C++ Code Analysis: an Open Architecture for the Verification of Coding Rules

Alessandra Potrich and Paolo Tonella

ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, Povo (Trento), Italy

Abstract

The analysis of C++ code is the basic building block of the collaboration between ITC-irst and CERN, aimed at improving the quality of the software by exploiting the information that can be automatically gathered from the code. The first objective of the collaboration is the development of a coding rule check tool. Successive steps will include a reverse engineering module and an intelligent refactoring tool. Since all planned applications, and possibly also those not yet considered, share a common analysis bulk, particular attention was devoted to the development of an open architecture for the analysis of C++ code.

In this paper the adopted architectural solutions are presented and discussed, highlighting their generality, the possibilities of extension that they offer, and the way details could be encapsulated within packages, so that a clear and sharp interface between the subsystems is defined. The peculiarities of the C++ language are also described, together with the way they were approached and the state of the current implementation.

Keywords: Code analysis, coding rules, reverse engineering, software architecture.

1 C++ analysis model

The model of the C++ language that was adopted for the development of the coding rule check tool is very general and highly independent from this particular application. It is also possible to adapt it for a different object oriented programming language. The choice of a general model is the basis for the development of an architecture open to a variety of future applications.

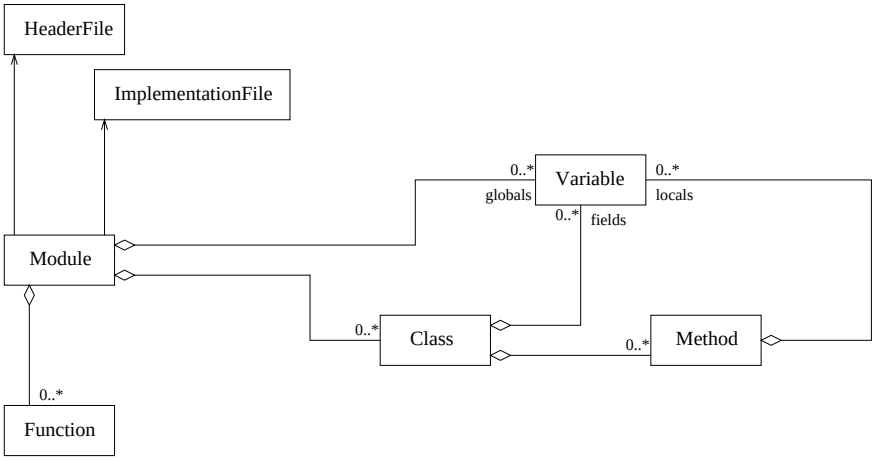

Figure 1: Simplified model of the C++ language.

Figure 1 shows a general view of the C++ analysis model, given in the Unified Modeling Language (UML) notation [2]. A C++ program is made of several composing modules, each of which is analyzed separately. A C++ **Module** may contain a set of global variables, a set of classes and a set of functions. Such constituents are represented by the three aggregations respectively with classes **Variable**, **Class** and **Function**. Since class **Variable** plays different roles in different relations, its specific role (**globals**) with class **Module** is explicitly indicated. Each **Module** entity can also be associated to the header file and implementation file actually containing the code (classes **HeaderFile** and **ImplementationFile**).

A class contains fields and methods, respectively represented as aggregations with classes **Variable**, playing the role **fields**, and **Method**. Each method has a set of local variables, instantiated by objects of type **Variable**, with the role of **locals**.

2 Architecture

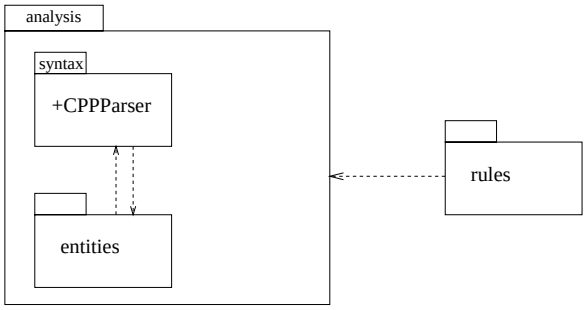

Figure 2: Architecture of the coding rule checker.

Figure 2 depicts the architecture of the coding rule check tool in the formalism of UML. It includes two main packages, **analysis** and **rules**. The first package in turn contains two nested packages, **syntax** and **entities**[1].

Package **analysis** is responsible for the analysis of an input C++ program, independently of the application for which such analysis is performed. External packages relying on package **analysis** can be developed for different applications. In this paper the coding rule check application is discussed in detail, but future work will be devoted to additional applications like, e.g., a reverse engineering engine and a refactoring system. Package **rules** implements the first application, aimed at checking if the source program was written according to a given set of coding rules.

The package **analysis** is organized into two subpackages, **syntax** and **entities**. The most important class exported by package **syntax** is the class **CPPParser**. Such class is automatically generated by the public domain tool `javacc`, which converts an input grammar into a top down parser, written in Java. The C++ grammar that was used as a starting point for this work is freely distributed with `javacc`. It is based on the C++ grammar that can be found in [3].

The package **entities** contains the C++ analysis model described in the previous Section. Method **parse** from class **CPPParser** is responsible for generating the objects that populate the package **entities**, instantiating the classes of the language model. For example, each time a class is encountered in the input code, a **Class** object is created and appended to the list of classes contained in the **Module** under analysis. The two packages **syntax** and **entities** collaborate so as to generate a network of objects that represents the input code in a form structured according to

[1] In the current implementation such grouping is only conceptual.

the chosen C++ model. The **analysis** package as a whole can only be queried on the entities that were generated for a given input program. Each user of this package is only allowed to ask for entities and for their properties, while their computation is performed by the **parse** method within the **syntax** package.

The package **rules** contains a hierarchy of coding rules to be checked against an input program. A **Module** object is obtained within the **main** method of this package by asking the package **syntax** to parse the source code and populate the **entities** package. Then the **check** method is activated for each coding rule, which has access to every entity and entity property obtainable via a query to the **entities** package. The values of the retrieved properties are checked for compliance with the coding rules. For example, a hypothetical coding rule may require that the class **name** starts with a capital letter. The associated **check** method could iterate on the list of classes contained in the **Module** entity, and query for the **name** attribute on each instance. Coding rule violations are signalled on the standard output. The document with the coding conventions of the Alice experiment at CERN, most of which are currently implemented in the tool under development, is available at the site [1].

The adoption of this architecture for coding rule check provides a remarkable flexibility. All rules relying on the properties of the entities in the C++ model can be encoded in the tool. In turn the C++ model can be extended if additional properties need to be collected for the check of a given coding rule.

Adding a new application package is extremely simple within the architecture described above. If the new application requires an extension of the C++ analysis model, the related properties and classes have to be added, and the **CPPParser** needs to be adapted so as to generate the related entities. These modifications should not interfere with the existing structure of the analysis model; otherwise all user packages must be updated accordingly. The generality of the reference C++ model adopted in this work should result in a pretty stable core structure. Then the new user package accesses the entities generated by the parser with the simple query protocol that is exported by the **entities** package.

3 Preprocessing

Each module under analysis has to be preprocessed before the parser can accept it. In fact, the C++ language provides the possibility to define macro's that are expanded in the code by the preprocessor. Macro's do not necessarily comply to the C++ syntax. They can contain any text sequence, possibly parameterized, that is literally substituted with the associated macro definition.

The invocation of the C++ preprocessor requires that all included files (header files) be provided, since they contain the macro definitions necessary to transform the source module into the preprocessed one. Under most UNIX systems, the command to preprocess a source C++ file is `g++ -E`. The preprocessed files contain information that has to be discarded before moving to the parsing phase. This is the responsibility of the `strip` filter application.

The C++ preprocessor prepends all directly and indirectly included files to the source code of the module under analysis. Therefore the preprocessed file contains several class definitions and functions that are not part of the current module, being simply included by some header file. The `strip` filter removes them from the preprocessed file. Moreover, the C++ preprocessor inserts some flags in the code that are useful for the successive compilation step. If, for example, preprocessor `g++ -E` is used, its output is expected to be successively parsed using the `g++` compiler, to generate machine code. Therefore the preprocessor inserts compilation directives specific for the `g++ compiler`. Examples include flags `__extension__`, `__const`, `__attribute__`. They are all removed by the `strip` filter.

4 Language issues

Building a C++ analysis tool is a challenging task, because the C++ language has several peculiarities that make it quite complex. In part this is due to its historical origin. It was conceived as an evolution of the C language, able to incorporate the features of object oriented programming. A strong requirement in its development was a total backward compatibility with C, so that a C++ compiler could handle C code as well. Consequently, C++ has all the characteristics of the C language, plus several additional features, and the two can be intermixed in a program. In addition to being not developed from scratch, C++ had also a controversial evolution in its more advanced funtionalities, like exception handling and generic classes.

In order to deal with the complexity of the C++ language, it is important to distinguish between the *compilation* perspective and the *analysis* perspective. The compiler checks the compliance of the source code with the language, and then generates machine code accordingly. Its input may either be an incorrect or a correct program, so that the associated grammar has to be extremely restrictive and recognize only valid strings of the language. On the contrary, an analyzer may assume that the input program was successfully compiled without errors (otherwise it is considered non analyzable). Its grammar can be consequently simplified. The kind of information to be extracted is also different. The compiler needs to capture the statement level semantics, to translate it into machine code, while an analyzer may be interested only in a higher level view. Moreover the performances expected from a compiler are substantially superior to those expected from an analyzer, which is executed less frequently and typically just once per session (while compilers are re-executed after code modifications). All these considerations led to the choice of the `javacc` based C++ grammar. It is not as restrictive as that of a compiler, but it is consequently much more readable and understandable. It is also simpler to extract the needed information from its productions. A prerequisite for its usage is that the input program compiles with no errors. The parser generated by `javacc` is not particularly efficient, but the missing optimizations correspond to a much clearer organization of its internal structure, so that it is simpler to modify its semantic actions to compute the information of interest.

5 Conclusion

An open architecture for the analysis of C++ code was implemented in a code analysis package, developed within the ITC-irst collaboration with CERN. It is exploited by the coding rule check tool. To make analysis independent of the applications using its outcomes, a C++ language model was developed, reflected in the organization of the analysis package.

The current version of the analysis package was checked on the code currently available from the ALICE experiment. It allowed parsing and gathering information about all the software (more than 100.000 lines of code). No parse error was reported by the analysis package, that could extract the information in the C++ language model for all ALICE modules. The coding rule check tool was also run on the code entities retrieved during analysis, resulting in a violation report associated to each source module under analysis.

References

1 Alice Experiment: Coding Conventions, `http://AliSoft.cern.ch/offline/codingconv.html`

2 J. Rumbaugh, I. Jacobson and G. Booch, “The Unified Modeling Language – Reference Guide”, Addison-Wesley, 1998.

3 B. Stroustrup, “The C++ Programming Language (2nd edition)”, Addison-Wesley, 1992.