Chapter 4

Using software history to guide deployment of coding standards

Authors: Cathal Boogerd, Leon Moonen

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. Therefore, we propose to use information from software and issue archives to link standard violations to known bugs. In this chapter we introduce such an approach and apply it to three industrial case studies. Furthermore, we discuss how to use the historical data to address two practical issues in using a coding standard: which rules to adhere to, and how to rank violations of those rules.

4.1 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 to 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., [Johnson, 1978] [Engler, 2000] [Flanagan, 2002]). Such tools generally come with their own sets of rules, but can often be adapted such that 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 [Li, 2006]. However, in spite of the availability of appropriate standards and tools, there are several issues hindering adoption.

[1] www.programmingresearch.com

[2] www.klocwork.com

[3] www.grammatech.com

Automated inspection tools are notorious for producing an overload of non-conformance warnings (referred to as violations in this chapter). For instance, 30% of the lines of one of the projects 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% [Kremenek, 2004]. Furthermore, rules may not always be appropriate for all contexts, and many of their violations can be considered false positives. For instance, we find that one single rule is responsible for 83% of all violations in one of the analyzed projects, which is unlikely to only point out true problems. As a result, manual inspection of all violating locations adds a significant overhead without clear benefit.

Although coding standards can be used for a variety of reasons, such as maintainability or portability, in this chapter we will focus on their use for fault prevention. From this viewpoint, there is an even more ironic aspect to enforcing 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 [Adams, 1984].

Therefore, in previous work [Boogerd, 2008-a], [Boogerd, 2009] we investigated the link between violations and known faults in two software archives using the MISRA standard [MISRA, 2004], a widely adopted industrial standard for C. We found that a small subset of the rules can be linked to known faults, but that this set differs between the two cases investigated. In this chapter, we expand the previous case studies and show how to use the same approach to address a number of practical challenges.

4.1.1 Challenges

Given the fact that interaction of code and coding standard can vary so dramatically, and that it has a great impact on the coding habits of developers, managing a coding standard for a software project is not a trivial task. Specifically, we identify two challenges:

1. Which rules to use from a standard? A standard may be widely adopted, but still contain rules that, at first sight, do not look appropriate for a certain project. In addition, when comparing several different coding standards an explicit evaluation of individual rules can help in selecting the right standard.
2. How to prioritize a list of violations? Although a standard may be carefully crafted and customized for a project, developers may still be faced with too many violations too fix given the limited time. To handle this problem most efficiently, we need to prioritize the list of violations, and define a threshold to determine which ones have to be addressed, and which ones may be skipped.

In other words, we define a rule selection criterion and a violation ranking strategy. We discuss our approach in Section 4.2, introduce our cases in Section 4.3 and discuss results of the approach in Section 4.4. We evaluate the results and describe how to meet the challenges in Section 4.5. Finally, we compare with related work in Section 4.6 and summarize our findings in Section 4.7.

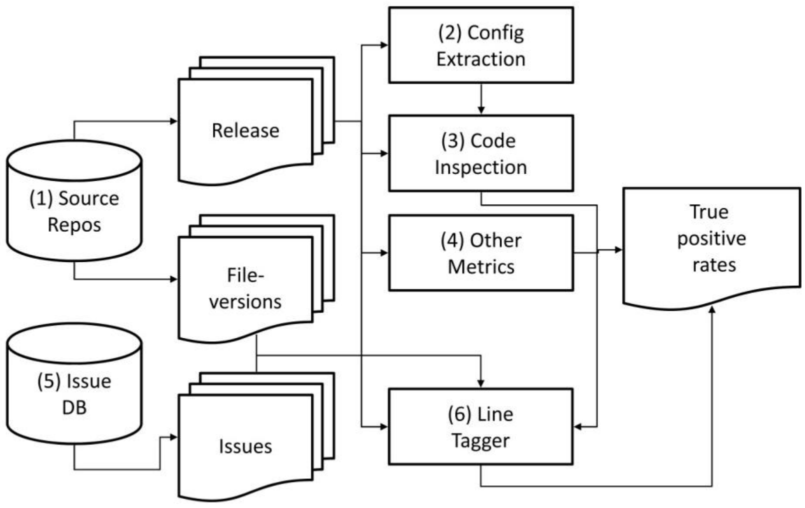

Figure 4.1 *Measurement process overview*

4.2 Approach

The approach uses a Software Configuration Management (SCM) system and its accompanying issue database to link violations to faults. Such an issue database contains entries describing observed problems or change requests, and these issues can be linked to the source code modifications made to solve it. Using this system 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; a *bug* to be an issue in the database for which corrective modifications have been made to the source code (the *fix*); those original faulty lines constitute the *fault.*

Measuring violations Figure 4.1 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.

4.2.1 Matching faults and violations

Matching requires information on the precise location(s) of a fault. We start gathering this information by checking which file versions are associated with bugs 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 involved in a fix. All this takes place in the Line Tagger (6), described below.

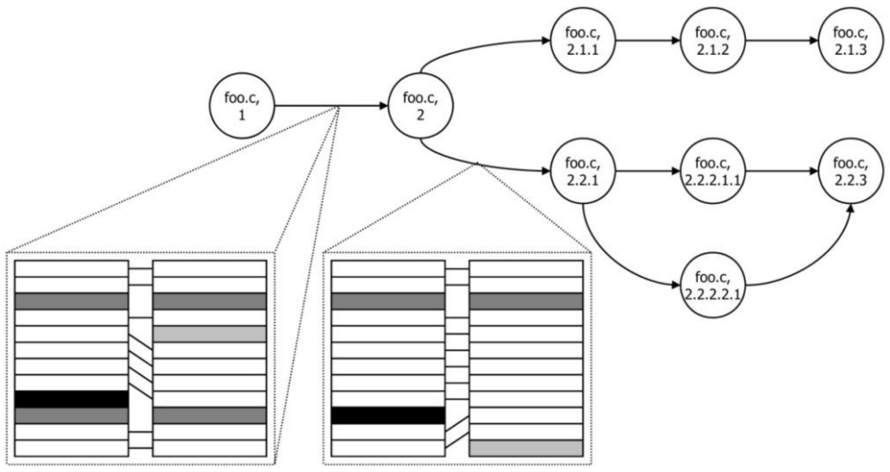

Figure 4.2 *File-version and annotation graphs*

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 4.2. 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).

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 4.2: if a line is modified, the modified version is considered a new unique 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 fix lines in [Kim, 2007-b], although we track violations instead.

4.2.2 Determining significance of matchings

Dividing the number of hits for a certain rule (violations on faulty lines) by the total number of its 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 precise or efficient. 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. This will give us an intuition as to whether the matchings are simply chance or not.

We model this by viewing the project 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 successful 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).

Because we know that only files that were actually changed contain faults, we only consider lines from these files for the analysis. This is a stricter requirement for significance than including also non-changed files.

4.2.3 Requirements

There are two important requirements for the approach 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. The link may be (partially) reconstructed by looking at the content of commit log messages, or be supported by the database systems themselves, as in our case. Many studies have successfully used such a link before [Li, 2006] [Sliwerski, 2005] [Kim, 2006-b] [Weiß, 2007] [Kim, 2007-c] [Williams, 2005] [Kim, 2006-a].

4.3 Three case studies

4.3.1 Projects

For this study we selected three projects from NXP [4], denoted TVoM, TVC1 and TVC2. The first project was a part of our pilot study [Boogerd, 2008-b], the second was previously studied in [Boogerd, 2009], the third one has been added in this chapter. We shortly describe each of them below.

TVoM is 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 standard. The complete source tree of the latest release 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.

TVC1 and TVC2 are software components of the NXP TV platform (referred to as TVC, for TV component). Both are part of a larger archive, structured as a product line, primarily containing embedded C code. This product line has been under development for five years and most of the code to be reused in new projects is quite mature. We selected from this archive the development for two different TV platforms: TVC1 comprises 40 releases from June 2006 until April 2007; TVC2 has 50 releases between August 2007 and November 2008.

In all these projects, 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 immediately remove all these violations. The issues we selected for these projects 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.

[4] www.nxp.com

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 [Hatton, 1995]. 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.

4.4 Results

Three tables list the results for each of the three cases: TVoM in Table 4.1, TVC1 in Table 4.2, and TVC2 in Table 4.3. In these tables we display detailed results for all the significant rules, that is, the ones that outperformed a random predictor with significance level $\alpha = 0.05$. Also, we include three categories of aggregated results: all rules, non-significant rules, and significant rules. For each of these, the second and third columns hold the total number of violations and the number of those that could be linked to fixes (i.e., true positives). The fourth and fifth hold the true positive ratio and its significance. The last column displays the number of issues for which at least one involved line contained a violation of the rule in question.

4.4.1 Case comparison

Noticeable are the differences in the set of significant rules between the three cases. All cases agree on only one rule, 8.1, which states:

"Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call."

A possible explanation of this is that some functionality may be implemented at first without too much attention for quality, so a function prototype is forgotten. Since it is new code, it will likely be changed a number of times before it is mature, and as a result is easily linked to a bug fix.

The agreement is (as expected) higher between the two cases from the TVC archive, which have eight rules in common: 1.1, 5.1, 8.1, 10.1, 10.2, 12.5, 14.10, and 19.5. Especially interesting is the concentration of rules in chapter 10 of the standard concerning arithmetic type conversions. For instance, rule 10.1 states that the value of an integer expression should not be implicitly converted to another underlying type (under some conditions). The reason for these rules to show up so prominently in our results is that the core code of TCV projects consists of data stream handling and manipulation, which is also the most complex (and fault-prone) part of the code. This leads to a concentration of fixes and these particular violations in that part of the code. Also in an informal discussion with an architect of the TVC project, the chapter 10 rules 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.

MISRA rule	Total violations	True positives (abs)	True positives (ratio)	Significance	Issues covered
5.3	2	1	0,50	0,97	1
8.1	56	15	0.27	0.99	2
8.5	2	1	0.50	0.97	1
9.2	2	1	0.50	0.97	1
11.1	38	10	0.26	0.97	4
12.7	91	22	0.24	0.98	5
12.13	36	10	0.28	0.98	1
14.2	258	68	0.26	1.00	9
16.9	2	1	0.50	0.97	1
All rules	9898	572	0.06	n/a	25
Non-significant rules	9411	443	0.05	n/a	25
Significant rules	487	129	0.26	n/a	16

Table 4.1 *Summary of rules for TVoM*

These results suggest the importance of tailoring a coding standard to a specific domain, as the observed violation severity differs between projects. These differences are clearly smaller within a single archive, showing that it is feasible to employ this approach in longer-running projects, with regular updates to assess whether rules should be included or excluded from the set of adhered rules.

4.4.2 Limitations

This section points out some limitations of the approach, how we deal with them, and to what extent they influence the results of the approach.

Measurement correctness

Some residual faults may be present in the software at the end of development that remain invisible to the approach. However, Fenton et al. found in their case study that the number of faults found in prerelease testing was an order of magnitude greater than during 12 months of operational use [Fenton, 2000]. Furthermore, 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 is typically small. For instance, in TVC1 this accounts for only 3% of the total number of violations, and is therefore unlikely to change the existing results significantly.

MISRA rule	Total violations	True positives (abs)	True positives (ratio)	Significance	Issues covered
1.1	7	2	0.29	0.99	2
3.4	22	4	0.18	0.98	1
5.1	65	25	0.38	1.00	1
8.1	35	5	0.14	0.97	3
10.1	1081	179	0.17	1.00	35
10.2	15	6	0.40	1.00	2
11.3	532	58	0.11	1.00	13
12.5	148	21	0.14	1.00	8
12.13	15	4	0.27	1.00	1
14.8	88	10	0.11	0.96	4
14.10	63	10	0.16	1.00	5
19.5	25	4	0.16	0.97	1
All rules	51529	1743	0.03	n/a	121
Non-significant rules	49433	1415	0.03	n/a	118
Significant rules	2096	328	0.16	n/a	52

Table 4.2 *Summary of rules for TVC1*

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 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.

Generalizing results

Since results differ significantly between projects, it is difficult to generalize them. They are consistent in the sense that there is a small subset of rules performing well, while no relation can be found for the other (non-significant) rules.

However, the rules themselves differ. There are a number of important factors that play a role in this difference. In discussing this, we must separate the cases by archive rather than project.

MISRA rule	Total violations	True positives (abs)	True positives (ratio)	Significance	Issues covered
1.1	116	8	0.07	0.96	8
1.2	73	6	0.08	0.97	7
5.1	169	11	0.07	0.96	4
5.2	27	3	0.11	0.98	2
6.1	5	1	0.20	0.99	1
6.2	8	3	0.38	1.00	2
8.1	160	15	0.09	1.00	8
9.2	15	8	0.53	1.00	1
10.1	4006	204	0.05	1.00	44
10.2	40	10	0.25	1.00	5
10.6	190	13	0.07	0.98	7
12.1	2698	210	0.08	1.00	45
12.4	91	9	0.10	1.00	8
12.5	584	116	0.20	1.00	28
12.6	302	27	0.09	1.00	9
12.7	1653	108	0.07	1.00	19
13.1	4	2	0.50	1.00	1
13.2	1256	119	0.09	1.00	34
13.3	265	43	0.16	1.00	11
14.6	4	1	0.25	0.99	2
14.10	135	22	0.16	1.00	14
15.2	4	2	0.50	1.00	1
16.10	1620	133	0.08	1.00	30
17.4	935	72	0.08	1.00	8
19.5	27	4	0.15	1.00	2
20.10	4	1	0.25	0.99	1
All rules	77158	3143	0.04	n/a	96
Non-significant rules	62767	1992	0.03	n/a	86
Significant rules	14391	1151	0.08	n/a	77

Table 4.3 *Summary of rules for TVC2*

Since results differ significantly between projects, it is difficult to generalize them. They are consistent in the sense that there is a small subset of rules performing well, while no relation can be found for the other (non-significant) rules. However, the rules themselves differ. There are a number of important factors that play a role in this difference. In discussing this, we must first separate the cases by archive rather than project, since these factors differ more between archives than within archives.

One difference between the TVoM archive and the TVC archive is that the former is a single, new project, whereas the latter is an archive containing five years of development. To counter influences of maturity of code, we only analyzed the new and edited code in TVC. There are two further major factors that were not under our control: (1) the application domain; and (2) the development team. TVC contains a significant number of domain-specific algorithms and procedures, affecting the type of code written and violations introduced, and requiring specialized developers.

When comparing projects within the TVC archive, these two factors also play a role. Although the team is mostly the same, developers join and leave over the course of the two projects, that together span almost 2.5 years. The type of TV component developed is the same in both projects, but the hardware platform for which the software is written is different, also impacting the type of violations introduced.

Finally, note that the set of rules analyzed for these cases is always a subset of all the rules in the MISRA standard (typically 50-60%), as in none of the cases violations were found for all rules. However, the analyzed rules cover almost all of the topics (i.e., chapters) of the standard. Only rules from chapters 4 (two rules on character sets) and 7 (one rule on constants) were not present.

4.5 Discussion

In this section, we will discuss how to use the results of our approach to meet the two challenges as identified in Section 4.1: rule selection and ranking.

4.5.1 Rule selection

There are three criteria involved in rule selection: the true positive rate, its significance, and the number of issues covered. The first tells us how likely a violation is to point to a faulty location, the second indicates whether this number is due to chance, and the third expresses the effectiveness of the rule. In this chapter, we have used the significance as a cutoff measure to eliminate rules that have a relatively high true positive rate simply because they occur so often. Using only the significant rules reduces the number of violations to inspect by 95% in TVoM, while still covering 64% of the issues covered by all the rules. For TVC1 and TVC2 reduction is 96% and 81%, with 43% and 80% of the total issues covered, respectively. The number of rules can be further reduced by setting a threshold for the true positive rate until the desired balance between number of violations and issues covered has been reached. For instance, setting a threshold of 0.10 for the true positive rate in TVC2 induces a reduction of 98% with a coverage of 41%. This explicit tradeoff between number of violations and number of covered issues makes the approach especially useful when introducing a coding standard in a legacy project with many violations.

4.5.2 Rule ranking

While rule selection is performed when choosing a coding standard, rule ranking is the problem of presenting the most relevant violations to a developer at compile time. From a fault prevention point of view, the most logical ranking criterion is the true positive rate. However, there is the added problem of the number of violations to inspect. Even with a customized standard, an inspection run may result in too many violations to inspect in a single sitting. In this case, one may define a maximum number of violations to present, but it is also possible to set a maximum on the number of *issues* to inspect. Using the true positive ratio attached to each violation we can compute the expected number of issues covered in a ranking. A contrived example would be finding two violations of rule 13.1 (TP = 0.50) and four of 20.10 (TP = 0.25) in TVC2: in this case we expect to find two issues (2 * 0.5 + 4 * 0.25). Since we may expect solving real issues to require significantly more time than inspecting violations, limiting inspection effort based on this number can be useful.

4.6 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 [Li, 2006], techniques for automatic identification of bug-introducing changes [Sliwerski, 2005] [Kim, 2006-b], bug-solving effort estimation [Weiß, 2007], prioritizing software inspection warnings [Kim, 2007-a] [Kim, 2007-b], prediction of fault-prone locations in the source [Kim, 2007-c], and identification of project-specific bug-patterns, to be used in static bug detection tools [Williams, 2005] [Kim, 2006-a].

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 [Rutar, 2004]. 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 [Zitser, 2004]. Engler et al. evaluate the warnings of their defect detection technique in [Engler, 2001]. Heckman et al. proposed a benchmark and procedures for the evaluation of software inspection prioritization and classification techniques [Heckman, 2008]. 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 [Wagner, 2005]. 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 [Li, 2006]. 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 [Basalaj, 2006]. 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 [Wagner, 2008]. 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 [Hatton, 1995]. In [Hatton, 2004] 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 [Hatton, 2007], 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.

4.7 Conclusions

In this chapter, we have discussed an approach that uses historical software data to customize a coding standard. The results from our three case studies indicate that rule performance in fault prevention differs significantly between projects, stressing that such customization is not a luxury but a must. After all, adhering to rules that are not related to faults may increase, rather than decrease, the probability of faults in the software.

We return to the challenges as stated in the introduction and summarize how our approach can assist in meeting them:

- *Which rules to use from a standard?* In a customization step, rules can be selected based on the measured true positive rate and the number of issues covered by each rule. The true positive rate expresses the likelihood of pointing to actual faults, and includes a mechanism to exclude accidental matches.
- *How to prioritize a list of violations?* Violations can be ranked using the measured true positive rate. In addition, this measure can be used to compute the expected number of issues covered, which is useful in defining a threshold in the number of violations that need to be inspected.

Of course, fault prevention is only one of the reasons for choosing a coding standard, so also other criteria (such as maintainability and portability) should be considered in rule selection and ranking. Nevertheless, the historical approach allows us to quantify the fault prevention performance of rules and makes the tradeoff of rule adherence with regard to this aspect explicit.

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

4.8 References

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

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

[Boogerd, 2008-a] C. Boogerd and L. Moonen. *Assessing the Value of Coding Standards: An Empirical Study*. Proc. 24th IEEE Int. Conf. on Software Maintenance, pp. 277-286. IEEE, 2008

[Boogerd, 2008-b] 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

[Boogerd, 2009] C. Boogerd and L. Moonen. *Evaluating the relation between coding standard violations and faults within and across versions*. Proceedings of the Sixth IEEE Working Conference on Mining Software Repositories (MSR). IEEE. To Appear

[Engler, 2000] D. Engler, B. Chelf, A. Chou, and S. Hallem. *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*. Proc. 4th Symp. on Operating Systems Design and Implementation, pp. 1-16, 2000

[Engler, 2001] 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*. Symp. on Operating Systems Principles, pp. 57-72, 2001

[Fenton, 2000] N.E. Fenton and N. Ohlsson. *Quantitative Analysis of Faults and Failures in a Complex Software System*. IEEE Trans. Softw. Eng., 26(8). (2000)

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

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

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

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

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

[Johnson, 1978] S.C. Johnson. *Lint, a C program checker*. Unix Programmer’s Manual, volume 2A, chapter 15, pp. 292-303. Bell Laboratories, 1978

[Kim, 2006-a] S. Kim, T. Zimmermann, K. Pan, and E.J. Whitehead Jr. *Automatic Identification of Bug- Introducing Changes*. Proc. 21st IEEE/ACM Int. Conf. on Automated Software Eng. (ASE), pp. 81-90. IEEE, 2006

[Kim, 2006-b] S. Kim, K. Pan, and E.J. Whitehead Jr. *Memories of bug fixes.* Proc. 14th ACM SIGSOFT Int. Symp. on Foundations of Software Eng. (FSE), pp. 35-45. ACM, 2006

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

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

[Kim, 2007-c] S. Kim, T. Zimmermann, E.J. Whitehead Jr., and A. Zeller. *Predicting Faults from Cached History*. Proc. 29th Int. Conf. on Software Eng. (ICSE), pp. 489-498. IEEE, 2007

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

[Li, 2006] 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.* Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID), pp. 25-33. ACM, 2006

[MISRA, 2004] MISRA (2004). *Guidelines for the Use of the C Language in Critical Systems*. MIRA Ltd., http://www.misra.org.uk/ , ISBN 0-9524156-2-3

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

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

[Wagner, 208] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. *An evaluation of two bug pattern tools for java*. 1st Int. Conf. on Software Testing, Verification, and Validation, pp. 248-257. IEEE, 2008

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

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

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

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