Safer Language Subsets:
an overview and a case history, MISRA C.

Les Hatton

Computing Laboratory

University of Kent at Canterbury

L.Hatton@ukc.ac.uk

1.Abstract

This paper gives an overview of safer language subsets in general and considers one widely-used one, MISRA C, in particular.

The rationale, specification, implementation and enforcement of a safer language subset each introduce particular problems which has led to their inconsistent take-up over the years even in applications which may be safety-related and definitely need subset restrictions. Each of these areas will be discussed illustrating practical problems which may be encountered with standards in general before focussing on the widely used MISRA C standard, MISRA (1998). The approach taken is necessarily empirical and where it is able quotes measurements.

The real objective of this paper is to produce an empirically based taxonomy of programming language subset rules to bring all these issues together and promote the concept that a safer subset must be based on measurement principles however crudely they are practised currently in software development.

The concept of signal to noise ratio of a programming standard is also introduced.

© Copyright, Les Hatton, UKC, 2002-, version $Date: 2003/09/15 15:57:09 $

2.Overview

Rationale of a Safer Subset

The notion of subsetting a programming language for various reasons is firmly rooted in the history of programming. From the earliest days, *programming standards* governing the production of source code seemed to proliferate. As a result most organisations have often very complex documents which purport to restrict how a programmer may produce acceptable source code in a relevant programming language. Every organisation the author has been involved with since he started programming in 1968 seems to follow this pattern.

The question is why ? Why is that we feel a collective need to do this ? There seem to be two main strands of thought:

3.Promotion of a common style. The predominant idea here is that if only programmers wrote using the same style, they would find it much easier to read their colleague's code and that this would contribute to an improvement in response time for changes, either *corrective*, (fixing defects), *perfective*, (cleaning up without changing functionality), or *adaptive*, (adding new features). This is in general an entirely internal view in the sense that the end-user normally never sees source code and simply doesn't care what it looks like. The end-user is really only interested in the behaviour of the system. Rules whose rationale is based on the concept of style will be called ***category A*** rules here.

4.Avoidance of programming features which are in some sense suspect. In this case, the rationale is the recognition that particular features of a programming language should be avoided. This may either be because they attract general opprobrium such as the much-maligned *goto* statement or because they are known by a measurement process to fail. Rules whose rationale is based on suspicion of failure proven or otherwise, will collectively be known as ***category B*** rules and are further sub-divided below.

The essential difference between these two is that the former is inherently artistic in that it is very difficult to measure the effect on abstract concepts such as maintainability (the ease in which

© Copyright, Les Hatton, UKC, 2002-, version $Date: 2003/09/15 15:57:09 $

desired changes can be implemented for whatever reason). In short, they are subjectively or stylistically based. In contrast, the latter is objectively or empirically based, or is it ? Even here two issues must be distinguished by asking a key question. Is there any real evidence for failure or not ? Regrettably, lack of evidence is no barrier to ideas in computing science and as a result, mighty controversies can rage such as that which raged over the use of the *goto* statement without any objective evidence for or against failure being presented, Yourdon (1979). The many papers written do not contain a single experimental result to the author's knowledge. As a result of this controversy, most programming standards contain a rule even today to the effect that the *goto* statement shall not be used . Some languages have gone even further and have simply made it illegal; all this with no apparent failure evidence. Rules based on this concept will be called ***category B.1*** rules here - the 'folklore' rules, however well-intentioned.

In contrast, failure data for a language feature might actually exist. For example, using the value of an object when it has had no specific initialisation is known to have led directly to systems failure in many programming languages. In C, such dependence was reported as often as 1 occurrence per 846 source lines by Hatton (1995), although published failure rates are rather harder to quantify. As it happens, for this particular example, a value of 1.46% of all failures was quoted by Beizer in his extensive study, Beizer (1990) and the author has personally seen numerous failures directly due to this. Unfortunately, the relationship between the static occurrence rate and the dynamic failure rate is very complex, (one is a volumetric measure and the other a temporal measure), and not well understood although they are known to be strongly correlated, Pfleeger and Hatton (1997). Rules for which there exists known failure data however quantifiable will be known here as ***category B.2*** rules. For example, the existence of just one known failure involving such a language feature will be considered sufficient for it to be a member of this category on the grounds that if it has happened once, it can happen again.

Any categorisation should be accompanied by some comments on orthogonality. The above categories are based on observation and analysis of many programming standards and are clearly not orthogonal. Indeed, it is important that there be movement between them as a category B.1 item if supported by any quantitative evidence of a failure whatsoever would immediately become by definition, a category B.2 item..

Programming standards often seem to be an unstructured amalgam of category A, B.1 and B.2 rules with no particular organisation or emphasis, although category A tends to dominate. Some

standards will attempt to promote certain rules above others, for example, MISRA C, (MISRA (1998)) uses the words *required rule* for a rule which must be adhered to (a *mandatory* rule) unless a good reason can be given for not doing so and *advisory rule* for one which seeks to recommend only. Other standards use the words *rule* and *guideline* for these two concepts. Yet other standards make no attempt to distinguish at all.

Using these concepts, a safer language subset will be defined as follows:-

> *A **safer language subset** shall contain only category B.2 rules as mandatory and may contain category B.1 rules as recommended. It shall contain no category A rules.*

This definition allows the subset to grow in a natural way in the sense that has already been pointed out. If measurement data becomes available indicating that a category B.1 rule has caused failure, then it can automatically become a category B.2 rule.

There is an important reason for doing this. It is observed that one of the biggest barriers to acceptance of language subsetting by practising engineers is the absence of data. If an engineer knows that a feature has a measurable and quantified risk of failure, it is usually very simple to convince him or her not to use that feature. Professional engineers do not deliberately court failure. However, the absence of such data will usually be seen as an arbitrary attempt to interfere with the programmer's natural way of working which many programmers have strong feelings about. For better or worse, the author used to have little patience with programmer's naturally artistic leanings but programmers are under so much pressure to perform today that he has completely recanted and now believes that any such interference needs objective support which leaves the benefits in no doubt.

The principle reason then for documenting and promoting safer subsets of languages is to prevent the occurrence of common mode failures. Such failure modes can even be built into a language from the very earliest days of its use. C contains an excellent example of this principle at work. C has an unusually large set of operators for a programming language compared with languages like Fortran and Ada for example. Now mathematicians also have large numbers of operators but make no attempt to define natural precedence for them other than the very simplest distinction between multiply and divide on the one hand, and addition and subtraction on the other. In other words they use parentheses. In contrast, programming languages have lots of operators but prefer

to have the 'convenience' of a default precedence table so that users don't have to write all those tedious parentheses. Unfortunately, this has been a known failure mode in programming languages for years. Indeed in the case of C, a pioneering language which has 15 levels of precedence, one of its architects, Dennis Ritchie, published a posting on Usenet in 1982 explaining that there were problems with the default precedence ordering based on already observed failures but that it was too late to change it, van der Linden (1994). In the intervening years, C has been an extremely influential language spawning numerous other languages such as C++, Javascript, Java and so on *in each of which the original defects reported by Dennis Ritchie are faithfully reproduced.*

The above discussion leaves open the topic of what should be done with the category A items. Common style certainly has a place particularly as 'componentware' becomes more important, many examples of which rely on stylistic conventions, (COM, .NET and so on). Even so such stylistic conventions often prove difficult to enforce properly so *a separate document is best suited.* In this way, the inevitable controversy which seems to follow stylistic rules is kept separate from a safer subset document which by comparison, should invite very little controversy as by definition, there is objective evidence of failure for all of its component items.

Specification of a Safer Subset

The form of words in a safer subset is very important. In an ideal world, the rules would be specified in an unambiguous, complete and consistent way using one of the many forms of formal specification language. Unfortunately, the rule wording has to be accessible to a variety of users including but not limited to:-

a) The developer

b) The auditor or any person verifying adherence to the standard

c) The designer of any support tools used for verification.

The use of a formal specification language however restricts their general use to the relatively small number of people who understand such specifications. It could be argued that anybody working on a safety-related system for which adherence is important should be familiar with such specifications but again, market pressures mean there is a much bigger demand for such systems

than there are trained engineers to build them. At the present, society has chosen to accept this risk and so computer scientists must find a workable solution to bridge the gap in so far as it can be bridged.

The problems of specification go deeper than a safer subset. All safer subsets are derived from a *base language definition*, usually an international standard such as ISO C 9899: 1990 or 1999. Unfortunately the base documents themselves are not written in formal specification languages and are themselves contaminated with ambiguity, inconsistency and incompleteness, (Hatton (1995)). It could be argued that the absence of formality in the base definition should not preclude its use in a safer subset but it can be surprisingly difficult, particularly in the C-like languages where even basic concepts such as the character string have no formal definition.

The real, (and as yet generally unsolved) problem of using languages and their safer subsets remains how to specify such subsets in language which is both accessible and precise. It is not so much that we don't know how to do it as we simply don't do it. Specification problems will be highlighted in various forms in the examples that follow.

Implementation and Enforcement

The existence of a safer subset automatically begs the question as to how it should be enforced. Given a typical safer subset of say 100 rules and a typical piece of consumer code of say 100,000 lines, manual enforcement checks are simply untenable for most so some form of tool support is essential. This in turn raises the ultimate question of *Quid custodiae custodies* - who checks the checkers ? This leads to additional problems.

Again in an ideal world, there would be a simple yes/no test for every rule; an Oracle. In practice, this is extremely difficult to achieve and depends crucially on the quality of the wording not only of the safer subset but also its base language document. Ideally, there should exist both positive tests (i.e. a conforming tool must flag a feature) and negative tests (i.e. a conforming tool must NOT flag a feature). Inevitably there will also be tests for features whose conformance is simply uncertain given the existing wording of a rule. Consider the following definitions which will be useful in discussing these problems:-

1. Rule orthogonality or cross-talk. This measures the degree to which rules are independent of each other. Often in practice when writing tests, it is necessary to break one rule in order to test another thoroughly. Such rules will be defined here to be non-orthogonal. Such cross-talk can be displayed in a simple form as shown in Figure 1, a cross-talk analysis on the MISRA C standard discussed shortly.
2. Rule decidability. This measures the degree of agreement on whether a test actually does what it is supposed to. Examples of this will be shown for MISRA C.
3. Rule atomicity. Some safer subsets use short, succinct rules and others tend to use prose forms. A prose form frequently says too much and the rule is essentially non-atomic in that some parts of it can be adhered to successfully and others not. Alternatively, a rule may contain exceptions to itself embedded within the prose. Ideally rules should be atomic and not contain exceptions.
4. Rule weighting. Rules vary greatly in their importance and enforceability. For example in MISRA C, Rule 1 requires ISO C 9899:1990 conformance whereas Rule 28 forbids the use of the *register* keyword in C. Rule 1 is in every way a BIG rule which is effectively impossible to decide given ambiguities in the base language document. Furthermore, it is strongly biased to category B.2 behaviour. In contrast Rule 28 is almost frivolous by comparison, trivially easy to decide and apparently a member of category B.1, (the author is not aware of any reliability implications of this rule, indeed most C compilers completely ignore it nowadays). It is clearly inappropriate to weight these two rules the same, but they usually are so weighted in the absence of any objective way of doing it.

Each of these concepts contributes to the enforceability of a rule and its value.

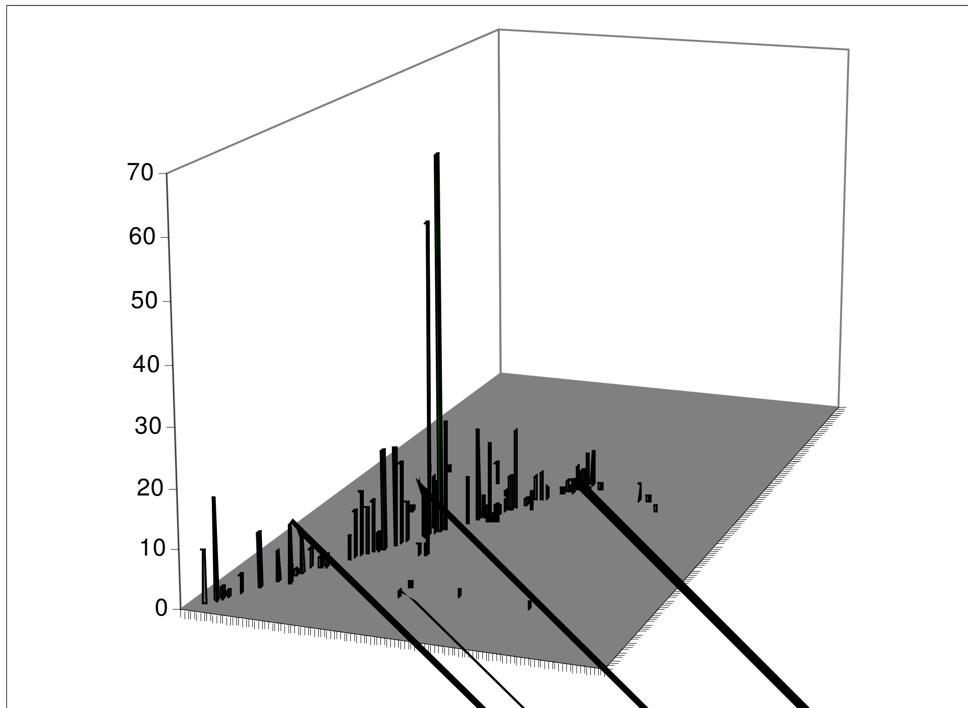

Figure 1: Cross-talk between rules in an incomplete analysis of the MISRA C standard, described later. The number of test cases is shown in the z (up) direction, the rules form the x axis and the files containing the test cases, (one file of test cases for each rule) form the y axis in a right-handed system. For complete freedom from cross-talk no test case for one rule would trigger another rule so the x-y plane would be diagonal. As can be seen, MISRA C achieves a gratifyingly low degree of cross-talk. The exemplary test suite used to produce this is given in Hatton (2002), which also contains a detailed rationale describing how this data is extracted..

5.General analysis of a population of standards

To illustrate some of the above issues and show that they are indeed present, the author analysed 10 commercial standards as being representative of about 50 which have been given to him for reviewing over the years mostly from the embedded systems community using the C language. It would have been nice to analyse all 50 but each one is time-consuming so these were considered to be representative after cursorily browsing through them all. The results are shown in table 1 below. Note that all standards were developed inhouse by their respective companies and all were in use as official documents of those companies.

Number	*Safety-related*	*Rules Numbered*	*# Rules*	*Category A/B*	*Pages*	*Presence of goto rule*
1	Yes	Yes	83	79/4	15	Yes
2	No	No	~50	50/0	75	Yes
3	No	No	~130 but impossible to analyse due to prose	-	90	Yes
4	No	Yes	48	44/4	66	Yes
5	Yes	No	Impossible to analyse	Nearly all category A	89	Yes
6	Yes	Yes	166	162/4	37	Yes
7	No	Part	Impossible to analyse due to prose	Nearly all category A	25	No
8	No	Separate checklist numbered	Standard is pure category A	Checklist 15/8	33	No
9	Yes	Yes	66	62/4	17	Yes
10	No	Yes	48	44/4	18	Yes

Table 1: An analysis of 10 typical standards for C programming. The documents cited as impossible to analyse were effectively in the form of essays rather than rules.

All these documents contained mistakes in their understanding of C and most of them consumed a considerable amount of resource to produce. For example, number 3 had no less than 6 reviewers, was obviously subjected to significant quality control and appears to have taken somewhere between 12 and 18 person months to produce from its quality control documentation. In other words it consumed something upwards of $ 100,000 in company cost and probably significantly more as such standards tend to be produced by experienced staff. Most of these standards will

have cost upwards of $ 50,000 to produce so it is obvious that the individual companies thought them worth doing, (or didn' t realise how much they cost).

Studying table 1 demonstrates that by far the biggest thrust of a standard is in stylistic issues, termed category A here. This is probably a natural development given the absence of guiding measurements. There is nothing special about C here. The author has in his possession Fortran, C++ and Ada standards which are comparable in length, content and emphasis. Furthermore, the standards which govern safety-related development were not noticeably biased toward category B which is both puzzling and disconcerting.

Given then that these companies deem them worthwhile, it is pertinent to ask the question how well are they adhered to in practice. In other words, do these companies get value from their efforts ? The answer is a resounding no as can be seen by inspecting Table 2 which contains the results of code audits based on the standard in half of these companies. In each case, a static analysis tool has been configured to flag code features which break the relevant standard. Substantial amounts of code produced by each company were then compared against their own standard and the number of code items which broke a rule in the standard are recorded in the third column as a frequency. Of course the same line of code could trigger multiple rules. Here the total number of transgressions was divided by the executable line count to provide the frequency so a frequency of 1 every 14 lines means that there were approximately 71 such occurrences in each 1000 lines of executable code.

Language	***Safety-Related***	***Transgression rate (transgressions per lines of non-comment source code)***
C	Yes	1 every 14 lines
C	No	1 every 12 lines
C	No	1 every 121 lines
Fortran	No	1 every 66 lines
Fortran	No	1 every 771 lines

Table 2: Some standards transgression rates measured by the author. Note that in each case, only a subset of the standard was used owing to the presence of

non automatically verifiable rules so real transgression rates are at least as bad as this.

Given that each of these standards was category A dominated, the clear message from this is that a category A dominated programming standard is of very questionable value. It certainly doesn' t protect against failure by definition and transgression rates are so high that they are also generally failing to achieve their primary goal of promoting a common look and feel.

Given that society is becomingly increasingly sensitive to the cost of software failure, there needs to be a change of emphasis to category B dominated standards and preferably to category B.2 domination. There follows an extensive discussion of a safer subset standard which attempts to move in this direction.

6.An example in detail: MISRA C

In the last few years, there has been a proliferation of consumer embedded control systems. Such systems are ubiquitous appearing in everything from a washing machine to a high-end medical scanner. In particular, they now appear in many critical applications where failure can lead to human or environmental damage. Prior to about 1993, most of the systems were developed in assembler. However, their rapid growth in lines of code began to preclude this option and by 1995 or so, most developers had switched, for better or worse, to C. The size of the systems produced today is quite simply staggering with a motor car containing around 3 million lines of C.

C is the perfect language for non-controversial safer subsetting as it is known to suffer from a number of potential fault modes and the fault modes are very well understood in general. They are certainly well-documented, (Koenig (1989), van der Linden (1994), Spuler (1995), Hatton (1995)), are certainly present in released systems, (c.f. Figure 2), and are highly correlated with failure, (Pfleeger and Hatton (1997)), but codes of practice have only evolved slowly. In 1998, building on this work, MIRA, the Motor Industry Research Association, an international consortium of major car and car component manufacturers based in the UK, produced an official set of guidelines for the use of C in automobile electronic systems, MISRA (1998), with the laudable goal of eliminating as many fault modes as possible. The guidelines describe a restricted subset of C defined by a number of rules backed by informal explanatory text and some code fragments.

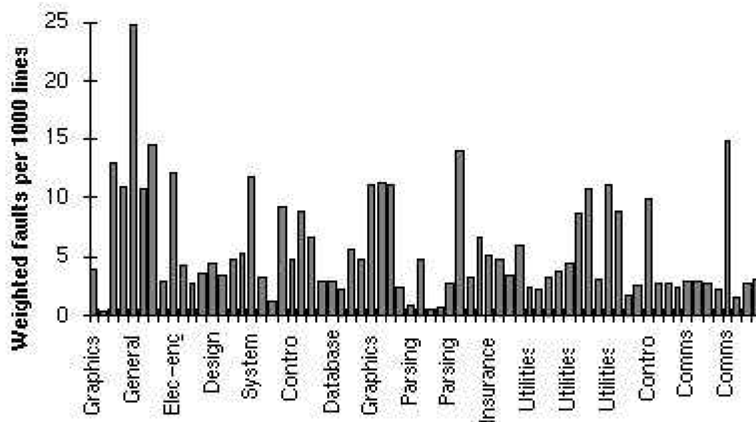

Figure 2: The occurrence rate in occurrences per 1000 lines of source code of a class of statically detectable defects known to cause problems in C as measured by the author and reported in Hatton (1995).

These guidelines have become very widely used around the world, not only in the automobile industry but also in other industries in which there is a substantial safety-critical component, such as aircraft and medical devices industries. They have also recently been translated into Japanese by a consortium of members of the Japanese automotive industry.

To see the significant change of emphasis embodied in this standard, Table 3 compares MISRA C with the standards analysed in the previous section using the same criteria and shown as Table 1.

Safety-related	***Rules Numbered***	***# Rules***	***Category A/B***	***Pages***	***Presence of goto rule***
Yes	Yes	127	0/127 (72/55) split on B.1 and B.2	69	Yes

Table 3: MISRA C compared with the standards in Table 1 using the same criteria.

It can immediately be seen that the thrust of this standard is towards category B with some reasonable attempts to bias it towards category B.2 This is clearly a step in the right direction.

There remain a number of fundamental problems however which illustrate the points made in the first part of this paper well.

1. The wording of informal standards such as MISRA is insufficiently precise, and problems have arisen because of the considerable leeway in its interpretation.
2. Conformance is a key issue in critical systems, yet there is currently no way of measuring conformance to this standard.
3. The underlying core standard for MISRA C is ISO C 9899:1990, however C has evolved since then without MISRA-C keeping track, and there appears to be no inclination or intention amongst the ISO C community to produce a safety and reliability annex for ISO C 9899:1999.
4. The lack of precision has led to rule cross-talk and rule inconsistency which forces users to deviate rules. (*Deviation* is a formal way of declaring that a rule can be broken on reasonable grounds which must be stated.)
5. Further measurement is necessary in order to provide a complete picture of real failure data.
6. MISRA-C does not address all known fault modes, and does not incorporate the full range of analysis checks that it might.
7. MISRA-C still has too low an emphasis on category B.2 rules.

In spite of these deficiencies, MISRA C is a huge step forward over the state of the art represented by the standards analysed in Table 1.

A quick browse through MISRA C

There are 93 required rules together with 34 advisory rules and as can be seen by Table 3 above, it is about average in size. The required rules are intended to be adhered to and the advisory rules are recommendations. The rules cover relatively simple rules such as Rule 46 - no dependency on evaluation order, a well-known problem in most programming languages and particularly in C and C++ - to all-encompassing rules such as Rule 1 which requires compliance with ISO C90 as

described above. Their success has come at the expense of some precision, since they are currently written in plain English.

The drawbacks described above will be exemplified under the following headings:-

Rule incorrectness

MISRA C is pretty good in general but there are some places which contain mistakes in interpretation of the base document ISO C90, for example, Rule 22 which talks about objects needing to be declared at function scope. Unfortunately, in ISO C90, only labels can have function scope. This miswording renders such a rule immediately questionable.

Rule redundancy

There are a few places where rules overlap, most notably with Rule 1, which requires compliance to the base document, ISO C90. An example is Rule 80 which prohibits passing void expressions as function parameters.

Rule cross-talk

Cross-talk in MISRA was illustrated in Figure 1, but to give specific examples, there is considerable cross-talk between Rules 1, 72, 76 and 78 which all relate to the security of function interfaces.

Rule decidability

An example of this can be found in Rule 37 which prohibits bitwise operations on signed integer types. Unfortunately the rule does not make it clear whether the integral promotions should be taken into consideration.

Rule atomicity

Rule 1 requiring compliance to the ISO C90 standard is the ultimate non-atomic rule but there are other simple examples in MISRA for example, rule 50 (don' ttest floating point variables for equality / non-equality) which has an exception in the body of rule 117 (values passed to library functions shall be checked for validity).

Adherence to MISRA C

The focus on category B rules in MISRA C means that it is much easier to determine transgression rates than in the standards analysed in Table 2. This is both good and bad. It is obviously good in that detected transgressions of category B.2 items is directly related to improved reliability. It is bad in the sense that the greater decidability of conformance means that transgressions rates are even worse than shown in Table 2. Table 4 shows some typical transgression rates against the MISRA C standard.

Language	***Safety-Related***	***Transgression rate (transgressions per lines of non-comment source code)***
C	Yes	1 every 7 lines
C	No	1 every 2 lines
C	No	1 every 14 lines

Table 4: Some standards transgression rates measured by the author against MISRA C. As can be seen, engineers currently find it very difficult to adhere to this standard even given the fact that perfect adherence is currently impossible owing to the presence of various forms of imprecision described in the text.

The high frequency of occurrence of these rule transgressions is cause for much concern and will be known here as *'static noise'*. This raises the possibility of trying to quantify this concept.

7.Signal to Noise ratio in safer subsets

The discrepancy between the data in Tables 2 and 4 and the frequencies of failure of unequivocal category B.2 items shown in Figure 2 is disturbingly large. Translated into frequency, the average rate of occurrence in Figure 2 corresponds to a frequency of around 1 per 120-140 lines of executable code. This is between one and two orders of magnitude less frequent than the data of Tables 2 and 4. To reduce the environmental differences between these two sources of measurement, Figure 3 is a repeat of the experiment in Figure 2 but on systems which appear in Table 4. These particular systems using the measurement scheme of Figure 2, (this is given in detail in Hatton (1995)) gave an average transgression rate of around 12 per 1000 lines of executable code. When measured against the MISRA C standard however, the transgression rates

recorded are shown in Figure 3. These are added together for each package so the total should be divided by 7. The result is an average transgression rate across all systems of around 600 per 1000 lines of executable code with 4 rules contributing around 75% of all these.

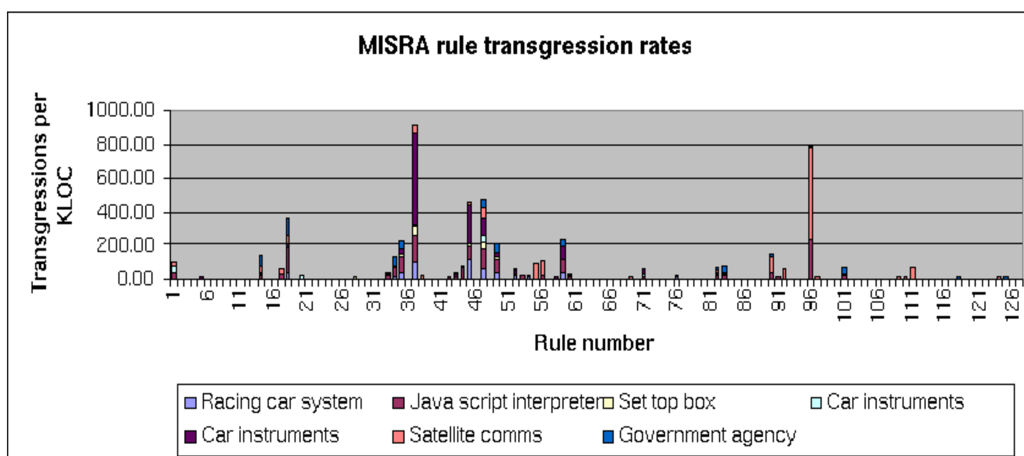

Figure 3: Cumulative transgressions of MISRA rules in a number of systems some of which appear in the population showed in Figure 2 with an average detection rate of around 12 per KLOC there. As can be seen, MISRA has some very noisy rules which do not reflect observed static fault rates well, (the four worst offenders are Rule 37, (no bit operations on signed int), Rule 96 (arguments to macros must be parenthesized) , Rule 47, (no dependence on precedence) and rule 18 (numeric constants should have an appropriate suffix).)

In other words even though MISRA C makes a significant effort to track category B rules, detection rates against it are still around 50 times those reported for documented fault modes as shown in Figure 2. By definition, an entirely category B.2 related standard should be much closer to Figure 2 than this. We can therefore define a rather loose but useful concept of signal to noise ratio for a standard as:-

$$S=(B/D)$$

where S is the approximate signal to ratio, signal here meaning the ability to find known defects, B is the base rate at which known fault modes of a programming language appear in a population as shown in studies like Hatton (1995) and D is the detection rate of a standard where 'rate' is taken to mean the total number of transgressions per 1000 lines of executable code.

Taking into account the variation in interpretation of the standard reported by Parker et. al. (2001), the signal to noise ratio of MISRA C is currently in the range 0.01 – 0.1. Low signal to noise ratios mean that engineers in simple terms, 'can't see the wood for the trees'. Real and perhaps serious defects are too easily hidden in a wealth of other less useful information. The problem is well known in static analysis but as can be seen, not much progress has been made in resolving it and it should be recalled that MISRA C is at least attempt ing to concentrate on category B issues unlike many other standards. Category A based standards will have a much lower signal to noise ratio in the area of avoiding defect.

In the case of MISRA C, after 4 years of experience with the application of this standard, the MISRA steering committee, a consortium of members from both industry and academia is now attempting to resolve these issues. It remains to be seen how successful this will be but an improvement in the signal to noise ratio of a factor of 10 will be a very significant step forward if it can be achieved. When the updated standard appears, this experiment will be repeated to measure the progress.

8.Conclusions

Safer language subsets are an important step forward in the use of programming languages to utilise existing knowledge about common failures to avoid their future occurrence.

This paper attempts to define the ground rules for a successful safer subset by analysing deficiencies in previous standards and by further analysis of a recent successful standard which nevertheless still has some deficiencies. The paper first of all defines a taxonomy of rule types into category A, stylistic, and categories B.1, (folklore but notionally based on failure) and B.2, (objective evidence of failure). The paper then proposes that by definition safer language subsets should consist only of category B rules with the difference between B.1 and B.2 clearly marked.

The paper goes on to stress the importance of the concepts of rule incorrectness, rule redundancy, rule cross-talk, rule decidability and rule atomicity in the production of a safer language subset and the measurement of conformance to such a subset. An as yet incomplete example of an exemplary test suite being developed alongside the MISRA C standard is given by Hatton (2002). Only when the above concepts are used to firm up the wording of a safer language subset can such a suite be called a conformance suite.

Finally, this paper attempts to quantify the concept of signal to noise ratio in such a standard. It is hoped that these definitions can be made more precise in future but for now they reflect empirical observations well.

9.References

Beizer, B. (1990) "Software Testing Techniques", Van Nostrand, ISBN 0-442-20672-0

Hatton, L. (1995) "Safer C: developing software in high-integrity and safety-critical systems", McGraw-Hill, ISBN 0-07-707640-0.

Hatton, L. (1997) "The T experiments: errors in scientific software", IEEE Computational Science and Engineering, 4(2), p. 27-38.

Hatton, L. (2002) "An exemplary test suite for the MISRA C standard", www.cs.ukc.ac.uk/national/SLS

Koenig, A. (1989) "C traps and Pitfalls", Addison-Wesley, ISBN 0-201-17928-8

MISRA C guidelines (1998) ISBN 0-9524156-9-0, www.misra.org.uk

Parker, S. and “JJ (no name given)” (2001) “A comparison of MISRA C testing tools”, http://www.pitechnology.com/downloads/files/MISRA_C_tools.pdf

Pfleeger, S.L. and Hatton L. (1997) "Do formal methods really work ?", IEEE Computer, 30(2), p. 33-43.

Spuler, D. (1995) "C++ and C debugging, testing and reliability", Prentice-Hall, ISBN 0-13-308172-9

van der Linden, P. (1994) "Expert C Programming", Prentice-Hall, ISBN 0-13-177429-8.

Yourdon, E. (1979) "Classics in Software Engineering", Yourdon Press, ISBN 0-917072-14-6.

© Copyright, Les Hatton, UKC, 2002-, version $Date: 2003/09/15 15:57:09 $