facultad de informática

universidad politécnica de madrid

Automatic Coding Rule Conformance Checking Using Logic Programming

Guillem Marpons
Julio Mariño
Manuel Carro
Ángel Herranz
Juan José Moreno-Navarro
Lars-Åke Fredlund

TR Number CLIP6/2007.0

Printing date: 1st September 2007

Automatic Coding Rule Conformance Checking Using Logic Programming

Technical Report Number: CLIP6/2007.0

Printing date: 1st September 2007

Keywords

Coding rule checking, Declarative domain-specific languages and applications, Logic programming, Programming environments.

Acknowledgements

Work partially supported by PROFIT grants FIT-340005-2007-7 and FIT-350400-2006-44 from the Spanish Ministry of Industry, Comunidad Autónoma de Madrid grant S-0505/TIC/0407 (PROMESAS), Ministry of Education and Science grant TIN2005-09207-C03-01 (MERIT/COMVERS) and EU IST FET grant IST-15905 (MOBIUS).

Abstract

An extended practise in Software Engineering and programming in industry is the application of *coding rules*. Coding rules are customarily used to constrain the use (or abuse) of certain constructions in a programming language. However, these rules are usually written using natural language, which is intrinsically ambiguous and which may complicate its use and hinder their automatic application. This paper presents some early work aiming at defining a framework to formalise and check for coding rule conformance using logic programming. We show how a certain class of rules – *structural* rules – can be reformulated as logic programs, which provides both a framework for formal specification and also for automatic conformance checking using a Prolog engine. Some examples of real rules are discussed, along with the corresponding Prolog code. Experimental data regarding the practicality and impact of their application to real-life software projects is presented and discussed. Also, a domain-specific language for specifying rules is proposed based on expressiveness and efficiency considerations.

Resumen

Una práctica industrialmente extendida en la Ingeniería del Software es el uso de *reglas de codificación*. Éstas se usan usualmente para restringir el uso (o abuso) de determinadas construcciones en un lenguaje de programación. Esas reglas se escriben normalmente en lenguaje natural, que es intrínsicamente ambiguo y que puede hacer complejo su uso e impedir su aplicación automática. Este artículo presenta un trabajo inicial dirigido a la definición de un entorno para formalizar y comprobar, utilizando programación lógica, la adecuación de código existente a reglas. Veremos como una clase de reglas (reglas *estructurales*) pueden formularse como programas lógicos, lo que da un entorno para su especificación formal y para la comprobación automática de adecuación utilizando un intérprete o compilador de Prolog. Se muestran ejemplos de reglas junto con el programa Prolog correspondiente, así como datos empíricos sobre la aplicabilidad de dichas reglas en casos reales. Adicionalmente, se propone un lenguaje específico del dominio para escribir reglas, en el cual se toman en cuenta consideraciones de expresividad y eficiencia.

Contents

1 Introduction

Although there is a trend towards increased use of higher-level languages in the software industry, offering convenient programming constructs such as type-safe execution, automatic garbage collection, etc., it is equally clear that more traditional programming languages like C (which is notorious for fostering dubious practises) remain very popular. The reasons for this range from efficiency concerns (especially on embedded platforms), need for compatibility with existing products, or the simple fact that a huge amount of acceptably trained programmers are available.

However, a good practise of a language like C involves using the language in a disciplined manner, such that the hazards of its weaknesses and more error-prone features are minimised. To that end, it is common to require that code rely only on a well-defined subset of the language, following a set of coding rules. For C, for example, MISRA-C [1], elaborated by The Motor Industry Software Reliability Association (MISRA, mainly fostered by the British automotive industry, but later applied to other realms) is one of the leading initiatives. MISRA-C contains a list of 141 coding rules aimed at C code for critical systems. Examples of typical coding rules for C are that "*all automatic variables shall have been assigned a value before being used*" and "*functions shall not call themselves, either directly or indirectly.*" For C++ no equally accepted set of coding rules exists, but a notable initiative is *High-Integrity C++* (HICPP [2]) which provides around a hundred code rules for C++.

Another use of coding rules is to enforce domain specific language restrictions. Java Card [3], for example, is a subset of Java for programming Smart Cards. In such an environment memory is scarce, and coding rules typically forbid language constructs that lead to heavy memory usage.

Each organisation – or even project – can establish its own coding rule sets. However, for these to be of practical use, an automatic method is needed to check code for coding rule conformance.[1] Added to the intrinsic difficulty of mechanically checking rules, they are typically described using (necessarily ambiguous) natural language, which shifts the difficulty of interpreting them to whoever implements the checking tool. Still there exists a number of commercial quality assurance tools from vendors such as IAR Systems (`http://www.iar.com`) and Parasoft (`http://www.parasoft.com`) that claim to be able to check code for MISRA-C compliance. 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.

This paper presents a framework to precisely specify rule sets such as MISRA-C and to, later, automatically check (non-trivial) software projects for conformity. In the rule-coder side, a logic-based language will make it possible to easily capture the meaning of coding rules; this language will be compiled into a Prolog program with which code conformity is checked.

This work is developed within the scope of the Global GCC project (GGCC, [4]), a consortium of European industrial corporations and research labs funded under the Eureka/ITEA Programme. GGCC aims at extending the free GNU Compiler Collection (GCC) with project-wide optimisation and compilation capabilities (Global GCC). In the context of GGCC, we seek the inclusion of a facility to define new sets of coding rules, and providing mechanisms to check code compliance using a logic programming engine and making heavy use of the static analysers and syntax tools already present in GCC. Such a mechanism for extensibility is a requirement for many organisations of the GGCC Consortium having their own coding policies or the necessity to adapt existing ones.

Since GCC is a multi-language compilation framework, it is natural to provide support to express coding rules for different target languages. We have initially focused on C, C++, and Java since they are widely used in industry, in particular by the industrial partners of the GGCC project. Throughout the paper, however, we will only consider C++ (and HICPP).

The rest of the paper is structured as follows: Section 2 contains a classification of coding rules. An initial framework for defining and checking structural rules is presented in Sect. 3 using some examples and highlighting certain constructs that often occur in the rules. A formal description of a rule specification language is given in Sect. 4. Experimental results obtained with a small

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

prototype are presented in Sect. 5. Section 6 comments on related work and Sect. 7 concludes.

2 A Classification of Coding Rules

In order to make a first classification of coding rules to have an initial idea of the difficulty of the task and the type of code schemata we will have to deal with, a survey was conducted within the project partners asking for examples of coding rules internally followed in their organisations. This gave us clues about which types of rules were actually perceived as interesting in order to focus primarily on them. We analysed in parallel in some detail MISRA-C and HICPP, which resulted in a categorisation of coding rules which roughly ranks the difficulty of formalising them and of verifying they are met:

Trivial. The appearance in the source code of a simple pattern that can be expressed with a regular expression violates the rule. E.g.: "*Do not call the* `malloc()` *function*" (MISRA-C, rule 20.4).

Syntactic. Slightly modifying the grammar (e.g., by eliminating productions) or the lexical analysis, is enough to catch violations to the rule. E.g.: "*Do not use the 'inline' keyword for member functions*" (HICPP, rule 3.1.7).

Type-enforceable. An extended type system is needed to deal with it. E.g.: "*Expressions that are effectively Boolean should not be used as operands to operators other than (*`&&`*,* `||` *and* `!`*)*" (MISRA-C, rule 12.6).

Structural. The rule has to do with permanent relations between objects in the code. Information not present in the Abstract Syntax Tree (AST) but completely static, such as the inheritance hierarchy, needs to be analysed. E.g.: "*If a virtual function in a base class is not overridden in any derived class, then make it non virtual*" (HICPP, rule 3.3.6).

Dynamic. The rule refers to sequences of events occurring at runtime. Control flow graph information is typically taken into account, but many other things might be necessary, as a memory model, pointer alias, or data-flow information. E.g.: "*All automatic variables shall have been assigned a value before being used*" (MISRA-C, rule 9.1). By its nature, only approximated information can be statically deduced for the kind of problems involved in automating these rules.

Hard to automate. Either the rule is difficult to formalise or it involves non-computable properties: for instance, that two procedures compute the same function. E.g.: "*Behaviour should be implemented by only one member function in a class*" (HICPP, rule 3.1.9).

As it is clear that very different verification techniques are required to deal with different classes of coding rules, we have decided to focus first on *structural* rules, i.e., those that depend on static relations among objects in the code. On one hand, these were perceived as interesting by industrial project partners and, on the other side, a customary research on the literature threw the result that these aspects of software construction had not been treated with the depth they deserved. More than 20 rules of this kind have been detected in HICPP and MISRA-C.

It is interesting to note that, with the exception of the last category, a significant part of the data structures needed to deal with these rules are already constructed by any modern compiler, in particular GCC. Needless to say, rules of a certain category may need machinery characteristic of previous categories.

3 A Framework for Structural Rule Validation

We will introduce in this section the core idea underlying the framework, and we will give some extended examples of how this idea can be applied in practise.

3.1 Coding Rules, Program Representation, and Logic Programming

We will later (Sect. 4) define the core of a Domain-Specific Language (DSL) to make formalising coding rules easier. We think, however, that it is more illustrative to proceed from the initial idea outwards. Our selection of the basic toolkit stems from the observation that, in general, structural coding rules are not very complex (in a linguistic sense), they do not need to deal with time, and they do not need to model beliefs or approximate reasoning, either – in other words, first order logic increased with sorts (as we will see later) seems a well-known and completely adequate formalism.

Detecting whether some software project violates a particular rule can be made as follows:

1. Write the coding rule in logic, assuming an appropriate representation of the entities the rule needs to know about. This is a one-time step.

2. Transcribe the necessary program information into the aforementioned representation. This is necessary for every project instance.

3. Prove (automatically) that there is (or there is not) a counterexample for the rule. If there is such a counterexample, the rule is not met; otherwise, the code conforms to the rule.

We will, for the moment, directly transcribe the *violations* of coding rules as Prolog predicates, its arguments being the entities of interest to the programmer, because in this way the verification method can return *pointers* to the elements in the source code which have to be corrected. Furthermore, for many rules coding its negated form as logical formulæ results more natural.

Accordingly, the knowledge base which represents a particular software project is stored as facts of a Prolog program. An answer to a query to this predicate will flag a violation of the corresponding rule, and the culprits will be returned in the form of bindings for the predicate arguments. On the other hand, failure to return an answer means that the project conforms to that particular rule.

The codification of the rules requires a set of language-specific predicates representing structural information about, e.g., the inheritance graph of the checked program, its call graph, etc. We use a *order sorted logic* [5] to define these predicates in order to categorise different concepts of the language. Sorts in Prolog are implemented as unary predicates, which is not suitable for constructive negation and the meaning we want to give to quantifiers, as will be seen in Sect. 3.4.

Some representative predicates targeting C++ (in particular those used in the next rule examples), and a significant fraction of the sorts relevant to them are listed in Table 1. Some predicates appear categorised as *primitive*: they concern base facts that have to be provided by the compiler (i.e. GCC) in the process of compiling a source program. Note that, in general, processing a single compilation unit is not enough: extracting information pertaining a full program or library is required for the analysis we are aiming at. More sophisticated predicates can be constructed in terms of primitive predicates: some examples are given in the table in the *derived* predicates section.

In what follows we will examine some examples of formalisations of some actual rules of HICPP using logic programming.

3.2 Some Examples of HICPP Rule Formalisation

Rule 3.3.15 of HICPP reads "*ensure base classes common to more than one derived class are virtual*". This can be interpreted as requiring that all classes with more than one immediate descendant class are virtually derived, which seems far too restrictive. In the justification that accompanies the rule, it is made clear that the problem concerns repeated inheritance only (that is, when a replicated base class is not declared virtual in some of the paths). Whether all paths need to use virtual inheritance, or only one of them, is difficult to infer from the provided explanation and examples. This kind of ambiguity in natural language definitions of coding rules is not uncommon, and is a strong argument in favour of providing, as we do, a formalised rule definition amenable to be used by an automatic checker.

PREDICATE	MEANING
Sorts	
$class(C)$	C is a class.
$method(M)$	M is a member function.
$type(T)$	T is a C++ well-formed type.
$class\_template(CT)$	CT is a class template
$class\_template\_instance(CTI)$	CTI is a class template instance.
$identifier(I)$	I is a class, method or template identifier.
Primitive predicates	
$immediate\_base\_of(a : C, b : C)$	Class a appears in the list of explicit base classes of class b.
$public\_base\_of(a : C, b : C)$	Class b immediately inherits from class a with public accessibility. There are analogous predicates for other accessibility choices and also for virtual inheritance.
$has\_method(c : C, m : M)$	Class c has defined the m method.
$sig(m : M, i : I, s : T, a : [T], r : T)$	Method m has name i, self type s, argument types a and result type r.
$constructor(c : M)$	Method c is a constructor.
$destructor(d : M)$	Method d is a destructor.
$virtual(v : M)$	Method v is dynamically dispatched.
$calls(a : M, b : M)$	Method a has in its text an invocation of method b.
Derived predicates	
$base\_of(a : C, b : C)$	Transitive closure of $immediate\_base\_of/2$.
$inheritance\_path(a : C, b : C, p : [C])$	Class b can be reached from class a through the inheritance path p.
$in\_call\_graph\_of(a : M, b : M)$	Transitive closure of $calls/2$.

Table 1: Sorts and predicates necessary to describe structural relations in C++ code. Sorts of predicate arguments are abbreviated: C for *class* sort, M for *method*, etc. $[S]$ means a list of elements of sort S.

The C++ definition of virtual inheritance makes clear that, in order to avoid any ambiguities in classes that repeatedly inherit from a certain base class, all inheritance paths must include the repeated class as a virtual base. As we want to identify violations of the rule, a reformulation is the following:

> Property 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 declared as a virtual base of C.

The formalisation in Prolog of the above property appears in Fig. 1. The success of a query to `violate_hicpp_3_3_15/4` would exemplify an example of violation of HICPP Rule 3.3.15.[2]

The fact that all variables refer to classes is codified at the moment – as Prolog is used – with a predicate `class/1`. The four classes are not necessarily distinct, but it is required that `B` and `C` do not refer to the same class, and that both are immediate descendants of `A`. The terms `base_of(B,D)` and `base_of(C,D)` point out that class `D` must be a descendant of both `B` and `C`, through an arbitrary number of subclassing steps. Finally, for the rule to be violated we require that class `A` is not virtually inherited by class `C`. Use of negation in rules is further developed in Sect. 3.4.

[2]We will use a similar naming convention hereinafter: `violate_hicpp_X_Y_Z/N` is the rule which models the violation of the HICPP rule X.Y.Z.

```
violate_hicpp_3_3_15(A, B, C, D) :-
    class(A), class(B), class(C), class(D),
    B \= C,
    immediate_base_of(A, B), immediate_base_of(A, C),
    base_of(B, D), base_of(C, D),
    \+ virtual_base_of(A, C).
```

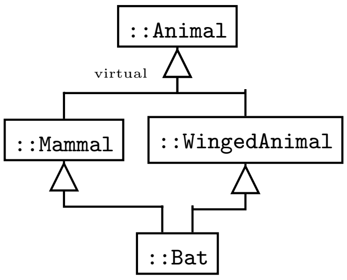
```
violate_hicpp_3_3_13(Caller, Called) :-
    method(Caller), method(Called),
    has_method(SomeClass, Caller),
    (
        constructor(Caller)
    ;
        destructor(Caller)
    ),
    has_method(SomeClass, Called),
    virtual(Called),
    calls(Caller, Called).
```

```
violates_hicpp_3_3_2(BaseClass) :-
    class(BaseClass),
    exists_some_derived_class_of(BaseClass),
    does_not_exist_virtual_destructor_in(BaseClass).

exists_some_derived_class_of(BaseClass) :-
    immediate_base_of(BaseClass, _).

does_not_exist_virtual_destructor_in(Class) :-
    \+ (
          has_method(Class, Destructor),
          destructor(Destructor),
          virtual(Destructor)
        ).
```

```
violate_hicpp_16_2(ClassTemplate) :-
    class_template(ClassTemplate),
    has_method(ClassTemplate, Method1),
    has_method(ClassTemplate, Method2),
    Method1 \== Method2,
    sig(Method1, Name, SelfT, ArgsT, ResT),
    sig(Method2, Name, SelfT, ArgsT, ResT).
```

```
violate_hicpp_3_3_1(Base, Derived) :-
    class(Base), class(Derived),
    (
        private_base_of(Base, Derived)
    ;
        protected_base_of(Base, Derived)
    ).
```

Figure 1: Formalisation of some HICPP rules.



```
class('::Animal').
class('::Mammal').
class('::WingedAnimal').
class('::Bat').
immediate_base_of('::Animal','::Mammal').
immediate_base_of('::Animal','::WingedAnimal').
immediate_base_of('::Mammal','::Bat').
immediate_base_of('::WingedAnimal','::Bat').
virtual_base_of('::Animal','::Mammal').
```

Figure 2: Violation of rule HICPP 3.3.15 and automatically generated Prolog facts.

Figure 2 depicts a set of classes and their inheritance relations which make the goal `violate_hicpp_3_3_15('::Animal` deducible using the rule, thus constituting a counterexample thereof. If inheritance from `'::Animal'` to `'::WingedAnimal'` is also *virtual*, the goal would fail (no counterexample could has been found).

Note that the rule is more general than it may seem: for example, it does not require that classes `B` and `D` are different. Thus, if (nonsensically) `'::Mammal'` and `'::Bat'` in Fig. 2 were the same, a property violation would still be detected.

Another rule that can be easily implemented in this framework but requires disjunction is rule HICPP 3.3.13 specified as "*do not invoke virtual methods of the declared class in a constructor or destructor*". This rule needs, additionally, information about the call graph of the project.

Rule HICPP 3.3.2 says "*write a 'virtual' destructor for base classes*". The rationale behind this requirement is that if an object will ever be destroyed through a pointer to its base class, the correct version of the destructor code should be dynamically dispatched. This rule illustrates the necessity of existential quantification; also, the construction "does not exist" appears repeatedly in rule formalisation. Some hints about quantification are also provided in Sect. 3.4.

```
template< typename T > class A {
  public:
  void foo( T );
  void foo( int );
};

template class A< int >;
// void foo(int) declared twice!
```

Figure 3: Violation of rule HICPP 16.2.

Rule HICPP 16.2 reads "*do not define a class template with potentially conflicting methods*". The code snippet in Fig. 3, taken from [2], illustrates how not following the rule can hamper software maintenance and reusability: the template instantiation generates two methods with identical signature.

A formalisation of the rule negation (and, in fact, a program capable of catching non-compliant code) can be easily written using unification and logic variables to represent template parameters (see Sect. 3.3 for more details), as shown in predicate `violate_hicpp_16_2/1` (Fig. 1).

Syntactic rules are also trivially expressible in this framework provided that enough information about the abstract syntax tree is reified into the knowledge base about the program, even if more efficient ways of dealing with that rules exist. Predicate `violate_hicpp_3_3_1/2` (Fig. 1) shows a Prolog representation of rule HICPP 3.3.1, that reads "*use public derivation only.*"

HICPP has a conservative spirit which we want to retain in our rules: any potentially dangerous situation is altogether forbidden, regardless of whether it can actually happen in a given environment (i.e., in a *complete* program). This is very sensible: on one hand, from the point of view of program maintenance, initial conditions may evolve in unforeseeable directions, and protecting the project is very reasonable; on the other hand, when libraries are being developed, these will be used in an *a priori* unknown environment. Besides what HICPP dictates, if more accuracy is required, global analysis is in general needed to detect which situations can actually happen in a program.

The following sections look more in depth to the Prolog translation of rules.

3.3 Types, Classes, and Generic Constructs (Templates)

Parametric polymorphism is realised in C++ by means of *templates.* We are not dealing, at the moment, with a complete representation of templates, but only with those whose parameters are classes and basic types. The names of the parametrised classes are introduced in the knowledge base as function symbols belonging to the `class` sort. Template parameters are modelled with logic variables, which makes it possible to deal with template instantiation directly by means of variable instantiation. It also makes the power of constraint logic programming available to write coding rule violations with a richer language. This turns out to be necessary since the possibility of arbitrary instances of a template makes the cardinality of the `class/1` sort to become infinite. A constructive use of `type/1` to *generate and test* class instances (in order to verify, e.g., HICPP 16.2) can very well lead to an infinite failure or to infinite recursions. These can in many cases be worked around, if necessary, by using tabled resolution [6]. A correct treatment of disequality constraints needs, however, dealing explicitly with negation of non-ground goals.

3.4 Negation and Quantifiers

Negation appears in coding rules in various forms. On one hand, there is the issue, already mentioned, that predicates specifying the violation of rules (i.e. their negation) tend to be easier to translate into Prolog. But this is not necessarily the case for every structural rule, so some method for automatically generating one version from the other would be very useful. Also, we have seen that disequalities and predicates representing the complement of basic relations occur frequently in the rules.

Our choice has been to use our own implementation of *intensional negation* [7], as it deals properly with the kind of knowledge bases that we are generating from the projects code – positive and potentially infinite. Negative constructive answers are possible thanks to a disequality constraint library replacing standard Prolog disequality.

The case of (universal) quantification is similar. It appears both explicitly (in the rules specification) and also implicitly (generated during the transformation implementing intensional negation of clauses with free variables). Fortunately, our intensional negation library comes equipped with a universal quantification mechanism which is semantically soundand complete for bases satisfying certain conditions.

Finally, for this application, both negation and universal quantification must be implemented in a *sort aware* way, i.e. negated predicates must represent the complement of relations w.r.t. their sorts and the universal quantification of a predicate must hold if, and only if, it holds for every element of its sort, etc. This precludes a naive treatment of sorts as regular Prolog predicates. For example, looking at typical rules like HICPP 3.3.15 or HICPP 16.2, whose Prolog formalisations are shown in Fig. 1, we see that both clauses start with some sort requirements (`class(A), class(B), ...`, etc.). A naive negation of such clauses would not produce the right code. In general, if we have a clause of the form

$$\mathit{rule\_n}(\overline{X}) \leftarrow \mathit{sorts}(\overline{X}) \wedge p(\overline{X})$$

its negation will correspond to

$$\mathit{viol\_rule\_n}(\overline{X}) \leftarrow \mathit{sorts}(\overline{X}) \wedge \mathit{neg\_p}(\overline{X})$$

where p is the part of the clause not containing any sort requirements and *neg_p* has been obtained using an standard intensional negation transformation. Notice how *sorts* remains positive in the negated version.

4 A Domain-specific Language for Coding Rules

While, as we have just seen, Prolog can be used to express structural rules, its use as the only language to specify coding rules may not be a good idea, for several reasons: people writing or

Signatures:

$$\begin{array}{rcl} \Sigma_L & = & \langle S_L, B_L, FS_L \rangle \\ \Sigma & = & \langle S_L, PS, FS_L \cup FS_P \rangle \\ PS & = & B_L \cup \{=, \neq\} \cup DP \end{array}$$

Syntax:

$$\begin{array}{llcll} (\textsc{Programs}) & P & ::= & \mathcal{S}et(C) & \\ (\textsc{Clauses}) & C & ::= & A[\leftarrow G] & \\ (\textsc{Atoms}) & A & ::= & R(T_1, \ldots, T_n) & R \in PS, T \in Term(\Sigma, \mathcal{X}) \\ (\textsc{Goals}) & G & ::= & A \mid G \wedge G \mid G \vee G \mid \neg G \mid QX.G & \\ (\textsc{Quantifiers}) & Q & ::= & \forall \mid \exists \mid \exists! \mid \exists^{>1} & \end{array}$$

Figure 4: Abstract syntax of CRISP$_L$ rule sets.

$$\begin{array}{rcll} \mathit{violates\_hicpp\_3.3.2}(b) & \leftarrow & \exists c.\mathit{immediate\_base\_of}(b, c) & \wedge \\ & & \neg\exists m.(\mathit{has\_method}(b, m) \wedge \mathit{destructor}(m) \wedge \mathit{virtual}(m)) & \\ \mathit{violates\_hicpp\_3.3.13}(m, m') & \leftarrow & \exists c.(\mathit{has\_method}(c, m) \wedge \mathit{has\_method}(c, m')) & \wedge \\ & & (\mathit{constructor}(m) \vee \mathit{destructor}(m)) & \wedge \\ & & \mathit{calls}(m, m') \wedge \mathit{virtual}(m') & \end{array}$$

Figure 5: Some CRISP$_{\mathrm{C++}}$ specifications for the previous examples. Observe how sort information for defined predicates can be deduced from the sorts of primitive ones.

reading the rules are not likely to be Prolog experts, full Prolog contains too much extra (perhaps non-declarative) stuff that does not fit in our setting, etc. Moreover, as mentioned above, proper compilation of rules into Prolog demands a careful use of several extensions to the language. As an alternative to the direct use of Prolog we are therefore experimenting with a Domain-Specific Language for structural coding rules, maximising declarativeness and focusing on those constructions that appear more often.

Abstract syntax for such a language is sketched in Fig. 4. The language is called CRISP[3] and in fact is a family of languages (CRISP$_L$) parameterised by the actual programming language (L). Σ_L is the language-dependent signature, including a set of sorts (S_L), a set of (sorted) primitive relation symbols (B_L), and a set of function symbols (FS_L). The signature for writing the rules is Σ, that extends the set of primitive relation symbols with equality, disequality, user-definable relations and program-dependent function symbols (FS_P) – used to represent non-flat domains like types or instances of generic constructions (templates, etc). The grammar for programs (or rule sets) is that of a highly sugared logic programming language where the aforementioned issues regarding sorts, negation and quantification are conveniently hidden to the specifier. Figure 5 shows some of our test rules specified in CRISP$_{\mathrm{C++}}$.

Both the final language definition and its desugaring into actual Prolog code are matter of current research and are subject to improvements depending on further experiments in rule specification and efficiency considerations.

5 Experimental Results

We have developed a prototype that allows for implementing some syntactic and structural coding rules and running them over a C++ source code tree, reporting rule violations.[4] Some of the rules described in Sect. 3.2 have been applied to the C++ open-source software projects appearing in Table 2.

The tool has been constructed on top of the Prolog engine Ciao [8]. Until full integration into GCC pipeline (which is a hard work to undertake) is done, we take advantage of a source

[3]CRISP is an acronym for "Coding Rules In Sugared Prolog."

[4]The source code of the prototype is available at `https://babel.ls.fi.upm.es/trac/ggcc/wiki#documents`.

Project	Description
Bacula	A network based backup program.
CLAM	C++ Library for Audio and Music.
Firebird	Relational database with concurrency and stored procedures.
IT++	Library for signal and speech processing.
OGRE	Object-Oriented 3D Graphics Rendering Engine, scene-oriented.
Orca	Framework for developing component-based robotic systems.
Qt	Application development framework and GUI widgets library.

Table 2: A brief description of the open-source C++ projects that have been analysed for rule violations.

project	Bacula	CLAM	Firebird	IT++	OGRE	Orca	Qt
rule \ Kloc	20	46	439	39	153	89	595
3.3.1	0 (0)	1 (0)	16 (0)	0 (0)	0 (0)	1 (0)	9 (0)
3.3.2	3 (0.03)	15 (0.64)	58 (1.07)	6 (0.04)	10 (0.56)	12 (0.17)	76 (11.11)
3.3.15	0 (0)	0 (0.16)	0 (0.2)	0 (0)	0/1 (0.09)	0 (0.12)	4/5 (1.16)

Table 3: Experimental results on a sample of HICPP coding rules and open-source projects. Cells show *[no. of true violations /] no. of violations found (running time in seconds)*. Kloc measured by `sloccont`.

code analysing tool named Source-Navigator [9], that generates a database of architectural and cross-reference information of a software project. The prototype queries this database to extract the necessary facts from the analysed software and implements the primitive predicates described in Table 1. On top of this, many structural are currently written, at the time being in Prolog, as some characteristics of CRISP are under fine adjustment. Nevertheless, the prototype has been useful to test the feasibility of the overall approach. Even if the necessary infrastructure for Ciao to be able to handle the sugared language and its automatic translation into Prolog is still to be completed, some interesting preliminary results are worth commenting.

Table 3 shows, for each software project, the number of violations automatically detected for each implemented rule. A measure of the size of each project is also provided in the form of physical lines of code. They all can be considered mature projects but diverge in its size and application area, covering as a whole a wide range of C++ design techniques. Some of them are final applications, and others are libraries of reusable components. Testing code and examples included in many projects have been excluded from the analysis wherever possible.[5]

As the implemented rules can be completely determined statically, caught violations have revealed themselves as true violations excepting two cases: those spurious HICPP 3.3.15 violations detected in Ogre and Qt. They are consequence of our prototype getting confused by the existence of different classes with the same (fully qualified) name in different header files. Presumably, the intent of the library developers is that only one of those header files gets included in a particular project, which is also a dubious API design but not a violation of the rule being checked.

The fact that rule HICPP 3.3.15 is violated by only one project (Qt) is a direct consequence of the close to zero use of repeated ("diamond-like") inheritance in actual C++ projects. The same infrastructure devoted to check coding rules compliance has been used to detect multiple inheritance and repeated inheritance instances. The conclusion is that multiple inheritance – an important object-oriented design and implementation mechanism – is used very rarely, and the only analysed projects that have taken advantage of repeated inheritance are IT++ and Qt, and even there is applied very few times. Despite the efforts done to include those features into the language [10], its hazards and subtleties seem to have convinced many developers that they have to be completely avoided. Rule HICPP 3.3.15 (in combination with other HICPP rules) is

[5]In the case of Ogre, limitations in our prototype have forced us to do without the complementary tools and the plug-in infrastructure.

oriented towards permitting a reliable use of multiple and repeated inheritance. A wider adoption of coding standards like HICPP could have the paradoxical effect of popularising some denigrated – but useful – C++ features.

The other interesting aspect of these first experiments is that they have confirmed that relying on manual checking only is not a feasible approach to enforce coding rules. Manually approve or reject those violations reported by the checking tool has turn out to be a too tedious and error-prone task. Even for simple rules as HICPP 3.3.2 (easy to state and to program in CRISP), rule requirements can be fulfilled in many ways, some of them not easy to grasp from the sources. In this particular case a virtual destructor has to be looked for in all the superclasses of a given class.

Execution times have been included in Table 3 to show that a non-negligible time is used to run non-trivial rule violation detectors over the biggest projects, but they are still bearable – as far as rule enforcing machinery is not expected to run in every compilation.

6 Related Work

To our knowledge, our proposal is the first attempt at using declarative technology for formally specifying and automatically checking coding rules. A related area where some academic proposals exist that apply first-order logic and LP is formalisation and automatic verification of design patterns [11, 12, 13]. In [13], facts about a Smalltalk program are reified into a logic programming framework. In [12] a very similar setting is developed targeting the Java language. Both formalisms can deal with the structural relations necessary to define the static aspects of design patterns. But none of them use true sorts for quantification nor can cope with infinite domains or recursively defined objects in the target language. This fact makes both approaches unable to represent C++ templates, not to mention reasoning about hypothetical instantiations of templates as we do in rule HICPP 16.2. A different area where some interesting ideas can be borrowed from is automatic bug finding techniques. The "bug pattern" term in [14] is very close to our concept of what a rule violation is. It uses structural information, but no mechanism is provided for the user to define its own bug patterns. On the other hand, [15, 16] define a DSL to create custom checks for C code, and [17] uses a declarative language for checks on Java. All three are based on automata and syntactic patterns, and are specially oriented to the kind of program properties related with dynamic rules. The language in [16] is the less expressive of the three but, interestingly, the checking facility has been integrated into a GCC branch.

As mentioned in the Introduction, existing industrial tools for checking coding rule conformance are not extensible and cannot be considered semantically reliable due to the absence of a formal specification of their intended behaviour.

7 Conclusion

This paper presents a logic programming-based framework to specify industrial coding rules and use them to check code for conformance. These coding rules express what are perceived as good programming practises in imperative/object-oriented programming languages. Our framework is in principle language-agnostic, and the particular characteristics of a given language (kinds of inheritance, etc.) can be modelled seamlessly and with little effort.

The properties we tackle range from syntactic to semantic ones, although in this paper we have focused on the so-called "structural properties", which address programming projects as a whole and have as basic components entities such as classes, methods, functions, and their relations.

In order to make it possible to easily express coding rules, we also propose CRISP, a highly enriched logic-based DSL, aimed at bridging the gap between the expressive but ambiguous natural language and the rigorous, but somewhat more constraining language of formal logic. The declarative nature of CRISP makes it possible to give it a clear semantics and to efficiently compile it to a logic inference engine. On the other hand, its design for the ground up is aimed at making it easy to understand and to code with for people with no previous knowledge of logic programming, as its building blocks are those an object-oriented programmer should already be familiar with: classes, methods, inheritance, etc.

The inference engine we are currently using to perform rule violation detection is plain Prolog, which can be queried to discover witnesses of evil patterns. Speed is, so far, very satisfactory, and we have been able to run non-trivial rules in projects with half a million LOC in around ten seconds using a standard PC. We have specified a good number of coding rules, of which we have selected what we think is a relevant and interesting subset. As expected, the main problem is not in the coding itself, but in understanding clearly what is the exact meaning of the rule. This is, of course, part of the *raison d'être* of the coding rule formalism. Tests have shown rule violations in very well-known and well-regarded projects, although it is true that the amount of violations, if we take into account the size of the projects, is relatively small.

This work is part of a bigger project which is just giving its first results and whose final aim is to be delivered as part of the GCC suite. Future directions for research include: a full translation scheme from CRISP into (extended) Prolog, connecting the framework with other parts of the GGCC project (e.g. static analysis) in order to cover more complex rules and, of course, gaining more practical experience both in terms of expressiveness and performance when analysing very large projects. Regarding the latter, there seems to be some room for improvement by discovering recurring patterns – e.g. the fact that transitive closures appear very often in coding rules suggests a potential for tabling, etc.

Bibliography

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

[2] The Programming Research Group: High-Integrity C++ Coding Standard Manual. (May 2004)

[3] Sun Microsystems: http://java.sun.com/products/javacard/

[4] Global GCC project website: http://www.ggcc.info/

[5] Kaneiwa, K.: Order-Sorted Logic Programming with Predicate Hierarchy. Artificial Intelligence **158**(2) (2004) 155–188

[6] Ramakrishnan, I.V., Rao, P., Sagonas, K.F., Swift, T., Warren, D.S.: Efficient tabling mechanisms for logic programs. In: ICLP. (1995) 697–711

[7] Muñoz, S., Mariño, J., Moreno-Navarro, J.J.: Constructive intensional negation. In Stuckey, Kameyama, eds.: Functional and Logic Programming Symposium. Number 2998 in Lecture Notes in Computer Science, Nara, Japan, Springer Verlag (April 2004) 39–54

[8] Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: The Ciao System. Reference Manual (v1.10). Technical report, School of Computer Science (UPM) (2004) Available at `http://www.ciaohome.org`.

[9] Source-Navigator: http://sourcenav.sourceforge.net/

[10] Ellis, M.A., Stroustrup, B.: The Annotated C++ Reference Manual. Addison-Wesley (1990)

[11] Taibi, T.: An Integrated Approach to Design Patterns Formalization. In: Design Pattern Formalization Techniques. IGI Publishing (March 2007)

[12] Blewitt, A., Bundy, A., Stark, I.: Automatic verification of design patterns in java. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA, ACM (2005) 224–232

[13] Mens, K., Michiels, I., Wuyts, R.: Supporting software development through declaratively codified programming. In: SEKE. (2001) 236–243

[14] Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Notices **39**(10) (October 2004) 132–136

[15] Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific, static analyses. ACM SIGPLAN Notices **37**(5) (May 2002) 69–82

[16] Volanschi, E.N.: A portable compiler-integrated approach to permanent checking. In: ASE, IEEE Computer Society (2006) 103–112

[17] Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), New York, NY, USA, ACM Press (2005) 363–385