Delft University of Technology
Software Engineering Research Group
Technical Report Series

Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions

Cathal Boogerd and Leon Moonen

Report TUD-SERG-2009-008

TU Delft

SERG

TUD-SERG-2009-008

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

© copyright 2009, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions*

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. In previous work, we performed a pilot study to assess the relation between rule violations and actual faults, using the MISRA C 2004 standard on an industrial case. In this paper, we investigate three different aspects of the relation between violations and faults on a larger case study, and compare the results across the two projects. We find that 10 rules in the standard are significant predictors of fault location.

1. Introduction

Coding standards have become increasingly popular as a means to ensure software quality throughout the development process. They typically ensure a common style of programming, which increases maintainability, and prevent the use of potentially problematic constructs, thereby increasing reliability. The rules in such standards are usually based on expert opinion, gained by years of experience with a certain language in various contexts. Over the years various tools have become available that automate the checking of rules in a standard, helping developers locate potentially difficult or problematic areas in the code. These include commercial offerings (e.g., QA-C,[1] K7,[2] CodeSonar[3]) as well as academic solutions (e.g., [12, 5, 7]). Such tools generally come with their own sets of rules, but can often be adapted so also custom standards can be checked automatically. In a recent investigation of bug characteristics, Li et al. argued that early automated checking has contributed to the sharp decline in memory errors present in software [18]. However, in spite of the availability of appropriate standards and tools, there are several issues hindering adoption.

Automated inspection tools are notorious for producing an overload of non-conformance warnings (referred to as violations in this paper). For instance, 30% of the lines of the project used in this study contained such a violation. Violations may be by-products of the underlying static analysis, which cannot always determine whether code violates a certain check or not. Kremenek et al. observed that all tools suffer from such false positives, with rates ranging from 30-100% [17]. Furthermore, rules may not always be appropriate for all contexts, so that many of their violations can be considered false positives. Again using our current case as an example, we find that one single rule is reponsible for 83% of all violations, unlikely to only point out true problems. As a result, manual inspection of violating locations adds a significant overhead without clear benefit. But there is an even more ironic aspect to enforcing such noisy, ineffective, rules. Any modification of the 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 [1].

Clearly, it is of great interest to investigate if there is empirical evidence for the relation between rule violations and actual faults. Knowledge of this relation will aid developers in selecting the most ‘promising’ violations from the myriad of possibilities. In previous work, we presented a pilot study on a small industrial case and the MISRA C 2004 standard, where we found that this relation changes with different parts of the history and different rules [3, 4]. The MISRA standard is a widely adopted industrial standard for C, making it an interesting candidate for study. In this paper, we chose to repeat and expand the previous study for a large and mature product line from industry. We aim to answer more questions and use the same standard, to assess consistency of results across the two cases.

* 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

We discuss research questions plus 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 answer the research questions in Section 5. Finally, we compare with related work in Section 6 and summarize our findings in Section 7.

2. Study Design

This paper aims to answer three research questions with regard to the relation between violations and observed faults. All three are meant to test the idea that violations can be considered *potential* faults, and that they will truthfully point out problem areas at least some of the time. We discuss the questions and outline our approach, after which we explain some important measures and the data collection process.

2.1. Research Questions

A first intuition as to the relation between violations and faults might deal with absolute numbers: more violations in a certain file should indicate more faults, and more violations in a new release means more faults than in the previous one. However, larger codebases probably contain more violations as well as more faults than smaller ones. Since we are not interested in the relation between size and violations or faults, we will use fault and violations *densities*, i.e., the number of faults or violations per line of code. Concretely, consider two files, one of size n with v violations and f faults, and another of size $2n$. If the second file has $2v$ violations and $2f$ faults, we do not want to attribute the increase in faults to the increase in violations, but rather the increase in size. On the other hand, if there are $3v$ violations and $3f$ faults, the extra addition may be related to the increase in violations, aptly expressed using densities.

> **RQ1** *Are releases with a higher violation density more fault-prone?*

To answer this question, we record the densities of violations and faults over time (sequence of releases) for the complete project. As part of the changes between two releases are related to fixing faults, this should also cause a change in the observed violations. If consistent, this shows by a correlation between both measures. Although this investigation treats the project as a whole, it is also important to take the location of faults into account, which we will do next.

> **RQ2** *Are files or modules with a higher violation density more fault-prone?*

Here we measure the fault and violation densities for every file or module within a single release. These measures are subsequently used to create two rankings (based on either fault or violation density) and compute a correlation. We test two different levels of granularity to find out whether there are any differences in the prediction accuracy; file and module, the latter one in our case being a small number of functionally-related files. These levels are rather coarse-grained, whereas violations are usually concerned with a single line of code. This motivates our third question:

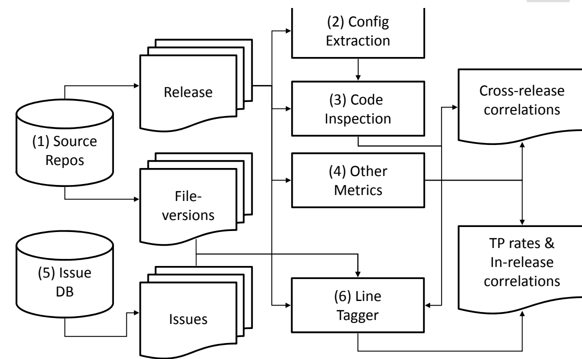


Figure 1: Measurement process overview

> **RQ3** *Are lines with violations more likely to point to faults than lines without?*

Answering this in a similar fashion as the previous question is infeasible. After all, it will almost be a binary classification: some lines will have a violation, some others will not, and only very few may have more than one. Instead, we compute the ratio with which violations correctly predict faulty lines. This is accomplished by tracking the violations over the history of the project, finding out whether the line was involved in a bugfix at some point. By doing this for every violation we can obtain a *true positive rate*, an indication of the predictive strength of violations on a line level.

These questions are stated using ‘violations’ in general, but individual rules may behave very differently indeed. We therefore differentiate these questions per rule. In particular, we investigate *which rules are most effective in fault-prediction* as per the three questions. In the following, we refer to the approaches used to answer these questions by the number of the corresponding question, for instance, A1 is the approach used to solve RQ1.

2.2. Data collection

We analyze a Software Configuraton Management (SCM) system and its accompanying issue database to link violations to faults. This impacts the definition of our measures: we define a *violation* to be a signal of non-conformance of a source code location to any of the rules in a coding standard; and a *fault* to be an issue in the database for which corrective modifications have been made to the source code. The corresponding densities are the number of violations or faults divided by the number of lines of code of the corresponding unit (be it project, module or file). We opt for physical lines of code rather than the number of statements, as not

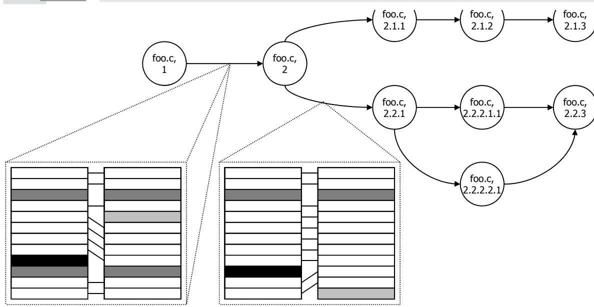


Figure 2: File-version and annotation graphs

all rules in a standard need be concerned with statements, but can also involve comments or preprocessor directives. To compute correlations we chose a non-parametric statistic, Spearman's ρ, instead of Pearson's r, to minimize issues with the distribution of the input populations.

2.2.1. Measuring densities

For RQ1, we need the number of violations, the number of faults, and the size (LoC) for every release. Figure 1 illustrates the steps involved in the measurement process. First we select the range of releases relevant to the study, i.e., the ones part of the selected project. We iterate over all the releases, retrieved from the source repository (1), performing a number of operations for each. From a release, we extract the configuration information (2) necessary to run the automatic code inspection (3), which in turn measures the number of violations. This configuration information includes the set of files that are to be part of the measurements, determined by the project's build target. We record this set of files and take some additional measurements (4), including the number of lines of code in the release. Once we have these measures for every release, we can determine a correlation between violation density and fault density.

2.2.2. Matching faults and violations

Answering the two other research questions also requires location information. We start gathering this information by checking which file versions are associated with faults present in the issue database (5). We proceed to compute a difference with previous revisions, indicating which changes were made in order to solve the issue, and marking the modified lines. When all issues have been processed, and all lines are marked accordingly, the complete file history of the project is traversed to propagate violations and determine which ones were removed with a fix. All this takes place in the Line Tagger (6), described below.

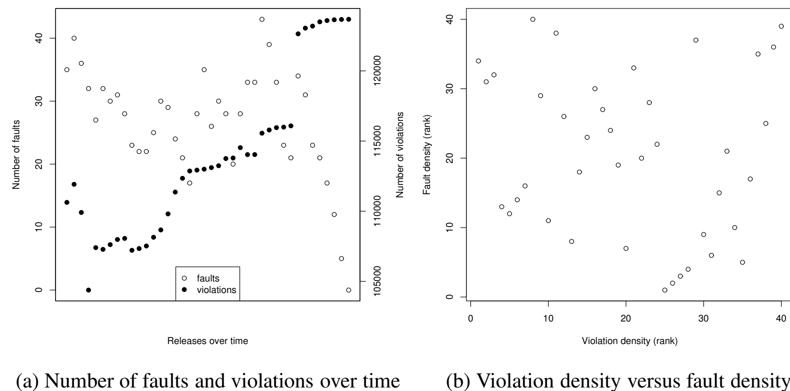
Using the set of files recorded in (2) the Line Tagger constructs a file version graph, retrieving missing intermediate file versions where necessary. For every file-version node in the graph, we record the difference with its neighbors as annotation graphs. An example is displayed in Figure 2. The round nodes denote file versions, each edge in that graph contains an annotation graph, representing the difference between the two adjoining nodes. Lines in the annotation graph can be either new (light grey), deleted (black) or modified (dark grey, pair of deleted and new line).

Consequently, we can trace the lines involved in a fault-fix, designated as faulty lines, to their origin. We use this to estimate the number of (latent) faults in a file or module in a particular release. Combined with the other measures, violations and LoC, we can determine a correlation between fault and violation density. Using the file-version graph, matching faulty and violating lines becomes straightforward. To compute the true positive ratio, we also need the total number of lines, defined as the number of unique lines over the whole project history. What we understand by unique lines is also illustrated in Figure 2: if a line is modified, it is considered a whole new line. This makes sense, as it can be considered a new candidate for the code inspection. In addition, it means that violations on a line present in multiple versions of a file are only counted once. Our line tagging is similar to the tracking of fault-fixing lines in [14], although we track violations instead.

2.2.3. Determining significance of matchings

Dividing the number of hits (violations on faulty lines) by the total number of violations results in the desired true positive rate. But it does not give us a means to assess its significance. After all, if the code inspection flags a violation on every line of the project, it would certainly result in a true positive rate greater than zero, but would not be very worthwhile. In fact, any random predictor, marking random lines as faulty, will, with a sufficient number of attempts, end up around the ratio of faulty lines in the project. Therefore, assessing significance of the true positive rate means determining whether this rate is significantly higher than the faulty line ratio. Modeling this proceeds as follows. The project is viewed as a large repository of lines, with a certain percentage p of those lines being fault-related. A rule analysis marks n lines as violating, or in other words, selects these lines from the repository. A certain number of these (r) is a succesful fault-prediction. This is compared with a random predictor, which selects n lines randomly from the repository. Since the number of lines in the history is sufficiently large and the number of violations comparably small, p remains constant after selection, and we can model the random predictor as a Bernoulli process (with p=p and n trials). The number of correctly predicted lines r has a binomial distribution; using the cumulative distribution function (CDF) we can compute the significance of a true positive rate (r/n).

Another way to assess true positive rates is by comparison with the *fault injection rate*, or the ratio of fault-related newly-written lines. Since every modification has a non-zero probability of introducing a fault, fixing violations with



(a) Number of faults and violations over time (b) Violation density versus fault density

Figure 3: Project fault and violation development

a low fault probability might have an adverse effect on the software. The fault injection rate can provide us with a safety threshold. Computing this number is fairly straightforward using the file-version and annotation graphs; we simply propagate the fault-related lines backwards through the history, marking each related line. After completion of the marking step, we extract the total number of inserted lines as well as the number of new lines marked faulty.

3. Case Description

Project For this study, we selected a software component of the NXP TV platform (referred to here as TVC, for TV component). The project was selected because of its size, maturity and high quality of the data in the repository. No coding standard or inspection tool was actively 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 been forced to remove all these violations right away.

The TVC project is part of a larger archive, structured as a product line, primarily containing embedded C code. This product line has been under development for a number of years and most of the code to be reused in new projects is quite mature. We selected from this archive the development for one particular TV platform, from the first complete release in June 2006, until the last one in April 2007. The sequence comprises approximately 40 releases. In addition, we selected issues from the issue database that fulfilled the following conditions: (1) classified as ‘problem’ (thus excluding feature implementation); (2) involved with C code; and (3) had status ‘concluded’ by the end date of the project.

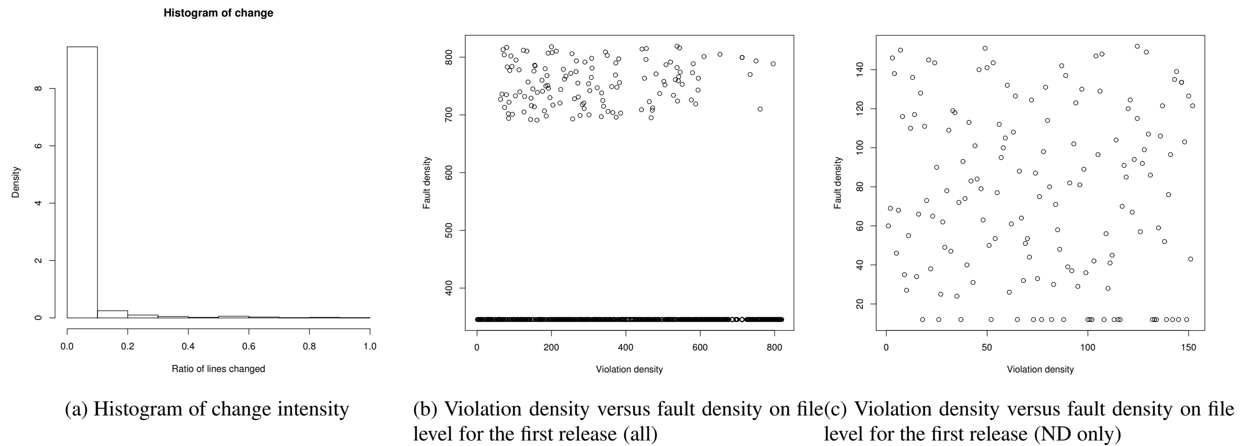
Coding standard The standard we chose for this study is the MISRA standard, first defined by a consortium of automotive companies (The Motor Industry Software Reliability Association) in 1998. Acknowledging the widespread use of C in safety-related systems, the intent was to promote the use of a safe subset of C, given the unsafe nature of some of its constructs [8]. 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.

Implementation The study was implemented using Qmore and CMSynergy. Qmore is NXPs own front-end to QA-C version 7, using the latter to detect MISRA rule violations. Configuration information required by Qmore (e.g., preprocessor directives, include directories) is extracted from the configuration files (Makefiles) driving the daily build that also reside in the source tree. For the measurements, all C and header files that were part of the daily build were taken into account. The SCM system in use at NXP is CMSynergy, featuring a built-in issue database. All modifications in the SCM are grouped using tasks, which in turn are linked to issues. This mechanism enables precise extraction of the source lines involved in a fix.

4. Results

In this section, we present the results of our study, as well as some interesting observations and how they impact the approaches we used. We first consider some characteristics of the software that may impact our analyses. Figure 3a illustrates the evolution of faults and violations over time. The number of faults fluctuates between 20 and 40 per release for most of the history, and later decreases rapidly. From discussions with the development team, we learned that this is typical behavior for this kind of project, where at first the focus is on feature implementation, fixing only the most severe faults, and near the end of the project all open issues are solved. Figure 3b plots the overall relation between fault and violation density, but no relation becomes apparent here, nor does it highlight different phases of the project.

However, one salient detail is that in TVC only 17% of the involved files is modified during the project. This is



(a) Histogram of change intensity

(b) Violation density versus fault density on file level for the first release (all)

(c) Violation density versus fault density on file level for the first release (ND only)

Figure 4: Illustration of impact of change distribution

clearly visible in the distribution of changes, depicted in Figure 4a. It can be explained by the fact that the TVC development is based on a product line, which includes a large amount of mature code, used in different settings over time. Since our approaches determine the relation between violations and faults based on changes, we need to determine how this impacts their results, for presentation in this paper as well as comparison with our earlier study. We discuss this point as we discuss results for each of the three approaches. In this discussion, we consider both the project as a whole (all files, marked 'all'), and the set of files part of the new development (the aforementioned 17%, marked ND for short).

4.1. A1: Cross-release analysis

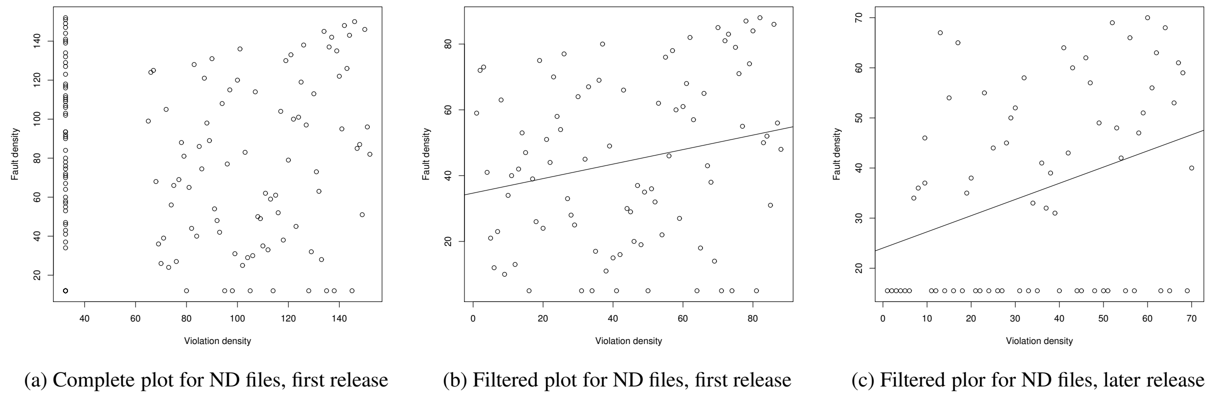
This approach is not influenced by the distribution of changes throughout the project, as it treats the project as a whole, looking at changes in violations and faults with no regard for location. As is shown in Figure 3b, there is no overall relation between fault and violation density across releases. However, a relation could be discerned for some individual rules. We have summarized the rules in three classes, having either a positive, negative or no correlation. An overview of the data for every rule, class and the complete body of violations is provided in Table 1. The first column for each aspect holds the data for the complete project, the second for ND only. The first four columns contain data relevant to this approach: the total number of violations found and the strength of the cross-release relation (R^2) with the classification obtained by manual inspection of scatter plots between parentheses.

4.2. A2: In-release analysis

Figures 4b and 4c illustrate the differences for A2 between the complete set of files (all) and the set of changed files (ND). Clearly visible is the large band of points on the bottom of Figure 4b, representing the set of non-faulty (and largely unchanged) files, which is absent in the other Figure. Again, in these cases both graphs show no relation at all. For a practical application of A2 (i.e., finding the most faulty locations) it makes little sense to include files that we know are probably not faulty. Therefore, in Figures 5a-5c we choose to illustrate ND files only. They show the relation for a selected set of rules, i.e., the ones performing consistently well in the related approach, A3. The figures show the rankings for each file, in violation density on the horizontal axis, and fault density in the vertical axis, where fault density is measured as number of faulty lines/loc. We see in the leftmost figure that there are some files for which no violations were found, resulting in the vertical 'bar' of points. If we zoom in on the set of files for which we have found violations (middle figure), we can observe a positive, significant rank correlation, albeit small ($R^2 = 0.05, p = 0.04$). The picture changes later on in the release history, as we find more files with no known faults present (the rightmost figure). The relation continues to exist ($R^2 = 0.11, p = 0.004$), although more flawed, as now we have a horizontal 'bar' of mispredictions on the bottom. Note that the discussion in these sections focuses on the file level, for the module-level results were similar.

4.3. A3: Line-based analysis

For A3, the skewed change distribution makes a comparison with a random predictor over the whole project to determine significance awkward, since we have more prior knowledge about the distribution of faults. To be fair, we would have to compare to a random predictor choosing only from the set of changed files. To understand the impact of the change distribution, we have computed two different precision rates, one for the complete project, and one for the set of ND files. Both have been displayed in columns 6-7 (absolute) and 8-9 (ratio) of Table 1, with an asterisk marking those rules that perform significantly better than random ($\alpha = 0.05$). Using the ND results, we can filter out 18 rules that would



(a) Complete plot for ND files, first release (b) Filtered plot for ND files, first release (c) Filtered plor for ND files, later release

Figure 5: Illustration of in-release relation for selected rules

otherwise have been considered as outperforming a random predictor; 10 remain that perform consistently well. Specifically, these are 3.4, 5.1, 8.1, 10.1, 10.2, 10.6, 11.3, 12.5, 12.13, and 19.5 (rows marked in bold in the table).

5. Discussion

In this section, we discuss the meaning of the results in light of our research questions, as well as some validity threats. For each of the questions, we include a comparison with our previous case study, TVoM [4].

5.1. RQ1: Cross-release analysis

A number of observations can be made as to the cross-release relation between violations and faults. First of all, only for a subset of the rules (10/88) we witness a positive correlation. This in itself is similar to our previous study, but the rules themselves are different this time. Second, looking at different phases of the project did not yield a consistent image; we tried investigating the correlation for the first and second part (dominated by either feature implementation or fault-fixing) but to no avail. This is different from our previous study, where those two phases were also clearly distinguishable in the scatter plot between fault and violation density. Finally, for 15 rules we observe a negative correlation, which is very counter-intuitive.

To understand this, recall that correlations are merely an indication of a causal relation. Specifically, there could be underlying factors influencing both measures, resulting in a correlation. For instance, suppose that, whenever a developer fixes a fault in the code, he adds a few lines of comments to describe what the fault was and how it is now fixed. Instead of using regular C comments he uses C++ style comments, which is forbidden by rule 2.1 of the MISRA standard. This would cause a consistent increase in violations with a decrease in faults, and could lead us to observe a negative correlation. Apart from this issue, there is the influence of other modifications than fault-fixes on the violation density, which may also impact the correlation. This issue is not present in the other two approaches used, and is likely to be responsible for the limited agreement between A1 and A2/A3 on the rules that are positively correlated.

5.2. RQ2: In-release analysis

In this analysis, we assess the relation between the location of faults and violations on a file or module level, and it raises some interesting issues. First, in TVC many files were never part of active development, thus unlikely to be involved in fault-fixing at any point during the project. This shows by the horizontal bar in Figure 4b; A2 can only be reliably used on the set of modified files as in Figure 4c. In other words, if we were to use A2 to predict fault-proneness, we would have to couple it with an approach to predict change intensity. Interestingly, this does not appear to be very hard in this case. For instance, the set of changed files between the first two releases that we analyzed comprises 85% of the total set of files changed at some point in the project history.

Second, since there is no overall relation between fault and violation density, we use a subset of rules to create a ranking. However, this results in a set of files for which there are no violations, as can be observed in Figure 5a. Instead of treating a lack of violations as a vote of confidence, this graph suggests it is better to see it as a lack of inspection. In other words, no predictive value can be attached to files without violations.

5.3. RQ3: Line-based analysis

As approaches A2 and A3 are based on the same technique of tracking faults and violations throughout the history, their results are similar. However, the results for A3 do not entirely correspond with A1. Although again, there is a subset of rules that indicate a relation between violations and faults, the two methods agree on only two rules (3.4 and 10.6). As a general principle, A3 is more strict than A1, as it has an

MISRA rule	Total violations	Total violations	Cross-release (R^2)	Cross-release (R^2)	True positives (abs)	True positives (abs)	True positives (ratio)	True positives (ratio)
1	Environment							
1.1	120	5	0.00 (o)	0.04 (o)	1	1	0.01	0.20*
1.2	5K	733	0.02 (o)	0.07 (o)	6	6	0.00	0.01
2	Language extensions							
2.2	183K	33K	0.01 (o)	0.02 (o)	755	755	0.00	0.02
3	Documentation							
3.1	145	97	0.10 (+)	0.00 (o)	0	0	0.00	0.00
3.4	**45**	**22**	**0.21 (+)**	**0.02 (o)**	**4**	**4**	**0.09***	**0.18***
5	Identifiers							
5.1	**477**	**65**	**0.17 (o)**	**0.09 (o)**	**25**	**25**	**0.05***	**0.38***
5.2	16	1	0.31 (-)	0.33 (+)	0	0	0.00	0.00
5.3	41	14	0.11 (+)	0.04 (o)	0	0	0.00	0.00
5.4	4		0.04 (o)		0		0.00*	
5.6	2K	852	0.13 (o)	0.02 (o)	23	23	0.01	0.03
6	Types							
6.1	7		0.18 (-)		0		0.00	
6.2	22		0.19 (o)		0		0.00	
6.3	5K	824	0.29 (-)	0.00 (o)	31	31	0.01	0.04
6.4	1		0.20 (+)		0		0.00*	
7	Constants							
7.1	3		0.20 (+)		0		0.00*	
8	Declarations and definitions							
8.1	**233**	**35**	**0.01 (o)**	**0.00 (o)**	**5**	**5**	**0.02***	**0.14***
8.4	4	1	0.05 (o)	0.33 (+)	0	0	0.00*	0.00
8.5	49	14	0.12 (o)	0.01 (o)	0	0	0.00	0.00
8.6	14	7	0.34 (o)	0.09 (o)	1	1	0.07*	0.14
8.7	547	71	0.08 (o)	0.06 (o)	2	2	0.00	0.03
8.8	820	114	0.10 (o)	0.02 (o)	4	4	0.00	0.04
8.10	383	69	0.02 (o)	0.02 (o)	1	1	0.00	0.01
8.11	123	54	0.33 (-)	0.09 (o)	0	0	0.00	0.00
8.12	35	5	0.20 (+)	0.06 (o)	0	0	0.00	0.00
9	Initialisation							
9.1	38	23	0.20 (-)	0.01 (o)	0	0	0.00	0.00
9.2	129	41	0.14 (o)	0.00 (o)	0	0	0.00	0.00
9.3	106	3	0.21 (o)	0.07 (o)	0	0	0.00	0.00
10	Arithmetic type conversions							
10.1	**3K**	**1K**	**0.00 (o)**	**0.01 (o)**	**157**	**157**	**0.04***	**0.15***
10.2	**27**	**15**	**0.00 (o)**	**0.01 (o)**	**6**	**6**	**0.22***	**0.40***
10.6	**226**	**64**	**0.12 (+)**	**0.01 (o)**	**7**	**7**	**0.03***	**0.11***
11	Pointer type conversions							
11.1	463	127	0.00 (o)	0.04 (o)	2	2	0.00	0.02
11.3	**1K**	**516**	**0.05 (o)**	**0.15 (+)**	**52**	**52**	**0.03***	**0.10***
11.4	7K	1K	0.02 (o)	0.02 (o)	21	21	0.00	0.02
11.5	5K	592	0.01 (o)	0.05 (o)	13	13	0.00	0.02
12	Expressions							
12.1	2K	941	0.11 (-)	0.02 (o)	64	64	0.02*	0.07
12.4	83	18	0.01 (o)	0.03 (o)	0	0	0.00	0.00
12.5	**790**	**143**	**0.00 (o)**	**0.21 (o)**	**17**	**17**	**0.02***	**0.12***
12.6	252	57	0.11 (-)	0.00 (o)	6	6	0.02*	0.11
12.7	5K	734	0.00 (o)	0.05 (o)	44	44	0.01	0.06
12.8	1		0.19 (o)		0		0.00*	
12.10	865	512	0.01 (o)	0.04 (o)	3	3	0.00	0.01
12.13	**51**	**15**	**0.00 (o)**	**0.02 (o)**	**4**	**4**	**0.08***	**0.27***
13	Control statement expressions							
13.1	10		0.01 (o)		0		0.00	
13.2	2K	1K	0.40 (-)	0.07 (o)	42	42	0.02*	0.04
13.3	29	11	0.01 (o)	0.08 (o)	0	0	0.00	0.00
13.5	31		0.20 (o)		0		0.00	
13.6	33	3	0.05 (o)	0.07 (o)	0	0	0.00	0.00
13.7	154	32	0.29 (-)	0.03 (o)	1	1	0.01	0.03
14	Control flow							
14.1	859	171	0.28 (-)	0.05 (o)	1	1	0.00	0.01
14.2	2K	770	0.02 (o)	0.07 (o)	18	18	0.01	0.02
14.3	4K	779	0.18 (+)	0.01 (o)	30	30	0.01	0.04
14.4	6		0.23 (o)		0		0.00	
14.5	15	3	0.06 (o)	0.00 (o)	0	0	0.00	0.00
14.6	20	10	0.20 (+)	0.04 (o)	0	0	0.00	0.00
14.7	591	50	0.09 (-)	0.00 (o)	0	0	0.00	0.00
14.8	665	88	0.19 (-)	0.09 (o)	10	10	0.02	0.11*
14.10	125	59	0.28 (+)	0.02 (o)	6	6	0.05*	0.10
15	Switch statements							
15.2	14	11	0.10 (o)	0.01 (o)	0	0	0.00	0.00
15.3	29	22	0.06 (o)	0.03 (o)	0	0	0.00	0.00
15.4	36	15	0.02 (o)	0.01 (o)	0	0	0.00	0.00
16	Functions							
16.1	11	3	0.08 (o)	0.19 (o)	0	0	0.00	0.00
16.2	18	5	0.00 (o)	0.04 (o)	0	0	0.00	0.00
16.4	863	180	0.02 (o)	0.29 (-)	4	4	0.00	0.02
16.5	75	9	0.08 (o)	0.00 (o)	0	0	0.00	0.00
16.7	2K	381	0.24 (-)	0.05 (o)	8	8	0.00	0.02
16.9	344	172	0.23 (-)	0.17 (+)	2	2	0.01	0.01
16.10	14K	2K	0.04 (o)	0.07 (o)	78	78	0.01	0.04
17	Pointers and arrays							
17.4	5K	609	0.01 (o)	0.04 (o)	16	16	0.00	0.03
17.6	1	1	0.11 (+)	0.01 (o)	0	0	0.00*	0.00
18	Structures and unions							
18.1	110	11	0.01 (o)	0.00 (o)	0	0	0.00	0.00
18.4	275	25	0.19 (+)	0.06 (o)	0	0	0.00	0.00
19	Preprocessing directives							
19.1	11	2	0.01 (o)	0.01 (o)	0	0	0.00	0.00
19.2	2	1	0.31 (o)	0.01 (o)	0	0	0.00*	0.00
19.4	382	28	0.02 (o)	0.27 (+)	1	1	0.00	0.04
19.5	**55**	**25**	**0.00 (o)**	**0.04 (o)**	**4**	**4**	**0.07***	**0.16***
19.6	84	40	0.13 (o)	0.00 (o)	0	0	0.00	0.00
19.7	1K	104	0.02 (o)	0.00 (o)	7	7	0.01	0.07
19.8	11	1	0.20 (+)	0.08 (o)	0	0	0.00	0.00
19.10	574	47	0.18 (+)	0.01 (o)	0	0	0.00	0.00
19.11	150	35	0.10 (o)	0.02 (o)	0	0	0.00	0.00
19.12	27	6	0.00 (o)	0.16 (+)	0	0	0.00	0.00
19.13	208	9	0.01 (o)	0.05 (o)	0	0	0.00	0.00
19.14	4		0.02 (o)		0		0.00*	
20	Standard libraries							
20.2	477	9	0.01 (o)	0.06 (o)	1	1	0.00	0.11
20.4	100	23	0.05 (o)	0.01 (o)	0	0	0.00	0.00
20.9	167	14	0.21 (-)	0.10 (-)	0	0	0.00	0.00
20.10	6	1	0.20 (+)	0.21 (o)	0	0	0.00	0.00
21	Runtime failures							
21.1	544	191	0.02 (o)	0.01 (o)	2	2	0.00	0.01
neg	16K	194	0.29 (o)	0.32 (o)	165	4	0.01	0.02
none	248K	49K	0.03 (o)	0.01 (o)	1K	1K	0.01	0.03
pos	5K	724	0.25 (o)	0.15 (o)	47	55	0.01	0.08*

Table 1: Relation between violations and faults per rule

extra requirement (location). Thus, the difference in results can be tracked back to this requirement, in two ways: (1) the otherwise positive relation is obscured by changes other than fault-fixes (relation in A1, no relation in A3); and (2) the relation is inadvertedly read in changes other than fault-fixes (relation in A1, no relation in A3).

In addition, the set of well-performing rules found here does not correspond with the set found in our previous study, they agree only on one rule (8.1). Apparently, the difference in domain (from SD-card driver to TV sofware component) is still large enough to cause such changes. The question remains why these rules in particular feature so prominently. Three of those rules involve arithmetic type conversions (chapter 10 in MISRA C). Also in an informal discussion with an architect of the TVC project, those were identified as potentially related to known faults. In fact, the development team was making an effort to study and correct these violations for the current release. However, the other rules selected by our analysis were less harmful in his experience. Summarizing, the set of rules found to be performing well can identify more fault-prone lines (RQ3) or files (RQ2, to a lesser extent). The significance tests rule out the possibility of a chance occurence, but it may still not be easy to find a direct causal relation between the violations and the faults they are matched with. Such a causal relation is difficult to determine, even in case of manual inspection of all fault-violation matchings by a domain expert.

The results suggest the importance of tailoring a coding standard to a specific domain, as the observed violation severity differs between projects. They also show that, within one project, it is possible to identify rules that are good fault predictors. As such, the approaches in our study can assist in tailoring a coding standard, by providing continuous monitoring of rule violation severities (i.e., true positive rates) over the course of a project. In other words, one can start a project with a minimal set of rules, deemed useful for that specific domain, then gradually add rules that have proven to be good predictors of faults. While less appropriate for small projects, it is useful for long-running projects or product lines. Also, when a coding standard is introduced on an existing codebase, the approaches can assist developers in prioritizing the multitude of violations.

5.4. Threats to validity

Internal validity With regard to the validity of the measurements themselves, we can make a few remarks. First, the correlations in A1 are sensitive to changes other than fault-fixes; they may obscure an otherwise positive relation or even inadvertedly signal one (as mentioned in the discussion before). Although we analyzed different phases of the project seperately, this did not result in a consistent picture.

Second, the number of faults per release in A1 is inaccurate, as only the number of open issues is considered. This excludes dormant faults, which is partly alleviated in the two other methods as they propagate faults through the history. Some faults may be present in the software at the end of development; these remain invisible to all approaches. However, given the heavy testing before shipping the TV software we expect this number to be minimal. The development only ends after the product has been integrated into the clients products, and therefore all major issues will have been removed. Also, it is possible that some violations, removed in non-fix modifications, pointed to latent faults. However, this category contains only 3% of the total number of violations (in the ND files), and is therefore unlikely to change the existing results significantly.

Finally, the matching between violations and faults may be an underestimation: some fault-fixes only introduce new code, such as the addition of a previously forgotten check on parameter values. Overestimation is less likely, although not all lines that are part of a fault-fix may be directly related to the fault (for instance, moving a piece of code). Even so, violations on such lines still point out the area in which the fault occurred. In addition, by computing significance rates we eliminated rules with coincidental matchings.

External validity Generalizing our results appears difficult if even a comparison with a previous case from the same company does not yield consistent results. They are consistent in the sense that there is a small subset of rules performing well, while no such relation exists for the other rules. However, the rules themselves differ. There are two important differences between both projects that could impact the way in which code is written: (1) TVC is a product line, with more mature code than TVoM; and (2) TVC contains a significant number of domain-specific algorithms and procedures, requiring specialized developers. An interesting question remains whether multiple projects based on the same product line will exhibit similar behavior. In addition, note that the set of rules analyzed in this study is a subset (88/141) of all the rules in the MISRA standard, as no violations were found for all rules. However, the analyzed rules cover almost all of the topics (i.e., chapters) of the standard. Only chapters 4 (two rules on character sets) and 7 (one rule on constants) are not present.

There are two requirements for the used approaches that should be considered when replicating this study. The first is that of the strict definition of which files are part of the analysis as well as what build parameters are used, as both may influence the lines included in the subsequent analysis, and thus the number of faults and violations measured. The second requirement is a linked software version repository and issue database, which may not always be defined as strictly as in our case, but which many studies have succesfully used before [18, 21, 16, 24, 15, 25, 20].

6. Related Work

In recent years, many approaches have been proposed that benefit from the combination of data present in SCM systems and issue databases. Applications range from an examination of bug characteristics [18], techniques for automatic identification of bug-introducing changes [21, 16], bug-solving effort estimation [24], prioritizing software inspection warnings [13, 14], prediction of fault-prone locations in the source [15], and identification of project-specific bug-patterns, to be used in static bug detection tools [25, 20].

Software inspection (or defect detection) tools have also been studied widely. Rutar et al. studied the correlation and overlap between warnings generated by the ESC/Java, FindBugs, JLint, and PMD static analysis tools [19]. They did not evaluate individual warnings nor did they try to relate them to actual faults. Zitser et al. evaluated several open source static analyzers with respect to their ability to find known exploitable buffer overflows in open source code [26]. Engler et al. evaluate the warnings of their defect detection technique in [6]. Heckman et al. proposed a benchmark and procedures for the evaluation of software inspection prioritization and classification techniques [11]. Unfortunately, the benchmark is focused at Java programs.

Wagner et al. compared results of defect detection tools with those of code reviews and software testing [23]. Their main finding was that bug detection tools mostly find different types of defects than testing, but find a subset of the types found by code reviews. Warning types detected by a tool are analyzed more thoroughly than in code reviews. Li et al. analyze and classify fault characteristics in two large, representative open-source projects based on the data in their SCM systems [18]. Rather than using software inspection results they interpret log messages in the SCM.

More similar to the work presented in this paper is the study of Basalaj [2]. While our study focuses on a sequence of releases from a single project, Basalaj takes an alternative viewpoint and studies single versions from 18 different projects. These are used to compute two rankings, one based on warnings generated by QA C++, and one based on known fault data. For 12 warning types, a positive rank correlation between the two can be observed (reportedly, nearly 900 warning types were involved in the study). Wagner et al. evaluated two Java defect detection tools on two different software projects [22]. Similar to our study, they investigated whether inspection tools were able to detect defects occurring in the field. Their study could not confirm this possibility for their two projects. Apart from these two studies, we are not aware of any other work that reports on measured relations between coding rules and actual faults. There is little work published that evaluates the validity of defects identified by automated software inspection tools, especially for commercial tools. One reason is that some license agreements explicitly forbid such evaluations, another may be the high costs associated with those tools.

The idea of a safer subset of a language, the precept on which the MISRA coding standard is based, was promoted by Hatton [8]. In [9] 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. He assessed MISRA C 2004 in [10], arguing 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”. This study complements these assessments by providing new empirical data and by investigating opportunities for selecting an effective non-noisy subset of the standard.

7. Conclusions

In this paper, we have discussed three analyses of the relation between coding standard violations and observed faults, and presented relevant empirical data for an industrial software project. Summarizing the discussion and the results, we arrive at the following conclusions for our case study:

> **RQ1** *Are releases with a higher violation density more fault-prone?*

Overall we did not find such a relation, only for some individual rules. Also, we did not find any phases in the project for a which a relation does exist, contrary to our previous experience [4]. The results from the approach used for this question did not agree with the other two, and must considered less reliable due to the less precise measuring of the number of faults. Future work is to employ the same fault-tracking approach to arrive at more consistent results.

> **RQ2** *Are files or modules with a higher violation density more fault-prone?*

This holds for 10 rules in the standard, with some reservations. There is no reliable prediction for files without active development (no changes) nor for files without violations. Also, the observed relation becomes less pronounced in time, as the number of registered open faults decreases.

> **RQ3** *Are lines with violations more likely to point to faults than lines without?*

As with RQ2, we cannot make this claim overall. For the same set of rules as in RQ2 this did appear to be the case (significant with $\alpha = 0.05$). Also this set differs from the one found in our previous study [4], indicating that a change of domain, even within the same organization with similar processes, can have a large impact on these measurements.

Apart from answering our three research questions, this study also indicates that making an effort to adhere to all

rules in a standard should not be considered erring on the safe side of caution. On the contrary: we reiterate the findings of our previous study, where we found that adherence to a complete coding standard without customization may increase, rather than decrease, the probability of faults [4]. Only 10 out of 88 rules significantly surpassed the measured fault injection rate. Even if fixing violations (be it immediately or later on an existing codebase) would be less fault-prone than an average modification, we are still left with a true positive rate of zero for half of the rules. In addition, we conclude that the particular rules that perform well for one project may not hold for another, even in a similar context. To test this aspect further, we intend to repeat this study for multiple projects within the same product line. In any event, both conclusions argue for careful selection of rules from a coding standard, taking into account the context at hand.

Acknowledgements The authors wish to thank the people of NXP for their support in this investigation.

References

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

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

[3] C. Boogerd and L. Moonen. Assessing the Value of Coding Standards: An Empirical Study. In *Proc. 24th IEEE Int. Conf. on Softw. Maintenance*, pages 277–286. IEEE, 2008.

[4] C. Boogerd and L. Moonen. Assessing the Value of Coding Standards: An Empirical Study. Technical Report TUD-SERG-2008-017, Delft University of Technology, 2008.

[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] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Symp. on Operating Systems Principles*, pages 57–72, 2001.

[7] 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.

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

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

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

[11] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM '08: Proc. 2nd ACM-IEEE Int Symp. on Empirical Softw. Eng. and Measurement*, pages 41–50. ACM, 2008.

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

[13] S. Kim and M. D. Ernst. Prioritizing Warning Categories by Analyzing Software History. In *Proc. 4th Int. Workshop on Mining Softw. Repositories (MSR)*, page 27. IEEE, 2007.

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

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

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

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

[18] 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 Softw. Dependability (ASID)*, pages 25–33. ACM, 2006.

[19] N. Rutar and C. B. Almazan. A comparison of bug finding tools for java. In *ISSRE'04: Proc. 15th Int. Symp. on Softw. Reliability Engineering*, pages 245–256. IEEE, 2004.

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

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

[22] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *1st Int. Conf. on Softw. Testing, Verification, and Validation*, pages 248–257. IEEE, 2008.

[23] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. 17th Int. Conf. on Testing of Communicating Systems (TestCom'05), volume 3502 of LNCS*, pages 40–55. Springer, 2005.

[24] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How Long Will It Take to Fix This Bug? In *Proc. 4th Int. Workshop on Mining Softw. Repositories (MSR)*, page 1. IEEE, 2007.

[25] 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.

[26] M. Zitser, R. Lippmann, and T. Leek. Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. In *Proc. 12th ACM SIGSOFT Int. Symp. on Foundations of Softw. Eng.*, pages 97–106. ACM, 2004.

TUD-SERG-2009-008
ISSN 1872-5392