Coupling Static and Dynamic Models Information

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

***Abstract*—In Model Driven Engineering (MDE) a system may be represented by a model conforming to a given metamodel. The joint use of several models representing the same system is called multimodeling. In this work we show how different models representing the same legacy system may be used for program comprehension. More precisely, we show how to jointly exploit static (structural) and dynamic (behavioral) models of the same program to enhance understandability. The key advantage of MDE is that all models are based on a uniform representation and thus the joint use of these models is greatly facilitated.**

***Keywords*-Model Driven Reverse Engineering;**

I. Introduction

Nowadays, developing a software system is a complex task involving a lot of people and technologies. By using engineering methods and techniques software engineers aim at reducing systems development cost while improving its intrinsic quality. But engineers also face a crucial challenge: the maintenance of such complex systems. As stated in [1] program comprehension task represent an important portion of time consumed in software maintenance.

Doing program comprehension by hand is time consuming and error prone: software engineers need tool support. In order to manage information provided by comprehension tools, software engineers must be provided with flexible and extendable frameworks.

In addition, comprehending a software system only relying on a source of information such as an Abstract Syntax Tree (AST) is insufficient, as claimed by [2], [3]. The same authors advocate the need of looking at dynamic views acquired while executing the system (i.e., profiling, monitoring). Dynamic information is jointly exploited with static information in order to obtain some new information: called a fused[1] view.

The goal of this work is to demonstrate that MDE approach is useful for both unifying heterogeneous sources of information and providing enough flexibility for comprehension tasks. In that context views and fused views are also models conforming to metamodels. As a consequence they can be processed as any models and reused in generic transformation chains. In section II we present a model driven framework for reverse engineering. In section III we depict the view fusion (joint exploit) of dynamic and static views thanks to model transformations. Section IV present future work and open discussion.

[1]According to [3] the term fusion is used to denote the establishement of connection between views

II. Model Discovery

Maintenance projects face with both technology combination and modernization situations. MDE approaches offer the requisite abstraction level to build up flexible and extendable tools for maintenance. In this work, we follow the MoDisco [2] approach. One of MoDisco objective is allowing practical extraction of models from legacy systems.

The figure 1 is an overview of a model discovery architecture strongly inspired by the MoDisco framework. This architecture stands on a modeling framework such as EMF [3].

At left of figure 1, legacy system artifacts (Source code, CVS information, etc.) are transformed into models. This type of transformation is called ”injection”. It can be done either by a generic or a legacy-specific injection layer. Partical applications of this layer are described in the subsections II-A and II-B.

The middle of figure 1 presents a generic framework for managing and transforming discovered models. The application of this layer is the main concept of the section III.

At right, models of the system can be transformed into different formats.This type of transformation is called ”extraction”. The extracted information are meant to be used by third party software, for example a graph visualizer. In this paper, we only focus on the injection and transformation phases (left and middle layers).

MDE advocates the use of existing tools and technologies by making explicit bridges between them and metamodel concepts [4]. We promote the reuse of existing tools that are known to be performant in their domain without compromising genericity of models behind.

Data exchanged or produced by tools are unified by models conforming to explicit metamodels. In addition, we build transformations to ensure interoperability between technologies, manage data and compute views.

[2]The MoDisco project: http://www.eclipse.org/gmt/modisco/
[3]Eclipse Modeling Framwork http://www.eclipse.org/modeling/emf/

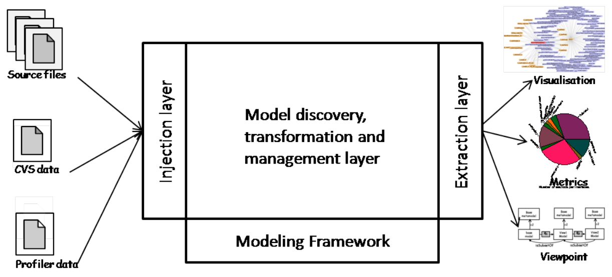

Figure 1. Overview of Model Discovery Architecture

Pratically, we use the following tools for building a static view (AST) and a dynamic view (Profile) of the studied system:

- The Java Developement Toolkit (JDT) for building AST model.
- The Java Profiler (HPROF) for having a model of the execution.

A. Linking Model Element to Tools API

Some tools provide Application Programming Interface (API) for plugging third party software. Making a metamodel based on this API facilitate the bridging between a used tool and a modeling framework. In that way data produced by the API are models conforming to the established metamodel. We have not enough space here to detail the AST metamodel but a version of Java AST metamodel is available on MoDisco website[4].

The Java model discovery system we develop uses the Eclipse Java Development Tools (JDT) parser. However, we do not directly query JDT's object model but store the decorated Abstract Syntax Trees (ASTs) as models (i.e., graphs). Therefore, we are a bit slower than the vanilla JDT parser but we are more generic. A modeling framework (EMF for example) is in charge of creating model elements when the JDT visit (using a specific implementation of JDT visitor) a Java element (e.g., statements, expressions).

Our early experiments with JDT have demonstrated good performances and scalability: it can manage around two hundred Java classes per seconds (without resolving binding) and handles large model of five millions elements. Furthermore, we had already proposed a benchmark [5] that consists in sets of large AST models obtained thanks to that discovery technique.

Despite AST gives a lot of clues about the program structure and may lead to some optimizations, it cannot precisely identify problem that occurs at run-time. For example the reaction of the system to particular parameters, to user inputs, etc.

[4]http://www.eclipse.org/gmt/modisco/technologies/JavaAbstractSyntax/

```
TRACE 301956:
org.amma.test.c.ClassC.getValue
org.amma.test.b.ClassB.getValue
org.amma.test.a.ClassA.getValue
org.amma.test.a.Intializer.main
[...]
CPU TIME (ms) BEGIN (total = 657)
rank self accum count trace method
1 16.74% 16.74%   9 302155 org.amma.test.c.ClassC.
getValue
2 16.74% 33.49%   1 302183 org.amma.test.a.
Intializer.main
3 16.44% 49.92%   9 302157 org.amma.test.a.ClassA.
getValue
4  9.59% 59.51%   9 302156 org.amma.test.b.ClassB.
getValue
5  7.00% 66.51%   2 301956 org.amma.test.c.ClassC.
getValue
```

Table I
EXCERPT OF A PROGRAM TRACE AND CPU TIME USING HPROF TEXTUAL OUTPUT

B. Injecting Model from Textual Syntax

Tools do not always offers an open API. Nevertheless, they often provide textual or binary output file. Thanks to a parser linked with a metamodel, we may be able to produce models from the studied tool output files.

For this purpose we use Textual Concrete Syntax (TCS) [6]. TCS is an Eclipse/GMT component that enables the specification of textual concrete syntaxes for Domain-Specific Languages (DSL) by attaching syntactic information to metamodels.

The Java profiling tool, HPROF[5] can be used to track performance problems involving memory usage and inefficient code. This tool provide a simple textual format for program traces.

The HPPROF example,in Table I describe a series of method calls from main program (Intializer.main) to a succession of method invocations (*getValue*) among different classes(ClassA, ClassB, ClassC). In addition, the HPROF tool gives us a view of the percentage of CPU consumed time so as the number of call of a function per traces. However, we may not have enough information to retrieve all program infrastructure (e.g., some program part may have be escaped regarding some execution value).

We build a HPROF metamodel inspired by this textual format. We propose an excerpt of HPROF metamodel written KM3 [7] in Listing 1. Each traces will be represented as a *Trace* class with identification number attribute and a container reference to *Method* class. In addition the class *Time* contains some values, such as percentage of CPU used by a method regarding a particular trace. Every class should extend *LocatedElement* class, directly or indirectly. This is a technical constraint to support text-to-model traceability.

[5]we run the HPROF tool using the command line: -Xrunhprof:cpu=times,heap=dump,interval=1,lineno=y,depth=1000

Listing 1. Part of HPROF metamodel (written in KM3) with regard to program traces and CPU time

```
class Trace extends LocatedElement {
    attribute id : Integer;
    reference methods[*] ordered container : Method
        ↪oppositeOf traces; }

class Method extends LocatedElement {
    attribute fullyQualifiedName : String;
    reference traces[*] : Trace oppositeOf methods; }

class Time extends LocatedElement {
    attribute rank : Integer;
    attribute self : Double;
    attribute accum : Double;
    attribute count : Integer;
    reference trace : Trace;
    reference method : Method;  }
```

The TCS excerpt (Listing 2), along with HPROF metamodel(see Listing 1), describes a text-to-model transformation that creates *Time* model elements from HPROF textual syntax.

Listing 2. TCS excerpt that create Time classes and attributes from textual syntax

```
template Time
  : rank self "%" accum "%" count trace{refersTo = id}
    method{refersTo = fullyQualifiedName}
  ;
```

III. Model/View Fusion

In order to enhance the precision of views for comprehension, different models can be jointly exploited. Multimodeling is the MDE techniques that exploit different models of a same system. Thus, view/model fusion can be considered as a part of multimodeling. The result of fusion is also a model: it can be exploited as any models. We focus here on the manual coordination of information supported by models. The coordination is based on a metamodel definition for each fused models.

Regarding our example we have, on the one hand an AST model (we put the focus on static call tree), on the other hand we have a profiling metamodel. This last one contains execution time, number of calls and percentage of resources used for each method. But here, the idea is to identify cost of method call, number of call and type of call (e.g., if it is just a method propagation) by fusing AST (call tree) and profile models together.

A. Specifying Fused Metamodel

The first thing to do is to specify the structure of the fused view as a metamodel. Coarsely, it can be defined as the cartesian product of views to be fused. Actually this is a bit more difficult, we have to design a specific combination of model elements that can be of any nature (this list is non-exhaustive):

- clarify inaccurate or incomplete model information.
- add information *versus* simplify model.
- decorate models.

Actually, we want to *decorate* a static method call graph obtained from AST model with information concerning methods call provided by the profile model.

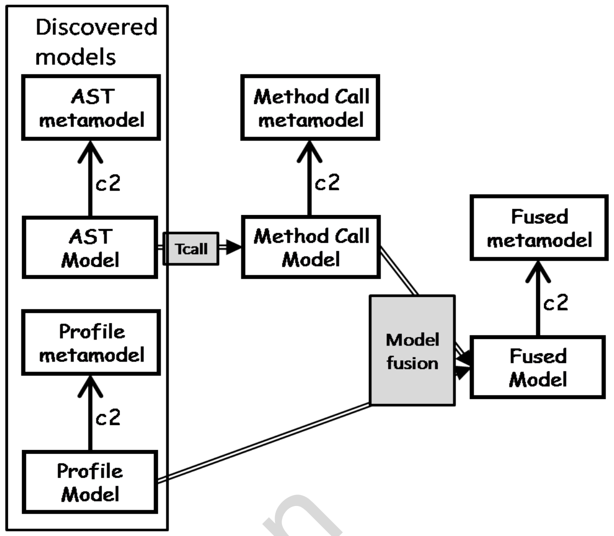

Figure 2. Transformation Flow with the conformance relation (C2)

B. Fusion by Model Transformation

The figure 2 depicts the transformation chain used in this example. It is composed of a transformation that computes a method call tree and the fusion transformation. The first phase (transformation *Tcall* in figure 2) transforms an AST model into a static call tree. It contains, method invocations and references to their declaration, see Listing 3. The main method declaration is the root of method invocations control flow. We will not describe this transformation in this paper. It has the benefit to reduce the AST model by removing superfluous model elements for our purpose. The static call tree model gives the base structure that will be decorated by the profile model. All classes inherit from *AbstractNode* class which contains the fully qualified name of each methods.

Listing 3. Static call tree metamodel

```
class MethodNode extends AbstractNode {
  reference invoke[*] ordered container :
      ↪MethodInvocationNode oppositeOf isInvoked;
  reference invocations [*] : MethodInvocationNode
      ↪oppositeOf declaration; }

class MethodInvocationNode extends AbstractNode {
  reference isInvoked[0-1] : MethodNode oppositeOf
      ↪invoke;
  reference declaration[0-1] : MethodNode oppositeOf
      ↪invocations; }
```

We use The AtlanMod Transformation Language (ATL) to build model transformations. ATL provides a concise and declarative syntax to express model transformation based on OCL expressions. In addition, the ATL engine has shown good scalability and performances as demonstrated in [8]. Achieving model fusion is quite a standard task as ATL can support multiple input models. The transformation excerpt Listing 4 describes a rule that makes the fusion between two models. The rule input pattern is composed of two elements j_me and p_me that designate respectively Java Method and Profile Method classes. We put a guard on the qualified names of the methods otherwise the ATL engine would have done the cartesian product between every AST methods and HPROF methods. This transformation associates the total number of CPU time use with the corresponding method in static call tree model.

Listing 4. Merging AST Method with HPROF Method information
```
rule MethodNodetoMethodDeclaration {
  from
    j_me : M_CFG!MethodNode,
    i_me : HPROF!Method(i_me.fullyQualifiedName = j_me.
         ↪fullyQualifedName)

  to
    o : MCFGSD!MethodNode(name<-j_me.name,
      fullyQualifiedName<-j_me.fullyQualifedName,
      sumOfInvocation<-i_me->collect(e | e.trace)
                                  ->collect(e | e.times)->flatten
                                       ↪()
                                  ->iterate(e; res : Integer = 0 |
                                       ↪ res + e.count))
```

IV. Discussion and Future Work

Multimodeling uses several MDE techniques, particularly model weaving [9]. Model weaving is a technique that makes an explicit model (called weaving model) based on links between two (or more) models. Instead of writing a transformation, as explained is this paper, we could have defined a weaving model between AST and profile models. The whole set of models: weaving, AST, profile, represent the fused view. This definition of fusion is closer to the original one in [3].

In addition a weaving model can be achieved thanks to automatic model matching [10]. The process of model matching stands on the semantic closeness of model elements into two different models. In our example, method calls are common concepts (with common names) in the two models. As a consequence, we are able to automatically make references between these two models. Thus, even a raw algorithm that makes cartesian product of models elements is able to build interesting views on the basis of matched elements.

V. Conclusion

This paper has presented a MDE solution for jointly exploiting models based on the reverse engineering of different sources from the same system. MDE methodology provides a unified representation of models conforming to metamodels. Thanks to that unifying power, we are able to manage heterogeneous sources of information as one.

MDE tools do not need adaptation for managing multiples models. In the example described in this paper, we shown that the standard ATL language performs the fusion transformation. In addition, ATL has already demonstrated that it fits with industrial concerns on program comprehension in terms of perfomances and scalability. The next step is to test MDE against existing approaches, on a more complex comprehension tasks, .

However, software engineers may feel disappointed regarding the genericity power of MDE: ATL is general purpose transformation language. Putting the focus on comprehension tasks is necessary. In conjunction with the elaboration of a complex use case, our future work will be to build a transformation DSL for program comprehension that lead to more usability for software engineers.

Acknowledgment

The authors would like to thank all members of the AtlanMod Team. This work has been partially supported by the Lambda project.

References

[1] R. K. Fjeldstad and W. T. Hamlen, “Application program maintenance study: Report to our respondents,” in *GUIDE 48*, 1983.

[2] L. Qingshan, C. Hua, H. Shegming, C. Ping, and Z. Yun, “Architecture recovery and abstraction from the perspective of processes,” in *Proceeding of the 12th Working Conference on Reverse Engineering (WCRE’05)*, 2005.

[3] R. Kazman and S. Carriere, “View extraction and view fusion in architectural understanding,” in *Proceedings of the Fifth International Conference on Software Reuse*, 1998.

[4] J. Bezivin and I. Kurtev, “Model-based technology integration with the technical space concept,” in *Proceedings of the Metainformatics Symposium, MIS2005*, 2005.

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

[6] F. Jouault, J. Bezivin, and I. Kurtev, “Tcs: a dsl for the specification of textual concrete syntaxes in model engineering,” in *Proceedings of the fifth international conference on Generative programming and Component Engineering, (GPCE’06)*, 2006.

[7] F. Jouault and J. Bezivin, “Km3: a dsl for metamodel specification,” in *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, 2005.

[8] 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.

[9] M. Didonet Del Fabro and P. Valduriez, “Towards the efficient development of model transformations using model weaving and matching transformations,” in *Software and Systems Modeling SoSyM*, 2008.

[10] K. Garces, F. Jouault, P. Cointe, and J. Bezivin, “A domain specific language for expressing model matching,” in *Proceedings of the 5ime Journe sur l’Ingènierie Dirigée par les Modèles (IDM09)*, 2009.