ISO26262 For Software Developers

Daniel Liezrowice
Development tools Specialist

Daniel.l@eswlab.com

Introduction

Who we are?

- Parasoft – Source Code Compliance Tools manufacture
- Intland CodeBeamer – Specific ALM Tools for regulated industries
- ESL – The local representative of Parasoft Intland , responsible for integration and support in Israel
- Me (Daniel Liezrowice) Development Tools Specialist, in the last 12 years I was responsible for 170+ implementations of SW tools in Regulated industries (Automotive, Medical Device, Civil Aviation etc..) https://www.linkedin.com/in/liezrowice/

Introduction

What Is ISO 26262?

- ISO 26262 is a functional safety standard used in the automotive industry. A.K.A "Road vehicles — functional safety".
- Complying with ISO 26262 is rather critical for Automotive Products developers OEMs, their suppliers, and developers of automotive components all need to comply.
- In the next slides I will give an overview of ISO 26262 and the concept of ASIL (Automotive Safety Integrity level) …

The purpose of ISO 26262 is to ensure safety throughout the lifecycle of automotive equipment and systems.

Specific steps are required in each phase. This ensures safety from the earliest concept to the point when the vehicle/product is retired.

By complying with ISO 26262, you'll avoid and/or control systematic failures. And you'll detect and/or control random hardware failures.

Typical ASIL Examples

Typical Automotive Classifications

Rear lights
Both side failure,
ASIL-A

Instrument cluster
loss of critical data,
ASIL-B

Rear view camera
No valid sensor data,
ASIL-B

Airbag
Inadvertent deploy,
ASIL-D

Brake lights
Both side failure,
ASIL-B

Engine management
Unwanted acceleration,
ASIL-C to D

Antilock braking
Unintended full power braking,
ASIL-D

Headlights
Both side failure,
ASIL-B

Active suspension
Suspension oscillates,
ASIL-B to C

Radar cruise control
Inadvertent braking,
ASIL-C

Electric power steering
Self-steering,
ASIL-D

Vision ADAS
Incorrect sensor feedback,
ASIL-B

The 10 (general) Parts of ISO 26262

Part 1: Vocabulary.	**Part 2:** Management of functional safety.	**Part 3:** Concept phase.	**Part 4:** Product development at the system level.	**Part 5:** Product development at the hardware level.
Part 6: Product development at the software level.	**Part 7:** Production and operation.	**Part 8:** Supporting processes.	**Part 9:** ASIL-oriented and safety-oriented analysis.	**Part 10:** Guideline on ISO 26262.

1. Vocabulary

2. Management of functional safety

- **2-5** Overall safety management
- **2-6** Project dependent safety management
- **2-7** Safety management regarding production, operation, service and decommissioning

3. Concept phase

- **3-5** Item definition
- **3-6** Hazard analysis and risk assessment
- **3-7** Functional safety concept

4. Product development at the system level

- **4-5** General topics for the product development at the system level
- **4-6** Technical safety concept
- **4-7** System and item integration and testing
- **4-8** Safety validation

7. Production, operation, service and decommissioning

- **7-5** Planning for production, operation, service and decommissioning
- **7-6** Production
- **7-7** Operation, service and decommissioning

12. Adaptation of ISO 26262 for motorcycles

- **12-5** General topics for adaptation for motorcycles
- **12-6** Safety culture
- **12-7** Confirmation measures
- **12-8** Hazard analysis and risk assessment
- **12-9** Vehicle integration and testing
- **12-10** Safety validation

5. Product development at the hardware level

- **5-5** General topics for the product development at the hardware level
- **5-6** Specification of hardware safety requirements
- **5-7** Hardware design
- **5-8** Evaluation of the hardware architectural metrics
- **5-9** Evaluation of safety goal violations due to random hardware failures
- **5-10** Hardware integration and verification

6. Product development at the software level

- **6-5** General topics for the product development at the software level
- **6-6** Specification of software safety requirements
- **6-7** Software archtectural design
- **6-8** Software unit design and implementation
- **6-9** Software unit verification
- **6-10** Software integration and verification
- **6-11** Testing of the embedded software

8. Supporting processes

- **8-5** Interfaces within distributed developments
- **8-6** Specification and management of safety requirements
- **8-7** Configuration management
- **8-8** Change management
- **8-9** Verification
- **8-10** Documentation management
- **8-11** Confidence in the use of software tools
- **8-12** Qualification of software components
- **8-13** Evaluation of hardware elements
- **8-14** Proven in use argument
- **8-15** Interfacing an application that is out of scope of ISO 26262
- **8-16** Integration of safety-related systems not developed according to ISO 26262

9. Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses

- **9-5** Requirements decomposition with respect to ASIL tailoring
- **9-6** Criteria for coexistence of elements
- **9-7** Analysis of dependent failures
- **9-8** Safety analyses

10. Guidelines on ISO 26262

11. Guidelines on application of ISO 26262 to semiconductors

Figure 1 — Overview of the ISO 26262 series of standards

ISO 26262 For Software Developers

- ISO 26262 Part 6 is the most important part for software developers. It details the steps Software developers must take to ensure the safety of each component.
- Part 6 includes several tables that define the methods that must be considered in order to achieve compliance with the standard.
- (going through all these tables is out of the scope of the presentation)

ISO 26262 For Software Developers

- In that presentation we will:
- Review and demonstrate (as much as it will be possible) the processes and the tools needed for complying with Part 6 of ISO26262

DL-E1

ISO 26262 For Software Developers

- Going through these tables is out of the scope of the presentation but here is a typical table

Table 1 — Topics to be covered by modelling and coding guidelines

	Topics	ASIL A	ASIL B	ASIL C	ASIL D
1a	Enforcement of low complexity[a]	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques[d]	+	+	++	++
1e	Use of well-trusted design principles[e]	+	+	++	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++
1i	Concurrency aspects[f]	+	+	+	+

[a] An appropriate compromise of this topic with other requirements of this document may be required.

[b] The objectives of topic 1b include:

— Exclusion of ambiguously-defined language constructs which may be interpreted differently by different modellers, programmers, code generators or compilers.

— Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.

— Exclusion of language constructs which could result in unhandled run-time errors.

[c] The objective of topic 1c is to impose principles of strong typing where these are not inherent in the language.

[d] Examples of defensive implementation techniques:

— Verify the divisor before a division operation (different from zero or in a specific range).

— Check an identifier passed by parameter to verify that the calling function is the intended caller.

— Use the "default" in switch cases to detect an error.

[e] Verification of the validity of the underlying assumptions, boundaries and conditions of application may be required.

[f] Concurrency of processes or tasks is not limited to executing software in a multi-core or multi-processor runtime environment.

Slide 10

DL-E1 Daniel Liezrowice - ESL, 10/07/2019

workspace - C/C++ - Parasoft C/C++test

File Edit Source Refactor Navigate Search Project Parasoft Run

Project Explorer

- asm_test
- security testing example
- sensor
- testcase

sensor.c gm.h

```
    printf("Val
    fflush(stdo
}

void reportSens
{
    initialize(

    // FIX: ch
  // zero();
    printMessag

    finalize();
}

void handleSens
{
    int index =
    initialize(
```

Problems Tasks

15 test results, 0 code r

[15] DanielLiezrow

Test Configurations

Create, manage, and run test configurations

Filter

- User-defined
  - Application Monitoring
  - Compliance Packs
    - Aerospace Pack
    - Automotive Pack
      - KW to Parasoft - Marina Sapir
      - My_MISRA C 2012
      - My_MISRA C 2012 (1)
      - My_MISRA C++ 2008
    - Medical Devices Pack
    - Security Pack
  - Embedded Systems
  - Static Analysis
  - Unit Testing
- Builtin
  - Application Monitoring
  - Code Review
  - Compliance Packs
  - Embedded Systems
  - Static Analysis
  - Unit Testing
  - Utilities
- Team
- DTP

New Delete

Name My_MISRA C 2012

Parent

Scope | Static | Generation | Execution | Code Review | Common | Goals

- [x] Enable Static Analysis
- [x] Limit maximum number of tasks reported per rule to: 1000
- [ ] Do not apply suppressions

Code parsing problems: Ignore

Rules Tree | Flow Analysis Advanced Settings

Filter namin

Number of rules: 3,575 total; 420 enabled; 0 hidden

- [x] Naming Conventions [NAMING] - (94/94 enabled)
  - [x] Hungarian Notation [NAMING-HN] - (44/44 enabled)
  - [x] Identifiers for constant and enumerator values shall be lowercase [NAMING-42 - 2]
  - [x] All "#define" constants shall be in uppercase [NAMING-01 - 3]
  - [x] In an enumerated list, list members (elements) shall be in uppercase and names or tags for the li
  - [x] Use lowercase for file names [NAMING-03 - 3]
  - [x] Global prefixes should only be used for global variables [NAMING-04 - 3]
  - [x] Begin local variable names with a lowercase letters [NAMING-05 - 3]
  - [x] Begin global variable names with a lowercase letters [NAMING-06 - 3]
  - [x] Begin member variable names with a lowercase letters [NAMING-07 - 3]
  - [x] Begin all boolean type variables with 'b' [NAMING-08 - 3]
  - [x] Begin class, struct, union, enum, and typedef names with an uppercase letter [NAMING-09 - 3]
  - [x] The names of abstract data types, structures, typedefs, and enumerated types are to begin with
  - [x] The name of enumeration type shall begin with an uppercase letter and contain a suffix '_t' at th

Hid | Edi | Prin

Revert

Run Test

ensor.c - Parasoft C/C++test

igate Search Project Parasoft Run Window Help

sensor.c | gm.h | gmvt_main.cpp | gmvt.cpp | Web Browser | gmvt_api.cpp | TestSuite_sensor_c.c | Web Browser

```
#include <string.h>
#include "C:\Users\DanielLiezrowice-ESL\parasoft\workspace\sensor\gm.h"

const int STATUS_OK = 0;
const int STATUS_FAILED = 1;
const int STATUS_STOPPED = 2;
int i;

const int MAX_NUMBER_OF_SAMPLES = 30;

int readSensor(int * value)
{
    static int v = 0;
```

Problems | Tasks | Console | Properties | Quality Tasks | Naming convenations for ISO 26262 ... | Suppressions | Coverage | Stubs

135 test results, 0 code review

- [135] >DanielLiezrowice-ESL
  - [7] >Severity 2 - High
    - [7] >Identifiers for constant and enumerator values shall be lowercase (NAMING-42-2)
      - [7] >C:\Users\DanielLiezrowice-ESL\parasoft\workspace\sensor\sensor.c
        - >[Line 6] Variable's name 'STATUS_OK' should be lowercase
        - >[Line 7] Variable's name 'STATUS_FAILED' should be lowercase
        - >[Line 8] Variable's name 'STATUS_STOPPED' should be lowercase
        - >[Line 11] Variable's name 'MAX_NUMBER_OF_SAMPLES' should be lowercase
        - >[Line 23] Variable's name 'VALUE_LOW' should be lowercase
        - >[Line 24] Variable's name 'VALUE_HIGH' should be lowercase
        - >[Line 25] Variable's name 'ERROR' should be lowercase
  - [128] >Severity 3 - Medium

And the tables are becoming more specific and complex as we go further down the document…..

Table 6 — Design principles for software unit design and implementation

Principle		ASIL			
		A	B	C	D
1a	One entry and one exit point in subprograms and functions[a]	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation[a]	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names[a]	++	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Restricted use of pointers[a]	+	++	++	++
1g	No implicit type conversions[a]	+	++	++	++
1h	No hidden data flow or control flow	+	++	++	++
1i	No unconditional jumps[a]	++	++	++	++
1j	No recursions	+	+	++	++

[a] Principles 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

NOTE For the C language, MISRA C (see Reference [3]) covers many of the principles listed in Table 6.

Pattern Matching issue

That Means, We can't write like that

EXAMPLE

```
int foo(int i)
{
   if (i == 0) {
       return 0;          // Violation
   } else if (i == 1) {
       return 1;          // Violation
   } else {
       return 2;          // Violation
   }
}

int foo2(int a) {
   int result;
   if (a > 0) {
       return result;  // Violation
   }
}
```

Pattern Matching issue

It is a violation of all these coding standards…

REFERENCES

1. Origin: Misra Guidelines - Rule 82

2. MISRA-C:2004 Guidelines for the use of the C language in critical systems
   Chapter 6, Section 14

3. MISRA C:2012 Guidelines for the use of the C language in critical systems
   Section 8: Rules, Rule 15.5

4. MISRA C++:2008 Guidelines for the use of the C++ language in critical systems
   Section 6.6.6 Jump statements, Rule 6-6-5

5. JOINT STRIKE FIGHTER, AIR VEHICLE, C++ CODING STANDARDS
   Chapter 4.13 Function, AV Rule 113

6. ISO/DIS 26262
   point 8.4.4

Pattern Matching issue

And we need a tool that will not “leave us a lone” until the code will look like that:

REPAIR

```
int foo(int i)
{
 int result = 0;
   if (i == 0) {
       result = 0;
   } else if (i == 1) {
    result = 1;
   } else {
    result = 2;
   }
   return result;   // OK
}
```

ISO 26262 Tool Qualification

- Any tools used in automotive development need to be qualified. Part 8 of ISO 26262 provides guidance for tool qualification.
- It requires the following:
- Software tool qualification plan.
- Software tool documentation.
- Software tool classification analysis.
- Software tool qualification report.
- Using a certified tool makes life much easier and saves time
- Going through the phases is out of scope for now but the end result looks like that….

ZERTIFIKAT ◆ CERTIFICATE ◆ 認証書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

TÜV SÜD

Product Service

CERTIFICATE

No. Z10 075084 0005 Rev. 00

Holder of Certificate:	**Parasoft Polska Sp. z o. o.** Kielkowskiego 9 30-704 Krakow POLAND
Factory(ies):	075084
Certification Mark:	TÜV SÜD
Product:	**Software Tool for Safety Related Development**
Model(s):	**C/C++test**
Parameters:	Tool classification: T2 (acc. to IEC 61508) The verification tool fulfills the requirements for support tools according to IEC 61508-3. The tool is qualified to be used in safety-related software development according to IEC 61508 and ISO 26262. The test report is a mandatory part of this certificate.
Tested according to:	IEC 61508-3:2010 ISO 26262-8:2011

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. See also notes overleaf.

Test report no.: PK83996C

Valid until: 2023-08-30

Date, 2018-08-31

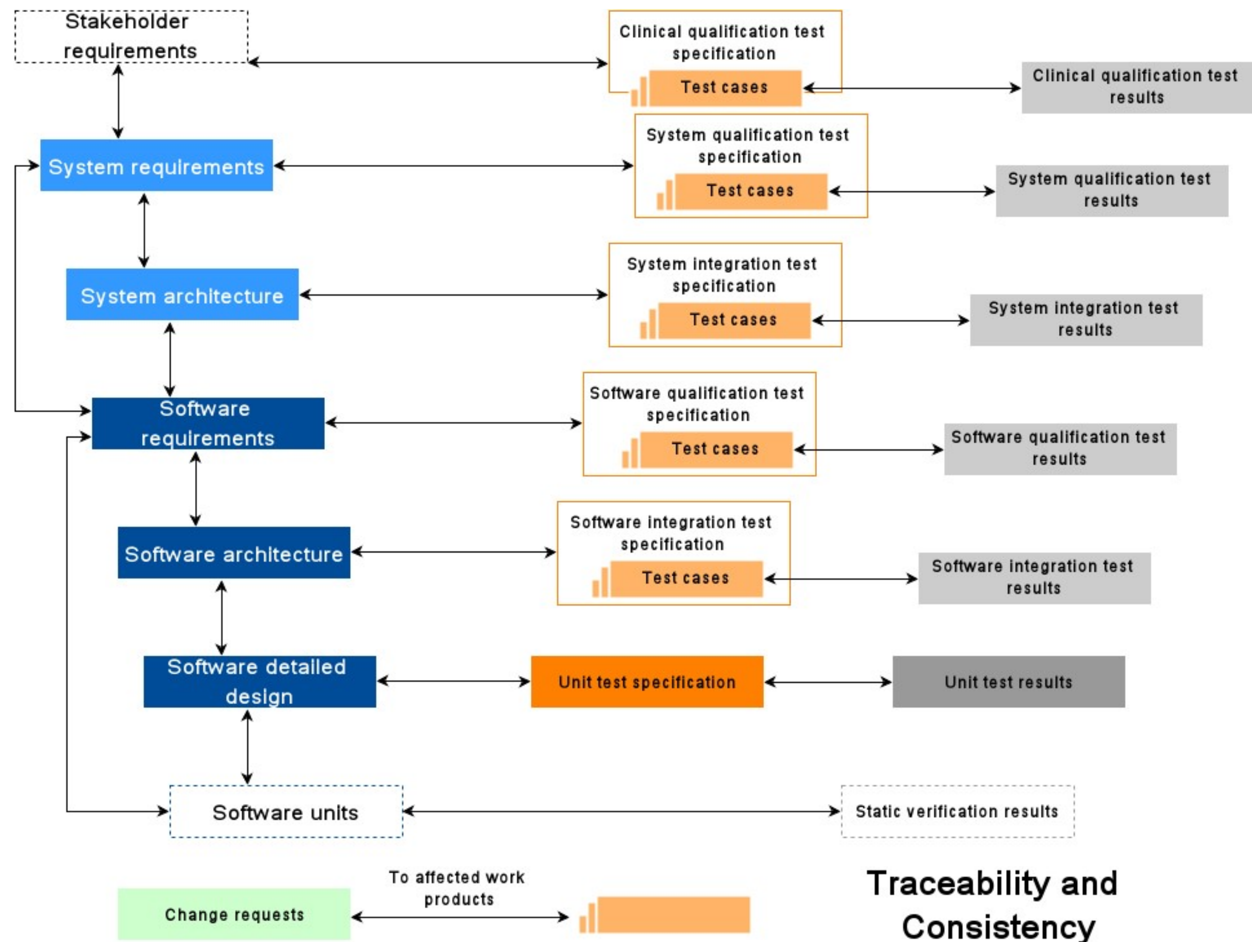
(Christian Dirmeier)

TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

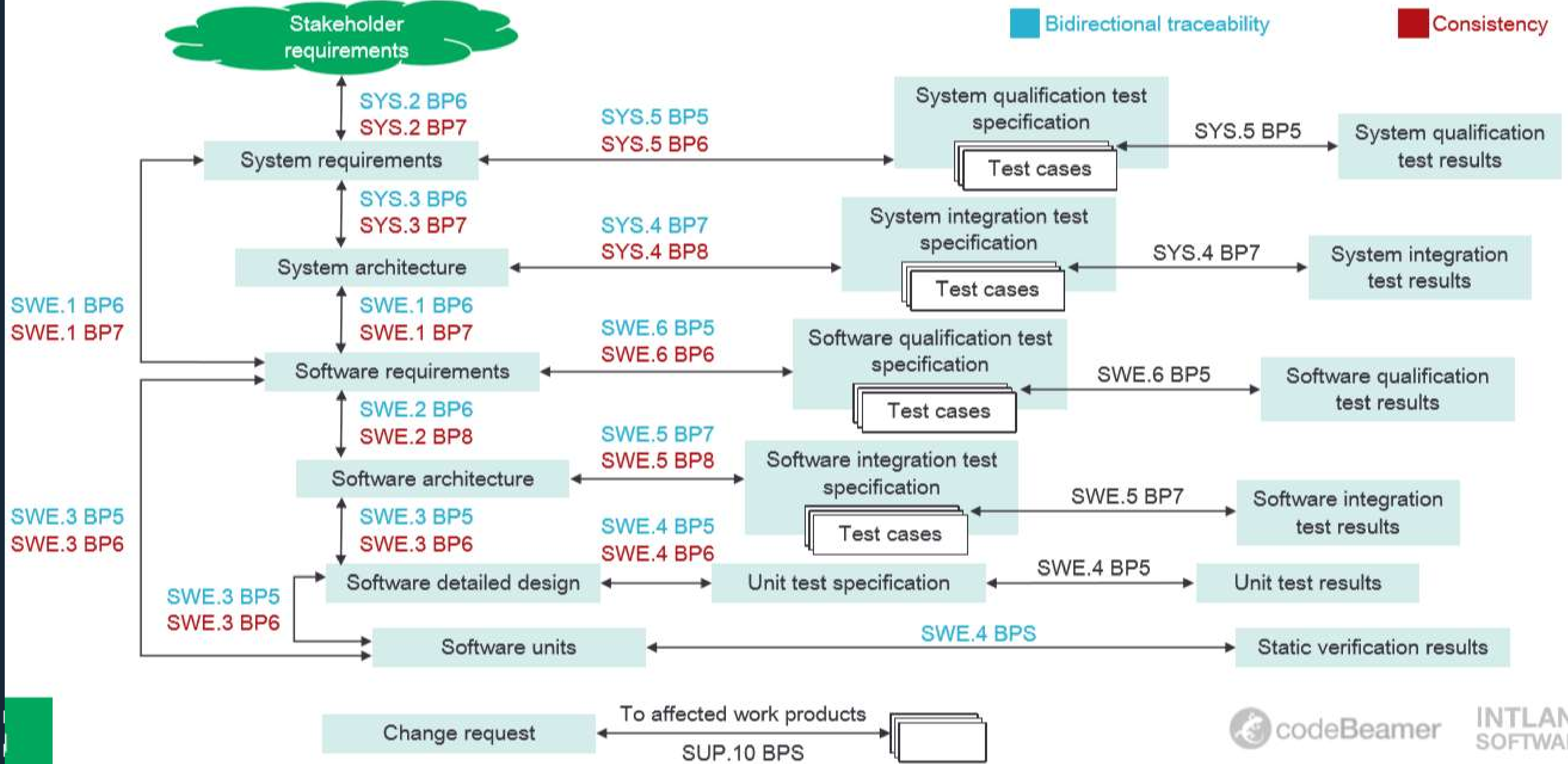
TÜV®

So What Tools the Software team needs?

- 1. Requirements Tractability , this is achieved by an ISO26262 compliant and certified ALM (Application Life Cycle Management) tool that has a built in ISO2626 Templates



ASPICE and ISO26262 compliance



So What Tools the Software team needs?

- 1. Coding Standards Tool that fully supports the Relevant Coding standard for the Automotive Industry
- Since Safety Critical applications are written in C/C++ the relevant standards are **MISRA C 2012** and **Autosar C++ 14**

workspace - C/C++ - sensor/sensor.c - Parasoft C/C++test

```
}

void printMessage(int msgIndex, int value)
{
    const char* msg = messages[msgIndex];
    printf("Value: %d, State: %s\n", value, msg);
    fflush(stdout);
}

void reportSensorFailure()
{
    finalize(); // FIX: change order + add initialize()
    printMessage(ERROR, 0);
}

void handleSensorValue1(int value)
```

108 test results, 0 code review

- >[Line 133] An 'else' statement in a function 'main' shall be followed by a block
- [4] >A conversion should not be performed from pointer to void into pointer to object (MISRAC2012-RULE_11_5-a-4)
- [13] >A declaration shall be visible when an object or function with external linkage is defined (MISRAC2012-RULE_8_4-a-2)
- [4] >All 'if...else-if' constructs shall be terminated with an 'else' clause (MISRAC2012-RULE_15_7-a-2)
- [4] >Always check the returned value of non-void function (MISRAC2012-DIR_4_7-b-2)
- [4] >Avoid null pointer dereferencing (MISRAC2012-DIR_4_1-b-2)
- [1] >Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category (MISRAC2012-RULE_10_4-a-2)
- [1] >Do not use resources that have been freed (MISRAC2012-DIR_4_13-b-4)
  - [1] >sensor.c
    - >[Line 51] Usage of freed resource "messages"
      - >sensor.c (58): finalize(); // FIX: change order + add initialize()
      - >sensor.c (59): printMessage(ERROR, 0);
        - >sensor.c (51): const char* msg = messages[msgIndex];
- [1] >Do not use resources that have been freed (MISRAC2012-RULE_1_3-c-2)

The only coding standard that is mentioned specifically in the ISO 26262 Standard Is MISRA C

5.4.3 Criteria for suitable modelling, design or programming languages (see 5.4.2) that are not sufficiently addressed by the language itself shall be covered by the corresponding guidelines, or by the development environment, considering the topics listed in Table 1.

EXAMPLE 1 MISRA C (see Reference [3]) is a coding guideline for the programming language C and includes guidance for automatically generated code.

EXAMPLE 2 In the case of model based development with automatic code generation, guidelines can be applied at the model level as well as the code level. Appropriate modelling style guides including the MISRA AC series can be considered. Style guides for commercial tools are also possible guidelines.

NOTE Existing coding guidelines and modelling guidelines can be modified for a specific item development.

So What Tools the Software team needs?

- The coding Standard tool must have a reporting interface that shows you what is your compliance level across the teams/ developers modules etc..

So What Tools the Software team needs?

- Since it is extremally hard and no common to have a 100% compliance to MISRA Coding Standard, the coding Standard tool must have a reporting interface that allows an easy deviation management and control

PARASOFT

MISRA Deviation Report

Filter: bc **Compliance Profile:** MISRA C 2012 **Compiler:** gcc 4.9 **Analysis Tool:** Parasoft C++test 10.3.2 **Target Build:** bc-2017-06-28

Dir 4.1 (Required) Run-time failures shall be minimized ✔ - No Deviations

Dir 4.2 (Advisory) All usage of assembly language should be documented ✔ - No Deviations

Dir 4.3 (Required) Assembly language shall be encapsulated and isolated ✔ - No Deviations

Dir 4.4 (Advisory) Sections of code should not be commented out"" ! - 1 Deviations

Rule ID: MISRA2012-DIR_4_4-a
Deviation Type: DTP Suppression
Action: Suppress
Risk/Impact: Undefined
Reference #: PRA-123

Modification History

User: demo
Date: 2017-08-17 08:20:30 PM
Field: referenceNumber
Old Value: N/A
New Value: PRA-123
Field: violationAction
Old Value: None
New Value: Suppress
Comment: As per Deviation Permit #PRA-123

User: demo
Date: 2017-08-17 08:20:37 PM
Field: priority
Old Value: Not Defined
New Value: Do not show

Dir 4.5 (Advisory) Identifiers in the same namespace with overlapping visibility should be typographically unambiguous ✔ - No Deviations

Dir 4.6 (Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types ! - 112 Deviations

Rule ID: MISRA2012-DIR_4_6-b
Deviation Type: In-Code Suppression
Action: None
Risk/Impact: Undefined
Suppression Reason: suppress rule MISRA2012-DIR_4_6-b
Suppression Author: igarg

Powered by Parasoft Development Testing Platform. Copyright © 1996-2017.

Advantages of the C Language

- C compilers exist for almost any processor
- C compiled code is very efficient and without hidden costs
- C allows writing compact code (many built-in operators, limited verbosity, ...)
- C is defined by an ISO standard
- C, possibly with extensions, allows easy access to the hardware
- C has a long history of usage in critical systems
- C is widely supported by tools

I Trust the programmer

II Let the programmer do anything

III Keep it fast, even if not portable

IV

V

Which C actually we talking about?

The C Language

- Which C?
- K&R C (1978)
- ANSI-C (K&R2 1989) 228 Pages
- ISO-C 9899:1990 (C90) 271 pages
- ISO-C 9899:1990 +A1 & TC's 1-3 (C95)
- ISO-C 9899:1999 (C99) 568 Pages
- ISO-C 9899:2011 (C11) 853 Pages

C can have Unspecified behavior

- Simple Example of Unspecified behavior in C

```
a=f(b)/g(b);
```

where f and g both modify b, the result stored in a may be different depending on whether f(b) or g(b) is evaluated first.

C can have Undefined Behavior

```
int foo(unsigned char x)
{
     int value = 2147483600; /* assuming 32 bit int */
     value += x;
     if (value < 2147483600)
        bar();
     return value;
}
```

The value of `x` cannot be negative and, given that signed integer overflow is undefined behavior in C, the compiler can assume that `value < 2147483600` will always be false. Thus the `if` statement, including the call to the function `bar`, can be ignored by the compiler since the test expression in the `if` has no side effects and its condition will never be satisfied. The code is therefore semantically equivalent to:

```
int foo(unsigned char x)
{
     int value = 2147483600;
     value += x;
     return value;
}
```

C has *Implementation-defined behavior*

- Trying to allocate 0 bytes of memory using

```
int *o = malloc(0 * sizeof *o);
```

My result in o either being NULL or Unique pointer

It is even written in the C99 standard

7.20.3 Memory management functions

The order and contiguity of storage allocated by successive calls to the **`calloc`**, **`malloc`**, and **`realloc`** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

If C wasn't broken, we wouldn't have a need to fix it

Rigidly defined areas of uncertainty and doubt

- Implementation defined 3.4.1
- Undefined 3.4.3
- Unspecified 3.4.4

ISO C 99 has over 400 of the above.

3.4.2?

Locale-Specific behaviour.

Conclusion

We described:

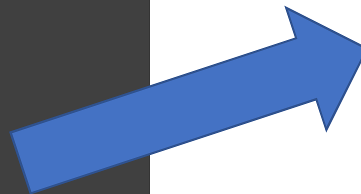
- Undefined behavior
- Unspecified behavior
- Implementation-defined behavior
- (and we glossed over locale-specific behavior)

Why is the standardized language not fully defined?

- Because implementing compilers is easier
- Because compilers can generate faster code

Enter Coding Standards!!!

- If we had a coding standard that dictate: "Don't write code that depends on the order of evaluation of expression that involves a function call "
- Then that wouldn't be a problem



C can have Unspecified behavior

- Simple Example of Unspecified behavior in C

```
a = f(b) + g(b);
```

where f and g both modify b, the result stored in a may be different depending on whether f(b) or g(b) is evaluated first.

And we have such a standard....

secure | caxapa.ru/thumbs/468328/misra-c-2004.pdf

tps://redlist.parks.... MISRA C Training in... Helly Bikereyes Hell... Dani renewal ~ Sale... Altoro Mutual Partner Dashbo

promotions. The following example written for a 16-bit implementation demonstrates that addition is not associative and that it is important to be clear about the structure of an expression:

```
uint16_t a = 10U;
uint16_t b = 65535U;
uint32_t c = 0U;
uint32_t d;

d = (a + b) + c;   /* d is 9; a + b wraps modulo 65536 */
d = a + (b + c);   /* d is 65545                        */
/* this example also deviates from several other rules */
```

Note that Rule 12.5 is a special case of this rule applicable solely to the logical operators, `&&` and `||`.

Rule 12.2 (required): The value of an expression shall be the same under any order of evaluation that the standard permits.

[Unspecified 7–9; Undefined 18]

Apart from a few operators (notably the function call operator `()`, `&&`, `||`, `?:` and `,` (comma)) the order in which sub-expressions are evaluated is unspecified and can vary. This means that no reliance can be placed on the order of evaluation of sub-expressions, and in particular no reliance can be placed on the order in which side effects occur. Those points in the evaluation of an expression at which all previous side-effects can be guaranteed to have taken place are called "sequence points". Sequence points and side effects are described in sections 5.1.2.3, 6.3 and 6.6 of ISO/IEC 9899:1990 [2].

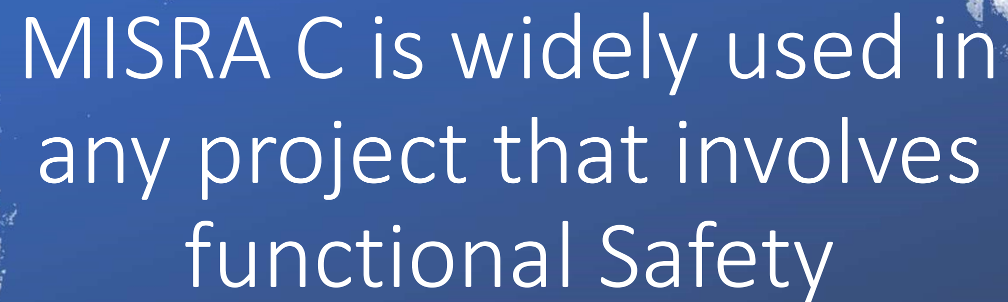
Note that the order of evaluation problem is not solved by the use of parentheses as this is not a precedence issue.

The following notes give some guidance on how dependence on order of evaluation may occur, and therefore may assist in adopting the rule.

- ***increment or decrement operators***

MISRA C (2012)

Standard defined and owned by The Motor Industry Software Reliability Association
Full details at: http://misra.org.uk/



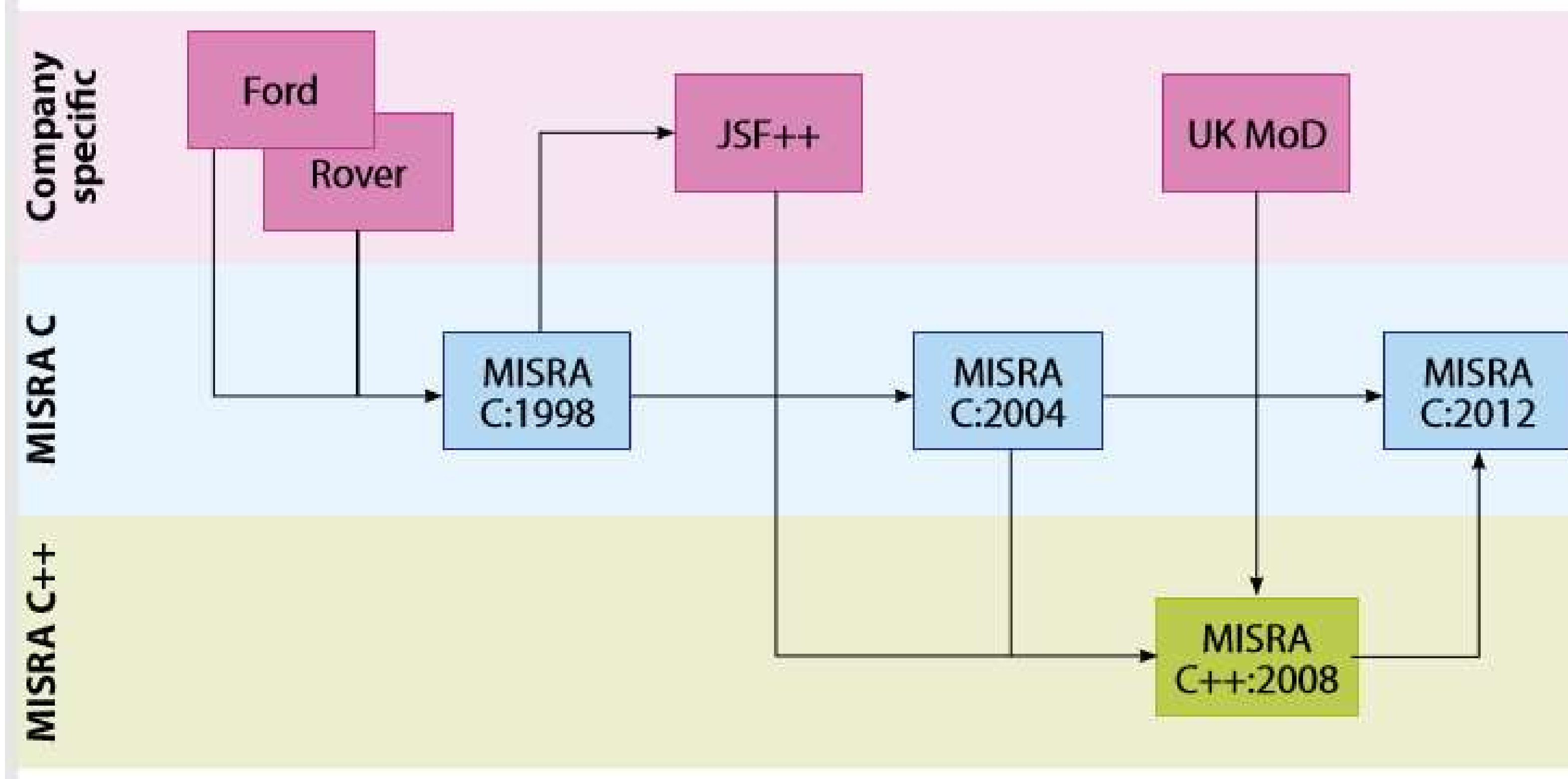
MISRA C is widely used in any project that involves functional Safety

DO178C

ISO 26262

IEC 61508

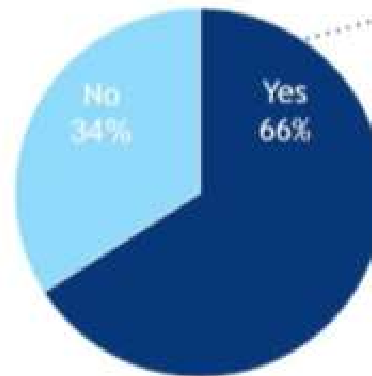
And more..

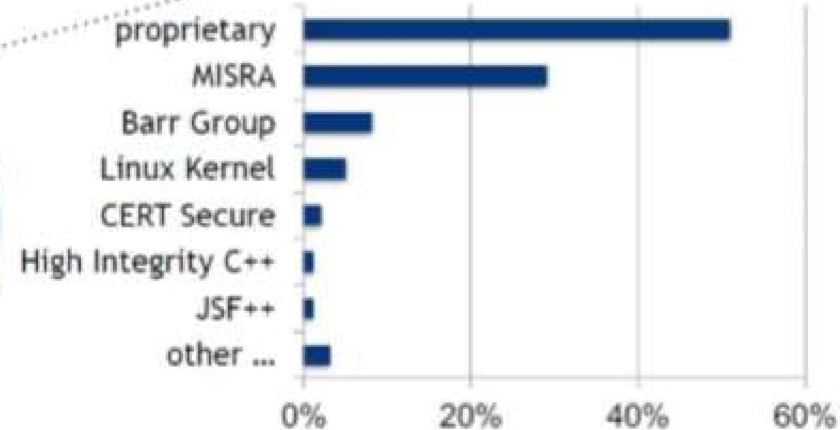

History of MISRA C/C++ Guidelines
Company specific
MISRA C
MISRA C++
Ford
Rover
JSF++
UK MoD
MISRA C:1998
MISRA C:2004
MISRA C:2012
MISRA C++:2008

Which Coding Standard to use?

CODING STANDARDS

Responses: 1,726





21 Copyright © 2017 by Barr Group. All rights reserved.

BARR group

How is the MISRA C Standard constructed?

- Directives and Rules
  - Rules are completely described without a context and checking tools should be able to verify them; may be *decidable* or *undecidable*
  - Directives need context or other information
- Categories
  - Mandatory
    - Non-optional
  - Required
    - Formal deviation required
    - A Required guideline may be treated as Mandatory
  - Advisory
    - Recommendations
    - An Advisory guideline may be treated as Required or Mandatory

Directives

- Modest number – 16
- Many are quite obvious
- Tend to address “big picture” issues

- Examples …

Directives

Dir 2.1 – All source files shall compile without any compilation errors

- Some compilers may produce executable code despite error messages
- This code's behavior is unpredictable

Dir 4.4 – Sections of code should not be "commented out"

- Dangerous because of comment nesting issues
- Readability is impaired
- Better to use preprocessor directives: `#if` or `#ifdef`

Dir 4.12 – Dynamic memory allocation shall not be used

- Generally bad to use `malloc()` etc. in a real time system
- Issues with insufficient memory and fragmentation
- Very non-deterministic

Rules

- Many
  - 143 in total
  - 22 categories
- Examples ...

Rules

Rule 1.2 – Language extensions should not be used

- Reduces code portability
- Often needed for embedded
  - e.g. `interrupt` keyword reduces assembler code

Rule 2.1 – A project shall not contain unreachable code

- Indicates a logical fault in the code
- Just allowing compiler to remove is dangerous
- May need to take care that compiler does not remove defensive code:
  - e.g. `switch (*(volatile uint32_t *) &svar)`

Rules

Rule 11.4 – A conversion should not be performed between a pointer to object and an integer type

- This rule is only advisory
- Very likely to be necessary in embedded applications to address device registers:

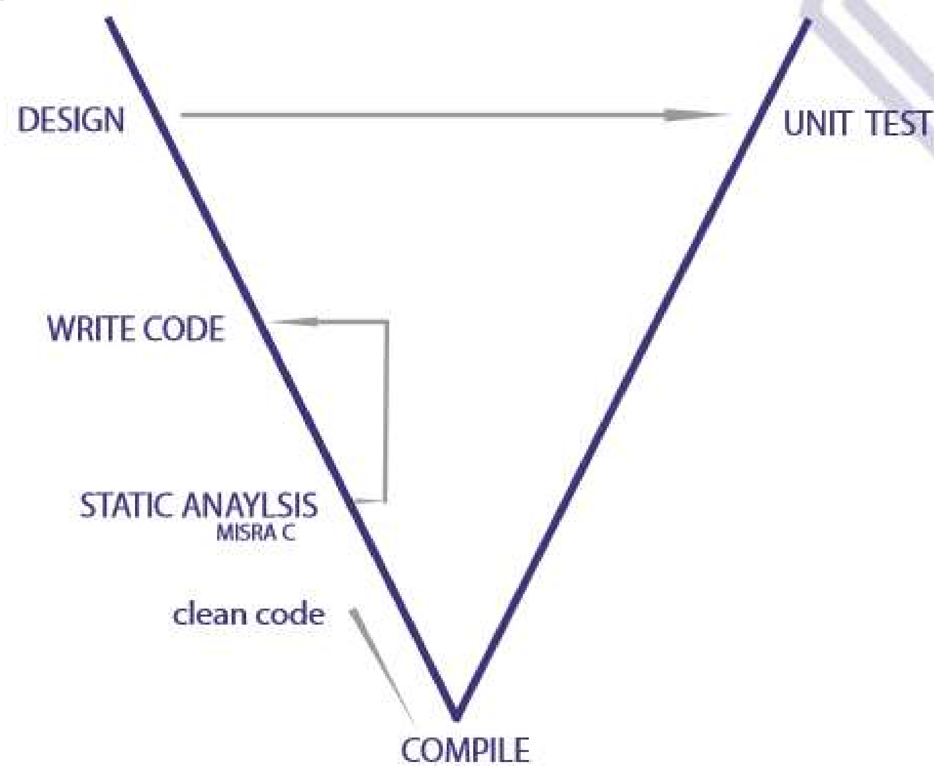
```
uint32_t *device_reg = (uint32_t *) 0x8000;
```

Rule 12.3 – The comma operator should not be used

- Generally leads to reduced readability/clarity
- Might be acceptable in a for-loop:

```
for (i=0, x=0.0; i<10; i++, x += 1.0)
        ...
```

Implementing MISRA C:2012



Implementing MISRA C:2012

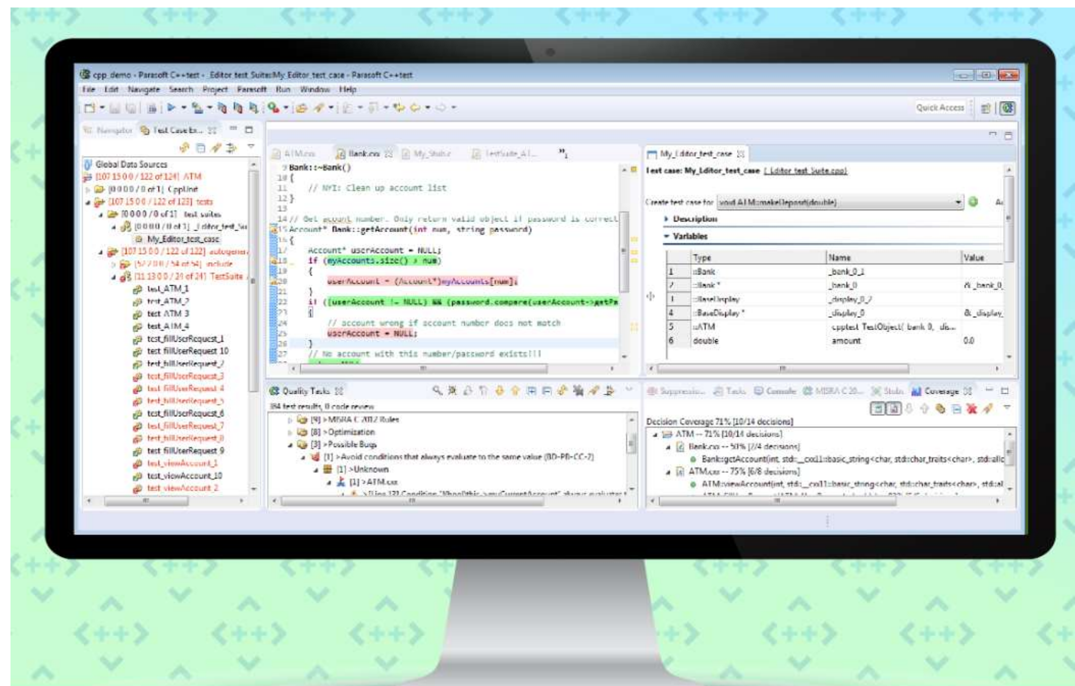
- Static Analysis
  - Essential for ALL C code
- Compiler is a TRANSLATOR
  - It is not a static analyser
  - It will compile legal code
    - no matter how dangerous

Implementing MISRA C:2012

- First the compiler, according to MISRA C it must be at least ISO 99 (or 95) compatible
- The First “unofficial” rule of MISRA C is “rule No 1 of MISRA C must be always <u>deviated</u>”
- (simply, no such thing as fully ISO C compliant Compiler)

Implementing MISRA C:2012

Secondly the Static Code Analyzer



Deviations

PARASOFT

demo

MISRA Deviation Report

Filter: bc **Compliance Profile:** MISRA C 2012 **Compiler:** gcc 4.9 **Analysis Tool:** Parasoft C++test 10.3.2 **Target Build:** bc-2017-06-28

Dir 4.1 (Required) Run-time failures shall be minimized ✔ - No Deviations

Dir 4.2 (Advisory) All usage of assembly language should be documented ✔ - No Deviations

Dir 4.3 (Required) Assembly language shall be encapsulated and isolated ✔ - No Deviations

Dir 4.4 (Advisory) Sections of code should not be commented out"" ! - 1 Deviations

Rule ID: MISRA2012-DIR_4_4-a
Deviation Type: DTP Suppression
Action: Suppress
Risk/Impact: Undefined
Reference #: PRA-123

Modification History

User: demo
Date: 2017-08-17 08:20:30 PM

Field: referenceNumber
Old Value: N/A
New Value: PRA-123

Field: violationAction
Old Value: None
New Value: Suppress

Comment: As per Deviation Permit #PRA-123

User: demo
Date: 2017-08-17 08:20:37 PM

Field: priority
Old Value: Not Defined
New Value: Do not show

Dir 4.5 (Advisory) Identifiers in the same namespace with overlapping visibility should be typographically unambiguous ✔ - No Deviations

Dir 4.6 (Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types ! - 112 Deviations

Rule ID: MISRA2012-DIR_4_6-b
Deviation Type: In-Code Suppression
Action: None
Risk/Impact: Undefined
Suppression Reason: suppress rule MISRA2012-DIR_4_6-b
Suppression Author: igarg

Powered by Parasoft Development Testing Platform. Copyright © 1996-2017.

Deviations

- Deviations must be considered carefully and clearly recorded — May be Specific or Project — Document all details, <u>including justification and why it is not unsafe</u>

Deviations

- Common example of deviation in embedded code is direct memory access

- Compliance — can only be claimed for a specific project — can not be claimed for an organization

Conclusion

- MISRA C is a useful approach to writing better, safer C code
- It is not specific to automotive
- The use of the guidelines is flexible in a well defined way
- For the most part, the guidelines just structure common sense practices

So What Tools the Software team needs?

- Code Coverage Tool with Coverage Features according to your ASIL needs, The Code Coverage Tools should be integrated To the Coding Standards Tool for ease of use and the to the ALM and Reporting Tool

- Demo available at our Booth

Coverage Level	ASIL A	ASIL B	ASIL C	ASIL D
Statement Coverage	+ +	+ +	+	+
Branch Coverage	+	+ +	+ +	+ +
MC/DC (Modified Condition/Decision Coverage)	+	+	+	+ +

workspace - C/C++ - sensor/sensor.c - Parasoft C/C++test

File Edit Source Refactor Navigate Search Project Parasoft Run Window Help

Project Explorer: asm_test, security testing example, sensor, Includes, Debug, stubs, tests, sensor.c, Makefile

```
void mainLoop2()
{
    int sensorValue;
    int status = -1;
    while (1) {
        status = readSensor(&sensorValue);
        if (status == STATUS_STOPPED) { // FIX: == instead of =
            break;
        } else if (status == STATUS_FAILED) {
            reportSensorFailure();
            break;
        }
        handleSensorValue2(sensorValue);
    }
    finalize();
}
```

Outline: stdio.h, stdlib.h, string.h, STATUS_OK : const int, STATUS_FAILED : const int, STATUS_STOPPED : const int, i : int, MAX_NUMBER_OF_SAMPLES : const int, readSensor(int*) : int, messages : char**, VALUE_LOW : const int, VALUE_HIGH : const int, ERROR : const int, initialize() : void

Line Coverage 63% [40/63 executable lines]

- sensor -- 63% [40/63 executable lines]
  - sensor.c -- 63% [40/63 executable lines]
    - handleSensorValue1 -- 0% [0/7 executable lines]
    - mainLoop1 -- 0% [0/11 executable lines]
    - reportSensorFailure -- 0% [0/2 executable lines]
    - main -- 80% [4/5 executable lines]
    - mainLoop2 -- 82% [9/11 executable lines]
    - finalize -- 100% [5/5 executable lines]
    - handleSensorValue2 -- 100% [7/7 executable lines]
    - initialize -- 100% [8/8 executable lines]
    - printMessage -- 100% [3/3 executable lines]

Writable | Smart Insert | 102 : 1

Table 9 — Structural coverage metrics at the software unit level

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

NOTE 2 The structural coverage can be determined by the use of appropriate software tools.

NOTE 3 In the case of model-based development, the analysis of structural coverage can be performed at the model level using analogous structural coverage metrics for models.

EXAMPLE 4 The analysis of structural coverage performed at the model level can replace the source code coverage metrics if it is shown to be equivalent, with rationales based on evidence that the coverage is representative of the code level.

NOTE 4 If instrumented code is used to determine the degree of structural coverage, it can be necessary to provide evidence that the instrumentation has no effect on the test results. This can be done by repeating representative test cases with non-instrumented code.

How It is Done? - Instrumentation

```
1 void foo()
2 {
3   found=false;
4   for (i=0;(i<100) && ( ! found );i++)
5   {
6     if (i==50) break;
7     if (i==20) found=true;
8     if (i==30) found=true;
9   }
10   printf("foo\n");
11 }
```

How It is Done? - Automatic Instrumentation.

Instrumentation for statement coverage

```
1 char inst[5];
 2 void foo()
 3 {
 4   found=false;
 5   for (i=0;(i<100) && (! found);i++)
 6   {
 7     if (i==50 ) { inst[0]=1;break;}
 8     if (i==20 ) { inst[1]=1;found=true;}
 9     if (i==30 ) { inst[2]=1;found=true;}
10     inst[3]=1; }
11    printf("foo\n");
12     inst[4]=1; }
```

How It is Done? - Instrumentation

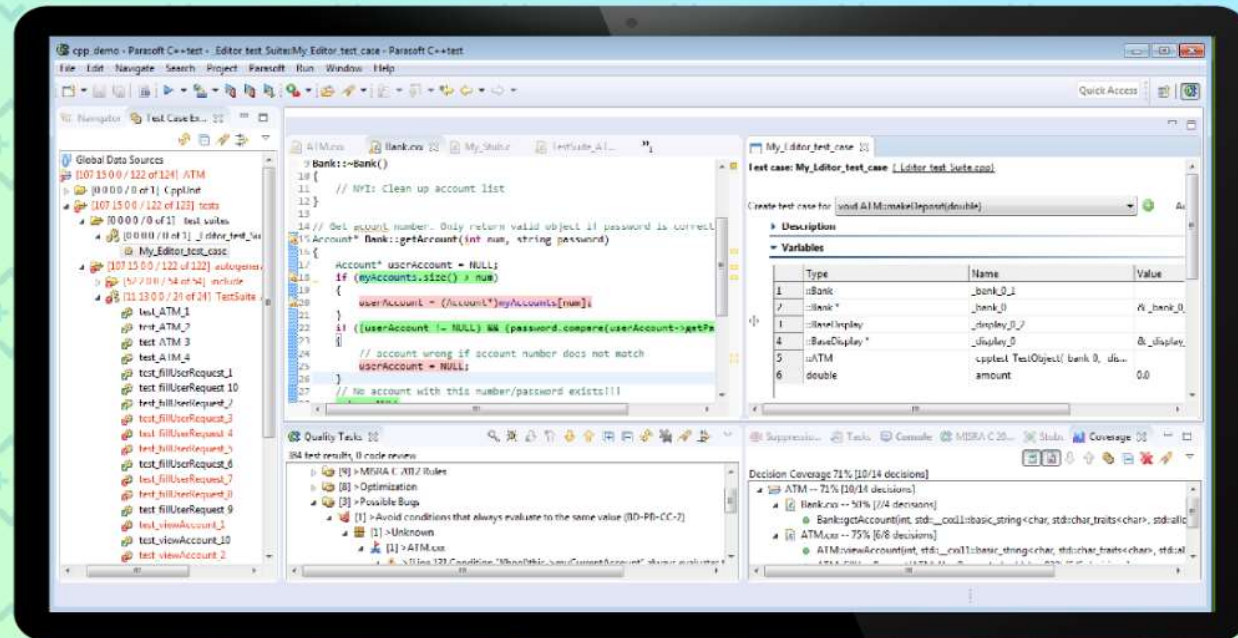
Inserting the full instrumentation code for the Branch coverage in this example will produce the following code:

```
   1 char inst[15];
   2 void foo()
   3 {
   4   found=false;
   5   for (i=0;((i<100)?inst[0]=1:inst[1]=1,0) && ((! f
ound)?inst[2]=1:inst[3]=1,0);i++)
   6   {
   7     if ((i==50?inst[4]=1:inst[5]=1,0) ) {
inst[6]=1;break;}
   8     if ((i==20?inst[7]=1:inst[8]=1,0) ) {
inst[9]=1;found=true;}
   9     if ((i==30?inst[10]=1:inst[11]=1,0) ) {
inst[12]=1;found=true;}
  10   inst[13]=1; }
  11   printf("foo\n");
  12 inst[14]=1; }
```

Full code coverage instrumentation at condition level

So What Tools the Software team needs?

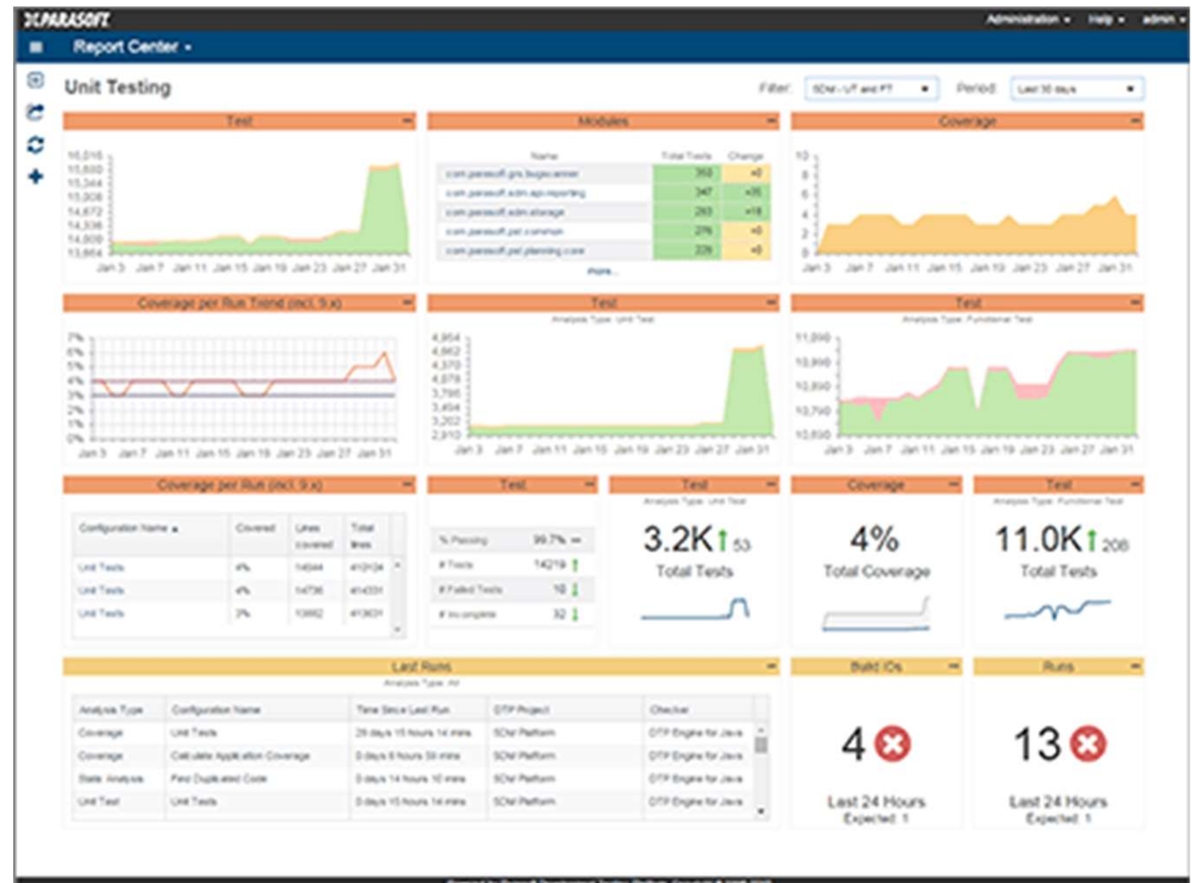
- Unit Testing Tool with Features according to your ASIL needs, The Unit Testing Tools should be integrated To the Coding Standards Tool and Code Coverage Tool for ease of use, and the to the ALM and Reporting Tool
- Demo available at our Booth



So What Tools the Software team needs?

- Unit Testing Tool should be integrated to reporting tool that gives you an overview on the Unit tests progress across your projects developers and teams

- Demo available at our Booth



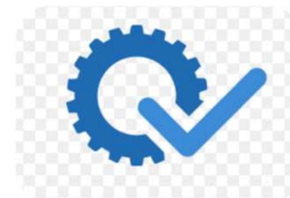
Software verification: A birdeye view

Typical development cycle and the relevant testing techniques

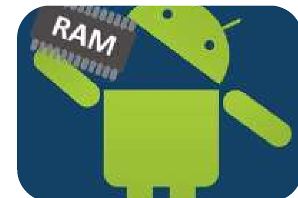
Coding standards

Unit testing

Integration testing



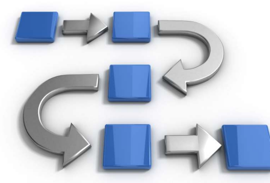
Functional testing



Memory error detection

Coverage analysis

Regression testing



Workflow automation

Peer code review & document inspections

Thank you !

Daniel Liezrowice
Daniel.l@eswlab.com
https://www.linkedin.com/in/liezrowice/