A Coding Rule Conformance Checker Integrated into GCC

Guillem Marpons[1,4] Julio Mariño[1,4] Manuel Carro[1,3,4]
Ángel Herranz[1,4] Lars-Åke Fredlund [1,2,4]

Universidad Politécnica de Madrid
Boadilla del Monte, Spain

Juan José Moreno-Navarro[1,4]

Universidad Politécnica de Madrid, IMDEA Software
Boadilla del Monte, Spain

Álvaro Polo[5]

Telefónica I+D
Madrid, Spain

Abstract

Coding rules are often used in industry to foster best practices when coding software and to avoid the many hazardous constructions present in languages such as C or C++. Predictable, reliable tools are needed to automatically measure adherence to these practices, as manually checking for compliance is cumbersome. Moreover, due to the wide range of possible coding rule sets, easy of customization is a need in order for these tools to be of practical use. With this aim in mind, we present an extension of the GNU Compiler Collection (GCC) that flags those code fragments that do not conform to coding rules belonging to a given set. These sets of coding rules can be defined using a high-level declarative language based on logic programming, thereby making it possible to easily check code for conformance with respect to rules addressing the particular needs of a project, company, or application area.

Keywords: Coding Rule Checking, Programming Environments, Quality Assurance, Logic Programming.

[1] Work partially supported by PROFIT grants FIT-340005-2007-7 and FIT-350400-2006-44 from the Spanish Ministry of Industry, Trade and Tourism, and grant S-0505/TIC/0407 (PROMESAS) from Comunidad Autónoma de Madrid.

[2] Work partially supported by a Ramón y Cajal grant from the Spanish Ministry of Science and Innovation.

[3] Work partially supported by the Spanish Ministry of Science and Innovation grant 2008-05624/TIN (DOVES) and by EU Project number 215483 of the Communication and Technologies programme (S-Cube)

[4] Email: {gmarpons,jmarino,mcarro,aherranz,lfredlund,jjmoreno}@fi.upm.es

[5] Email: apv@tid.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Languages such as C or C++ need to be used in a disciplined manner to minimize hazards due to their weaknesses and more error-prone features. To that end, it is common in industry to require that code rely only on a well-defined subset of the language, following a set of coding rules.

Some standard rule sets do exist listing good general programming practices for a given language, like *High-Integrity C++* (HICPP [**?**]). MISRA-C [**?**] is another leading initiative elaborated by *The Motor Industry Software Reliability Association* (MISRA). It contains a list of 141 coding rules aimed at writing robust C code for critical systems. In practise many organisations – or even projects – need to establish their own coding rule sets, or adapt the existing ones.

However, an automatic method to check code for conformance is needed[6] for coding rules to be of practical use, no matter who devises the coding rule set or dictates its use. There exists a number of commercial compilers and quality assurance tools from vendors such as IAR Systems [**?**] and Parasoft [**?**] that claim to be able to check code for compliance with a subset of HICPP, MISRA-C or other standards. Other tools, e.g. Klocwork [**?**], define their own list of informally described rules aimed at avoiding hazards, and users can add new rules by means of complex application program interfaces (usually in C or C++). But, in absence of a formal and concise definition of rules, it is difficult to be certain about what these tools are actually checking, and two different tools could very well disagree about the validity of some particular piece of code with respect to, e.g., the same MISRA-C rule.

In [**?**] we proposed a framework to precisely specify rule sets and automatically check (non-trivial) software projects for conformity. On the rule-writer side, a logic-based language (currently a subset of Prolog with minor syntactic extensions) permits to easily capture the meaning of coding rules, constituting a more practical mechanism for user-defined rules than those provided by most other tools. There have recently appeared other tools providing high-level languages to define code checks, such as Klocwork Insight, Parasoft RuleWizard or Semmle Code [**?**]. In contrast with these other tools, our proposal relies on general logic programming (in Semmle Code all code queries are translated into Datalog). With the expressiveness of full logic programming we can cope with the potentially infinite sets that appear when reasoning about C++ templates and template instance properties. It also gives us the opportunity of using unification on structured terms and, for example, use logic variables as template parameters, greatly simplifying the definition of rules on templates (see [**?**]). This comes at the cost of jeopardizing the termination of the execution of some rules, unless suitable constraints are established on the rule definition language.

The other salient feature of our coding rule checking tool is that it has been integrated into the GNU Compiler Collection (GCC, `gcc.gnu.org`) development tool-chain. In this work we present a new version of our coding rule checker, improved the one presented presented in [**?**], that extracts all the needed information about C++ programs while compiling them with GCC. One reason to put together the rule checker and the GCC tool-chain, which is a non-trivial task, is to make

[6] Although some rules may be undecidable, finally needing human intervention.

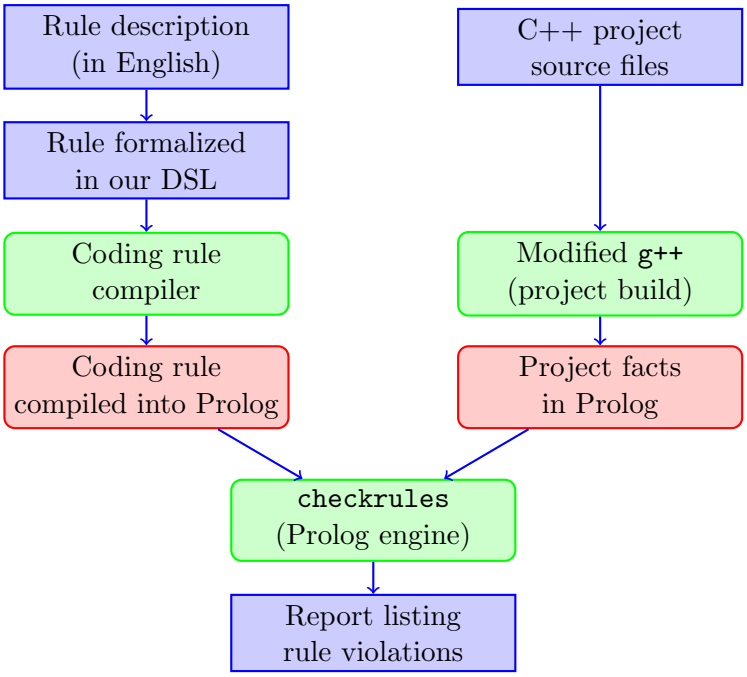

Figure 1. Overall rule checking procedure. The left-hand part of the diagram contains the steps related to rule formalization. On the right, program feature extraction is depicted. Rule conformance checking is shown at the bottom.

the checker readily available in the everyday tool of thousands of developers, which will undoubtedly foster the adoption of coding rules in many projects. Moreover, by using a single parser and semantic analysis engine (used for object code generation) both for the compilation and to gather information for the code checker, any possible discrepancy on how code is interpreted (which could happen if a different parser / anlyzser were used) is avoided. In addition, information gathered by static analyses already present in GCC can be reused to implement rules that need it.

2 Coding Rule Checking

Our procedure for checking if some software conforms to a given coding rule consists of three main steps, depicted in Figure 1:

(i) Formalize the rule [7] in a logic-based domain-specific language that is automatically translated into Prolog. Standard rule sets typically use plain English for definitions.

(ii) Transcribe the necessary program information into the same representation, i.e. as Prolog facts. Programs to be analysed are compiled with our modified version of GCC, with an added flag `-fipa-codingrules` for dumping these facts to a file.

(iii) Analyze the rule violation predicate together with the Prolog facts which describe the program at the appropriate abstraction level. This is done by seeking counterexamples to the rule with a standard Prolog system – Ciao Prolog [**?**],

[7] In fact, its violation.

in our case. The Prolog checker is available as a command line tool called `checkrules`[8] that receives as input the file of facts generated by GCC and the (precompiled) coding rules themselves.

The current rule definition language is a subset of Prolog with some syntactic extensions (Section 3). In the future we plan to facilitate rule formal definition with a completely declarative logic language featuring sorts, constructive negation, and appropriate quantifiers (some details are given in [**?**]). Our aim is to enable developers not familiar with Prolog to formalize coding rules.

More details on rule formalization can be found in Section 3, and program feature extraction is discussed in Section 4.

3 Structural Coding Rule Definition in Prolog

We have focused first on what we term *structural* coding rules: those that have to do with objects in the code such as classes or functions, their static properties, and static relations among them such as inheritance, containment, or usage.

A good example of this kind of rules is Rule HICPP 3.2.4 (see Figure 2), that reads "*An abstract class shall have no public constructors.*" Abstract classes are those that have at least one unimplemented member function. This rule helps to make explicit the fact that abstract classes cannot be instantiated, but need to be used through subclasses.

Formalizing the rules requires a set of language-specific predicates representing structural information about, e.g., the inheritance graph of the checked program. Table 1 shows some predicates used for defining a number of rules intended for the C++ language, including the example above. These predicates constitute the programming interface for writing rules and are defined on top of the information generated by the compiler, as explained in Section 4. For example, the unary predicate *abstract_class* is used to define the aforementioned rule.

The Prolog formalization of the rule 3.2.4 codifies a violation of the rule, i.e., that an abstract class `Class` has a member `Ctor` which is a public constructor. If the rule can be violated, concrete instances of `Ctor` and `Class` in the software analysed will be returned to the user by `checkrules`, along with a warning message, associated with the rule by means of operator `#`. The arguments of the predicate can be displayed as part of the user message.

Another example of structural rule is HICPP 3.3.13, that reads "*do not invoke virtual methods of the declared class in a constructor or destructor.*" The rationale behind it is that member functions of the same object are always statically bound if called from a constructor or a destructor. In this case two methods pertaining to the same class are returned as witnesses of a violation of the rule. Note the use of disjunction (`;`) and the special syntax *predicate*`+` to denote the transitive closure of *predicate*. We are interested in methods directly or indirectly called by `Caller`, thus we use the transitive closure of the predicate *calls* defined in Table 1.

Rule HICPP 3.3.15 exemplifies the use of negation. It says: "*ensure base classes*

[8] Both our extended GCC and the `checkrules` tool are available at the web site of the GlobalGCC project: www.ggcc.info/?q=download, licensed under GPL. We try to keep our GCC in sync with the GCC trunk.

Table 1
A subset of the predicates necessary to describe structural relations in C++ code.

Predicate	Meaning
Properties of classes and methods	
$abstract\_class(c)$	Class c has some non-implemented member function.
$constructor(m)$	Member m is a constructor. There is an analogous predicate $destructor$.
$public\_member(m)$	Member function m has public visibility. There are analogous predicates for protected and private visibility.
$virtual\_member(m)$	Member function m is virtual (invocations of the method are dynamically dispatched).
Inheritance	
$immediate\_base\_of(a, b)$	Class b directly inherits from a.
$base\_of(a, b)$	Transitive and reflexive closure of $immediate\_base\_of/2$.
$public\_base\_of(a, b)$	Class b immediately inherits from class a with public accessibility. There are analogous predicates for other accessibility choices and also for virtual inheritance: $virtual\_base\_of$.
Relations between (member) functions	
$calls(a, b)$	(Member) function a has in its text an invocation of (member) function b.
$overrides(a, b)$	Member function a is defined in a derived class as a re-declaration of member function b (they have the same signature but different implementation).
$overloading\_members(a, b)$	Member functions a and b are declared in the same class and have the same name (but different signature).
$declares\_member(c, m)$	Member function m is declared (or re-declared with a new implementation) in class c.
$has\_member(c, m)$	Class c has defined a member function m. m can be inherited from a base class.
$have\_same\_sig(a, b)$	Functions a and b have the same arity and argument types.

common to more than one derived class are virtual." With the help of the justification that accompanies the rule, we can reformulate it as follows:

```
'HICPP 3.2.4'(Class, Ctor)
    # 'Class ' # Class # ' has pure virtual members and a constructor '
    # Ctor # ' that is public.'
:-
    abstract_class(Class),
    has_member(Class, Ctor),
    constructor(Ctor),
    public_member(Ctor).


'HICPP 3.3.13'(Caller, Callee)
    # 'Member function ' # Caller # ' is a constructor or destructor '
    # 'and calls virtual function ' # Callee # ' of the same class.'
:-
    has_member(SomeClass, Caller),
    (
        constructor(Caller)
    ;
        destructor(Caller)
    ),
    has_member(SomeClass, Callee),
    virtual_member(Callee),
    calls+(Caller, Callee).


'HICPP 3.3.15'(A, B, C, D)
    # 'Class ' # D # ' repeatedly derives from class ' # A # ' through '
    # B # ' and ' # C # ', and ' # C # ' does not use virtual '
    # 'derivation for extending ' # A # '.'
:-
    immediate_base_of(A, B),
    immediate_base_of(A, C),
    B @< C,
    base_of(B, D),
    base_of(C, D),
    \+ virtual_base_of(A, C).


'HICPP 3.3.5'(Method, Overriding, Derived)
    # 'Virtual function ' # Method # ' is overridden by ' # Overriding
    # ' in class ' # Derived # ', and some function that overloads '
    # Method # ' has no overriding in ' # Derived # '.'
:-
    has_member(Derived, Overriding),
    overrides(Overriding, Method),
    overloading_members(Overloads, Method),
    \+ (
           has_member(Derived, Overriding2),
           overrides(Overriding2, Overloads)
       ).


overrides(M1, M2) :-
    declares_member(Derived, M1),
    immediate_base_of(Base, Derived),
    has_member(Base, M2),
    have_same_sig(M1, M2).
```

Figure 2. Prolog formalization of some rules in the HICPP rule set.

> Rule 3.3.15 is violated if there exist classes A, B, C, and D such that: class A is a base class of D through two different paths, and one of the paths has class B as an immediate subclass of A, and the other has class C as an immediate subclass of A, where B and C are different classes. Moreover A is not a virtual base of C.

The rule is written in Prolog as shown in Figure 2. Negation-as-failure (operator `\+`) appears in the last line. Since the semantics of this last Prolog goal relies on the instantiation state of logic variables `A` and `C`, the order of goals inside a clause becomes relevant, not only for performance concerns. In order to avoid this, we plan to replace negation-as-failure by constructive, logically meaningful nega-

tion. In particular, we are investigating the applicability of constructive intensional negation [**?**]. In this example we also use operator `@<` to get pairs of distinct classes where `B` and `C` cannot interchange their roles.

Finally, rule HICPP 3.3.5 exemplifies a more complex use of negation: one must "*override all overloads of a base class virtual function.*" Code for the rule is shown in Figure 2, along with the implementation in Prolog of one of the predicates used to define it: the *overrides* relation between two class methods.

4 Using GCC to Gather Program Information

The middle-end of GCC contains a set of transformation and optimisation passes that are independent from both the compiled language and the target architecture. Many (but not all) of the program features needed for writing rules are common to multiple languages and have a common representation in the middle-end, even if the semantics may differ between different languages. For example, constructs as templates and friends are almost exclusive of C++, and only the C++ front-end knows about them.

In our extended version of GCC we have instrumented both the middle-end and the C++ front-end (totalling around 2.8 KLocs of new code), but most of the analysis is done in a new middle-end pass. Implementing functionality in the middle-end has many advantages: adding a new pass is simple and clean, and there is no overhead unless the pass is enabled using a corresponding flag. Furthermore, the new functionality may be reused for other GCC languages.

Our modified GCC writes Prolog facts describing structural properties of the analysed software (which can be either a program or a library) to a file. All the source files in a project have to be analysed because structural rules typically involve project-wide properties. The global analysis is carried out by relying on the building process used in the project (e.g., `make`, `cmake`, `ant`, etc.) and accumulating all the Prolog facts of different compilation units in a single file, which is subsequently analysed with `checkrules`.

For every relevant entity in the code a Prolog term of the form *`entity`*`(`*`GLOBAL_KEY`*`)` is generated, where *`entity`* is one of `enum`, `enum_value`, `union`, `record` (either a *struct* or a *class*), `function`, `global_var`, `method`, `field`, and `bit_field`. *`GLOBAL_KEY`* is a project wide identifier of the entity, based on *name mangling* [**?**]. *Mangled* names are a special encoding of names of functions, variables, etc. generated by the compiler for the linker and other tools that have to deal with information coming from different compilation units. They resolve, among other possible name clashes, overloaded function names, including overloading originated by templates.

The naming scheme for local entities (local variables, function arguments, etc.) is based upon the scope in which they are defined (that is a global entity) and its local identifier. For anonymous entities (e.g., anonymous union fields) a numerical identifier is generated.

Following this identification scheme, a Prolog predicate exists for every relevant property of global and local entities, and terms are generated in the output for every occurrence of the property. These terms have the structure shown in Table 2 for

Table 2
Structure of Prolog terms representing properties and relations among C++ global entities. *ACCESS_SPECIFIER* is one of `public`, `protected`, or `private`.

`virtual(method(`*GLOBAL_KEY*`))`

`accessibility(method(`*GLOBAL_KEY*`),`*ACCESS_SPECIFIER*`)`

`contains(namespace(`*GLOBAL_KEY*`),`*entity*`)`

`contains(record(`*GLOBAL_KEY*`),`*entity*`)`

`enumerates(enum(`*GLOBAL_KEY_1*`),enum_value(`*GLOBAL_KEY_2*`))`

`extends(record(`*GLOBAL_KEY_1*`),record(`*GLOBAL_KEY_2*`))`

`virtual(extends(record(`*GLOBAL_KEY_1*`),record(`*GLOBAL_KEY_2*`)))`

Table 3
C++ projects used in our experiments, with a measure of their number of C++ lines.

Project	Version	Description	KLoc
Bacula	2.2.8	A network based backup program.	19
IT++	3.10.12	Library for signal and speech processing.	46
PPL	0.9	Parma Polyhedra Library: numerical abstractions for analysis of complex systems.	60
dlib	17.0	Library about threading, networking, data compression, and more.	77

e.g. `virtual` and `accessibility` properties of global entities. Besides individual properties, relations among global entities exist. Some examples of binary relations among global entities can be found in Table 2: `contains`, `enumerates`, and `extends`. Relations such as `extends` can also have properties attached, such as e.g. `virtual`.

Prolog terms are generated to represent types and attaching types to entities, and also to associate code locations to entities, which is needed for user output.

The higher-level abstract predicates in Table 1 are defined using the low-level predicates introduced before in this section. Such higher-level predicates follow the usual C++ terminology (*base classes*, *member functions*, etc.), facilitating the formalization of coding rules for a C++ expert. This two-layered predicate architecture is also intended to better support extending the rule checking facility to other target languages. More details on the implementation of our modified GCC compiler can be found in [**?**].

5 Experimental Results

Our tool-chain is capable of tackling arbitrary C++ code. In combination with any `make`-like software building tool, it can be used to catch violations of structural coding rules on existent C++ projects. We have implemented 12 structural rules from the HICPP rule set so far, including those mentioned in this paper, and checked some small and medium-sized free software projects for compliance with

Table 4
Number of rule violations and user time (in seconds) consumed by the different steps in our rule checking procedure. BT is the total build time of the project, and BT_{CR} is the total build time with structural data gathering enabled. LT is the time that takes `checkrules` to load the project. Columns labeled with a rule number show, in each cell, the number of violations (above) and the checking time (below).

Project	BT	BT_{CR}	LT	HICPP rules (*num. of violations*, checking time) 3.2.4	3.3.1	3.3.2	3.3.5	3.3.6	3.3.7
Bacula	76.1	78.9	68.5	*0* 0.01	*0* 0.00	*1* 0.00	*0* 0.00	*0* 0.00	*0* 0.00
IT++	307.4	338.7	300.4	*10* 4.40	*0* 0.00	*16* 0.12	*84* 44.43	*113* 25.08	*23* 9.21
PPL	296.3	477.2	235.5	*0* 1.91	*20* 0.00	*53* 0.09	*41* 53.93	*33* 2.64	*0* 0.07
dlib	8.6	10.7	41.3	*8* 0.98	*7* 0.00	*17* 0.13	*40* 34.69	*40* 17.32	*6* 0.80

Project	HICPP rules (*num. of violations*, checking time) 3.3.8	3.3.13	3.3.14	3.3.15	3.4.5	3.4.6
Bacula	*0* 0.00	*0* 0.01	*0* 0.01	*0* 0.00	*6* 0.00	*0* 0.00
IT++	*72* 116.95	*32* 7.48	*0* 0.25	*0* 0.02	*82* 0.19	*7* 0.09
PPL	*75* 97.38	*0* 0.97	*0* 0.08	*0* 0.00	*56* 0.08	*0* 0.04
dlib	*224* 49.15	*2* 0.95	*0* 0.04	*0* 0.02	*142* 0.19	*29* 0.05

them. Table 3 briefly describes the analyzed projects.

Table 4 reports the number of rule violations found on each project, along with time consumption information that helps assessing the feasibility of our approach for real projects. Experiments have been run in an Intel Mobile Core 2 Duo 1.20 GHz with 2 Gb RAM and projects are built with `-O2` optimization flag.

Build time increases when `-fipa-codingrules` is enabled to extract structural information during compilation. The time penalty is less than 5% in the case of Bacula, but more than 60% for PPL. In general, the slow down is more noticeable when templates are extensively used. Reported build time is small for dlib, in spite of its size, because of the simplicity of the build procedure, consisting in the generation of one single object file.

In general, it takes quite a long time for `checkrules` to load the data of a project (in most cases in the same order of magnitude of a complete build, with the exception of dlib for the aforementioned reason). This happens because GCC does not have the concept of global compilation for C++, and header files are compiled many times during a project build, generating a lot of redundant information.

On the other hand, checking time goes from a fraction of a second (i.e., cells

with 0.00 time) to over one minute. Empirically, rules where complex sub-goals appear negated are the ones which take longer. In fact, as there is no restriction on the computational complexity of the rules, execution time is potentially unbounded. Despite this, the observed performance is reasonable for a project-wide static analysis tool that is not meant to be run in every compilation.

Note that, despite the significant number of violations we found for many rules, HICPP 3.3.14 and 3.3.15 are not violated in those projects. One reason is that they deal with language features rarely used by programmers (the declaration of an assignment operator in abstract classes and repeated inheritance, respectively).

6 Conclusions and Future Work

We have presented a tool for structural coding rule validation where rules for C++ are formally defined by means of a declarative rule definition language. Users can define their own rules and the tool is seamlessly integrated into the work-flow of the developers. Basic information about programs is taken from the very same compiler (GCC) used to generate object code, which avoid inconsistencies.

Only about 20% of the rules in HICPP are purely structural. Implementing more rules requires modifying other parts of the compiler and gain access to syntactic information unavailable in the middle-end of GCC and to the results of sophisticated analyses performed by GCC in its optimisation steps. We plan to extend our rule definition language to support new logic formalisms that help in the definition of non-structural rules. The approach should be easily adaptable to other languages supported by GCC, which will require instrumentation of other front-ends.