TITLE: KriCode Research Report I: Comparative Study of MISRA-C Compliancy Checking Tools

Authors: Marijn Temmerman, Hugo Van Hove, Kris Bellemans
Project: TETRA IWT 090169

Version: 1.0
Date: October 6th 2010
Document status: Preliminary

This document contains the extensive report of work package WP4 of the KriCode TETRA project. The goal of WP4 is the comparative study of software tools for checking the compliancy of source code written in the programming language C with the MISRA-C:2004 coding rules.
Contents

Geen inhoudsopgavegegevens gevonden.

List of figures

Figure 1: classification of tool............................................................... 3
Figure 2: overview test process............................................................ 39
Figure 3: Structure of the violations database...................................... 41
Figure 4: Contents Excel workbooks per rule........................................ 43
Figure 5: results for rule 11.1 ............................................................. 45
Figure 6: results for rule 2.3 ............................................................... 46
Figure 7: representation precision and recall ........................................ 49
Figure 8: precision and recall for PC-Lint ............................................ 49
Figure 9: Appendix A ........................................................................... 53
Figure 10: Appendix B .......................................................................... 54
Figure 11: Appendix C .......................................................................... 55
Figure 12: Appendix D .......................................................................... 56
Figure 13: Appendix E .......................................................................... 57
Figure 14: Appendix F .......................................................................... 58
Figure 15: Appendix G .......................................................................... 59
Figure 16: Appendix H .......................................................................... 60
Figure 17: Appendix I .......................................................................... 61
Figure 18: Appendix J .......................................................................... 62
Figure 19: Appendix K .......................................................................... 63
Figure 20: Appendix L .......................................................................... 64
Figure 21: Appendix M .......................................................................... 65
Figure 22: Appendix N .......................................................................... 66
Figure 23: Appendix O .......................................................................... 67
Figure 24: Appendix P .......................................................................... 68
Figure 25: Appendix Q .......................................................................... 69
Figure 26: Appendix R .......................................................................... 70
Figure 27: Appendix S .......................................................................... 71
Figure 28: Appendix T .......................................................................... 72
Figure 29: Appendix U .......................................................................... 73
Figure 30: Appendix V .......................................................................... 74
Figure 31: Appendix W .......................................................................... 75
Figure 32: Appendix X .......................................................................... 76
Figure 33: Appendix Y .......................................................................... 77
Figure 34: Appendix Z .......................................................................... 78
Figure 35: Appendix AA ......................................................................... 79
Figure 36: Appendix AB ........................................................................ 80

List of tables

Table 1: List of selected tools ................................................................. 3
Table 2: Selected rules ........................................................................... 12
1. Selection of tools

To select the tools in this study, an internet search was executed. The websites of the producers were used, as well as some articles on comparison of static analysis tools. We only have included those tools for which it is explicitly mentioned that the tool supports MISRA-C. The tools which we encountered during the preliminary study, contained code generating tools and code testing tools.

Code generating tools use an alternative way of representing programs (e.g. UML state diagrams, matlab-simulink, sysML, ...). From this representation, MISRA-C compliant code can be generated. Our industrial partners were mainly interested in testing C code they had written themselves, and less interested in generated C code.

The code testing tools can be subdivided in tools which can be used as standalone applications or tools which are part of a compiler.

Practical considerations also limited the number of tools which could be tested, e.g. availability of an evaluation license, man hours at the disposal of the project.

![Figure 1: classification of tools](image)

The following 8 tools were used in this study:

- Development Assistant for C (DAC) - RistanCASE
- IAR embedded workbench
- Parasoft C++test
- QAC - ProgrammingResearch (PhaedruS SystemS)
- PC-Lint - Gimpel
- LDRA Testbed
- Prevent - Coverity
- Klocwork Insight

| Table 1: List of selected tools |

2. Test criteria
The static code analysis tools have been empirically investigated on a number of criteria which can be categorized as soft (subjective) and hard (objective). In this study the following criteria have been used:

**Soft criteria:**
1. ease of integration in the existing development environment
2. usability: the ease with which people can employ a particular tool
3. extensibility: Can the user extend the program easily with extra functionality?
4. quality of violation messages
5. command-line functionality (for automatisation)
6. extra functionality, e.g. metrics?

**Hard criteria:**
1. correctness: Are the violations reported by the tool real violations (true positives)
2. completeness: Are all violations present in the code, reported by the tool? (no false negatives)

### 3. Soft criteria

#### 3.1. Test set-up of soft criteria

For every criterion a number of questions have been expressed. The answers to these questions will give an idea of how well a certain tool performs generally and meets a given criterion in particular. However, such a study is by definition informal and thus incomplete; it has nowhere the intention to make any judgments nor a ranking regarding the above criteria vis-à-vis the tools studied here. Any interpretation of this study, we leave at the reader’s discretion.

The following questions have been coupled to the various criteria:

- **Usability and adaptability:** Needs for static code analysis may vary throughout the development process. At a certain stage, code needs to be analyzed for a limited set of coding rules, whereas at other stages extra coding standards need to be included in the analysis. Does the tool support such varying needs?
  - Q1: Is it possible to switch rules off and on, individually or in group?
  - Q2: Is it possible to analyze an entire project?
  - Q3: Is it possible to analyze a part of a project or to exclude separate files?
  - Q4: How is the processing of large amounts of messages supported?

- **Quality of error and warning messages:** Quality is defined here as clarity and user-friendliness of the messages produced. Are the messages clear and easy to understand? Does the user know what's causing the error or warning and does he know what needs to be done to resolve them?
  - Q5: What is the quality of the error and warning messages?

- **Ability to integrate tool into development environment:** As the users of Kricode user commission use different tools and tool chains, it is important to investigate whether the tool can be integrated into a certain tool chain.
  - Q6: Are the existing development environments supported? Is it possible to integrate the static code analysis tool into the IDE? Which operating systems are supported?

- **Automation (through CLI):** Static code analysis is more and more automated within the development process by means of batch scripts and CLI-based tools. Therefore it is important to find out whether a tool can be operated via a command-line interface.
  - Q8: Is a command-line version available?

- **Extra functionalities:** Such tools tend not to be limited to static code analysis, but provide extra features such as reporting functionalities, teamwork functionality, e.g. svc
support/integration, bug tracking and other features that support and promote any collaboration during software development, and management functionalities, for example software metrics or possibilities for the project manager to follow up on the reported violations.

- Q9: What are the extra functionalities?
3.2. Test results for soft criteria

- Usability and adaptability

**Q1. Is it possible to switch rules off and on, individually or in group?**

<table>
<thead>
<tr>
<th>From the command line:</th>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individually</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>In groups</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>The user can compose groups</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From the GUI:</th>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individually</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No GUI</td>
</tr>
<tr>
<td>In groups</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No GUI</td>
</tr>
<tr>
<td>The user can compose groups</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No GUI</td>
</tr>
</tbody>
</table>

**Conclusion:** From the above answers, it is clear that most tools support the flexibility to customize the static code analysis. About half of the tools focus make this customization accessible only through a GUI; PC-Lint is the exception as it only features a CLI.

**Q2. Is it possible to analyze an entire project?**

**Q3. Is it possible to analyze a part of a project or to exclude separate files?**

<table>
<thead>
<tr>
<th></th>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire project</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Select part of an object</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Exclude files</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Is it possible to add custom rules?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Conclusion:** All tools provide the possibility to analyze an entire or parts of a project. Moreover, most tools also support the possibility to add custom rules; Raincode even features an elaborate framework to create and alter custom coding rules.

**Q4. How is the processing of large amounts of messages supported?**

**DAC:**

No direct support. DAC does not provide any functionalities for further processing of the output nor can the output be copied or redirected to files for further processing by means of external tools. There is, however, the possibility to limit the number of warnings and errors that should be reported.

**C++test:**

Output messages are organized according to MISRA rule and severity. Analysis output can be assembled in a XML/HTML generated report, a PDF-file or any other format specified by a XSL schema of your choice.

**Raincode:**

Static analysis results are stored in a mysql database, which can be freely accessed by any mysql client. This approach enables the user to process the results directly or through a 3rd party or in-house developed data processing tool.

**IAR:**

No support.
LDRA:
LDRA has no direct support for processing large amounts of messages.

QAC:
The standard message system in a QAC is organized into levels and groups. Exactly which message belongs to which group and level is defined in a text file called the Standard Message File. This file can be edited or replaced using a text editor. (QA-C-7.2-Win-UsersGuide.pdf p.29 and p.113)

PC-Lint:
No support.

Klocwork:
All results are reported through an interactive web interface. This makes a batch-like processing of output rather cumbersome.

- Quality of output

Q5. What is the quality of the error and warning messages?

<table>
<thead>
<tr>
<th></th>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location of violation given?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Possible to jump to this location (GUI)?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No GUI</td>
</tr>
<tr>
<td>Is a possible solution presented?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Clarity of the documentation of a violation</td>
<td>***</td>
<td>*</td>
<td>**</td>
<td>*</td>
<td>*</td>
<td>***</td>
<td>***</td>
<td>**</td>
</tr>
</tbody>
</table>

Conclusion: For the majority of the tools, the documentation of the violations could be better. Often only a minimal summary of the violation is provided. Most of the tools also fail to suggest a solution. C++test, QAC and LDRA are exceptions to the above as they provide an extensive set of documentation which allows the user to quickly understand and fix the violation.

- Ability to integrate

Q6. Are the existing development environments supported? Is it possible to integrate the static code analysis tool into the IDE? Which operating systems are supported?

<table>
<thead>
<tr>
<th>Dev. Software</th>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes</td>
</tr>
<tr>
<td>IAR</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Although LDRA officially supports Visual Studio, integration test failed
Support for operating systems which are used by the members of the user commission as primary development system:

<table>
<thead>
<tr>
<th>OS</th>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Linux</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Conclusion:** Eclipse, Visual Studio and IAR are three commonly used development environments in the Kricode user commission. Although a number of these tools serve as standalone programs, often a plugin for Eclipse and Visual Studio is provided, e.g. C++test, Raincode, etc. Integrating LDRA into Visual Studio failed as it wasn’t possible to verify any violations in the test code without starting LDRA.

Regarding the host operating system, only Raincode, Klocwork, QAC and LDRA support operating systems other than Windows.

**Q7. How long does the tool take to analyze a program?**

As our test programs are limited in size (+/- 30 LOC’s), the time required to analyze the test code is short (less than a minute). In this respect the test programs are not representative for real-life programs, nor can there any conclusions be deduced from these timings regarding the scalability of the static analysis tools.

- Automation

**Q8. Is a command-line version available?**

<table>
<thead>
<tr>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
• Extra features

**Q9. What is the additional functionality?**

<table>
<thead>
<tr>
<th></th>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there warnings for non-</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MISRA rules or problems?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reporting functionality</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Teamwork functionality</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Management functionality</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Automatic correction of</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>violations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Conclusion:** Most of the tools can be configured to check for coding rules other than MISRA rules. C++test for instance features an extensive set of coding standards. Other tools provide the possibility to import or create custom-defined coding rules.

With the exception of Raincode, all tools provide reporting functionalities, enabling the user to generate reports and organize the reported violations according to MISRA-rule, severity, assignment to a certain developer, etc. Some tools like C++test and Klocwork also show a tight coupling with teamwork functionalities such as version control. All tools, however, must bear the support for automatic correction of violations.

**Q10. Customer support, helpdesk?**

<table>
<thead>
<tr>
<th>C++test</th>
<th>DAC</th>
<th>Raincode</th>
<th>Klocwork</th>
<th>IAR</th>
<th>QAC</th>
<th>LDRA</th>
<th>PC-Lint</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>***</td>
<td>**</td>
<td>***</td>
<td>***</td>
<td>***</td>
<td>***</td>
<td>**</td>
</tr>
</tbody>
</table>
Q11. Additional remarks:

**IAR:**
The MISRA functionality for IAR Embedded Workbench is integrated into the compiler of the Workbench.
Rule checks are executed during compilation. This implies that rules can only be checked on code for targets supported by IAR Embedded Workbench.

**DAC:**
For every piece of code that needs to be checked, a new project needs to be created. Every project needs to be reconfigured.
Setting up a medium size project (e.g. FreeRTOS) takes two to three hours.
The absence of functionalities to process output, and the lack of support for interaction with third party tools makes Development Assistant not suited as a static analysis tool for medium to large scale projects.
DAC is non-deterministic in such a way that different analysis’s return different results/violations.
Customer support is unresponsive. All communication with Ristancase regarding licensing, pricing and customer support remained unanswered.

**PC-Lint:**
Support is only available through email. There is a tech support faq and a discussion forum. No helpdesk available.
4. Hard criteria

Completeness and correctness are the two major criteria for which we, in an objective way, assess the tools. Completeness as in “Are all violations in a certain piece of code reported?”; correctness as in “Are the reported violations truly an infraction against a certain MISRA rule for that piece of code?”.

Regarding the evaluation of these hard criteria, a test method has been drawn up which has the objective to investigate the completeness and correctness of each static code analysis tool vis-à-vis the MISRA violations it generates.

4.1. Test method setup

The initial intention was to have three kinds of test code:

1. For every MISRA-C rule, a few short pieces of code to assert if violations against this rule were recognized. These pieces of code will be called probes.
2. The code of FreeRTOS, to test the different tools on an interdependent set of files
3. The code of the linux kernel, to assess the relative speed of the tools

At first, an attempt was made to write probes for every rule. However, this task turned out to take too long in the context of this project. For example the rule 5.1: “Identifiers (internal and external) shall not rely on the significance of more than 31 characters.” results in 51 probes, if we try to cover every case suggested by the C90 standard. The reason is that identifiers are used to name different kinds of syntactical classes.

How are identifiers used in C90?

1. as enumeration constants p186
2. struct or union tags p190
3. enum tag p190
4. typedef name p191
5. label name
6. #ifdef identifier p193
7. ifndef
8. #define identifier
9. #undef
10. variable name p188
11. function name p188
12. names of struct members

To exhaustively test this rule, all combinations of the 12 applications of identifier in the standard should be tested.

Therefore it was necessary to select a feasible sample from the MISRA-C rules.

4.2. Selection of rules

The set of rules used in this study was selected in cooperation with a panel of our industrial partners. First, we asked these partners to select rules which they considered to be important. The panel was split into five groups. Each group was given a mutually exclusive part of the rules. Each group was asked to choose four rules. The selected rules per group were:

| Group | 5.2 | 6.1 | 9.1 | 9.2 | 8.10 | 8.12 | 10.2 | 10.5 | 14.1 | 14.7 | 12.6 | 12.4 | 15.2 | 16.4 | 16.10 | 17.2 | 17.6 | 19.4 | 19.5 | 19.10 | 19.7 | 20.2 |
|-------|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|

Two groups were unable to select fewer than five rules.
In a second stage we consulted Jan Verbeke, who has several years of experience in this field. He advised us to classify the rules into three groups:

1. crashing: rules which can cause a crash if violated
2. maintainability: these rules decrease the probability that a programmer introduces an error when changing code
3. portability: improve portability

From the 22 rules above we have selected 7 rules which were in the crashing or maintainability groups. (8.12; 9.1; 12.4; 14.7; 15.2; 17.6; 19.10). We added 4 more rules which are in the crashing or maintainability group in order to get a more evenly spread, representative sample of the rules.

Our final selection of rules is:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>The character set and the corresponding encoding shall be documented.</td>
</tr>
<tr>
<td>8.12</td>
<td>When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.</td>
</tr>
<tr>
<td>9.1</td>
<td>All automatic variables shall have been assigned a value before being used.</td>
</tr>
<tr>
<td>11.1</td>
<td>Conversion shall not be performed between a pointer to a function and any type other than an integral type.</td>
</tr>
<tr>
<td>12.8</td>
<td>The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.</td>
</tr>
<tr>
<td>14.7</td>
<td>A function shall have a single point of exit at the end of the function.</td>
</tr>
<tr>
<td>15.2</td>
<td>An unconditional break statement shall terminate every non-empty switch clause.</td>
</tr>
<tr>
<td>15.3</td>
<td>The final clause of a switch statement shall be the default clause.</td>
</tr>
<tr>
<td>16.6</td>
<td>The number of arguments passed to a function shall match the number of parameters.</td>
</tr>
<tr>
<td>17.6</td>
<td>The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.</td>
</tr>
<tr>
<td>19.10</td>
<td>In the definition of a function-like macro each instance of a parameter shall be enclosed in parenthesis unless it is used as the operand of # or ##.</td>
</tr>
</tbody>
</table>

Other studies have been performed in this area. (see [1] and [2]) However, our study is the first one oriented mainly at the MISRA-C rules, with a industry driven process to select the rules to be tested.

4.3. Testcode per rule

Depending on the complexity of the rules one or more were written. So for example for

Rule 2.3(r): The character sequence /* shall not be used within a comment

There is only one file with the following content:

```c
1 /*
2 Rule 2.3(r)
3 The character sequence /* shall not be used within a comment.
4 */
5
6 #include <stdio.h>
7 typedef int int32_t;
8
9 void mc2main_0203V001NonCompliant_01 ( void );
10 void mc2main_0203V001NonCompliant_01 ( void )
```
Expected results:
Line 3 tp
Line 14 tp
Line 19 tp
Line 21 tp

This gives several possibilities of the compliance or non-compliance of the tool for this rule.

Remarks:
- As a support in deciding on probes the PC-Lint tool was used.
- For the definition of terms, the C90 standard was used. For example for terms like linkage, side effect, ...

**Rule 8.12 (r)** When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

**Filename**: mc2main_0812V001NonCompliant_02.h

```c
1 /*
2  Rule 8.12 (r)
3  When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
4 */
5 ifndef mc2main_0812V001NonCompliant_02_h
6 #define mc2main_0812V001NonCompliant_02_h
7 8
typedef int int32_t;
9 10 #endif
11 12 13 14
```

**Filename**: mc2main_0812V001NonCompliant_01.c

```c
1 /*
2  Rule 8.12 (r)
3  When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
4 */
5 #include <stdio.h>
6 #include "mc2main_0812V001NonCompliant_02.h"
7 8 extern int32_t a[];
9 10 int int32_t main(void)
11 {
12 
```
13  a[0]=0;
14  a[1]=a[0];
15  return 0;
16 }

**Expected results:** Line 9: tp

By using a global variable, with extern modifier, it is sure that this array has external linkage. This example shows also that the initialization only defines size if {} are used.

**Rule 9.1 (r)** All automatic variables shall have been assigned a value before being used.

According to C90, there are two storage durations for variables:

- static
- automatic

static storage duration means that the memory for this variable exists during the full runtime of the program. **This is true for** (see C90 6.1.2.4 p28)
  - variables with file scope
  - variables with block scope with static in front

- The first probe for this rule is a compliant test. The tools should not give violations for static variables.

**Filename:** mc2main_0901V001Compliant_05.c

1 /*
2 Rule 9.1 (r)
3 All automatic variables shall have been assigned a value before being used.
4 */
5
6 #include <stdio.h>
7
8 typedef int int32_t;
9
10 int32_t d;
11
12
13 int32_t main(void)
14 {
15   printf("%d\n",d);
16   return 0;
17 }
18 }

**Expected results:** no violations

For the non-compliant probes, a distinction has been made between different kinds of variables, i.e. arrays and structs. And there are also probes which test if a variable which has been passed to a function, and has not been initialised are recognized as violations.

- Test for struct with one member

**Filename:** mc2main_0901V001NonCompliant_02.c

1 */
2 Rule 9.1 (r)
3 All automatic variables shall have been assigned a value before being used.
4 */
6 #include <stdio.h>
8 typedef int int32_t;
10 struct str{
11   int32_t i;
12 };
14
15 int32_t main(void)
16 {
17   struct str s;
18   printf("%d\n",s.i);
20   return 0;
22 }

Expected results: Line 19

- Test for struct with two members

Filename: mc2main_0901V001NonCompliant_03.c
1 /*
2 Rule 9.1 (r)
3 All automatic variables shall have been assigned a value before being used.
4 */
6 #include <stdio.h>
8 typedef int int32_t;
10
11 struct str2{
12   int32_t i;
13   int32_t j;
14 };
16
17 int32_t main(void)
18 {
19   struct str2 s2;
20   s2.i=1;
22   printf("%d\n",s2.j);
23   return 0;
24 }

Expected results: Line 22

- Test for array

Filename: mc2main_0901V001NonCompliant_04.c
1 /*
2 Rule 9.1 (r)
All automatic variables shall have been assigned a value before being used.

* /*

#include <stdio.h>

typedef int int32_t;

void mc2main_0812V001NonCompliant_01 ( void );

void mc2main_0812V001NonCompliant_01 ( void )
{
  int32_t ar1[3];
  printf("%d\n",ar1[1]);
}

int32_t main(void)
{
  mc2main_0812V001NonCompliant_01();
  return 0;
}

Expected results: Line 18

- Test for variable passed to function

#include <stdio.h>

typedef int int32_t;

void mc2main_0812V001NonCompliant_01 ( int32_t i );

void mc2main_0812V001NonCompliant_01 ( int32_t i )
{
  printf("%d\n",i);
}

int32_t main(void)
{
  int32_t v;
  mc2main_0812V001NonCompliant_01(v);
  return 0;
}

Expected results: Line 22
QAC correctly identifies use of the unset variable 'v' in line 22 which is the where the problem originates.
• Test for variable passed to function with function definition in another file

**Filename:** mc2main_0901V001NonCompliant_07_2.h

```c
1 #ifndef mc2main_0901V001NonCompliant_07_2_h
2 #define mc2main_0901V001NonCompliant_07_2_h
3 typedef int int32_t;
4 void mc2main_0812V001NonCompliant_01 ( int32_t i  );
5
6 #endif
```

**main file:** mc2main_0901V001NonCompliant_07_1.c

```c
1 /*
2 Rule 9.1 (r)
3 All automatic variables shall have been assigned a value before being used.
4 */
5
6 #include <stdio.h>
7 #include "mc2main_0901V001NonCompliant_07_2.h"
8
9 int32_t main(void)
10 {  
11   int32_t v;
12   mc2main_0812V001NonCompliant_01(v);
13   return 0;
14 }
```

**File with function definition:** mc2main_0901V001NonCompliant_07_2.c

```c
1 /*
2 Rule 9.1 (r)
3 All automatic variables shall have been assigned a value before being used.
4 */
5
6 #include <stdio.h>
7 #include "mc2main_0901V001NonCompliant_07_2.h"
8
9 void mc2main_0812V001NonCompliant_01 (int32_t i )
10 {  
11   printf("%d\n",i);
12 }
```

**Expected results:** Line 17 in file mc2main_0901V001NonCompliant_07_1.c

• The last probe is a combination of all the previous tests.

**Filename:** mc2main_0901V001NonCompliant_01.c

```c
1 /*
2 Rule 9.1 (r)
```
All automatic variables shall have been assigned a value before being used.

```c
#include <stdio.h>

typedef int int32_t;

struct str{
    int32_t i;
};

struct str2{
    int32_t i;
    int32_t j;
};

int32_t d;

void mc2main_0812V001NonCompliant_01 ( void );

void mc2main_0812V001NonCompliant_01 ( void )
{
    int32_t v;
    int32_t ar1[3];

    printf("%d\n",v);
    printf("%d\n",ar1[1]);
}

int32_t main(void)
{
    int32_t v;
    struct str s;
    struct str2 s2;
    s2.i=1;
    printf("%d\n",v);
    printf("%d\n",d);
    printf("%d\n",s.i);
    printf("%d\n",s2.j);
    return 0;
}
```

**Expected results:**
Line 36  
Line 39  
Line 25  
Line 26

**Rule 11.1 (r)** Conversion shall not be performed between a pointer to a function and any type other than an integral type.

- The first example for this rule is a simple compliant example of a conversion from a function pointer to an integral type, in this case an integer.
Filename: mc2main_1101V001Compliant_01.c

1 /*
2  Rule 11.1 (r)
3  Conversion shall not be performed between a pointer to a function and any type other than an integral type.
4 */
5
6 #include <stdio.h>
7
typedef int int32_t;
typedef char char_t;

12 void (*fp)(int32_t i,char_t c)=NULL;
13 void f(int32_t i,char_t c);
14 void f(int32_t i,char_t c)
15 {
16   printf("in f: %d,%c\n",i,c);
17 }
18
19 int32_t main(void)
20 {
21   fp=f;
22   (*fp)(1,'Z');
23   return 0;
24 }

Expected results: No violations

- For the non-compliant probes, i.e. test for conversion to non-integral types, two cases have been considered: float and double.

To avoid violations on MISRA rules concerning the built-in types
typedef float float_t;
typedef double double_t;
are being used.

To avoid type errors a cast to float and double have been used:

Filename: mc2main_1101V001NonCompliant_03.c

1 /*
2  Rule 11.1 (r)
3  Conversion shall not be performed between a pointer to a function and any type other than an integral type.
4 */
5
6 #include <stdio.h>
7
typedef int int32_t;
typedef char char_t;
typedef float float_t;

13 void (*fp)(int32_t i,char_t c)=NULL;
14 void f(int32_t i,char_t c);
void f(int32_t i, char_t c)  
{
  printf("in f: %d,%c\n", i, c);
}

int32_t main(void)  
{
  float_t fl=0;
  fp=f;
  (*fp)(1,'Z');
  fl=(float_t) fp;
  printf("%f\n", fl);
  return 0;
}

#include <stdio.h>

typedef int int32_t;
typedef char char_t;
typedef float float_t;
typedef double double_t;

void (*fp)(int32_t i, char_t c)=NULL;

int32_t main(void)  
{
  double_t d=0;
  fp=f;
  (*fp)(1,'Z');
  d=(double_t) fp;
  printf("%lf\n", d);
  return 0;
}

Rule 12.4 (r) The right-hand operand of a logical && or || operator shall not contain side effects.

According to C90 there are four kinds of side effect (see C90 5.1 5.2 p13):
   1. accessing a volatile object
   2. modifying an object
3. modifying a file
4. calling a function which does 1, 2 or 3

- **In the first probe, a volatile object is being accessed:**

*Filename:* mc2main_1204V001NonCompliant_04.c

```c
/*
Rule 12.4 (r)
The right-hand operand of a logical && or || operator shall not contain side effects.
*/

#include <stdio.h>
typedef int int32_t;

int32_t b=0;

int32_t main(void)
{
  volatile int32_t n=1;
  int32_t a=0;

  printf("Geef b:\n");
  scanf("%d", &b);
  if((a==b) && ((a=n)==1)){
    printf("\n&& is true\n");
  }
  return 0;
}
```

*Expected results:* Line 22

- **In the second probe, an object is modified:**

*Filename:* mc2main_1204V001NonCompliant_05.c

```c
/*
Rule 12.4 (r)
The right-hand operand of a logical && or || operator shall not contain side effects.
*/

#include <stdio.h>
typedef int int32_t;

int32_t b=0;

int32_t main(void)
{
  int32_t n=1;
  int32_t a=0;
```
In the following code, the side effect of opening a file has been tested. Because this side effect has to occur in the right-hand side of a && or || operator, the code opening the file is used in a function.

**Filename:** mc2main_1204V001NonCompliant_03.c

```c
#include <stdio.h>

typedef int int32_t;

int32_t f(void);
int32_t f(void) {
    int32_t n=1;
    FILE *fp;
    fp=fopen("text.txt","3");
    if( fp==NULL) {
        printf("File not found\n");
    } else{
        fclose(fp);
    } return(n);
}

int32_t b=0;

int32_t main(void) {
    int32_t a=0;
    printf("Geef b:\n");
    scanf("%d",&b);
    if((a==b) && ((a==n)==1)){
        printf("\n&& is true\n");
    }
    return 0;
}

Expected results: Line 22

- In the following code, the side effect of opening a file has been tested.

/*
Rule 12.4 (r)
The right-hand operand of a logical && or || operator shall not contain side effects.
*/

#include <stdio.h>

typedef int int32_t;

int32_t f(void);
int32_t f(void) {
    int32_t n=1;
    FILE *fp;
    fp=fopen("text.txt","3");
    if( fp==NULL) {
        printf("File not found\n");
    } else{
        fclose(fp);
    } return(n);
}

int32_t b=0;

int32_t main(void) {
    int32_t a=0;
    printf("Geef b:\n");
    scanf("%d",&b);
    if((a==b) && ((a==n)==1)){
        printf("\n&& is true\n");
    }
    return 0;
}
The last two probes contain the access for a volatile object and the modification for an object from a function:

**Filename:** mc2main_1204V001NonCompliant_01.c

```c
#include <stdio.h>

typedef int int32_t;

int32_t f(void);
int32_t f(void)
{
    volatile int32_t n=1;
    return(n);
}

int32_t b=0;

int32_t main(void)
{
    int32_t a=0;
    printf("Geef b:\n");
    scanf("%d",&b);
    if((a==b) && (f() == 1)){
        printf("\n&& is true\n");
    }
    return 0;
}
```

**Expected results:** Line 38

- The last two probes contain the access for a volatile object and the modification for an object from a function:

**Filename:** mc2main_1204V001NonCompliant_02.c

```c
#include <stdio.h>

typedef int int32_t;

int32_t f(void);
int32_t f(void)
{
    volatile int32_t n=1;
    return(n);
}

int32_t b=0;

int32_t main(void)
{
    int32_t a=0;
    printf("Geef b:\n");
    scanf("%d",&b);
    if((a==b) && (f() == 1)){
        printf("\n&& is true\n");
    }
    return 0;
}
```

**Expected results:** Line 30

- The last two probes contain the access for a volatile object and the modification for an object from a function:
#include <stdio.h>

typedef int int32_t;

int32_t n=1;
int32_t f(void);
int32_t f(void)
{
    n=n+1;
    return(n);
}

int32_t b=0;

int32_t main(void)
{
    int32_t a=0;

    printf("Geef b:\n");
    scanf("%d",&b);
    if((a==b) && (f() == 1)){
        printf("\n&& is true\n");
    }
    return 0;
}

Expected results: Line 31

**Rule 14.7 (r) A function shall have a single point of exit at the end of the function.**

*Filename: mc2main_1407V001NonCompliant_01.c*
Expected results:
Line 19
Line 23

Filename: mc2main_1407V001NonCompliant_02.c
1 /*
2 Rule 14.7 (r)
3 A function shall have a single point of exit at the end of the function.
4 */
5
6 #include <stdio.h>
7
typedef int int32_t;
9 int int32_t b;
11 int32_t f(void);
13 int32_t f(void)
14 {
15   int32_t a=1;
16   scanf("%d",&b);
18   if(a==b){
19     return(1);
20   } else
22   {
23   }  
25
26 int32_t main(void)
27 {
28   int32_t i=0;
30   i=f();
32   return 0;
33 

Expected results:
Line 19
Line 32

Filename: mc2main_1407V001NonCompliant_03.c
1 /*
2 Rule 14.7 (r)
3 A function shall have a single point of exit at the end of the function.
#include <stdio.h>

typedef int int32_t;

int32_t b;

int32_t f(void);

int32_t f(void)
{
  int32_t a=1;
  scanf("%d", &b);
  if(a==b){
  }
  else{
    return(0);
  }
}

int32_t main(void)
{
  int32_t i=0;
  i=f();
  return 0;
}

Expected results:
Line 23
Line 32

Filename: mc2main_1407V001NonCompliant_04.c
```c
int32_t main(void)
{
    int32_t i=0;
    i=f();
    return 0;
}
```

**Expected results:**

Line 19

Line 23

**Filename:** mc2main_1407V001NonCompliant_05.c

```c
#include <stdio.h>

typedef int int32_t;

int32_t b;

int32_t f(void);
int32_t f(void)
{
    int32_t a=1;
    scanf("%d", &b);
    if(a==b){
        return(1);
    }
    else{
        return(0);
    }
    return(2);
}

int32_t main(void)
{
    int32_t i=0;
    i=f();
    return 0;
}
```

**Expected results:**

Line 19

Line 23
**Rule 15.2 (r)** An unconditional *break* statement shall terminate every non-empty switch clause.

A distinction has been made between ordinary switch clauses and the default clause. The first probe contains a test on the missing break for a default clause:

*Filename:* mc2main_1502V001Compliant_01.c

```c
1 /*
2 Rule 15.2 (r)
3 An unconditional break statement shall terminate every non-empty switch clause.
4 */
5
6 #include <stdio.h>
7
typedef int int32_t;
8
9 int32_t b;
10
11 void f(void);
12 void f(void)
13 {
14   scanf("%d","b");
15   switch(b){
16     case 1: printf("1
");
17       break;
18     case 2: printf("2
");
19       break;
20     default: printf("end
");
21   }
22 }
23
24 int32_t main(void)
25 {
26   f();
27   return 0;
28 }
```

**Expected results:** Line 24

The second probe tests on a missing break on an empty clause, which should be compliant.

*Filename:* mc2main_1502V001Compliant_04.c

```c
1 /*
2 Rule 15.2 (r)
3 An unconditional break statement shall terminate every non-empty switch clause.
4 */
5
6 int32_t main(void)
7 {
8   f();
9   return 0;
10 }
```
```c
#include <stdio.h>
typedef int int32_t;

int32_t b;

void f(void);
void f(void)
{
    scanf("%d",&b);
    switch(b){
        case 1: printf("1\n");
        break;
        case 2:
            default: printf("end\n");
    }
}

int32_t main(void)
{
    f();
    return 0;
}
```

Expected results: Line 23

The third probe tests for a missing break in an ordinary switch clause

Filename: mc2main_1502V001NonCompliant_02.c

```c
/*
Rule 15.2 (r)
An unconditional break statement shall terminate every non-empty switch clause.
*/

#include <stdio.h>
typedef int int32_t;

int32_t b;

void f(void);
void f(void)
{
    scanf("%d",&b);
    switch(b){
        case 1: printf("1\n");
        break;
        case 2:
            default: printf("end\n");
```
23   default: printf("end\n");
24  }
25 }
26
27
28
29 int32_t main(void)
30 {
31   f();
32   return 0;
33 }

Expected results:
Line 22
Line 23

The fourth probe tests on missing break in the first switch clause.

Filename: mc2main_1502V001NonCompliant_03.c

1 /*
2 Rule 15.2 (r)
3 An uncoditional break statement shall terminate every non-empty switch clause.
4 */
5
6 #include <stdio.h>
7
typedef int int32_t;
8
9 int32_t b;
10
11 void f(void);
12 void f(void)
13 {
14   scanf("%d",&b);
15   switch(b){
16     case 1: printf("1\n");
17     case 2: printf("2\n");
18     break;
19     default: printf("end\n");
20   }
21 }
22
23
24 int32_t main(void)
25 {
26   f();
27   return 0;
28 }

Expected results:
Line 20
The last probe tests if the tool will recognize violations if switch statements are nested.

**Filename:** mc2main_1502V001NonCompliant_05.c

```c
/*
Rule 15.2 (r)
An uncoditional break statement shall terminate every non-empty switch clause.
*/

#include <stdio.h>

typedef int int32_t;

int32_t b;
int32_t c;

void f(void);

void f(void)
{
    scanf("%d", &b);
    scanf("%d", &c);
    switch(b){
        case 1: {
            switch(c){
                case 1: printf("1.1\n");
                case 2: printf("1.2\n");
                break;
                default: printf("1.end\n");
            }
            break;
        case 2: printf("2\n");
        break;
        default: printf("end\n");
    }
}

int32_t main(void)
{
    f();
    return 0;
}
```

**Expected results:**
Line 24
Line 27
Line 34
QAC correctly identifies an additional problem - a missing break statement following line 34.
QAC analysis
Rule 15.3 (r) The final clause of a switch statement shall be the default clause.

Filename: mc2main_1503V001NonCompliant_01.c

```c
#include <stdio.h>
typedef int int32_t;

int32_t b;

void f(void);
void f(void)
{
   scanf("%d", &b);
   switch(b){
      case 1: printf("1
\n");
          break;
      case 2: printf("2\n");
          break;
   }
}

int32_t main(void)
{
   f();
   return 0;
}
```

Expected results:
Line 24

Rule 16. 6(r) The number of arguments passed to a function shall match the number of parameters.

The test cases here are
- a function call which has a matching number of arguments
- a function with one argument to many
- a function call with one argument too few

Filename: mc2main_1606V001Compliant_01.c

```c
#include <stdio.h>
typedef int int32_t;

int32_t b;

void f(void);
void f(void)
{
   scanf("%d", &b);
   switch(b){
      case 1: printf("1\n");
          break;
   }
}

int32_t main(void)
{
   f();
   return 0;
}
```
#include <stdio.h>

typedef int int32_t;
typedef char char_t;

void f(int32_t i, char_t c);

void f(int32_t i, char_t c) {
   printf("%d %c\n", i, c);
}

int32_t main(void) {
   f(2, 'e');
   return 0;
}

/*
Rule 16. 6(r)
The number of arguments passed to a function shall match the number of parameters.
*/

#include <stdio.h>

typedef int int32_t;
typedef char char_t;

int32_t b;

void f(int32_t i, char_t c);
void f(int32_t i, char_t c) {
   printf("%d %c\n", i, c);
}

int32_t main(void) {
   f(2, 'e', 5);
   return 0;
}

/*
Rule 16. 6(r)
The number of arguments passed to a function shall match the number of parameters.
*/
Rule 16. 6 (r)  The number of arguments passed to a function shall match the number of parameters.

```c
#include <stdio.h>

typedef int int32_t;
typedef char char_t;

int32_t b;

void f(int32_t i, char_t c)
{
  printf("%d %c\n", i, c);
}

int32_t main(void)
{
  f(2);
  return 0;
}
```

Expected results: Line 26

**Rule 17. 6 (r)** The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Two cases have been considered for this rule. In the first case an array is returned as a pointer. Because the storage for the array is automatic and will be removed from the stack at the end of the function, the array object will not persist beyond the end of the function.

**Filename:** mc2main_1706V001NonCompliant_01.c

```c
#include <stdio.h>

typedef int int32_t;

int32_t* f(void);

int32_t* f(void)
```
int32_t ar[3];
return(ar);

int32_t main(void)
{
   int32_t *ar2=NULL;
ar2=f();
   return 0;
}

Expected results:
Line 29

In the second probe, the address of an automatic variable is returned. For the same reason as above
the variable will not persist beyond the end of the function.

Filename: mc2main_1706V001NonCompliant_02.c
Rule 19.10 (r) In the definition of a function-like macro each instance of a parameter shall be enclosed in parenthesis unless it is used as the operand of # or ##.

This last rule is the only one in our selection which considers the possible problems with the preprocessor.

The first probe is a straight forward test on the arguments of a macro not enclosed in parenthesis. This should give a violation for each instance of the parameters.

**Filename:** mc2main_1910V001NonCompliant_01.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MIN(x,y) \ 
5   (x < y ? x : y)
6
7 typedef int int32_t;
8
9 int32_t main(void)
10 {
11   printf("%d
\n",MIN(2,3));
12   return(0);
13 }
14
Expected results:
Line 5: 4 tp
```

The second and third probe are almost identical. In the second one there is a macro with two parameters without parenthesis. This should give two violations. The third probe has the same macro, but in this case the parameters are operands of #. This should give no violations.

**Filename:** mc2main_1910V001NonCompliant_02.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAC1(begin,end) \ 
5   printf("%d %d\n",begin,end)
6
7 typedef int int32_t;
8
9 int32_t main(void)
10 {
11   MAC1(2,3);
12   return(0);
13 }
14
Expected results:
```

```
Line 5: 2 tp
Filename: mc2main_1910V001NonCompliant_03.c

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAC1(begin,end) \\
5   printf("%s \n",#begin #end)
6
7 typedef int int32_t;
8
9 int32_t main(void)
10 {
11   MAC1("ee","ff");
12   return(0);
13 }
14

Expected results: No violations
QAC does not insist on parenthesising the arguments "begin" and "end" in the replacement text of this macro. This is a deliberate design decision because in this particular macro, parentheses are completely unnecessary in the context in which the arguments are used. This design decision is again an attempt to reduce "noise".
4.4. Tool setup

Once we are done writing the test code, it is ready to be subjected to a MISRA-C code analysis executed by the static analysis tools. Therefore the tools need to be setup properly and configured for the MISRA-C 2004 coding standard. When this is done, the actual analysis can be performed.

4.5. Processing of the results

![Figure 2: overview test process]

Fig. 2 gives an overview of the data-processing executed to obtain the results in the remainder of this report.

1. Source code: The source code used for these test has already been discussed.
2. Tool 1..8: every tool was applied to every test file.
3. The output of these test runs was entered into a database, partially using Python scripts,
4. Partially by hand
5. The structure of the database will be discussed shortly
6. From the database, data was extracted and entered into excel sheets (see below)
7. For every rule in our selection and for every file for that rule, every violation was marked true or false positive. For the selected rules, the false negatives were added.
8. A normalisation was performed.
9. These changes were added to the database.
10. False negatives derived from one tool were added to the others

- **Output processing**

(see figure 2: number 2: Tool 1...8)

During an analysis, static analysis tools generate a lot of output. Furthermore, the way in which the output is formatted, differs from tool to tool. As a result, the output of every tool needs to be handled differently. For some tools, we have developed small programs, written in Python programming language. Each program is tailored to process a specific tool’s output. From this output,
the program extracts and consolidates information regarding MISRA-C infractions and stores it in a MySQL database. For other tools the output was processed by hand (see figure 1: number 3 and 4).

If we regard two samples of output, one for the DAC tool and the other for PCLint, it is clear that there are considerable differences between the output of these tools.

For example, for the PC-lint and for the LDRA tool, for the file:

```
mc2main_1910V001NonCompliant_01.c
```

the output is presented below:

**Output PCLint**

```bash
\MISRA\documents\ComparativeStudyMISRAC\Tools\SourceCode\Rules\Rule19.10>lint-nt.exe -i"C:\lint" -i"C:\lint\lint" -i"C:\ProgramFiles\MicrosoftVisualStudio\VC\include" au-misra2.lnt +libh(stdio.h) -elib(*) -e974 -e829 -e751 -e754 -e534 mc2main_1910V001NonCompliant_01.c

--- Module: mc2main_1910V001NonCompliant_01.c (C)

#define MIN(x,y) \
mc2main_1910V001NonCompliant_01.c 5  Note 961: Violates MISRA 2004 Advisory Rule 19.7, Function-like macro defined

```bash
#... (2 < 3 ? 2 : 3)
printf("%d
",MIN(2,3));
```

mc2main_1910V001NonCompliant_01.c 14  Note 961: Violates MISRA 2004 Advisory Rule 12.1, dependence placed on C's operator precedence; operators: '<' and '?'

```bash
#... (2 < 3 ? 2 : 3)
printf("%d
",MIN(2,3));
```

mc2main_1910V001NonCompliant_01.c 14  Warning 506: Constant value Boolean  

[ MISRA 2004 Rules 13.7 and 14.1]

```bash
--- Wrap-up for Module: mc2main_1910V001NonCompliant_01.c
```

**Output LDRA for Testcase MISRA-rule r19_10:**

```bash
#include <stdio.h>
/* (M) STATIC VIOLATION : 130 S : MISRA-C:2004 20.8,20.9,20.12: Included file is not permitted. : 7F#include <stdio.h> */
#include <stdlib.h>
#define MIN(x,y) \
/* (M) STATIC VIOLATION : 78 S : MISRA-C:2004 19.10: Macro parameter not in brackets. : 9F#define MIN( x , y ) ( x < y ? x : y ) */
/* (M) STATIC VIOLATION : 78 S : MISRA-C:2004 19.10: Macro parameter not in brackets. : 9 */
/* (M) STATIC VIOLATION : 78 S : MISRA-C:2004 19.10: Macro parameter not in brackets. : 9 */
```
typedef int int32_t;

int32_t main(void)
{
    printf("%d\n",MIN(2,3));
    return(0);
}

Because the disparate output for every tool, it was decided to gather the data in a mysql database. On the next page we discuss the structure of this database. (see figure 2: number 5)

Figure 3: Structure of the violations database
The database consists of six tables, with the following data:

1. Table file:
   i. ID: a unique id for every row in this table
   ii. NAME: the filename as used in the file system
   iii. PATH: the path to this file
   iv. LOC: lines of code
   v. Project_ID: refers to the ID of the project table

2. Table message:
   i. ID: a unique id for every row in this table
   ii. TEXT: not used

3. Table project:
   i. ID: a unique id for every row in this table
   ii. NAME: Name of this project. A project consists of all the c source code files for 1 rule.
      The name of the project is the rule to which it refers.
   iii. Description: not used

4. Table rule:
   i. ID: a unique id for every row in this table
   ii. MISRA_CODE: code of the MISRA-C 2004 rule
   iii. TEXT: not used

5. Table tool:
   i. ID: a unique id for every row in this table
   ii. NAME: name of the tool
   iii. VENDOR: vendor for this tool

6. Table violation:
   i. ID: a unique id for every row in this table
   ii. RULE_ID: refers to the ID of the rule table
   iii. TOOL_ID: refers to the ID of the tool table
   iv. LINE: the line on which this violation is reported
   v. MESSAGE_ID: refers to the ID if the message table
   vi. FILE_ID: refers to the IF of the file table
   vii. TRUE_POS: 1 if the violation is a true positive, 0 otherwise
   viii. FALSE_POS: 1 if the violation is a false positive, 0 otherwise
   ix. FALSE_NEG: 1 if the violation is a false negative, 0 otherwise

From this database, an Excel sheet was produced for every rule. (see figure 2: number 6)
Figure 4: Contents Excel workbooks per rule

See figure 4 for an example of the contents of these files. The files contain from left to right:

1. The violation id
2. The MISRA number of the rule
3. The line
4. The file name
5. The violation message
6. And then 3 columns which indicate of the violation was a true positive, a false positive or a false negative

- **Result processing**

Once the outputs of these analysis’s, i.e. the reported violations against the MISRA coding standard, have been brought together in a database, we can start interpreting the results.

The results are handled in two ways:

1. **General approach by interpreting histograms**:
   
   The general approach in which the results are first queried per test program and then merged into histograms. From these histograms, general tendencies are derived.
   
   With this approach however, it is quite hard to interpret the results to the full extent.

2. **Objective tool assessment by means of Precision and Recall**
   
   In-depth approach in which the results are manually verified in the test code and compared to the predicted outcome. If a certain reported violation matches an expected result, it is classified as a true positive. If not, the reported violation is to be considered as a false positive. The same
goes for unreported infractions. When a violation isn’t reported when it should be, it is classified as a false negative. Otherwise it is regarded as a true negative. Determining the true/false positives/negatives, enables us to calculate the precision and recall [ref.] of a tool for a given MISRA rule.

4.6. General observations

The results of the tests for the rule set 2.3 to 19.10 have been comprised in histograms, which can be found in respectively Appendices A to K. For every test code a certain MISRA-rule is tested for, the histogram indicates which and what number of violations a tool found in this particular piece of test code. Analyzing, comparing and interpreting these histograms, isn’t very straightforward. Nevertheless a number of interesting tendencies could be made:

1. Test validation
   The first clearly noticeable tendency, is that the majority of the tools generate one or more violations against the MISRA rule for which the test has been developed. Establishing this trend allows to conclude that the test code has been properly developed and serves as a validation of our tests and test method.

2. Check on return values
   Only C++test and QAC check quite strictly on MISRA rule 16.10. This rule states that all return values of functions should be tested. As a result, the aforementioned tools generate a violation whenever a trivial printf function is used. This becomes clear in below table, of which the histograms are depicted in appendix D and E, and where a peak is noticeable on rule 16.10 as no other tool reports any violations against rule 16.10.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Occurrence</th>
<th>Tool</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++test</td>
<td>9</td>
<td>C++test</td>
<td>17</td>
</tr>
<tr>
<td>DAC</td>
<td>0</td>
<td>DAC</td>
<td>0</td>
</tr>
<tr>
<td>Raincode</td>
<td>0</td>
<td>Raincode</td>
<td>0</td>
</tr>
<tr>
<td>Klocwork</td>
<td>0</td>
<td>Klocwork</td>
<td>0</td>
</tr>
<tr>
<td>IAR</td>
<td>0</td>
<td>IAR</td>
<td>0</td>
</tr>
<tr>
<td>QAC</td>
<td>8</td>
<td>QAC</td>
<td>14</td>
</tr>
<tr>
<td>LDRA</td>
<td>0</td>
<td>LDRA</td>
<td>0</td>
</tr>
<tr>
<td>PC-Lint</td>
<td>0</td>
<td>PC-Lint</td>
<td>0</td>
</tr>
</tbody>
</table>

3. Some MISRA rules are harder to test
   For certain rules, big differences in the number of reported violations are noticeable. This indicates that these particular rules are harder to test in a uniform way. Rules 14.7 and 19.10 are good examples of this, which are shown in below table.
Table 2: Reported violations on MISRA rules 14.7 and 19.10

<table>
<thead>
<tr>
<th>Tool</th>
<th>Rule 14.7 Occurrence</th>
<th>Tool</th>
<th>Rule 19.10 Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++test</td>
<td>9</td>
<td>C++test</td>
<td>4</td>
</tr>
<tr>
<td>DAC</td>
<td>5</td>
<td>DAC</td>
<td>2</td>
</tr>
<tr>
<td>Raincode</td>
<td>8</td>
<td>Raincode</td>
<td>4</td>
</tr>
<tr>
<td>Klocwork</td>
<td>8</td>
<td>Klocwork</td>
<td>0</td>
</tr>
<tr>
<td>IAR</td>
<td>11</td>
<td>IAR</td>
<td>2</td>
</tr>
<tr>
<td>QAC</td>
<td>5</td>
<td>QAC</td>
<td>1</td>
</tr>
<tr>
<td>LDRA</td>
<td>5</td>
<td>LDRA</td>
<td>6</td>
</tr>
<tr>
<td>PC-Lint</td>
<td>10</td>
<td>PC-Lint</td>
<td>0</td>
</tr>
</tbody>
</table>

4. Cluster formation

When a violation against a certain MISRA rule is found, violations against rules of the same category are often reported as well. This cluster formation is shown in below graph and can be clearly distinguished for rule categories 11.x, 16.x, 20.x and 8.x. This indicates there’s either a certain overlap between rules of the same category, or the distinction between these rules is rather small, causing tools to report multiple violations.

Even though a number of trends could be distilled from these histograms, it is hard to make any judgments on the static analysis capabilities of these tools without examining the code and the several violations. Therefore the results need to be evaluated in a more exhaustive manner.
4.7. In detail observations

The need for a more in depth approach arose, when looking at the actual violation figures and the differences between them for a given tool and MISRA rule.

For instance, considering the above histogram of Rule 2.3, nearly every tool that has been tested, (has) found 4 violations against MISRA rule 2.3 as shown in Table 1.

Table 3: Number of violations found on MISRA rule 2.3

<table>
<thead>
<tr>
<th>Tool</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++test</td>
<td>4</td>
</tr>
<tr>
<td>DAC</td>
<td>4</td>
</tr>
<tr>
<td>Raincode</td>
<td>4</td>
</tr>
<tr>
<td>Klocwork</td>
<td>1</td>
</tr>
<tr>
<td>IAR</td>
<td>4</td>
</tr>
<tr>
<td>QAC</td>
<td>4</td>
</tr>
<tr>
<td>LDRA</td>
<td>4</td>
</tr>
<tr>
<td>PC-Lint</td>
<td>4</td>
</tr>
</tbody>
</table>

This is to be expected as this piece of test code was developed with exactly 4 violations against MISRA rule 2.3 inserted (see below code snippet).

```c
/*
Rule 2.3(r)
The character sequence /* shall not be used within a comment.
```
#include <stdio.h>
typedef int int32_t;

void mc2main_0203V001NonCompliant_01 ( void );
void mc2main_0203V001NonCompliant_01 ( void )
{
    /*ggggggggggggg/*gggggggggggg*/
}

int32_t main(void)
{
    /*vvvvvvv /*vvvvvvv*/
    /*yyyyyy*/
    /*jjjjjjjjjjjj/*ooo*/
    return 0;
}

/*
 * Expected results:
 * Line 3:  Tp
 * Line 14: Tp
 * Line 19: Tp
 * Line 21: Tp
*/

However, one tool, namely Klocwork, did report only one violation against this MISRA rule. Two possible conclusions may be deduced from this table. Either Klocwork is short on 3 occurrences (false negatives), which is the most likely conclusion as 4 violations have been provided in the test code, or the other tools have counted 3 occurrences too many (false positives). Such discrepancies require a manual verification to determine whether a violation, or absence thereof, is a true positives (tp), false positives (fp), true negatives (tn) or false negatives (fn). This is even more so when we consider the remaining MISRA-violations for the same test code; the corresponding histogram showing a rather diffuse image of the result set.

As discussed before, calculating the Precision and Recall based on the classification of the violations allows to compare the static code analysis capabilities of the tools. Therefore these scores have been mapped to below graphs with respect to the selected MISRA rules.

- Tool assessment by means of Precision and Recall

We will use the precision and recall statistical classifications as measures for resp. correctness or fidelity and completeness.
To be able to define precision and recall, we first need some additional terminology:
We say that a violation of a rule R reported on a line L is a true positive if there is indeed a violation of R at L
A false positive is the reporting of violation of R at L, when there is no violation.
We speak of a false negative if a violation is present, but not reported. Finally, we have a true negative if there is no violation reported on a line where there is no violation.
<table>
<thead>
<tr>
<th>Violation present</th>
<th>Violation not reported</th>
</tr>
</thead>
<tbody>
<tr>
<td>True positive tp</td>
<td>False negative fn</td>
</tr>
<tr>
<td>Violation not present</td>
<td>False positive fp</td>
</tr>
</tbody>
</table>

Now, with this terminology in place, we can define:

With the data present on our database we can calculate these measures. In the aforementioned Excel workbooks, every violation was checked manually to decide if it was a true or false positive. For every selected rule the false negatives were added. Additionally, a Python script was written and used to find more false negatives. The method used is as follows:

For every true positive fore rule R reported by tool A on line N, the script checks if tool B also has a true positive on N for R. If not, a false negative for R on N is added to the violations of tool B. This process is repeated for each combination of tools.

To be able to compare the tools under investigation, the data had to be normalized. Let us consider, for example,

**Rule 14.7: A function shall have a single point of exit at the end of the function.**

Some tools return a violation for every return statement, other tools only give a violation on the last line of the function. To be able to compare true and false positives for this rule, all violations were put on the last line of the function. (see figure 1: number 7, 8 and 9)
Often there is an inverse relationship between Precision and Recall. While some tools are prepared to sacrifice finding all violations in favor of reducing the amount of false positives, some are not; as a result the former will in general produce relatively few false positives at the expense of some false negatives (violations which are not reported). In contrast, the latter will find most, if not all, violations of which a certain number are to be classified as false positive.

Having the Precision and Recall calculated, these calculations are then mapped to the representation of Figure 1 from which we, on the one hand, can compare the static code analysis capabilities between the various tools. On the other hand, this representation allows us to see which rules are best suitable for static code analysis by a given tool.

As can be seen for the figure above, PCLint has five rules (bottom left) for which the tool scores badly.
5. Conclusions:

1. The differences between tools are large
   1. GUI’s, the command line, install procedures, licencing systems, ... are very different between tools. A considerable effort is needed to start working with any of the programs in this study.
   2. There are also significant differences between tools concerning the rules to which violations are produced (see Appendix S to Appendix AB). Most tools give the possibility to tailor the generation of violation message to the specific needs of the user. This process is non-trivial and requires the attention of at least one employee.

2. The large number of rules in MISRA will (for any tool) result in an unmanageable number of violation messages.

3. Our expectation was to encounter a large number of false positives. Contrary to this expectation we found more false negatives, i.e. several tools fail to find obvious violations. This could be because none of the tools was created as a “MISRA-compliance” tool. These programs were designed to find portions of code which likely contain an error, and were latter adapted to find MISRA rule violations.

4. The more expensive tools, generally give better results (see Appendix L to S) i.e. more rules have precision and recall in the left upper corner.

5. Most tools do not give the option to obtain fully annotated code (source code + violation messages+line numbers) as an exported file. This limits the possibilities for the integration of the tool in existing software development processes.

6. An important problem encountered with all tools (some more than others) is the fact that the tools give no results if the code cannot be parsed completely. This is mainly problematic for large existing code bases.

6. Suggestions:

If a company wants to introduce MISRA-rule checking or more generally static code checking, we propose the following suggestions:

1. At the start: Keep it Simple
   1. a tool like PC-Lint contains a lot of functionality for a low price and is relatively simple in use.
   2. Limit the number of rules and choose rules which are relevant for the application domain of the company

2. Later on the number of rules can be expanded.

7. Future work:

A large amount of data has been generated applying the tools to the FreeRTOS real-time operating systems. However, there has been insufficient time to process this data. More detailed information about the tools could be obtained by studying this data.
References:


Contents of the appendices:
Appendices A through K contain a bar graph for ever rule of the list of 11 selected rules. On the X-axis, we find all the rules for which a violation was generated by any tool on all the code. The height of the bar indicates number of reported violations for every tool (see color code) over the code for the rule under consideration.
Appendices L through S a Recall-Precision digram fro every tool.
Appendices T through AB contain a bar chart per tool which gives the true positives for every reported rule over all the test source code.
Appendix A

Figure 9: Appendix A

Rule 2.3

Legend:
- DAC
- IAR
- QAC
- C++Test
- PCLint
- rainCode
- LDRA
- Klocwork
Appendix B

Figuur 10: Appendix B
Rule 9.1
Appendix D

Rule 11.1

Figuur 12: Appendix D
Rule 12.4
Appendix F

Rule 14.7

Figuur 14: Appendix F
Rule 16.6

Figuur 17: Appendix I
Appendix J

Rule 17.6

Figuur 18: Appendix J
Appendix K

Figuur 19: Appendix K
Appendix L

Figuur 20: Appendix L
Appendix N

![Graph depicting performance metrics for C++ Test.]
Appendix O

Figuer 23: Appendix O
Figuur 25: Appendix Q
Appendix R

Figuur 26: Appendix R
Figuur 27: Appendix S
Appendix T

Figuur 28: Appendix T

dac

Figuur 28: Appendix T
Appendix W

Figuur 31: Appendix W
Appendix X

Figuur 32: Appendix X
Figuur 33: Appendix Y
Figuur 35: Appendix AA