Reprinted from the

Proceedings of the GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Google*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Adding Coding Rule Checking Capabilities to the GCC Toolchain

Guillem Marpons
Universidad Politécnica de Madrid
gmarpons@fi.upm.es

Álvaro Polo
Telefónica I+D
apv@tid.es

Julio Mariño
Universidad Politécnica de Madrid
jmarino@fi.upm.es

Abstract

Coding rules, which codify software best practices by constraining the set of "admissible" programs, are often used in industry to increase program reliability and maintainability. We present a tool that seamlessly integrates coding rule checking capabilities into the main development workflow. In the proposed framework, the necessary source code features are extracted from the GCC compilation process. In this way, both compiler infrastructure and compilation stages can be reused. The coding rules themselves are defined using a high-level declarative language.

1 Introduction

Although there is a trend towards increased use of high-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 or C++, which are notorious for fostering dubious practices, remain very popular.

However, a good usage of C or C++ involves using the language in a disciplined manner, such that the hazards brought by their 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.

Some standard rule sets do exist, listing good general programming practices for a given language. This is the case of *High-Integrity C++* (HICPP [15]) which provides about a hundred coding rules for C++.

Another leading initiative is MISRA-C [13], 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.

Coding rules can also be used to enforce domain-specific language restrictions. Java Card [18], for example, is a subset of Java for programming Smart Cards, an environment where memory is scarce.

At the other end of the spectrum, each organisation—or even project—can establish its 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.[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.

Note that, although there are similarities, there is a significant difference between a style guide, being mainly devoted to naming conventions, organisation of the code, etc., and coding rules, which typically constrain the kind of programs that can be written.

This work is developed within the scope of the Global GCC project (GGCC, [7]), a consortium of European industrial corporations and research labs funded under the Eureka/ITEA Programme. GGCC aims at extending GCC with project-wide optimisation and compilation capabilities. The project was introduced to the GCC community in a prior GCC Developers' Summit [17].

In the context of GGCC, we seek the inclusion of a facility integrated into the GCC toolchain for defining new sets of coding rules for C++, and providing mechanisms to automatically check (non-trivial) software projects for conformity. We think that adding this feature to the day-to-day tools of the developers will facilitate the adoption of coding rules in many projects. Moreover, we can make heavy use of the syntax and static analysis

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

tools already present in GCC. On the rule-writer side, a logic-based language makes it possible to easily capture the meaning of coding rules. Later on, a Prolog engine is used for checking code compliance [11].

2 Structural Coding Rules

Coding rules greatly differ on the aspects of the target language they rely on, and the verification techniques required to automatically enforce them. A good number of rules in HICPP (and to a lesser extent in MISRA-C) 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 rule is HICPP 3.3.2; it states, "*write a 'virtual' destructor for base classes*." The rationale behind this requirement is that if ever an object is destroyed through a pointer to its base class, the correct version of the destructor code will be dynamically dispatched.

Another example is Rule HICPP 3.3.11 that says "*do not overload or hide inherited non-virtual functions*." This aims at avoiding unexpected behaviour, as non-virtual functions are statically bound.

We have termed these rules *structural*, and our framework for automatic coding rule checking is designed for coping with this kind of rules in the first place, even if other rules could be accommodated within it in the future (see Sect. 5.1).

Some rules combine these structural attributes with purely syntactic information. For instance, Rule HICPP 3.3.16 states to "*explicitly declare polymorphic member functions 'virtual' in a derived class*." A member function is semantically *virtual* if it is declared as `virtual` in the base class where it is declared for the first time. In order to conform to this rule we must put the word `virtual` in every re-declaration of the member function.

Other rules involve, besides optional structural information, dynamic or run-time properties that are undecidable in most cases. Furthermore, rule standard sets use sometimes very vague properties that are difficult or impossible to formalise. Nonetheless, approximate information given by a tool like the one described in this paper can still be very useful and reduce manual inspection efforts. For example, Rule HICPP 3.1.3 reads "*declare or define a copy constructor, a copy assignment operator, and a destructor for classes which manage resources.*" There is no formal definition of what a "resource" is, but we can consider some particular cases of (signs of) resources, as classes which contain pointer data members (suggesting that some dynamic memory should possibly be de-allocated on destruction). We can also take a conservative strategy and warn on every class that does not contain the full set of members requested by the rule above, for manual searching of "resources."

One important feature of structural rules is that they rely on global, project-wide properties. Information coming from different compilation units may be involved in its checking. For instance, in Rule HICPP 3.3.2 above, all the code in a project must be searched for subclasses of a given base class. This also gives an idea of the difficulty of manual inspection.

2.1 Tools for (Structural) Rule Validation

There exist a number of proprietary tools that claim to be able to check code for compliance with a subset of HICPP, MISRA-C, or other standards. They fall into two categories: compilers (such as those by IAR Systems [9]), or quality assurance tools (Parasoft is a good example [14]). Other quality assurance and static analysis tools, e.g. Klocwork [10] or Coverity [5], define their own list of informally described checks aimed at avoiding hazards.

Compilers provide a closed set of checkings, whereas quality assurance tools usually have some kind of extensibility mechanism. The main drawback in this second case is that in absence of a formal definition of rules, it is difficult to be certain about what they are actually checking. Two different tools could very well disagree about the validity of some particular piece of code with respect to, e.g., the same HICPP rule.

Even if the source code of the rules were available, they are written in C or C++, on top of an API for traversing the Abstract Syntax Tree (AST). This low-level representation is the same that must be used for writing new checks.

Checkstyle is a free software tool originally aimed at guaranteeing adherence to a coding style guide [2]. It has evolved to allow some structural checks providing a similar API for Java code.

A more interesting approach is that of the proprietary SemmleCode [6], also targeting Java, that provides

a declarative language for querying a database with knowledge about some program source code. These queries can implement many of the coding rules we are interested in. Semmle code is integrated into Eclipse.

Declarative technology brings the possibility of easily—if a suitable formalism is found—translating rules from their informal description in standard sets into an executable specification. While Semmle code uses an SQL-like language for specifying queries, we propose a logic programming-based formalism.

3 A Framework for Structural Coding Rule Checking Based on Logic Programming

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

1. Express the coding rule in a suitable formalism, assuming an appropriate representation of the entities the rule needs to know about. This is a one-time step, independent from the particular software project to be checked.
2. Transcribe the necessary program information into the aforementioned representation. This is necessary for every project instance.
3. Prove (automatically) whether there is a counterexample for the rule. In that case the rule is not met; otherwise, the code conforms to the rule.

For the time being we transcribe the *violations* of coding rules as Prolog predicates,[2] their arguments being the entities of interest to the programmer. In this way the verification method can return *references* to the elements in the source code which have to be corrected. The use of logic programming for manipulating architectural information about software has been already investigated in the field of formalisation and automatic verification of design patterns [19, 1, 12]. We plan to improve usability of our tool with a Domain-Specific Language (DSL) for rule definition (see Section 5.2).

Gathering structural information from a particular C++ software project takes place during its compilation with `g++`, in the way detailed in Section 4. The full software project must be re-compiled in order to obtain all the necessary information (see details in Section 4.3).

Validation of the C++ project against the coding rules is carried out by executing, in the Ciao Prolog System (GPL'd, [8]), each of the Prolog predicates representing a coding rule violation together with the Prolog facts representing the project structural information. A positive answer to a query 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.

3.1 Formalisation of Coding Rules as Logic Programs

Coding 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 in the following examples. This predicates constitute the programming interface for defining rules and are defined on top of the information generated by the compiler, as explained in Section 4.

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 (i.e., 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.

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:

[2] For a quick introduction to Prolog and pointers for further reading, the reader can go to `http://en.wikipedia.org/wiki/Prolog`.

PREDICATE	MEANING
immediate_base_of(a,b)	Class *a* appears in the list of explicit base classes of class *b*.
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_function(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 of *c*.
constructor(c)	Member *c* is a constructor.
destructor(d)	Member *d* is a 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*.
overrides(a,b)	Member function *a* is defined in a derived class as a re-declaration of member function *b*.
overloads_member(a,b)	Some class contains methods *a* and *b* (maybe declared in different classes in the same inheritance hierarchy). They have the same name but different argument types.
sig(f,i,a,r)	Function *f* has name *i*, argument type *a* (a list), and result type *r*.

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

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 declared as a virtual base of *C*.

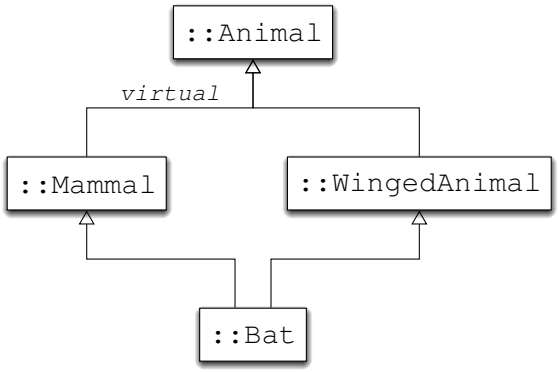

Figure 2: An example of violation of Rule HICPP 3.3.15.

Figure 1 shows, among others, the Prolog formalisation of a violation of Rule HICPP 3.3.15. The success of a query to `violate_hicpp_3_3_15/4` would exemplify a violation of Rule HICPP 3.3.15.[3]

The four involved 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 (maybe zero) of subclassing steps. Finally, for the rule to be violated we require that class `A` is not virtually inherited by class `C`.

Note the use of dis-equality and negation by failure (operator `\+`) in the definition of the rule.

Figure 2 depicts a set of classes and their inheritance relations, which make the literal

```
violate_hicpp_3_3_15('::Animal',
'::Mammal', '::WingedAnimal', '::Bat')
```

deducible, thus constituting a counterexample. If the inheritance from `'::Animal'` to `'::WingedAnimal'` was also *virtual*, the goal would fail (no counterexample could have been found).

[3]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) :-
    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).


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).
```

```
violates_hicpp_3_3_2(BaseClass) :-
    has_derived_classes(BaseClass),
    has_no_virtual_destr(BaseClass).

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

has_no_virtual_destr(Class) :-
    \+ (
          has_member(Class,Destr),
          destructor(Destr),
          virtual_member(Destr)
       ).
```

Figure 1: Prolog formalisation of some HICPP rules.

Another rule that can be easily implemented in this framework but requires disjunction (operator `;`) 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.

Figure 1 shows also Rule HICPP 3.3.2, from Section 2, translated into Prolog. Note that auxiliary predicates are used in the definition of this rule.

Having the rules defined in this or a similar high-level formalism allows for clearly understanding what they check, beyond the possible ambiguities that the informal description contains. This is a clear advantage over most of the commercial compilers or tools for quality assurance that incorporate rule checking facilities.

4 Using GCC for Gathering Program Information

As a first test of our rule checking procedure we developed a prototype [11] that works with program information provided by Source-Navigator [16]. This information is both incomplete (many relevant features of C++ source code are not present in the Source-Navigator knowledge base) and imprecise (e.g., as namespaces and templates are not well supported, sometimes is not possible to distinguish among different entities with the same identifier). In fact, Source-Navigator carries out an incomplete parsing of the source code.

Among the advantages of using a compiler as GCC as the main source of program analysis information that we expect to exploit, we find:

- Non duplication of compiler capabilities.
- A complete parsing gives us potential access to any feature of the code.
- We can easily guarantee that only correct programs are analysed.
- Object code generation and software analysis tools support exactly the same input (e.g. the same language version), rely on the same data and interpret the code in the very same way.
- As compilation stages are reused, a better integration into the main development workflow is possible (see Section 4.3).

All these characteristics distinguish our approach from existing quality assurance tools, including Semmle-Code [6] (not from compilers with coding rule checking capabilities as [9]).

4.1 A Middle-End Pass for Program Feature Extraction

Some of the program features needed for writing rules are common to many languages and have a representation at GIMPLE level—even if the semantics are not exactly the same in different languages. Other properties or constructs are language-specific. Since adding a

middle-end (ME) pass is quite simple and clean with the current infrastructure for optimisation, this has been our first step towards a facility for program feature extraction.

Our pass writes to a file Prolog facts describing structural properties of the analysed software, which can be either a program or a library. It has been designed to provide useful information about C++ code. But, as no language-hooks are used, it should be largely language-independent and easy to adapt to other languages. This is a natural extension, provided that GCC is a multi-language compilation framework.

Another advantage of implementing the functionality in an ME pass is that no overhead exists if the new functionality is not activated with the corresponding flag.

On the other hand, C++-specific features such as templates or friends can not be used for defining rules, at the time this is written.

Another possible drawback of our system (but not completely avoidable in other approaches) is that we analyse only one of many potential instantiable programs generated from a given source code: the one determined by the pre-processing stages in a given environment.

Our development is available at the web site of the project [7], and it is in sync with the GCC trunk.

4.2 Extracting Program Information as Prolog Facts

The *output* of the information-extraction facility is a list of Prolog facts, one per line. Each Prolog fact represents a feature of the source code that needs to be stored/streamed for later use by the coding rule checker.

We distinguish between global and local entities because they use a different identification scheme.

Global entities are those that can appear outside a function or method definition (e.g. namespaces, classes, etc.) For each global entity that is either declared or defined into the compilation unit under consideration, a fact is generated stating that this entity exists. The following grammar rules define the syntax of the unary Prolog predicates for these facts:

global_entity_term ::= *global_entity* (*GLOBAL_KEY*) .
global_entity ::= `namespace`
| `enum`
| `enum_value`
| `union`
| `record`
| `function`
| `global_var`
| `method`
| `field`
| `bit_field`

A *record* is either a `class` or a `struct`. A *method* (a *field*) is a function (a data variable) declared inside a *record* or *union*. *GLOBAL_KEY* is a key identifying the entity. We have tentatively used a character string, and more specifically the *mangled* name of the object.

Mangled names [20] 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. This is the case for structural coding rule checking. Using C++ mangled names has many advantages for our specific task:

- They identify global entities appearing into the source code giving us a name that is (excepting some corner cases that have to be specifically handled) unique across all the source code files to be linked together (modulo they can actually be linked with no errors).
- They resolve overloaded function names, including overloading originated by templates.
- They integrate C++ operators into the global naming scheme.

More specifically, we use the same coding scheme used for mangling by GCC: the mangling standard for the Itanium C++ ABI [4]. The mangling capabilities of GCC have been extended to get names for entities that are not named in object code (e.g. classes). Since mangling has a compositional structure (the mangled name of a method contains the mangled name of the class that contains it, etc.), this can be straightforwardly done in many cases. Still, some difficulties remain for anonymous *record*s and *unit*s, for which some kind of project-wide naming scheme must be devised, presumably including the file name.

PROPERTIES OF GLOBAL ENTITIES
`virtual(method(GLOBAL_KEY))` Stated for each method identified by `GLOBAL_KEY` *iff* the method is virtual.
`accessibility(method(GLOBAL_KEY),ACCESS_SPECIFIER)` A fact of this form exists for each method reported, indicating an access specifier for it. `ACCESS_SPECIFIER` is either `public`, `protected`, or `private`.
RELATIONS AMONG GLOBAL ENTITIES
`contains(namespace(GLOBAL_KEY),GLOBAL_ENTITY)` `GLOBAL_ENTITY` is a *global_entity* defined in the namespace identified by `GLOBAL_KEY`. `GLOBAL_ENTITY` can represent a *namespace*, an *enum*, an *enum_value*, a *union*, a *record*, a *function*, or a *global_var*.
`contains(record(GLOBAL_KEY),GLOBAL_ENTITY)` `GLOBAL_ENTITY` is a *global_entity* defined in the record identified by `GLOBAL_KEY`. `GLOBAL_ENTITY` can represent an *enum*, an *enum_value*, a *union*, a *record*, a *method*, a *field*, or a *bit_field*.
`enumerates(enum(GLOBAL_KEY_1),ENUM_VALUE(GLOBAL_KEY_2))` `GLOBAL_KEY_1` identifies an enumeration that contains a label identified by `GLOBAL_KEY_2`.
`extends(record(GLOBAL_KEY_1),record(GLOBAL_KEY_2))` `GLOBAL_KEY_1` immediately inherits from `GLOBAL_KEY_2`.
PROPERTIES OF RELATIONS
`virtual(extends(record(GLOBAL_KEY_1),record(GLOBAL_KEY_2)))` This is stated for each inheritance link that is virtual.
`accessibility(extends(record(GLOBAL_KEY_1),record(GLOBAL_KEY_2)),ACCESS_SPECIFIER)` A fact of this form does exist for each inheritance link, indicating an access specifier for it.

Table 2: Structure of some Prolog terms representing properties and relations among C++ global entities.

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

Besides individual properties, relations among global entities exist. These relations usually involve two global entities—a binary predicate being used—but relations of greater arity could arise. Some examples of binary relations among global entities can also be found in Table 2.

Relations such as *extends* can also have properties attached. Two examples are given in the same Table 2.

Local entities are those that can appear inside a function or method, and are not a global entity. Their keys are composed because they have to reference the function in which the entity is defined. A Prolog term is generated in the output for local entities such as local variables and function arguments. Some relations among local entities, or among local entities and global entities can be defined. In the future, even statements, expressions, and declarations could be defined in this flattened representation of the AST.

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. This two-layered architecture is also intended to better support extending this rule-checking facility to other target languages.

4.3 User Interface of the New Facility

As it has been stated in Section 2, we need to process the full C++ source code of a project (application or library). We also pursue to smoothly integrate the rule-checking procedure into the development workflow and the building process of new and existing projects.

What we do is to process each compilation unit separately, in the usual way, dumping the Prolog facts to a shared file where all data about the project are accumulated. Two new flags `-fipa-codingrules` and `-fipa-codingrules-file=FILENAME` are provided in `g++` and `cc1plus` commands. The first one activates the feature extraction facility and the second one specifies the file where Prolog terms are stored.

After a full cleaning of the building tree, the project is built by means of `make`, `scons`, `ant` or whatever tool is used to that end. These tools usually allow changing compilation options with some environment variables or by editing some configuration file. For example, if `automake`/`autoconf` are used, a `$CXXFLAGS` environment variable is usually available for setting `g++` compilation options. Note that an absolute pathname must be passed to `-fipa-codingrules-file` to store all the needed Prolog facts in a single file. This is step (2) in Section 3.

For performing step (3) we use a `checkrules` command that is basically a Prolog interpreter generated with Ciao tools [8]. It takes two inputs: the file with facts about the analysed software, and a file with definitions of coding rules (previously compiled into bytecode, i.e. step (1) in Section 3). This command has to take into account that some facts could be repeated in the input, as some files can be included in many compilation units. `checkrules` generates a complete report with all rule violations found.

Note that not relying on the building process of existing projects would be quite challenging, as the appropriate compiling options should be found for every compilation unit (as we need a compilation with no errors for our ME machinery being executed).

Even if the architecture and operation mode of our tool is still rudimentary, it brings the opportunity of integrating coding rule checking into the main toolchain of many developers. The `checkrules` command is not necessarily called in every compilation. But for many developers it can be more comfortably used than more sophisticated quality assurance tools. The consequence is that software weaknesses are caught earlier. Moreover, it facilitates the adoption of this technology thanks to the huge user base of GCC.

4.4 Implementation of the Middle-End Pass

The algorithm for extracting the needed information traverses the call graph, i.e. a graph data structure provided by ME which contains information about all function calls in a given compilation unit. Using this call graph, the GIMPLE tree node for callee and caller function declarations are reachable. For each of them, a recursive algorithm is started from a function `process_tree(tree t)`, that receives a ME tree node of arbitrary type (not only functions). This function allocates a new `struct artifact` object, stores it in a hash table, and returns it. The record contains the following data:

- The ME tree node for the declaration received as argument.
- A unique artifact name (see Section 4.2).
- The parent artifact, i.e. the artifact which defines a context for the current artifact (e.g. classes for function members, records for fields, etc.)
- The children artifacts, i.e. the entities for which this artifact is a parent.
- An access specifier (public, private, protected) for this artifact in its context.
- Specific information for each kind of artifact (namespaces, records, functions, etc.)

This artifact object is all the pass needs to produce the Prolog facts that describe this entity and its relations with other artifacts. The algorithm explores not just the initial tree node passed to `process_tree()`, but any entity related with it. This goal is achieved by recursively processing these related tree nodes as follows:

1. The context declaration of a given tree node is obtained and processed by a recursive call to `process_tree()`. The resulting artifact acts as the parent of the current entity.
2. Depending on the kind of tree node a type-specific processing function is invoked. These functions, which follow a `process_*_tree()` nomenclature (where $\star$ is the specific artifact type), receive the tree node and parent artifact as arguments.

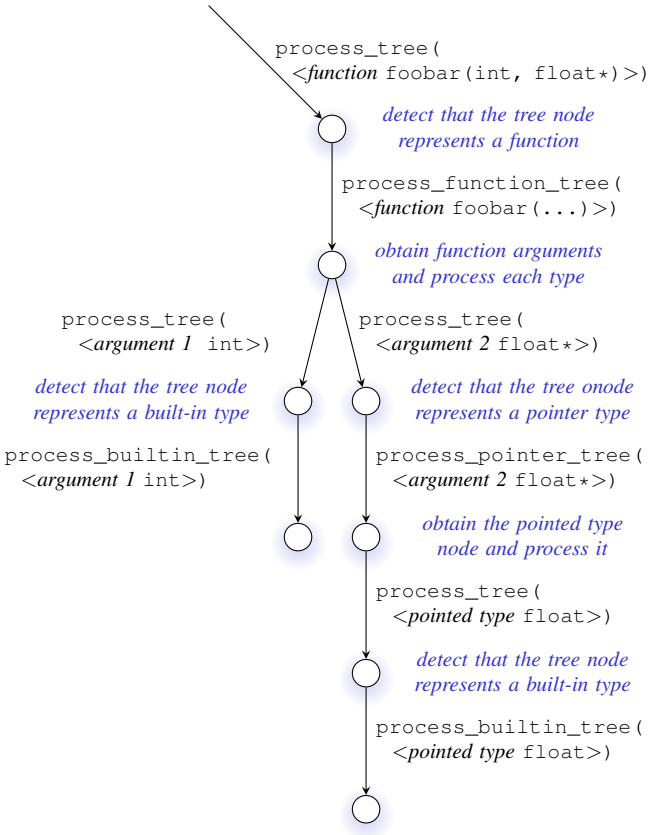

Figure 3: Recursive call graph for `foobar(int, float*)` function processing.

3. The `process_*_tree()` function checks if this artifact was previously processed by searching in the artifacts hash table. If not found, the new artifact is created and its common and specific information is filled up by browsing the ME tree. Also, all related tree nodes are used as argument to invoke `process_tree()`, which returns the artifact object for this declaration and creates it if it does not exist.

4.4.1 The Extraction Algorithm at Work

To illustrate the extraction algorithm, think of a function `foobar(int, float*)` in C. This compilation unit is processed recursively as shown in Figure 3 as follows:

1. `process_tree()` is called with the tree node for `foobar()` function as argument. This function obtains the declaration context for `foobar()` and processes it. As no declaration context is found, no job is done and parent is NULL. (It would be the same for a C++ function declared in the outermost namespace).

2. Since a function-declaration tree node has been found, `process_function_tree()` is called, which extracts the function arguments from the node. For each of `int` and `float*`, the type declaration is obtained and used as the argument for `process_tree()`. When processed, these two artifacts are attached as argument types for `foobar()` function in its artifact object.

3. As a built-in-type declaration tree node is found (for the first argument type of `foobar()`) by `process_tree()`, function `process_builtin_tree()` is invoked. This call creates a new artifact object for the built-in type `int`.

4. As a pointer-type declaration tree node is found (for the second argument type of `foobar()`) by `process_tree()`, function `process_pointer_tree()` is invoked. This call creates a new artifact object for the pointer type `float *`, and processes the pointed type declaration, obtaining the tree node and using it as the argument to `process_tree()`.

5. Finally, a built-in-type declaration tree node is found (for the pointed type of the second argument type of `foobar()`) by `process_tree()`, and a new call to `process_builtin_tree()` is realised. This call creates a new artifact object for the built-in type `float`.

4.4.2 Output Formatting

Once all function calls and their related entities have been processed and stored in the artifacts hash table, this table is traversed to produce the output. This task is performed by using an output formatter function, set by default to a Prolog fact formatter. Some other formatters could be written to provide an alternative output, like CLIPS facts [3], SQL sentences, LaTeX, XML, etc., extending the usefulness of this ME pass. In the future, this output formatter could be set by a `cc1plus` argument like `-fipa-codingrules-format=FORMAT`.

5 Future Extensions

Besides adapting the tool to other languages supported by GCC, there are two main improvements we plan to introduce. They are discussed in more detail in the following subsections.

5.1 More Syntactic and Semantic Properties

Syntactic information and other language-independent properties that are lost at the GIMPLE level are needed for many rules. One example is Rule HICPP 3.3.16 of Section 2. We need to integrate to our framework information coming from the GCC front-ends. The challenge is to achieve this in a non-intrusive and maintainable way, and with no overhead for users that do not need this functionality.

Flattening the full AST as Prolog facts is going to generate an enormous amount of facts. It would be interesting to devise some procedure to selectively enable data generation for those entities actually interacting in some rule.

Many other rules refer to sequences of events occurring at run-time, for which data-flow analysis information computed by some optimisation passes could be useful. Even more powerful static analysis is being incorporated into GCC through the GGCC project [17]. Is is our goal to interface with these passes and static analysers to obtain properties and use them in rule definition.

5.2 A Domain-Specific Language for Coding Rule Definition

While, as we have seen, Prolog is able to express structural rules, making it the native language for specifying coding rules has several drawbacks: people writing or 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 and which needs care regarding, e.g., evaluation order and instantiation state of variables, etc.

Moreover, a proper compilation of rules into Prolog demands a *careful* use of several extensions to the language.

We plan to provide rule writers with a DSL for defining new rules, which we have codenamed CRISP (Coding Rules in Sugared Prolog). In fact, CRISP is a family of languages with a common core and some constructs specific to the source language being analyzed.

CRISP is still in an early stage of development, but there are already several key features taking shape. For instance, CRISP is intended to be fully declarative, i.e. it will not contain impure features like those in Prolog, and will abstract away many execution details—e.g. rule designers should not need to worry about order of execution of goals inside a clause, as this can be optimized by the CRISP compiler.

Negation, transitive closure, and tabulation of frequently used results will also be built into the system along with a model for declaring errors associated with rules that can serve as an interface to programming environments.

CRISP will be strongly typed, thus requiring less input from the rule writer, as many typing constraints can be inferred at compile time.

6 Conclusions

The distinctive features of our approach to structural coding rule conformance checking can be summarised as follows:

1. Rules are formally defined by means of a declarative rule definition language.
2. Users can define their own rules, or adapt existing ones.
3. Seamless integration into the workflow of the developers.
4. Basic information about programs is taken from the very same compiler used to generate object code.
5. Potentially multi-language.

With our first prototype based on Source-Navigator, we ran some experiments and detected a number of rule violations on some well-known open-source C++ projects [11]. We are currently reproducing these results with the new GCC-based tool, which will allow checking a larger number of rules. The new results will be published on the GGCC website ([7]) when available.

Acknowledgments

We would like to thank Albert Cohen and Cupertino Miranda from INRIA-Futurs, and Basile Starynkevitch from CEA-LIST, for introducing us to this wild world of GCC hacking. Also Arnaud Laprevote from Mandriva and Víctor Pablos from UPM have been very helpful.

This work is 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 the Madrid Regional Government.

References

[1] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in java. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 224–232. ACM, 2005.

[2] Checkstyle. `http://checkstyle.sourceforge.net/`.

[3] CLIPS Rules. `http://clipsrules.sourceforge.net/`.

[4] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. `http://www.codesourcery.com/cxx-abi/abi.html#mangling`.

[5] Coverity. `http://www.coverity.com/`.

[6] Oege de Moor. .QL: Object oriented queries made easy. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. Tutorial for International Summer School GTTSE 2007; to appear.

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

[8] M. Hermenegildo, 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*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[9] IAR Systems. `http://www.iar.com/`.

[10] Klocwork. `http://www.klocwork.com/`.

[11] Guillem Marpons-Ucero, Julio Mariño-Carballo, Manuel Carro, Ángel Herranz-Nieva, Juan José Moreno-Navarro, and Lars-Åke Fredlund. Automatic coding rule conformance checking using logic programming. In Paul Hudak and David Scott Warren, editors, *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2008.

[12] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming. In *SEKE*, pages 236–243, 2001.

[13] MIRA Ltd. *MISRA-C:2004. Guidelines for the Use of the C Language in Critical Systems*, October 2004.

[14] Parasoft. `http://www.parasoft.com/`.

[15] The Programming Research Group. *High-Integrity C++ Coding Standard Manual*, May 2004.

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

[17] Basile Starynkevitch. Multi-stage construction of a global static analyzer. In *Proceedings of the GCC Developers' Summit*, pages 143–151, July 2007.

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

[19] Toufik Taibi. *Design Pattern Formalization Techniques*, chapter An Integrated Approach to Design Patterns Formalization. IGI Publishing, March 2007.

[20] Wikipedia. `http://en.wikipedia.org/wiki/Name_mangling`.