Generic Metric Extraction Framework

El Hachemi Alikacem[1], *Houari A. Sahraoui*[2]

[1] Centre de Recherche Informatique de Montréal, Québec, Canada

alikacem.el-hachemi@crim.ca

[2] Département d’informatique et de recherche opérationnelle

Université de Montréal, Québec, Canada

sahraouh@iro.umontreal.ca

Abstract:

Nowadays, a large number of extraction tools are available. However, using them, it is often difficult to gather and incorporate new metrics. On the other hand, the metric specifications often lack precision and therefore lead to multiple implementation patterns. In this paper, we propose a new approach of metric gathering. This approach, which is at the same time generic and practical, is based on a metric description mechanism. It uses a language that makes it possible to manipulate data from the source code representation model. In a first phase, we have defined a generic model for object oriented code representation. A second phase defines a description language that offers the syntactic constructions necessary for data manipulation of the generic mode.

Keywords

Metric extraction, object-oriented metrics, code source representation, static analysis.

1 Introduction

Metrics are powerful support tools in software development and maintenance. They are used to assess software quality [5], to estimate complexity [10], cost and effort [11, 2], to control and improve processes. A huge number of metrics have been proposed (see for example [1, 2, 7, 8, 11]), and new metrics are invented continuously. However, most of them are defined informally, using natural language. Due to this lack of formalization, the tools often fail to implement the metrics according to the end user expectations. Moreover, the results produced by a given tool may vary depending on the language that is used. This makes it almost impossible to compare studies and analysis that use the same metrics [3]. From another point view, tools are not guarantied to be language independent. In addition, a significant limitation of the modern tools is their inability of extension. Indeed, it is often very difficult to define new metrics without putting a lot of efforts for the implementation.

Our goal is to develop a generic and flexible tool for metric collection that offers extension capabilities. In our approach, we provide a simplified language for metric description. Non-programmers use this language to define new metrics. Metric computation is done by an interpretation module. It evaluates the metric descriptions using information extracted from the source code and represented according to a generic meta-model.

This paper is structured as follows. In the next section, we present an overview of our approach. Section 3 is dedicated to the description of the program representation meta-model. Metric description language is explained in section 4. In section 3, we discuss briefly tool support issues. Finally, related works and a conclusion are presented in the two last sections.

2 Approach Overview

We propose a generic environment to collect metrics for object-oriented applications. Our environment, shown in figure 1, has two components: (1) the source code representation sub-system that is responsible for parsing and mapping programs and (2) an evaluation sub-system that implements a mechanism for metric collection starting from metric descriptions.

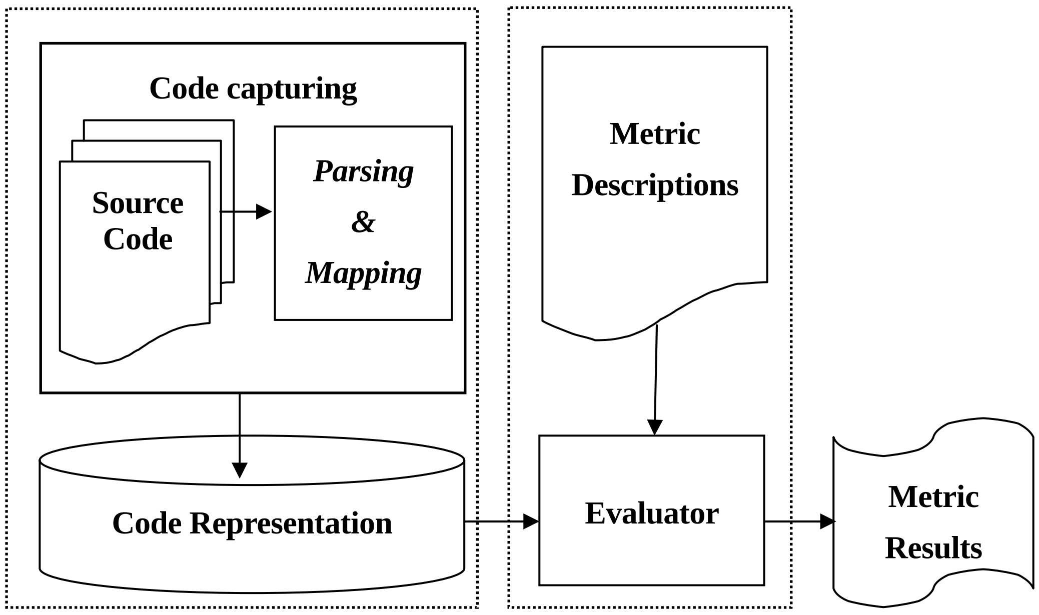

Figure 1: Architecture of the Metric Extraction System

The goal of the source code representation sub-system is the extraction of the necessary information, from the source code, that is used for metric calculation. For a given programming language, a module, named *parsing & mapping* (see figure 1), constructs a source code representation based upon a generic meta-model that we will describe in the next section.

The metric collection mechanism includes a description language and an evaluation module. We have defined this new language to describe metrics in a non-ambiguous way. It offers Primitives and Operators that are essential for the manipulation of data from the source code representation. The module named *Evaluator* in figure 1 interprets the metric descriptions and calculates the values accordingly. Thanks to this approach, the non-programmer has the ability to define, modify or customize a metric description when needed with a very reasonable effort.

3 Object-Oriented Program Representation

As stated before, metrics are computed using data extracted from the source code. To facilitate the manipulation of this data, we defined a meta-model that includes object-oriented programming key concepts. Indeed, it was designed to support typed object-oriented programs written in different programming languages. In order to adapt the environment to a particular programming language, a module should be written specifically for this language. This module generates a representation in accordance with the meta-model and taking into account the semantics of this language. This makes the metric extraction mechanism language independent.

Achieving independence from programming languages is not an easy task. In fact, even if some concepts are syntactically equivalent in different languages, their semantics might be significantly different from one language to another. In [6], examples of such variations are presented. However, it is important to keep in mind that we limit the quest of language independence to uniquely to those concepts that are concerned by the metric calculation.

When designing the meta-model, we considered three parts; each one corresponds to a category of concepts: (1) basic concepts for which the syntax and the semantic are similar for the different languages, (2) concepts with similar syntax but variation in the semantics, and finally (3), concepts that are specific to the considered languages.

Basic concepts, for which the transformation is straightforward, correspond to the common concepts of object-oriented languages such as classes, methods and attributes. They are represented in the meta-model by the notions *ClassDef*, *Method* and *Attribute* respectively.

Inheritance is one of the concepts that have variation in the semantic from one language to another. To circumvent this problem, the semantic of the concept is interpreted and explicitly represented during the mapping of the source code. In the case of inheritance relationship, methods and attributes are duplicated in the sub-classes according to the semantic of the considered language. Therefore, when comes the time of the calculation of a metric, its description is interpreted without any adaptation to the source language. In addition to inheritance, the other interpreted concepts are polymorphism (gathering overloaded methods), attribute access (for each method, the manipulated attributes are gathered), method invocation (gathering methods called by each method), and scopes.

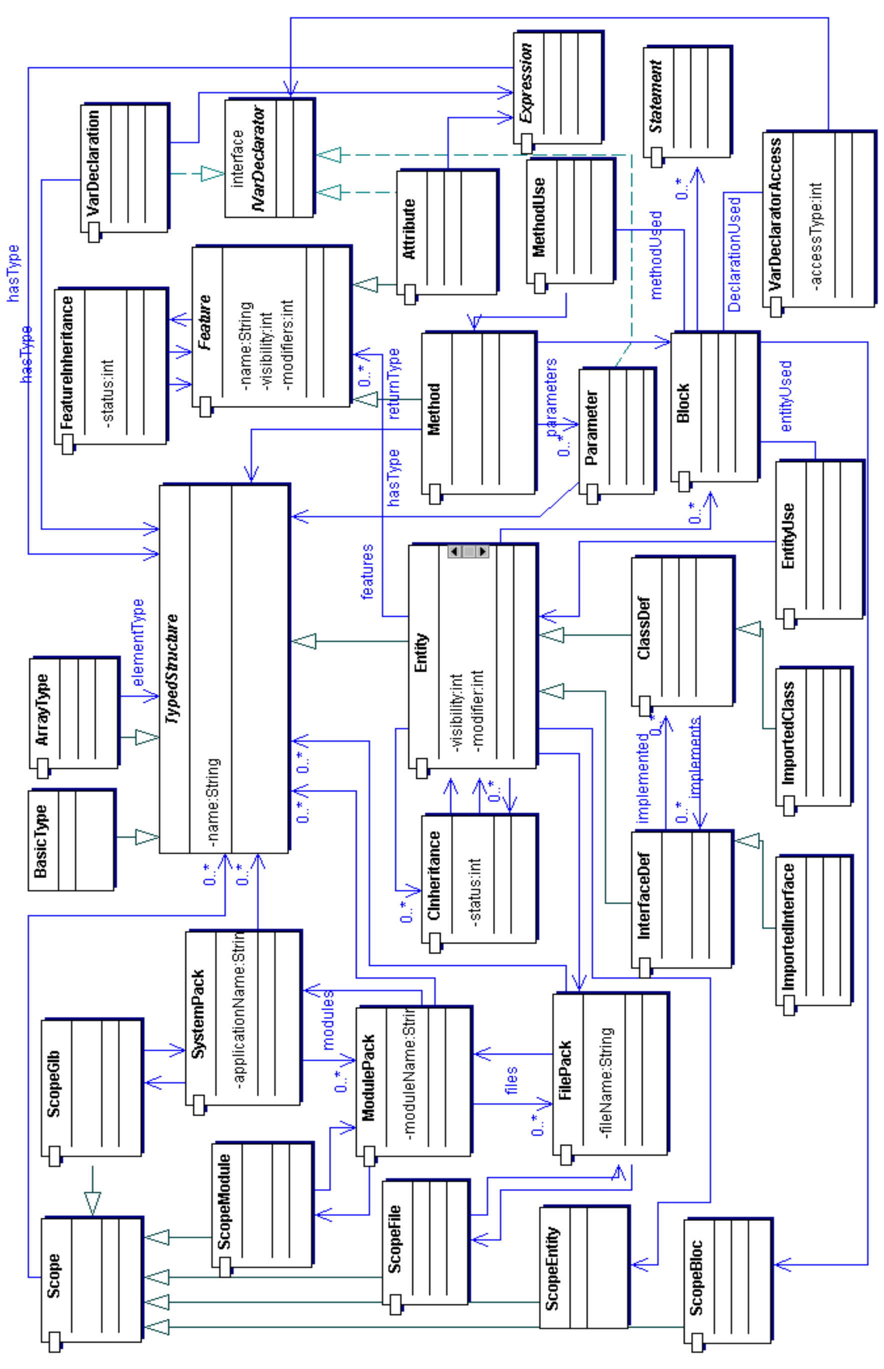

Figure 2: Main Part of the Meta-Model

Language specific concepts are integrated in the metamodel explicitly. For example, *Entity* represents the notion of abstraction in the different OO languages. It is specialized in Interface, Union, or Template that are present in Java or C++.

4 Metric Description Language

Metrics are calculated using information that relies on the source code representation. Therefore, to express a metric, the description language offers a set of Primitives and Operators that help manipulate the information within the code representation.

- Primitives: These are the base sets extracted from the code representation. They are inspired by the formalism proposed in [5]. Among those base sets we find the set of classes of the program (*classes()*), the set of methods of a class (*methods(c)*) and the set of parents of a given class (*parents(c)*). These primitives are implemented within our tool and can be used as a function call.

- Operators: They define the access to the object properties (i.e. attributes and associations) of the meta-model (figure 2). For instance, the operator *c.visibility* returns the visibility of the class *c* as defined in the source code. In the same way, the operator *m→ files* returns a set of files composing module *m*. This, knowing that *files* is the association label in the model.

- Operations on numbers and sets such as arithmetical operations, comparison operations, union and intersection operations as well as common functions (*min*, *max*, *sum*, etc.). There is also the cardinality operation which is heavily used, to compute size of sets. The notation of this operation is a set put between "|" symbol.

- Iterator: It enables the manipulation of a set's elements. The simplified syntax of this operator is :

  forAll *(x : inputSet ; condition_clause ;* ***SET*** *AssignOperator expression)*

  Where *inputSet* is a set. For each element *x* of *inputSet*, if the condition (namely *condition_clause* in the syntax definition) is evaluated to true, then the predefined variable *SET* receives the value of *expression*. The returned value is the set of data received by the *SET* variable during the iteration. This operator is commonly used to select a sub set of elements from the input set.

The language has only a few syntactic constructions which are quite simple. It does not require a high learning effort or any specific knowledge. Moreover, this language is a mean to calculate the literature's most popular metrics [7, 8]. In fact, the review of metrics of different types such as complexity, inheritance, cohesion and coupling, allows us to identify primitives and operators that should be in the description language.

We now have a list of about 40 metrics of different types that were expressed using our description language. The following table (Table 1) illustrates a small subset of this list.

CLS: Number of classes in the System	$CLS = \vert classes() \vert$
NIC : Number of Independent Classes	$NIC = \vert forAll\,(x : classe();\ \vert parent(x) \vert == 0\ \&\&\ \vert children(x) \vert == 0;\ SET += x) \vert$
AID: Average Inheritance Depth	$AID = \frac{sum(forAll(x : classe();\ ;SET += DIT(x)))}{CLS}$
CIS(c): Class Interface Size	$CIS(c) = \vert\ forAll\,(x : methods(c)\,;\ x.visibility\ == PUBLIC\ ;\ SET += x) \vert$
LCOM1(c): Lack of Cohesion in Methods	$LCOM(c) = \vert\ forAll(x : methods(c)\,,\ y : methods(c)\,;\ x\ != y\ \&\&\ (ar(x) \cap ar(y) \cap attributs(c)) == emptySet\,;SET += \{x, y\}) \vert$

Table 1: Some examples of system- and class-level metric descriptions

5 Tool Support

The development realized for the prototype concerns the source code representation part and the evaluation of metric description part. In the current version, the module for source code representation generates models for Java and C++ programs.

The evaluation module performs several tasks. It parses the description file, verifies the types of expressions, and, finally, evaluates them. These two last tasks are performed on the syntax tree which is generated during parsing. We used Flex[1] and Cup[2] to generate the parser.

In the previous section, we presented the main constructs of our language. For the syntax of some constructs, we used rules similar to those of Java. This is the case of function call, assignment, comparison operations and arithmetic operations.

For the data types, in addition to the numerical (integers and floats), Boolean and String types, we defined Set type and "representation" types such as class, method, attribute and parameter types, in order to manipulate objects from the representation model. Variables are not declared but parameters should be.

1 flex.sourceforge.net

2 http://www2.cs.tum.edu/projects/cup/

The metric extraction module could be used as a stand-alone tool, but it was designed as part of larger project called BOAP. This project consists in the definition of an integrated environment for quality evaluation and flaws detection, in which the metric extraction mechanism is part of the quality evaluation process.

6 Related works

To tackle the limitations of metric computation tools, a number of approaches are suggested. Mens and al. [12] defines a meta-model of source code representation as graphs. He also defines formalism for metric definition based on the graphs' nodes and arcs manipulation. The main limitation of this work is the limited number of metrics that we can express using this language.

The solution brought by Harmer [9] represents the source code using relational databases. He uses SQL requests to define the metrics. In spite of the declarative aspect of the requests, the complexity is similar to using a programming language.

Baroni and al. [3, 4] introduce OCL as a means to describe metrics. Since OCL is defined to express constraints on UML models, only design related metrics can be implemented. Metrics that involve implementation like some coupling measurement metrics cannot be directly represented using OCL.

7 Conclusion

In this work, we defined a meta-model for oriented-object programs representation and a rigorous language for metric description. Based on this approach, we offer a flexible and an extensible solution for metric extraction. A non-programmer can define new metrics with a moderate effort.

In a first time, our goal was to collect metrics from Java and C++ programs. Later, we plan to consider other languages. In the current state of progress, Java program representation is completed. For C++, a first version has been developed using a proprietary C++ parser. Due to numerous limitations found in this parser, we are exploring other possibilities. For the metrics, more than 40 have been described and tested using our tool. We are currently extending our environment to support other types of metrics.

The first experiences with our tool showed that it is convenient and easy to use it. However, more experiences in real world must be conducted to rigorously assess the flexibility, the extension capabilities, and the expression power of our metric description language.

References

1. Abreu, F.B., M. Goulao, and R. Esteves. “Toward the Design Quality Evaluation of Object-Oriented Software Systems”. Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, October 1995
2. Abreu, F.B. and W.L. Melo. “Evaluating the Impact of Object-Oriented Design on Software Quality”. 3rd International Software Metrics Symposium (Metrics'96), Berlin, Germany, March. 1996.
3. Baroni, A.L. and F. Brito e Abreu. “An OCL-Based Formalization of the MOOSE Metric Suite”. In Proc. of QUAOOSE'2003, at ECOOP'2003. Darmstadt, Germany. July, 2003.
4. Baroni, A.L. and F. Brito e Abreu. “A Formal Library for Aiding Metrics Extraction”. International Workshop on Object-Oriented Re-Engineering at ECOOP'2003. Darmstadt, Germany. July, 2003.
5. Basili, V.R., L. Briand, and W.L. Melo. “A Validation of Object-Oriented Design Metrics as Quality Indicators”. IEEE Transactions on Software Engineering, Vol. 22, No. 10, pp. 751-761, October 1996.
6. Beugnard A., “Method overloading and overriding cause encapsulation flaw: an experiment on assembly of heterogeneous components”, Proceedings of the 2006 ACM Symposium on Applied Computing , (SAC), Dijon, France, April 2006
7. Briand, L., J. Daly, and J. Wüst. “A Unified Framework for Coupling Measurement in Object-Oriented Systems”. Technical report ISERN 96-14, Fraunhofer Institute for Experimental Software Engineering, Germany, 1996.
8. Chidamber, S.R. and C.F. Kemerer. “A Metrics Suite for Object Oriented Design”. IEEE Transactions on Software Engineering, 20 (6), 476-493, 1994.
9. Harmer, T.J. and F.G. Wilkie. “An Extensible Metrics Extraction Environment for Object-oriented Programming Languages”. Proceedings of IEEE International Conference on Software Maintenance, Montreal, Canada, October 1 2002, IEEE Computer Society, ISBN 0-7695-1793-5. 2002.
10. Henry, S. and C. Selig. “Predicting Source-Code Complexity at the Design Stage”.IEEE Software 7, 2, pp. 36-44. 1990.
11. Li, W. and S. Henry. “Object-Oriented Metrics that Predict Maintainability”. Journal of Systems and Software, 23(2), pp. 111-122. 1993.
12. Mens, T. and M. Lanza. “A Graph-Based Metamodel for Oriented-Oriented Software Metrics”. Electronic Notes in Theoretical Computer Science, vol. 72, no. 2, 2002. http://h20000.www2.hp.com/bizsupport/TechSupport/Home.jsp