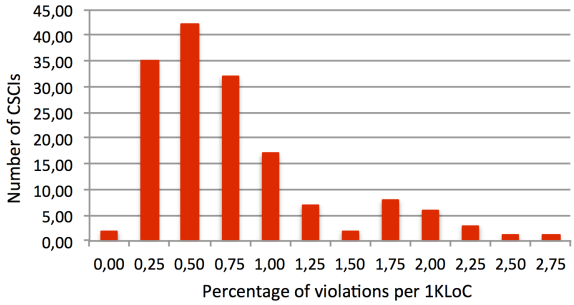


Fig. 3: Distributions of CR violations.

The top-15 rule violations account for most of violations: the top-15 BD rules amount to the 94.69% of total BD violations; the top-15 CR rule violations amount to the 80.05 % of the total CR violations. We can reasonably neglect the remaining ones, reducing significantly the rules that developers need to check against (their removal is considered as a lower priority activity with respect to the top-15). Figures 2 and 3 show the distribution of the percentage number of BD violations per each 10KLoC and of CR rule violations per each 1KLoC rule, with respect to the 156 CSCIs and to the top-15 rules. The distributions are both right-tailed, with 10% of CSCIs accounting for about 47% of total BD violations per 10KLoC and for 69% of total CR violations per 1KLoC. Both results on rule types and distribution across CSCIs suggest that efforts could be focused on few CSCIs and few rules. In the next section, the models to perform the allocation automatically and optimally are presented.

### B. Optimization Model

The key is to exploit static code analysis to suggest how much effort to devote to improve the quality of a CSCI code, and by means of what type of activities – namely, which, among the violated rules, are worth to be corrected. The objective is to focus on CSCI and activities more likely to lead to a cost reduction while satisfying a user-specified constraint on remaining violations.

*1) Model formulation:* Models are parametrized and applied separately to the top 15 BD and CR rules (i.e., one model for BD and one for CR). Let us denote by $v_i$ the total number of violations of the $i-$th type, and by $v_{i,j}$ the number of violation of type $i$ revealed in the CSCI $j$. A violation, if not removed, can result in a software defect in later stages; on the other hand, its removal requires some effort. The effort required to the removal of a violation is, of course, lower than the effort to remove a defect. We assume each type of violation being characterized by a pair $<RE_i, w_i>$, where:

- $RE_i$ (*Removal Effort*) is the effort per unit needed to remove one violation of the $i-$th 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.

The $RE_i$ values are assessed by historical data about the removal of violations of the $i-$th type observed in the past. Weights are assigned by experts judgement (i.e., SELEX engineers), who gives more importance to specific type of violation they deem more critical (also based on the tool's suggestions about the severity of each rule type and about the affected characteristics, e.g., reliability, maintainability, security). Let us denote with $E_d$ the effort per unit to remove a defect in the integration testing phase. Similarly to $RE_i$, an estimate of this value is obtained by historical repository tracing the man-hours devoted to defect removal activities; clearly, $E_d >> RE_i$. We have:

$$NRE_i = w_i * E_d \quad (1)$$

where $NRE_i$ is the *Non-Removal Effort*, and represents the expected effort per unit incurred if the violation of type $i$ is not removed. It is given by the the unitary effort for a defect removal weighted by the impact of violation $i$. Obviously, it would be extremely expensive to eliminate all the violations of

all types from all CSCIs. The problem is to find the optimal trade-off between the number and the type of violations to remove (i.e., determining the cost of removing violations), and the risk of not removing them. In the following, we denote with $x_{i,j}$ the number of violations of the $i$-th type that we decide to eliminate from CSCI $j$, which are our decision variables. Violations removal needs to be carried out within a budget constraint and a minimal quality constraint. Depending on how the quality constraint is expressed, we have different variants of the model:

**Target Average** (**T-A**): in this model variant, engineers want the mean number of violations over CSCIs to be under a specified target, $\bar{v}_{max}$.

**Target Average and Standard Deviation** (**T-A & T-STD**): this is a more restricted choice than the previous one, requiring also the standard deviation to be less than a user-specified target ($std_{max}$), besides the mean.

**Target Number** (**T-N**): engineers want a given number of CSCIs, denoted as $CN$, with less violations than a user-specified $v_{max}$.

For all the variants, the objective is the same: reducing the total cost, denoted as $C$, that is the sum of costs on all the CSCIs ($C_j$). Considering the above definitions:

$$C_j \;=\; \sum_{i=1}^{m} x_{i,j} \cdot RE_i + \sum_{i=1}^{m} \Big[ (v_{i,j} - x_{i,j}) \cdot NRE_i \Big] \qquad (2)$$

It represents the cost, for a CSCI $j$, of removing $\sum_i x_{i,j}$ violations plus the expected cost of not removing them. The optimization model follows:

$$\text{Minimize } C = \sum_{j=1}^{n} E_j$$
$$= \sum_{j}^{n} \Big[ \sum_{i=1}^{m} (x_{i,j} * RE_i) + \sum_{i=1}^{m} \big[ (v_{i,j} - x_{i,j}) * (w_i * E_d) \big] \Big] \qquad (3)$$

subject to:

$$0 \le x_{i,j} \le v_{i,j}$$

*and one of the following constraints according to the variant:*

$$\tfrac{1}{N} \sum_{j=1}^{n} \textstyle\sum_{i=1}^{m} (v_{i,j} - x_{i,j}) \le \bar{v}_{max} \qquad \text{(T-A Model)}$$

$$\tfrac{1}{N} \sum_{j=1}^{n} \textstyle\sum_{i=1}^{m} (v_{i,j} - x_{i,j}) \le \bar{v}_{max} \qquad \text{(T-A\&T-STD Model)}$$
$$STD[\textstyle\sum_{i=1}^{m} (v_{i,j} - x_{i,j})] \le std_{max}$$

$$\textstyle\sum_{j=1}^{n} q_j \ge CN \qquad \text{(T-N Model)}$$

$$where \;\; q_j = \begin{cases} 1 & \text{if } \sum_{i=1}^{m} v_{i,j} - x_{i,j} > v_{min} \\ 0 & \text{otherwise} \end{cases}$$

where $j = 1 \ldots n-1, n$ are the CSCIs, $x_{i,j}$ are the decision variables, and STD denotes the standard deviation, STD[$X$] = $\sqrt{\frac{\sum_i (x_i - \bar{x})^2}{n}}$.

*2) Application of the Models:* The model output suggests how many violations of each type we should remove from each of the 156 CSCI, in order to attain the desired target at minimal effort. We compare four different scenarios: the three variants against each other and against a *Random* (**R**) strategy. The latter reflects the current practice, where each CSCI manager independently decides how many violations to remove for its CSCI, arbitrarily choosing the type of violation. We force the random strategy to consume all the available budget so as to have a fair comparison with the other strategies. Additionally, note that the solution computed by the proposed models is an exact one; instead, the solution provided by the random allocation may be different from run to run. Therefore, we repeated the execution of the random strategy 50 times, in order to have statistically meaningful results, and considered the average solution in terms of total cost.

**Model Parametrization**

Costs of removal per violation type, $RE_i$, are, in our case, estimated by averaging the effort employed in the past to remove violations of each type, adjusted by SELEX engineers. For BD rules, *RE* values are between 1 and 5 minutes per violation in the average, depending on the violation type, while, in the case of CR rules, for which the tool support for automatic removal is remarkable, the time is less than 1 minute. Moreover, SELEX engineers considered a defect removal effort, $E_d$, amounting to 10 times the average effort for removing a rule violation[8]. Weights are assigned as mentioned above, i.e., by means of severity scales. RE and NRE parameters for BD and CR rule are reported in Table IV.

TABLE IV: RE and NRE Values (minutes).

| **BD Rules** | RE | NRE | **CR Rules** | RE | NRE |
|---|---|---|---|---|---|
| **1** | 5 | 22.6 | **1** | 0.1 | 0.01 |
| **2** | 4 | 18.8 | **2** | 0.5 | 1.0 |
| **3** | 3 | 21.6 | **3** | 0.1 | 1.25 |
| **4** | 3 | 13.6 | **4** | 0.05 | 1.42 |
| **5** | 4 | 8.8 | **5** | 0.1 | 1.53 |
| **6** | 4 | 8.8 | **6** | 0.2 | 0.88 |
| **7** | 2 | 16.8 | **7** | 0.1 | 1.12 |
| **8** | 5 | 18.4 | **8** | 0.3 | 0.77 |
| **9** | 2 | 28.0 | **9** | 0.2 | 0.22 |
| **10** | 3 | 25.8 | **10** | 0.1 | 0.22 |
| **11** | 1 | 5.2 | **11** | 0.1 | 0.33 |
| **12** | 1 | 7.6 | **12** | 0.05 | 0.55 |
| **13** | 1 | 3.2 | **13** | 0.1 | 0.44 |
| **14** | 2 | 11.6 | **14** | 0.1 | 0.66 |
| **15** | 2 | 8.8 | **15** | 0.2 | 1.25 |

The budget for violations removal satisfying these constraints is included in the minimization objective as

[8]Note, by this choice, the cost is different in the case of BD and CR: in principle, the cost of removing a defect is independent from the type of violation (BD or CR); however, engineers consider CR rules less related to defects (despite the mathematical correlation, which does not mean causation), as their role is different than bug detective rules. Therefore the $E_d$ is kept proportionally lower than in the case of BD violations.

$\sum_j \sum_i x_{i,j} * RE_j$; we also impose it to be not greater than 800 man-hours for BD and 800 man-hours for CR treatments (i.e., this means that solutions not satisfying the constraints within just 800 man-hours are infeasible; thus, they would require more relaxed constraints on achievable quality or more man-hours). SELEX engineers set up the following BD-related quality objectives, for the three models: *i)* an average number of violations ($\bar{v}_{max}$) reduced of at least 40% wit respect to the current average[9]; *ii)* a standard deviation $std_{max}$ reduced of at least 75% with respect to the current standard deviation along with an average reduced of at least 30% in this case; *iii)* at least the 70% of CSCIs with a number of violations less than 50% of the current average. As for CR rules: *i)* an average number of violations reduced of at least 25% with respect to the current one; *ii)* a standard deviation reduced of at least 30% with respect to the current one along with an average reduced of at least 20%; *iii)* at least the 70% of CSCIs with less than 25% than the current average.

**Evaluation Metrics**

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 $\sum_i v_{i,j} < v_{max}$. Note that the cost of non-removal may be considered as a measure of risk: in fact, since the cost of violation removal is given as input (i.e., it is the budget, 800 man-hours, and is always exploited fully by the algorithm), the remaining cost is only the non-removal one, which indirectly indicates the risk of having those residual violations. A higher cost means having residual violations with the greatest expected impact.

**Results**

Tables V and VI report the results of the experiment in terms of gain or reduction with respect to random case, as well as 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 provided solution is also able to satisfy the last target on the number of CSCI with $v < v_{max}$, even if not required. The standard deviation is also reduced considerably, being it 57% (for BD) and 28% (for CR) less than the original one, and 33% and 14% less than the random solution. 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 stander deviation, while also meeting the target on number of CSCI with $v < v_{max}$. 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 while also meeting the target on the average (not the one on the standard deviation). Results are very similar to the standard deviation model. Also in this case, a great reduction is devoted to standard deviation reduction.

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. This means favoring a type-aware removal of violations with respect to just reducing the number of violations. Conversely, whenever the targets are met with margin (e.g., in the T-A case), the tester can decide to spend less money and re-compute a solution with a lower available budget (e.g., 600 man-hours). Multi-objective models would also make sense. In any case, results enable the models as useful tools for quantitative reasoning and decision support in the code cleaning phase.

## V. Conclusion

This paper presented the results emerged from a study about static analysis performed on the industrial software systems developed by SELEX ES. The study revealed several practical implications of the systematic use of static analysis results for early detection of software defects and for the effective allocation of sanitization efforts on a lower set of most critical CSCIs. In addition, the developed models represent a tool for quantitative reasoning and decision support in the code cleaning phase before starting system tests and release the software. Presented results are the fruit of a research-industry collaboration between SELEX ES and the University of Naples Federico II. We attribute the success of this partnership to the following upfront choices: the primary factor for success is to have the commitment by key people in the company on a real concrete need, whose solution would bring tangible benefits in the future. However, while high-level objectives cannot produce tangible results soon (e.g., improving the V&V process), we found it useful to break a high-level goal into sub-goals whose outcomes are immediately visible to practitioners. This, in turn, is expected to foster practitioners to provide good-quality data and their support for further experiments, enabling a virtuous feedback loop between the parties. Moreover, we experienced as further essential requirement the guarantee that

[9]As mentioned in the introduction, we cannot disclose, for confidentiality reasons, the absolute number of violations.

TABLE V: Results of the allocation for BD rules compared to the random solution.

| Model | % Gain on Total # of Viol. to Remove | % Reduction of TOT$_{Cost}$ | % Reduction of Avg # Viol. [% *Reduction w.r.t. initial Avg*] | % Reduction Std of # Viol. [% *Reduction w.r.t. initial Std*] | Number (and %) of CSCI with $v < v_{max}$ |
|---|---|---|---|---|---|
| **T-A** | **+16.12%** | **-32.95%** | **-14.94%** [*-55.83%*] | **-33.15%** [*-57.68%*] | **122 (78.20% of CSCIs)** |
| **T-A & T-STD** | **+16.14%** | **-29.13%** | **-14.96%** [*-55.84%*] | **-59.61%** [*-75.43%*] | **110 (70.51% of CSCIs)** |
| **T-N** | **+16.82%** | **-25.12%** | **-15.6%** [*-56.17%*] | **-57.65%** [*-73.19%*] | **110 (70.51% of CSCIs)** |

TABLE VI: Results of the allocation for CR rules compared to the random solution.

| Model | % Gain on Total # of Viol. to Remove | % Reduction of TOT$_{Cost}$ | % Reduction of Avg # Viol. [% *Reduction w.r.t. initial Avg*] | % Reduction Std of # Viol. [% *Reduction w.r.t. initial Std*] | N. CSCI with $v < v_{max}$ |
|---|---|---|---|---|---|
| **T-A** | **+78.47%** | **-42.21%** | **-15.99%** [*-30.22-%*] | **-14.50%** [*-28.83%*] | **117 (75 %)** |
| **T-A & T-STD** | **+22.09%** | **-2.84%** | **-4.50%** [*-20.67%*] | **-21.65%** [*-34.79%*] | **110 (70.51 %)** |
| **T-N** | **+22.66%** | **-2.34%** | **-4.62%** [*-20.77%*] | **-22.89%** [*-35.82%*] | **110 (70.51%)** |

any introduced innovation (technique, method, model, tool) has, at least initially, a low impact on current processes – hence on the everyday work of employees – and a low "cost" (e.g., no additional work required to practitioners, for instance, to produce data, to run experiments, to customize a tool). In fact, instead of trying to change the process from the top, we found that starting from engineers' daily work and from exploiting the information available 'for free' is by far more useful. On the other hand, such an agile approach do not allow to fully exploit all the amount of information potentially available. This could lead to produce approximate results until more accurate information is available, and to repeat some steps depending on the quality of the data available at a given time. In this specific case, the method introduced was a good trade-off between the effort required and the potential benefits of the models on provided data. Also, being the models not bound to company-specific information, replicating this approach in a new company is straightforward, provided that the same success factors are ensured in a new scenario.

## ACKNOWLEDGMENT

This work has been partly supported by the SVEVIA (Innovative methods and techniques for Software VErification and ValIdAtion of near-realtime complex systems) Research Project funded by the Italian Ministry of Education and Research (Grant no.: PON02_00485_3487758).

## REFERENCES

[1] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *Software Engineering, IEEE Transactions on*, vol. 32, no. 4, pp. 240–253, April 2006.

[2] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[3] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[4] Static code analysis best practices. [Online]. Available: www.parasoft.com/StaticCodeAnalysis

[5] N. Rutar, C. Almazan, and J. Foster, "A comparison of bug finding tools for java," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, Nov 2004, pp. 245–256.

[6] F. Wedyan, D. Alrmuny, and J. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, April 2009, pp. 141–150.

[7] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 580–586.

[8] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, Sept 2008, pp. 277–286.

[9] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, "Preliminary results on using static analysis tools for software inspection," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, Nov 2004, pp. 429–439.

[10] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer, "On the relation between external software quality and static code analysis," in *Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE*, Oct 2008, pp. 169–174.

[11] G. Carrozza, R. Pietrantuono, and S. Russo, "Dynamic test planning: a study in an industrial context," *International Journal on Software Tools for Technology Transfer*, pp. 1–15, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10009-014-0319-0

[12] M. Lyu, S. Rangarajan, and A. van Moorsel, "Optimal allocation of test resources for software reliability growth modeling in software development," *IEEE Trans. on Reliability*, vol. 51, no. 2, pp. 336–347, 2002.

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

[14] G. Carrozza, M. Faella, F. Fucci, R. Pietrantuono, and S. Russo, "Engineering air traffic control systems with a model-driven approach," *IEEE Software*, vol. 30, no. 3, pp. 42–48, 2013.

[15] G. Carrozza, M. Faella, F. Fucci, R. Pietrantuono, and S. Russo, "Integrating mdt in an industrial process in the air traffic control domain," in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012, pp. 225–230.

[16] D. Cotroneo, R. Pietrantuono, and S. Russo, "Combining operational and debug testing for improving reliability," *Reliability, IEEE Transactions on*, vol. 62, no. 2, pp. 408–423, June 2013.

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

[18] U. D. of Defense, *Overview and Tailoring Guidebook*, MIL-STD 498, 1996.