Language subsetting in an industrial context: a comparison of MISRA C 1998 and MISRA C 2004

Les Hatton
CISM, University of Kingston[*]

November 20, 2005

Abstract

The MISRA C standard [7] first appeared in 1998 with the objective of providing a set of guidelines to restrict features in the ISO C language of known undefined or otherwise dangerous behaviour. The standard was assembled by representatives of a number of companies in the automobile sector in response to the rapidly growing use of C in electronic embedded systems in automobiles. The standard attempts to build on the earlier work of [6], [3] and others. Due to various perceived deficiencies, notably considerable ambiguity in the rule definitions, a revision was planned and eventually appeared in 2004. This paper measures how well the two standards compare on the same population of software and also determines how well the 2004 version achieved its stated goals. Given its increasing influence, the results raise important concerns.

Keywords: safer subsets, MISRA C, embedded control systems

1 Overview

Pragmatic safer subsetting of languages to remove dependence on poorly defined features is finally becoming a mainstream activity with the recent recommendation to form a high-integrity study group under the auspices of the ISO, [8] with the intention of producing sets of rules to restrict features with undefined or otherwise dangerous behaviour in programming languages in common use.

It frequently comes as a surprise to developers that significant parts of a programming language can fall into this category. In practice, all standardised programming languages contain problematic features for a variety of reasons which include the inability of the standardising committee to agree on the behaviour of a particular feature, the use of unintentionally ambiguous language in the standards document itself, omitting to say anything at all and so on. It must of course be remembered that many users of a programming language never see the underlying standard document, but these documents are usually

[*]L.Hatton@kingston.ac.uk, lesh@leshatton.org

between 300 and around 800 pages and contain complex and highly technical language. In other words, in spite of the great care language committees take, there will always be problems with their production, consistency and interpretation and sometimes surprisingly bad programs can legitimately survive compilation as a result. Regrettably, successful compilation is usually not a quality measure and in some languages, of which C and C++ are probably the most well-known, when a program compiles successfully for the first time, the programmer's problems are just beginning rather than ending.

In addition to the continuing work of a language committee in trying to tighten the definition as much as possible, over time, word gets round amongst users of a particular language that certain features are of suspect behaviour and best avoided. Often these are of a rather different nature and may be well-defined but simply surprising for one reason or another. As a result of this, codes of practice spring up to attempt to avoid problematic language fragments with greater or lesser success. C has always contained a rich set of such features, this richness resulting at least in part from the extraordinary success of the language over the years in an astonishingly wide range of applications, and it was arguably the first language where systematic avoidance of problematic constructs was enshrined in an excellent book, [6]. It should be noted in passing that the ISO standard document itself also goes to considerable pains to document poorly defined areas in appendix G of ISO C 9899 (1990) and appendix J of ISO C 9899 (1999). Other languages have also moved to address such issues, for example Ada95 which has a safety annex, (Annex H) and also carefully designed subsets such as SPARK Ada, [2].

In C, problematic features might be very simple such as the misuse of the comparison operator '==' to give expressions like:-

```
c == d;
```

which does nothing at all, (the programmer meant to say c = d which assigns the value in d to the object c), or rather more subtle involving the meaning of expressions such as:-

```
a[i] = ++i;
```

where i is used and modified in the same expression leading to a dependence on the order of evaluation of the operands, (which in most modern programming languages is left up to the compiler writer for optimisation reasons). In both C and C++, the above expression is entirely legal but leads to undefined behaviour. In languages such as Ada and Fortran, dependence on order of evaluation can also occur but there are considerably fewer opportunities for this to happen and it is consequently rather less frequently occurring, (a fragment such as the above occurs about every 7,000 lines in C, [3]). Suffice it to say that the practice of safer subsetting simply compiles lists of legal constructs such as these which have undefined or surprising behaviour, so that they can be avoided in future, beneficially affecting the reliability and perhaps also the safety, (the relationship between reliability and safety is non-trivial). An example of a compact measurement based subset for ISO C is given by [5]. Such rulesets are often accompanied by tools which automate the detection process to some extent. The greater the extent to which this can be done, the better as static detection takes place earlier in the development life-cycle than dynamic detection and is therefore much cheaper to do per fault found. This area has been particularly well studied in Ada.

2 Safer subsets and noise

The original MISRA C 1998 standard was discussed in detail by [4] from a number of points of view. The most pertinent aspect here is the issue of *static rule induced noise*. When a static analysis toolset detects dependence on an undefined feature, for example use of a variable which has not been initialised, it will issue a warning. Unfortunately, as pointed out by [4], such rules in practice fall into three broad categories, A, B.1 and B.2. Of these, category A rules are generally stylistic and have no known relationship with failure. An example of a category A rule is that all local variables should have a name beginning with 'L_'. Category B.1 rules could conceivably be associated with failure, (for example the frequently repeated rule that the 'goto' statement shall not be used), but for which there is still no measurement support as yet. In contrast, category B.2 rules are known by measurement to be associated directly with failure.

The ideal safer subset and the one most convincing to its users is one made up of category B.2 rules only, for example [5]. Professional engineers do not like their programs to fail and the detection of an obvious fault known to fail, (such as dependence on an uninitialised variable) is a palpable benefit. However, in static analysis, the run-time behaviour of even the most blatant faults is unknown and there is therefore a probability of failure invariably less than one and sometimes very much less than one. It can even be zero if an obvious fault is never executed, a not uncommon scenario in a complex program. Not withstanding these comments, there is evidence of causal quantifiable connections between such faults and actual failures, [10]. When typical commercial populations of C are subjected to a standard based entirely on category B.2 faults, a detection rate of around 8 per 1000 lines of source code is reported on average, [3], [4], a value which hasn't changed much in the last 10 years. In other words, a necessary condition for a safer subset related directly to failure is a detection rate of around this figure. It is not sufficient as it could be the wrong 8 of course.

The dangers of false positives In very simple terms, if a set of rules causes many more transgressions than around 8 genuine faults per 1000 lines of source code on average, there is an obvious danger that the programmer will not be able to see the wood for the trees. Detection of items belonging to category B.1 or even A are simply false positives. That such false positives may greatly outnumber real positives and simply obscure them is a well known problem in practical safer subsetting. There is however a greater danger first identified by [1]. The whole point of a subset is that transgressions from that subset should be corrected by the programmer before continuing. However, as showed by [1], there is a non-zero probability p (about 0.15 in Adam's work), that such a 'correction' will in fact inject a new defect of about the same severity because of unintended side effects.

To quantify this simply, suppose there are a total of f category B.2 faults correctly detected by a safer subset per 1000 lines of source code, where f is around 8 on average, [3]. If all these are corrected, the number of faults remaining after correction will be *fp*. Suppose now that a safer subset causes f genuine faults to be detected and g false positives to be detected, i.e. warnings unrelated to any known failure mode. Suppose further that all these faults, genuine or false positive have to be corrected for compliance reasons. The number of faults after correction will therefore be

$$fp + gp \tag{1}$$

However, there were f to start with so there is a chance that adhering to this noisy subset will actually increase the total number of faults and this will occur if:-

$$fp + gp > f \tag{2}$$

which can be simplified to the condition:-

$$\frac{f}{g} < \frac{p}{1-p} \tag{3}$$

As was pointed out in [4], even in the best case if it is assumed that the MISRA 1998 standard detected all detectable faults, and this is far from clear, the ratio (f/g) is around (1/50). In other words, if adherence to this standard were forced by some conformance requirement, then p, the fault injection rate reported by [1], would have to be less than around 0.02 in order to have the desired effect of reducing the total number of faults. Unfortunately, this appears significantly less than values quoted for this ratio including the value of 0.15 actually quoted by Adams. It follows that in an unadulterated form, there is a significant chance that strictly adhering to MISRA C 1998 might actually make things worse by increasing the number of faults.

The goals of MISRA C 2004 The essential goals of the MISRA C 2004 update were:-

1. To remove ambiguity in the wording of some of the rules, (c.f. [4] for examples).
2. To correct some of the rules which were actually wrong due to misunderstandings of the underlying ISO C standard, ISO 9899:1990.
3. To preserve the relatively simple nature of the wording for better accessibility to engineers
4. To reduce the noise so that the standard was much closer in transgression rates to the actual failure data.

The re-write of the standard took some 3 years after it was decided upon and in the end turned out to be a major re-write in which all the rules were renumbered, some rules removed and a number of rather complex rules related to arithmetic conversions added. Table 1 summarises:-

Version	Rules	Sections	Pages
MISRA C 1998	127	17	69
MISRA C 2004	141	21	111

Table 1: Size comparison between the two versions

15 rules in MISRA C 1998 were rescinded and 29 rules added leading to an overall increase of 14. Unfortunately, the update certainly fails on the first and third of its goals, although it partially addresses the second, (the fourth will be considered shortly). Some of the rules are still ambiguous and the standard suffers from creating its own words and concepts which were not used by the international committee which produced the underlying C standard, ISO C 9899, (1990). For example, an extraordinarily complex section in MISRA C 2004 more than 12 pages long introduces a number of concepts such as "underlying type" and "complex expression", the latter conflicting with the introduction of the complex type in ISO C 9899, (1999). The whole point of this section is to attempt to enforce the following long-established reliability principle in any programming language:-

> *Significant bits shall not be lost nor change their signedness (if relevant) by an implicit conversion.*

Unfortunately, apparently from over-complication, the MISRA C 2004 version also makes statements like:-

```
float32 = 1.5;
```

non-compliant, (because 1.5 is of type double in C leading to an implicit and entirely safe conversion). It is very difficult to see what possible contribution this distraction can make towards improving reliability. Distraction is an appropriate word here. When applying static rules, an obligation to comply is passed to the programmer which may be distracting from more important areas like algorithm correctness. Unless the static rule is directly relevant to a failure mode, the rule is simply a distraction and may therefore make things worse.

To assist users of the original standard in view of the renumbering and very significant rewording, an appendix in MISRA C 2004 conveniently cross-references the rule numbers where appropriate in the standard. This cross-reference table was used in the generation of the data which follows in the next section. It should also be noted that the original MISRA C standard was aimed specifically at *automobile* developments. MISRA C 2004 targets itself at the general purpose market-place making it all the more important that its intended goals should be met. The fourth and in the view of the author, the most important of these goals, will now be considered.

3 Signal to noise in MISRA C 1998 and 2004

In order to compare the two standards in terms of static rule noise, the same population of commercially released software was used. This consisted of seven packages totalling around 135,000 lines of source code. The packages were as follows:-

- Racing car control system
- The Javascript interpreter
- A digital television controller (set-top box)
- Car instrument package
- Engine management system
- Satellite communication system
- A government agency system

The biggest package used was around 61,000 lines and the smallest around 2,200 lines of source code. Note that this population is a little different to that studied by [4] for accessibility reasons.

In each case, the relevant MISRA C standard was set up by defining rules within a tool, the "Safer C toolset". For further confidence in the measurements, a second tool, "PC Lint" was used in parallel. Unfortunately, significant ambiguities remain within MISRA C 2004 so it was only possible to set up about 80% of the standard in each case but within this subset, no surprises resulted in using the two tools. It should be noted however that very significant differences were noted between different toolsets operating on MISRA C 1998 when this was not done, [9]. This also of course implies that the transgression rates reported here in the two standards are under-estimates.

The results are rather disturbing and the two comparisons on this population are shown in Figure 1 (MISRA C 1998) and 2 (MISRA C 2004). Note that in these figures, only rule 1 is common to both, being the requirement to comply with the underlying ISO C 9899 (1990) standard. Rule renumbering and in a number of cases complete rewording, affects all the other rules and there is no longer a one to one relationship between them. However, the overall transgression rates tell the story sufficiently well.

As can be clearly seen, the overall reduction in false positives is a disappointing 29%, meaning that the real to false positive ratio is still no higher than about (1/35). This leaves MISRA C 2004 still squarely in the zone where correcting all the transgressions is likely to *increase* the total number of faults rather than decrease them according to the argument above and the data of [1], and once again it must be concluded that the standard is unsuitable for use in an unadulterated form. Very careful and selective rule deviation will be necessary to recover any value.

A further disturbing feature of this analysis is that 5 out of the 7 packages contained features which unintentionally violated the underlying ISO C 9899 (1990) standard and which the corresponding compilers clearly did not flag. This does not include any deliberate extensions to the ISO standard associated with embedded control system concepts such as the handling of interrupts. The aggregated rate at which this occurred was around 4 per 1000 lines of source code with one package (the Satellite Communication system) being particularly bad contributing about 80% of these. This may be evidence of growing departures in compilers from compliance since the demise of compiler validation testing for any programming language by any of the national standards bodies in the year 2000. This does not bode well for the future.

3.1 Identifying noisy rules

It is immediately clear from Figures 1 and 2 that a relatively small number of rules contribute much of the noise and it is of interest to compare the offending rules in the two versions. However, simple identification of the most frequently occurring transgressions is not sufficient because this might be influenced by unusually high transgression rates in one or two packages and there was evidence of this in the 7 commercial packages used. It is important therefore to distinguish between noise in the standard, (i.e. noise which affects all packages more or less) and noise in the package, (where the programmers had a penchant for breaking some rules unusually often compared with the population at large). It is the former which is of the most concern here. In order to discriminate those rules which were transgressed frequently *and* which affect all packages more or less equally, the following statistic D based on the mean number of transgressions modulated by a statistic which is capable of discriminating data containing peaks will be used.

$$D = \{\frac{1}{N}\sum_{i=1}^{N} a_i\}.\{\frac{\{\sum_{i=1}^{N} {a_i}^2\}^2}{\sum_{i=1}^{N} {a_i}^4}\} \tag{4}$$

where N is the number of packages used and a_i is the number of transgressions of a particular rule in package i. Here the first term on the right hand side is the average and the second term is the inverse of the varimax statistic which discriminates against data with large peaks compared with more uniform data as can be seen by substituting a few sequences of numbers. The results of calculating this statistic for the two versions of MISRA C are shown in figures 3 and 4. As can be seen by comparing these with the corresponding figures 1 and 2, the distribution of noisiest rules changes somewhat.

Using the D statistic, the noisiest 8 rules for each of the two versions are shown in decreasing order of importance in Tables 2 and 3 for MISRA C 1998 and MISRA C 2004 respectively.

Rule	Rule type
18	Suffixing of numeric constants
49	Explicit tests of values against zero in predicates
59	if, while and for statements are brace-enclosed
45	No type casting between pointers
37	No bitwise operations on signed integer types
47	No dependence on operator precedence - must use parentheses
77	function arguments and defined parameters must be compatible
34	Operands of logical operators must be primary expressions

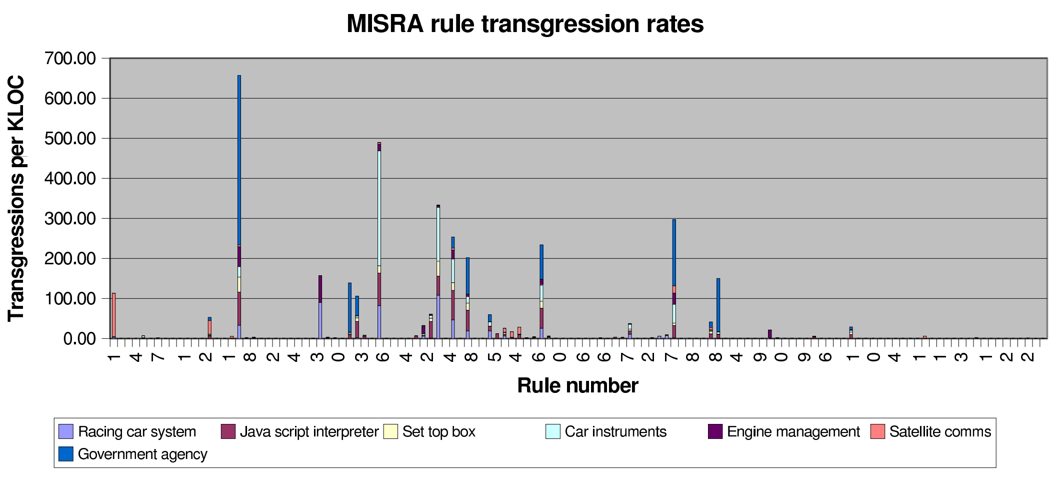

Figure 1: Transgression rates in warnings per 1000 lines of source code (KLOC) on 7 packages using MISRA C1 (1998).

Figure 1: Transgression rates in warnings per 1000 lines of source code on a test population of commercial C software using the MISRA 1998 standard

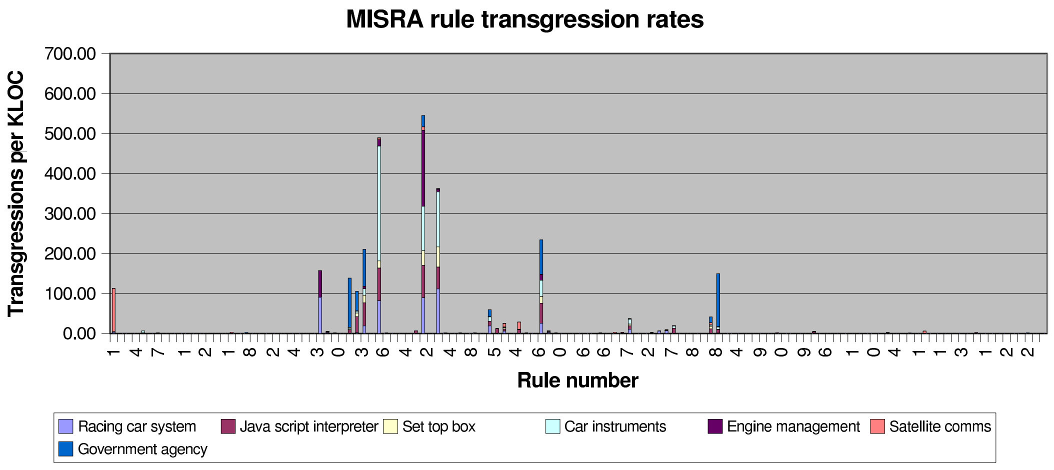

Figure 2: Transgression rates in warnings per 1000 lines of source code (KLOC) on 7 packages using MISRA C2 (2004).

Figure 2: Transgression rates in warnings per 1000 lines of source code on a test population of commercial C software using the MISRA 2004 standard

Table 2: The noisiest rules in decreasing value of D statistic in MISRA C 1998

Rule	Rule type
10.1	Implicit conversion of integer types
13.1	No assignment in Boolean valued expressions
14.8/14.9	if, while and for must be compound statements
11.1-4	Conversions between pointers
12.7	No bitwise operations on signed types
12.5	Operands of logical operators must be primary expressions
16.8	All exits in non-void function must have a return
14.7	A function shall have a single point of exit

Table 3: The noisiest rules in decreasing value of D statistic in MISRA C 2004

As can be seen, in spite of the complete re-structuring and non-simple relationship between the rules in the two versions, the noisiest rules essentially embody the same concepts. The conclusion is very simple. Unless these concepts are refined to be much closer to the underlying failure modes or excluded by a deviation policy, both versions of the MISRA C standard are far too noisy to be of any real use.

4 Conclusions

In view of the apparent widening influence of the MISRA C standard, this paper attempts to assess whether important deficiencies in the original standard have been addressed satisfactorily. Unfortunately, they have not and the important real to false positive ratio is not much better in MISRA C 2004 than it was in MISRA C 1998 and it is unacceptably low in both. A novel method of using a varimax-modulated statistic to demonstrate this was also presented.

Two of the other three goals (the first and third in the list quoted earlier) also do not appear to have been met satisfactorily and overall, little progress appears to have been made unfortunately as MISRA C 2004 is bigger, very much more complex certainly in some of its rules and has not solved the most fundamental problem of MISRA C 1998, viz. that its unadulterated use as a compliance document is likely to lead to *more* faults and not less because of the fault re-injection phenomenon first noted by [1]. A simple model of this behaviour was also introduced in this paper.

In its present form, the only people to benefit from the MISRA C 2004 update would appear to be tool vendors and it is to be hoped that steps will be taken both to simplify the wording and to reduce the false positive ratio in future revisions by taking a little more notice of published experimental data and being less tempted to invent rules on the basis that they seem a good idea. Otherwise it will go the way of many of its predecessors and fail to have any beneficial impact in this important area and may even make things worse.

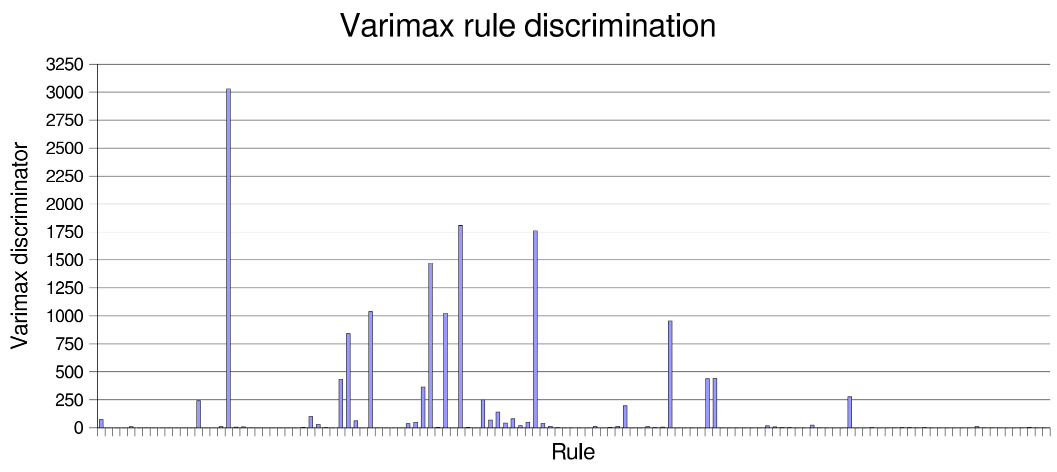

Figure 3: The D statistic of the text plotted against rule number for MISRA C 1998

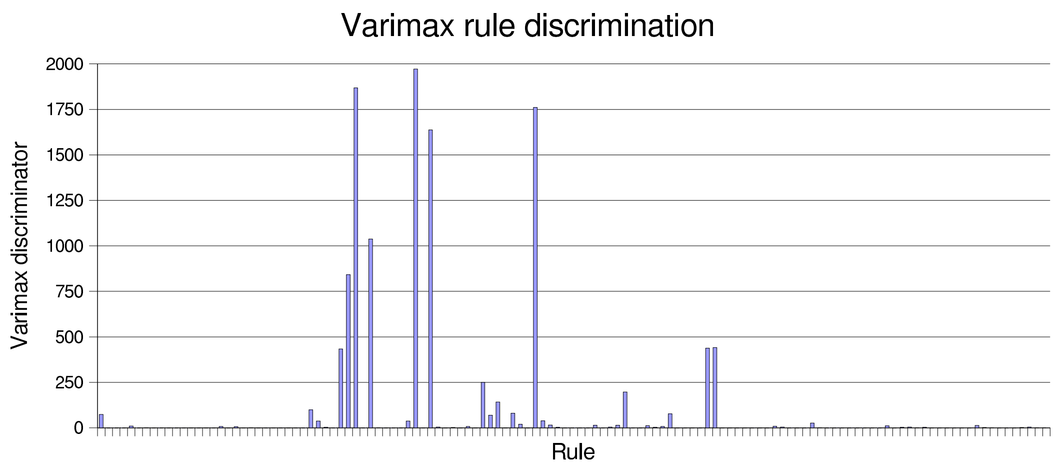

Figure 4: The D statistic of the text plotted against rule number for MISRA C 2004

References

[1] E. Adams. Optimising preventive service of software products. *IBM Journal of Research and Development*, 1(28):2–14, 1984.

[2] B. Carre and J. Garnsworthy. Spark - an annotated Ada subset for safety-critical programming. *Proceedings of conference on TRI-ADA '90*, pages p.392–402, 1990. ISBN 0-89791-409-0.

[3] L. Hatton. *Safer C: Developing software in high-integrity and safety-critical systems.* McGraw-Hill, 1995. ISBN 0-07-707640-0.

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

[5] L. Hatton. EC– a measurement based safer subset of ISO C suitable for embedded system development. *Information and Software Technology*, 47:181–187, 2005.

[6] A. Koenig. *C Traps and Pitfalls.* Addison-Wesley, 1989. ISBN 0-201-17928-8.

[7] MIRA Ltd. Guidelines for the use of the programming language C in vehicle based systems, 1998. http://www.misra.org.uk/.

[8] J. Moore. Proposal to SC 22 future directions study group regarding support of high-integrity applications. *Web*, 2005. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1109.pdf.

[9] S. Parker. Comparison of MISRA C testing tools. *Web*, 2001. http://www.pitechnology.com/downloads/files/MISRA_C_tools.pdf.

[10] S.L. Pfleeger and L. Hatton. Do formal methods really work ? *IEEE Computer*, 30(2):p.33–43, 1997.