International Journal of Engineering & Technology, 7 (2.28) (2018) 344-350

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Safety critical software ground rules

Krishna Chaya Addagarrala [1] *, Patrick Kinnicutt [1]

[1] Department of Computer Science, Central Michigan University, Mount Pleasant, MI 48859, USA
**Corresponding author E-mail: kinni1p@lau.edu.lb*

Abstract

Safety critical software development field is one of the active research areas in many industries like automotive, medical, railways, nuclear and aerospace are placing increased value on safety and reliability. Safety critical software systems are those systems whose failure could result in the death or a serious injury to the people's life, security is one of the important topics in the field of safety-critical systems and it must be addressed completely in order to operate safety critical software successfully. In this paper we present a study about the set of standards and different ground rules to be followed in critical software development practices in different industries and the challenges in applying these standards. We also discuss the role of static analysis and software integrity levels in these standards, similarities in these standards and the set of activities followed in the development process of these standards.

Keywords: *MISRA; Safety Critical Standard; Static Analysis.*

1. Introduction

Safety critical systems in the automotive industry are life crit-ical systems which if malfunctioning may result in death or serious to human life. Due to these significant costs, safety critical systems must be designed, implemented and tested to ensure robust, efficient performance and no potentially haz-ardous software bugs. The C programming language is used commonly in automotive safety systems because it executes quickly, but it is a language prone to errors. Many of the prob-lems with using C in embedded systems arises from memory management errors from pointer misuse or buffer overflows. Another main difficulty with the C programming language lies in the differences in compiler implementations of the language grammar, leading to executables with different be-haviors based on the compiler options used and the different architectures of embedded systems. Standardized processed have been developed by standards bodies such as ANSI [4]. Many of the vulnerabilities possible in C can be mitigated through the use of well-designed programming rules. Devel-oping software for safety critical systems needs to consider all aspects of security and quality.

The safety integrity concept grew from development of safety critical systems in various industries such as the automotive, aerospace, medical and railway industries. The safety integrity concept was first introduced by the IEC 61508 standard and later it was taken up and inherited in various offshoot stan-dards. "Safety" as used in the safety critical software refers to developing software to prevent harm or catastrophe from happening. The safety integrity concept comprises two com-ponents:

- Integrity against random failures
- Integrity against systematic failures

The main difference between the two are the systematic failure cannot be quantified by the way of probabilistic computation and they mainly occur due to human errors during the different phases of software development process. Random failures result from hardware malfunctions and they occur randomly over time, due to aging and wear and tear on the hardware. Because of the nature of software, software applications are not subjected to random failure.

In this paper we present an overview and analysis of a set of good standards developed in different industries for the development of different safety critical software systems, as well as a list of static analysis tools used and their role in developing high-quality code.

2. Need for guidelines

2.1. Software quality

In general, people increasingly rely on more safety critical software systems in their mode of transportation, so devel-oping such safety critical software always to be correct and perceived to be correct becomes more important to help en-sure that catastrophes do not happen. In order to ensure the embedded software is correct, a unified approach is needed in software development with agreed standard techniques across any industry. In the automotive industry as an example, one safety critical system installed in the vehicle may include brak-ing and controlling a particular function like antilock braking during emergency stops. These braking components (hard-ware and software) are supplied by the original equipment manufacturer or a third party entity. Normally in automotive industry, most software developed as a part of the entire sys-tem is embedded software. Every vehicle manufacturer will specify some system specifications which ensure the follow-ing requirements:

- A set of interfaces to communicate with the sensors and other vehicle components;
- The functional performance of the software; and
- External environmental requirements such as climatic extremes and electromagnetic compatibility.

Software is considered one of the major components in automotive industry. When we compare the software with other hardware components we can find some similarities and dif-ferences between them:

Similarities:

Copyright © 2018 Krishna Chaya Addagarrala, Patrick Kinnicutt. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

- Both may be subject to continual improvement and development;
- both should be subject to strict quality control proce-dures; and.
- specialist skills are required in its development.

Differences:

- Errors in the software are systematic, not random;
- Software is considered intangible; and
- Software is perceived to be easy to change.

Any embedded safety critical software developed should un-dergo proper software development practices. Procedures and standards must be followed during the development and vali-dation of software in embedded systems to efficiently improve and maintain quality control. MISRA (Motor Industry Soft-ware Reliability Association) developed the first standards in November 1994; up to that time, no specific standards or guidelines existed in national or international vehicle software development. The primary reason behind the development of these standards is that every critical system development has standards and their integrity levels more strict than other software systems, and the automotive test environment uses many vehicle components and simulations to test systems and software extensively before they reach the customer. This lead to the development of standards for vehicle-based software systems [4].

Coding standards are used to improve software reliability and security. These coding standards include the set of rules that help the developers avoid dangerous language constructs; they also help limit the complexity of functions and maintain the standard coding structure by following the consistent syntac-tical and commenting styles specified in the standard. These coding standard rules help to reduce the occurrence of flaws and make it easier to maintain and test the software [11], as the code becomes more readable and is better documented.

It is very common that as the coding standards improve over time, it includes a set of rules whose objective is to accom-plish human code reviews. During code review, developers try to improve the software quality prior to deployment by exam-ining the code, fixing potential bugs and ensuring he coding standards are met. Developers use a set of static analysis tools during the code review that helps them determine whether any set of warnings may be related to code style or design or documentation. The role of static analysis tools is to analyze the source code, with the aim of complementing the compiler by highlighting potential issues that may arise in the software system like uninitialized variables, poorly commented code , etc. [15].

Compilers and other link chains like linkers/loaders by default often emit warnings rather than halt a build with a fatal error in the event of uninitialized variables or other potential issue. A warning during compilation is an indicator to the developer that a construct may be technically legal but questionable, or may be exercising a corner of the language that is not well defined. Such constructs are frequently the cause of subtle bugs. To ensure that developers do not intentionally or accidentally ignore warnings, the compiler can be configured to treat all warnings as errors. Many compilers have such an option [4], [1].

2.2. Static analysis

The software source code can be analyzed with static analysis using manual or automated methods. In manual analysis either a checklist or coding standards are used, while static analysis tools are used in the automated approach. The main objective of analyzing software is to ensure the absence of bugs in the software [11, 15]. Some coding standards used in the manual analysis are: MISRAC/C++, GNU Standards–C, JSFC++, CERT–java, JPL, Netrino, RUNTIME, CERT, CMSE, CON-FORM, CWE, DERA, and EADS.

Motor Industry Software Reliability Association (MISRA) is an organization that provides guidelines for embedded soft-ware development to support electronic components used in the automotive industry [12]. The main intention of MISRA is to give assistance to the automotive industry to make vehicle systems reliable and secure. These guidelines were geared towards the use of the C programming language in vehicle based software systems. The MISRA Guidelines are intended to achieve the following objectives to assure safety, robust-ness and security to the software, and minimization of both accidental and regular faults in the system design.

Currently, the MISRA standards are meant for the C and C++ programming languages. The first MISRA standard was issued as MISRA C:1998; this first release disseminated a set of guidelines for the use of the C programming language in vehicle based software. MISRA C:1998 has 127 rules, of which 93 are required and 34 are advisory; these rules are numbered in sequence 1 to 127 [15]. The second edition of the standards released was MISRA C:2004. These guidelines are used in safety critical systems; it contains 142 rules, of which 122 are required and 20 are advisory. Most of these guidelines can be reviewed using static analysis tools, while the remaining rules may be reviewed using dynamic analysis tools. For good software design of safety critical systems, both required and advisory rules must be considered in all projects even if they are not fully MISRA-compliant. The required rules must be implemented by developer, and the advisory rules should also be addressed or examined even though it's not compulsory in the standards. Several selected rules are normally not checked by the compiler [10], as shown in Table 1.

We conducted several experiments using these rules with some test results. These experiments were conducted using the lint static analysis tool and IDEAS development environment as well as the MISRA C:2004 standard for a Chrysler project. To explain what types of violations are specified and how the violations can be corrected, we have taken two required rules from MISRA to concentrate on.

2.2.1. Rule: 10.1

The value of an expression of integer type shall not be implic-itly converted to a different underlying type if:

1) It is not a conversion to a wider integer type of the same signedness;
2) The expression is complex;
3) The expression is not constant and is a function argument; or
4) The expression is not constant and is a return expression.

Here is an example:

```
Uint8 a = 0xffU;

Uint8 b = 0u;

Uint16 c =10u;

b = b + 5; /* not OK, 5 is signed */ b = b + 5U; /* OK, same signedness */
```

The test result using Qtool is shown in Fig. 1. The MISRA 10.1 rule violation report is shown in Fig. 1(a), while the violation code and the resolved code are shown in Fig. 1(b) and 1(c), respectively.

2.2.2. Rule12.5

The operands of a logical && or jj shall be primary-expressions. For example,

```
if ((x>c1) && (y>c2) || (z>c3)) /* not OK */ if ((x>c1) &&
((y>c2) || (z>c3))) /* ok extra braces () used.
```

Note the extra parentheses () used to explicitly specify the order of precedence for the logical or operation. The result of a MISRA 12.5 test violation is given in Fig. 2.

Again, this code snippet shows that the resolved code contains explicit parentheses in the logical expression to make it clear what the order of operations is intended to be.

2.3. Automated static analysis tools

MISRA is an organization with a lot of influence in the soft-ware development of automotive software systems. Members of MIS-RA include but are not limited to: Bentley Motors; Del-phi Diesel Systems; Ford Motor Company; and Jaguar Cars Ltd [10]. Be-cause of their influence, several software vendors sell analysis tools that support the MISRA standards [12];

(A) Qtool report for the MISRA 10.1 violation



(B) MISRA 10.1 violation code snippet (before change)

```
491 static uint8 LP_poll_digital_input( void )
492 {
493   IoHwAb_BoolType digital_value;
494   uint8 have_to_wakeup;
495   uint8 lub_i;
496
497   have_to_wakeup = 0U;
498
499   for(lub_i=0; lub_i < MAX_DIGITAL_WAKEUP_INPUTS; lub_i++)
500   {
501     raub_have_to_wakeup_digital[lub_i] = FALSE;
502   }
```

(C) MISRA 10.1 violation code snippet (after change)

```
491 static uint8 LP_poll_digital_input( void )
492 {
493   IoHwAb_BoolType digital_value;
494   uint8 have_to_wakeup;
495   uint8 lub_i;
496
497   have_to_wakeup = 0U;
498
499   for(lub_i=0u; lub_i < MAX_DIGITAL_WAKEUP_INPUTS; lub_i++)
500   {
501     raub_have_to_wakeup_digital[lub_i] = FALSE;
502   }
503
```

Fig. 1: Result Using Qtool.

Some of these vendors include Gimpel Software, Axivion, Cosmic Software and Green Hill Software, among others. Two popular automated static analysis tools are PC-Lint and RSM (Resource Standard Metrics). This section summarizes these tools.

2.3.1. PC-lint

The PC-lint is a static analysis tool. It will check the source code of C/C++ and figure out the bugs, inconsistencies, non-portable constructs, redundant code, etc. It is developed by Gimpel Soft-ware and it has been continuously maintained for more than 25 years [19].
Two examples demonstrating the types of violations PC-Lint can catch are [12]:

a) The goto keyword shall not be used, The PC-Lint can be configured to generate a warning message each time the go-to keyword appears in your C/C++ code by including dep-recate(keyword, goto, violates coding standard) in our local lint configuration file.
b) Comments Shall never be nested,The PC-Lint generates an error whenever such comments are found in the source code [12].

2.3.2. RSM (resource standard metrics)

The RSM is another static analysis tool for programs written in C/C++ and in general it is a source code static analysis tool. It's present version is 7.75, and it is provided by M Squared Technol-ogies. Some of the static analysis features in RSM includes being able to count number of lines of code vs white space and com-ments, number of function points, computing the cyclomatic com-plexity of individual functions, and much more, including lint-like features.

Table 1: Set of Rules Not Checked by Compiler

MISRA-C 1998	MISRA-C 2004	Definition	Reason
12 (adv)	6.3 (adv)	typedefs that indicate the size and the signed-ness should be used in place of basic type. Ex: unsigned long Received; /* not OK */ uint32 Received; /* ok */	Different compilers use differ-ent underlying types for the basic types. Com-mon case type is int which is 16 bit wide for one compiler and 32 bit wide for other com-pilers. Also if I declare as uint16 it is clear that the literal is never be negative and also it is easy to read and understand that the expres-sion never be negative.
24 (reg)	8.11 (reg)	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.	We know that by declaring a variable or func-tion static , it can be used by other module in the application. This rule forces you to really design the interfaces of new module
59 (reg)	14.8 (reg)	The statement forming the body of a switch, while, do ... while or for state-ment shall be a compound statement.	

One example where RSM may be useful is if automated no-tification for the lines of code exceeding the maximum line char-acter count: The length of all lines in a program shall be limited to a maximum of 80 characters.
It is always cheaper and easier to prevent a bug from creeping into code than it is to find and kill it after it has entered. A key strategy in this fight is to write code in which the compiler, linker, or a static-analysis tool can detect bugs automatically, or in other words, before the code is allowed to execute [10]

3. Universal standards

Different sets of standards have been developed for safety critical sofware systems in several industries such as railroad, medical, and aerospace. Table 2 shows the industry and the corresponding standard that applies to the industry.

Table 2: Industries and Applicable Safety Critical Software Standards

Industry	Standard
Automotive	ISO26262
Aerospace	DO178B
Medical	IEC62304
Railway	EN50128

3.1. ISO26262 automotive functional safety standard

The ISO 26262 discusses the importance of an automotive specific international standard that focuses on the safety crit-ical compo-nents. It is a derivative of the IEC 61508, the generic functional safety standard for electrical and electronic systems. The high increase of complexity in the automotive industry resulted in the industry putting significant efforts to provide robust and respon-sive safety compliant systems. ISO 26262 uses a set of steps to manage the functional safety and to regulate the product develop-ment on a system at both the hardware and software levels.
The main goal of the ISO 26262 is as listed below [13].

a) It provides an automotive safety lifecycle (management, de-velopment, production, operation, service, decommis-sioning) and supports tailoring the necessary activities dur-ing these lifecycle phases;

b) It provides an automotive specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs);
c) Covers functional safety aspects of the entire develop-ment process (including such activities as requirements specification, design, implementation, integration, verifi-cation, validation, and configuration); and
d) It provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety is being achieved.

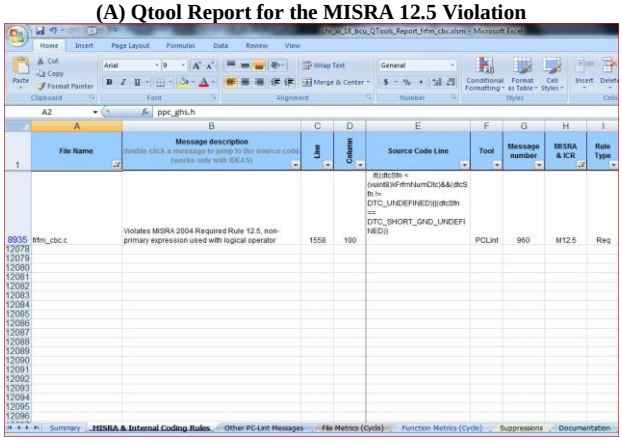
Here in this paper we mainly discuss the product development at the software level and safety integrity levels as defined by ISO 26262. The ISO 26262 specifies the set of steps needed to be considered for the product development at the software level in the automotive industry to provide the safety required. Steps discussed in ISO 26262 include [18]

a) Requirements for initiation of product development at the software level;
b) Specification of the software safety requirements;
c) Software architectural design;
d) Software unit design and implementation;
e) Software unit testing;
f) Software integration and testing; and
g) Verification of software safety requirements.

3.1.1. Initiation of software development

In this phase, the software development process to be used needs to be selected as well as the tools used. As an example, if we are using model based software development in our organization, then we will need to create the development plan for the development of one or more models and also the plan to confirm that the implementation behaves as the same as model. For tool selection, we need to decide what language the model and implementation will use and which tool we will be using during entire the software development phase [6], [18].

(A) Qtool Report for the MISRA 12.5 Violation



(B) MISRA 12.5 Violation Code Snippet (Before Change)

```
1553 void CBC_FrfmDebounceEventFailed_BOSS(t_FrfmDtcSfnType dtcSfn, const uint8 entrynum, vuint8 stepSize, vuint16 *
1554 {
1555    uint8 ret1;
1556    uint8 ret2;
1557    /*Check for a undefined DTC*/
1558   if((dtcSfn < (vuint8)kFrfmNumDtc)&&(dtcSfn != DTC_UNDEFINED)||(dtcSfn == DTC_SHORT_GND_UNDEFINED))
1559    {
1560       switch(dtcSfn)
1561       {
```

(D) MISRA 12.5 Violation Code Snippet (After Change)

```
1553 void CBC_FrfmDebounceEventFailed_BOSS(t_FrfmDtcSfnType dtcSfn, const uint8 entrynum, vuint8 stepSize, vuint16 * op
1554 {
1555    uint8 ret1;
1556    uint8 ret2;
1557    /*Check for a undefined DTC*/
1558   if((dtcSfn < (vuint8)kFrfmNumDtc)&&((dtcSfn != DTC_UNDEFINED)||(dtcSfn == DTC_SHORT_GND_UNDEFINED)))
1559    {
1560       switch(dtcSfn)
1561       {
```

Fig. 2: Result Using Qtool.

3.1.2. Safety requirement specification

The main objective of this phase is to find out all the software based functions which could potentially impact the safety by either: directly producing an unsafe state, such as a software module which controls traction while braking; or failing to correctly handle a hardware or software fault. The require-ments for each identified software component must be deter-mined, including such things as timing requirements and the interface between the software component and other system components [6], [18].

3.1.3. Architectural design

During this phase, a high-level design for each software component is developed and a validator confirms that the safety requirements must be verified for design [6, 18].

3.1.4. Software unit design and implementation

During this step, each subsystem is designed and implemented [6], [18].

3.1.5. Unit Testing

Getting integrated into the overall system.

3.1.6. Integration testing

During integration testing, the safety of the software system is validated to ensure the different units communicate well with each other [6], [18].

3.1.7. Safety requirements verification

Safety requirements verification is performed in order to en-sure that the embedded software works correctly in its target environment [6], [18].
ASIL (Automotive Safety Integrity level (or SIL)) defines the required degree of rigor in technical, organizational and pro-cess work. These are divided into four levels based on safety requirements ASIL A to ASIL D. Level A defines the low-est safety classification, while D defines the highest. This is the key component of the ISO 26262 and it is determined at the beginning of the development process. Each of the four levels specifies the item or elements necessary requirements of ISO 26262 and safety measures to apply for avoiding an unreasonable residual risk. In ISO 26262 safety requirements are structured into different levels where the main intention is that "safety requirements at one level fully implement all the safety requirements of previous level". These functional safety requirements are derived from the safety goals and these are assigned to ASIL (Extending Contract Theory with safety Integrity levels).

3.2. DO178B standard for aircraft software systems

The DO 178B is the first software safety standard that address safety for an avionics industry. It provides guidelines in the area of software development, configuration management, verification and the interface to approval authorities. Here the software development is much like the waterfall model [8].

3.2.1. Software integrity levels

DO 178B is the only document that describes the lifecycle process of embedded software development in aircraft systems. It has five levels of software also called as design assurance levels as shown below:
Level A Catastrophic: Failure may cause a crash. Error or loss of critical function required to safely fly and land air crafts.
Level B Hazardous: Failure has large impact negatively on safety or performance or passenger injury.
Level C Major: Failure is significant but less impact than level B. Passenger may feel discomfort rather than injury.

Level D Minor: Failure is noticeable but lesser than level C. Passenger may feel inconvenience or a routine flight plan change.
Level E No Effect: Failure has no impact on safety, aircraft operation or crew work load.

3.2.2. DO178B software development process (aeronautics)

Here each low level software component is tested indepen-dently [6], [18]. All components must pass unit testing before
The software development process involves five main steps [8] given in Table 3.

Table 3: DO178B Aircraft Software Development Process

Step	Description
Software Planning	System Requirements are the main inputs to the entire project.
Software Development	The DO178B is not considered as a standard for software but it is assumed as a software assurance using the set of tasks to meet the objectives and the level of accuracy. The outcome of the development process includes • Software Requirements Data (SRD) • Software Requirements Data (SRD) • Source code •Executable object code Traceability from the high level requirements to source code or object code is highly required and typically uses one of the software development process
Software Verification	This step include testing and verification, including these tasks: •Software Verification cases and Procedures (SVCP) • Software Verification Results (SVR): Review of all requirements, Design and Code. Testing of executable Object code. Code coverage analysis. • Analysis of the entire code and traceability from tests and results to all requirements is typically required (depending on software levels). It typically involves requirement based test tools and code Coverage analysis tools • Other names for test performance in this process are often used, such as unit testing, integration testing, and black-box or acceptance testing
Configuration Management	The documents are maintained by the configuration management process, including •Software Configuration index (SCI) • Software lifecycle environment Configuration index (SECI) It handles the problem reports, changes and relevant activities. It mainly provides the archives and revision identification of •Source code analysis development • Other development environments such as test/analysis tools. •Software integration tool. •All other document's, Software and hardware.
Quality Assurance	The outputs of the quality assurance process are •Software quality assurance record (SQAR) • Software Conformity review (SCR) • Software Accomplishment Summary (SAS)

3.3. IEC62304 for medical device software

The IEC62304 standard mainly applicable to the medical device software development. it may be the independent device or embedded one or an integral part of the final device. This standard doesn't cover the validation and testing of the final release of the medical device [16].
This standard defines the lifecycle requirements for the med-ical device software, defines the life of a medical software from the definition of the requirements for manufacturing, given in Table 4.
The software safety classification identifies three classes of medical software in accordance with the possible effects on patients, operator or any other people effected by the software related hazard. The medical software should be classified based on the severity of the software. The classes and the activities involved are given in Table 5.

Table 4: IEC 62304 Standard Steps for Medical Device Software

Step	Description
1	Identifies the methods, activities, and tasks involved in the development of a software
2	Describes the sequence of dependencies between the activities and tasks
3	Identifies the milestones at which the completeness of the specified deliverables is verified
4	The set of activities involved in this development process are • Software development planning • Software requirement analysis • Software architectural design
5	The IEC62304 lifecycle process involves • Detailed design • Unit implementation and verification • Integration and integration testing • System testing • Release
6	The maintenance of the software process is considered as important as development process which includes • Generate software maintenance plan • Problem and Modification Analysis
7	The two additional processes involves in the medical software development are • Software configuration management process • Software problem resolution process

Table 5: IEC62304 Standard Classes for Medical Device Software

Class	Description and Activities
A	No injury or damage to health is possible. Activities: • Development plan • Requirement analysis • Unit implementation and verification
B	Non-Serious injury is possible. Activities: • Development plan • Requirement analysis • Architectural design • Detailed design • Unit Implementation and Verification • Integration and integration testing • System testing • Release
C	Death or Serious injury is possible. Activities: • Development plan • Requirement analysis • Architectural design • Detailed design • Unit Implementation and Verification • Integration and integration testing • System testing • Release

3.4. EN50128 standard for railway software systems

The EN50128 standard is mainly used for Railway applica-tions – Communications, Signaling and processing systems software for railway control and protection systems gives set of procedures and technical requirements with which the development, deployment and maintenance of any pro-grammable electronic software particularly related to safety and mainly intended for railway control and protection appli-cations [3].
In order to produce the "0-fault" software, the below list of techniques is recommended by CENELEC EN50128 [3].

1) Use of conventional methods
2) Use of dynamic tests
3) Use of qualifiable development environment
4) Use of simulation methods to validate the model and /or select the tests to be run
5) Formalization of team work methods (Specification, encoding, verification, validation,etc.)

6) Implementation of strong quality assurance (using a pre-established disciplined system)

This standard defines several software safety integrity levels (SSIL) as SIL_i, i = 0; 1; 2; 3; 4. The classification number determines the measure that have to be met in order to reduce residual software faults to an appropriate level. EN50128 have 0 through 4 where $SSIL_4$ represents the highest level and $SSIL_0$ represents the lowest level of safety integrity and the CENELEC 50128 is applied for both safety related and non-safety related applications for this reason it introduces $SSIL_0$ which is for non-safety related applications. The SIL_i levels are:

Catastrophic impact (high level) impact on the Community major protection of the installation and of production, or risk of injury to employees minor protection of the installation and of production (lowest level).

Static analysis is a way of performing a check without execut-ing the code of the application. The different static analysis measures applied in software safety integrity levels (SSIL) are given in Table 6. In the table, R is for Recommended, HR is for Highly Recommended.

Table 6: Static Measures Applicable for SSIL [CENELEC 50128:2011]

Measure	$SSIL_0$	$SSIL_1$, $SSIL_2$	$SSIL_3$, $SSIL_4$
1. Boundary-value analysis	–	R	HR
2. Control lists	–	R	R
3. Control flow analysis	–	HR	R
4. Data flow analysis	–	HR	HR
5. Error guessing	–	R	R
6. Design review / check	HR	HR	HR

4. Challenges in safety critical systems

In one or the other way for people in the software community working on Safety Critical Systems technology, safety critical system is an application where human safety depends on the correct operation of the application. The major challenges are specified in [9], as we also briefly describe below.

The important point in safety critical systems is security and it must be named in order to work successfully. The major difficulty here exists very much in the software engineering than security. Many security difficulties that arise in network information systems appears because of software defects make the systems weak to attacks. Even now such attacks exist's because system continues to be deployed with vulnerabilities.

In some cases, what amounts to completely new technologies are required. The number of interacting safety-critical sys-tems present in a single application will force the sharing of resources between systems. This will eliminate a major ar-chitectural element that gives confidence in correct operation-physical separation. Knowing that the failure of one system cannot affect another greatly facilitates current analysis tech-niques.

5. Related work

There are many research papers published on the topics of safety standards for critical software development. A review of activities that MISRA is provided in [20]. The review presented in this paper is somewhat different, though, that we presented the real time implementation results for some of the required rules of MISRA.

Johansson [7] presented the functional safety requirements allocated to an environment with different ASIL values. In Bhansali's book [2], the different safety standards for the minimum subset required for truly universal safety critical software standard were analyzed. Different from the contents in the book, we presented in this paper an overview of differ-ent activities and software development process, involved in those standards.

The ISO 26262 and IEC 61508 Safety integrity levels applied to the top level safety requirements on a system was given in [21]. The difference between our work and [21] is that we presented the overall overview of safety integrity levels involved in the different industry standards and differences and similarities between them. Erkkinen presented the safety critical software development practices and tools used in that process [5]. Our work differs from [5] is that we mainly discussed in static analysis and safety integrity levels.

Panaroni et al. did a survey [14] on safety critical behavior of the vehicle and designing such functions by adopting the opportune methods and practices. Lee et al. discussed the evolution of the DO-178 and other ssafety tandards and the set of features to be considered while developing new standards. Our work presented in this paper is different in safety integrity levels, in addition, we also stated an overview of software development process involved in different standards.

Major challenges and directions in safety critical systems were described in [9]. We briefly discussed these challenges and directions in Section 4.

6. Similarities between Different Standards

EN50128, DO-178B, and ISO26262 are derived from the IEC 61508 standard (General Elctrical/Programmable elec-tronic devices) The considerable similarities between these standards. All these standards are offers the guidance for core software development process. The EN50128 process is based on waterfall and V-model where as the ISO 26262 describes the software lifecycle under V-model frame work and it is allowed to use the agile development for all the stan-dards. Related to the safety integrity levels all these standard use the different terminology like Safety integrity levels (SIL), Automotive Safety integrity Levels (ASIL) to check the safety levels. With the higher safety systems need more checks and high control [17].

7. Conclusion

This paper is meant to provide sets of standards and guidelines followed in different industries in the field of safety critical software development. We hope the paper can help as a point of reference in the automotive, medical, aerospace, and rail-way industries for the developer creating embedded software to get an overview about the development process and SIL involved.

Acknowledgments

Thanks go to Dr. Poonam Dharam and Dr. Gonzhu Hu for their assistance in helping with the literature review and the proofreading.

References

[1] M. Barr. How to enforce coding standards automatically. https://www.embedded.com/ electronics-blogs/barr-code/4218283/ How-to-enforce-coding-standards-automatically, 2011.

[2] P. V. Bhansali. Universal software safety standard. SIGSOFT Software Engineering Notes, 30(5), September 2005.

[3] J.-L. Boulanger. CENELEC 50128 and IEC 62279 Standards. Wiley, 2015.

[4] David and M. Kleidermacher. Using coding standards to improve software quality and security. https://www. embedded.com/design/safety-and-security/4418986/2/ Using-coding-standards-to-improve-software-quality-and-security, 2013.

[5] T. J. Erkkinen. Safety critical software generation. In IEEE International Symposium on Computer Aided Control System Design, pages 237–242. IEEE, 1999.

[6] ISO. Iso 26262-10:2012(en) road vehicles – functional safety – part 10: Guideline on iso 26262. https://www.iso.org/obp/ui/#iso: std: iso: 26262:-10: ed-1:v1:en, 2012.

[7] R. Johansson and J. Nilsson. The need for an environment perception block to address all asil levels simultaneously. In IEEE Intelligent Vehicles Symposium, pages 1–4. IEEE, 2016.

[8] L. A. S. Johnson. Do-178b, "software considerations in airborne systems and equipment certification". http://www.stsc.hill.af.mil/CrossTalk/1998/oct/schad.asp, 1998.

[9] J. C. Knight. Safety critical systems: Challenges and directions. In International Conference on Software Engineering, pages 547–550. IEEE, 2002.
[10] A. Lindgren. Misra c — some key rules to make embedded systems safer. https://www.scribd.com/document/333699819/ MISRA-C-Some-key-rules-to-make-embedded-systems-safer-pdf.
[11] N. Manzoor, H. Munir, and M. Moayyed. Comparison of static analy-sis tools for finding concurrency. In IEEE 23rd International Sympo-sium on Software Reliability Engineering Workshops, pages 129–133. IEEE, 2012.
[12] MISRA. MISRA Standards. https://www.misra.org.uk.
[13] National Instruments. What is the iso 26262 functional safety standard? http://www.ni.com/white-paper/13647/en, 2014
[14] P. Panaroni, G. Sartori, and F. Fabbrini. Safety in automotive software: An overview of current practices. In 32nd Annual IEEE International Conference on Computer Software and Applications, pages 1053–1058. IEEE, 2008.
[15] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In IEEE 22nd International Conference onSoftware Analysis, Evolution and Reengineering, pages 161–170. IEEE, 2015.
[16] PharmOut Pty Ltd.Implementation of ansi/aami/iec 62304 medical device software lifecycle processes. https://40rik02ft2xye26xv2i0y0yc-wpengine. netdna-ssl.com/downloads/ white-paper-medical-device-software-lifecycle-processes. Pdf, 2016.
[17] QA Systems. Achieving en 50128 compliance with qa.c and qa.c++. https://www.qa-systems.de/ressourcen/details/ achieving-en-50128-compliance-with-qa-c-and-qa-c.
[18] Reactive Systems. Achieving iso26262 compliance with reac-tis. http://www.reactive-systems.com/papers/iso-26262. Pdf, 2015.
[19] L. Torri, G. Fachini, L. Steinfeld, V. Camara, L. Carro, and E. Cota. An evaluation of free/open source static analysis tools applied to embedded software. In 11th Latin American Test Workshop, pages 1–6. IEEE, 2010.
[20] D. D. Ward. Misra standards for automotive software. In The 2nd IEE Conference on Automotive Electronics, pages 5–18. IEEE, 2006.
[21] J. Westman and M. Nyberg. Extending contract theory with safety integrity levels. In IEEE 16th International Symposium on High Assurance Systems Engineering, pages 85–92. IEEE, 2015.