A Framework to Compare Alert Ranking Algorithms

Simon Allier, Nicolas Anquetil, Andre Hora, Stephane Ducasse
RMoD Team
INRIA, Lille, France
{firstName.lastName}@inria.fr

***Abstract*—To improve software quality, rule checkers statically check if a software contains violations of good programming practices. On a real sized system, the alerts (rule violations detected by the tool) may be numbered by the thousands. Unfortunately, these tools generate a high proportion of "false alerts", which in the context of a specific software, should not be fixed. Huge numbers of false alerts may render impossible the finding and correction of "true alerts" and dissuade developers from using these tools. In order to overcome this problem, the literature provides different ranking methods that aim at computing the probability of an alert being a "true one". In this paper, we propose a framework for comparing these ranking algorithms and identify the best approach to rank alerts. We have selected six algorithms described in literature. For comparison, we use a benchmark covering two programming languages (Java and Smalltalk) and three rule checkers (FindBug, PMD, SmallLint). Results show that the best ranking methods are based on the history of past alerts and their location. We could not identify any significant advantage in using statistical tools such as linear regression or Bayesian networks or *ad-hoc* methods.**

I. Introduction

Rule checkers [2], [8], [9], [13]–[15], [19], [28], [30], [32] are tools that check whether some source code violates some good programming rules. For example, rules can specify an upper bound for method size [32], perform flow analysis to verify that a stream is properly closed [19], or check the correct use of an array to avoid out of bound accesses. These rules model good programming standards or frequent programming mistakes. Rule checkers thus raise alerts (or warnings) each time one of their rules is violated. These alerts must then be manually inspected by developers, to be eventually fixed. Correcting alerts improves the quality of the target system [31] and may prevent future bugs [18]. The systematic use of these tools facilitate and improves software maintenance [38].

However, not all alerts are corrected by the developers. Some may be ignored because it would be too difficult to fix them or because the developers do not agree that they represent an instance of bad code. Following Heckman *et al.* [17] terminology, when an alert is actually fixed by developers, we say it is an *actionable* ("true") alert, conversely, an un-actionable ("false") alert is one that does not require to be fixed, whatever the reason. On real systems, a rule checker may commonly raises thousands of alerts, many of which un-actionable. In fact, between 35% to 91% [5], [16], [21], [22], [24], [25] of reported alerts are un-actionable. In this case, the high rate of un-actionable alerts adversely impacts the review of the alerts and the identification of the actionable ones. Additionally, this suggests that the rules are wrong or do not make sense and may discourage developers to use these tools.

To overcome this problem, one solution is to use more complex rules and tools to increase the precision of the rule checkers [18], [31]. But even with more sophisticated analyses, the rate of un-actionable alerts can still be high [33]. Another solution is to add a filtering step by computing the probability of an alert being actionable in a given context. The filtering may also be performed on individual alerts or on rules, to identify the ones that, in the given context, are more likely to produce actionable alerts. Thus, by selecting the alerts or rules with the higher probability, the rate of un-actionable alerts can be reduced.

Despite a review of approaches for ranking alerts done by Heckman *et al.* [17], to our knowledge, these approaches were never compared with each other to understand the strengths and limits of each one. In this paper, we propose a framework for comparing different alert ranking approaches. It uses a benchmark covering two programming languages (Java and Smalltalk) and three rules checker (FindBug, PMD, and SmallLint). Six algorithms for the ranking of alerts are selected and compared using this framework, three ranking individual alerts and three ranking rules.

This paper is structured as follow. Section II discusses related work. Section III introduces the study design, detailing the research questions, the approach used to answer such questions and finally the threats to the validity of the experiments. In Section IV we analyze the results of the experiments according to the research questions. Finally, we conclude the paper in Section V.

II. Related Works

Different aspects of rules checker tools have already been studied.

The study closest to our work, by Heckman *et al.* [17], synthesizes available research results on ranking algorithms. Eighteen algorithms are detailed and described according to different criteria like information used, or algorithm used. One can see the study presented here as an extension of the one of Heckman *et al.* Indeed, in this paper we propose a framework for comparing the alerts ranking synthesis.

To compare the fault detection algorithms, various benchmarks have been proposed in the literature. BUGBENCH [26] is a benchmark containing seventeen buggy, open-source, C/C++ applications ranging from seven thousand lines of code (KLOC) to 1 MLOC in various domains. This benchmarck includes defects like buffer overflows, uninitialized reads, memory or semantic bugs but does not provide data on the alert from rule checkers. PROMISE [4] is a repository for data sets from empirical research in predictive modeling, and half of the 60 data sets are for anomaly prediction. However, most of the PROMISE data sets provide metrics without the project source, and some data sets refer to large, open-source projects while the remainder refers to commercial products. To our knowledge, FAULTBENCH [16] is the only benchmark precisely developed for comparing ranking algorithms for rule checkers. It contains three Java programs, alerts from the rule checkers FindBugs and PMD and a set of metrics on subject programs.

Other publications are more related to the pertinence of the domain. For example, Zheng *et al.* [38] are following the GQM[1] paradigm to determine whether rule checkers can help an organization to improve the economic quality of software products. Their results indicate that rule checkers are an economic complement to other verification and validation techniques.

Nagappan *et al.* [29] proposed an empirical approach for the early prediction of pre-release defect density based on the defects found using ASA. The defects identified by two tools, PREfix and PREfast, are used to fit and predict the actual pre-release defect density on the 199 component of Windows Server 2003. They concluded that warning density can be used to predict pre-release defect density at statistically significant levels and warning can be used to discriminate between components of high and low quality.

In recent years, some approaches have been proposed to study the relation between alerts and bugs. Such approaches are remotely connected to our study as one could consider that alerts that cause practical faults should be what we call actionable alerts (*i.e.*, alerts that are actually corrected when reported). Boogerd *et al.* [6], [7] empirically assess the relation between faults and violations of coding standard rules raised by MISRA C, using coding standard rules for embedded C development on industrial cases. The authors found that only 10 out of 88 rules for the case study presented in [7], and 12 out of 72 in rules for the case study presented in [6] were significant predictors of fault location.

Basalaj *et al.* [3] studied the link between QA C++ warnings and faults for snapshots from 18 different projects and found a correlation for 12 out of 900 rules. Wagner *et al.* [36] evaluated two Java bug-finding tools (FindBugs and PMD) on two different software projects, in order to evaluate their use in defect-detection. Their study could not confirm this possibility for their two projects. Couto *et al.* [10] also showed that overall there is no correspondence between the static location of the warnings raised by FindBugs and the methods changed by software maintainers in order to remove defects.

In [20], Araujo *et al.* study the relevance of the alerts reported by FindBug and PMD in several Java systems. They conclude that better relevance (less false positives) can be achieved when FindBugs is configured in a proper way, *i.e.*, when the tool is configured to report warnings that make sense for the system under analysis. Hora *et al.* [18] investigate the relation between generic or domain specific warnings (reported by SmallLint on a Smalltalk system) and observed defects. They have shown that domain specific rules provide better defect prevention than generic ones for the case study under analysis. This is has also shown by Renggli *et al.* [31].

Rutar *et al.* [33] studied the correlation and overlap between warnings generated by ESC/Java, FindBugs, JLint, and PMD. Their experimental results show that none of the tools strictly overlap another. In addition, there is little or no correlation between the warnings generated by these tools. This suggests that results for one tool may not be easily generalized to another and one needs to study each of them. In this paper, we studied three rule checkers (FindBugs, PMD, and SmallLint).

III. Experiment Description

A. Definition of the experiment

The first purpose of this study is to set up a framework to evaluate different alert ranking algorithms and find the "best" one. The notion of best algorithm is too vague, and needs to be refined. On one hand, rule checkers are typically noisy, they generate many false positive (un-actionable alerts). On the other hand, it would be illusory to expect these checkers to identify all bugs in a system [21]. Therefore, we chose to give more importance to the precision of the results than to their completeness. Thus, we prefer filtering methods that accept *only* actionable alerts over the ones that recognize *all* actionable alerts. This is measured by an effort metric that counts the average number of filtered alerts to (manually) inspect to find an actionable one. The effort metric is further detailed in Section III-D.

Q1 *Which algorithm has the best (lowest) effort?*

H_0^1 There are no differences between ranking methods effort.

H_a^1 Some ranking methods have a lower effort than other.

Assuming we have an answer to this research question, we may want to get into more details to better understand the differences between the different alert ranking methods.

First, ranking methods can be split into two categories, those working on rules (all the alerts of a rule are considered equally good), which are generally faster; and those working on alerts:

Q2 *Is it better to rank alerts on rules or on individual alerts?*

H_0^2 Ranking methods on rules and alerts have similar effort.

H_a^2 The effort of ranking methods is different for alerts and rules.

[1]Goal/Question/Metric, a process used to define a set of metrics to answer an abstract question.

To rank alerts, some methods use statistical tool such as Bayesian networks or linear regression. Implementing such tools may be complex and their results might be difficult to explain or understand. If "simpler" more intuitive methods are as successful, it could facilitate their implementation and/or adoption.

Q3 *Is there any difference between statistical and more* ad-hoc *alert ranking methods?*

H_0^3 Ranking methods using statistical algorithms give the same effort than *ad-hoc* methods.

H_a^3 The effort of ranking methods based on statistical models is different from the effort of *ad-hoc* ranking methods.

B. Subjects selection

To realize this experiment, we must select appropriate subjects. To compare the ranking alert methods we need a set of alerts already classified as actionable/un-actionable. We must also run the ranking methods on the subjects, which implies having access to all the required information, or being able to collect it.

1) Software systems: We first selected three datasets from the FaultBench 0.3 benchmark[2] [16]. It contains three real, Java programs for comparison and evaluation of alert ranking methods. The revision history of each subject covers a period of approximately seven years, from this, a number of revisions have been sampled by the authors of this benchmark. For each sample revision, FaultBench provides the alerts generated by FindBugs and PMD and also a set of metrics on the subjects.

We also generated three datasets from systems in Smalltalk for which we had access to the required data. Alerts were generated with the SmallLint rule checker (see below) and we extracted the information required to run the different alert ranking methods tested.

We shortly describe each system here. Table I also gives some data on them.

- **JDOM** is a Java-based solution for accessing, manipulating, and outputting XML data.
- **Runtime** is part of the Eclipse platform and provides support for the runtime platform.
- **Logging** is part of Apache Commons and provides a wrapper around a variety of logging API implementations.
- **Seaside-Core** is an open-source web application framework written in Smalltalk [12].
- **Pharo-Collection** is part of Pharo[3] kernel. It conposed of collections, such as arrayed or sorted collections, sets, dictionaries, bags, in Pharo.
- **Pharo-System** is also part of the Pharo kernel. It defines several system tasks, such as processes, number types, exceptions, and even the Smalltalk object model.

[2]http://www.realsearchgroup.org/faultbench

[3]Pharo is the Smalltalk dialect used in this work (www.pharo-project.org)

TABLE I
EXAMPLE SYSTEMS USED IN THE EXPERIMENTS. THE FIRST THREE ARE IN JAVA, THE LAST THREE IN SMALLTALK

	LOC (min–max)	init. rev.	final rev.	# rev.
JDOM	9 035–13 146	05/2000	12/2008	1 168
Runtime	2 066–15 516	05/2000	08/2001	1 324
Logging	355–1 785	08/2001	09/2008	710
Seaside-Core	4 235–9 507	11/2007	07/2011	943
Pharo-Collection	14 661–17 206	07/2006	08/2011	422
Pharo-System	24 985–34 044	03/2009	10/2011	555

2) Rules checkers: We use three different rule checkers, two for Java and one for Smalltalk.

FindBugs [19] is an open-source tool for Java with more than 360 rules (called bug patterns). They are written using BCEL, an open-source byte-code analysis and instrumentation library that works on the abstract syntax three (AST) of the source code. The rules can also rely on intra-procedural control and data flow analysis. The rules are classified in categories (such as correctness, performance, malicious code, bad practice, etc.) and priorities (high, medium, or low).

PMD[4] is an open-source tool that supports a rich set of rules for detecting potential bugs and checking coding style in Java. For example, PMD supports rules for detecting empty try/catch/finally/switch statements or dead code like unused local variables, parameters and private methods. There are also rules for dealing with particular frameworks, such as Java Beans, JSP, JUnit, Android etc. PMD requires the source code of the target program, because constraint rules are defined over the AST of the programs. Finally, as in FindBugs, rules have a priority and a category.

SmallLint [32] is a generic rule checker for Smalltalk. It accepts two types of rules: those working on the AST (similar to Findbug and PMD) and those using the reflectivity of Smalltalk by accessing the objects representing the classes and methods. For coherence with Findbug and PMD, we use only rules working on the AST. But our framework could have used the other rules too. Again SmallLint rules have a category (e.g., possible bug, unnecessary code, bug, etc.).

3) Classification of alerts: Alerts generated by the rule checkers are classified as (un-)actionable as follows: considering an alert appearing in an artifact (class, method, or package) in revision i. If, in revision $i+n$, the artifact still exists and the alert is removed (the artifact no longer contains it) then it is an *actionable alert*. An alert is *un-actionable* if the artifact is removed before the alert is corrected. Finally, if the alert and the artifact still exist in the latest revision of the benchmark, we can not determine whether the alert is actionable or not, and it is removed from the benchmark. Table II summarizes the number of alerts for each of the three rule checkers.

C. Treatments

For this experiment, the different treatments are the alert ranking methods applied. From the result of the rule checkers and other data collected on the system checked, these methods

[4]pmd.sourceforge.net

TABLE III
TYPE OF INFORMATION USED BY THE ALERT RANKING ALGORITHMS

	entities ranked	use of statistical approach	type of information used by the ranking algorithm: alert history	age of alert	alert artifact	size of code artifact	history of code artifact	rule of alert	rule severity or category
AWARE	alerts		✓		✓			✓	
FeedBackRank	alerts	✓	✓		✓			✓	
RPM	alerts	✓	✓	✓	✓	✓	✓	✓	✓
ZRanking	rules	✓				✓			
AlertLifeTime	rules			✓					
EFindBugs	rules		✓					✓	✓

TABLE II
NUMBER OF ALERTS IN THE BENCHMARK

	FindBugs	PMD	SmallLint
# rules	110	160	90
# alerts	1 611	25 867	22 860
Actionable alerts	588	4 447	3 908
Un-actionable alerts	903	13 373	2 761
Alerts not classified	120	8 047	16 191

try to filter out un-actionable alerts. This is done either by classifying them as (un-)actionable or by ranking them in decreasing probability of being actionable.

We selected five methods from the 18 presented by Heckman *et al.* [17] according to the following criteria:

- The ranking methods require only data from static analysis (they don't require to run the subject system). This is important to be able to use rule checkers in the early stages of the project development. This criterion excludes AJ06 [1] and CHECK' N' CRASH [11].
- Required data is available to us, either in the FaultBench benchmark or we can compute it for the Smalltalk subject systems. This excludes the method of Boogerd and Moonen [5] that uses call graphs, HWP [21] requiring commit messages and code changes, or SCAS [37] that relies on lexical analysis.
- The methods should be generic enough to work on any kind of rule. This excludes HISTORYAWARE that ranks only one rule (check return value rule) based on return information obtained in the system history.
- The methods must be able to run on the results of a single rule checker. This excludes MMW08 [27] and ISA [23] that compare and merge the results of several rule checkers.

To these five methods, we added EFindBugs [35] that is too recent to appear in Heckman *et al.* review. It respects the selection criteria.

We now present each of the six treatments. Some of them work on individual alerts, other work on rules, considering that all alerts from a rule are equally actionable or not. Table III summarizes some information on the alert ranking algorithms.

AWARE: Heckman and Williams [16] rank individual alerts using their rule and their location in the source code. An assumption of the model is that alerts from the same rule and/or from the same source code artifact are similarly actionable or un-actionable. AWARE ranks alerts on a scale from -1 to 1, where alerts close to -1 are more likely to be un-actionable and alerts close to 1 are more likely to be actionable. Alerts are ranked by considering the developer's feedback, via past actionable and un-actionable alerts, to generate a measure of the set of alerts sharing either the same rule or code location.

FeedBackRanking: Kremenek *et al.* [24] developed an adaptive prioritization algorithm based on the intuition that alerts sharing an artifact location (method, class and package) tend to be either all actionable or all un-actionable (similarly to AWARE). Each inspection of an alert by a developer adjusts the ranking of un-inspected alerts. After each inspection, the set of inspected alerts is used to build a Bayesian Network, which models the probabilities that groups of alerts sharing a location are actionable or un-actionable. In our case, the training set (inspected alerts) is static and we do not used feedback to improve the accuracy of this method.

RPM: Ruthruff *et al.* [34] use a logistic regression model to predict actionable alerts. Thirty-three alert characteristics are considered for the logistic regression model. Reducing the number of characteristics for inclusion in the logistic regression model is done via a screening process, whereby logistic regression models are built with increasingly larger portions of the alert set. Characteristics with a contribution lower than a specified threshold are thrown out until some minimum number of characteristics remains. In our case, we used only 18 characteristics of the 33 because some of them, like the *Google warning descriptors*, are specific to the context of their research.

ZRanking: Kremenek and Engler [25] proposed a statistical model for ranking rules for MC, a specific rule checking tool (not considered in our experiment). Unlike most other rule checkers, MC reports for each rules where it is satisfied (successful checks) and where it is not (failed checks). The ZRanking statistical method is based on this information and a "grouping operator" gathering together alerts that are likely of being equally actionable. The grouping operator use characteristics of the artifacts checked by the rules and/or of the alert, for example, call site, number of calls to free memory, function. A limitation of this technique is that the ranking success depends on the grouping operator. For our experiments we used a very simple one where alerts are grouped by their rule.

Also, because we use FindBugs, PMD, and SmallLint rule checkers (see Section III-B) that don't output successful/failed checks, we approximate this result. Considering a rule that

applies on methods, for example to check whether a call to a given primitive is correctly performed. All alerts reported by our rule checkers are failed checks; all other methods are considered successful checks which obviously might not be true.

AlertLifeTime: Kim and Ernst [22] prioritize rules by the average lifetime of alerts in this rule. The idea is that alerts fixed quickly are more important to developers. In our case, the lifetime of an alert is measured at the file level for Java and, in Smalltalk, from the revision of this rule's first alert until closure of its last alert. Alerts still existing in the last revision studied[5] are given a penalty of 365 days added to their lifetime.

EFindBugs: In [35] Shen *et al.* present EFindBugs that prioritizes rules by using their defect likelihood. It is computed as the ratio of the actionable alerts over the un-actionable alerts for each rule. Periodically in the system's lifetime, new sets of (un-)actionable alerts are recomputed and the defect likelihood of each rule is updated accordingly. The initial set of classified alert may be obtained from another system or from the alert history of the current system.

D. Experimental Design and Analysis

We provide in this section a brief description and justification of the analysis procedure that we used.

The six alert ranking algorithms are implemented in Smalltalk. To compute the binomial regressions and Bayesian network necessary for the algorithms [24], [34], we use the R statistical language/system.

The datasets are split in two: a training set and a test set. Each set contains 50% of the benchmark alerts. Then, each algorithm is trained on the training set and used on the test set. The results are compared which the correct results computed as described in Section III-B3. This process (training set generation, algorithm training, result testing) is repeated 100 times for each ranking algorithm to avoid bias. The results presented in this article are the average of the 100 executions. Additionally, we also take care that training set and test set have the same percentage of (un-)actionable alerts as the whole dataset. This helps us comparing the different alert ranking methods by ensuring that they all learn from training datasets with similar characteristics and then all run on test datasets with similar characteristics.

To evaluate the ranking algorithms, different measures have been proposed in the literature. We use the measure of effort proposed in [25]. Alerts are ordered in decreasing probability of being actionable or not. *Effort* is computed for the first $x\%$ of the alerts output by a ranking algorithm, and it is equal to the average number of alerts one must inspect to find an actionable one. The best value of effort, for a perfect alert ranking algorithm, would be 1, meaning that any alert picked in the first $x\%$ is actionable, or all $x\%$ alerts are actionable. An effort of 2 tells that one out of two alerts is actionable. As alerts are ranked in decreasing order of probability of being actionable, one expects that, as $x\%$ increases, the effort will also increase because their will be more and more un-actionable alerts in the results.

We will also use the Fault Detection Rate Curve. For a given algorithm, the curve is formed by the percentage of all actionable alerts found within the first y alerts of the alert ranking algorithm. For an optimal alert ranking algorithm, when y is less than the total number of actionable alerts, the curve raises linearly from 0% to 100% as all extracted alerts are actionable (in Figure 1, this is materialized by the upper dotted line). When y is equal or greater than the total number of actionable alerts, the curve is constant with value 100%. In the figures, we also materialize a random alert ranking algorithm (lower dotted line) that achieves 100% only when all the alerts are considered.

We compare the alert ranking algorithms between themselves using these curves. For this we compare the curves two by two with a Chi-squared test of homogeneity (χ^2). This test allows us to determine if two curves follow a different distribution. When the p-value of the test is less than a given threshold (we use 5% and 1‰) we can conclude that the two curves do not follow a similar distribution. Note that the comparison is not symmetric, but in practice, comparing A to B rarely leads to a different result than comparing B to A.

E. Threats to Validity

1) Internal validity: Our implementation of the ZRanking algorithm is rough. The grouping operator implemented is overly simple and the successful/failed check behavior could only be approximated as the rule checking tools used do not output successful checks. The results of this ranking algorithm might be different from the original implementation.

2) External validity: We tried to be as generic as possible by selecting a range of systems from different domains and two programing languages. The systems are real-world and non-trivial application, with a consolidated number of users. Data are collected for a large number of versions over an extended time frame (several years).

3) Conclusion validity: We did not test the statistical validity of the difference in the results of the effort metric. For this reason we tried to be very conservative in drawing conclusions from these results and back them up with the χ^2 test comparing the fault detection rate curves.

IV. Analysis Result

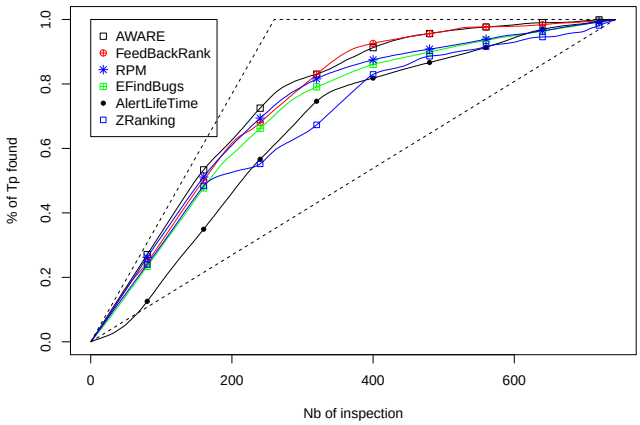
In this section, we analyze the results of the experiment according to the research questions presented in the previous section.

A. Q1: What is the best alert ranking method?

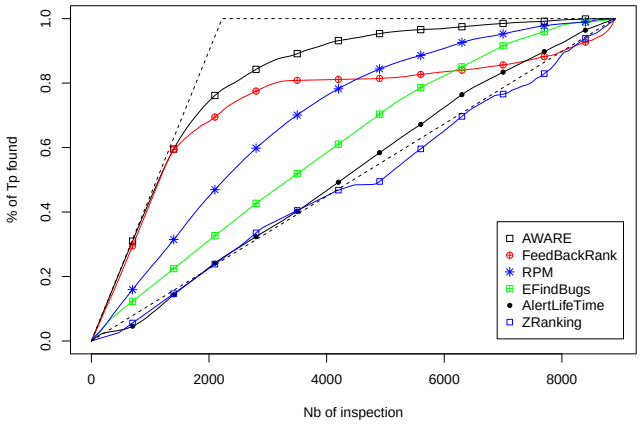
This question was formalized as: Q1 – Which algorithm has the lowest effort?

The results are given in Table IV: effort for the first 20%, 50%, or 80% of all alerts. Figure 1 also presents all the fault detection rate curves, and Table V gives the results for the

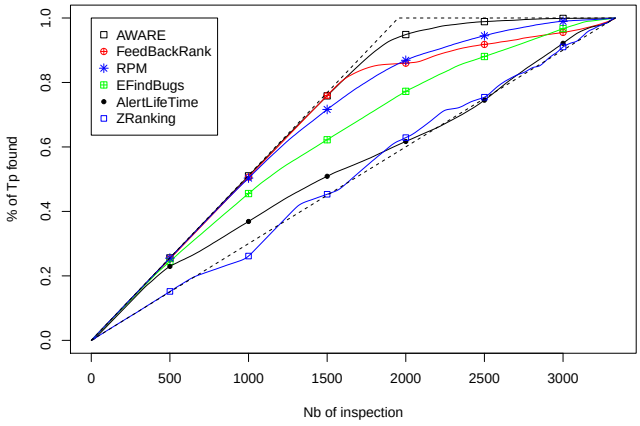
[5]To train the algorithm we don't eliminate undecidable alerts as was explained above (Section III-B3).



(a) FindBugs



(b) PMD



(c) SmallLint

Fig. 1. Fault detection rate curves of the alert ranking algorithms tested for the three rule checkers. Dotted lines represent a random ranking algorithm (lower dotted line) and a perfect ranking one (upper dotted line).

TABLE IV
RESULTS FOR THE EFFORT METRIC FOR THE FIRST 20%, 50%, AND 80%. BOLD VALUES SHOW THE BEST RESULTS (CLOSER TO 1).

	PMD			FindBugs			SmallLint		
Threshold	20%	50%	80%	20%	50%	80%	20%	50%	80%
AWARE	**1.01**	**1.03**	**1.36**	**1.13**	**1.14**	**1.35**	**1.01**	**1.02**	**1.03**
FeedBackRank	1.08	1.04	1.75	1.19	1.22	1.45	**1.01**	**1.02**	1.04
RPM	1.9	2.02	2.46	1.17	1.19	1.47	**1.01**	**1.02**	1.11
ZRanking	4.05	4.44	4.2	1.27	1.3	1.84	1.71	1.69	1.67
AlertLifeTime	4.06	3.79	3.72	2.05	1.65	1.76	1.04	1.49	1.69
EFindBugs	2.7	3.01	3.22	1.3	1.28	1.58	1.02	1.14	1.35

TABLE V
RESULTS OF THE χ^2 TEST OF DIFFERENCE BETWEEN ALL FAULT DETECTION RATE CURVES (PRESENTED IN FIGURE 1). (*) CURVES ARE DIFFERENT AT THE 5% LEVEL (P-VALUE ≤ 0.05) ; (**) CURVES ARE DIFFERENT AT THE 1‰LEVEL (P-VALUE ≤ 0.001. ABBREVIATIONS: A=AWARE; FBR=FEEDBACKRANK; ZR=ZRANKING; ALT=ALERTLIFETIME; EFB=EFINDBUGS.

	A	FBR	RPM	ZR	ALT	EFB
FindBugs						
A	-	3	8	28**	66**	13
FBR	3	-	16	38**	73**	22*
RPM	8*	14	-	12	43**	2
ZR	26**	34**	10	-	42**	8
ALT	60**	64**	31**	33**	-	20**
EFB	13	20*	2	10	26**	-
PMD						
A	-	289**	82**	436**	379**	203**
FBR	283**	-	149**	373**	332**	223**
RPM	71**	172**	-	111**	90**	22*
ZR	334**	235**	91**	-	5	41**
ALT	294**	214**	73**	6	-	32**
EFB	153**	224**	18*	49**	42**	-
SmallLint						
A	-	28**	10	127**	88**	39**
FBR	25*	-	5	81**	51**	11
RPM	10	6	-	83**	50**	11
ZR	142**	66**	82**	-	28*	49**
ALT	95**	42**	48**	32**	-	16*
EFB	38**	9	11	60**	18*	-

homogeneity tests between the curves. In Table V results of the χ^2 tests are marked with (*) when the difference between the curves is statistically significant at the 5% level, and marked with (**) when the difference is statistically significant at the 1‰ level.

First, we observe that the AWARE algorithm has consistently the lowest effort (see Table IV). Its effort is typically close to 1. Even when one inspects almost all (80%) of the alerts ranked by the algorithm the effort is still only about 1.33 which means that 3 out of 4 inspected alerts are actionable.

FeedBackRank also shows good results and is consistently the second best.

On the other end of the spectrum, we have the ZRanking and AlertLifeTime algorithms with efforts going up to 4.44 (only 1 out of 4 inspected alerts is actionable). The two alert ranking algorithms present the interesting characteristic that the effort may decrease when one inspects more alerts. This means that the actionable alerts are not ranked first, but after the 20% or 50% first. One can see in Figures 1(b) and 1(c) that ZRanking can even perform worse than a random classifier (*i.e.*, its fault detection rate curve is under the lower dotted line).

Such results for the ZRanking algorithm may be explained by the simple grouping operator used in our implementation (see Section III-C) and because of the approximation of the successful/failed check behavior.

Figure 1 and Table V allow one to statistically test the differences between the alert ranking methods. One must note that results depend on the used rule checker. For PMD, all alert ranking algorithms are different two by two, except for the ZRanking and AlertLifeTime algorithms. For FindBugs and SmallLint, we can not draw conclusions taking into account their pairs.

In summary, for two test bed (SmallLint and PMD), the Aware and FeedBackRank algorithms are significantly better than the other alert ranking methods while the ZRanking and AlertLifeTime algorithms are significantly worse.

B. Q2: Is it better to rank alerts on rules or on individual alerts?

A first answer comes from the fact that from the three ranking algorithms working on individual alerts (Table III), two (Aware and FeedBackRank) were found to be better than the other methods. Moreover, from the three algorithms working on rules, two (ZRanking and AlertLifeTime) were found to be worse.

To analyze these results with more details, we plot the cumulative percentage of actionable alerts in the rules in figures 2, 3, and 4. The *x* axis represents rules with a given percentage of actionable alerts. Thus, the left side of the *x* axis shows the "bad rules" with low percentage of actionable alerts, and the right side of the *x* axis shows the "good rules", with high percentage of actionable alerts.

In FindBugs and SmallLint test beds (Figures 2 and 3), the bar plot raises steadily, whereas for the PMD test bed (Figure 4), it raises very fast at the beginning. This means for PMD that 80% of all actionable alerts are raised by "bad rules" (with less than 35% of their alert being actionable). For FindBugs, this means that less than 50% of all actionable alerts come from reasonably "good rules" (up to 85% of their alerts are actionable). Therefore, alert ranking algorithms working on rules give poor results for PMD, while for FindBugs and SmallLint the results are better.

Additionally, we explain the staged shape of the ZRanking fault detection rate curves by the grouping of alerts (see definition of ZRanking). Because of the defective grouping operator used, when a group comes in, its actionable alerts raises the curve steeply. Then when the other (un-actionable) alerts of the same group are considered, the curves remain stable (no improvement) until the next group comes in. This shows the consequence of grouping alerts together.

In conclusion, overall one should probably opt for ranking algorithms that work on individual alerts. However, on a practical point of view, it would be interesting to check whether the training on one system can easily be ported to another one. Because of the fine grained training at the level of individual alerts, on new systems without history, it might be better to start with an algorithm trained (on some other system) at the level of rules?

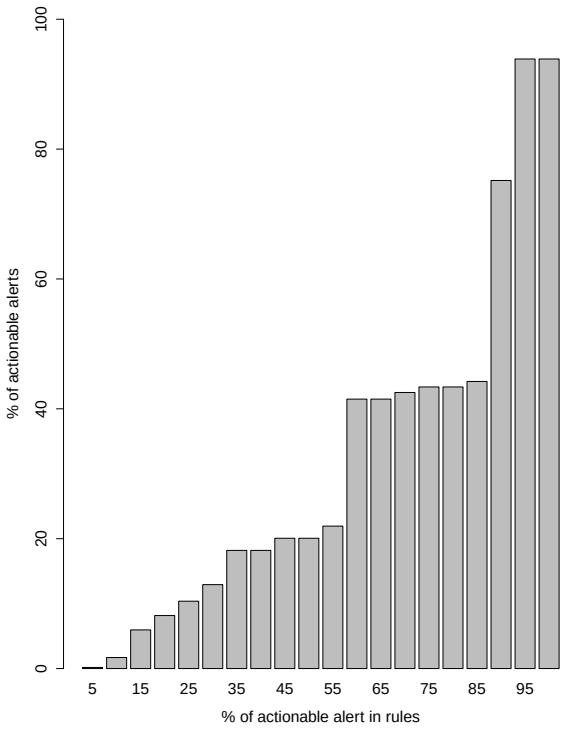


Fig. 2. % of actionable alerts in FindBugs rules

C. Q3: Is there any difference between statistical and more ad-hoc *alert ranking algorithms?*

Among the two best ranking algorithms (AWARE and FeedBackRank), one is *ad hoc* and the other is based on a statistical algorithm. Similarly, among the two worst ranking algorithms (ZRanking and AlertLifeTime), one is *ad hoc* and the other is based on a statistical algorithm. Therefore it seems difficult to clearly decide between the two approaches based on these data. From our experience, we can only say that the *ad-hoc* algorithms were easier to implement and usually required less information. We therefore suggest to go with one of these (typically AWARE) if one has a choice.

D. Additional Comments

All ranking methods do not require the same information. For instance, the RPM algorithm requires lot of data about the alerts and obtaining such information is difficult and expensive.

As shown in Table III, the two best ranking algorithms (FeedBackRank and AWARE) use exactly the same type of data: the artifact which contains the alert, the rule of the alert, and the actionable and un-actionable alerts. In addition, the best ranking algorithm working at rule-level (EFindBugs) also uses the actionable and un-actionable alerts as well as the rule of the alert. In such ranking algorithms, the history of past

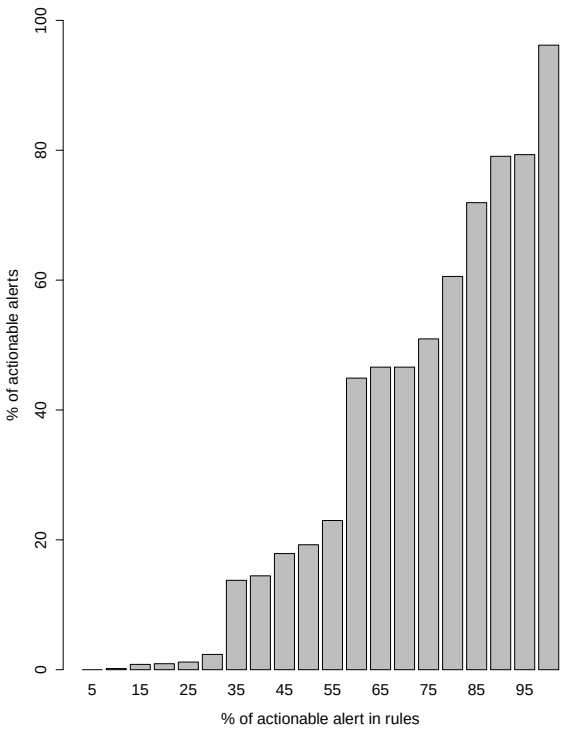


Fig. 3. % of actionable alerts in SmallLint rules

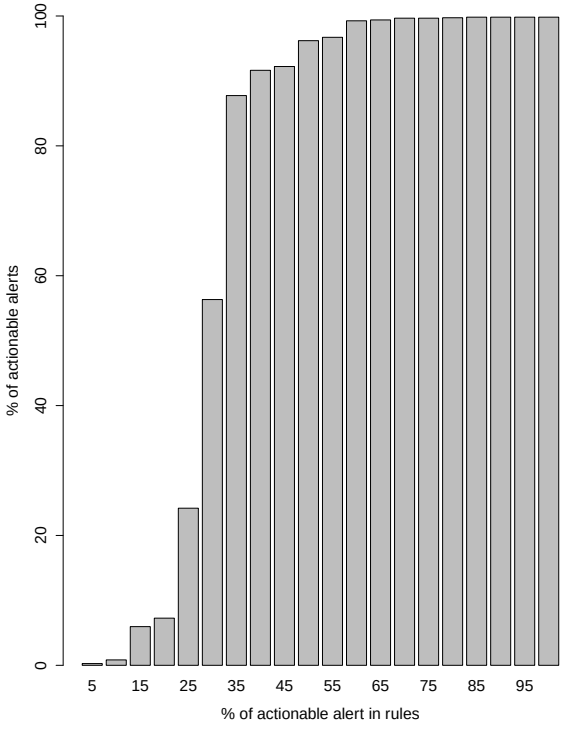


Fig. 4. % of actionable alerts in PMD rules

alerts is used to determine the ratio of actionable alert/un-actionable alert for the system code artifact and the rules of the rule checker.

One might hypothesize first that not all data sources have the same relevance in ranking alerts or rules, and second that history of past alerts and the location of the alerts (artifact containing the alert) are probably good sources.

Looking again at the results in Table III, one may notice that SmallLint has results consistently better than the two others, or that FindBugs lead to consistently better results than PMD for ranking algorithms working on rules (see also Section IV-B).

One might hypothesize that all rule checkers do not behave equally. This was expected from the discussion at the end of Section II. More experiments could be undertook to understand why and how these differences occur, whether due to the tool themselves or to the rules they contain.

V. Conclusion

In this paper, we propose a framework for comparing six alert ranking algorithms and identify the best conditions to separate actionable from un-actionable alerts. We selected six alert ranking algorithms described in the literature and identified some of their characteristics that could be meaningful in this comparison:

- three algorithms rank individual alerts and three rank rules;
- three algorithms are *ad-hoc* and three are based on a statistical approach;
- All algorithms do not use the same input data.

For comparison, we use a benchmark covering two programming languages (Java and Smalltalk), three rule checkers (FindBug, PMD, SmallLint) and six different systems.

The conclusions of our experiments are:

- All ranking algorithms are not equally efficient, particularly AWARE is the best alert ranking algorithm on our benchmark, closely followed by FeedBackRank;
- Ranking algorithms working on alerts give better results than the ones working on rules. This implies that rules are not inherently good are bad but depend on the context. We did not try to qualify better how the context influences the results.
- We could not identify any significant advantage in the results of *ad-hoc* or statistical approaches. We, however, suggest hat the *ad-hoc* solutions are easier to implement and might give results easier to understand.

We also identified possible future research directions regarding the respective relevance of different data sources used by the ranking algorithms or the difference in result yielded by the various rule checking tools.

References

[1] A. Aggarwal and P. Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *International Computer Software and Applications Conference*, pages 343–350, 2006.

[2] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proceedings of the 13th Australian Conference on Software Engineering*, pages 68–, 2001.

[3] W. Basalaj and F. van den Beuken. Correlation Between Coding Standards Compliance and Software Quality. Technical report, Programming Research, 2006.

[4] G. Boetticher, T. Menzies, and T. Ostrand. Promise repository of empirical software engineering data. Technical report, West Virginia University, Department of Computer Science, 2007.

[5] C. Boogerd and L. Moonen. Prioritizing Software Inspection Results using Static Profiling. In *International Workshop on Source Code Analysis and Manipulation*, pages 149–160, 2006.

[6] C. Boogerd and L. Moonen. Assessing the Value of Coding Standards: An Empirical Study. In *International Conference on Software Maintenance*, pages 277 –286, 2008.

[7] C. Boogerd and L. Moonen. Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions. In *Working Conference on Mining Software Repositories*, pages 41–50, 2009.

[8] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software–Practice & Experience*, 30:775–802, jun 2000.

[9] B. Chelf, D. Engler, and S. Hallem. How to Write System-specific, Static Checkers in Metal. *SIGSOFT Software Engineering Notes*, 28(1):51–60, nov 2002.

[10] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente. Static Correspondence and Correlation Between Field Defects and Warnings Reported by a Bug Finding Tool. *Software Quality Journal*, pages 1–17, 2012.

[11] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *Transactions on Software Engineering and Methodology*, 17(2):1–37, may 2008.

[12] S. Ducasse, L. Renggli, C. D. Shaffer, R. Zaccone, and M. Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010.

[13] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19:42–51, 2002.

[14] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna - A Static Model Checker. In *FMICS/PDMC*, pages 297–300, 2006.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Conference on Programming language design and implementation*, pages 234–245, 2002.

[16] S. Heckman and L. Williams. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. In *International Symposium on Empirical Software Engineering and Measurement*, pages 41–50, 2008.

[17] S. Heckman and L. Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53:363–387, apr 2011.

[18] A. Hora, N. Anquetil, S. Ducasse, and S. Allier. Domain specific warnings: Are they any better? In *International Conference on Software Maintenance*, page *to appear*, 2012.

[19] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[20] S. S. Joao Araujo Filho and M. T. Valente. Study on the Relevance of the Warnings Reported by Java Bug-finding Tools. *Software, IET*, 5(4):366 –374, aug 2011.

[21] S. Kim and M. D. Ernst. Which Warnings Should I Fix First? In *European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, pages 45–54, 2007.

[22] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting Faults from Cached History. In *International Conference on Software Engineering*, pages 489–498, 2007.

[23] D. Kong, Q. Zheng, C. Chen, J. Shuai, and M. Zhu. Isa: a source code static vulnerability detection system based on data fusion. In *International Conference on Scalable Information Systems*, pages 1–7, 2007.

[24] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation Exploitation in Error Ranking. In *International Symposium on Foundations of Software Engineering*, pages 83–93, 2004.

[25] T. Kremenek and D. Engler. Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *International Conference on Static Analysis*, pages 295–315, 2003.

[26] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[27] N. Meng, Q. Wang, Q. Wu, and H. Mei. An approach to merge results of multiple static analysis tools (short paper). In *International Conference on Quality Software*, pages 169–174, 2008.

[28] C. Morgan, K. De Volder, and E. Wohlstadter. A Static Aspect Language for Checking Design Rules. In *International Conference on Aspect-oriented Software Development*, pages 63–72, 2007.

[29] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *International Conference on Software Engineering*, pages 580–586, 2005.

[30] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. SABER: Smart Analysis Based Error Reduction. *SIGSOFT Software Engineering Notes*, 29(4):243–251, jul 2004.

[31] L. Renggli, S. Ducasse, T. Girba, and O. Nierstrasz. Domain-Specific Program Checking. In *Objects, Models, Components, Patterns*, pages 213–232. Springer-Verlag, 2010.

[32] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3:253–263, 1997.

[33] N. Rutar, C. B. Almazan, and J. S. Foster. A Comparison of Bug Finding Tools for Java. In *International Symposium on Software Reliability Engineering*, pages 245–256, 2004.

[34] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting Accurate and Actionable Static Analysis Warnings: an Experimental Approach. In *International Conference on Software Engineering*, pages 341–350, 2008.

[35] H. Shen, J. Fang, and J. Zhao. EFindBugs: Effective Error Ranking for FindBugs. In *Software Testing, Verification and Validation*, pages 299–308, 2011.

[36] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An Evaluation of Two Bug Pattern Tools for Java. In *International Conference on Software Testing, Verification, and Validation*, pages 248–257, 2008.

[37] S. Xiao and C. Pham. Performing high efficiency source code static analysis with intelligent extensions. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 346–355, Washington, DC, USA, 2004. IEEE Computer Society.

[38] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *Transactions on Software Engineering*, 32(4):240–253, apr 2006.