Automatic Checking of Coding Rules

Paolo Bolzoni
Relatore: Prof. Roberto Bagnara
Correlatore: Prof. Enea Zaffanella

2009.04.22

Abstract

Experience shows that programming is difficult and error-prone. This dissertation is about a way to reduce the number of mistakes: to write code following some well-thought coding rules and to detect accidental violations, as far as it is possible, routinely and automatically. The scope of the rules is to help developers to avoid problems in the whole development process: writing the code, compiling and debugging it. When writing the code, rules can impose clear constructs with an easy-to-understand semantics. When compiling, rules can avoid language constructs whose compilation is known to be problematic. When debugging, rules can help to avoid obscure code. The use of coding rules has been introduced in the embedded system world where updating the code is difficult and expensive and where failures are dangerous. After the embedded system has been built, updating the code might mean reprogramming electronic cards in many appliances that do not have any standard connection, or even changing electronic circuits. A failure in an embedded system might cause erratic behaviour of machinery putting people in danger. In embedded systems the programming language of choice is C. Unfortunately the C language is difficult, so that many rules focus on the most common pitfalls. We show a subdivision of the rules by the technologies needed to automatically verify them. We always need a preprocessor, the subdivision is by the other technologies needed: a simple line inspector program; a parser; a static semantic analyzer or a dynamic semantic analyzer. Some of the rules require more than a single technology; for a few rules we think that no automatic verification is possible. We subdivide also for wideness, that is depending on how big is the chunk of software that has to be examined to verify the rule holds. The widenesses we describe are, from small to large: a fixed number of characters, a token, a global declaration, a compilation unit or the whole software project. For each rule we describe the risks of ignoring it and the possible disadvantages of following it. For some of the rules we present a simple sketch of a possible implementation of an automatic checker capable of detecting violations. Depending on the kind of rule, we present suitable possible implementation approaches: inspecting the token stream, visiting the abstract syntax tree or hijacking the compilers' warning.

Contents

Abstract **iii**

Contents **v**

1 Coding Rules **1**
- 1.1 Source . . . 3
- 1.2 Reason . . . 4
- 1.3 Applicability . . . 11
- 1.4 Dissertation Plan . . . 11
- 1.5 Acknowledgement . . . 12

2 Checking Rules **13**
- 2.1 Manual Checking . . . 13
- 2.2 Automatic checking . . . 13

3 Kind of Rules **15**
- 3.1 Trivial Rules . . . 15
- 3.2 Type Enforceable Rules . . . 16
- 3.3 Syntactic Rules . . . 17
- 3.4 Structural Rules . . . 18
- 3.5 Dynamic Rules . . . 19
- 3.6 Complex Rules . . . 21
- 3.7 Difficult Rules . . . 21

4 Wideness **23**
- 4.1 Character . . . 24
- 4.2 Token . . . 24
- 4.3 Global Declaration . . . 24
- 4.4 Compilation Unit . . . 25
- 4.5 Whole Project . . . 25

5 Implementing Checkers **27**
5.1 Token Stream 28
5.2 Compiler Warnings 35
5.3 Abstract Syntax Tree Rules 36

Conclusion **45**
Related Work 46
Future Work 46

Bibliography **47**

Chapter 1

Coding Rules

> There are only two kinds of languages: the ones people complain about and the ones nobody uses.
>
> Bjarne Stroustrup

There shall be code and people who write code. It is true that the tools in the hands of developers will become more and more powerful, the languages will become more and more expressive and easy to use; languages will change, tools will change, we have already seen that: the time where machine code or Assembly were the only choices are long gone, now we have compilers and various helping tools. We have languages that are very different from each others, and directly support and encourage very different programming styles, the choice is not only the traditional imperative approach.

Languages change, but there will be always code. The code is the way we tell our computers to do what we need. Thinking that code will disappear is like thinking that someday mathematics will not need being formal. It is like thinking that one day we will have machines that do what we need and not what we say. Even humans beings, with all their intelligence and common sense, only from the feelings of others often fail to make the correct tasks. There is nothing that let us think machines will ever be able.

Writing code is a complex activity: learning the basics of a serious programming language is often quite easy. Writing code that does what it is supposed to do and only what it is supposed to do is very hard. There are many components who interact with each other, many implicit conditions that are easy to forget. Often even the concept of "supposed to do" is unclear.

To simplify this difficult task, almost all realities where writing code is necessary make a set of "good coding rules." Rules do not ensure good quality code, neither they ensure the lack of bugs. But rules have been (or they should have been, at least) decided from real-world experience and written in order to help developers to avoid well known problems. It is like: "we do not solve all your coding problems, but we help you to avoid mistakes we already seen many times."

The more the language is used and the more is liberal, the more the rules tend to be strict. This is because experience found more pitfalls in those languages than in unused or strict languages. As an example, regulating the maximum length of a line of code in Fortran77 would be meaningless. The compiler ensures that no lines can be longer than 78 characters. It would also be meaningless to write rules regulating the language Funge[1] because no one will ever use it in production code.

The C programming language is very liberal. Here is an extreme example:

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_
,main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&
t==2?_<13?main(2,_+1,"%s %d%d\n"):9:16:t<0?t<-72?main(_,t,"@n\
'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{\
n+,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q\
#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')){}#}w'){}{nl]'/+#\
n';d}rw'i;# ){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{n\
l]'/w#q#n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}\
{nlwb!/*de}'c ;;{nl'-{}rw]'/+,}##'*} #nc,',#nw]'/+kd'+e}+;#'r\
dq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/"):t<-50?_==*a?putchar
(31[ a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1):0<t?main(2,2,
"%s"):*a=='/'||main(0,main(-61,*a,"!ek;dc i@bK'(q)-[w]*%n+r3#\
l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

This piece of code is indeed a correct C program with meaningful output: "The Twelve Days of Christmas," a cumulative joyful song that lists the gifts received by the singer.

Now imagine being a boss asking an employee to implement a program with "The Twelve Days of Christmas" as output. The employee delivers that code. The task is accomplished correctly, but the code is very difficult to read and unusable for any other purpose. Correctness is important, but the readability and reusability are almost as important. It is unthinkable to use that code as a starting point to output another cumulative song, it is even impossible to change a single gift of the list. Even for the smallest modification the code has to be rewritten from scratch. This is something developers try to avoid.

[1] `http://quadium.net/funge/spec98.html`

The site `http://www.langpop.com/` presents the popularity of programming languages. The results are calculated using different internet search bot. The C language is always first or second in terms of number of developers and size of existing code. In short, C is very common and very liberal. Hence, it is a perfect candidate for strict and hopefully useful coding rules.

1.1 Source

The C language is widely used, just think about programs included in any Linux distributions. Its low level nature allowed the development of compilers for any platform, including specialized processors of embedded systems.

This means the C language is used in systems like flight recorders, car control units and all kind of home appliances. While a segmentation fault in a word processor can at most annoy some users, a failure in a car control unit may cause catastrophes, including the loss of human lives.

Sources

We collected rules from "Guidelines for the use of the C language in critical systems" (MISRA from now on) [7], a specifically thought set of rules for programming with C in embedded systems in the automotive industry. Processors in those contexts have little support from compiler vendors so, aside from Assembly itself, C is almost the only choice.

"The Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program" (JSF from now on) [4] is a set of rules for programming in embedded system for air vehicles, covering the C++ language. The intent of this document was to extend MISRA from C to C++, but nowadays there is also a MISRA official document for C++.

"The High Integrity C++ Coding Standard Manual" (HICPP from now on) [8] instead is different because it does not focus on programming inside embedded system, but gives C++ advice of broader domain.

We worked on the C programming language, not C++. The C language is *almost* a subset of C++. But we have to be careful, because in the "almost" word reside many subtle differences. For example, in C an identifier is reserved if it starts with underscore and its second character is an other underscore or an uppercase letter, if it just start with an underscore it is reserved in tag and normal name spaces (the C99 standard [2, Section 7.1.3]). Instead in C++ an identifier is reserved if begins with an underscore followed by an uppercase letter or contains two adjacent underscores. All identifiers starting with an underscore are reserved in the global namespace (the C++03 standard [3, section 17.4.3.1.2]). So using the label '`_hello:`' or calling a variable '`a__b`' is legitimate in C, but not in C++. C++ adds many keywords, so the function declaration '`int and();`' will be valid in C, but not in C++. All those differences thrust us to alter some rules or remove them completely.

The rules have the final objective of ease the complexity of writing code, in particular rules try to reduce bad programming habits, reduce confusion to the programmers or reduce triggering of compiler bugs.

Rules Strictness

The coding sources divide the rules by strictness: it is not an order of importance; it is an order meant to underline the risks of ignoring the rule. In fact not following a stricter rule means risking more.

MISRA divides its rules in "advisory" and "required." A required rule must be followed. In order to ignore a required rule, a written formal request and a corresponding agreement from the customer of software are required. The request can be made project-wide or for a specific situation: project-wide deviations are usually decided before starting the real implementation work, specific deviations are usually decided when the need arises. To ignore an advisory rule it is not required any particular request, but it does not mean that the rule can safely be ignored. It means the rule should be followed as far as it is practical.

JSF uses three level of strictness for its rules: "should," "will" and "shall." The procedure to follow for being authorized to ignore a rule changes between strictness levels. The procedure to ignore a "should" rule is simpler than the procedure to ignore a "will" or a "shall" rule. Curiously the "shall not" rules cannot be ignored in any case. "Should" rules are advisory rules: they express strongly recommended ways to code in C++. "Will" rules must be followed, but it is not needed a formal proof that the project adheres to them. "Shall" rules, instead, must be followed and it a formal proof of their correctness is needed. The formal proof can also be a manual verification.

HICPP describes the levels as "guideline" and "rule:" adhering is expected and needed respectively. HICPP describes no formal ways to obtain authorizations to ignore rules. The rule 2.1 simply states "Thoroughly document in the code any deviation from a standard rule." MISRA and JSF need formal authorizations, HICPP does not. The reason is that the addressees of the rules are different, HICPP rules are meant for everyone while JSF and MISRA rule are meant for developers of critical embedded system.

1.2 Reason

Bad Programming Styles

The C language is free-form, the blocks are decided only using curly brackets, those brackets can be written in two different ways (three in C99). Using the escaped newlines the programmer can start a newline everywhere, using the operator short circuit he may often avoid writing an explicit control flow construct.

The two pieces of C code below are equivalent:

```
f(), g(), h();
main() {
  (!f() || (g(), 0)) && h(); ??>
```

and

```
int f(void);
int g(void);
int h(void);

int main()
{
  if (f() != 0)
    g();
  else
    h();
}
```

How quickly did you understand that they are equivalent?[2] Probably you needed more than five seconds to understand the first and less than one second to understand the second. Once again, this is a somewhat extreme example, but many rules focus on writing more readable code for fellows developers, not the compiler.

Those rules cover coherent indentation styles, variable and function naming, but also subtler problems, like side effects in assertions. Strict rules about style are mainly due to the fact that the C programming language always assumes the programmer knows what she is doing. The effect is that many small errors might pass unnoticed at compile time and bring disasters at runtime.

```
const char* no_answer = "";
const char* correct_answer = "blue";
char answer[128];

get_answer_from_user(answer, 128);

if (answer == no_answer) {
  printf("You did not even try!\n");
}
else if (answer == correct_answer) {
  printf("Correct!\n");
}
else {
```

[2]Given the same body to `f`, `g` and `h` '`gcc -O 2 -S`' returns exactly the same Assembly for both files.

```
    printf("Wrong, I am sorry.\n");
}
```

The above piece of code contains errors, but to the eyes of a beginner developer they might pass unseen. The identifier `answer` is the array name, using it in the equality operators returns the array first element *address* as a char pointer. Instead `"blue"` and `""` are string literals, the developer assigned the first character *address* to the `const char` pointers `correct_answer` and `no_answer`.

Those addresses will never be equal. The effect is that, no matter what the user answers the program will always returns "Wrong, I am sorry." Yet, the C compiler is perfectly happy, because comparing two pointers of the same unqualified type is always legitimate, and usually it does not even warn about the comparison.

Implementation-Defined Behaviour

Many aspects of the C programming language are not completely defined and compilers' implementors are free to decide what to do. Code that uses those aspects and is correct on a given platform might become erroneous on a different platform, with different compiler or even a different version of the same compiler.

An example is the value returned in case of domain error by functions of `<math.h>`: an implementation might decide to return a peculiar value, a programmer might check that value instead of the error status as the standard says. The program will behave correctly, alas when using a different compiler the "domain error" return value might be different and the program will be erroneous.

Implementation defined code is seldom needed; so the rules do not ban it, but strictly regulate its use because it needs to be checked for correctness every time a different compiler is used.

The old compiler GCC 1.21 showed well the point, there was an easter egg: trying to compile a certain implementation defined behaviour code GCC created an executable file which started emacs with a simulation of the Tower of Hanoi. GCC 1.21's behaviour sounds lunatic, but it was completely standard compliant.

An example of needed implementation defined feature, very common when working with embedded systems, is the name of the function that starts the program in a freestanding environment.[3]

Compilers, especially commercial ones, often extend the language. This might be done for the genuine will to provide a better language, but also for binding the customers to the compiler vendor; anyhow language extensions

[3] *Freestanding environment* means no operating system support, and so no one that calls `main`.

are considered as implementation defined features, they should be avoided if not strictly needed.

An example of a reasonable language extension, provided by GCC compiler suite, that might be banned by rules because not strictly needed is the *aligned* attribute. Some types cannot be allocated at every memory offset, but only to multiples of a certain value. In many machines, for example, `int` can be stored only at addresses whose value is multiple of 4. Failing to respect the alignment requirement of a type will cause problems, from simple slowdowns to bus errors that crash the program.

Imagine your program has an important slowdown because it calls `malloc` and `free` many times. Your functions often use temporary small pieces of memory. In order to avoid this problem you want to allocate a block of memory every function might use. The idea sounds simple: every function can use this memory, the functions will not use `malloc` if the memory they need is temporary and smaller than the block, functions will not assume the block contains anything meaningful when they start and after they called any other function. Every thread will have its own block.

Just using an array of `unsigned char`s might not work: since `unsigned char` does not have alignment requirements the compiler might decide to place the array everywhere in memory. If the array is not placed at an address that is correctly aligned for every data type of the language the program might unexpectedly crash when a function tries to use that piece of memory.

The GCC's *aligned* attribute ensures that the memory allocated will be correctly aligned for every type. The standard compliant solution is using an `union` of all types that can have different alignments; the `union` type ensures that its address is correctly aligned for all its fields.

It means that those two pieces of code both give an usable block of memory of size 255 with the maximum alignment, starting at the address obtainable using the identifier `block`.[4]

```
/* multi purpose memory, GCC version */
#define BLOCK_SIZE 0xFF
unsigned char block[BLOCK_SIZE] __attribute__((aligned));
```

```
/* multi purpose memory, standard version */
union aligned {   /* this type has the maximum alignment */
  short a;        /* integers */
```

[4]The list of all needed types inside the union comes from taking a representative of every type and removing redundant ones. Unsigned integers have the same alignment requirements of the relative signed types, `_Bool` has the same alignment of another integer type; `_Complex` and arrays have the alignment requirements of the underling type; structures have alignment requirements of their first element type; pointers to void have the largest pointer-wise alignment requirement; all pointers to functions have the same requirement.

```
  int b;
  long c;
  long long d;
  float e;        /* floats */
  double f;
  long double g;
  void* h;        /* pointers */
  void(*i)(void); /* pointers to functions */
};

#define BLOCK_SIZE 0xFF
union aligned rawmem[(BLOCK_SIZE - 1) / sizeof(union aligned) + 1];
unsigned char* block = (unsigned char*)rawmem;
```

The standard version works fine, but it is harder to read and it might waste some memory. It is harder to read because the size of the array must be divided by the size of the union round up and you need to understand why the type indeed have maximum alignment. Since the size of union is at least (usually exactly though) the size of long double the rounding might be slightly expensive. Using this technique to allocate a block of memory large enough for storing 7 doubles in a x86_64 machine uses 64 bytes for the standard version and only 56 for the GCC version.

Unspecified and Undefined Behaviour

A code has unspecified behaviour if it may act in two or more well defined ways.

```
unsigned g(unsigned, unsigned);

unsigned f(unsigned a)
{
  if (a == 0)
    return 0xdeadbeef;
  else
    return a;
}

int main()
{
  unsigned value = 0;
  g(f(value), ++value);
}
```

`main` will call `g(1, 1)` or `g(0xdeadbeef, 1)`, but which one? You cannot know. Yet, you can be sure that it is one of the two.

In contrast, when code has undefined behaviour it is completely unpredictable. As far the language is concerned in front a situation defined as having "undefined behaviour" it might happen anything, or nothing at all.

A well-known example of undefined behaviour is the signed integer overflow; in fact in many implementations the overflow of signed integer "wraps" cleanly, but it is only a fortunate effect of how machine stores the integers in memory as this is not granted at all by the standard. At most something can be granted by compilers as language extension. But if it is not and a developer assumed the behaviour seeing that the program worked fine. Then he has been just lucky.

Code that relies on unspecified or undefined behaviour is unpredictable, so it cannot be accepted in safety critical systems. All of the sources' for coding rules completely ban code having this behaviour.

Compiler Errors

The compiler itself might contains bugs, so even if the standard implies the program is correct the generated code could not. This kind of error is the most rare, but also the most difficult to detect because the translation phase from source code to machine code is usually seen as a black box by the programmers. In order to reduce this kind of problems some rules ban or regulate those language constructs whose compilation is known as problematic.

As example, the MISRA rule 19.8 states that function-like macros shall be called with all its arguments. Code that violate this rule should not even pass the preprocessor phase, yet it is known that some preprocessors ignore the problem.

Verifying Rules

The programmers must know the language, must know every weakness of their compiler and be extremely careful about not inserting bugs or difficult to read code.

The rules help to avoid many pitfalls, even if several of them will seem trivial to expert developers. Other rules ban constructs that have perfectly reasonable uses, but that can easily be abused. The obvious example is `goto`, but also `continue`, `break` (outside `switch`es) and the comma operator '`,`' might be banned.

```
void do_the_work_dir(file_ref* dir)
{
  assert(file_is_dir(dir));
  file_ref file[1];

  while (dir_get_next_file(file, dir), file_ok(file)) {
    if (strcmp(file_name_str(file), ".") == 0
```

```
        || strcmp(file_name_str(file), "..") == 0)
      continue;

    if (file_is_dir(file)) {
      do_the_work_dir(file);
    }
    else if (file_is_normal_file(file)) {
      do_the_work_file(file);
    }
    else
    ; /* do nothing on different file types.
         (e.g., symlinks, sockets, ...) */
  }
}
```

The above function uses both `continue` and the comma operator, yet it should be clear what it does. Assuming we know the meaning of "do the work," this function does the work on all normal files in the passed directory and in all its subdirectories. The next piece of code does the same without using any of the dubious constructs, is it really more readable?

```
void do_the_work_dir(file_ref* dir)
{
  assert(file_is_dir(dir));
  file_ref file;
  int valid_file;

  do {
    dir_get_next_file(&file, dir);
    valid_file = file_ok(&file);
    if (valid_file
        && strcmp(file_name_str(&file), ".") != 0
        && strcmp(file_name_str(&file), "..") != 0) {

      if (file_is_dir(&file)) {
        do_the_work_dir(&file);
      }
      else if (file_is_normal_file(&file)) {
        do_the_work_file(&file);
      }
      else
      ; /* do nothing on different file types.
           (e.g., symlinks, sockets, ...) */
    }
  } while(valid_file);
}
```

It is a good idea to ban those constructs? Even `goto` has its reasonable uses to separate error management code from normal execution code in an exception-like manner. It seems a bad idea, it seems that the language will be incomplete without some of its parts, but experience shown that those constructs can indeed easily be abused. Those bans are meant to help novices to avoid mistakes, and there always are more novices than experts.

1.3 Applicability

Most of the coding rules just impose what is commonly considered a good programming style and so are applicable to every context. Applying those coding rules can then benefit every developer. Other rules, instead, have a limited applicability because they are meant for automotive industries or airplane embedded systems, where the peculiar hardware limits the available compilers and libraries.

The most evocative example is the MISRA rule 20.4 that bans the use of heap memory. Most of the programs use heap memory, yet MISRA bans it because the targeted hardware the rules refer have little memory and implementing `malloc` and friends would be too expensive. Moreover heap-allocated memory survives until the program releases it explicitly and experience shows it is easy to forget to release memory correctly. The effect is a program that consume more and more memory. It is not a problem for short-lived programs, because when the program terminates it releases all its memory. But it is a problem for programs inside embedded systems, because that software may never stop at all and failures are unacceptable.

1.4 Dissertation Plan

In this chapter we have introduced the problem dissertation started focusing on the problems of the C programming language and the reason of coding rules existence. In Chapter 2 we will see what automatic and manual checking are. Then, in order to understand what we need to implement automatic checkers or in general to ensure that rules has been respected we will categorize the rules. In Chapter 3 we will subdivide by the technologies we need. In Chapter 4 we will categorize the rules by how large is the piece of the software we have to analyze. In Chapter 5 we will see examples of rules: we will explain the rationale and see a simple sketch of the possible implementation of an automatic checker capable of detecting violation to the rule. The final chapter concludes, briefly discussing related and future work.

1.5 Acknowledgement

The work described in this dissertation has been made in collaboration with Roberto Bagnara, Patricia Hill and Enea Zaffanella.

Chapter 2

Checking Rules

> Computers are unreliable, but
> humans are even more unreliable.
>
> Tom Gilb

2.1 Manual Checking

Manual checking is reading the code, see if it violates any rule or if it might be buggy. It is actually effective, but it needs people that know the project and the rules very well. It is the only possible way for some rules that contain non-computable elements. On the other hand it is just plain boring and error prone for the simpler rules.

2.2 Automatic checking

Automatic checking is using a program that takes as input the source code and returns as output a list of violations it could detect.

The absolute best would be: detection means violation, no detection means no violation. Unfortunately this is not obtainable in all cases: depending on the rule a compromise could be needed. Automatic checking does not exclude manual checking, it points to the right direction with the following meanings:

- The detection means violation, and vice versa. There are no compromises, rules checked with this precision do not need manual checking. If the checker detects a violation you just have to fix the source; if the checker does not detect, the code does not violate. We call checkers in this category "exact" checkers.

- "No detection means no violation" checkers. Rules checked with this precision level need manual checking. There are cases where the checker detects a violation and the code actually does not violate, those cases are known as false positives. If there is no detection it means the code does not violate the coding rule. But on the other hand, if the checker detects a violation, still it must be checked manually to see if the code needs to be fixed or if it is a false positive.

- "Detection means violation" checkers. The manual checking is helped by this checkers because upon the detection of a violation then the code need fixing. But if the checker does not detect there might be a false negative and the code might still violate in some way.

- "Informative" checkers, both false negatives and false positives are possible. The detection is just a hint for manual checking. It is just like: "I sense something strange, maybe there are problems there..."

There are two points of view for checkers: correctness and quality. A checker is correct if it does respect what it promises: it means no false positive or negative for exact checkers; no false negative for "no detection means no violation" checkers; no false positive for "detection means violation" checkers. The quality is how many false positives and false negatives it delivers. The less means the better.

Checkers must be correct, but it is also important that the relation with manual checking is known. So developers using the automatic checker know what to expect. If you promise the "no detection means no violation" level, for example, then your checker must never fail to detect a violation and, in front of one, the developers know there might be a coding error or a false positive. It is better an informative checker than an incorrect checker.

Chapter 3

Kind of Rules

> If you think it's simple, then you have misunderstood the problem.
>
> Bjarne Stroustrup

Here we categorize the rules according the technologies we need inside automatic checkers to verify their violations. One peculiarity of the C programming language is the preprocessor: the source code is textually altered before reaching the proper compilation.

The automatic analyzers must be able to read the original code, the preprocessed code and the abstract syntax tree. We will see that we need technologies commonly used inside the compiler, but we will use them differently.

The idea of this subdivision comes from a reinterpretation of the Global GCC (also known as GGCC) work in the same context. [5]

3.1 Trivial Rules

A rule is trivial to verify if we just need a line-based text inspector program such as `grep` and the preprocessor to detect its violation. The simple presence of a piece of text in the preprocessor output implies the violation of the rule and its absence implies the lack of its violation. Rules like "do not use `goto`" are in this category. Having the preprocessor output, we can simply use a regular expression to detect the keyword goto outside string literals.

Older compilers (pre-ansi) allowed multi line string literals, in other words between the quotes you could find a newline (not the escape sequence, a real newline). In this cases you might find a `goto` apparently outside quotes. The simple checker implemented with `grep` would detect a non-existent violation.

Accepting the pre-ansi syntax the checker would be of in the class "no detection means no violation," but it would be an exact checker in the case of standard compliant compilers.

The trivial rules are fewer than it seems: there are rules whose violation might seem easy to detect, but actually the mere presence or the absence of the banned word might mean nothing about the violation of the rule. For example, a rule like: "do not use the function `evil_one`" is not trivial at all. The simple presence of `evil_one` means nothing as it can be an innocuous declaration; its absence does not ensure a non violation because the user might call the function using an extern function pointer.

3.2 Type Enforceable Rules

For type enforceable rules we need a C type system including all its rules about casts, promotions, conversions and interpretation of literals.

The C language is pretty loose about types and allows any kind of conversions. Those operations hide many little pitfalls that often programmers ignore and so deserve great attention from the rules.

```
int main()
{
  int Dead_Beef = 0xdeadbeef;
  printf("the dead beef number is: %d\n", Dead_Beef);
}
```

What number will be printed on standard output? You cannot actually know: if the machine has integers of 32 bits the number will be -559038737; if the machine has larger integers the number will be 3735928559 (the number the user probably expected;) if the machine has smaller integers the program is ill-formed.

Is the program ill-formed? If not, is the number negative or positive? The answer cannot be known without knowing the hardware. Assuming the machine that will run our example has 32 bit long integers, the integer constant is of `unsigned int` type (according to C99 standard [2, Section 6.4.4.1,5] or C89 standard [1, Section 3.1.3.2]). It means that the code that defines `Dead_Beef` explicitly would be `int Dead_Beef = (int)(0xdeadbeefU)`: many developer could be confused.

A rule in this category is HICPP guideline 10.7 that bans expressions that rely on implicit conversion of an operand. Implicit conversions are often a source of confusion for the developers that easily leads to bugs in the final program.

MISRA rule 10.6 states: "A 'U' suffix shall be applied to all constants of unsigned type." If the suffix is there, it must be uppercase. But if it is missing, without being able to read the developer mind, how can we detect a violation

of this rule? We have a violation in two cases: the first case is a unsuffixed integer literal whose type is actually unsigned, as in the previous example; the second case is the presence of an implicit cast whose operand is the integer literal.

3.3 Syntactic Rules

Syntactic rules would be unneeded if we could alter the grammar of the language so as to ban ambiguous constructs. Ambiguity is mainly a problem for developers, but also for the compiler:

```
i(n) { return n; }
```

This is a valid C89 function. It is an integer identity function, but it uses the "implicit int" feature of C89 [1, Section 3.7.1]. To the eyes of many developers it is difficult to read. This function has the same semantics:

```
int i(int n) { return n; }
```

But it results in much clearer code. The implicit `int` is banned by the rules and the new standard also removed this feature. The reason is the same, it is better to diagnose that a type has been forgotten than just assuming the developer meant `int`. Nowadays compilers usually accept the unclear code also in C99 mode, but they warn about its use.

```
void f();

void f(s1, s2)
  char *s1, *s2;
{
  printf("%s -- %s", s1, s2);
}

void g()
{
  f("hello there!\n");
}
```

Should this code compile? The function call is made with a wrong number of arguments, yet some compilers (including GCC 4.3.3) will happily accept it. Why? Because the type of `f` is "function that returns `int` without prototype." No prototype means no obligation of checking the arguments, this is a case where ambiguity is not only for the developers, but for the compiler too. (The compiler is still standard compliant, the program is ill formed. See the C89 standard [1, Section 3.7.1])

In this very example the lack of such a diagnostic is surprising, but in general the definition itself might be in another compilation unit. If the definition was not there, the compiler could only trust the user and trusting the user is an important design point of the C language.

Moreover many developers see the declaration '`int f()`' as "function that returns an `int` and takes no arguments," not as it is actually: "function that returns an `int` and takes an unknown number and type of arguments." For these reasons, rules also ban the declaration of functions without the prototype.

In order to verify syntactic rules, we need parsers technologies. We need to parse the source code and the preprocessor output. Often, violations corresponds to the use of some specific productions of the language grammar.

3.4 Structural Rules

```
void foo()
{
  struct A {
    int A;
  } A;
  A: {
  /* ... */
  }
}
```

This is a valid C function. What is the meaning of the identifier `A`? A single answer is not possible, its meaning is disambiguated by surrounding context. Inside the function `foo` the identifier `A` in isolation will be a reference to the variable; if the identifier appears as `struct A` then it is a type reference; if it appears after `->` or `.` (dot) then it is a reference to the field of `struct A`; finally if it appears after a `goto` it is the label. A compliant compiler will not get confused, but the developer might.

The rules in this category are meant to avoid or reduce this kind of ambiguities: to verify we cannot simply use the grammar of the language, we need to understand the meaning of source code entities.

While making cleaner code for the developers is the first priority, the structural rules often also help to avoid bugs due the compiler misunderstanding complex constructs. The above example might be miscompiled in older compilers where the name spaces support is incomplete.

The MISRA advisory rule 5.6 states: "No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names." It means that the name spaces: ordinary, labels and tags cannot have common identifiers. To check this rule,

and also the others in this category, we need to navigate the abstract tree of the program to collect enough informations.

3.5 Dynamic Rules

A rule is dynamic if it refers to anything that happens at runtime. We can use static analysis technologies to exclude that banned situations happen at runtime without too many false positives.

A famous example is "Ensure the divisor is non-zero." A simple checker might just warn for every division of the program. A slightly better checker might at least exclude divisions where the divisor is a compile-time constant different from zero (e.g., `float h = a/2;`). A more refined implementation, instead, may use static analysis to compute a sound approximation of all possible values of the divisor and see if zero has been excluded or not.

There are various ways to try to verify if a program has runtime proprieties without executing it; but verifying that a program has any interesting propriety is an undecidable problem so we can never have exact checkers.

Moreover the C language is pretty complex to analyze statically. A very difficult problem is, for example, the approximation of pointer operations. For instance, since an object of structure type has a fixed layout in memory you can use a pointer to any structure field to have a correct pointer to any other field. In the following example the function `f` alters the field `first` of `my_aggl` pointee without apparently using it.

```
#define MOVE_PTR(STRUCT_TYPE, PTR, START, FINISH) ((void*)        \
    (((byte*)(PTR)) + (   ((byte*)&(((STRUCT_TYPE*)0)->FINISH))   \
                        -  ((byte*)&(((STRUCT_TYPE*)0)->START)))))

typedef unsigned char byte;
typedef struct agglomeration agglomeration;
struct agglomeration {
  int first;
  int second;
};

void f(agglomeration* my_aggl)
{
  int* first_field
      = MOVE_PTR(agglomeration, &(my_aggl->second), second, first);
  *first_field = 0;
}
```

While this example might seem silly, the idea behind the macro is used in real implementations of data structures. Here is another example of the problems: if a sound analyzer reads '`*a = 5;`' and it does not know where `a`

points-to, the analyzer must assume that every entity in the program might have the value formed by the bit pattern of an integer 5 and that the program might be ill-formed. The point of the examples is showing how making a sound approximation of the whole C type system is difficult.

A correct static analysis can give as result "I do not know" every time there is a difficult to understand construct. So to do some analysis we might not need a complete analysis of the language. It is possible to extract useful informations with a precise analysis of a well-selected subset of language.

Since C is pretty loose about types, this kind of analysis is simpler if the code already respects the type enforceable rules. For instance if the code does not have any type-crossing cast between pointers, the analysis in the previous example can at least assume that every non-integer is untouched.

JSF rule 164 states: "The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive)." This rule might seem strange to the eyes of many developers. This code asserts at runtime that the banned situation never happens:

```
#include <limits>
unsigned f(unsigned a, unsigned b)
{
  assert(b < (sizeof(unsigned) * CHAR_BIT));
  return a << b;
}
```

Such situation must be avoided because it would result in undefined behaviour (see the C99 standard [2, Section 6.5.7,4]).

`(sizeof(unsigned) * CHAR_BIT)` is a compile time known constant so, knowing the hardware, we can compute it. But in order to ensure `b` will always be lesser than this constant and to make a "no detection means no violation" checker without too many false positives we need a static analysis. The following code, for example, should not trigger a violation:

```
#include <limits>
unsigned left_shift(unsigned a, unsigned b)
{
  if (b < (sizeof(unsigned) * CHAR_BIT))
    return a << b;
  else
    return 0U;
}
```

JSF rule 111 states: "A function shall not return a pointer or reference to a non-static local object." When a function returns the program unrolls its stack memory, so trying to use the local objects' memory has undefined behaviour.

It means that what was a local object's address is now an invalid address, so any pointer with those addresses cannot be dereferenced. Violating this rule gives a program with a known semantics, but it is confusing: the function will return an address that the caller cannot use. Returning the address of a static variable does not violate the rule because static objects exist for the whole program execution time, so the returned pointer is usable normally.

In order to verify such violations, a stack points-to analysis is needed. For each call of a function that return a pointer we need to verify that the possible pointees are not in the top stack frame.

3.6 Complex Rules

A rule is said to be *complex* when it does not fit exactly in to any of the categories above, but we think it might be possible to effectively verify it.

```
typedef int table[][4];

table first = { 1,2,3,4, 5,6,7,8 };
table second = { {1,2,3,4}, {5,6,7,8} };
```

The C language is pretty loose about brackets in initializers. Both those forms are correct. Yet MISRA rule 9.2 bans the first form and it imposes that the brackets follow the shape of the original type. Is this a syntactic rule or a type enforceable rule? It is both. Checking the initializer syntax and the type we can detect violations.

JSF rule 181 states: "Redundant explicit casts will not be used." Violations of this rule can be found checking the syntax, to find explicit casts, and the type system, to verify if lacking the explicit cast the semantics of the program changes. If removing the explicit cast the semantics does not change there is a violation. To verify rules in this category, we might need any of the above technologies.

3.7 Difficult Rules

These are rule in which automatic checking is impossible because they involve non-computable concepts or are clearly directed to the semantics aspects of comments. JSF rule 129 states: "Comments in header files should describe the externally visible behavior of the functions or classes being documented." For an automatic checker it is already hard to understand if the comments have a human readable content, it seems impossible to detect if it actually describes the external behaviour of code.

JSF rule 132 states: "Each variable declaration, typedef, enumeration value, and structure member will be commented. Excepted cases where com-

menting would be unnecessarily redundant." It would difficult but possible check whether the comment exists, but impossible to automatically understand if it is meaningful and non-redundant.

While all the rules, including the hard ones, have their importance, hard rules are outside the scope of automatic checkers. The only way to verify them is manual checking.

Yet, we collected all the rules including the ones that seem impossible to automatically verify. Maybe there is an esoteric way to reasonably verify them we do not know. An example is JSF rule 127 that states: "Code that is not used (commented out) shall be deleted." The rule is absolutely reasonable; developers should use a versioning system and commented code should be a very short term way to test. This rule completely refers to comments, is it computable? A possible way to detect violations is trying to parse with the language grammar the comments. If there are only few errors, most likely there is a violation, of course it would be an informative checker, but it might help.

Chapter 4

Wideness

> Every problem is replaceable
> with a bigger one.
>
> Common wisdom

In this section we categorize rules considering how large is the piece of project we need to know in order to detect violations. Ignoring preprocessing, let up recall a possible sequence of abstractions the compiler might follow:

- the bytes, simple unadorned sequences of numbers (e.g., `68 65 6C 6C 6F`);
- the characters, interpretations of the bytes following some kind of encoding. ASCII is by far the most common encoding, and for the bytes of values 0–127 you can be pretty safe that bytes represent the relative ASCII character, but it is good to keep the concepts bytes and characters separate (the example is `hello` in ASCII);
- the tokens, categorized blocks of characters. How the tokens are computed varies a lot between languages but we focus on C (the example might be ⟨identifier, `hello`⟩);
- the parse tree, a syntactic tree where every token is placed;
- the abstract tree (AST), a simplified tree where many non necessary elements have been removed. For instance, the tree structure makes the parentheses redundant, the implicit `int` feature can be normalized.

4.1 Character

The smallest piece of information upon which a sensible coding rule may be defined: a fixed number of characters at a time. JSF rule 9 states "Only those characters specified in the C++ basic source character set will be used. [...]" The rule defines exactly what characters might appear in the source code, all others being forbidden. So, in order to detect violations, you just need to see one character at a time. "Trigraphs shall not used" rules are also in this category, because you need to see exactly three characters at a time and see if a forbidden sequence appears.

4.2 Token

Rules in this category can be checked analyzing the single tokens. There are many trivial rules that ban the use of a particular keyword, but the most interesting one are about the readability of identifiers. HICPP rule 8.4.1 states "Do not write the characters 'l' (ell) and '1' (one) or 'O' (oh) and '0' (zero) in the same identifier." Seeing a single identifier token we can verify and implement an exact checker. Here is an example of code that violate.

```
int swap(int* Il1l, int* l1Il)
{
  int l1I1 = *Il1l;
  *Il1l = *l1Il;
  *l1Il = l1I1;
}
```

4.3 Global Declaration

A C translation unit is a sequence of global declarations. In order to verify rules in this category you need the full knowledge about a single global declaration. There are many different rules in this category: some ones where you just need a little context, like JSF rule 191 "The break statement shall not be used (except to terminate the cases of a switch statement);" others that need a type system behind like HICPP rule 10.7 "Do not use expressions which rely on implicit conversion of an operand."

Other rules can be in this category provided we accept to give up some precision: HICPP's rule 5.11 states "Include explicit cases for all alternatives in multi-way conditional structures." The multi-way conditionals structures in C are `if`s and `switch`es; so the rule can be read as "Ensure every `if` have its `else` and every `switch` have one `default` or a `case` for each possible value of the guard domain." In most cases the information in the global declaration is enough to verify this rule automatically, but we cannot if the `switch` does

not have a `default` and its guard is of an enumerated type defined elsewhere. So accepting some false positives we can read the rule as "Ensure every `if` have its `else` and every `switch` has one `default`." Under this hypothesis the wideness of the rule would be that of a global declaration; however to the rule in its original form is compilation unit wide.

4.4 Compilation Unit

These are the rules that can be verified collecting information from all the global declarations. MISRA rule 18.1 states "All tag types shall be complete at the end of a translation unit." To verify this rule you need to collect all the forward-declared tags and verify if the definitions are also present.

The MISRA rule 8.4 states "For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical." In a compilation unit there can be any number of declarations and at most one can be the definition for a function. This rule says that all declarations must have always exactly the same type. Declaration with different types are usually a sign of a programming error, but some differences are accepted by the compiler.

```
int f(int* a);
int f(int a[]);

typedef int pod;
int g(int a);
int g(pod a);
```

To detect violations of this rule we have to collect all the declarations and verify the return type and the type of the arguments are indeed exactly the same. Indirectly also all the declarations without prototype are a violation. 'Exactly the same type' is strict: it means, for example, that the type `A` of `typedef int A` and `int` are two different types.

4.5 Whole Project

The rules in this category can only be verified by collecting information from different source files used in the project. HICPP rule 8.3.4 states "Ensure each identifier is distinct," once formally defined what "distinct" means you have to collect all the identifiers of the program and verify that any pair of identifiers is well distinct.

Two definitions of a type might appear in separate compilation units (if they are syntactically equivalent), but, in order to remove possibles incoherences, JSF rule 139 states that the definition of types must appear in exactly one

file. It will be placed in the compilation unit using the include directive of the preprocessor. In order to verify you have to check all the compilation units and verify the definitions of each type come from the same file. This rule is important because differences in the type definitions can bring very hard to detect bugs as the linker usually does not diagnose this kind of problems.

HICPP rule 11.2 states "Enclose all non-member functions that are not part of the external interface in the unnamed namespace in the source file." The C language does not have namespaces, but for practical purposes enclosing a function in a C++ anonymous namespace or using the storage specifier `static` in its definition is the same. So for the C language we read the rule as "Ensure all functions that are not part of the external interface have static storage specifier."

To detect violations of this rule we need to enumerate the function forward declarations from the header files, enumerate the non-static function definitions in the implementation files and see if there are any function defined and not forward declared. Each of those functions is a violation. This is an example of "detection means violation"checker. A developer might erroneously put the forward declaration in the header even if it is an implementation function, it would be a violation too. But it will go undetected.

Chapter 5

Implementing Checkers

> Investment in reliability will increase until it exceeds the probable cost of errors, or until someone insists on getting some useful work done.
>
> Tom Gilb

Dividing the rules following different points of view was a preliminary work to decide the best way to implement automatic checking. We needed compiler technologies and something to implement finite state automata and push down automata.

We used *clang*, a front-end for the "LLVM" compiler sponsored by Apple. Both the compiler and its front-end are relatively young projects, but it is easier to interact with their AST than with the one of the better known GCC. So, we used "clang" to lex, parse and make semantic analysis of correctness of the source files.

We wanted to implement rule checkers in the easiest and possibly extensible way. Many rules can be implemented as a finite state automaton or a push down automaton, so we decided to use the language Prolog where that technologies are expressed in a very natural way following the idiom of "a clause is a state." The Prolog source file are often pretty easy to read and a casual user might modify to the rules to his needs even without an extensive programming experience. Programming with the same idiom directly in C++ would mean "a class (or a function) is a state" that would be less intuitive to many people.

A C program is a sequence of declarations: using clang we made that every compilation units become a list of declarations. All kinds of declarations are thoroughly described by clang and we tried to avoid any information loss.

Thinking a lower level, a C program is also a sequence of tokens or characters. We will see that we cannot check some rules from the AST. So we used clang to tokenize. The standard states the comments are replaced with a single space before tokenizing, but a reasonable implementation may tokenize considering comments, spaces and newlines at the same time, after all '`int a;`' and '`int/**/a/**/;`' have the same semantics. This is the approach adopted by clang, but it is possible using a "raw mode," where spaces and comments are explicitly kept.

5.1 Token Stream

Token stream level checkers are usually separated from the rest because the information they need are lost in the subsequent phases. HICPP rule 6.5 states: "Do not write character string literal tokens adjacent to wide string literal tokens."

The C language allows to write `"A" "B"` that is exactly the same of `"AB"`, but you cannot mix wide and normal strings (for instance, this is forbidden: `"NA" L"ÏVE"`, you have to write `L"NAÏVE"`). A reasonable, and standard compliant, tokenizer might join all adjacent string literals in a single token and ignore this possible problem: the resulting string will be wrong. Yet, you cannot notice this problem in the successive phases of compilation, you will have a weird string literal.

Early Stages of Compilation

Trigraphs are three-characters sequences that C-compliant compilers replace with an other character before every other operation. The characters that can be expressed as trigraphs are:

trigraph	character
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

It means that wherever one of the sequences in the left column appears the compiler will see the character in the right column.

```
void erroneous_date()
{
  printf("Date: ????-??-??\n");
}
```

Calling the above function will print on standard output: `Date: ??~~??.`[1] Probably it is surprising to many developers. A standard compliant solution to write the intended string is using the escape sequence `\?` or two adjacent string literals like `"?""?"` to avoid to form a trigraph:

```
void erroneous_date()
{
  printf("Date: ???\?-?" "?-??\n");
}
```

The trigraphs were created so you could use the C language only with characters from the invariant set of the encoding ISO 646, which is a subset of ASCII where those nine characters are not available. Nowadays, ASCII is widely used and there no need to worry about the encoding of those nine characters, they can safely be used. So, all the coding sources ban trigraphs as unclear and unneeded.

```
int f(void)
{
  int x = 0;
  // An harmless or a nefarious comment????????/
  ++x;
  return x;
}
```

The function `f` returns zero. To understand why, we use the GCC's preprocessor to see the preprocessed output:

```
int f(void)
{
  int x = 0;


  return x;
```

[1] GCC made the *non-standard*, but understandable, choice of ignoring the trigraphs by default. You need to use `-trigraphs` to enable them. Instead `-Wtrigraphs` (implied by `-Wall`) will warn when one trigraph is ignored.

```
}
```

No trace of any `++x` statement. The `??/` is the trigraph representing '\', and it occurs at the end of line. The preprocessor removes the '\' joining the two lines, the `++x` becomes part of the comment. Immediately after, the whole comment is replaced by a single space. There are two empty lines because GCC puts an empty line to compensate the line lost with the escaped newline. Doing so GCC preserves line numbering.

Many developers might get confused, but it is impossible to verify that nothing like that happened looking only at the AST. In the AST there is nothing left of spaces, trigraphs, comments and newlines. So simply using the clang's AST is not possible.

MISRA rule 14.3 states: "Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character." The reason is simple: a single semicolon in an unusual place is easy to miss while reading the code.

```
while (*++a != 0)/* We seek the end of the string. */;
f(a);
```

The `f` function is called only once, but it might seem the second line is just badly indented and `f` is called for each `while` cycle. Instead writing the example as the rule states the code becomes clearer:

```
while (*++a != 0)
  ; /* We seek the end of the string */
f(a);
```

To detect violation of this rule we need to know the location of spaces, comments and newlines. But also the presence of a null statement. To implement an exact checker we use the AST to detect null statements and we tokenize around it, and only around the null statement, to verify if spaces and newlines are where are supposed to be.

Rules Related to Preprocessor Directives

HICPP guideline 14.7 states "Do not include comment text in the definition of a pre-processor macro." Tokenizing in raw mode we obtain a list of tokens representing every character in the file, including newlines. Seeking for newlines, the hash (`#`) and the `define` keyword we find the lines where a comment cannot appear. The technology we use is a deterministic finite automaton.

A little more complex is HICPP guideline 14.14 that states "Enclose macro arguments and body in parentheses." Since we only have a list of tokens we need to reimplement a piece of the preprocessor to detect what are the parameter names and in the expansion verify that there are parentheses around them. Moreover, the parentheses must be avoided if the arguments are operands of `#` or `##`. This rule is very important because function-like macros might have behaviour difficult to understand if it is not respected.

```
#define SQUARE(A) A * A

int square_of_sum(int a, int b)
{
  return SQUARE(a + b);
}

int main(void)
{
  return square_of_sum(3, 3);
}
```

This program returns 15. Probably the developer was expecting the value 36. Seeing the preprocessed file we see what happened:

```
int square_of_sum(int a, int b)
{
  return a + b * a + b;
}

int main(void)
{
  return square_of_sum(3, 3);
}
```

The solution is to follow the rule and put parentheses around parameter names and around the whole definition. Just putting the parentheses around `A * A` is not enough: since `A` expands to an arbitrary piece of text, even defining the macro as `(A * A)` we might have unwanted behaviour in situations like:

```
#define SQUARE(A) (A * A)

int f(void)
{
  return SQUARE(3 + 3);
}
```

The definition of the macro might contain any number of nested parentheses, so a finite automaton is not enough: a push down automaton is required.

MISRA rule 19.4 states "C macros shall only expand to a braced initialiser, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct." The reason of limiting macro definitions is that macros expand textually, and without following this rule the code can easily become hard to read.

We already wrote about the parentheses around parameters and expressions. A braced initializer macro cannot hide surprises because its expansion can only appear in very precise contexts: the initialization of variables of a correct type. A macro that expands to a single constant becomes just like any other literal. The type qualifiers and storage qualifiers also do not pose problems thanks the precise places where they can appear in correct source codes.

The most interesting allowed macro is the do-while-zero construct that regulates macros that expand to compound statements. Developers can use the function-like macros to place the compound statement, as a function body, inside the caller code and thus removing the price of the call.

C99 introduced the `inline` keyword that is an hint to the compiler to do just that. But it is only in the new standard and it is only an hint: to ensure the inlining often developers still use function-like macros. The disadvantages are: the address of a function-like macro cannot be taken as it does not exist; shadowing and name space separation do not work with function-like macro names as the preprocessor does not have knowledge of blocks or types.

```
#define SWAP(T, A, B) { \
T t = *(A);             \
*(A) = *(B);            \
*(B) = t; }
```

The previous macro seems a reasonable way to implement a swap that will be always inlined. The following piece of code uses the macro and, as expected, outputs '`Before: 0 1`' and '`After: 1 0`':

```
int main(void)
{
  int a = 0;
  int b = 1;
  printf("Before: %d %d\n", a, b);
  SWAP(int, &a, &b);
  printf("After: %d %d\n", a, b);
}
```

Even if it seems working fine, this macro violates the above mentioned MISRA rule, the reason is that the compound statements do not end with a semicolon as the other statements. Let us see the preprocessor output:

```
int main()
{
  int a = 0;
  int b = 1;
  printf("Before: %d %d\n", a,b);
  { int t = *(&a); *(&a) = *(&b); *(&b) = t; };
  printf("After: %d %d\n", a,b);
}
```

The semicolon placed after the macro is still there, where it denotes an empty and innocuous statement. But it might introduce hard to understand compile-time errors. For instance, this code will not compile:

```
void bing(int d, int* a, int* b, int* c)
{
  if (d > 50)
    SWAP(int, a, b);
  else
    SWAP(int, a, c);
}
```

Your compiler will give a message like "error: `else` without a previous `if`." Without remembering clearly the definition of the macro it might be hard to understand the reason. Here is the preprocessor output:

```
void bing(int d, int* a, int* b, int* c)
{
  if (d > 50)
    { int t = *(a); *(a) = *(b); *(b) = t; };
  else
    { int t = *(a); *(a) = *(c); *(c) = t; };
}
```

The semicolon placed an empty statement after the `if` statement closing it. The `else` is indeed without an `if`. Here we define the same macro as MISRA rule 19.4 permits:

```
#define SWAP(T, A, B) do { \
T t = *(A);                \
*(A) = *(B);               \
*(B) = t; } while(0)
```

With this definition the `bing` function compiles fine and works fine. The preprocessed code becomes:

```
void bing(int d, int* a, int* b, int* c)
{
  if (d > 50)
    do { int t = *(a); *(a) = *(b); *(b) = t; } while(0);
  else
    do { int t = *(a); *(a) = *(c); *(c) = t; } while(0);
}
```

The semicolon does not represent an empty statement anymore as it is part of the `do while` statement. A `do while` statement with zero as guard executes only once and becomes nothing in the final program. In other words, keeping in mind the aforementioned disadvantages we considered, a macro defined as a `do while` statement really works as an inlined function.

To detect violation of MISRA rule 19.4 we have to analyze the macro definitions to see if they are in one of the allowed forms. Once again we can use a push down automaton.

Functions or Functions-like Macro Identifiers

Many rules ban the use of some standard functions because those functions make the code difficult to read or are difficult to implement in the hardware of planes or cars. Often the standard allows for these functions to be implemented as preprocessor macros. This flexibility complicates the implementation of the checkers. An example is `setjmp` of `<setjmp.h>` that might be, a macro, a function or both. So it is not possible to check directly the AST, because the real function name (if any) is implementation-defined.

Moreover even verifying that the token corresponding to the function name never appears might be not enough: first, we need to distinguish between declarations and uses; secondly, the user might use a forbidden function without the token appearing in the source code. Being `FORBIDDEN` a function macro that is banned, in this example the developer manages to call it without the name ever appears as used:

```
#define FORBIDDEN(A) unknown_name(A)
#define HARMLESS(F, A) F ## IDDEN(A)

int f(void)
{
  HARMLESS(FORB, 0);
}
```

To implement the checker we need to inspect the work of the preprocessor, taking the name of every macro that is actually expanded, not just defined.

If the macro `FORBIDDEN` ever appears as expanded we have a violation. The checker is of "no detection means no violation" kind, because it will detect as violation also eventual user-defined macros with the banned name. But it is better than the informative checker we would have just by seeking for the forbidden identifier. In order to make an exact checker we would need to also verify the work of the linker, which would be much more complicated.

5.2 Compiler Warnings

There are few rules whose violation is usually warned by compilers. Translating the compiler's warnings into rule violations is extremely convenient as most of the work is done by the compiler. But we must be very careful about the quality of the checker: without an intimate knowledge of the compiler we cannot assume the checker is exact; usually we have to assume the checker is only informative.

JSF rule 142 states: "All variables shall be initialized before use," GCC has the flag `-Wuninitialized` whose use should catch this kind of problems. But there are few drawbacks: the controls are made only in case of optimization, so `volatile` variables will never be checked. We might get false positives:

```
int foo(int x);

void f(int level)
{
  int x;

  if (0 <= level && level <= 3) {
    switch (level) {
      case 0: x = 1; break;
      case 1: x = 8; break;
      case 2: x = 32; break;
      case 3: x = 128;
    }
    foo (x);
  }
  else {
    foo (1024);
  }
}
```

`foo()` will always be called with a meaningful value. Yet compiling with GCC 4.3.3 we get the warning about `x` that might be used uninitialized. We can get also false negatives:

```
int answer_to_everything()
```

```
{
  int answer = answer; /* The developer meant 42, but she was */
  return answer;       /* tired and wrote the variable name   */
}                      /* a second time.                      */
```

Obviously the variable `answer` is used uninitialized. Yet, GCC 4.3.3 stays silent.

There are cases where the compiler warnings can be used and grant an exact checker. For example, the rules that bans the use of trigraphs can be implemented just using the warning provided by clang. The peculiarity of the warning implies we obtain an exact checker.

5.3 Abstract Syntax Tree Rules

The abstract syntax tree level is represented using a Prolog list of all the global declarations. In particular the functions' definitions have a compound statement as definition.

JSF rule 145 states: "In an enumerator list, the '=' construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized." Every time we are visiting an `enum` definition we have to check how it is defined, and verify that no values are initialized, only the first value is initialized or all the values are initialized. If none of this cases applies, it is a violation.

This rule is meant to avoid confusion about the real value of the enumerate constants: if the enumerate constants are just names, there is no need to assign a value; otherwise if the value is meaningful, the more those values are clearly defined the better. Hence the rule accepts code that sets only the first enumerated constant, as it is clear that the value increments by one for each following enumerators; the rule also accepts code that gives an explicit value to all enumerators or simply does not give a value to any.

In the C programming language it is possible to interchange integers and enumerate constants without writing explicit code, so often enumerated constants are used in operations. Here is an example to show how an enumerator where explicit values are important can become hard to read if the rule is not followed.

```
enum colours {
  red    = 1,
  green,
  blue   = 4,
  fuchsia,
  aqua,
  white,
  yellow = 3,
```

```
  black  = 0
};
```

If you check, you will see that using the enumerate constants you can simulate colour mixing: red + green makes yellow; blue + red makes fuchsia; white − aqua makes red and so go on. Yet, the same enumeration written following JSF rule 145 is much clearer, since all the enumerated constants are explicitly defined:

```
enum colours {
  red     = 1,
  green   = 2,
  blue    = 4,
  yellow  = red | green,
  fuchsia = red | blue,
  aqua    = green | blue,
  white   = red | green | blue,
  black   = 0
};
```

HICPP rule 5.2 states "For boolean expressions ('if', 'for', 'while', 'do' and the first operand of the ternary operator '?:') involving non-boolean values, always use an explicit test of equality or non-equality." In the C language integral type can be used as truth values. The meaning is: zero means false, non-zero means true. A function that verifies if an array contains null pointers might be written as:

```
int null_pointers(const char* const array, size_t array_length)
{
  int found_null_pointer = 0;
  while (! found_null_pointer && array_length) {
    --array_length;
    found_null_pointer = (array[array_length] == NULL);
  }
  return found_null_pointer;
}
```

An expert eye is needed to understand outright what is going on. The meaning of '`! found_null_pointer`' should be clear as it is conceptually a truth value (either it is found or it is not), but the meaning of `array_length` a as truth value is much less clear. Here is an other example:

```
void print_kind(unsigned n)
{
  (n % 2) ? printf("even ") : printf("odd ");
```

```
}
```

Did you notice the bug? `(n % 2)` means `(n % 2 != 0)` that considering type and operators is the same as `(n % 2 == 1)`; in other words is a test of oddness, not of evenness. The problem is once again bound to the use of an integer as truth value. In fact the operator `%` in general does not return a truth value, but it returns a remainder. Using the explicit form, overall the one with equality, not only satisfy the rule: it is also much clearer.

Functions that return the order relation between two objects often return a positive integer when the first object is greater than the second, a negative integer when the first object is smaller than the second, and zero when the objects are equal. The standard function `int strcmp(const char*, const char*)` works this way.

Conceptually the order relation is not a truth value, yet we might see code like:

```
if (!strcmp(name, "Smith")) f();
```

that is less readable than

```
if (strcmp(name, "Smith") == 0) f();
```

Some developers might argue that they read '`if (!f())`' exactly as '`if (f() == 0)`' and '`if (f())`' exactly as '`if (f() != 0)`'. But once again, rules are meant to help avoiding mistakes, especially from novices. Writing clear code is the key. Keeping the concept of 'integer used as truth value' and 'integer used in some other way' separate makes code clearer.

We can implement a "no detection means no violation" checker by for this rule inspecting all the boolean expressions mentioned by the rule: if the type of the expression is not `_Bool` and the code does not have an explicit equality or relational operator, then there is a violation.

Unfortunately C89 does not have a boolean type and even if the C99 introduces one, integral types are often used as truth value. It means that every time a developer with critical awareness uses an integral expression that represents a truth value the checker might detect a false positive. To reduce the number of false positives we can add a list of functions that are known to represent a truth value, or follow some conventions based, e.g., on a known prefix or suffix for integral variables used a truth value. For example, all functions whose name begin with `is_` or contain `_is_`, like `is_sorted()` and `list_is_empty()`, might never trigger this checker.

JSF rule 166 states: "The `sizeof` operator will not be used on expressions that contain side effects." The `sizeof` operator accepts types and expressions: using `sizeof(expr)` where the type of `expr` is `T` is just like using

`sizeof(T)`.[2] Unfortunately experience shown that some developers misunderstand the meaning of `sizeof` and think that the expression is actually executed.

```
size_t f(int* a)
{
  return sizeof(++(*a));
}
```

Calling the function `f` does not change the value of `*a`. The compiler, at compile-time, computes the type of `++(*a)`: the type of `a` is pointer to `int`, the type of `*a` is `int`, the type of `++(*a)` is still `int`. Finally the compiler reads the expression as `sizeof(int)`.

On the other hand it is legitimate to use `sizeof` with expressions and let the compiler compute the type. So the rule does not ban the use of expressions as operands of `sizeof`, but it bans only those expressions having side effects.

The usual definition of expression without side effects is: "expression whose only effect on the program is its return value." In C composing variable references and many operators you can only form expressions free of side effects (e.g, `+`, `-`, `==`, ...) So we can implement a "no detection means no violation" checker flagging all the occurrences of `sizeof` whose operand contains at least one operator with side effects (e.g., `++`, `=`, `+=`, ...) or a function call.

The possible false positives are about functions calls whose execution do not have side effects, but the use of `sizeof` with function calls as operands is rare.

HICPP rule 7.7 states "Do not cast pointers to and from fundamental types." MISRA 11.3 states something similar. Casting a pointer to a fundamental type has implementation defined or undefined results, so it should be avoided.

Yet it is needed, for example developers might need to use a known integer value as device address. Converting a pointer to an integer might be needed to pass the address of a piece of memory belonging to a memory-mapped hardware device.

A standard compliant compiler follows its implementation defined behaviour when converting the integer to, or from, a pointer if the two types have the same size (see the C99 standard [2, Section 6.3.2.3]). Implementation defined behaviour might give surprises: for example, there is a version of the IBM ILE C compiler that from the cast of an integer to pointer always returned

[2] Actually if `expr` is the name of an array whose type is "array of unknown size," the `sizeof` expression cannot be computed at compile time. Expression `expr` will be actually executed at runtime. But it cannot have side effects since any operator applied to an array name transforms it to an other type: the underlying type (`*`, `[]`, ...) or a pointer type (`+`, `=`, ...).

a NULL pointer.[3]

Of course such behaviour is unacceptable if the conversion is needed; but even if the compiler performs the expected conversion the developers must be careful that the size of pointers and the size of integer are equal. Here an example of safe code:

```
void set_thermometer_output(thermometer* ther, int* temperature)
{
  int ulong_keeps_pointer[sizeof(int*) == sizeof(long) ? 1 : -1];

  unsigned long temperature_address = (unsigned long)temperature;
  ther-> output = temperature_address;
}
```

The definition of the array is based on a compile-time constant: if the first expression of the ternary operator is true the array will be large one; otherwise it will be large minus one. Of course arrays cannot have a negative size and the program will fail to compile.

The compiler will say there is an array of negative size. It will give the file position and, with a little luck, even the array name. The developers will immediately know they wrote code that, in that particular platform, have undefined behaviour. On the other hand, if the program compiles, it means `unsigned long int`s have the same size of pointers and so the conversion has a known effect.

Defining the array does not have any negative effect on the final program, since any decent optimizing compiler will remove it from the object file. Yet, it is a good way to ensure that the assumptions the developer make actually hold.

To implement a "no detection means no violation" checker we have to analyze the casts. Any cast, both implicit or explicit, between a non-pointer fundamental type and a pointer is a violation. If the cast is between pointers and integers there might be a false positive. In order to reduce their number we can seek for static asserts (made with the array and ternary operator trick) in the blocks where we found the possible violations: the existence of such an assertion implies the developer meant to make the cast.

All considered, developers should ask if there is really need of casting between pointers and integers. Most of the time what is actually needed is filling in a pointer with a known value or store the pointer somewhere for later retrieval. If the developer is in one of those cases, there is a standard compliant solution that does not use integers, so it works even in platform where no integer type is capable of storing pointers or when using compilers where the cast cannot be used. It uses an array of `unsigned char`s and `memcpy`

[3] See documentation in `http://publib.boulder.ibm.com/infocenter/iadthelp/v7r0/topic/com.ibm.etools.iseries.pgmgd.doc/cpprog448.htm`

of <string.h>. Here is an example of filling in an IO device pointer with a meaningful known value:

```
IO_device* gimme_my_device(void)
{
  int sizes_ok[sizeof(IO_device*) == 8 ? 1 : -1];
  static int init = 0;
  static unsigned char device_address[]
      = { 0x15, 0xEE, 0xAB, 0xAD, 0xDE, 0xAD, 0xFA, 0xCE };
  static IO_device* device_address_ptr;

  if (init == 0) {
    init = 1;
    memcpy(&device_address_ptr, device_address, 8);
  }

  return device_address_ptr;
}
```

As using integers, it is programmer responsibility that the address is meaningful, the disadvantage is that byte endianness of the machine must be taken in account. In a little endian machine the example's pointer value will be 0xCEFAADDEADABEE15.[4]

It is also possible doing the opposite, copying the content of the pointer's piece of memory to an array of unsigned chars in order to store the value somewhere. In this case there are no disadvantages: a situation where the pointer value is stored in a machine and it is read in an other machine with different endianness seems very rare. (see the C99 standard [2, Section 6.2.6.1,4]).

HICPP rule 8.4.5 states "Do not use the plain 'char' type when declaring objects that are subject to numeric operations. In this case always use an explicit 'signed char' or 'unsigned char' declaration."

The reason of this rule is that char does not follow the rules of other the integers type. While for other integers saying signed is exactly as omitting it, the character types char, signed char and unsigned char are three different types. The fact that char is signed or not is implementation defined, but in any case it is a different type than signed char or unsigned char.

This rule, for the sake of clarity, imposes to divide the concept "character" and "small integer." char must be used for characters, unsigned char or signed char must be used for small integers.

[4] *Endianness* in this context is the byte order used by the machine to memorize types larger than one char. There are three orders: *big-endian* where numeric significance of bytes decrease with the memory address, numbers are stored as we would read them; *little-endian* where numeric significance increase with the address; *mixed-endian* bytes are stored in an other order. For instance the PDP-11 stored some types so that the significance order was: 2nd byte, 1st, 4th and 3th byte.

Characters used as small integer have the expected result: the `unsigned char` type works as an integer modulo $2^{\text{CHAR\_BIT}}$ (usually 256); the `signed char` type gives you (at least) the interval $[-127, 127]$ with the usual integer rules (see the C99 standard [2, Section 5.2.4.2.1]).

Plain `char` works as one of the two. The first problem is that you do not know one as it is implementation defined. The second, and main, problem is that there is no standard relation between the number and the character. While common, the following example code is not portable:

```
char uppercase(char A)
{
  return ('A' <= A && A <= 'Z') ? A + 32 : A;
}
```

It might work, if the implementation-defined representation of characters is a subset of the ASCII where the value associated to letters is the same of the ASCII. It is probably true in all major platforms, yet we are speaking of code that will run in embedded systems. Certain assumptions are risky.

Casting between the three types does not present any risk, so developers might be tempted to eliminate the plain `char` type and always use the `unsigned char` type as it has a known semantics and it cannot overflow. The following function has been written with the '`unsigned char` only' policy. It verifies the passed filename has extension `.bzb`:

```
#include <string.h>

int check_extension(const unsigned char* filename)
{
  int correct = 0;
  unsigned char* extension
      = (unsigned char*)strrchr((const char*)filename, '.');
  if (extension != NULL) {
    correct = (strcmp((char*)extension, ".bzb") == 0);
  }
  return correct;
}
```

The idea of using only signed or unsigned `char` cannot be applied easily because many useful standard functions take as parameter pointer to `char`s or plain `char`s. To use those functions the developer has to cast very often: as you can see, even this simple example is already so cluttered of casts that it is hard to read. Moreover, the C syntax of casting is simple and clean, so simple and clean that it can easily cast away `const` and `volatile` qualifiers introducing extremely hard-to-find bugs.

The solution is to follow the rule and consider the three types as separate: `unsigned char` and `signed char` types as small integers that can be used in

mathematical operations and not as characters; and `char` as characters, which must altered only through standard functions.

We can implement a "no detection means no violation" checker for this rule: we mark as violations all the casts between the three character types and the use of mathematical operators (`+`, `-`, `/`, ...) where at least one operand is a plain `char`. We might have false positives in reasonable casts.

Conclusion

> When designing a program to handle all possible dumb errors, nature creates a dumber user.
>
> One of the Murphy's law

To program embedded systems often the languages available are only C and Assembly, so the industry tends to use C because it is more flexible, expressive and yet fairly low level. On the other hand the C programming language is hard, because it is concise and subtle. This is the reason of the existence of the rules: avoiding common programming mistakes. The mere existence of the rule does not avoid errors nor force developers to follow them. If a team of developers decided to follow some rules, in order to be useful the rules must be respected in the final software and accidental violations must be detected and fixed. For this reason we presented two different approaches that complete each other: automatic and manual checking. Manual checking completes automatic checking because automatic checking might find false positive or negatives and because there are non-computable rules. Automatic checking completes manual checking because it gives hints to developers: exact checkers without fail find violations; the "no detection means no violation" checkers might give false positives, but in absence of detections the source code never violates; the "detection means violation" checkers might fail to get violations, but if they detect there is a violation. Finally there are also the informative checkers whose results are indeed only a hint. In order to understand what we needed to implement an automatic rule checker we organized the rules by technologies needed and by wideness. Sorting by technologies shown that we need technologies usually inside the compilers. Sorting by wideness shown that we need an infrastructure to keep information from different source files of the software project. We finally saw examples of rules: for each rule we saw a detailed explanation of the rationale of the rule with examples and a sketch of a possible implementation.

Related Work

This is not the first attempt of automatically verify code quality, here we present some of the existing works: GGCC [5] is an automatic checker. But it takes another approach, it does not directly try to find code flaws: it an early attempt to formalize the rules using logic programming and to make an interpreter that reading the rules' clauses does the checking.[5] *LDRA Testbed* is a static analyzer capable of detecting various possible flaws in the C source code. Some of the flaws correspond to violations of coding rules.[6] *ROSE* of LLNL (Lawrence Livermore National Laboratory) is a source code transformation tool. *Compass* uses it to implement a tool capable of detecting code flaws based of known "anti-patterns." The already implemented checkers are quite simple, but they cover many of the common errors and relative rules. Moreover *Compass* is intended to show an interesting use of *ROSE* more than being a checker itself.[7] *Fortify Source Code Analysis* (also known as *Fortify SCA*) is a software thought to analyze the source code of various languages, including C and C++.[8] It finds few flaws by itself, but it has been used to implement a rule checker for a subset of the coding rules of CERT (Carnegie Mellon University's Computer Emergency Response Team).[9] We did not see any CERT rule in this dissertation, but the rationale is always the same.

Future Work

Future work should move in two directions: improving the rules, improving the checkers. The rules exist to improve the code. So we need to understand why developers do mistakes in order to write, or improve existing, practical rules whose adoption reduces programming errors. For instance the older MISRA document [6] banned the use of plain `char`s. For the reason we saw in the Section 5.3 following that rule is impractical; so it has been removed. The same document indirectly also banned the EBCDIC encoding, the effect was that some hardware for embedded systems could not be used. The newer MISRA [7] states that the selected encoding has to be documented, which is both practical and reasonable. Once a rule is decided it must hold in the source code, so for each rule there must be the most convenient way to verify it. Automatic exact checkers are the most convenient way to verify coding rules. However, since there are cases were an exact checker cannot be coded we have to reduce as much as possible the rate of the false positives and false negatives.

[5] `http://www.ggcc.info/`
[6] `http://www.ldra.com/testbed.asp`
[7] `http://rosecompiler.org/projects.html`
[8] `http://www.fortify.com/products/sca/`
[9] https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=7766071

Bibliography

[1] International Organization for Standardization. *ISO/IEC 9899:1990: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, 1989.

[2] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, 1999.

[3] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003.

[4] Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*. Lockheed Martin Corporation, 2005.

[5] Guillem Marpons-Ucero, Julio Mariño-Carballo, Manuel Carro, Ángel Herranz-Nieva, Juan José Moreno-Navarro, and Lars-Åke Fredlund. Automatic coding rule conformance checking using logic programming. In Paul Hudak and David Scott Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2008.

[6] Motor Industry Software Reliability Association. *MISRA-C:1998 - Guidelines for the use of the C language in critical systems*. MIRA Limited, Nuneaton, Warwickshire, UK, 1998.

[7] Motor Industry Software Reliability Association. *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*. MIRA Limited, Nuneaton, Warwickshire, UK, 2004.

[8] The Programming Research Group. *High Integrity C++ Coding Standard Manual*, 2.4 edition, December 2006. Available from `http://www.codingstandard.com/`.