Towards Checking Coding Rule Conformance Using Logic Programming*

Guillem Marpons[1], Manuel Carro[1], Julio Mariño[1], Ángel Herranz[1], Lars-Åke Fredlund[1], and Juan José Moreno-Navarro[1,2]

[1] Universidad Politécnica de Madrid [2] IMDEA Software
{gmarpons,mcarro,jmarino,aherranz,lfredlund,jjmoreno}@fi.upm.es

Coding Rules An approach to increasing program reliability and maintainability involves a disciplined use of programming languages so as to minimise the hazards introduced by error-prone features. This is accomplished by writing code that relies only on a well-defined language subset described by a collection of so-called *coding rules*. There are several corpora thereof, such as MISRA-C ([1], aimed at the C language) or HICPP ([2], for C++). Some of them are intended for domains with special restrictions, like JavaCard (`http://java.sun.com/products/javacard/`). Additionally, organisations can set up their own internal rules. Checking the conformance of source code to a set of rules needs, in general, statically analysing such code.

One feature of coding rules is their diversity: they range from being trivially checkable (MISRA-C 20.4: "*do not use the `malloc()` function*") to expressing non-computable properties (HICPP 3.1.9: "*behavior must be implemented by only one member function*"). Among the rules that can be statically enforced (or, at least, checked in such a way that the user can be directed to probably non-compliant code) we have focused on a particular class that we have termed *structural rules*: those which deal with static entities in the code (classes, member functions, etc.) and their properties and relationships (inheritance, overriding, etc.) There seems to be comparatively less efforts towards analysing these *programming-in-the-large* issues than towards checking other runtime properties (pointer sharing, index ranges, etc.) which are however interesting for other rule types.

This work is part of the Global GCC project (GGCC, `http://www.ggcc.info/`), a consortium of EU industrial corporations and research labs funded under the ITEA Programme, which aims at enriching the capabilities of the GNU Compiler Collection with advanced project-wide compile-time analysis capacities. In this context we are adding a facility to define coding rule sets and providing mechanisms to check code compliance. We plan to take advantage of the static analysers and syntax tools in GGCC.

A Framework for Formalisation and Checking A major drawback of actual coding rules is that they are written in natural language, which bears ambiguity and undermines any effort to apply them automatically. We chose to formalise them using first order logic (similarly to [3]) as it is expressive enough to easily capture the meaning of short sentences in well-defined domains. Relationships between program entities are encoded as facts (thus giving an abstract description of the program) and a formula is generated for every coding rule. When these, together, are inconsistent, the program violates the

* 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).

coding rule. We automate this process by generating a Prolog program containing such set of facts and clauses defining a predicate which describes a rule violation. Successful queries to this predicate pinpoint infringements of the rule and the answer substitutions identify a source of the violation. A closely related approach, applied to a different realm, has been followed by [4,5].

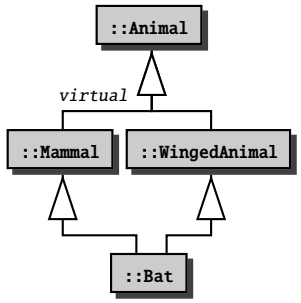
We plan to provide a user-friendly domain-specific language [6, Section 4] which works around Prolog peculiarities. A set of predefined predicates describing (structural) program properties, gathered by a compiler, sits at the core of this DSL.

An Example

Rule 3.3.15 of HICPP reads "*ensure base classes common to more than one derived class are virtual*", where the domain of discourse is the set of the classes in the program. A scenario violating the coding rule is exemplified by the predicate at the right.

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

`class/1`, `direct_base_of/2`, and `virtual_base_of/2` are synthesized from the program to analyse. The diagram below is a concrete example which violates this rule, and the facts at its right are a representation of the relationships in the diagram, needed to apply the code for the rule violation above. A query to `violate_hicpp_3_3_15(A,B,C,D)` would instantiate the variables to the class names in the example.




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

%% base_of/2 is the transitive
%% closure of direct_base_of/2
base_of(A,A).
base_of(A,B) :-
    direct_base_of(A,C), base_of(C,B).
```

References

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. Taibi, T.: An Integrated Approach to Design Patterns Formalization. In: Design Pattern Formalization Techniques. IGI Publishing (March 2007)
4. Fabry, J., Mens, T.: Language-independent detection of object-oriented design patterns. Computer Languages, Systems and Structures **30**(1–2) (April/July 2004) 21–33
5. Blewitt, A., Bundy, A., Stark, I.: Automatic verification of design patterns in java. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: ASE 2005, ACM (2005) 224–232
6. Marpons, G., Mariño, J., Herranz, Ángel., Fredlund, L.Å., Carro, M., Moreno-Navarro, J.J.: Automatic coding rule conformance checking using logic programs `http://www.ggcc.info/?q=codingrules`.

Towards Checking Coding Rule Conformance Using Logic Programming

G. Marpons[1] **M. Carro**[1] **J. Mariño**[1] **A. Herranz**[1] **L. Fredlund**[1] **J. Moreno-Navarro**[1,2]

(1) Universidad Politécnica de Madrid (2) IMDEA Software

Coding Rules

Constrain admissible constructs (e.g. forbidding error-prone features or coding styles) to help producing safer code.

Standard coding rule sets do exist, e.g.:

MISRA-C (C language): automotive industry standard

High-Integrity C++ (HICPP): sponsored by a private company

Javacard: addressing specific restrictions of Java Smart Cards

Enormous diversity in:

- Program features involved
- Analysis techniques required
- Static enforceability

We focus on **structural rules**, which deal with **relationships** between **static entities** in the code (classes, member functions, etc.), e.g.:

Rule HICPP 3.3.15:

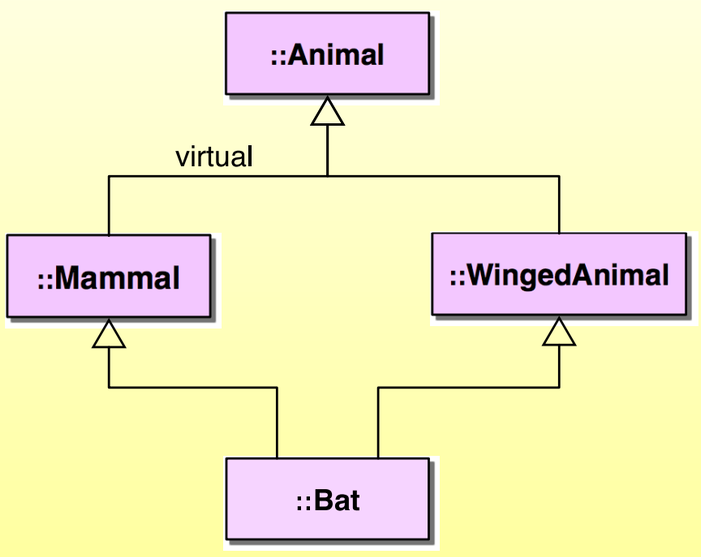
"Ensure base classes common to more than one derived class are virtual."

Natural language is **inherently ambiguous**: Which inheritance links must be tagged as "virtual"?

A framework to formalise coding rules is necessary to statically check that programs conform to a given set. We are developing such a framework in the environment of the GGCC project.

Knowledge Base About a Program

A set of classes violating rule HICPP 3.3.15:

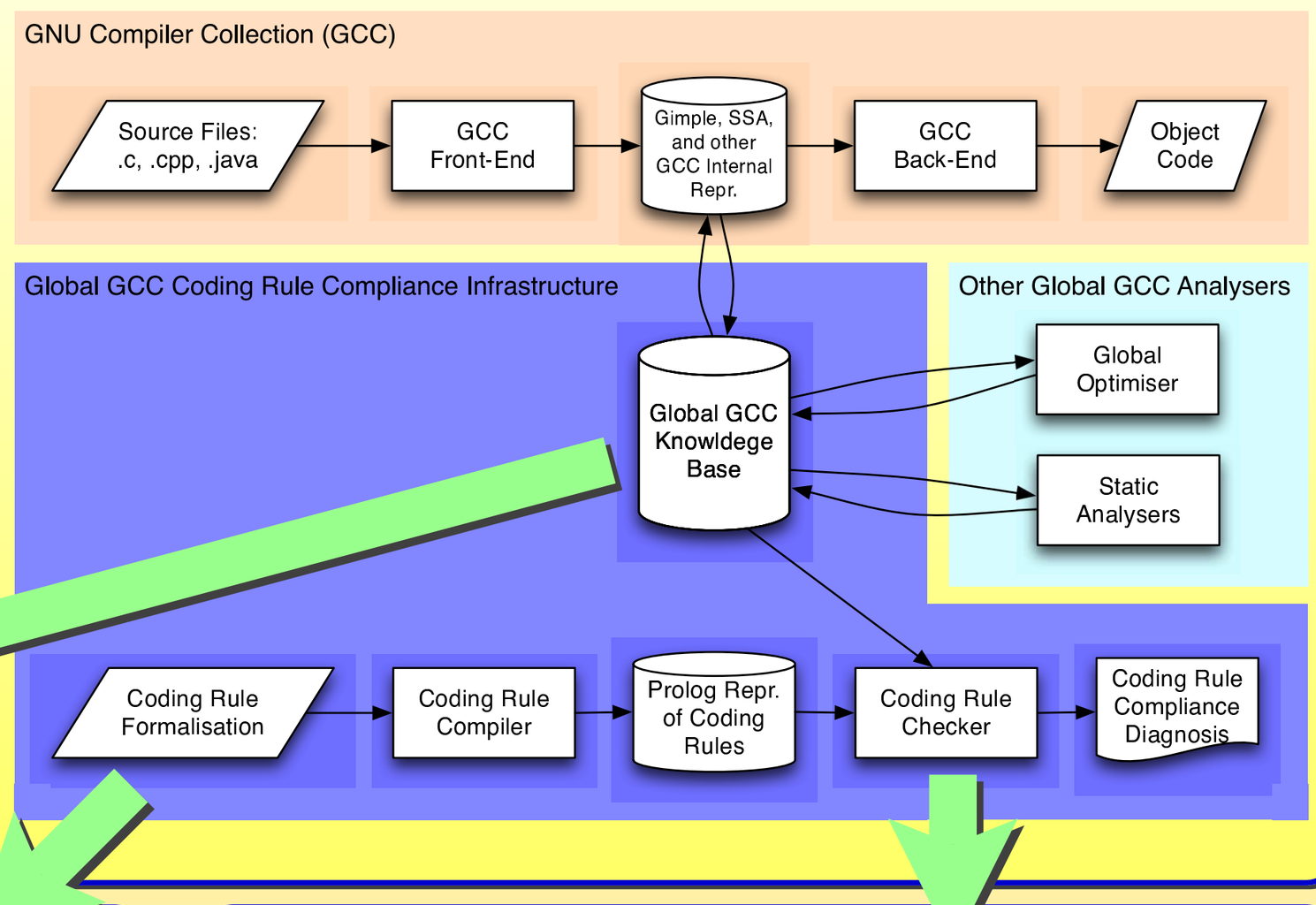


Program properties gathered during compilation for the above example and relevant to rule HICPP 3.3.15:

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

The Global GCC Project

- ITEA funded (2006-08) consortium of industrial / research partners
- Goal: improving static analysis capabilities of the GNU Compiler Collection (GCC)
- **Global GCC knowledge base:** share information among different GGCC analysers



Rule Formalisation

Based on **first-order logic** and written in a domain-specific language which is translated into Prolog and that:

- Formalises standard coding rule sets in a declarative style
- Makes it easier for the final user to define additional coding rules
- Provides a collection of predefined predicates about program facts (such as `class/1`, `base_of/2`, or `in_call_graph_of/2`)
- Quantification over certain domains
- Constructive negation

Rule Checking

Rule HICPP 3.3.15 translated into Prolog

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

We do not code the rule itself, but **its negation**. Any program that satisfies the negated rule thus violates the coding rule.

Predicates coding rule violations are queried against facts describing a program. A successful resolution flags a rule violation, providing a witness.

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).