Assessing the Value of Coding Standards: An Empirical Study

Conference Paper · November 2008

DOI: 10.1109/ICSM.2008.4658076 · Source: IEEE Xplore

2 authors, including:

Leon Moonen
Simula Research Laboratory
170 PUBLICATIONS **6,123** CITATIONS

Assessing the Value of Coding Standards: An Empirical Study*

Cathal Boogerd
Software Evolution Research Lab
Delft University of Technology
The Netherlands
c.j.boogerd@tudelft.nl

Leon Moonen
Simula Research Laboratory
Norway
Leon.Moonen@computer.org

Abstract

In spite of the widespread use of coding standards and tools enforcing their rules, there is little empirical evidence supporting the intuition that they prevent the introduction of faults in software. Not only can compliance with a set of rules having little impact on the number of faults be considered wasted effort, but it can actually result in an increase in faults, as any modification has a non-zero probability of introducing a fault or triggering a previously concealed one. Therefore, it is important to build a body of empirical knowledge, helping us understand which rules are worthwhile enforcing, and which ones should be ignored in the context of fault reduction. In this paper, we describe two approaches to quantify the relation between rule violations and actual faults, and present empirical data on this relation for the MISRA C 2004 standard on an industrial case study.

1. Introduction

Long have coding standards been used to guide developers to a common style or to prevent them from using constructions that are known to be potentially problematic. For instance, Sun publishes its internal Java coding conventions, arguing that they help new developers to more easily understand existing code and reduce maintenance costs [16]. Another example is the MISRA C standard [17], specifically targeted towards safe use of C in critical systems. The rules in these standards are typically based on expert opinion, and can be targeted towards multiple goals, such as reliability, portability or maintainability. In this paper, we focus on reliability, i.e., the prevention of faults by the adoption of coding rules. In this context, the MISRA standard is of particular interest, as it places a special emphasis on automatic checking of compliance. The rationale for automated checking is that it is of little use to set rules if they cannot be properly enforced.

Such a check typically takes the form of a static source code analyzer, in which well-known academic options (e.g., [10, 5, 6], for a description and more tools see [4]) and commercial tools (e.g., QA-C,[1] K7,[2] and CodeSonar[3]) abound. These tools are equipped with a pre-defined set of rules, but can often be customized to check for other properties as well. As such, when defining a coding standard for a certain project, one can use the existing tools as well as writing one's own. In a recent investigation of bug characteristics, Li et al. argued that such early automated checking has contributed to the sharp decline in memory errors present in software [15]. Thus, with rules based on long years of experience, and support for automated checking readily available, universal adoption of coding standards should be just a step away. But is it?

Developers often have less sympathy for these tools, since they typically result in an overload of non-conformance warnings (referred to as violations in this paper). Some of these violations are by-products of the underlying static analysis, which cannot always determine whether code violates a certain check or not. Kremenek et al. [14] observed that all tools suffer from such false positives, with rates ranging from 30-100%. In addition to the noise produced by the static analysis, there is the noise of the ruleset itself. For many of the rules in coding standards there is no substantial evidence of a link to actual faults. Moreover, coding standard necessary limit themselves to generic rules, that may not always be applicable in a certain context. These problems constitute a barrier to adoption of both standard and conformance checking tool.

Apart from barring adoption, there is an even more alarming aspect to having noisy rules (i.e., leading to many violations unrelated to actual faults) in a standard. Adams [1] first noted that any fault correction in software has a non-zero probability of introducing a new fault, and if this probability exceeds the reduction achieved by fixing the violation, the net result is an increased probability of faults in the software. Enforcing conformance to noisy rules can thus *increase* the number of faults in the software. As Hatton remarks when discussing the MISRA 2004 standard: "the false positive rate is still unacceptably high with the accom-

* This work has been carried out in the Software Evolution Research Lab at Delft University of Technology as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

[1] www.programmingresearch.com
[2] www.klocwork.com
[3] www.grammatech.com

panying danger that compliance may make matters worse not better" [8].

Clearly, it is of great interest to investigate if there is empirical evidence for the relation between coding standard rule violations and actual faults. This relation is especially important when, rather than striving for full compliance in order to meet certain regulations, the body of rule violations is used as a means to assess the software reliability (i.e., the number of dormant faults), as is increasingly becoming popular in industry. Therefore, the goal of this paper is:

> *To empirically assess the relation between violations of coding standard rules and actual faults*

Where we define a *violation* to be a signal of non-conformance of a source code location to any of the rules in the standard; and a *fault* to be an issue present in the issue tracking system, which may be related to multiple locations in the source code. In the process of this investigation, we make the following contributions:

- We propose two *methods* to quantify the relation between violations and faults, and describe their requirements with respect to the richness of the input data set (i.e., the software history).
- We present *empirical data* obtained from applying both methods to an industrial embedded software project and the MISRA C 2004 coding standard. The data consists of correlations between rule violations and faults, as well as measured true positive rates for individual rules.

We describe the two approaches and the measurement setup in Section 2, and the particular project and standard under study in Section 3. The resulting empirical data is presented in Section 4. We evaluate the results and the two methods in Section 5. Finally, we compare with related work in Section 6 and summarize our findings in Section 7.

2. Methodology

To assess the relation between violations and faults, we take two approaches. The first is a more high-level approach, studying the co-evolution of violations and faults over time, testing for the presence of a *correlation*. The second approach looks at individual violations in more detail, tracking them over time and seeing how often they correctly signal actual problem locations, giving us a *true positive rate* for every rule. We will discuss both methods in more detail, after which we describe how to obtain the necessary measures.

2.1. Temporal coincidence

To quantify the relation between rule violations and actual faults we look at the software history present in a Software Configuraton Management (SCM) system and the accompanying issue tracking system, further referred to as Problem Report (PR) database. Using the version information, we can reconstruct a timeline with all open PRs as well as the rule violations. If violations truthfully point at potential fault areas, we expect more violations for versions with more faults present. Also, the reverse holds: if a known fault was removed in a certain version, it is probable that violations, disappearing in that same version, were related to the fault. This covariation can then be observed as a positive correlation between the two measures. Of course, this approach results in some noise because also violations disappearing during modifications not related to fixing faults are included in the correlation.

2.2. Spatial coincidence

The noise present in the previous approach can be partially eliminated by requiring proximity in *space as well as time*. If we use the data present in the SCM system and PR database to link PRs with their related changes in the version history, we can determine which changes were fault-fixing. Violations on lines that are changed or deleted in non-fix modifications can thus be ruled out as likely false positives. With such information, we can determine for every rule how many violations were accurately pointing out a problem area, the true positives.

However, the prime prerequisite for this approach is also its Achilles' heel. Establishing a link between faults and source code modifications requires either integrated support by the SCM system, or a disciplined development team, annotating all fix changes with informative commit messages (e.g., referring to the identifier of the problem in the PR database). Such information may not always be present, which limits the applicability of the method. This the reason why we have chosen to use both methods on a case: it will allow us to use the results of the second to assess the impact of the noise present in the first. If the noise impact is small, we can safely use the first method on a wider array of cases, expanding our possibilities to gather empirical data.

2.3. Measurement approach

To gather the measurements for a large software history we need an *automated* measurement approach. This raises a number of challenges, and we will discuss for each measure how we have solved those. Summarizing from the previous sections, the measures needed in this investigation are the number of violations per version, the number of open (i.e., unsolved) PRs per version, and the true positive rate for all rules. The number of open PRs per version is probably easiest, as we can simply use the status field for the PR to determine whether it was unsolved at the moment the version was released or built. Determining the number of violations per version requires a bit more work, and comprises

the following steps:

- Retrieving a full version from the SCM. Since we will run a static analyzer on the retrieved source code, this needs to be a proper, compilable set of source files, as such tools often mimick the build environment in order to check compliancy.
- Extracting configuration information necessary to run the compliancy checker. This kind of information is usually present in Makefiles or project files (e.g., Visual Studio solution files).
- Running the compliancy checker using the extracted configuration information, saving the rule violations and their locations (file name, line number) within the source archive.

To determine the true positive rates we use a process similar to the one used in the warning prioritization algorithm by Kim and Ernst [11]. However, whereas they track bug-fix lines throughout the history, we track the violations themselves. Tracking violations over different versions, although not too difficult, is not widely addressed. We are only aware of the work by Spacco et al. [20].

In our approach, we build a complete version graph for every file in the project using the predefined successor and predecessor relations present in the SCM. The version graph allows us to accurately model branches and merges of the files. For every edge in the graph, i.e., a file version and one of its successors, the diff is computed. These differences are expressed as an annotation graph, which represents a mapping of lines between file versions.

Violations are uniquely identified by triplets containing the file name, line number and rule identifier. Using the version graph and the annotation graph associated with every edge, violations can be tracked over the different versions. As long as a violation remains on a line which has a destination in the annotation graph (i.e., is present the next file version) no action is taken. When a line on which a violation has been flagged is changed or removed, the PR database is consulted to see if the modification was fix-related. If it is, the score for the rule is incremented; in both cases, the total incidence count for the current rule is incremented. Violations introduced by the modification are considered to constitute a new potential fault, and are tracked similar to the existing ones. In the last version present in the graph, the number of remaining violations are added to the incidence count (on a per rule basis), as these can all be considered false positives.

3. Case Description

To the best of our knowledge, there is no previous complete empirical data from a longitudinal study on the relation between violations and faults. As a result, we do not even know whether assessing this relation is feasible. We therefore decided to use one typical industrial case as a pilot study. NXP (formerly Philips Semiconductors), our industrial partner in the TRADER [4] project, was willing to provide us with tooling and access to its TV on Mobile project.

3.1. TV on Mobile

This project (TVoM for short) was selected because of its relatively high-quality version history, featuring the well-defined link between PRs and source code modifications we need for our spatial coincidence method. It represents a typical embedded software project, consisting of the driver software for a small SD-card-like device. When inserted into the memory slot of a phone or PDA, this device enables one to receive and play video streams broadcasted using the Digital Video Broadcast (DVB) standard.

The complete source tree contains 148KLoC of C code, 93KloC C++, and approximately 23KLoC of configuration items in perl and shell script (all reported numbers are non-commented lines of code). The real area of interest is the C code of the actual driver, which totals approximately 91KLoC. Although the policy of the project was to minimize compiler warnings, no coding standard or code inspection tool was used. This allows us to actually relate rule violations to fault-fixing changes; if the developers would have conformed to the standard we are trying to assess, they would have probably removed all these violations rightaway.

3.2. MISRA-C: 2004

The first MISRA standard was defined by a consortium of UK-based automotive companies (The Motor Industry Software Reliability Association) in 1998. Acknowledging the widespread use of C in safety-related systems, the intent was to promote the use of a safe subset of C, given the unsafe nature of some of its constructs [7]. The standard became quite popular, and was also widely adopted outside the automotive industry. In 2004 a revised version was published, attempting to prune unnecessary rules and to strengthen existing ones. However, Hatton [9] argued that even these modifications could not prevent the standard from having many false positives among reported violations, so “a programmer will not be able to see the wood for the trees”. Currently, NXP is also introducing MISRA-C: 2004 as the standard of choice for new projects. Given the wide adoption of the MISRA standard, an assessment of its rules remains a relevant topic. Copyright laws prevent us from quoting the MISRA rules themselves. However, a discussion on the content of these rules is beyond the scope of this paper.

[4] www.esi.nl/trader

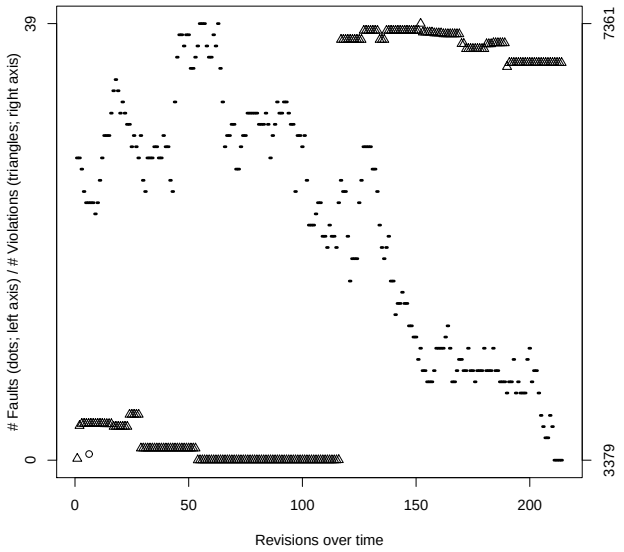


Figure 1. Number of faults and violations over time

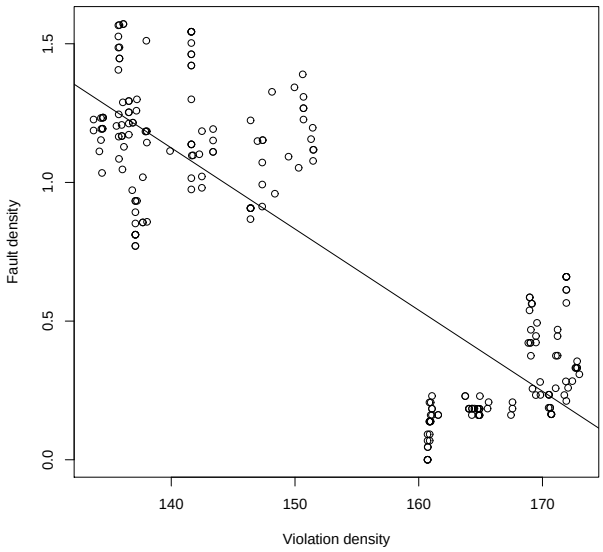


Figure 2. All MISRA violations versus open PRs

3.3. Experimental details

The complete version history of the project runs from February 2006 until June 2007, but up to August 2006 the organization and structure of the project was volatile and the version history (as a result) erratic. We therefore selected the period from August 2006 onwards as input to our analysis. It contains 214 versions, related to both daily builds and extra builds in case the normal daily build failed. In addition, we selected PRs from the PR database that fulfilled the following conditions: (1) classified as 'problem' (thus excluding change requests); (2) involved with C code; and (3) had status 'solved' by the end date of the project.

NXP has developed its own front-end to QA-C 7, dubbed Qmore, that uses the former to detect MISRA rule violations. Configuration information needed to run the tool (e.g., preprocessor directives or include directories) is extracted from the configuration files (Visual Studio project files) driving the daily build that also reside in the source tree.

The SCM system in use at NXP is CMSynergy, which features a built-in PR database next to the usual configuration management operations. The built-in solution allows linking PRs and tasks, which group conceptually related source code modifications. Using this mechanism, we are able to precisely extract the source lines involved in a fix.

4. Results

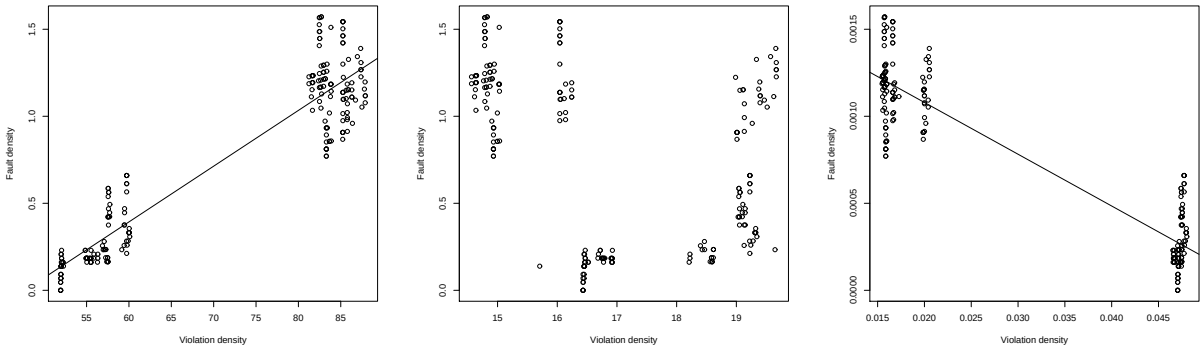
The results in this section comprise two different parts: first we look at some evolution data of the project, to rule out any abnormalities or confounding factors that might distort our measurements; in the second part we examine the results of the rule violation and fault relation analysis.

4.1. Project evolution data

In Figure 1 the number of open PRs for every version (using dots; left y-axis) is plotted. Since we have not included the first half year of the project, the number of open PRs starts at 27, steadily increasing until it almost hits 40, then jumps down (in February 2007) until it reaches 0 by the end of the project history. This is typical behavior in a project like this, where at first the focus is on implementing all features of the software, and only the most critical PRs are solved immediately. Only when the majority of features has been implemented, a concerted effort is made to solve all known problems.

The rule violation history, also in Figure 1 (using triangles; right y-axis), does not quite follow this trend. At the point where the number of faults decreases sharply, the number of violations actually increases. Also, the gap in the number of violations is striking. After manual inspection we found that this is due to one very large header file (8KLoC) being added at that point. In addition, the effect in the graph is emphasized because of the partial scale of the vertical axis.

Clearly, with the size of the software (LoC) having such a profound effect on the number of rule violations, an analysis of the relation between violations and faults should rule out this confounding factor. To this end, we will only use fault and violation *densities*, the numbers per version divided by the number of KLoC for that version. Figure 2 shows the relation between violation density and fault density for the complete set of MISRA rules. In this figure, every point represents one version of the software. If there were a positive correlation, we would see the points clustering around the diagonal to some extent. However, quite the contrary can be observed: the correlation for the MISRA standard as a



(a) positive relation (b) no relation (c) negative relation

Figure 3. Relation between violations and faults for different subsets

whole looks negative rather than positive.

4.2. Analysis of individual rules

Now that we have seen the correlation for MISRA as a whole, we will look more closely at the influence of individual rules. To do this, we create graphs, similar to Figure 2, but using only violations for a single rule. These are visually inspected to determine whether there exist a correlation between the two measures. If the distribution is acceptable, we classify the rule as either 'positive' or 'negative', if there are too many outliers it is classified as 'none'. The resulting classification for all rules can be observed in columns 2-4 of Table 1. This table does not include all rules in the MISRA standard, as we could only include those for which we have observed violations (in total 72 out of 141 rules).

In addition, we use a least squares linear regression model to summarize the characteristics of the correlation. Columns 5-7 display the properties of this model: the linear coefficient, or the slope of the fitted line; the percentage of variation explained by the model, or the square of the correlation coefficient R; and the statistical significance, expressed as the probability that the correlation found is simply coincidence. Significance has been presented in star-notation, with 'ns' for $p > 0.05$, $p < 0.05$ *, $p < 0.01$ **, $p < 0.001$ *** and $p < 0.0001$ ****. In general, we can observe that the linear models for rules in the classes 'positive' and 'negative' explain a large part of the variation; $R^2 > 0.75$.

From an automation point of view, it may be appealing to simply rely on the summary statistics, such as the correlation coefficient R. However, this does not guarantee existence of a true relation, as the input distribution may, for instance, contain too many outliers, something which can easily be ascertained using the aforementioned scatter plots [2]. Due to space limitations, we have not included scatter plots for all individual rules, but have plotted aggregated results for each class in Figure 3. These figures also display the fitted line of the regression model.

Another way in which we verified that the data does not violate the assumptions of the regression was by inspecting residual plots. One example is displayed in Figure 4 (for the 'positive' set). The upper left graph contains the data and the fitted line; the upper right plots residuals versus fitted, which should be spread with no apparent relation around the horizontal axis. Lower left is a histogram of the residuals, which should be a normal distribution, and lower right a normal plot of the residuals, where they should be laid out more or less on the diagonal. In this case, although slightly skewed, the distribution is acceptable.

The next two remaining columns in Table 1 display the total number of violations and the true positive rate for each rule. The violations are the number of unique violations over the complete observed history. That is, if exactly the same violation was present in two versions, it was counted only once. The true positive rate has been computed by dividing the number of violations on lines changed or deleted in fix-changes by the total number of violations. The corresponding per-class data are displayed in Table 2 in a manner similar to Table 1.

If we see rule violations as a line-based predictor of faults, the true positive rate can be considered a measure of the prediction accuracy. Consequently, when comparing rule 1.1 and 2.3 this rate suggests the former to be more accurate. However, in a population of 62 violations (for 1.1) it is likely that there are some correct violations simply due to chance, which is unlikely if there are few violations. Therefore, true positive rates cannot be used in their own right to compare rules on accuracy (unless the number of violations is the same).

We can assess the significance of the true positive rate by comparing it to a random predictor. This predictor randomly select lines out of a population of over 300K unique lines in the TVoM history, of which 17% was involved in a fix change. Since the population is so large, and the number of violations per rule relatively small (max. 0.5% of the

MISRA rule	Positive	None	Negative	Linear coefficient	Variation explained (R^2)	Significance	Total violations	True positives (ratio)	Prediction performance
1.1			√	-1.07	0.86	****	62	0.08	0.04
1.2	√			3.92	0.77	****	26	0.23	0.86
2.3	√			53.13	0.86	****	1	0.00	0.83
3.1		√		12.80	0.46	****	11	0.00	0.13
3.4		√		0.98	0.17	****	15	0.00	0.06
5.1			√	-19.43	0.85	****	2	0.00	0.69
5.2			√	-9.10	0.85	****	10	0.00	0.16
5.3	√			21.10	0.83	****	2	0.50	0.97
5.6		√		-0.56	0.57	****	278	0.08	0.00
6.1			√	-19.43	0.85	****	2	0.00	0.69
6.2			√	-2.43	0.85	****	16	0.00	0.05
6.3	√			0.06	0.82	****	1675	0.10	0.00
8.1	√			1.69	0.79	****	52	0.35	1.00
8.4		√		13.85	0.24	****	1	0.00	0.83
8.5		√		18.41	0.42	****	2	1.00	1.00
8.7			√	-7.97	0.82	****	46	0.30	0.99
8.8	√			0.51	0.84	****	115	0.10	0.03
8.10	√			0.83	0.80	****	90	0.01	0.00
8.11		√		-4.27	0.08	****	4	0.25	0.86
9.1		√		0.80	0.06	***	15	0.27	0.90
9.2		√		-3.90	0.01	ns	2	1.00	1.00
9.3			√	-4.55	0.85	****	12	0.00	0.11
10.1	√			0.36	0.79	****	490	0.13	0.02
10.6			√	-0.13	0.85	****	378	0.02	0.00
11.1		√		3.08	0.39	****	38	0.26	0.95
11.3			√	-1.92	0.86	****	108	0.06	0.00
11.4		√		-0.43	0.37	****	151	0.06	0.00
11.5		√		0.59	0.11	****	26	0.08	0.16
12.1	√			0.33	0.85	****	151	0.08	0.00
12.4		√		-2.07	0.16	****	19	0.11	0.35
12.5			√	-0.71	0.75	****	110	0.05	0.00
12.6			√	-4.72	0.84	****	11	0.00	0.13
12.7	√			1.40	0.87	****	91	0.24	0.97
12.8	√			1.21	0.83	****	68	0.09	0.04
12.10			√	-38.86	0.85	****	1	0.00	0.83
12.13	√			5.25	0.79	****	37	0.27	0.96
13.1			√	-0.39	0.85	****	121	0.01	0.00
13.2			√	-0.09	0.85	****	781	0.06	0.00
13.7			√	-2.51	0.84	****	26	0.12	0.33
14.1		√		-0.11	0.02	*	191	0.04	0.00
14.2	√			0.27	0.89	****	260	0.30	1.00
14.3			√	-0.11	0.85	****	818	0.01	0.00
14.5			√	-15.28	0.84	****	7	0.00	0.27
14.6			√	-0.74	0.82	****	70	0.01	0.00
14.7			√	-0.35	0.79	****	427	0.03	0.00
14.8			√	-0.17	0.85	****	282	0.02	0.00
14.10		√		4.58	0.21	****	53	0.09	0.09
15.2			√	-11.91	0.84	****	3	0.00	0.57
15.3			√	-6.29	0.87	****	15	0.00	0.06
15.4			√	-14.22	0.87	****	7	0.00	0.27
16.1		√		-13.26	0.27	****	15	0.00	0.06
16.4		√		1.04	0.03	*	64	0.12	0.22
16.5	√			2.50	0.82	****	20	0.15	0.55
16.7			√	-0.69	0.84	****	143	0.08	0.00
16.8	√			11.13	0.84	****	2	0.00	0.69
16.9	√			53.13	0.86	****	2	0.50	0.97
16.10			√	-0.10	0.85	****	1845	0.06	0.00
17.4			√	-0.39	0.78	****	205	0.09	0.00
17.6			√	-38.86	0.85	****	1	1.00	1.00
18.4		√		3.10	0.79	****	19	0.05	0.14
19.4		√		-3.05	0.32	****	37	0.05	0.04
19.5	√			10.63	0.86	****	5	0.00	0.39
19.6	√			10.63	0.86	****	5	0.00	0.39
19.7			√	-5.03	0.81	****	37	0.03	0.01
19.10		√		-1.83	0.09	****	37	0.00	0.00
19.11		√		8.03	0.79	****	5	0.00	0.39
20.2		√		-3.56	0.57	****	22	0.00	0.02
20.4		√		-1.23	0.06	***	39	0.00	0.00
20.9		√		5.92	0.57	****	16	0.00	0.05
20.10		√		13.85	0.24	****	1	0.00	0.83
20.12		√		-8.40	0.44	****	16	0.19	0.72
21.1			√	-2.80	0.84	****	25	0.08	0.18

Table 1. Relation between violations and faults per rule

population), we can model this as a Bernoulli process with $p = 0.17$. The number of succesful attempts or correctly predicted lines has a binomial distribution; using the cumulative density function (CDF) we can indicate how likely a rule is to outperform the random predictor in terms of accuracy. This value is displayed in the last column of the table, and can be compared across rules. Note that none of the three classes in Table 2 have an average true positive rate higher than the expected value of the random predictor (0.17). As a result, the value of the CDF for all three classes is near-zero.

5. Evaluation

In this section, we will discuss the meaning of the results, as well as the applicability and limitation of the methods used

Rule class	Linear coefficient	Variation explained (R^2)	Significance	Total violations	Average violations	True positives (ratio)
Positive (18)	0.03	0.87	****	3092	171.78	0.13
None (25)	-0.10	0.15	****	1077	43.08	0.07
Negative (29)	-0.02	0.85	****	5571	192.10	0.05

Table 2. Aggregated statistics per class

to obtain them.

5.1. Meaning of the correlations

The first important moderation for our results is the fact that they are correlations, and therefore not *proof* of causality. However, the positive and negative relations observed explain a large amount of the variation ($R^2 > 0.75$) and are statistically significant. Then what do they mean?

A positive correlation is rather intuitive: we oberve an increasing violation density during the first phase of the project, where most of the faults are introduced. An increase in violation *density* means, that the areas that developers have been working on violate the rule relatively often. The decrease of violation density in the second phase of the project signals that many have been removed at a time when there was a focus on fixing faults. Possibly, some violations were introduced in unrelated areas during the first phase, but seeing them removed in the second phase strengthens the intuition that they were indeed related to a fault.

Similar to rules with positive correlations, rules with negative correlations experience a relative high rate of injection, not just in the first phase, but also in the second phase. For a negative correlation to occur, the violation density for this rule must keep increasing, even when the fault density decreases. Since the the changes are relatively small compared to the complete codebase (on average only adding 9 lines), injection of a single violation without removal somewhere else already results in an increased density. In other words, violations of these rules are seldom removed: apparently, the areas they reside in were not considered relevant to fixing faults.

Intuitively, the magnitude of the coefficient of the linear model should be seen as a quantification of the ratio of true positives. When looking closer, however, we see that the coefficient is large (either positive or negative) in cases of a rule with few violations, as a small number of violations is compared to a greater number of faults. This small number of violations makes it more difficult to distinguish a

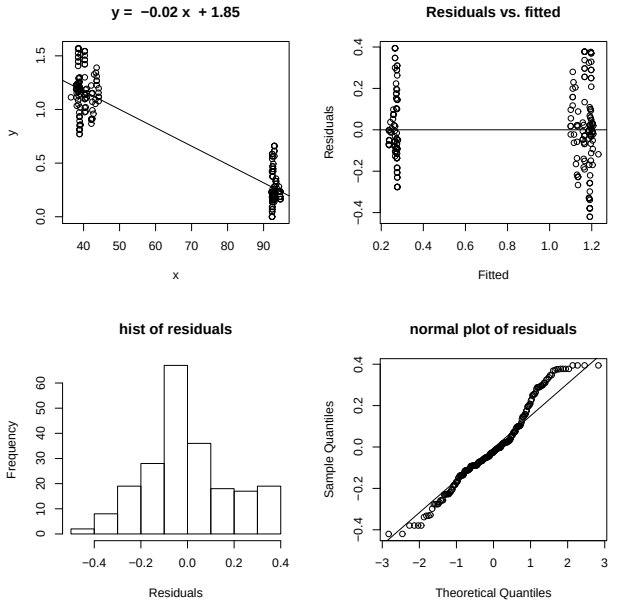


Figure 4. Residual plots for the positive class

relation, which is why many of those rules have been classified as 'none' in Table 1. This actually explains all but rules 5.6, 11.4 and 14.1 in this category, for which we did observe a large amount of violations. Re-examination of the correlation plots revealed distributions of points that suggest a correlation, except for a cluster of close outliers. This was the result of files being moved in and out of the main branch of development, these files containing a relatively high amount of violations of the particular rule. Although we try to compensate for this by using densities, in some cases the influence of one file can still distort an otherwise strong correlation.

5.2. Correlations and true positive rates

Overall, the true positive rates computed by also requiring spatial coincidence partially match the partitioning of the rules into classes using only temporal coincidence. This can be seen in Table 2, where the 'positive' class shows a higher true positive rate than both others; and 'none', in turn, higher than 'negative'. However, there are some individual rules whose inconsistency between class and measured true positive rate warrant further explanation. Two different categories of interest can be distinguished:

Class positive, yet zero true positive rate This category (e.g., rule 3.1) illustrates the extra precision gained by also requiring spatial coincidence. These rules have violations removed at the same time faults were removed from the software, and as a result, show a positive correlation. Spatial coincidence shows that, as these removals were not due to a fix-change, they are not likely to be related to the fault removal.

Class negative, yet non-zero true positive rate This category encompasses 18 rules out of the negative class, which itself contains a total of 29. Although at a first glance, having both a negative correlation and violations removed during fixes may be paradoxal, this can be understood by looking at the rate of removal as well as the rate of injection. Despite continuous removal (in fixes as well as non-fix related changes), the violation density keeps increasing, as more violations are being injected by all those changes. Consequently, the measured true positive rates also depend on the point of time at which they are measured (e.g., beginning, halfway or end of the project).

The true positive rates indicate that 15 out of 72 rules outperform a random predictor with respect to selecting fault-related lines. If we only select rules whose true positive rate is significantly different ($\alpha = 0.05$) from the expected value (0.17), 12 rules remain (those where the performance value ≥ 0.95). This means that the results of the two approaches agree on only 7 out of the 18 rules in class 'positive'. Since we can consider the true positive rates a more accurate characterization, it seems unwise to use the correlations as the *only* means to assess the relation between violations and faults.

However, as we mentioned before, the change in violations over time remains relevant. For instance, in the TVoM project, the most critical faults were removed before the project was suspended, but during maintenance more faults might have been found, altering the measured true positive rate. In addition, the character of the software written may change over time (e.g., from more development in the hardware layer to the application layer), which could also affect the number and type of violations in the software.

Finally, another criterion to assess the true positive rate worth mentioning is the rate at which faults are injected when violations are fixed. A non-zero probability means that the use of at least 25 out of 72 rules with a zero true positive rate is invalidated. Although the injection rate is not known for this project, in Adams' study, this probability was 0.15 [1]. This value would even invalidate 57 out of 72 MISRA rules. This lends credence to the possibility that adherence to the complete MISRA standard would have had a negative impact on the number of faults in this project.

5.3. Threats to validity

Although some of these issues have been mentioned elsewhere in the text, we summarize and discuss them here for clarity's sake.

Internal validity There is one measure in our case study that may suffer from inacurracies. The fault density for a given version has been measured using the number of open PRs for that version. Since a PR may only be submitted some time after the fault was actually introduced, this results in a 'lagging' measured fault density compared to the actual fault density. The number of violations is measured on the source that contains this 'hidden' fault, and as a result, the correlation between violation density and fault density may be slightly underestimated.

In addition, the correlations as used in this paper are sensitive to changes in the set of files between the different versions. These changes occur since in case of a failed daily build, the culprit module (i.e., a set of files) will be excluded from the build until repaired, to ensure a working version of the project. Those changes result in the different clusters of points in the scatter plots in Section 4. To see why, consider the following example. Suppose that for a certain rule r, most of the violations are contained in a small subset S_r of all the files in the project. Even if there would be a perfect linear relationship between violations of r and the number of faults, then this would not be visible in a scatter plot if files in S_r are alternatingly included and excluded from subsequent versions of the project. Two clusters of versions would appear; in one, there is no relation, while the other suggests a linear relation. Clearly, this distorting effect only increases as more rules and versions are considered. In future investigations we intend to address this issue by looking at official internal releases instead of daily build versions. This will minimize changes in the composition of files because of the aforementioned build problems. In addition, we intend to split the analysis between phases of the project that are distinctly different with respect to the composition of files.

The spatial coincidence method produces conservative estimates, for two reasons. Underestimation might occur with bug fixes that introduce new code or modify spatially unrelated code. Assume that the static analysis signals that some statement, requiring a check on input data, could be reached without performing such a check. Adding the check to solve the problem would then only introduce new code, unrelated to the existing violation. Violations addressing multiple lines are unlikely to occur with generic coding standards, which typically contain rules at the level of a single expression or statement (especially ones based on the safer subset paradigm, such as MISRA). Therefore, impact of this problem remains small, as there are usually no violations to link the additional code to. Another reason is the way we handle violations remaining at the end of the project. Those violations might point to faults that simply have not been found yet. But we since we do not *know* this, we take a conservative approach and assume they are not. The influence of dormant faults is somewhat mitigated in the case of a long-running project, where most of the faults will have been found. Moreover, we can expect that in short-term projects at least the most severe faults have been solved, so the relation between violations and the most critical problem areas in the code can still be assessed.

External validity Generalizing the measured correlations and true positive rates for the TVoM project to arbitrary other projects may not be easy. In fact, our investigation was partly inspired by the intuition that such relations would differ from project to project. Still, it may be possible to generalize results to other projects within NXP, since (1) all projects use a similar infrastructure and development process; (2) developers are circulated across projects, which lessens the impact of individual styles; and (3) all projects concern typical embedded C development, likely to suffer from some common issues. The only problem in this respect may be that the platform for which the embedded software is developed requires idiosyncratic implementation strategies, violating some coding rules but known to be harmless.

Although the approaches used in this paper are generic in nature, some specific (technical) challenges need to be faced when applying them to an arbitrary project. First of all, as mentioned before, linking known faults and source modifications made to fix them requires a rich data set. Although lately, many studies have succesfully exploited such a link [15, 19, 13, 21, 12, 22, 18], we found that in our industrial settings, most project histories did not contain the necessary information. This limits the applicability of the spatial coincidence approach. Second, in order to accurately measure the number of rule violations and known faults, it is important to precisely define which source code entities are going to be part of the measurements. Source files for different builds may be interleaved in the source tree, so selection may not always be a trivial matter. For instance, some source files can be slighly altered when built for different platforms, so this may influence the number of measured violations. In addition, some faults may be present in the program when built for one platform but not for others, thus the selection may also influence the measured number of faults. Finally, the inspection tool used to detect violations can also influence results, as it might produce false positives, i.e., signalling a violation when in fact the corresponding code complies with the rule. Unfortunately, some inaccuracy in (complex) static analysis is unavoidable, and the extent may differ from one implementation to the next.

6. Related Work

In recent years, many studies have appeared that take advantage of the data present in SCM systems and bug databases. These studies exhibit a wide array of applications, ranging from an examination of bug characteristics [15], techniques for automatic identification of bug-introducing changes [19, 13], bug-solving effort estimation [21], prediction of fault-prone locations in the source [12], and identification of project-specific bug-patterns, to be used in static bug detection tools [22, 18].

Of all those applications, closest to our work is the history-based warning prioritization approach by Kim and Ernst [11]. This approach seeks to improve the ranking mechanism for warnings produced by static bug detection tools. To that end, it observes which (classes of) warnings were removed during bug-fix changes, and gives classes with the highest removal rate (i.e., the highest true positive rate) the highest priority. The approach was evaluated using different Java bug detection tools on a number of open-source Java projects. Although they do use the version history as input to the algorithm, the empirical data reported (true positive rates) covers only warnings of a single version of every project. In addition, there is a potential issue with using true positive rates for comparing rules on accuracy if the number of warnings for those rules are different (as discussied in Section 4.2). Another difference with our study is the application domain: we assess a coding standard for embedded C development on an industrial case.

While we report on data obtained from a longitudinal study of a single project, Basalaj [3] uses versions from 18 different projects at a single point in time. He computes two rankings of the projects, one based on warnings generated by QA C++, and one based on known fault data. For certain warnings, a positive rank correlation between the two can be observed. Unfortunately, the paper highlights 12 positively correlated warning types, and ignores the negatively correlated ones (reportedly, nearly 900 rules were used). Apart from these two studies, we are not aware of any other work that reports on measured relations between coding rules and actual faults.

As many studies as exist using software history, so little exist that assess coding standards. The idea of a safer subset of a programming language, the precept on which the MISRA coding standard is based, was promoted by Hatton [7]. In [8] he assesses a number of coding standards, introducing the signal to noise ratio for coding standards, based on the difference between measured violation rates and known average fault rates. The assessment of the MISRA standard was repeated in [9], where it was argued that the update was no real improvement over the original standard, and "both versions of the MISRA C standard are too noisy to be of any real use". The methods we introduce in this paper can be used to specialize a coding standard for a certain project, so as to make use of them in the best way possible. In addition they can be used to build a body of empirical data to assess them in a more general sense, and the data presented here are a first step towards that goal.

7. Conclusions

The contributions of this paper are (1) a description and comparison of two approaches to quantify the relation between coding rule violations and faults; and (2) empirical data on this relation for the MISRA standard in the context

of an industrial case.

From the data obtained, we can make the following key observations. First, there are 12 out of 72 rules for which violations were observed that perform significantly better ($\alpha = 0.05$) than a random predictor at locating fault-related lines. The true positive rates for these rules range from 23-100%. Second, we observed a negative correlation between MISRA rule violations and observed faults. In addition, 25 out of 72 rules had a zero true positive rate. Taken together with Adams' observation that all modifications have a non-zero probability of introducing a fault [1], this makes it possible that adherence to the MISRA standard as a whole would have made the software less reliable. This observation is consistent with Hatton's earlier assessment of the MISRA C 2004 standard [9].

These two observations emphasize the fact that it is important to select accurate and applicable rules. Selection of rules that are most likely to contribute to an increase in reliability maximizes the benefit of adherence while decreasing the necessary effort. Moreover, empirical evidence can give substance to the arguments of advocates of coding standards, making adoption of a standard in an organization easier. However, correlations and true positive rates as observed in this study may differ from one project to the next. To increase confidence in our results, and to investigate if we can distinguish a consistent subset of MISRA rules positively correlated with actual faults, we intend to repeat this study for a number of projects. In addition, we intend to address some of the issues encountered when using correlations, as discussed in Section 5.

Acknowledgements The authors wish to thank the people of NXP for their support in this investigation and the anonymous reviewers for their valuable feedback.

References

[1] E. N. Adams. Optimizing Preventive Service of Software Products. *IBM J. of Research and Development*, 28(1):2–14, 1984.

[2] F. J. Anscombe. Graphs in Statistical Analysis. *The American Statistician*, 27(1):17–21, 1973.

[3] W. Basalaj. Correlation between coding standards compliance and software quality. White paper, Programming Research Ltd., 2006.

[4] C. Boogerd and L. Moonen. Prioritizing Software Inspection Results using Static Profiling. In *Proc. Sixth IEEE Intnl. Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 149–158. IEEE, 2006.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th Symp. on Operating Systems Design and Implementation*, pages 1–16, October 2000.

[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, 2002.

[7] L. Hatton. *Safer C: Developing Software in High-integrity and Safety-critical Systems*. McGraw-Hill, New York, 1995.

[8] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information & Software Technology*, 46(7):465–472, 2004.

[9] L. Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information & Software Technology*, 49(5):475–482, 2007.

[10] S. C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual*, volume 2A, chapter 15, pages 292–303. Bell Laboratories, 1978.

[11] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proc. 6th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Intnl. Symp. on Foundations of Software Engineering*, pages 45–54. ACM, 2007.

[12] S. Kim, T. Zimmermann, E. James Whitehead Jr., and A. Zeller. Predicting Faults from Cached History. In *Proc. 29th Intnl. Conf. on Software Engineering (ICSE)*, pages 489–498. IEEE, 2007.

[13] S. Kim, T. Zimmermann, K. Pan, and E. James Whitehead Jr. Automatic Identification of Bug-Introducing Changes. In *Proc. 21st IEEE/ACM Intnl. Conf. on Automated Software Engineering (ASE)*, pages 81–90. IEEE, 2006.

[14] T. Kremenek, K. Ashcraft, J. Yang, and D.R. Engler. Correlation Exploitation in Error Ranking. In *Proc. 12th ACM SIGSOFT Intnl. Symp. on Foundations of Software Engineering (FSE)*, pages 83–93. ACM, 2004.

[15] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33. ACM, 2006.

[16] Sun Microsystems. Code Conventions for the Java Programming Language, April 1999.

[17] Motor Industry Software Reliability Association (MISRA). Guidelines for the Use of the C Language in Critical Systems, October 2004.

[18] S., K. Pan, and E. James Whitehead Jr. Memories of bug fixes. In *Proc. 14th ACM SIGSOFT Intnl. Symp. on Foundations of Software Engineering (FSE)*, pages 35–45. ACM, 2006.

[19] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Intnl. Workshop on Mining Software Repositories (MSR)*. ACM, 2005.

[20] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *Proc. Intnl. Workshop on Mining Software Repositories (MSR)*, pages 133–136. ACM, 2006.

[21] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How Long Will It Take to Fix This Bug? In *Proc. Fourth Intnl. Workshop on Mining Software Repositories (MSR)*, page 1. IEEE, 2007.

[22] C. C. Williams and J. K. Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.