Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the Heart of South Leinster

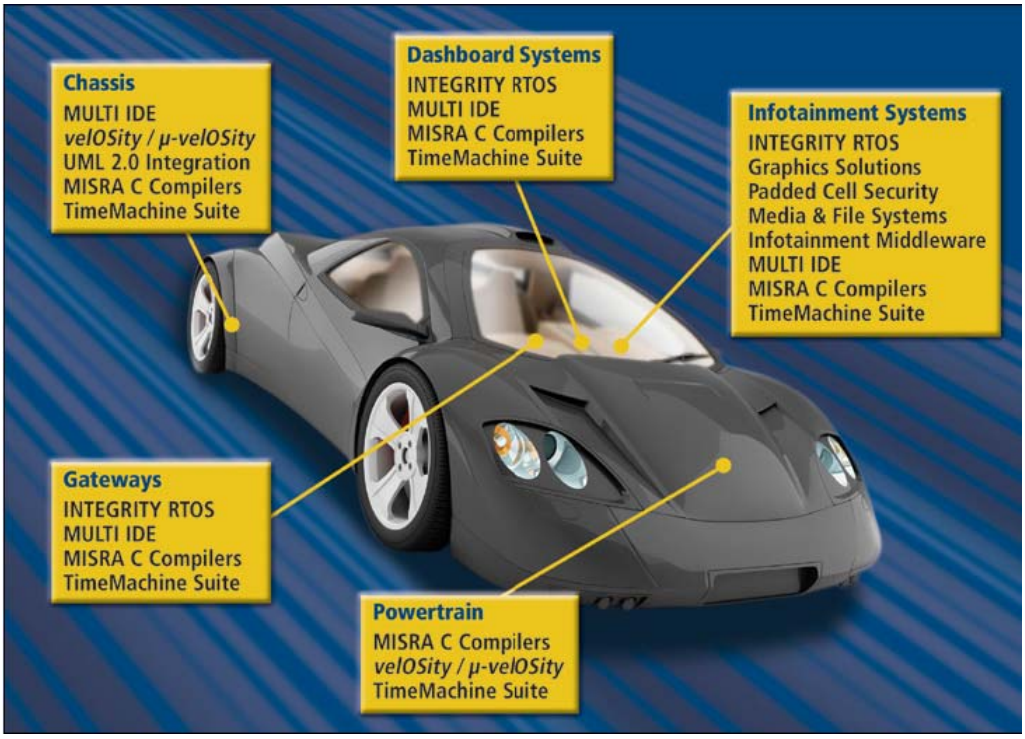
Institute of Technology, Carlow

B.Sc (Honour) in Software Engineering

Project Specification

For

MISRA C Code Compliance Checker Project



Student Name: **Mingjun Zhou**

Student ID: **C00094981**

Supervisor: **Dr. Christophe Meudec**

Date: **11/12/2009**

Table of Content

Vision

Revision History

Version	Date	Description	Author
Inception draft	11/12/2009	First draft. To be refined primarily during elaboration.	Mingjun Zhou

Introduction

I envision an MISRA C code compliance checker tool that will check if the C source code following the guideline which provided by MISRA C. This tool will help the program developers automatically checking their code and guide them to keep the good coding style. The

Positioning

Business opportunity

Most existing MISRA C code checkers are well developed, and functional. But some of them are seems a little complex before getting into it. I think that's not suits for beginner. My tool is going to be small, simple and concentrate on MISRA C-1998. If there are some program developers in beginner level and they want to build up programming style in good practice or people who is looking for a tool for checking C source code for small safety-relative project program, my tool will be their choice.

Problem Statement

For its own advantages, the C programming language is widely used in real-time embedded and safety-critical application field especially in motor vehicle. At the same time when the programming style of C gives programmer the full of flexibility, there is another problem become more obvious. The misunderstanding of code could cause serious problem in safety-related requirement, even could cost life.

Tool Position Statement

This tool I aim to make is targeted for the person who is responsible for making a C program for safety-relative industry. And also for those program developers who is going to study and check

their programming style in good way. The differentiates of this tool with other existing static code checking tool are:

- Independent
- Focus on MISRA C guideline
- Easy to implement
- Good and easy to read feedback information

Stakeholder Descriptions and Goals

Stakeholder summary

The main stakeholders involved in this system are the owners of the industry which must deal with safety-relative requirement for their product. For example, motor industry, unclear station, aircraft industry, etc. And the application development organizations will also be involved.

User summary

C language program developer

Key High-Level Goals and Problems of the Stakeholder

High-Level Goal	Priority	Problems and Concerns	Current Solutions
Fully check C source code, fast, precise, reliable, verify, valid	Very high	● Programmer makes mistakes ● Programmer misunderstands the language ● The compiler doesn't do what the programmer expects ● The compiler contains errors ● Run-time errors	● Existing static code analysis tools ● Compilers

User-Level Goals

The programmers need a tool can check their C source code and fast give them a feedback information which tell them the exactly part of code against the MISRA C guidelines.

User Environment

Command line prompt in Windows Operating System.

Tool overview

Summary of Benefits

Supporting Feature	Stakeholder Benefit
Technically, this tool will provide the main function to check C source code base on MIRSA C-1998 guideline, and gives properly feedback so that the program developer can make good program.	Get more reliable program for their own purpose.
Error capture and report	Easy to target the location of error Easy to understand the error

Assumptions and Dependencies

The code is syntactically correct and actually compiles

Cost and Pricing

Licensing and Installation

??

Summary of System Features

- Error detects
- Feedback function (give error or warning message)

Supplementary specification

Revision History

Version	Date	Description	Author
Inception draft	11/12/2009	First draft. To be refined primarily during elaboration.	Mingjun Zhou

Introduction

This document is the repository of all MISRA C code compliance checker requirements not captured in the use cases.

Functionality

Logging and Error Handling

Log all executing errors to persistent storage.

Usability

The feedback information should be clear and fully detailed in order to guide user to find the error and correct them.
Help information should be easy to understand and can call it straightforward.
Fast process speed.
Command line prompt tool

Reliability

Recoverability
If there is a failure to check source code
All the feedback information should be given correctly.

Performance

Fast processing speed is very important.
For example:

If the code is around 100 lines, processing speed would be a few seconds.
If it greater than 1000 lines, the processing could be around 1 minute.

Supportability

Adaptability: All the rules will be defined in different class or file. The new rules can be added by adding more class or rules files

Configurability: The user and specify which rule they are going to apply

Implementation Constraints

ANTLR
JAVA Language

Purchased Components

None

Free Open Source Components

ANTLRworks 1.3
Netbeans IDE 6.7.1

Interfaces

Noteworthy Hardware and Interfaces

- Command line prompt

Software Interfaces
Windows Operating System

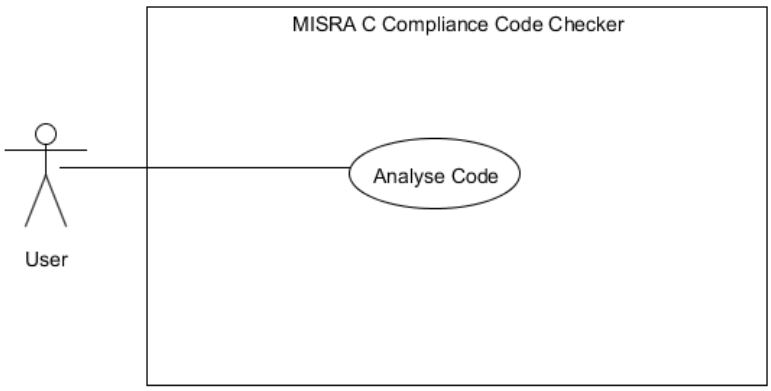
Legal Issues

Open source components is be licensed and free to user.

Use cases

Use case:	Analyse Code
Actor:	User
Action:	The code checker analyses the incoming course code base on the built in mechanism.

Use Case Diagram



Most existing MISRA C code checkers are well developed, and functional. But some of them are seems a little complex before getting into it. I think that's not suits for beginner. My tool is going to be small, simple and concentrate on MISRA C-1998. If there are some program developers in beginner level and they want to build up programming style in good practice or people who is looking for a tool for checking C source code for small safety-relative project program, my tool will be their choice.

Use Case UC1: Analyse Code
Scope: MISRA C Compliance Code Checker
Level：
Primary actor: User
Stakeholders and interests:

- User: wants to check if any parts of their written code have errors against MISRA C-1998 Guidelines, gets feedback information to find out the exactly location, and the code against which rule, etc.
- System: wants to locate C source code from user's input command. Get correct file and data stream, gives it to ANTLR for analysing.
- ANTLR: wants to analyse the source code and parse it.

Preconditions:

- User type in right command, and link with correct C source code.

- ANTLR has C Standard grammar file and has its construction by lexering and parsing.

Success Guarantee: Feedback information output correctly.

Main Success Scenario:

1. Input C source code system
2. The system analyse the source code stream
3. Compare its construction with MISRA C 1998 guidelines, user can have option to indicate which rules to apply
4. Output the feedback information on the screen and in a text file.

Extension:

1a. The C source code location is incorrect

   a. The system gives error message to inform user check path

1b. the input file is not a C source code.

   a. The system gives error message to inform user check file type

Output to be produced example

Example code:

Test.c

```
void func(void);
void func(void)
{
    int i=0;
    while(i<10) {
        i++;
        if (i == 5) {
            break;
        }
    }
}
```

Message Format:

Error: [FileName.c] [line number: ???] against MISRA (rule-number) {message}

Warning: [FileName.c] [line number: ???] against MISRA (rule-number) {message}

Output should look most likely as:

Error: [c:\test.c] [line number: 8] against MISRA (58) {'break' statement shall not be used (except in a 'switch')}

MISRA rules categories and examples

Categories

There are 17 categories in MISRA C, as following:

- Environment
- Character Sets
- Comments
- Identifiers
- Types
- Constants
- Declarations and Definitions
- Initialisation
- Operators
- Conversions
- Expressions
- Control Flow
- Functions
- Pre-processing Directives
- Pointers and arrays
- Structures and Unions
- Standard Libraries`

Example of Rules: [1]

Rule 33 (required): **The right hand operand of a `&&` or `||` operator shall not contain side effects.**

There are some situations in C code where certain parts of expressions may not be evaluated. If these sub-expressions contain side effects then those side effects may or may not occur, depending on the values of other sub expressions. The operators which can lead to this problem are `&&`, `||` and `?:`. In the case of the first two (logical operators) the evaluation of the right-hand operand is conditional on the value of the left hand operand. In the case of the `?:` operator, either the second or third operands are evaluated but not both. The conditional evaluation of the right hand operand of one of the logical operators can easily cause problems if the programmer relies on a side effect occurring. The `?:` operator is specifically provided to choose between two sub-expressions, and is therefore less likely to lead to mistakes.

For example:

```
if ( ishigh && ( x == i++ ) )   /* Incorrect */
```

```
if ( ishigh && ( x == f(x) ) )  /* Only acceptable if f(x) is known to
                                   have no side effects */
```

Rule 49 (advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean

Where a data value is to be tested against zero then the test should be made explicit. The exception to this rule is data which is representing a Boolean value, even though in C this will, in practice, be an integer. This rule is in the interests of clarity, and makes clear the distinction between integers and logical values.

For example, if x is an integer, then:

```
if ( x != 0 ) /* Correct way of testing x is non-zero */
if ( x )      /* Incorrect, unless x is effectively Boolean data
              (e.g. a flag) */
```

Rule 50 (required): Floating point variables shall not be tested for exact equality or inequality.

The inherent nature of floating point types is such that comparisons of equality will often not evaluate to true even when they are expected to. In addition the behaviour of such a comparison cannot be predicted before execution, and may well vary from one implementation to another. For example the result of the test in the following code is unpredictable:

```
F_32 x, y;    /* some calculations in here */
if ( x == y )
{ /* ... */ }
```

Evolutionary approach [1]

Version 1,

Since there still a lot to study, I attempt to implement some simple “required” rules first to make a try. For example:

Rule 9: Comments shall not be nested.

C does not support the nesting of comments. After a /* begins a comment, the comment continues until the first */ is encountered, with no regard for any nesting which has been attempted.

Rule 14: The type *char* shall always be declared as *unsigned char* or *signed char*.

The type *char* may be implemented as a signed or an unsigned type depending on the compiler. Rather than making any assumptions about the compiler, it is preferable

(and more portable) to always specify whether the required use of *char* is signed or unsigned.

Rule 119: The error indicator *errno* shall not be used.
errno is a facility of C which in theory should be useful, but which in practice is poorly defined by the standard. As a result it shall not be used. Even for those functions for which the behaviour of *errno* is well defined, it is preferable to check the values of inputs before calling the function rather than rely on using *errno* to trap errors

Rule120: The macro offsetof, in library <stddef.h>, shall not be used.
Use of this macro can lead to undefined behaviour when the types of the operands are incompatible or when bit fields are used.

Rule 121: <locale.h> and the *setlocale* function shall not be used.
This means that the locale shall not be changed from the standard C locale.

Rule 122: The *setjmp* macro and the *longjmp* function shall not be used.
setjmp and *longjmp* allow the normal function call mechanisms to be bypassed, and shall not be used.

Rule123: Rule 123 (required): The signal handling facilities of <signal.h> shall not be used.
Signal handling contains implementation-defined and undefined behaviour.

Rule 124: The input/output library <stdio.h> shall not be used in production code.
This includes file and I/O functions *fgetpos, fopen, ftell, gets, perror, remove, rename,* and *ungetc.* If any of the features of *stdio.h* need to be used in production code, then the issues associated with the feature need to be understood.

Rule 125: The library functions *atof, atoi* and *atol* from library <stdlib.h> shall not be used.
These functions have undefined behaviour associated with them when the string cannot be converted. They are unlikely to be required in an embedded system.

Rule 126: The library functions *abort, exit, getenv* and *system* from library <stdlib.h> shall not be used.
These functions will not normally be required in an embedded system, which does not normally need to communicate with an environment. If the functions are found necessary in an application, then it is essential to check on the implementation-defined behaviour of the function in the environment in question.

Rule 127: The time handling functions of library <time.h> shall not be used.

Includes *time, strftime*. This library is associated with clock times. Various aspects are implementation dependent or unspecified, such as the formats of times. If any of the facilities of *time.h* are used then the exact implementation for the compiler being used must be determined.

Version 2

Depends on how version 1 goes, I will add on more complicated rules on. For instance:

Rule 11: Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

The main purpose of this rule is to ensure that code can be ported between the majority of compilers/linkers without requiring modification (shortening) of parameter names.

Rule 17: *typedef* names shall not be reused.

Once a name has been assigned as a *typedef* it should not be used for any other purpose in any of the code files.

Rule 20: All object and function identifiers shall be declared before use.

Identifiers which represent objects or functions shall always have been declared before they are used, either by a declaration in the code file, or in an included header file.

Rule 21: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Hiding identifiers with an identifier of the same name in a nested scope leads to code which is very confusing. For example:

```
SI_16 i;
{
SI_16 i; /* This is a different variable */
         /* This is not permitted */
i = 3; /* It could be confusing as to which i this refers */
}
```

Reference

[1]Section 7: Rules, The Motor Industry Software Reliability Association, Guidelines For the Use Of The C Language In Vehicle Based Software, April 1998