2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation

A Metric Extraction Framework Based on a High-Level Description Language

El Hachemi Alikacem
Centre de Recherche Informatique de Montréal
Montreal, Qc, Canada
alikacem@crim.ca

Houari A. Sahraoui
Département d'informatique et recherche opérationnelle
Université de Montréal
Montreal, Qc, Canada
sahraouh@iro.umontreal.ca

***Abstract*—Nowadays, many tools are available for metric extraction. However, extending these tools with new metrics or modifying the calculation of existing ones is often difficult, sometimes impossible. Indeed, many of them are black box tools. Others can be extended only by modifying third-party code. Moreover, metric specifications often lack precision, which leads to implementations that do not correspond necessarily to users' expectations. In this paper, we propose a flexible approach for metric collection based on a metric description language that allows manipulating basic data extracted from the code. These data are mapped to a generic object-oriented meta-model that is language agnostic. This makes it easy to focus on the metric specification rather than language specific constructs. Metric specifications are interpreted automatically to extract their corresponding values for a target program.**

Keywords-Metric extraction; object-oriented metrics; source code representation; meta-model.

I. INTRODUCTION

Metrics are powerful support tools in software development and maintenance. They are used to assess software quality [5], estimate complexity [10], predict cost/effort [11, 2], and control/improve processes. A huge number of metrics have been proposed [1, 2, 7, 8, 11] during the last two decades, and new metrics are introduced continuously. However, most of them are defined informally, using natural language. Due to this lack of formalization, different tools implement the same metrics differently, and the results may vary significantly depending on the language targeted by a metric extraction tool. This makes it very difficult to compare different analysis approaches that use the same metrics based on empirical results [3].

Most of the existing tools are black boxes. The specifications of the implemented metrics are embedded in the code and may not correspond to the users' expectations. Moreover, they are significantly limited by their inability to be extended to support new metrics. Indeed, it is often very difficult to define new metrics without putting a lot of efforts understanding and modifying the existing code.

In order to circumvent the limitations presented above, we propose an approach for metric collection that uses two main mechanisms. The first is a source-code representation mechanism which is based on a language-independent meta-model. The elements of the meta-model allow representing relevant information derived from the source code. The second mechanism is a declarative language to describe metrics. It is constituted of operations and a set of primitives to manipulate the elements of the source-code representations. These two mechanisms form a generic and flexible framework that allows users to define new metrics without addressing specifically the particularities of a programming language. Indeed, the meta-model is designed in a way that allows supporting many object-oriented (OO) languages. The metric description language is also generic in the sense that new metrics can be added without modifying the code of the framework.

The remainder of 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. The metric description language is explained in Section 4. Section 5 is dedicated to case studies that illustrate the specification and the collection of a representative set of existing metrics. Related work is discussed in Section 6, and concluding remarks are presented in Section 7.

II. OVERVIEW OF THE APPROACH

We propose a generic framework to collect metrics for object-oriented applications. The architecture of our framework, shown in Fig. 1, contains two main components: (1) the source code representation sub-system that is responsible for parsing and mapping programs and (2) the metric collection sub-system that implements the mechanism for metric specification and extraction.

The goal of the source code representation sub-system is the extraction, from the source code, of the basic elements that are necessary for metric computation. For each supported programming language, a module, named parsing & mapping (see Fig. 1), maps a program to a representation that conforms to the generic meta-model described in Section 3.

The metric collection sub-system includes the metric description language and an evaluation module that interprets and executes the metrics descriptions on the generated representations. As metric descriptions are intended to be interpreted and executed automatically, metrics are specified in an unambiguous way. The details of

978-0-7695-3793-1/09 $26.00 © 2009 IEEE
DOI 10.1109/SCAM.2009.27

the metric description languages are given in Section 5. The module Evaluator uses a generic procedure to interpret the metric descriptions and to compute the values accordingly.

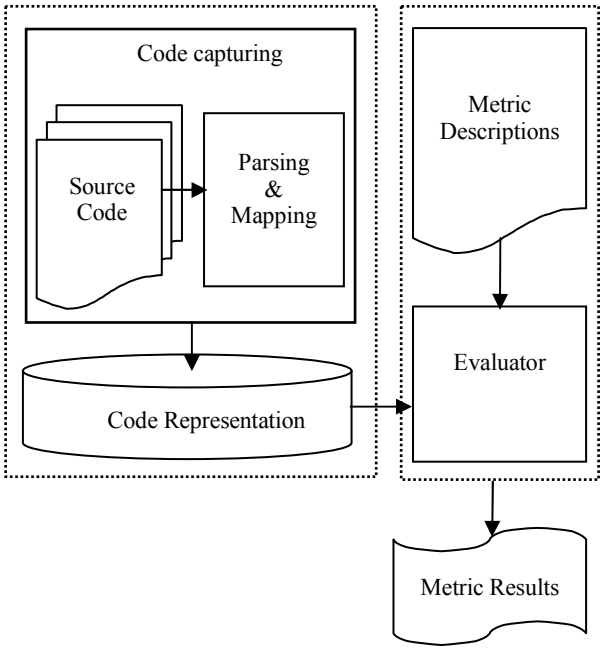

Figure 1. Architecture of the metric extraction framework

III. OO PROGRAM REPRESENTATION

As stated before, metrics are computed using data extracted from the source code. To facilitate the manipulation of these data, we defined a meta-model that includes OO programming key concepts. Indeed, this meta-model is designed to support typed object-oriented programs written in different programming languages (e.g., C++, C#, Eiffel, and Java).

Achieving independence from programming languages is far from being a trivial 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 semantics variations are discussed. Although the independence is difficult to achieve, in our case, the problem is made easier by the fact that we limit ourselves uniquely to those concepts that are relevant to the metric computation.

In the design of the meta-model, we consider three parts, each corresponding to a category of concepts:

1. **Common concepts** for which the syntax and the semantics are similar for all supported languages. These are the common concepts of object-oriented languages such as classes, methods and attributes. In this case, the mapping is straightforward. Such concepts are mapped to the elements *ClassDef*, *Method*, and *Attribute* in the meta-model for instance (see the partial view of the meta-model in Fig. 2).
2. **Variable concepts** with similar syntax but variation in the semantics. A well-known example of this category is inheritance. Indeed, different languages have different operational semantics of inheritance with, in particular, different method-lookup strategies. To circumvent this problem, the semantic of these concepts 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, during metric extraction, the description is interpreted without any adaptation to the source language. Other examples of concepts that fall into this category are polymorphism (gathering overloaded methods), attribute access (collecting attributes manipulated by a given method), method invocation (collecting methods called within a given method), etc.
3. **Specific concepts** that exist only in a particular language. Language specific concepts are integrated in the meta-model explicitly. For example, *Entity* represents the notion of abstraction in the different OO languages (Fig. 2). It is specialized in Interface, Union, or Template that are present in Java or C++. In other words, we simply include in the meta-model the specific concepts of each language we consider.

Considering the three categories of concepts presented above, we obtain a generic meta-model in which both syntactical constructs and semantic aspects are mapped. General metrics are specified using the two first categories concepts. The third category is sometimes used at an abstract level. For example, in some metrics we can count the entities independently from their specific nature. Language specific metrics are specified using the three categories.

As it is presented in Fig. 1, the code representation conforming to the meta-model is generated by a module *parsing and mapping*. Thus, in order to adapt the framework to a particular programming language, a module should be implemented specifically for this language. This module is responsible for identifying the three categories of concepts and mappings them according to their nature.

The proposed meta-model currently supports common OO concepts. Based on our experience with OO languages, we suggest that supporting new ones or considering new language releases will require very few changes in the meta-model, in addition to modifying/adding mapping modules.

IV. 4. METRIC DESCRIPTION AND EXTRACTION

A. Description Language

Our framework is designed for product metrics that can be calculated from the source code. These include design metrics (e.g., design coupling metrics). Extraction uses information derived from the generic representations. Therefore, to express a metric, we need mechanisms able to

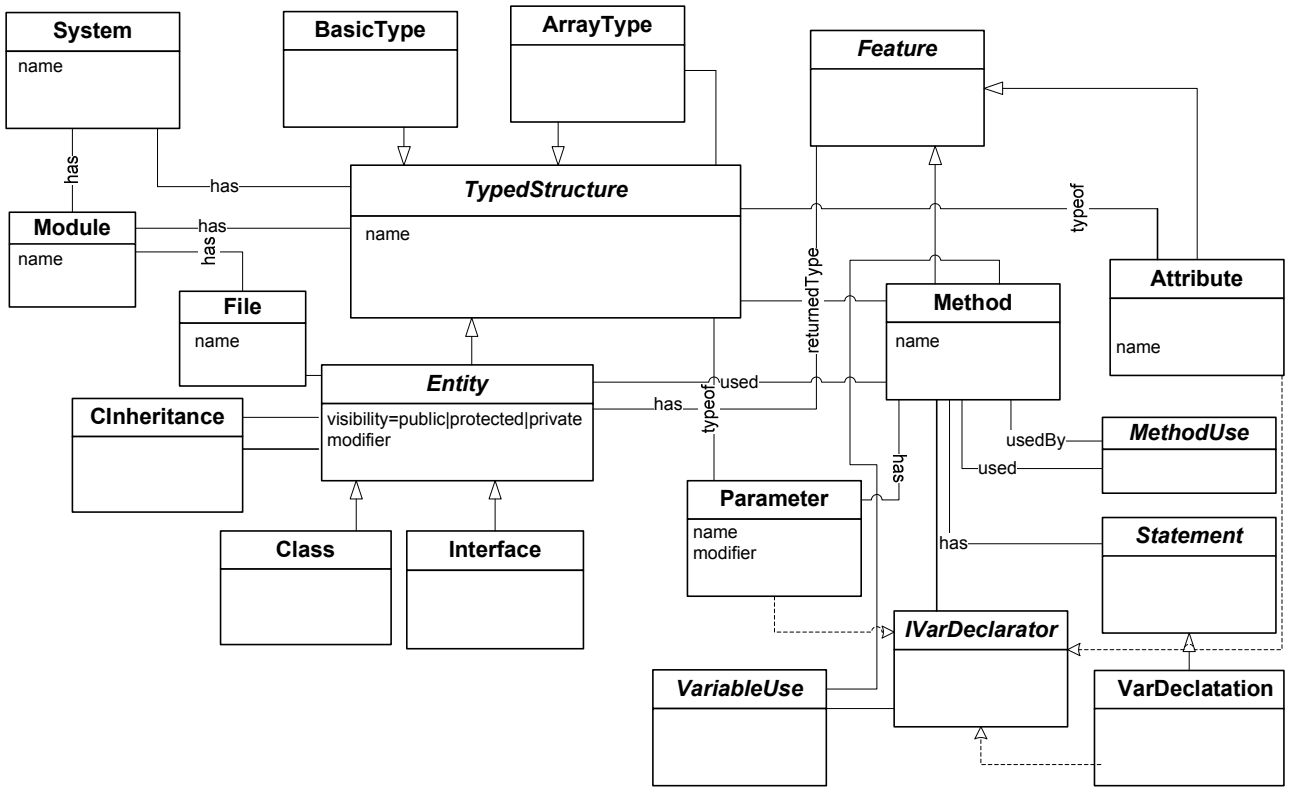

Figure 2. A Partial View of Source Code Meta-Model

access data in the representation and perform operations on them. Thus, we identify the following four main mechanisms that can be combined:

- **Selection**. Roughly speaking, in source code representations, occurrences (program elements) of a meta-model concept have properties and are related by different types of relationships. For many metrics, it is necessary to select the instances of a particular concept, e.g., all the classes of a program.
- **Filtering**. Selection allows collecting all the instances of the same concept. In many case, not all the instances are of interest. We need then to choose a subset whose elements share one or more properties, e.g., only the public attributes.
- **Navigation**. Selection and filtering concentrate on concepts and their properties. Navigation allows exploring the neighborhood of an element through its relations. As a result metrics that involve multiple concepts can be expressed. For example, by navigation, one can determine the set of attributes of a particular class or the parameters of a method.
- **Operations**. The three previous mechanisms allow mainly the derivation of sets of elements. To go from sets to metric values, specific operations are needed. These can be set or arithmetic operations. Comparators are also necessary for expressing conditions for filtering and navigation.

Starting from the four above-mentioned mechanisms, we designed the metric description language, named PatOIS (for Primitives, Operations, Iterator, and acceSsors), whose features are described in the remainder of this section.

The basic features of PatOIS are **Primitives**. These are procedures that allow selecting the base sets from the code representation. Primitives are hard-coded and are used as library functions when describing metrics. Two types of primitives are defined. The basic ones return a set of instances for a given concept in the meta-model. *classes()*, *interfaces()*, and *methods()* are examples of primitives that select the instances of respectively concepts *Class*, *Interface*, and *Method* for a particular program. The second type of primitives return a set of the elements related to a specified element by a particular relationship. Examples of these primitives are *methods(c)*, *parameters(m)*, *children(c)* that give respectively the set of methods of a class *c*, the parameters of a method *m*, and the subclasses of a class *c*. The third type of primitives is the user-defined ones. They are implemented either for complex calculation or to reuse recurring descriptions. Examples of these metrics are the calculation of the distance between a class *c* and a root/leaf classes in the inheritance tree.

The second important feature is **Operations**. Three categories of operations can be used: arithmetic (sum, minus, max, min, etc.), comparison (between numbers such as == and >, or between strings), and set (union, intersection, etc.) operations. A particular and useful operation is cardinality

that converts sets into numbers by giving their sizes. It can be applied to a basic primitive set such as |*classes()*| or relationship primitive such as |*parameter(m)*|.

Property access is the third category of features. They allow to access element properties. For example, the operation *c.visibility* returns the visibility of the class c.

The last feature is **Iterator**. It enables the manipulation of elements of a set. The simplified syntax of this operator is

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

For each element *x* of *inputSet*, if the condition (*condition_clause*) is true, then the predefined variable *SET* receives the value of expression. The returned value is the set of element in *SET* at the end of the iteration. For example, the set of public classes can be obtained by iterating on the elements of the primitive *classes()* with the condition that the property *visibility* is equal to *public*. Formally:

$$forAll\,(c : classes\,();c.visibility\ ==\ public\,;SET+=c)$$

The language has a very few simple syntactic constructs. It does not require an important learning effort. Moreover, as it will be seen in Section 5, it can be used to implement the commonly-used metrics such as complexity, inheritance, cohesion and coupling metrics with very few primitives and compact descriptions.

B. Evaluator Module

As we mentioned previously, metric computation is performed by the *Evaluator* module. The latter takes as input the metric descriptions file and evaluates it.

A metric description file is composed of two parts. The first part contains the individual descriptions of a set of metrics. The second part contains the operational statements that allow specifying the output format (text, tables, or histograms) and the computation scope. For example, class metrics can be computed on all the classes of a program or only on a subset satisfying a condition (same for method metrics).

The evaluator is a language interpreter. First, it parses the description file and generates a syntax graph. The parser is developed using Flex and Cup generators[1]. Then, it performs type checking on the graph nodes and tags them accordingly (primitive call, property access, etc.). Finally, it evaluates the nodes according to the implementation of the corresponding features.

V. CASE STUDY

To evaluate the usefulness of our framework, we first implement a large set of commonly used metrics at program level (10 metrics), class level (20 metrics) and method level (6 metrics). Metrics measure size/complexity, inheritance, coupling, and cohesion attributes. Some are at the design level, others at the implementation level. Examples of these metrics are given in the first subsection. Secondly, we use the framework on a set of C++ and Java programs with size varying between 100 and 400 classes (second subsection)

A. Implemented Metrics

When implementing the metrics, we identified four categories depending on features involved. We describe here these four categories.

1) Basic Metrics: These are metrics whose implementation requires mainly cardinality operators on basic primitives.

At the program level for example, the number of classes *CLS* is calculated as follows:

$$CLS() : \left| classes() \right|$$

Similarly, the number of methods *NOM* in the program or a class *c* is expressed respectively as:

$$NOM() : \left| methods() \right|$$
$$NOM(c : class) : \left| methods(c) \right|$$

Finally at the method level, the number of parameters *PAR* of a method *m* is simply described by:

$$PAR(m : method) : \left| parameters\,(m) \right|$$

2) Complex Metrics: For some metrics, complex constructs are involved (mainly Iterator). Consider, for example, at the program level, metrics *NIC*, measuring the number of independent classes, i.e., classes with neither parents nor children. Its calculation necessitates to iterate over the classes as follows:

$$NIC() : |\, forAll\,(c : classes();\ \left| parent(c) \right| == 0\ \&\&$$
$$\left| children(c) \right| == 0;\ SET+=c)|\,;$$

At the class level, the metric *ACAIC* (Ancestor Class-Attribute Import Coupling) is also a complex metric in which different features are involved in its implementation. This metric corresponds to the number of the attributes of a class *c*, whose types are within the ancestors of *c*. Its implementation is as follows:

$$ACAIC(c : class) :|\ forAll(a : attributes(c)\,;$$
$$isNew(a)\ \&\&\ typeof\,(a) \in ancestors(c);$$
$$SET+=a)\,|$$

In the implementation of *ACAIC*, in which iterator feature is used and the primitives *isNew* and *typeof* as well, we iterate over the attributes of the class *c*. The primitive *isNew(a)* is a predicate that returns true if the attribute is

1 Flex:flex.sourceforge.net
Cup: www2.cs.tum.edu/projects/cup/

defined in class *c*. The primitive *typeof (a)* returns the type of the attribute *a*.

The primitive *attributes(c)* (and *methods(c)* as well) returns the attributes (or methods) defined in the class *c* and those inherited from its ancestors. The predicates *isNew*, *isInherited* and *isOverriden* (only for methods) could be used to identify the status the attribute (or the method).

For some metrics, it is necessary to define intermediate calculations that can be called from the metric description. For example, *LCOM* [8] measures the lack of cohesion within a class *c* as the number of pairs of methods defined in *c* that do not access to the same attributes. Its calculation is based on the sets: *attributeDef(c)* (attribute defined in *c*, i.e., not inherited), *methodDef(c)* (same as for attributes), and *attributeUse(c, m, n)* (attributes defined in *c*, which are accessed jointly by methods *m* and *n*). These three sets are first described as follows:

$$attributeDef\,(c:class):forAll(a:attributes(c);\\ isNew(a);SET+=a)$$

$$methodDef\,(c:class):forAll(m:methods(c);\\ isNew(m)\,\&\&\,isOverriden(m);SET+=m)$$

$$attributeUse(c:class,m:method,n:method):\\ attributeDef\,(c)\cap attributeAccess(m)\cap\\ attributeAccess(n)$$

The metric is then described by call the above-mentioned sets:

$$LCOM\,(c:class):\\ |\,forAll\,(m:methodsDef\,(c),n:methodsDef\,(c);\\ m\;!=n\;\&\&\;attributeUse(c)==emptySet;\\ SET+=\{m,n\})\,|\,/\,2$$

3) Metrics based on other metrics: PatOIS allows the use of already implemented metrics to define new ones. Hence, already described metrics can be used as library functions and called when needed. Many metrics can be described concisely by reusing existing ones.

The metric *AID* (Average Inheritance Depth) [7] is such a metric which is based on other metrics. This metric is a ratio between the depth in inheritance tree for each class (metric *DIT* which is presented in the next sub-section) and the number of all classes (metric *CLS* presented previously).

$$AID\,():\\ sum\,(forAll\,(c:classes\,();;SET+=DIT\,(c)))\,/\,CLS\,()$$

The implementation of *AID* requires the use of the built-in function *sum*.

4) Metrics with specific primitives: In the previous examples, we used primitives that return a set of instances of a given concept in the meta-model. For some metrics more complex primitives are needed, for example, the metric *DIT* (Depth in Inheritance Tree) which represents the level of the class in its inheritance hierarchy. In the meta-model, inheritance is represented by a recursive link from the entity (e.g., class or interface) concept to itself. It corresponds to the inheritance relationship between an entity and its parent. Thus, an inheritance hierarchy is represented by a chain of entities. We defined a primitive, called *inheritanceLevel*, which returns the distance between two entities in the inheritance hierarchy. Therefore, *DIT* could be implemented by using this primitive. Its implementation is:

$$DIT\,(c:class):\max(forAll\,(d:ancestors\,(c);;\\ SET+=inheritanc\,eLevel\,(c,d)))$$

B. Program Mapping

In a first experiment, we considered C++ language. We used a parser, called Datrix, which produces an AST in text format. We implemented a mapping module to build the representation based on the meta-model. The constructs handled by the mapping module are classes, union, struct, template definitions, methods, attributes and functions. This module also mapped inheritance, method overriding, and friend relationship.

We applied our approach on a C++ system, called LALO. It is a multi-agent system developed internally, which contains about 120 classes. We implemented a number of metrics related to complexity mainly such as the number of classes, templates and other entities, number of methods and attributes. We also calculated some coupling metrics based on the number of classes used to declare the attributes of a given class. For instance, *ACAIC*, *OCAIC*, and *DCAEC*.

The parser Datrix produces an incomplete document (Text file of the AST) which does not contain some typing-related data. As a result, some concepts in the meta-model were not mapped (e.g., methods invocation and attributes access). Therefore, the calculation of some metrics was not performed, such as *LCOM*, *CBO* and metrics based on method calls and attribute access.

In the second experiment, we considered Java language, for which we implemented a Java parser that performs the mapping to all concepts of the meta-model. We used Cup[2] and JLex[3] generators for the implementation of the Java parser. It also includes Java typing which is needed to map specific concepts, e.g., method calls and attribute access.

The framework was applied on a set of small programs (less than 20 classes) for which it is easier to validate the results, on two medium size programs which contain 340 and 380 classes, and finally on one more larger program (almost 670 classes) to calculate over 35 metrics[4].

The results we obtained were verified manually. Actually, we calculated by hand all the metrics on the small programs and a subset of them (such as *DIT*, *CLS*, *NMA* and *NMN*) on the three larger programs used in the

2 www.cs.princeton.edu/~appel/modern/java/JLex/

3 www2.cs.tum.edu/projects/cup/

4 Tables 1,2, and 3 present a large selection of these metrics.

experimentation, and compared them with the values obtained by our framework. Some metrics, e.g., coupling metrics, are complex to calculated by hand on non small programs.

CLS: Number of classes	Number of classes defined in the whole program.
NBM: Number of module	Number of all the modules of the program.
NBF: Number of files	Number of files of the program.
NIS: Number of interface	Number of all the interfaces defined in the program.
NIC: Number of Independent classes	Number of classes that do not have neither super classes nor sub-classes.
TBI: Total base interfaces of system	Number of interfaces that do not have neither super interfaces nor sub-interfaces.
AID: Average inheritance depth	Ratio between the depth inheritance tree for each class and the total number of classes.

TABLE I. TABLE TYPE STYLES

TABLE II. TABLE TYPE STYLES

DIT: Depth of Inheritance tree	Class level in the inheritance tree.
CLD: Class to Leaf Depth	Higher distance between the class and its sub-classes.
NOC/NOP: Number Of Children/Parents	Number of direct sub/super classes.
NOD/NOA: Number Of Descendants/Ancestors	Number of all sub/super classes.
NMO/NMI/NMN: Number of Methods Overridden/ Inherited/New	Number of methods Overridden/ Inherited/New.
NMA: Number Of Attributes	Number of all the attributes. It includes the inherited ones.
ACAIC: Ancestor class-attribute import coupling	Number of classes, within the ancestors, used in attributes declaration.
OCAIC: Others class-attribute import coupling	Number of classes used in attributes declaration.
DCAEC: Descendants class-attribute export coupling	Number of classes, within the descendants, used in attributes declaration.
ACMIC: Ancestors class-method import coupling	Number of parameters whose type is within ancestors.
DCMEC: Descendants class-method export coupling	Number of parameters whose type is within descendants.
CBO: Coupling between Object	Number of other classes to which a class is coupled.
LCOM: Lack of Cohesion	Lack of cohesion in a class.

TABLE III. TABLE TYPE STYLES

PAR: Number of parameters	Parameters number of a method.
NEM: Number of external called method	Number of called method that are defined in other classes.
NEA: Number of external used Attribute	Number of accessed attributes that are defined in other classes.

The results of most of the metrics are obtained in a reasonable time (less then one minute for the larger programs). However, some metrics take a longer time to be calculated, such as *LCOM* and *CBO* since their implementations are complex, and iterate over many sets

As we mentioned previously, the Java parser performs typing. After implementing most of the typing specification [17], which is very long, we could use the framework to collect metrics on most of the existing Java programs. However, instead of completing the implementation of typing specification, we plan to implement a new mapping module based on Eclipse framework parser and migrate to Java 5 grammar at the same time.

VI. Related Works

Metric collection is the basis for any measurement program. The adoption of such program depends heavily on the availability of flexible and efficient measurement tools. In this context, a number of approaches/tools are proposed. Some of them adapt existing technologies for the purpose of computing metrics. Baroni et al. [3, 4], for example, use OCL as a means to express metrics. Since OCL is defined to express constraints on UML class diagrams, design-related metrics can be implemented mainly as post-conditions. On the other hand, metrics in which implementation-related data are involved, such as implementation coupling metrics, cannot be directly defined using OCL.

Similarly, Harmer and Wilkie [9] define a meta-model in the form of relational database schema, and use SQL to express metric calculations. However, for some complex metrics, SQL is mixed with a programming language.

The SQL-based approach is also used by Lavazza et al.[13], to collect only UML-based metrics (design level).

In the same family, El-Wakil et al. [14] propose the use of XQuery to compute metrics on UML models that are represented in XML documents. Eichberg et al.[15] also developed a framework, called QScope, for measuring software projects. It is built on top of Magellan framework in which all documents of a project are stored as XML documents. XQuery is used as a definition language to express metrics. This approach allows the user to collect metrics on different artifacts with the use of a uniform mechanism, i.e., XQuery.

Using XML technology to represent UML models is not a complex task when dealing with design artifacts. However, in the case of source code, its representation in XML is very complex. Therefore, the implementation of metrics requires less intuitive and very complex XQuery code.

The first limitation of the above-mentioned approaches is the lack of expressiveness of the languages used to implement many metrics. Indeed, these languages are not designed for this purpose. Moreover, they are by far more complex than our language because they use advanced programming language constructs.

The second family of approaches proposes dedicated formalisms/languages. Mens et al. [12] define an object-oriented meta-model as graphs and a formalism for metric definition based on graph manipulation. Using this formalism, they define three generic metrics: NodeCount, EdgeCount and PathLength, and a number of complementary higher-order metrics (e.g. ratio, sum, and average). Starting from these generic and higher-order metrics, more than 30 object-oriented metrics have been implemented. Although, this contribution is similar to our, it is limited to metrics that can be formulated in terms of nodes count, edges count, and path length.

Marinescu et al. [16] propose a simplified implementation of object-oriented design metrics. In their approach, a new interpreted language, called SAIL, is defined to express metrics. This language, which has similarities with a programming language augmented with SQL-style constructs, aims at simplifying and reducing the complexity overhead caused by the use of a programming language. Indeed, it offers constructs in order to implement key mechanisms, such as filtering, navigation, and selection. In comparison with our language, SAIL is at a lower abstraction level and can be rather compared to SQL queries.

VII. Conclusion

In this paper, we presented our framework for metric extraction, based on a high-level description language dedicated to metric computation. Our approach enables users to adapt existing metrics to their needs, and extend new metrics with reasonable effort. Moreover, it allows specifying metrics that can be calculated on programs written in different languages. The metric description language PatOIS allows combining existing primitives and eventually pre-existing metrics using operations and iterators to produce new metrics in an unambiguous way.

Using our framework, we implemented a variety of existing size/complexity, inheritance, coupling and cohesion metrics in the form of concise descriptions at program, class and method levels. Once, the language is understood few minutes are needed to implement most of the existing metrics compared to the long coding and debugging time when using a programming language. A screenshot of the framework is shown in Fig. 3.

Although the experiments showed that our approach is feasible and scalable, there is still a room for improvement. For example, the Evaluator module can be optimized to efficiently calculate metrics such as coupling metrics. Optimization techniques from compiler domain such as loop optimization might be considered. We also plan to consider implementation of mapping modules for other language such as C#.

References

[1] F. B. Abreu, 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] F. B. Abreu, 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.

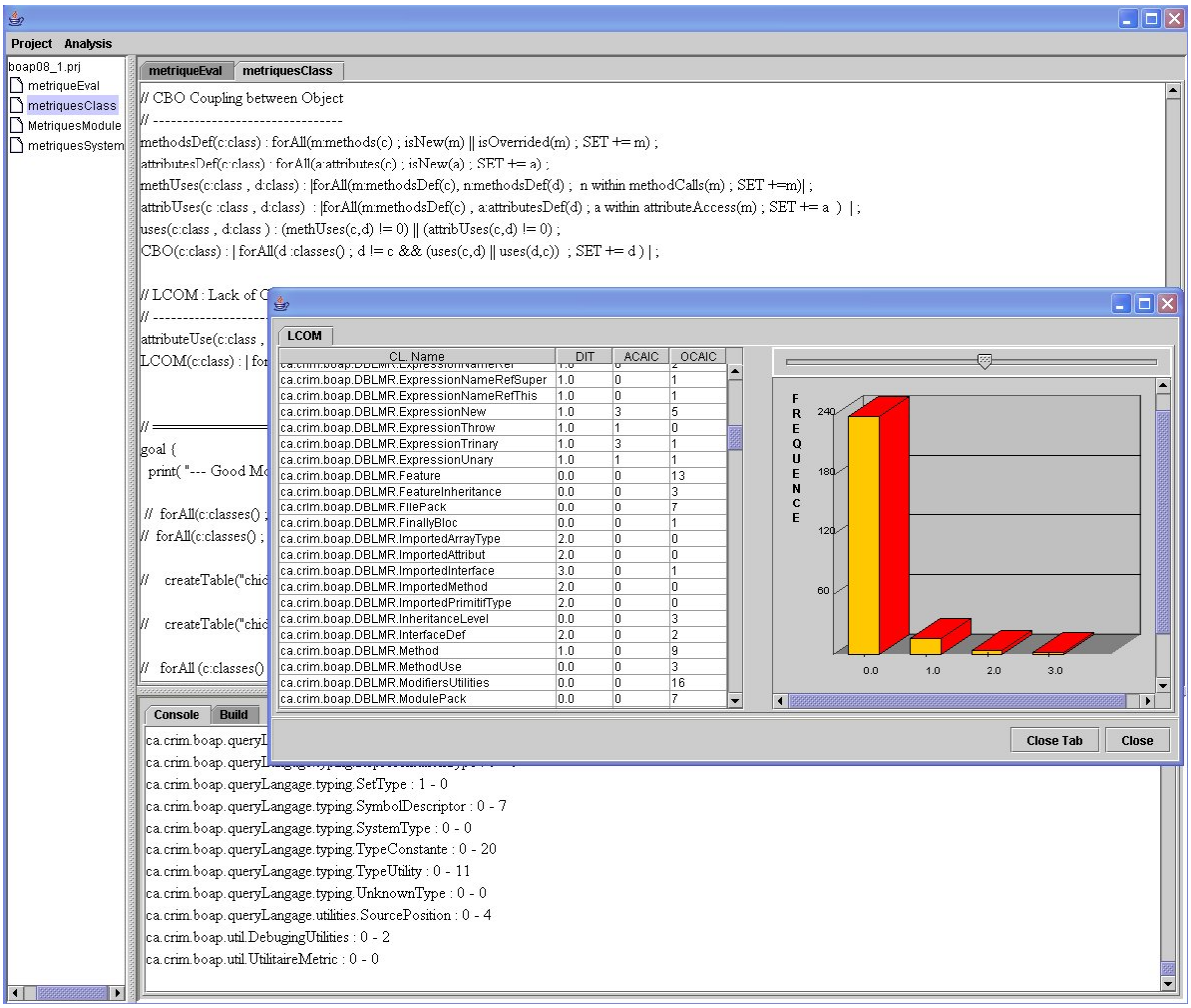

Figure 3. Screenshot of th framework

[3] A. L. Baroni, 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] A. L. Baroni, 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] V. R. Basili, 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] A. Beugnard, "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] L. C. Briand, J. W. 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] S. R. Chidamber, and C. F. Kemerer, "A metrics suite for object-oriented design," IEEE Transactions on Software Engineering, 20 (6), 476-493, 1994.

[9] T. J. Harmer, 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] S. Henry, and C. Selig, "Predicting source-code complexity at the design stage," IEEE Software 7, 2, pp. 36-44. 1990.

[11] W. Li, and S. Henry. "Object-oriented metrics that predict maintainability," Journal of Systems and Software, 23(2), pp. 111-122. 1993.

[12] T. Mens, 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.

[13] L. Lavazza, and A. Agostini, "Automated measurement of UML models: an open toolset approach," Journal of Object Technology, 4(4):115-134. 2005.

[14] M. El Wakil, A. El Bastawissi, M. Boshra, A. Fahmy, "A novel approach to formalize and collect object-oriented

design-metrics," In: Proceedings of the 9th International Conference on Empirical Assessment in Software Engineering. (2005).

[15] M. Eichberg, D. Germanus, M. Mezini, L. Mrokon, and T. Schäfer, "QScope: an open, extensible framework for measuring software projects," In Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06).

[16] C. Marinescu, R. Marinescu, and T. Gîrba, "Towards a simplified implementation of object-oriented design metrics," in IEEE METRICS, p. 11, 2005.

[17] J. Gosling, B. Joy, G. Steele, G. Bracha, "The Java Language Specification," 2nd Edition.