Measuring Discovered Models

Jean-Sébastien Sottet, Frédéric Jouault, Jean Bézivin
INRIA, Centre Rennes Bretagne Atlantique
4 rue Alfred Kastler
44307 Nantes, France
Email: firstname.lastname@inria.fr

Mathieu Venisse*, Vincent Fady[†]
Atos Origin, Systems Integration
6, impasse Augustin Fresnel
44812 Saint Herblain, France
**Email: mathieu.venisse@gmail.com,*

[†]*Email: vincent.fady@atosorigin.com*

***Abstract*—Model Driven Engineering (MDE) is based on the principle of unification. Most of the artifacts used or produced in a software project may be uniformly represented by models conforming to various metamodels. In this paper we show how two independent projects could be bridged to produce additional results. The first one is about discovering models from legacy code and the second one is about measuring various kind of models. Due to the interoperability properties of MDE, the integration of these projects was straightforward and allowed to provide a new generic legacy measurement infrastructure.**

***Keywords*-Model Driven Reverse Engineering; Quality of Models; Model Transformation Chains; Processes Reusability**

I. Introduction

Software design uses languages (usually UML) for specifiyng Object Oriented applications. Software analystes and designers have to optimize models that describe the application. Labor standards are developed to assist in the achievement of reliable, structured and maintainable software. Some of these standards can be automatically tested among design languages with some appriopriate tooling suite.

A Preliminary work has been done to analyze the quality of UML models using Model Driven Engineering (MDE) [1], using particularly Model to Model [1] (M2M) technology. This work was intended to do standard measurements on UML models thanks to a transformation chain.

In parallel, we lead a work on Model Driven Reverse Engineering of Java programs. This work advocates the use models obtained from a discovery tool. To do so, we followed the MoDisco approach [2] and build a Java discovery tool that creates models (i.e., Java Abstract Syntax Tree) from Java source code [2]. Additionally, we build a UML view (class model) of the Java discovered model for the comprehension of program structure.

The key purpose of this paper is to show that MDE, along with model transformations, enable an aided reusability of tools and processes. Particularly, we promote reusability transformation chains designed for different contexts. Thus, we reuse tranformations made for forward engineering (e.g., UML measurement) in the context of reverse engineering (e.g., Java Reverse Engineering). We demonstrate this reusability aspect on a representative case study.

The remainder of this paper is presented as following. We present a model-driven reverse engineering framework in section II. Then, section III proposes a model discovery case study base on the reverse engineering of Java source code. The section IV describes metrics and general measurement process used on UML class models. Finally, section V presents integration of UML measurement with model discovery.

II. Model Discovery

A. Overview

General process of model discovery is organised into three steps which carry out three layers (see Figure 1). From left to right, these layers can be summarized as:

- Injection layer: operates the translation of sources (code, CVS information, profiles, etc.) into one or more models.
- Management layer: permit the manipulations of models by the way of transformations (program query, translations, etc.)
- Extraction layer: operates the translation from models to output format that can be processes by a third party tool. It allows different interactions with the discovered model: specific computation, visualisation, measurement, etc.

B. Java Model Discovery

In this work we concentrate our effort on the reverse engineering of Java code. We want to obtain a first raw model that includes all Java code information. To do so, we reuse the Eclipse Java Development Tool compiler (JDT) for parsing Java code. The JDT API, including visitor, gives us a powerful tool for making reverse engineering of Java. Additionally, JDT provides an algorithm for resolving type binding.

The acquired model is an Abstract Syntax Tree (AST) conforming to the JDT AST metamodel inspired by JDT

[1]Eclipse foundation. Model To Model project (M2M). http://www.eclipse.org/m2m/

[2]The MoDisco project: http://www.eclipse.org/gmt/modisco/

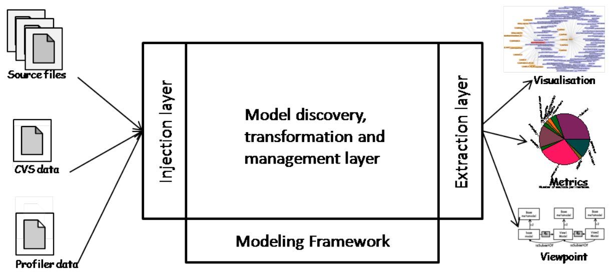


Figure 1. Overview of Model Discovery Architecture

	Number of Java classes	Number of model elements
set0	14	70447
set1	40	198466
set2	1605	2082841
set3	5769	4852855
set4	5984	4961779

Table I
JDT CASE STUDY: DIFFERENT JAVA PROJECT SETS IN RESPECT WITH SIZE.

documentation. This metamodel contains the Java program structure (classes, field and methods) as well as statements, expressions etc. JDT AST metamodel, thanks to a modeling framework, is plug in the JDT visitor. It creates the corresponding model element for each ”visited” Java element.

Of course, it is not as performant as the vanilla JDT parser but it shows very good performances. The version we developed can handle two hundred classes per seconds.

III. Case Study

Regarding MDE approach, one of our main objective is to test scalability. Finding large design models is not an easy task, while large open-source code is available everywhere.

A. Scalability benchmark

The Eclipse JDT seems, itself, a good candidate for scalability purpose. It consists in six thousand classes and corresponding model requires almost five million model elements. We split the original source code into different sets[3] composed of Java projects of growing size. So, the last set, set4, contains all the JDT sources as an AST model.

See Table I for information about set sizes in terms of number of classes and discovered model elements.

B. Java to UML2

Raw JDT AST model is too verbose to easiliy understand particular information about the Java code: we need a different perspective on it. Thus, UML2 class model brings an interesting view on the code structure. The class model does not contains control flow information. As a consequence, it

[3]Raw material available at http://www.emn.fr/x-info/atlanmod/index.php/GraBaTs_2009_Case_Study

Java AST meta element	UML2 meta element
TypeDeclaration	Class
Field (Primitive type)	Property
Field (Not primitive type)	Association + 2 AssociatioNEnd
MethodDeclaration	Operation
SingleVariableDeclaration (in MethodDeclaration)	Parameter

Table II
MAPPING HEURISTICS BETWEEN JAVA AND UML2 METAMODELS.

offers a reasonable reduced view of JDT AST model. We use the UML2 metamodel compliant with the OMG specification [3] and available under the UML2 Eclipse plugin.

There is not only one possible mapping between Java and UML2, so we use heuristics. The Table II gives a concise representation of heuristics used in our ATL[4] transformation rules.

In addition some mappings are not obvious. For example, Java provides a large set of collection (e.g., ArrayList,HashSet, etc.) that we need to refine into a association cardinalities (linked to association ends).

The benchmark (Table I) is composed of several subsets which gives incomplete Java AST model after injection. In the subsets we omit some JDT's Java projects; some referenced classes may have been ignored during the injection. Thus, the JDT AST model contains incomplete typing information. This may result an inconsistent UML2 class metamodel. We choose to tackle this issue by creating *proxy* classes in the UML2 model.

IV. Models Measurement

The UML measurement process is divided into four sub-processes. The first one transforms a UML2 model into an internal UML representation. The next two steps can be done in parallel: one is about generic measurement whereas the other one is parameterized by internal design rules. The final sub-process generates a report revealing the measurement results and some possible issues. This last step in this paper is out of the scope of this paper.

A. UML to internal UML representation

We use an internal UML representation. Firstly, because it facilitate the independence regarding UML versions and so reducing maintenance cost. Secondly, UML internal representation enhances the metrics computation. If the UML metamodel evolve, just a few transformations will have to be updated.

UML internal representation just keep the object-oriented structure: classes, attributes, operations, parameters. This model can be considered as a reduced view of the discovered AST model.

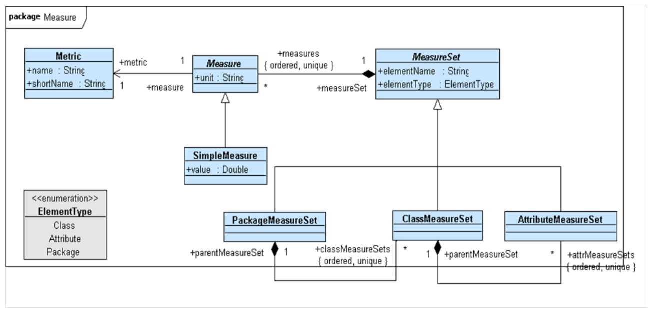


Figure 2. Generic metamodel for measurement

B. Generic Measure

For the internal model we use three kind of metrics: basics, Chidamber & Kemerer [5] and MOOD metrics [6].

The measures are based on a generic measurement metamodel presented in Figure 2.

1) Basic Metrics: these metrics enumerate the basic measures on UML elements (e.g., number of methods, number of classes, total number of methods in classes, etc.). They are used by the other metrics as basic elements for making more complexe computations. In order to be reused they are implemented as an ATL library of helpers.

2) Chidamber & Kemerer Metrics: these metrics are made for measuring the complexity of classes design and possible maintenances issues (e.g., antipatterns). This is organised around four main metrics:

- Weighted methods per classes: predicts time and effort needed to develop the class and its reusability.
- Depth Inheritance Tree: estimates classes reusability. A large inheritance tree is hard to understand due to excessive fragmentation.
- Number of children: shows the influence of the class on the system. This is a possible source of issues or a design mistake
- Couplage between objects: estimates the modularity, reusability and maintenance effort regarding this class.

3) MOOD Metrics: these metrics evaluate the system globally standing on inheritance and encapsulation by using several factors:

- Attributes inheritance
- Methods inheritance
- Attributes hiding
- Methods hiding

C. Control Design Rules

Generic metrics are known to be useful to complete some pre-defined tasks. However, many industrial actors (clients, team leaders, etc.) define internal design rules given a project. In addition, a design team leader may want to test precisely some design aspects. Control design rules offer this possibilty in the manner of a specific query engine.

A configuration DSL has been developed to cope with the building of control design rules. For example, (see Listing 1) it can be used for testing if a naming convention is applied: classes with stereotype A must start with A_.

Listing 1. Example of usage of DSL for control design rules

```
Configuration Example {
  rule NamingConvention1 'Operation in classes with
      ↪steretype A start with A_' {
    op : metadata <—(name), value <— A_;
    "class" : metadata <—(stereotype), value <—A;
  }
 }
```

This DSL is used,in addition to the tested model as inputs, by generic checking rules. It generates a report for checking which element enforce or agreed with the defined rules.

V. Discovered Models Measurement

The two previous sections seems apparently disjoined. We proposed two transformation chains: one for program comprehension (from Java code to UML classes diagram) the other one for measuring design quality. In this section we demonstrate that MDE approach facilitate the reusability of different transformation chains in different contexts.

A. Overview

The Figure 3 describes the joint use of the two transformations chains. The first chain of transformation (left of Figure 3 UML2 discovery chain) build an JDT AST model from Java source code (i.e., injection phase from Java grammar to modeling framework). Then it transforms the JDT AST model into a UML2 class model. This step skips a lot of information contained in the JDT AST model such as statements and expressions.

The second chain (right of Figure 3, metrics chain) takes a UML2 model as input. Thus, the previously transformed UML2 model will be reused here. The relatively independence of metrics computation is secured by the intermediate UML model.

The interoperability between these two transformation chains is ensured by the transformation from UML2 model to the internal UML model. As the UML2 model generated from discovered model may carry some incomplete information (e.g., particular cases unforeseen in heuristics) a few rules of *UML2 to internal* need to be modified.

B. JDT Discovered Model Evaluation

We only apply the transformation chain on set0 to set2 for performances reason. On a standard machine with 2GB of memory the Eclipse Modeling Framework (used for UML2 metamodel) does not scale above set2. However, some ATL transformations pass the set2 scalability test but use a different modeling framework [7].

On the one hand, the AtlanMod team provides the UML2 models from the injected Java code, on the other hand Atos Origin performs measurement transformations. We choose

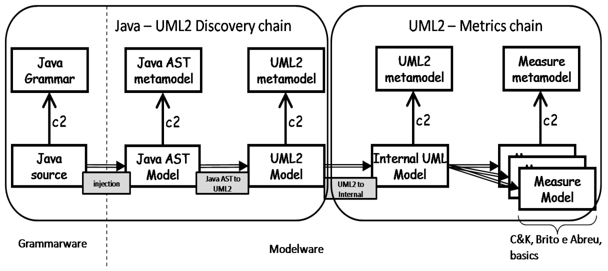
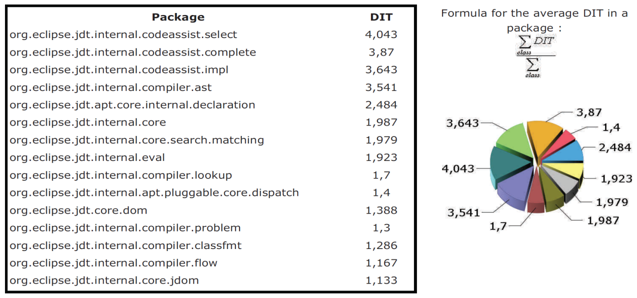


Figure 3. General process for discovering and measuring code-based model



Package	DIT
org.eclipse.jdt.internal.codeassist.select	4,043
org.eclipse.jdt.internal.codeassist.complete	3,87
org.eclipse.jdt.internal.codeassist.impl	3,643
org.eclipse.jdt.internal.compiler.ast	3,541
org.eclipse.jdt.apt.core.internal.declaration	2,484
org.eclipse.jdt.internal.core	1,987
org.eclipse.jdt.internal.core.search.matching	1,979
org.eclipse.jdt.internal.eval	1,923
org.eclipse.jdt.internal.compiler.lookup	1,7
org.eclipse.jdt.internal.apt.pluggable.core.dispatch	1,4
org.eclipse.jdt.core.dom	1,388
org.eclipse.jdt.internal.compiler.problem	1,3
org.eclipse.jdt.internal.compiler.classfmt	1,286
org.eclipse.jdt.internal.compiler.flow	1,167
org.eclipse.jdt.internal.core.jdom	1,133

Formula for the average DIT in a package :

$$\frac{\sum_{class} DIT}{\sum_{class}}$$

Figure 4. Average Depth Inheritance Tree (DIT) per packages in Set2.

to split the team's efforts to test the genericity of our approach. Of course, the transformation from UML2 to internal representation needed some tuning to work correctly. Nevertheless, it was not a complex task. The main modifications were induced by the heuristics used for generating UML cardinalities and parameters type from Java.

We produced a report using a measurement model extractor on our discovered model (extractor layer of Figure1). During the extraction phase measures were transformed into a specific XML file format which can be processed by an external tool: BIRT[4].

JDT is a very well designed Eclipse components: it should have good results. Of course, most of the measurements on JDT showed a good underlying design in terms of the metrics we used.

The example in Figure 4 presents a small part of the generated report: values of the average Depth of the Inheritance Tree (DIT) for packages. A value between 0 and 4 makes the compromise between performances provided and complexity induced by the inheritance. A value greater than 4 (such as in org.eclipse.jdt.internal.codeassist.select) for a package would compromise encapsulation and increase code complexity.

VI. Conclusion

In this paper we presented a MDE solution for measuring discovered models using M2M and MoDisco technologies. Thanks to the MDE approach and technologies we merged together two distinct works made for different contexts: design measurement and program comprehension. This work follows the OMG measurement standards for architecture modernization. Thus, the proposed measurement metamodel (Figure 2) is close to the OMG Software Measurement Metamodel (SMM) [8].

We did not aim at completely blurring the distinction between forward and reverse engineering. Nevertheless, we have demonstrated that MDE is a powerful approach for reusing some design processes (such as quality measurement) into a reverse engineering context. This integration of the model discovery and measurement processes was quite an easy task: only some transformation rules needed to be tuned.

For future work, we aim at targeting another UML models such as activity model. The Activity model gives another view on code, principally the control flow of a whole application, or just a method. Currently, we are leading a work based on Base UML (bUML) the subset of fUML [9] in order to have an executable view of Java discovered code.

[4]http://eclipse.org/birt/phoenix/

Acknowledgment

The authors would like to thank the AtlanMod Team. This paper has been partially supported by the Lambda project.

References

[1] M. Venisse, "Umlqualityanalysis: Uml models measurements with atl," in *Proceeding of the 1st International Workshop on Model Transformation with ATL (MtATL2009)*, 2009.

[2] J.-S. Sottet and F. Jouault, "Program comprehension," in *Proceedings of the 5th International Workshop on Graph-Based Tools*, 2009.

[3] *Introduction to the OMG's Unified Modeling Language (UML). Object Management Group, http://www.omg.org/gettingstarted/what_is_uml.htm.*

[4] F. Jouault, F. Allilaire, J. Bzivin, and I. Kurtev, "Atl: a model transformation tool," *Science of Computer Programming*, vol. 72, no. 3, Special Issue on Second issue of experimental software and toolkits (EST), pp. 31–39, 2008.

[5] S. R. Chidamber and C. F. Kemerer, "A metrics suite for objects oriented design," in *IEEE transactions on Software Engineering,*, vol. 20, no. 6, 2004.

[6] A. Lucia Baroni and F. Brito e Abreu, "Formalizing object-oriented design metrics upon the uml meta-model," in *In Proc. of the Brazilian Symposium on Software Engineering*, 2002.

[7] F. Jouault and J.-S. Sottet, "An amma/atl solution for the grabats reverse engineering case study," in *Proceedings of the 5th International Workshop on Graph-Based Tools*, 2009.

[8] OMG, *Architecture-Driven Modernization - Software Metrics Metamodel. http://www.omg.org/docs/ptc/09-03-03.pdf.*

[9] *Semantics of a Foundational Subset for Executable UML Models (FUML). http://www.omg.org/spec/FUML/.*