A Coding Rule Conformance Checker Integrated into GCC

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

Universidad Politécnica de Madrid
Boadilla del Monte, Spain

Juan José Moreno-Navarro[1,3]

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

Álvaro Polo[4]

Telefónica I+D
Madrid, Spain

Abstract

Coding rules are often used in industry for codifying software best practices and avoiding the many hazardous constructions present in languages such as C or C++. Predictable and customisable tools are needed to automatically measure adherence to these practices. Many of the properties about software needed for rule conformance analysis are calculated by modern compilers. We present an extension of the GNU Compiler Collection (GCC) that flags those code fragments that do not conform to a given set of rules. The user can define coding rules using a high-level declarative language based on logic programming.

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

1 Introduction

Languages such as C or C++ need to be used in a disciplined manner, such that the hazards of its weaknesses and more error-prone features are minimised. To that

[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] Ministry of Education and Science grant TIN2005-09207-C03-01 (MERIT/COMVERS), and EU IST FET grant IST-15905 (MOBIUS).

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

[4] Email: apv@tid.es

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

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 [4]). MISRA-C [3] 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. An important requirement is that many organisations – or even projects – need to establish their own coding rule sets, or adapt the existing ones.

However, no matter who devises and dictates the coding rule set, for it to be of practical use, an automatic method to check code for conformance is needed. There exists a number of commercial compilers and quality assurance tools from vendors such as IAR Systems (`www.iar.com`) and Parasoft (`www.parasoft.com`) 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 (`www.klocwork.com`), define their own list of informally described checks aimed at avoiding hazards. But, in absence of a formal definition of rules, it is difficult to be certain about what they 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 [2] we propose 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 (at time being a subset of Prolog) will make it possible to easily capture the meaning of coding rules, constituting a more practical mechanism for user-defined rules than those provided by most of the tools. There have recently appeared other tools relying on high-level languages for defining code checks, such as Semmle Code (`www.semmle.com`), Klocwork Insight or Parasoft RuleWizard, but to our knowledge, ours is the first one based on logic programming.

In this work we present a new version of our coding rule checker that extracts all the needed information about C++ programs during compilation with the GNU Compiler Collection (GCC, `gcc.gnu.org`). We think that adding this feature to the day-to-day tool of thousands of developers will facilitate the adoption of coding rules in many projects. More importantly, as we rely for program analysis on the very same parser and semantic analysis used for object code generation, we avoid any possible discrepancy between both tools on how code is interpreted. Moreover, some static analyses already present in GCC are potentially reused. Both our modified version of GCC and the Prolog code necessary for rule checking are available at the website of the GlobalGCC project: `www.ggcc.info`.

2 Checking Structural Coding Rules

Our procedure for checking a given rule consists of three steps:

(i) Formalise the rule (in fact its violation) – that is written in plain English in standard rule sets – in (a subset of) Prolog. In the future we plan to facilitate this step with a completely declarative domain-specific language based on logic programming, with sorts, constructive negation, and appropriate quantifiers (find some details in [2]).

```
violate_hicpp_3_3_13(Caller,Called) :-
    has_member(SomeClass,Caller),
    (
        constructor(Caller)
    ;
        destructor(Caller)
    ),
    has_member(SomeClass,Called),
    virtual_member(Called),
    calls(Caller,Called).
```

```
violate_hicpp_3_3_15(A,B,C,D) :-
    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).
```

Figure 1. Prolog formalisation of some HICPP rules.

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

(iii) Put together both the rule violation predicate and those Prolog facts giving a high-level description of the software to be analysed. With a Prolog Engine – Ciao Prolog [1] – we seek for counterexamples for the rule.

This last step is performed with a command line tool called `checkrules` that contains the Prolog interpreter. The other two steps are detailed in the next corresponding sections.

3 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 3.3.13, that reads "*do not invoke virtual methods of the declared class in a constructor or destructor.*" The rationale behind this requirement is that member functions of the same object are always statically bound if called from a constructor or a destructor.

Another example is Rule 3.3.15 of HICPP that says: "*ensure base classes 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:

> 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.

Formalising 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 C++ predicates used for defining some rules, as the two given 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 Sect. 4. Fig. 1 shows the Prolog formalisation of the aforementioned HICPP rules.

4 Using GCC for Gathering Program Information

The middle-end (ME) of GCC contains a set of lowering and optimisation passes that are independent from both the compiled language and the target architecture.

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

Predicate	Meaning
$immediate\_base\_of(a, b)$	Class b directly inherits from a.
$base\_of(a, b)$	Transitive 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.
$declares\_member(a, m)$	Class a (re-)declares a member m.
$has\_member(c, m)$	Class c has defined a member m (data or function). m can be inherited from a base class.
$constructor(c)$, $destructor(d)$	Member c is a constructor (destructor).
$virtual\_member(v)$	Member function v is dynamically dispatched.
$calls(a, b)$	(Member) function a has in its text an invocation of (member) function b.

Since it is simple and clean to add a new pass to the ME, and no overhead exists if the pass is not enabled with the corresponding flag, this has been our first step towards a facility for program feature extraction integrated into GCC.

Some of the program features needed for writing rules are common to many languages and have a representation in the ME – even if the semantics is not exactly the same in different languages. Other properties or constructs (e.g. templates and friends in C++) are language-specific. We plan to also instrument the C++ front-end in the future (and maybe other front-ends).

Our pass writes to a file Prolog facts describing structural properties of the analysed software, that can be either a program or a library. All the source files in a project have to be analysed because structural rules involve project-wide properties. This is carried out by relying on the building process used in the project (e.g. `make`, `scons`, `ant`, etc.) and accumulating all the Prolog facts of different compilation units in a single file, that 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`, `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. They resolve, among other possible name clashes, overloaded function names, including overloading originated by templates. There is also a naming scheme for local entities (local variables, function arguments, etc.) not described here.

Following this identification scheme, a Prolog predicate exists for every relevant property of global entities, and terms are generated in the output for every occur-

Table 2
Structure of some 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)))
```

rence of the property. These terms have the structure shown in Table 2 for some example properties: `virtual` and `accessibility`. Besides individual properties, relations among global entities exist. Some examples of binary relations among global entities can also be found in Table 2: `contains`, `enumerates`, and `extends`. Relations such as *extends* can also have properties attached, like `virtual`.

Some terms are generated for representing types and attaching types to entities, and also for associating code locations to entities (needed for user output).

The predicates used in Table 1 are defined on top of the predicates described in this section. Those predicates are of a higher level, closer to abstractions used in defining coding rules. They follow the usual terminology used for talking about C++ programs (*base classes*, *member functions*, etc.), facilitating the formalisation of the rules for a C++ expert.

5 Conclusions

We present 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 used to generate object code, which avoid inconsistencies. Only about 20% of the rules in HICPP are purely structural. In order to implement more rules we need to modify other parts of the compiler and gain access to syntactic information unavailable in the ME, and to the result 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 adapted to other languages supported by GCC.

References

[1] Hermenegildo, M., F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García and G. Puebla, *The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems*, in: *Parallelism and Implementation of Logic and Constraint Logic Programming*, Nova Science, Commack, NY, USA, 1999 pp. 65–85.

[2] Marpons-Ucero, G., J. Mariño-Carballo, M. Carro, Á. Herranz-Nieva, J. J. Moreno-Navarro and L.-Å. Fredlund, *Automatic coding rule conformance checking using logic programming*, in: P. Hudak and D. S. Warren, editors, *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science **4902** (2008), pp. 18–34.

[3] MIRA Ltd., "MISRA-C:2004. Guidelines for the Use of the C Language in Critical Systems," (2004).

[4] The Programming Research Group, "High-Integrity C++ Coding Standard Manual," (2004).