Towards a Single Software Quality Metric

The Static Confidence Factor

Paul Jansen (TIOBE Software, Paul.Jansen@tiobe.com)
René Krikhaar (Philips Medical Systems, Rene.Krikhaar@philips.com)
Fons Dijkstra (Océ Technologies, fdij@oce.nl)

[DRAFT]

Introduction

Static analysis is a way of analyzing software. It is done by parsing software source code and performing some analysis on the resulting abstract syntax tree. Since a static analysis can be executed without actually running the application, no application knowledge is needed to perform such an analysis. This makes static analysis a relatively inexpensive way of analyzing software.

One of the applications of static analysis is to measure deviations from industry specific or company specific quality standards, especially coding standards. Examples of well-known coding standards are [MISRA04], [Ambler00], [Sutter04], and [Hartog05]. Software applications that detect such deviations are called code checkers. Examples of code checkers are [PC-Lint], [C++Test], [QA-C/C++], [FindBugs], [PMD], [FxCop], and [ClockSharp].

Although many companies have gathered lots of such deviations from their coding standards with the aid of code checkers, hardly any management decision has been based on this kind of quality data so far. The major problem here is the gap between highly technical low level observations made by a code checker and the abstract way a manager thinks.

This article introduces an abstraction of coding standard deviations in such a way that it becomes a decision-enabling indicator for software management. The notion "static confidence factor" or, in short, the SC factor is defined which combines the most important results of static analysis into one single figure. The definition is an extension of similar notions used by Parasoft Company (see [Parasoft05] for more details). Its most important enhancements are that we introduce the notion of defect probability and that we explicitly claim that the confidence factor should not be relative to the size of a program.

The SC factor has been applied to real commercial software systems of various companies, most of them containing more than 1 million lines of source code. The management of these companies uses the daily updated SC factor as a way to monitor software quality now. The results of using the SC factor in practice are published in this article as well.

Problem Definition

Code checkers are capable of telling whether there are programming flaws in software. For example, it can check whether a comparison to a floating number occurs at a certain point in some source code. This is important information, because the result of such a comparison is machine dependent, thus making the software program less reliable. No wonder that the floating number

comparison rule is part of most coding standards.

Now suppose a code checker identifies 40,000 possible flaws (violations of the used coding standard) in a software program. What does this tell software management? Should they fire all software engineers now? Or can they ship the code now with much confidence? Or is this even an outstanding figure that can be used as a reliability showcase? Nobody knows.

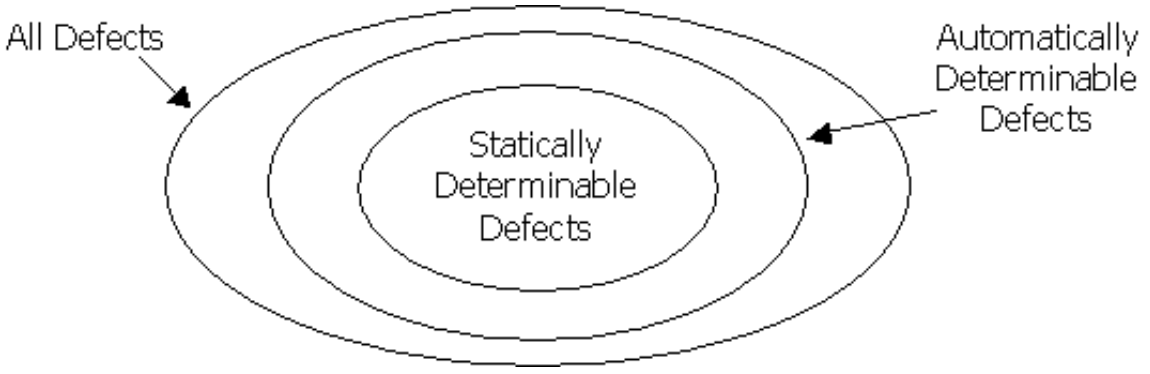
There are two problematic issues here:

1. What is the correspondence between statically detected flaws and software reliability?
2. How can software reliability based on coding standards violations presented in such a way that it can be handled by management?

In this article we will try to give an answer to both questions.

The second question is addressed by introducing the static confidence factor (in short SC factor). This is a single figure between 0 and 100 that summarizes all detected coding standard violations. The perfect score is 100. The way the SC factor is calculated indirectly answers the first question because it links coding standard violations to program reliability.

It is important to note that program reliability cannot be defined based on statically detected flaws alone.



This is because not all program defects can be detected by means of static analysis. Statically determinable defects are a subset of all automatically determinable defects. Other ways of determining defects are dynamic test results (to determine memory leaks), functional test results (to identify non-conformance to the specification), etc. But even the union of all these methods will only result in an approximation of the entire set of defects. Some defects cannot be detected automatically.

For this reason we will introduce the notion of static defect count. This is the estimated number of defects of a program based on the available static quality data. The adaptation of static defect count for daily usage results in the SC factor.

In order to define the SC factor we first have to define the terms coding standards, violations of coding standards, and static defect count more formally. This is the subject of the next two chapters. Don't be startled by the symbols, this article can also be read by skipping all the mathematical stuff.

Coding Standards

In this chapter some basic concepts concerning coding standards are defined. We start with the definition of a program.

Definition 1 (Program):

A program is defined as a set of modules, e.g. $\pi = \{ m_0, \ldots, m_n \}$. Observe that for programming languages with textual inclusion mechanisms such as C and C^{++} the notion of modules does not exists and thus will not suffice. For these languages one should use the notion of files instead throughout this article.

Definition 2 (Coding Standard):

A coding standard is defined as a set of rules, e.g. $C = \{ \Re_0, \ldots, \Re_n \}$. An example of a rule is "Avoid the use of multiple class inheritance" or "No attempt must be made to use the result of a function returning void".

Definition 3 (Defect Probability):

The defect probability of a rule denotes the severity of the rule with respect to program reliability and is denoted as $DP(\Re)$. The defect probability of a rule is a numerical value between 0 and 1. It indicates the chance that a violation of the rule is a real software defect. For instance, the rule "Use return values from functions" has a higher DP than "Use always braces for if-then-else statements".

The quality of a coding standard determines whether relative reliability approximates real reliability. In general one can say that the higher the sum of DPs of all rules of a coding standard, the better the coding standard.

Definition 4 (Violations):

A violation of a rule $\Re$ of a coding standard for a program π means that the rule does not hold for the program at hand. Violations are regarded as atomic entities in this article, generated by available code checkers. Hence, we will not dive into program semantics to prove that a certain rule has been violated.

The only important issue for this article is the number of violations for a certain rule or coding standard. This is denoted by $V_C(\pi)$ and indicates the number of violations of program π for coding standard C. This is a numerical value.

Static Defect Count

A program is more reliable than another program in case the program contains fewer defects than the other program. As stated earlier, we cannot strictly derive program reliability based on coding standard violations alone. That's why we introduce the notion of static defect count. The definition of this notion is rather complex, but this will be explained in detail.

Definition 5 (Static Defect Count):

The static defect count (in short SDC) of a program concerns program reliability as far as we can derive it only from static analysis data. Static defect count is indicated by $SDC_C(\pi)$ which denotes the static defect count for program π with respect to coding standard C. Its formal

definition in terms of coding standards is

$$SDC\{ \Re_0, \ldots, \Re_n \}(\pi) = \sum_{i=0 \text{ to } n} V\{ \Re_i \}(\pi) * DP(\Re_i) \quad (5)$$

Before explaining this definition in detail it is better to define the comparison operator on static defect count first.

Definition 6 (Relative Static Defect Count):

The relative static defect count states whether a program contains more, equal or less defects than another program. It is indicated with $\pi_0 \subset_C \pi_1$, meaning that program π_0 contains less defects than program π_1 with respect to coding standard C. Its formal definition is

$$\pi_0 \subset_C \pi_1 \Leftrightarrow SDC_C(\pi_0) < SDC_C(\pi_1) \quad (6)$$

This rather complex definition of static defect count will be explained step by step. Suppose there is a coding standard C with only one rule $\Re_0$ that is used for 2 programs π_0 and π_1. For this simple case, definition 6 of relative static defect count can be reduced to

$$\pi_0 \subset \{ \Re_0 \} \pi_1 \Leftrightarrow V\{ \Re_0 \}(\pi_0) < V\{ \Re_0 \}(\pi_1)$$

This reduction is possible because "n = 0" now and "$DP(\Re_0)$" occurs both at the left and right hand side of the "<" operator. The simple definition we obtained now states that program π_0 contains less defects than program π_1 in case there are fewer violations for rule $\Re_0$ in program π_0 than there are violations for $\Re_0$ in program π_1. This sounds sensible.

Now assume that there are 2 rules $\Re_0$ and $\Re_1$ in coding standard C instead of 1. Then it is tempting to extend the simple definition from above to

$$\pi_0 \subset \{ \Re_0, \Re_1 \} \pi_1 \Leftrightarrow (V\{ \Re_0 \}(\pi_0) < V\{ \Re_0 \}(\pi_1)) \wedge (V\{ \Re_1 \}(\pi_0) < V\{ \Re_1 \}(\pi_1))$$

But this appears to be far too weak. Usually coding standards contain a lot of rules and it rarely happens that for a program for each rule there are more violations in that program than in another program. As a consequence, most programs cannot be compared to each other with respect to static defect count if this definition would be used. We want the static defect count relation to be a total order, though.

Therefore the defect probability of rules (see definition 3) has been introduced. It is a natural way to relate coding standard rules. If a rule has a higher defect probability its violations should have more impact on the static defect count. If we reduce definition 6 for a coding standard containing 2 rules the result is

$$\pi_0 \subset \{ \Re_0, \Re_1 \} \pi_1 \Leftrightarrow$$
$$(V\{ \Re_0 \}(\pi_0) * DP(\Re_0) + V\{ \Re_1 \}(\pi_0) * DP(\Re_1)) <$$
$$(V\{ \Re_0 \}(\pi_1) * DP(\Re_0) + V\{ \Re_1 \}(\pi_1) * DP(\Re_1))$$

This definition takes the severity of rules into account by taking the product of the number of violations of a rule and its defect probability. From this it is easy to generalize to a coding standard containing an arbitrary number of rules, as is done in the eventual definitions 5 and 6.

There are a few interesting remarks to be made about the definition of relative SPR:

- The subset operator "$\subset$" of ordinary set theory has been used on purpose for the definition of static defect count. If programs are considered to be sets, then all mathematical properties of the inclusion operator also appear to hold for the static defect count operator. For example, the static defect count relation is transitive, non-reflexive, but it also adheres to set theory rules such as

  $$(\pi_0 \subset_C \pi_1 \wedge \pi_0 \subset_C \pi_2) \Rightarrow \pi_0 \subset_C \pi_1 \cup \pi_2$$

  This formula means that if a certain program contains more defects than another program and also more defects than yet another program, it contains also more defects than the union of these other two programs.

- So far the definition of SDC is coding standard dependent. If we want to compare programs against different coding standards and still want to be able to determine their mutual static defect count, definition 5 and 6 will not suffice.

  At first glance it might seem okay to just take the products of all violations and their defect probabilities for different coding standards. However, this can easily lead to contradictions, proving that a program is more reliable than itself. Suppose there are two coding standards C_0 and C_1. Coding standard C_0 only contains rule $\Re_0$ and coding standard C_1 contains rule $\Re_0$ and $\Re_1$. The definition of static defect count would look like

  $$\pi \subset_{SDC} \pi \;\Leftrightarrow$$
  $$V\{\Re_0\}(\pi) * DP(\Re_0) < (V\{\Re_0\}(\pi) * DP(\Re_0) + V\{\Re_1\}(\pi) * DP(\Re_1))$$

  which right hand side is true in case there is at least one violation of rule $\Re_1$ in program π. It would also mean that a program becomes less reliable in case more rules are added to a coding standard, which has certainly nothing to do with each other! We will call this observation the coding standard extension paradox.

  Observe by the way that we used the operator "$\subset_{SDC}$" (SDC stands for Static Defect Count) instead of "$\subset_C$". Ideally, we would like to have a definition that does not depend on a specific coding standard anymore. This is subject to future research.

Static Defect Count Example

Let's try the definition of static defect count in practice with an example.

Suppose we have a coding standard with three rules: $\Re_0$) "Do not use break statements" with DP = 0.005, $\Re_1$) "Do not use goto statements" with DP = 0.01 and $\Re_2$) "Do not change the loop variable in the body of a for loop" with DP = 0.05. There are two programs π_0 and π_1. Suppose these programs contain the following number of violations:

	$\Re_0$	$\Re_1$	$\Re_2$
π_0	7	3	12
π_1	5	9	4

Question is: What program is more reliable? We will prove that program π_1 is more reliable than program π_0 for the applied coding standard, i.e. $\pi_1 \subset \{ \Re_0, \Re_1, \Re_2 \} \pi_0$.

In terms of our definitions we have got the following assumptions.

1. $DP(\Re_0) = 0.005$
2. $DP(\Re_1) = 0.01$
3. $DP(\Re_2) = 0.05$
4. $V\{ \Re_0 \}(\pi_0) = 7$
5. $V\{ \Re_1 \}(\pi_0) = 3$
6. $V\{ \Re_2 \}(\pi_0) = 12$
7. $V\{ \Re_0 \}(\pi_1) = 5$
8. $V\{ \Re_1 \}(\pi_1) = 9$
9. $V\{ \Re_2 \}(\pi_1) = 4$

The proof is rather simple now by just applying the definition of static defect count.

$\pi_1 \subset \{ \Re_0, \Re_1, \Re_2 \} \pi_0 \Leftrightarrow$ [definition 6]
$SDC\{ \Re_0, \Re_1, \Re_2 \}(\pi_1) < SDC\{ \Re_0, \Re_1, \Re_2 \}(\pi_0) \Leftrightarrow$ [definition 5]
$(V\{ \Re_0 \}(\pi_1) * DP(\Re_0) + V\{ \Re_1 \}(\pi_1) * DP(\Re_1) + V\{ \Re_2 \}(\pi_1) * DP(\Re_2)) <$
$(V\{ \Re_0 \}(\pi_0) * DP(\Re_0) + V\{ \Re_1 \}(\pi_0) * DP(\Re_1) + V\{ \Re_2 \}(\pi_0) * DP(\Re_2)) \Leftrightarrow$
$(5 * 0.005 + 9 * 0.01 + 4 * 0.05) < (7 * 0.005 + 3 * 0.01 + 12 * 0.05) \Leftrightarrow$
$0.315 < 0.665$

Static Confidence Factor

The mathematical definition of static defect count of the previous chapter can be used to define the static confidence factor (in short SC factor). We have used two different terms "static defect count" and "confidence factor" here on purpose. The first one denotes a mathematical term in an ideal world whereas the second is about the way it is used in practice.

The SC factor maps the coding standard violations on a scale from 0 to 100. The mapping could have been defined as follows.

$$SC_C(\pi) = 100 * 1/(f(SDC_C(\pi)) + 1)$$

where "f" is some function to create a nice distribution between 0 and 100. It is not desirable to end up with SC factors that all lay between 95 and 100 or between 0 and 5.

Unfortunately, it is not that simple to just adopt the SDC definition and map it between 0 and 100. There are some practical complications with the definition of SDC that makes it hard to use in practice. As we will see in a moment, the SDC definition needs to make some shortcuts, even some mathematically irresponsible ones, to apply it to real software programs.

The complications of the SDC definition (definition 5) in practice are:

- The definition of static defect count makes use of the concept of "defect probability". Defect probability of rules is a theoretical concept until now. No empirical investigation has been conducted so far to determine the defect probability of coding standard rules. Hence, defect

probability as such cannot be used for the static confidence factor.

To simulate defect probability we group coding standard rules into severity levels. Experts in the field are supposed to assign the available coding standard rules to these severity levels. It appears to work best in practice in case the intervals between severity levels are logarithmic. Experience shows that a factor 4 works best. So, if level 1 denotes a defect probability of 1, the second level denotes a defect probability of 0.25, the third level 0.06, etc. In this way it is easier for experts to come to a unanimous choice about what rule belongs to what severity level without having empirical data available. It is for instance easy to distinguish a rule with defect probability 1 from a rule with defect probability 0.25.

To incorporate this insight in the aforementioned definitions, the definition of defect probability should be given in terms of severity levels:

$DP'(\Re_i) = 1 / power(4, (SL(\Re_i) - 1))$ (3')

where $SL(\Re_i)$ denotes the severity level of rule $\Re_i$. This is a natural number with 1 as lower bound.

- A way to get around the aforementioned "coding standard extension paradox" and thus become coding standard independent is to take the number of coding standard rules that have been applied explicitly into account. In case a coding standard contains 100 rules, the number of violations will be on average 5 times as high as the number of violations for a coding standard with only 20 rules. That's why part of the SC factor is to divide by the number of rules that have been applied. There is still a slight problem in case various programming languages are combined. In that case one should only divide by the number of rules for the programming languages at hand.

  The definition of defect probability needs to be changed once more to take this consideration into account:

  $DP''(\Re_i) = (1 / power(4, (SL(\Re_i) - 1)))/|\{ \Re_j \mid SL(\Re_i) = SL(\Re_j) \}|$ (3'')

  This definition is equal to the old definition of defect probability (definition 3'), but now it is also divided by the number of rules available at the same severity level. In this way adding a rule to the coding standard will lower the impact of other rules at the same level.

  The changes to the defect probability affects the definition of static defect count. The revised definition used in the context of the definition of the static confidence factor now looks as follows.

  $SDC'\{ \Re_0, \ldots, \Re_n \}(\pi) = \sum_{i=0 \text{ to } n} V\{ \Re_i \}(\pi) * DP''(\Re_i)$ (5')

- Although it is generally true that reliability decreases as software systems grow, there is a strong demand from the field to measure the static confidence factor independent of code size. To this end the result of applying the SDC definition to a program is divided by the number of kilo lines of code (denoted by $KLOC(\pi)$).

  Now we come close to our final definition of static confidence factor. The code size of a system is taken into account in the following way.

  $SC_C(\pi) = 100 * 1/((SDC'_C(\pi)) + 1)/ KLOC(\pi))$

- Finally, not all data is available to calculate the static defect count in practice. It could be the case that the code checking tooling crashes for various reasons. For instance, the input program was not compilable or there was an internal error in the code checker itself. Hence, $V\{ \mathfrak{R} \}(m)$ might be undefined for a certain rule $\mathfrak{R}$ and module m.

  Missing data due to failing code checkers is considered to have a serious impact on the SC factor. A module for which no violations can be calculated due to failing code checkers, i.e. $V\{ \mathfrak{R} \}(m)$ is undefined, is said to have a SC factor of 0.

  The percentage of code that can be checked in terms of modules is called the violation coverage, denoted by $VC_C(\pi)$. For instance, if only 89 of 100 modules can be checked, $VC_C(\pi)$ equals 0.89. Now we are capable to give the final definition of static confidence.

Definition 7 (Static Confidence):

The static confidence factor of a program for a coding standard C, denoted by $SC_C(\pi)$, is formally defined as follows.

$$SC_C(\pi) = 100 * VC_C(\pi) * 1/((SDC'_C(\pi)) + 1)/ KLOC(\pi)) \quad (7)$$

Experiences from Industry

The static confidence definition as defined in the previous section has been applied to various medium and large industrial projects to validate its meaning and see whether it works in practice. The table below shows the relevant data for these projects. The SC $Factor_0$ and SC $Factor_1$ denote 2 static confidence factors measured with an interval of 1 month. SC $Factor_1$ concerns the latest measurement.

Project	SC $Factor_0$	SC $Factor_1$	LOC
1	80.88	80.78	837,010
2	88.08	88.49	4,615,050
3	87.26	68.23	1,085,131
4	54.23	55.35	318,978
5	83.29	79.20	513,090
6	59.34	60.11	603,986
7	49.61	49.79	101,017
8	76.19	66.41	252,856
9	38.93	55.33	762,913
10	59.65	62.40	240,059
11	75.30	71.90	618,256
12	54.87	59.56	954,525
13	85.82	79.45	485,681
14	95.09	94.92	524,515
15	57.14	56.15	165,795
16	15.89	16.01	443,105
17	32.96	34.28	81,393
18	58.56	58.78	1,158,697

The static confidence factor appears to help managers and engineers to get a global impression of the quality of the software system they are responsible for. The side effect is that, now that this

quality indicator is available on a daily basis, it is used to steer the organization. It appears in monthly reports and is discussed during progress meetings.

Conclusions and Further Research

The static confidence factor, as defined in this article, appears to be a useful quality indicator to determine the static quality of a software system. Its basic principles are derived from a mathematically sound definition, but some enhancements had to be added to make it work in practice.

Future research should focus on the gap between the theoretical (static defect count) and practical (confidence factor) model. There are three main areas of further research:

1. The coding standard extension paradox states that coding standard extensions affect the reliability because it leads to more violations. Apparently, this is an anomaly in the current model. Statistical methods need to be applied to take the accuracy of the measurement into account. The more rules are checked the better the accuracy of the static defect count. The static defect count itself should not be influenced.
2. It is also interesting to extend research in the field of defect probability of coding standard rules. Besides coming to a real classification it will also help to compare coding standards on their quality and to come to more universally accepted coding standards for programming languages.
3. Finally, the undefinedness of some results, the violation coverage, should be addressed in more detail. The current solution of stating that if $V\{ \mathfrak{R} \}(m)$ is undefined, the static confidence should be 0, is a solution that is too simple. The undefinedness should be part of the theoretical model.

References

[Ambler00] Scott W. Ambler, "The Elements of Java Style", 2000, Cambridge University Press, ISBN 0521777682.

[C++Test] C++ Code Checker C++Test, Parasoft Company, URL "http://www.parasoft.com/jsp/products/home.jsp?product=CppTest".

[ClockSharp] C# Code Checker ClockSharp, TIOBE Software BV, URL "http://www.clocksharp.com".

[FindBugs] Java Code Checker FindBugs, SourceForge, URL "http://findbugs.sourceforge.net/".

[FxCop] C# Code Checker FxCop, Microsoft, URL "http://www.gotdotnet.com/team/fxcop/".

[Hartog05] Vic Hartog and Dennis Doomen, "Coding Standard: C#", revision 1.3, Philips Medical Systems, URL "http://www.tiobe.com/standards/gemrcsharpcs.pdf".

[MISRA04] Gavin McCall et al., "MISRA-C:2004 - Guidelines for the use of the C language in critical systems", 2004, MIRA Ltd., ISBN 0952415623.

[PC-Lint] C/C++ Code Checker PC Lint, Gimpel Software, URL "http://www.gimpel.com".

[Parasoft05] Parasoft Company, "Understanding the Workflow in a Coding Standards Implementation", URL "http://www.parasoft.com/jsp/products/article_reg.jsp?articleId=1158".

[PMD] Java Code Checker PMD, SourceForge, URL “http://pmd.sourceforge.net/”.

[QA-C/C++] C/C^{++} Code Checker QA-C/C++, Programming Research Ltd., URL “http://www.programmingresearch.com”.

[Sutter04] Herb Sutter and Andrei Alexandrescu, “C++ Coding Standards”, 2004, Addison-Wesley Professional, ISBN 0321113586.