ModelExtractor: an Automatic Parametric Model Extractor

Jean Bézivin, Régis Chevrel, Hugo Bruneliere, Albin Jossic, William Piers, Frédéric Jouault

▶ To cite this version:

ModelExtractor
an Automatic Parametric Model Extractor

Jean Bézivin, Régis Chevrel, Hugo Brunelière,
Albin Jossic, William Piers, Frédéric Jouault
ATLAS Group (INRIA & LINA, University of Nantes)
{bezivin | chevrel.regis | hugo.bruneliere| jossic.albin | william.piers | f.jouault}@gmail.com

Abstract

Building on previous experiments of reverse engineering with the Squeak language, we have designed a new prototype on the DotNet platform that should be able to exploit the full capabilities of the Visual Basic 9.0 language, including the facilities offered for dynamic typing. The objective of this work is to prove that we are able to implement an automatic model mining facility when the system consists of a legacy with good reflectivity capabilities. The initial work presented in this paper comforts the previous results obtained for Squeak and constitute the basis of a reverse engineering development toolkit that is intended to be extended in the coming months.

General terms
Reverse engineering, model extraction

Keywords
Parametric metamodel, tagged metamodel, dynamic model, MDRE

1. Introduction

Industrial returns of experience show that software maintenance projects are difficult, especially when the managed projects include different domain programs written by different peoples. Model Driven Reverse Engineering (MDRE) is a powerful possible response. Reverse engineering is an attractive technique to assist the understanding of unknown programs but the lack of fixed standards increases the management effort. Model Driven Engineering (MDE) provides some answer to this interoperability problem by using models as representations. MDE, particularly the Model Driven Architecture (MDA™), suggests using models to represent systems. These models conform to metamodels.

MDRE solutions have been already implemented, like (e.g. Modelisoft [6] in the DotNet environment), and are capable to extract models from a system (a program). This kind of work needs a reflexive language that permits introspection, like Smalltalk, DotNet or Java. But these extractions are limited to a specific fixed metamodel and produce static models based on a static point of view of the considered system.

We describe a work which is in process at INRIA in the ATLAS team in Nantes. Our aim is to offer a new solution with a tool capable to extract models from dynamic systems with parametric metamodels. Our approach is to give a decorated metamodel to our extractor which can analyze a program in execution and produce a model conforming to any metamodel. The metamodel is a view definition on the system modeled by the produced model. The decorations will guide the exploration by giving information about the local step knowledge extraction and about the next step discovery.

Another particularity is to implement this solution in the Microsoft DotNet technological space [4] while most of our tools are presently working on top of the Eclipse Modeling Framework (EMF) [5]. We permit inter-exchange between these two spaces by serializing the produced model in Ecore format and by providing an Ecore reader/writer for Visual Basic DotNet. So we can use the Atlas Model Management Architecture (AMMA) facilities [1] to define our models, in particular Kernel MetaMetaModel (KM3) to define metamodels and Atlas Transformation Language (ATL) to transform models.

This paper is organized as follow: section two presents the AMMA platform, principally KM3 and ATL. Section three introduces Microsoft DotNet architecture and Visual Basic. Section four presents some theory about models and metamodels relations and about the role of decorations. Section five defines the Data Structure For Models (DSFM) which defines a basic data structure to represent models and metamodels in memory. Section six describes our actual implemented prototype before section seven presents some ideas for extensions.

2. Presentation of the AMMA platform

The AMMA platform proposes a complete set of tools for models manipulation. It is composed of four main tools:

- ATL[1]: Atlas Transformation Language. A QVT-compliant model transformation language based on metamodel definitions and OCL rules. It has a large and rapidly growing user community.
- AMW[3]: Atlas ModelWeaver. A metamodel agnostic tool allowing to build weaving models between any kinds of models.
- ATP: Atlas Technical Projector. Defines a set of injectors/extractors enabling to import and export models from/to foreign technical spaces (Java classes, Relational databases, etc.).
- AM3: Atlas MegaModel Management. Defines the way the metadata is managed in AMMA (registry on the models, metamodels, tools, etc.). Visit the Atlantic zoo [2] to get an idea of some global model management possibilities of AM3.
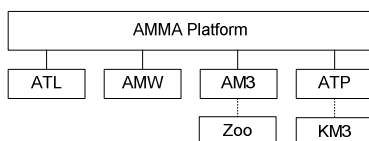
Figure 1 AMMA Platform overview

All of these tools use KM3 as a common language to define metamodels. KM3 defines a neutral metametamodel. The AMMA platform provides MOF, Ecore and others injector/extractor to KM3 (Figure 2).
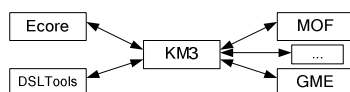

Figure 2 KM3 as a pivot

Two of the tools are particularly interesting for our work: ATL and KM3. KM3 provides the main ideas for the construction of our DSFM format which has to represent models and metamodels in memory. ATL allows us to transform a model from its native form to any kind of target metamodel.

3. Presentation of DotNet and VB

As we say, reverse engineering may be facilitated if we have a reflective language to operate. DotNet introduce this possibility by using a virtual machine to execute program from Visual Basic DotNet for example. The source code is translated to pseudo code, the Microsoft Intermediate Language (MSIL) which is interpreted by the virtual machine, Common Language Runtime (CLR). This pseudo code is called assembly. Figure 3 summarizes the DotNet mechanism.

Figure 3 DotNet global mechanism

Another need we have is to compile a generated source code to MSIL. In fact, exploring a decorated metamodel produces a kind of parser which can really extract information from the system to construct the target model. This parser is generated in text form, a Visual Basic source code. Then, we need to run this program automatically by calling the main method of the produced assembly. DotNet provide code compiler with these two interfaces:

- System.CodeDom.Compiler.VBCodeProvider (framework DotNet)
- Microsoft.VisualBasic.VBCodeProvider (Visual Basic API)

So, we can instantiate a compiler and run it on a string source code. This operation returns an executable assembly.

4. What is a model?

We now briefly make explicit the relation between models and metamodels. We say that a model conforms to a metamodel when all the entities which compose the model are related to a metaentity in the metamodel. The association between a class and its metaclass is called the μ link. It is comparable to the *instanceOf* EMF link but as the vocabulary infers lot of confusion, we explicitly rename it to μ.

In the MDA approach, we consider three levels in the metamodeling hierarchy:

- Metametamodels (M3): define the semantic of the metamodels and itself.
- Metamodels (M2): are conforming to M3 and define the semantic of the models.
- Models (M1): are conforming to M2 and represent the real world, the systems.

The Figure 4 sums up these concepts.
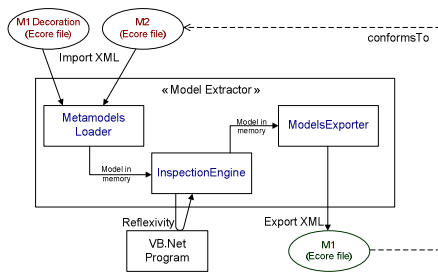

Figure 4 Relations between models and metamodels

The systems we consider here are programs. But a program is also a model which represents the real world. So we can transform it to obtain the wanted conformance to a specific metamodel.

The decorations define this transformation by specifying the local knowledge extraction and the next step to go. Decorations are like an executable weaving which automatically produces rules. Figure 5 illustrates the decorations machinery.

Figure 5 Decoration general mechanism

5. Model representation

To manipulate models, we need to define a structure which can store these and help us to manipulate and navigate. This is what the DSFM provides. This storage facility is inspired from the KM3 metametamodel and can also load M1, M2 and M3 in memory. Figure 6 presents the DSFM.

Figure 6 DSFM conceptual structure

We have prefixed our classes by DS to distinguish these from other classes. The μ symbol is replaced by the symbol "mu" in the drawing.

The most important entity is *DSElement* which is encapsulated in *DSModel*. Refer to KM3 [1] for more details.

At the implementation level, we added some properties and methods to help navigating the DSFM. Figure 7 focuses on the two major classes implemented in Visual Basic. This figure is generated by Visual Studio from our source code.

Figure 7 DSFM implementation details

We added methods in DSModel to navigate in the model, like *getModelElementFromName* which gives an element of a metamodel by giving his name (metamodel elements are named and names are distinct). Others, like *DSElement.MuName*, help to the construction by permitting notably logical μ link instead of physical association. It is practical for M3 construction because the entity can be its own metaclass (cycle).

We try to develop an explicit and elegant syntax by defining methods like *DSModel.newInstance DSElement.setValue* or *DSElement.updateReference* that properly instantiate and update model elements.

6. Global mechanism

Our aim is to extract a model (conforming to an arbitrary metamodel) from the exploration of a system. Figure 8 illustrates this idea.

Figure 8 Model Extractor global mechanism

We now present the behavior of the extractor. We divide the solution in three components:

- *MetamodelsLoader*: loads decorated metamodel from Ecore files
- InspectionEngine: browses the metamodel, extracts information from system and builds out model
- *ModelsExporter*: exports DSFM to Ecore files.

The Figure 9 presents this cut-out.

Figure 9 ModelExtractor components cut-out

All of these components use the DSFM to represent and exchange models whatsoever the level. The heart of the solution is the *InspectionEngine*. It runs in several steps presented in Figure 10.

Figure 10 Model Extraction mechanism steps

The *MetamodelsLoader* starts by uploading a KM3 metametamodel. Unlike MDR, we do not need to bootstrap this self reflective model due to the simplicity of this language (DSL). Then we can load a metamodel from Ecore files with correct µ initialization.

7. First results

Our prototype is a first attempt to implement a general automatic model extraction solution for the DotNet platform, and more particularly from Visual Basic 9.0 code. We have demonstrated the feasibility of the decoration process by producing an assembly to Visual Basic extractor. The generalization to a parametric metamodel is explained in the next section. In summary, we make an injector to our technical space by using decoration engine principles.

First we define a simplified Visual Basic metamodel and we establish the mapping for information extraction from assembly. Decorations

actually copy this mapping. We just consider static view for the time being in optic to test our DSFM and become familiar with the tools we need like *CodeProvider* (compiler). Figure 11 illustrates the actual implemented solution.
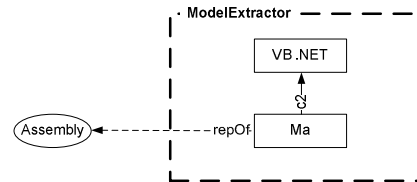
Figure 11 Actual implemented solution

We realized a complete simplified chain by loading a decorated metamodel from Ecore files (Figure 12), producing an executable extractor by exploring the metamodel, compiling the resulted code, running it on a static assembly and, finally, generating a model which represents the system and is conforms to the metamodel (Figure 13).
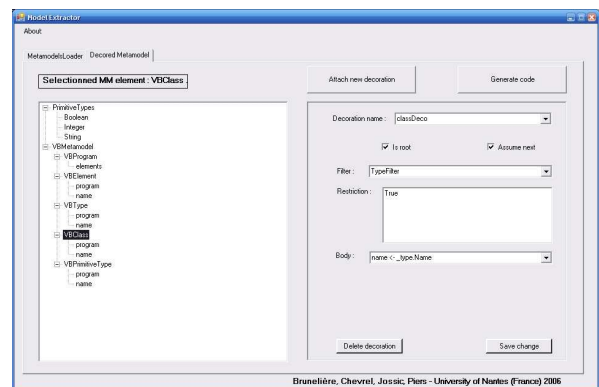
Figure 12 Prototype loads metamodel from ecore files

This metamodel is defined in KM3 and is exported to Ecore format by the existing tools. The DSFM is capable to store the metamodel and we can easily navigate it.
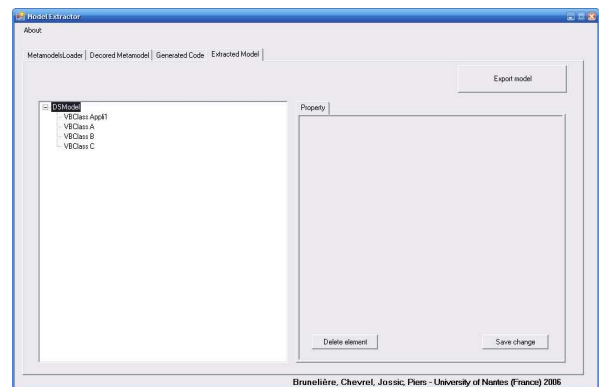

Figure 13 Prototype extract model from decorations compilation

The target model is generated from code, resulting from decoration discovery and following the organization given by the metamodel. For an element, the first word represents the metaclass name by navigating the μ links and the second is the first string attribute we find, if it exists (assuming that it is the name). It illustrates that the conformance is assured by the prototype and that the attributes are valuated. We now just have to export the *DSModel* to an Ecore file with the *ModelsExporter*.

8. Extensions and Future Work

The prototype has been tested on a given metamodel and for static systems. We suggest some realistic solutions to extend it to a parametric metamodel guidance extraction and for dynamic system execution snapshot. Furthermore, we want to formalize the conformance test for the out model by implementing a conformance checker component.

8.1. Parametric metamodels

The first improvement we want to do is to extend the solution to many metamodels. We use our actual prototype like an injector to our DSFM space and we transform the resulted model to a wanted conformance model using transformation language like ATL. Either by develop a decoration DSL inspired by ATL or by implement an engine for ATL in DotNet, then the decorations become weaving description. We privilege the second solution to integrate this realization to the actual AMMA platform. Figure 14 illustrates the goal.
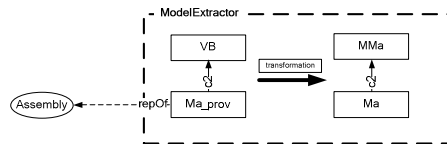

Figure 14 Extension to a parametric solution

8.2. Dynamic systems

We want to capture dynamic systems models by tracing the activity of the processes, like trace out the constructor methods invocation for example. There are two possible approaches: with intercession mechanisms, we can modify marshaled system comportment by adding a listening protocol, but it is not always admissible, or by extending the virtual machine to capture system events before transmitting these to the CLR, like the Visual Studio debug environment. The second proposition is better because we do not have to touch to the system, which is more correct, but very harder to achieve. Then the target model could be continuously alimented by the virtual machine extension. So it is possible that we need a representation to define infinite model structure and construction.

8.3. Conformance Checker component

For the moment, we strongly believe in the correctness of the extraction of our ModelExtractor, but we think that is will be useful to have a specific component to check the precision of the conformance construction. We want to implement a *ConformanceChecker* which will control the correct conformance of the model step by step or at the end of the execution.

9. Conclusion

We have obtained very encouraging results with our first prototype. The feasibility of the global approach has been proved and we already have ideas to extend our solution. We will concentrate our effort on two ways: principally we aim at implementing an ATL engine for Visual Basic and also trying to develop a capture environment to establish dynamic snapshots. Many problems will then rise, like distortion with time stamping or the risk to influence systems comportments by studying these in real-time.

10. Ackowledgements

11. References

[1] ATL, ATLAS Transformation Language Reference site http://www.sciences.univ-nantes.fr/lina/atl/ including KM3: Kernel Metametamodel definition.
[2] AM3 Homepage http://www.eclipse.org/gmt/am3/
[3] AMW Homepage http://www.eclipse.org/gmt/amw/
[4] Microsoft DotNet homepage http://www.microsoft.com/net
[5] Eclipse main page www.eclipse.org/
[6] Modelisoft http://www.modelisoft.com/Download.aspx?Modelisoft GenieLogiciel.pdf
[7] Basic 9.0 Meijer, E., Silver, A., Vick, P. Overview of Visual Basic 9.0, Microsoft Corporation,