The MISRA C Coding Standard: A Key Enabler for the Development of Safety- and Security-Critical Embedded Software

Roberto Bagnara*
BUGSENG and University of Parma
Parma, Italy
Email: roberto.bagnara@bugseng.com

Abramo Bagnara
BUGSENG
Parma, Italy
Email: abramo.bagnara@bugseng.com

Patricia M. Hill
BUGSENG
Parma, Italy
Email: patricia.hill@bugseng.com

***Abstract*—Building embedded control systems that embody industry best practices for safety and security is a challenging task: doing so in unrestricted C is even more challenging. C is a general-purpose programming language, partially defined by an ISO standard written in natural language with a slow evolution over the last 40+ years. Its many strong points make it the most used language for the development of embedded systems. Unfortunately, the origin of C's strength is also the origin of C's weakness: the language has many aspects that are not fully defined, it has some rather obscure aspects that can easily induce programmers into error, and it has no run-time error detection facilities. MISRA C is a coding standard defining a subset of the C language, initially targeted at the automotive sector, but now adopted across all industry sectors that develop C software in safety- and/or security-critical contexts. In this talk, we introduce MISRA C, its key role in the development of critical embedded systems' software and its relevance to industry safety and security standards. We explain why and how MISRA C retains 95% of the advantages of C and eradicates 95% of its drawbacks: with the right tools, training and professional expertise, the adoption of MISRA C, besides satisfying some important requirements imposed by safety standards, can significantly decrease development times and costs.**

I. Introduction

The development of the C programming language started 50 years ago in 1969 at Bell Labs where the language was used for the development of the Unix operating system [1]. Despite frequent criticism, C is still one of the most used programming languages overall[1] and the most used one for the development of embedded systems [2], [3]. There are several reasons why C has been and is so successful:

- C compilers exist for almost any processor, from tiny DSPs used in hearing aids to supercomputers.
- C compiled code can be very efficient and without hidden costs, i.e., programmers can roughly predict running times even before testing and before using tools for worst-case execution time approximation.[2]
- C allows writing compact code thanks to limited verbosity and the availability of many built-in operators.
- C is defined by international standards: it was first standardized in 1989 by the American National Standards Institute (this version of the language is known as ANSI C) and then by the International Organization for Standardization (ISO) [7], [8], [9], [10], [11], [12].
- C, possibly with extensions, allows easy access to the hardware and this is a must for the development of embedded software.
- C has a long history of usage in all kinds of systems including safety-, security-, mission- and business-critical systems.
- C is widely supported by all sorts of tools.

Claims that C would eventually be superseded by C++ do not seem very plausible, at least as far as the embedded software industry is concerned. In addition to the already-stated motives, there is language size and stability: C++ has become a huge, very complex language; moreover it is evolving at a pace that is in sharp contrast with industrial best practices. The trend whereby C++ rapid evolution clashes with the industry requirements for stability and backward compatibility was put in black-and-white at a recent WG21 meeting,[3] where the following statement was agreed upon [13]: "The Committee should be willing to consider the design / quality of proposals even if they may cause a change in behavior or failure to compile for existing code."

The characteristics that made the C programming language so successful have downsides: writing safe and secure applications in C requires particular care. The solution mandated or strongly suggested by all applicable industrial standards is

* While Roberto Bagnara is a member of the *MISRA C Working Group* and of ISO/IEC JTC1/SC22/WG14, a.k.a. the *C Standardization Working Group*, the views expressed in this paper are his and his coauthors' and should not be taken to represent the views of either working group.

[1] Source: TIOBE Index for June 2018, see https://www.tiobe.com/tiobe-index/.

[2] This is still true for implementations running on simple processors, with a limited degree of caching and internal parallelism. Prediction of maximum running time without tools becomes outright impossible for current multi-core designs such as Kalray MPPA, Freescale P4080, or ARM Cortex-A57 equivalents (see, e.g., [4], [5], [6]).

[3] WG21 is a common shorthand for ISO/IEC JTC1/SC22/WG21, a.k.a. the C++ *Standardization Working Group*. The cited meeting took place in Jacksonville, FL, USA, March 12–17, 2018.

language subsetting. In this paper, we introduce MISRA C, which is increasingly recognized as the most authoritative subset of the C programming language in all industry sectors. In doing so, we try to clear up some misconceptions around the language, its standardization process and MISRA C itself.

The plan of the paper is the following: Section II introduces the main shortcomings of C, explaining why C is (not completely) defined as it is, why it is not going to change substantially any time soon, and why subsetting it is required; Section III introduces the MISRA project and MISRA C focusing on its last edition, MISRA C:2012, with its amendments and addenda; Section IV presents a few MISRA C guidelines in order to convey the *look and feel* of the coding standard; Section V emphasizes some points that are crucial for a proper understanding of MISRA C; Section VI emphasizes some key factors for its successful adoption; Section VII concludes.

II. C Non-definite Behavior

The main source of problems with C comes from the notion of *behavior*, defined as *external appearance or action* [10, Par. 3.4] and the so-called *as-if rule*, whereby the compiler is allowed to do any transformation that ensures that the "observable behavior" of the program is the one described by the Standard [10, Par 5.1.2.3#5].[4] While all compiled languages have a sort of *as-if rule* that allows optimized compilation, one peculiarity of C is that it is not fully defined. There are four classes of not fully defined behaviors (in the sequel, collectively referred to as "non-definite behaviors"):

implementation-defined behavior: *unspecified behavior where each implementation documents how the choice is made* [10, Par. 3.4.1]; e.g., the sizes and precise representations of the standard integer types;

locale-specific behavior: *behavior that depends on local conventions of nationality, culture, and language that each implementation documents* [10, Par. 3.4.2]; e.g., character sets and how characters are displayed;

undefined behavior: *behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements* [10, Par. 3.4.3]; e.g., attempting to write a string literal constant or shifting an expression by a negative number or by an amount greater than or equal to the width of the promoted expression;

unspecified behavior: *use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance* [10, Par. 3.4.4]; e.g., the order in which sub-expressions are evaluated.

Setting aside locale-specific behavior, whose aim is to avoid some nontechnical obstacles to adoption, it is important to understand the intimate connection between non-definite behavior and the relative ease with which optimizing compilers can be written. In particular, C data types and operations can be directly mapped to data types and operations of the target machine. This is the reason why the sizes and precise representations of the standard integer types are implementation-defined: the implementation will define them in the most efficient way depending on properties of the target CPU registers, ALUs and memory hierarchy.

Overflow on signed integer types is undefined behavior because the C Standard allows different representations of signed integers, such as two's complement, ones' complement and sign-magnitude. For the latter two possibilities it is implementation-defined whether the *negative zero* bit pattern is a *trap representation*.[5] The C compiler can thus assume signed integer overflow cannot happen, omit all checks for overflow, and compile, e.g.,

```
int always_true(int v) {
  return (v + 1 > v) ? 1 : -1;
}
```

as if it was

```
int always_true(int v) {
  return 1;
}
```

Incidentally, implementation latitude on the representation of signed integers is also the reason why all bitwise operation on signed integers have implementation-defined behavior.

Attempting to write on string literal constants is undefined behavior because they may reside in read-only memory and/or may be merged and shared: for example, a program containing `"String"` and `"OtherString"` may only store the latter and use a suffix of that representation to represent the former.

The reason why shifting an expression by a negative number or by an amount greater than or equal to the width of the promoted expression is undefined behavior is less obvious. What sensible semantics can be assigned to shifting by a negative number of bit positions? Shifting in the opposite direction is a possible answer, but this is usually not supported in hardware, so it would require a test, a jump and a negation. It is a bit more subtle to understand why the following is undefined behavior:

```
uint32_t i = 1;
i = i << 32;  /* Undefined behavior. */
```

[4] In this paper, we refer to the C99 language standard [9] because this is the most recent version of the language that is targeted by the current version of MISRA C [14]. All what is said about the C language itself applies equally, with only minor variations, to all the published versions of the C standard.

[5] Trap representations are particular object representations that do not represent values of the object type. Simply reading a trap representation (except by an lvalue of character type) is undefined behavior. For instance, in a memory architecture with explicit parity bits, a representation with the wrong parity bit can be a trap representation.

One would think that pushing 32 or more zeroes to the right of `i` would give zero. However, this does not correspond to how some architectures implement shift instructions. IA-32, for instance [15, section on "IA-32 Architecture Compatibility"]:

> The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

This means that, on all IA-32 processors starting with the Intel 286, a direct mapping of C's right shift to the corresponding machine instruction will give:

```
  i = i << 32;
/* This is equivalent to... */
  i = i << (32 & 0x1F);
/* ... this, i.e., ...      */
  i = i << 0;
/* this, which is a no-op.  */
```

So also for this case, for speed and ease of implementation, C leaves the behavior undefined.

The recurring request to WG14[6] to "fix the language" is off the mark. In fact, weakness of the C language comes from its strength:

- Non-definite behavior is the consequence of two factors:
  1) the ease of writing efficient compilers for almost any architecture;
  2) the existence of many compilers by different vendors and the fact that the language is standardized.

  Concerning the second point, it should be considered that, in general, ISO standardizes existing practice taking into account the opinions of the vendors that participate in the standardization process, and with great attention to backward compatibility: so, when diverging implementations exist, non-definite behavior might be the only way forward.
- The objective of easily obtaining efficient code with no hidden costs is the reason why, in C, there is no run-time error checking.
- Easy access to the hardware entails the facility with which the program state can be corrupted.
- Code compactness is one of the reasons why the language can easily be misunderstood and misused.

Summarizing, the C language can be expected to remain faithful to its original spirit and to be around for the foreseeable future, at least for the development of embedded systems. However, it is true that several features of C do conflict with both safety and security requirements. For this reason, *language subsetting* is crucial for critical applications. This was recognized as early as 1995 [16] and is now mandated or recommended by all safety- and security-related industrial standards, such as IEC 61508 (industrial), ISO 26262 (automotive), CENELEC EN 50128 (railways), RTCA DO-178B/C (aerospace) and FDA's *General Principles of Software Validation* [17]. Today, the most authoritative language subset for the C programming language is *MISRA C*, which is the subject of the next section.

[6]Short for ISO/IEC JTC1/SC22/WG14, a.k.a. the *C Standardization Working Group*.

III. MISRA C

The MISRA project started in 1990 with the mission of providing world-leading best practice guidelines for the safe and secure application of both embedded control systems and standalone software. The original project was part of the UK Governments "SafeIT" programme but it later became self-supporting, with MIRA Ltd, now HORIBA MIRA Ltd, providing the project management support. Among the activities of MISRA is the development of guidance in specific technical areas, such as the C and C++ programming languages, model-based development and automatic code generation, software readiness for production, safety analysis, safety cases and so on. In November 1994, MISRA published its "Development guidelines for vehicle based software", a.k.a. "The MISRA Guidelines" [18]: this is the *first* automotive publication concerning functional safety, more than 10 years before work started on ISO 26262 [19].

The MISRA guidelines [18] prescribed the use of "a restricted subset of a standardized structured language." In response to that, the MISRA consortium began work on the MISRA C guidelines: at that time Ford and Land Rover were independently developing in-house rules for vehicle-based C software and it was recognized that a common activity would be more beneficial to industry. The first version of the MISRA C guidelines was published in 1998 [20] and received significant industrial attention.

In 2004, following the many comments received from its users —many of which, beyond expectation, were in non-automotive industries— MISRA published an improved version of the C guidelines [21]. In MISRA C:2004 the intended audience explicitly became constituted by *all* industries that develop C software for use in high-integrity/critical systems. Due to the success of MISRA C and the fact that C++ is also used in critical contexts, in 2008 MISRA published a similar set of *MISRA C++* guidelines [22].

Both MISRA C:1998 and MISRA C:2004 target the 1990 version of the C Standard [7]. In 2013, the revised set of guidelines known as MISRA C:2012 was published [14]. In this version there is support both for C99 [9] as well as C90 (in its amended and corrected form sometimes referred to as C95 [8]). With respect to previous versions, MISRA C:2012 covers more language issues and provides a more precise specification of the guidelines with improved rationales and examples. Figure 1 shows part of the relationship and influence between the MISRA C/C++ guidelines and other sets of guidelines. It can be seen that MISRA C:1998 influenced Lockheed's

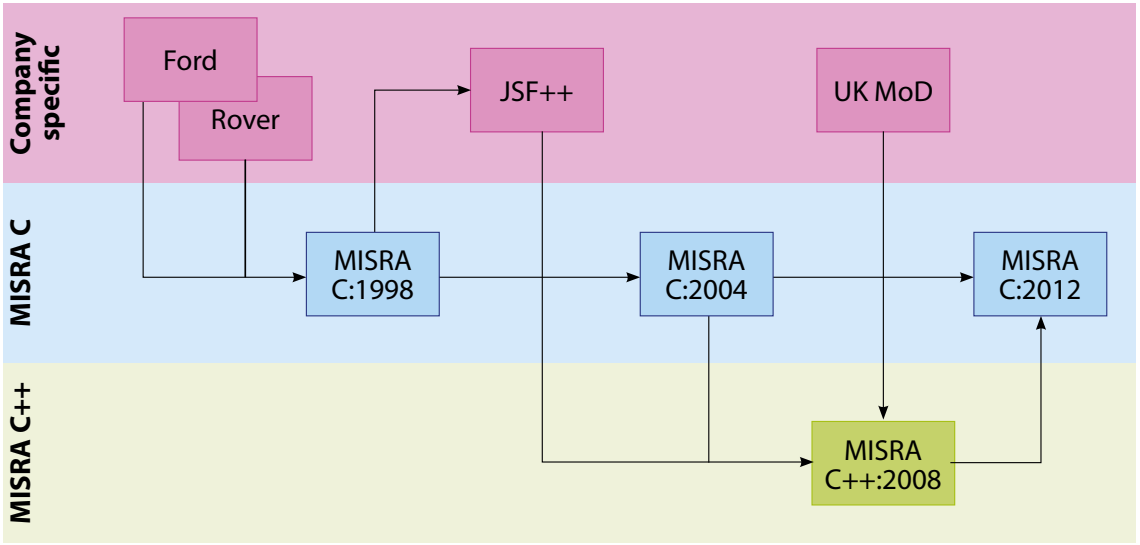

Fig. 1. Origin and history of MISRA C

"JSF Air Vehicle C++ Coding Standards for the System Development and Demonstration Program" [23], which influenced MISRA C++:2008, which, in turn, influenced MISRA C:2012. The activity that led to MISRA C++:2008 was also encouraged by the UK Ministry of Defence which, as part of its *Scientific Research Program*, funded a work package that resulted in the development of a "vulnerabilities document" (the equivalent of Annex J listing the various behaviors in ISO C, which is missing in ISO C++, making it hard work to identify them and to ensure they are covered by the guidelines). Moreover, MISRA C deeply influenced NASA's "JPL Institutional Coding Standard for the C Programming Language" [24] and several other coding standards (see, e.g., [25], [26], [27]).

The MISRA C guidelines are concerned with aspects of C that impact on the safety and security of the systems, whether embedded or standalone: they define "a subset of the C language in which the opportunity to make mistakes is either removed or reduced" [14]. The guidelines ban critical non-definite behavior and constrain the use of implementation-defined behavior and compiler extensions. They also limit the use of language features that can easily be misused or misunderstood. Overall, the guidelines are designed to improve reliability, readability, portability and maintainability.

There are two kinds of MISRA C guidelines.

Directive: a guideline where the information concerning compliance is generally not fully contained in the source code: requirements, specifications, design, ... all may have to be taken into account. Static analysis tools may assist in checking compliance with respect to directives if provided with extra information not derivable from the source code.

Rule: a guideline where information concerning compliance is fully contained in the source code. Discounting undecidability, static analysis tools should, in principle, be capable of checking compliance with respect to the rule.

A crucial aspect of MISRA C is that it has been designed to be used within the framework of a documented development process where justifiable non-compliances will be authorized and recorded as *deviations*. To facilitate this, each MISRA C guideline has been assigned a category.

Mandatory: C code that complies to MISRA C must comply with every mandatory guideline; deviation is not permitted.

Required: C code that complies to MISRA C shall comply with every required guideline; a formal deviation is required where this is not the case.

Advisory: these are recommendations that should be followed as far as is reasonably practical; formal deviation is not required, but non-compliances should be documented.

Every organization or project may choose to treat any required guideline as if it were mandatory and any advisory guideline as if it were required or mandatory. The adoption of MISRA Compliance:2016 [28] allows advisory guidelines to be downgraded to "Disapplied" when a check for compliance is considered to have no value, e.g., in the case of *adopted code*[7] that has not been developed so as to comply with the MISRA C guidelines. Of course, the decision to disapply a guideline should not be taken lightly: [28] prescribes the compilation of a *guideline recategorization plan* that must contain, among other things, the rationale for any decision to disapply a guideline.

Each MISRA C rule is marked as *decidable* or *undecidable* according to whether answering the question "Does this code comply?" can be done algorithmically. Rules are marked 'undecidable' whenever violations depend on run-time (dynamic) properties such as the value contained in a modifiable object

[7]Such as the standard library, device drivers supplied by the compiler vendor or the hardware manufacturer, middleware components, third party libraries, automatically generated code, legacy code, ...

Suppose, towards a contradiction, that we can complete the following C source code file `halt.c`:

```
#include <stdio.h>
int halts(const char *C_program) {
  /* Returns 1 if C_program is a valid C program source that terminates,
     0 otherwise. The function works perfectly for every input and always
     returns the correct result in finite time.*/
  /* ... */
}
int main(int argc, char **argv) {
  while (halts(argv[1])) {
    puts("This program will terminate");
  }
  puts("This program will not terminate");
  return 0;
}
```

We would then be able to compile it and execute the resulting executable as follows:

```
$ cc halt.c -o halt.exe
$ halt.exe halt.c
```

There are two possible outcomes:

1) `halt.exe` applied to `halt.c` prints

```
This program will terminate
This program will terminate
... [infinite repetitions]
```

so that in fact *it will not terminate!*

2) `halt.exe` applied to `halt.c` prints

```
This program will not terminate
```

but *it has just terminated!*

We reached a contradiction in both cases, hence the function `halts()` *cannot be written!*

Fig. 2. Termination: the father of all undecidable problems

or whether control reaches a particular point. Conversely, rules are marked 'decidable' whenever violations depend only on compile-time (static) properties, such as the types of the objects or the names and the scopes of identifiers. For rules marked 'undecidable', no algorithm can exist that allows emitting a message if and only if the rule is violated. Termination is a notoriously undecidable problem: see Figure 2 for a C-based proof of this fundamental theorem of computer science. Starting from the undecidability of termination, it is possible to prove that most interesting program properties (e.g., whether a program can result into a division by zero, a buffer overflow, a memory leak) are undecidable as well.

For rules marked 'decidable', although it is theoretically possible for a tool to emit a message if and only if the rule is violated, it may not be able to decide because of the limited computational resources. However, for a rule marked 'undecidable', any tool will have to deal with a *don't know* answer in addition to *yes* or *no*. In either case, if it is not practical (or even possible) for the tool to decide if the code is compliant with respect to a guideline at a particular program point, it can:

- suppress the *don't know* answer (i.e., possibly false negatives, no false positives);
- emit the *don't know* answer as a *yes* message (i.e., no false negatives, possibly false positives);
- a combination of the above (i.e., both possibly false negatives and possibly false positives);
- emit the *don't know* answer as a *caution* message (i.e., no false negatives, confined, possibly false positives).

The majority of the MISRA C guidelines are decidable,[8] and thus compliance can be checked by algorithms that:

- do not need nontrivial approximations of the value of program objects;
- do not need nontrivial control-flow information.

[8]On a total of 173 guidelines, 36 rules and 4 directives involve undecidable program properties [29].

Of course, these algorithms can still be very complex. For instance, the nature of the translation process of the C language, which includes a preprocessing phase, is a source of complications: the preprocessing phase must be tracked precisely, and compliance may depend on the source code before preprocessing, on the source code after preprocessing, or on the relationship between the source code before and after preprocessing.

MISRA C rules are also classified according to the amount of code that needs to be analyzed in order to detect all violations of the rule.

Single Translation Unit: all violations within a project can be detected by checking each translation unit independently.

System: identifying violations of a rule within a translation unit requires checking more than the translation unit in question, if not all the source code that constitutes the system.

MISRA C:2012 Amendment 1 [30], published in 2016, enhances MISRA C:2012 so as to extend its applicability to industries and applications where data-security is an issue. It includes 14 new guidelines (1 directive and 13 rules) to complete the coverage of ISO/IEC TS 17961:2013 [31], a.k.a. *C Secure Coding Rules*, a set of rules for secure coding in C.[9] Details of such complete coverage are provided in [33]. A similar document [34] shows that, with Amendment 1, coverage of *CERT C Coding Standard* is almost complete and that, consequently, MISRA C is today the language subset of choice for all industries developing embedded systems in C that are safety- and/or security-critical [35].

For the rest of this paper, all references to *MISRA C* will be for the latest published version MISRA C:2012 [14] including its Technical Corrigendum 1 [36] and Amendment 1 [30]: these will be consolidated into the forthcoming first revision of MISRA C:2012 [37]. It should be noted that both the MISRA C and MISRA C++ projects are active and constantly improving the guidelines and developing new works: for instance, the MISRA C Working Group is currently working at adding support for C11 [11] and, in response to community feedback, at further enhancing the guidance on undefined/unspecified behaviors [37].

IV. Selected MISRA C Guidelines

In this section we present a very small selection of MISRA C guidelines in order to convey the overall flavor of the coding standard.

Directive 4.6

> *typedefs* that indicate size and signedness should be used in place of the basic numerical types

Category Advisory
Applies to C90, C99

The directive advises against the use of the built-in numerical types like `char`, `short`, `unsigned`, `double` in favor of *typedefs* like, e.g., `int8_t`, `int16_t`, `uint32_t`, `float64_t` where the size and signedness must be explicit and match the actual size and signedness provided by the implementation. The rationale is that being aware of the signedness and size of the used data types is often crucial to ensure correctness of the algorithms. Moreover, in situations where the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object. Following this directive, portability is greatly improved, but adherence to this guideline alone does not guarantee portability (e.g., promotion may or may not take place depending on the size of `int`).

Rule 1.1

> The program shall contain no violations of the standard C syntax and *constraints*, and shall not exceed the implementation's translation limits

Category Required
Analysis Decidable, Single Translation Unit
Applies to C90, C99

In order to comply to this rule, the program:

- should only use features in the chosen version of the C Standard;
- should not exceed the implementation's translation limits;
- may use language extensions.

Moreover, except when using language extensions, a program must not:

- contain any violations of the language syntax;
- contain any violations of the constraints.

The reason is that language features that are outside the supported versions of C have not been considered when developing the MISRA guidelines. Furthermore, a conforming implementation does not need to generate a diagnostic when a translation limit is exceeded and an executable may be generated that does not work as expected. In addition, some non-conforming implementations fail to diagnose constraint violations and this, again, may result into undefined behavior.

Rule 3.2

> Line-splicing shall not be used in `//` comments

Category Required
Analysis Decidable, Single Translation Unit
Applies to C99

If a `//` commented line ends with a back-slash followed by a new-line, then the following line is part of the comment, and, if this was not intended, an important line of code maybe lost. The following example shows how a seemingly-innocuous path separator at the end of the comment may accidentally comment out the next line of code:

```
// see critical.* in c:\project\src\
critical_function();
```

[9] This technical specification has been slightly amended in 2016 [32].

Rule 9.1

> The value of an object with automatic storage duration shall not be read before it has been set

Category Mandatory
Analysis Undecidable, System
Applies to C90, C99

Note that array elements and structure members are considered as discrete objects and they must be (recursively) initialized. The rationale is that, according to the C Standard, objects with automatic storage duration are not automatically initialized and can therefore have indeterminate values.[10] Reading them while their value is indeterminate is undefined behavior.[11]

Rule 11.4

> A conversion should not be performed between a pointer to object and an integer type

Category Advisory
Analysis Decidable, Single Translation Unit
Applies to C90, C99

To avoid problems like misaligned pointers and truncation, integer to pointer conversion requires a deviation whenever it is necessary to address memory mapped registers or other hardware features. Conversion from pointer to integer which can also result in undefined behavior requires a deviation for the very rare cases, e.g., to perform bitwise manipulation on pointers, where this is needed. Note that the only integer types guaranteed to hold pointer values without truncation are the optional C99 types `intptr_t` and `uintptr_t`.

Consider the following example:

```
int main(int argc, char **argv) {
  uintptr_t v = (uintptr_t)argv;
  fdisplay("%d\n", memcmp(&v, &argv,
                          sizeof(uintptr_t)));
  return 0;
}
```

Even assuming that `uintptr_t` and `char **` have the same size, the result is still implementation-defined:[12] not all machines use the flat memory model and `uintptr_t` is always an integer.

The following example may trigger undefined behavior:[13]

```
int main(int argc, char **argv) {
  int16_t adr = (int16_t)argv;
  return 0;
}
```

[10]In contrast, in conforming implementations objects with static storage duration are automatically initialized to zero unless initialized explicitly.

[11]C99 Undefined 10: *the value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.8, 6.8).*

[12]C99 Implementation-defined J.3.7 1: *the result of converting a pointer to an integer or vice versa (6.3.2.3).*

[13]C99 Undefined 21: *conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).*

Another undefined behavior is caused by the following:[14]

```
int main(void) {
  float64_t a = 1.0;
  intptr_t i = (intptr_t)&a;
  int32_t *b = (int32_t*)i;
  *b = 0;
  fdisplay("%f\n", a);
  return 0;
}
```

Rule 13.2

> The value of an expression and its *persistent side effects* shall be the same under all permitted evaluation orders

Category Required
Analysis Undecidable, System
Applies to C90, C99

Between two sequence points the evaluation order is unspecified, so the following situations can lead to undefined behavior:[15]

- modifying an object more than once;
- modifying and reading an object, unless reading is necessary to store in the object;
- reading a `volatile` object more than once;
- modifying a `volatile` object more than once.

Note that the logical *and* and *or* operators, the conditional operator, and the comma operator have well defined operand evaluation orders.

In the following example, the function `main` will return `1` or `-1` depending on the evaluation order:[16]

```
static int32_t next(void) {
  static int32_t counter = 0;
  ++counter;
  return counter;
}

int main(void) {
  return next() - next();
}
```

In the next example, `TCNT1` and `TCNT2` are memory mapped hardware registers, and it is unspecified which side effect is triggered first:[17]

[14]C99 Undefined 34: *an object has its stored value accessed other than by an lvalue of an allowable type (6.5).*

[15]C99 Undefined 32: *between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored (6.5).*

[16]C99 Unspecified 15: *the order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, `&&`, `||`, `?:`, and comma operators (6.5).*

[17]C99 Unspecified 16: *the order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).*

```
static volatile uint16_t TCNT1;
static volatile uint16_t TCNT2;

static int32_t f(uint16_t tc1, uint16_t tc2);

int main(void) {
  return f(TCNT1, TCNT2);
}
```

V. Understanding MISRA C

In this section we highlight important aspects of MISRA C that are often misunderstood.

A. MISRA C Is Part of a Process

MISRA C has been designed to be used in the framework of a *documented software development process*. The process must ensure, e.g.:

- that software requirements are complete, unambiguous and correct;
- that design specifications reaching the coding phase are correct, consistent with the requirements and do not contain any other functionality;
- that object modules produced by the compiler behave as specified in the designs;
- that object modules have been tested, individually and together, to identify and eliminate errors.

Clearly, MISRA C should be used before code reaches the review and unit testing phases, for otherwise a lot of rework and retesting has to be expected.

Full requirements for safety-related software development processes are outside the scope of MISRA C and they are left to the applicable industrial standards, such as IEC 61508, ISO 26262, RTCA DO-178C, CENELEC EN 50128 and IEC 62304. MISRA C does require some process activities. In particular, in order to use MISRA C, it is necessary to develop and document:

- a *compliance matrix*, showing how compliance with the MISRA C guidelines is checked;
- a *deviation process* by which justifiable non-compliances can be authorized and recorded.

The software development process should also document the steps that are taken to demonstrate that run-time errors have been avoided, e.g., to make sure that:

- the execution environment provides sufficient resources, especially processing time/power and stack space;
- run-time errors, such as arithmetic overflows, are absent from (areas of) the program (e.g., by virtue of static and/or run-time checks).

There are activities that MISRA C endorses without providing specific prescriptions. For instance, it is well known that a consistent style assists programmers in understanding their own code and the code written by others. Thus MISRA C encourages the adoption of programming style guidelines, but it has always left this matter to individual organizations: "In addition to adopting the subset, an organisation should also have an in-house style guide. [...] However the enforcement of the style guide is outside the scope of this document" [21, Section 4.2.2] (see also [14, Section 5.2.2]). Similarly, many software process standards recommend metrics as a practical means to identify code that may require special attention (e.g., additional review and/or testing). The nature of the metrics being collected and how they are used is left to individual organizations: MISRA C does not make recommendations in that area, even though the *HIS metrics* can be considered a *de facto* standard [38].

B. MISRA C: Error Prevention, Not Bug Finding

As said earlier, MISRA C cannot be separated from the process of documented software development it is part of. In particular, the use of MISRA C in its proper context is part of an *error prevention* strategy which has little in common with *bug finding*, i.e., the application of automatic techniques for the detection of instances of some software errors. This point is so rarely understood that it deserves proper explanation.

To start with, the violation of a guideline is not necessarily a software error. For instance, let us consider Rule 11.4, which advises against converting integers to object pointers and vice-versa. There is nothing intrinsically wrong about converting an integer constant to a pointer when it is necessary to address memory mapped registers or other hardware features. However, such conversions are implementation-defined and have undefined behaviors (due to possible truncation and the formation of invalid and/or misaligned pointers), so that they are best avoided everywhere apart from the very specific instances where they are both required and safe. This is why the deviation process is an essential part of MISRA C: the point of a guideline is not "You should not do that" but "This is dangerous: you may only do that if

1) it is needed,
2) it is safe, and
3) a peer can easily and quickly be convinced of both 1) and 2)."

One useful way to think about MISRA C and the processes around it is to consider them as an effective way of conducting a *guided peer review* to rule out most C language traps and pitfalls.[18]

The attitude with respect to incompleteness between the typical audience of bug finders and the typical audience of MISRA C is entirely different. Bug finders are usually tolerant about false negatives and intolerant about false positives: for instance, by following the development of *Clang Static Analyzer*[19] it can be seen that everything is done to avoid false positives with very little regard to avoid false negatives. This is not the right mindset for checking compliance with respect to MISRA C: false positives are a nuisance and should be reduced and/or confined as much a possible, but using algorithms with false negatives implies that those in charge

[18] We are indebted to Clayton Weimer for this observation.

[19] https://clang-analyzer.llvm.org/, last accessed on January 18th, 2019.

of ensuring compliance will have to use other methods. So, compliance to MISRA C is not bug finding and, of course, finding some, many or even all causes of run-time errors does not imply compliance to MISRA C.

C. MISRA C: Readability, Explainability, Code Reviews

Another aspect that places MISRA C in a different camp from bug finding has to do with the importance MISRA C assigns to reviews: code reviews, reviews of the code against design documents, reviews of the latter against requirements. Note that these are explicitly needed by Directive 3.1 for which all the code must be formally traceable to the design documents. More generally, the need for code readability and explainability is clearly expressed in the rationale of many MISRA C guidelines.

This fact has some counter intuitive consequences on the use of static analysis, which is of course crucial both for bug finding and for the (partial) automation of MISRA C compliance checking. Consider Rule 9.1, whereby the value of an automatic object must not be read before it has been set, since otherwise we have undefined behavior. For bug finding, the smarter the static analysis algorithm the better. Use of the same smart algorithm for ensuring compliance with respect to Rule 9.1 risks obeying the letter of MISRA C but not its spirit.[20] Suppose on the specific program our smart algorithm ensures Rule 9.1 is never violated: we have thus ruled out one source of undefined behavior, which is good. However, the programmer, other programmers, code reviewers, quality assurance people, one month from now or six months from now may have to:

1) ensure that the automatic objects that are the subject of the rule are indeed initialized with the correct value;
2) confirm that the outcome of the tool is indeed correct.

If this takes more than 30 seconds or a minute per object, this is not good: the smart static analysis algorithm can track initializations and uses even when they are scattered across, say, switch cases nested into complex loops; a human cannot. So, ensuring compliance with respect to Rule 9.1 with deep semantic analysis is counterproductive to the final goal of the process of which MISRA C is part. For that purpose it is much better to use a decidable approximation of Rule 9.1 such as a suitable generalization of the *Definite Assignment* algorithm employed by Java compilers [39, Chapter 16].

VI. Successful Adoption of MISRA C

The highest payoff from the adoption of MISRA C is achieved when:

1) it is adopted at the very beginning of a project;
2) it is systematically enforced with the help of a high-quality tool;
3) personnel has been properly trained.

Unfortunately, point 1 is often out of reach, that is, the project has already started. It must be understood that imposing MISRA C on an existing code base with a proven track record may be counterproductive if not done properly. In general, applying MISRA C to existing code requires significant expertise (i.e., more training and/or access to qualified consulting services) and tools of even higher quality (i.e., providing powerful deviation mechanisms, baselining, ...).

[20] There are many ways to do that.

A. The Importance of Tools

It is entirely possible to manually verify code for compliance to MISRA C but, of course, the cost of doing that is huge. Hence, tools are highly recommended to semiautomate the check for compliance. Manual activities remain, such as:

- initial tool configuration;
- tool configuration for deviation.

A good static analysis tool will do a thorough job of automatically verifying compliance for most MISRA C guidelines, but not all of them:

- undecidable rules;
- directives;
- limitations of the tool (complexity-precision trade-off, incomplete handling of extensions, ...);
- issues with the project being analyzed (unavailability of part of the source code, extensive use of language extensions, ...).

In any case, the remaining manual activities, and peer review, are greatly facilitated by the level of partial compliance that can be achieved by using a good tool.

One aspect that should not be underestimated is the proper configuration of the static analysis tool. Depending on the tool, this task can be both challenging and error-prone. There are two main reasons for that: the first one is that C, rather than *a* language, is a *large family* of languages. In C99, there are 112 implementation-defined behaviors (i.-d.b.). As each i.-d.b. can be defined in 2 or more ways, there are more than $2^{112} \approx 5 \times 10^{33}$ possible languages. Actually, choosing integer and floating-types in the set of bit widths $\{8, 16, 32, 64\}$ brings us to more than 10^{36} possible languages (dialects of C). Generally speaking, a given compiler can implement, via options, several such dialects of C. For an extreme case, GCC/x86_64 implements, via options, hundreds of such dialects. As a consequence, the tool must adapt to the particular dialect implemented by that compiler with that set of options (which may be different for different translation units). Moreover, changing even one compilation option may change the language dialect and thus require a change in the tool configuration. We have seen many projects whose MISRA C compliance was completely led astray due to misconfiguration of the actual language dialect implemented by the toolchain in the static analysis tool. This often occurred because a change in the build process was not reflected by a corresponding change in the tool's configurations.

Another aspect that might require careful configuration of the tool has to do with *predefined macros*. Some of these are influenced by the compiler options, which may be given on the command line, in environment variables, or in configuration files. Failing to capture the predefined macros correctly may

result, due to conditional compilation preprocessing directives, into the wrong code being analyzed. Even when the right code is analyzed, tool misconfiguration has been known to lead to errors concerning where/how header files are searched, intrinsics and other extensions, the linker, ...

There are three possibilities to deal with the complexity of tool configuration:

1) The tool is integrated in the compiler/linker. At first sight, this would seem to be the ideal situation, since all the required information is in the compiler but, generally speaking:
   - such tools are usually not very good (limited audience, limited investments, limited testing and, most importantly, the static analysis algorithms used by compilers for optimization purposes are not adequate for verification;[21]
   - changing the compiler often implies one has to start from scratch.
2) Manual configuration of the static analysis tool:
   - this might be very error-prone;
   - any change to the compiler options must be matched by a corresponding manual change to the configuration.
3) The tool is designed in such a way so as to automatically extract the required information from the toolchain.

B. The Importance of Training

Staff competence is a crucial requirement in order to carry out the activities that allow describing a project as "MISRA Compliant" [28]. Without a proper understanding of C pitfalls and of the reasons behind each of the MISRA guidelines, developers often:

- perceive the adoption of the guidelines as a useless burden;
- misunderstand messages output by the tool and do not know what should be done;
- are unable to recognize false positives;
- change the code by trial-and-error in an attempt to silence the tools.

Lack of training always implies significant time loss. Moreover, without adequate training, more often than one might think, after code changes to force MISRA C compliance there is a strict decrease in the quality of the code produced. In contrast, in our experience, proper formal training of personnel involved in the development and quality assessment, enables a smooth and successful adoption of MISRA C into an organization. This kind of training significantly strengthens the skills and competences of teams involved in the design, development and verification of critical embedded software systems written in C.

VII. Conclusion

In this paper, having explained some of the advantages and disadvantages of using the C language for embedded systems and how the uncontrolled use of C conflicts with both safety and security requirements, we described the background, motivation and history of the MISRA project. We have explained how the MISRA C guidelines define a standardized structured subset of the C language, making it easier, for code that follows these guidelines (possibly with well-documented deviations), to verify that important and necessary safety and security properties hold.

We have looked at the different kinds of the MISRA C guidelines, distinguishing between those that can be automatically verified from the code syntax, those that need information beyond that contained in the source code, and those for which the question as to whether the code is compliant is algorithmically undecidable. We have presented a small selection of MISRA C guidelines along with a terse explanation of their rationale.

We have highlighted some points that often cause misunderstandings of the key role MISRA C plays in the development of safety- and security-critical embedded software. Particularly, the fundamental differences between so-called *bug finding* and the application of MISRA C in the context of the error prevention strategy it is part of. Finally, we have illustrated some important points for the successful adoption of MISRA C in an organization, that is, the necessity of high-quality automatic tools for the checking or partial checking of compliance, and the essential role of proper formal training.

Acknowledgments

For the notes on the history of MISRA and MISRA C we are indebted to Andrew Banks (LDRA, current Chairman of the MISRA C Working Group) and David Ward (HORIBA MIRA, current Chairman of the MISRA Project). We are grateful to the following people who provided useful comments and advice: Fulvio Baccaglini (PRQA — a Perforce Company, MISRA C Working Group), Dave Banham (Rolls-Royce plc, MISRA C Working Group), Daniel Kästner (AbsInt, MISRA C Working Group), Thomas Schunior Plum (Plum Hall, WG14), Chris Tapp (LDRA, Keylevel Consultants, MISRA C Working Group, current Chairman of the MISRA C++ Working Group), David Ward (ditto). We are also grateful to the following BUGSENG collaborators: Paolo Bolzoni, for some example ideas; Anna Camerini for the composition of Figure 1.

[21] See https://tinyurl.com/y998etdf .

References

[1] D. M. Ritchie, "The development of the C language," *SIGPLAN Notices*, vol. 28, no. 3, pp. 201–208, Mar. 1993.

[2] *Embedded Systems Safety & Security Survey*, Barr Group, Germantown, MD, USA, Feb. 2018, available at http://www.barrgroup.com/.

[3] *2011 Embedded Engineer Survey*, VDC Research, Natick, MA, USA, Aug. 2011.

[4] V. Nélis, P. M. Yomsi, and L. M. Pinho, "The variability of application execution times on a multi-core platform," in *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OASICS, M. Schoeberl, Ed., vol. 55. Toulouse, France: Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2016, pp. 6:1–6:11.

[5] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Proceedings of the Ninth European Dependable Computing Conference (EDCC 2012)*, C. Constantinescu and M. P. Correia, Eds. Sibiu, Romania: IEEE Computer Society, 2012, pp. 132–143.

[6] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*. Madrid, Spain: IEEE Computer Society, 2014, pp. 109–118.

[7] ISO/IEC, *ISO/IEC 9899:1990: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1990.

[8] ——, *ISO/IEC 9899:1990/AMD 1:1995: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1995.

[9] ——, *ISO/IEC 9899:1999: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1999.

[10] ——, *ISO/IEC 9899:1999/Cor 3:2007: Programming Languages — C*, Technical Corrigendum 3 ed. Geneva, Switzerland: ISO/IEC, 2007.

[11] ——, *ISO/IEC 9899:2011: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2011.

[12] ——, *ISO/IEC 9899:2018: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2018.

[13] T. Winters, "C++ stability, velocity, and deployment plans [R2]," ISO/IEC JTC1/SC22/WG21, Doc. no. P0684R2, Feb. 2018, available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0684r2.pdf.

[14] MISRA, *MISRA C:2012 — Guidelines for the use of the C language in critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Mar. 2013.

[15] *Intel 64 and IA-32 Architectures Software Developer's Manual — Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*, Intel Corporation, 2018.

[16] L. Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. New York, NY, USA: McGraw-Hill, Inc., 1995.

[17] *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*, Version 2.0 ed., U.S. Department Of Health and Human Services; Food and Drug Administration; Center for Devices and Radiological Health; Center for Biologics Evaluation and Research, Jan. 2002, available at http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085281.htm.

[18] The Motor Industry Research Association, *Development Guidelines For Vehicle Based Software*. Nuneaton, Warwickshire CV10 0TU, UK: The Motor Industry Research Association, Nov. 1994.

[19] ISO, *ISO 26262:2011: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Nov. 2011.

[20] Motor Industry Software Reliability Association, *MISRA-C:1998 — Guidelines for the use of the C language in vehicle based sofware*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Jul. 1998.

[21] Motor Industry Software Reliability Association, *MISRA-C:2004 — Guidelines for the use of the C language in critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Oct. 2004.

[22] Motor Industry Software Reliability Association, *MISRA C++:2008 — Guidelines for the use of the C++ language in critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Jun. 2008.

[23] VV. AA., "JSF Air vehicle C++ coding standards for the system development and demonstration program," Lockheed Martin Corporation, Document 2RDU00001, Rev C, Dec. 2005.

[24] VV. AA., "JPL institutional coding standard for the C programming language," Jet Propulsion Laboratory, California Institute of Technology, Tech. Rep. JPL DOCID D-60411, Mar. 2009.

[25] M. Barr, *Embedded C Coding Standard*. Germantown, MD, USA: Barr Group, 2013.

[26] CERT, *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*, 2016th ed. Software Engineering, Carnegie Mellon University, 2016.

[27] Software Engineering Center, *Embedded System Development Coding Reference: C Language Edition*. Information-Technology Promotion Agency, Japan, Jul. 2014, version 2.0.

[28] MISRA, *MISRA Compliance:2016 — Achieving compliance with MISRA Coding Guidelines*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Apr. 2016.

[29] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software," in *Static Analysis: Proceedings of the 25th International Symposium (SAS 2018)*, ser. Lecture Notes in Computer Science, A. Podelski, Ed., vol. 11002. Freiburg, Germany: Springer International Publishing, 2018, pp. 5–23.

[30] MISRA, *MISRA C:2012 Amendment 1 — Additional security guidelines for MISRA C:2012*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Apr. 2016.

[31] ISO/IEC, *ISO/IEC TS 17961:2013, Information technology — Programming languages, their environments & system software interfaces — C Secure Coding Rules*. Geneva, Switzerland: ISO/IEC, Nov. 2013.

[32] ——, *ISO/IEC TS 17961:2016, Information technology — Programming languages, their environments & system software interfaces — C Secure Coding Rules*. Geneva, Switzerland: ISO/IEC, Aug. 2016.

[33] MISRA, *MISRA C:2012 Addendum 2 — Coverage of MISRA C:2012 (including Amendment 1) against ISO/IEC TS 17961:2013 "C Secure"*, 2nd ed. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Jan. 2018.

[34] ——, *MISRA C:2012 Addendum 3 — Coverage of MISRA C:2012 (including Amendment 1) against CERT C 2016 Edition*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Jan. 2018.

[35] R. Bagnara, "MISRA C, for security's sake!" in *Informal proceedings of the 14th Workshop on Automotive Software & Systems*, G. Lami, Ed., Milan, Italy, 2016, available at http://www.automotive-spin.it/. Also published as Report `arXiv:1705.03517 [cs.SE]`, available at http://arxiv.org/.

[36] MISRA, *MISRA C:2012 Technical Corrigendum 1 — Technical clarification of MISRA C:2012*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Jun. 2017.

[37] A. Banks, "MISRA C — recent developments and a road map to the future," Presentation slides available at http://www.his-2018.co.uk/session/misra-c-updates-2016, Nov. 2016, presented at the *High Integrity Software Conference 2016*, Bristol, UK, November 1, 2016.

[38] H. Kuder, "HIS source code metrics," Herstellerinitiative Software, Tech. Rep. HIS-SC-Metriken.1.3.1-e, Apr. 2008, version 1.3.1.

[39] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification: Java SE 8 Edition*, 5th ed., ser. Java Series. Upper Saddle River, NJ, USA: Addison-Wesley, 2014.