Gra2MoL put into practice

Javier Luis Cánovas Izquierdo (jlcanovas@um.es), Jesús García Molina (jmolina@um.es)

Universidad de Murcia, Spain

Tool description

Introduction

The application of Model-Driven Development (MDD) techniques in the area of software modernization is an emerging discipline where methods, tools and techniques have to be still devised. The activities involved in a software modernization process can be described in the context of the horseshoe model [1]. Figure 1 shows the horseshoe model adapted to the application of MDD and the three main processes involved. A reverse engineering process is first applied to obtain models that are high-level abstractions of the information of the source software system artefacts. The restructuring process is then in charge of converting the obtained models into other models which represent the redesigned system conforming to the target architecture. Finally, the forward engineering process generates the target software system artefacts.

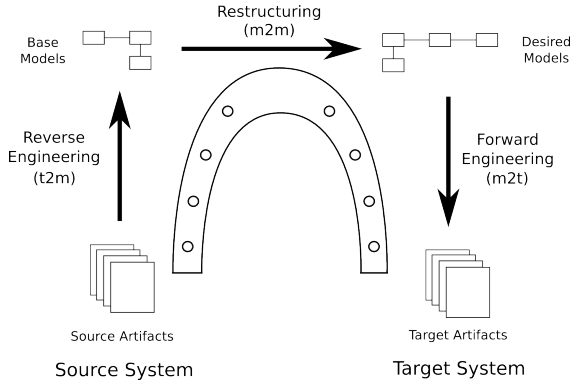

Figure 1. The horseshoe model adapted to MDD

The activities involved in the restructuring and forward engineering processes are usually performed by applying model-to-model (m2m) and model-to-text (m2t) transformations, respectively. However, the activities involved in a reverse engineering process, which could be denominated as text-to-model (t2m) transformations, are normally carried out by ad-hoc solutions (i.e. parsers).

Extracting models from the source code is one of the most important reverse engineering activities. Since source code files conforms to the grammar of a general programming language (GPL), *ad-hoc* parsers are usually developed to obtaining such models. However, developing an *ad-hoc* parser is a time-consuming and very expensive task. Furthermore, although other approaches such as xText or Stratego can be applied, they have a number of drawbacks that restrict their application in reserve engineering (in [2] a deep analysis about several existent alternatives is presented). With these problems in mind, we created the Gra2MoL language [2], a domain-specific language (DSL) for extracting models from source code.

In the past two years, we have successfully applied Gra2MoL to obtain models from source code of several programming languages and we have used it as part of a process for extracting ADM models [3]. Whereas the Gra2MoL language was presented in EC-MDA'2009 as a paper in the Foundations track [2], now we show the Gra2MoL development environment along with several practical experiences. This document is organized as follows. Firstly, we present the Gra2MoL features, secondly we show a transformation example, thirdly we describe the tool, and finally we indicate some examples of application.

Gra2MoL

In Gra2MoL, a model extraction process is considered as a grammar-to-model transformation, so mappings between grammar elements and metamodel elements are explicitly specified. As is

shown in Figure 2, a Gra2MoL transformation has four inputs: source code (P_G) along with the grammar definition (G) it conforms to, a target metamodel (MM_T) and a transformation definition ($Mapping_{Gra2MoL}$). The output is a model (M_T) which conforms to the target metamodel.

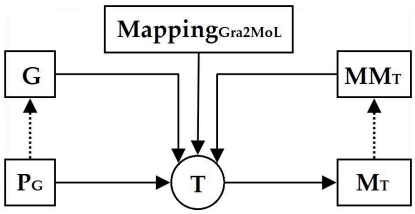

Figure 2. Process of extracting model from source code by using Gra2MoL.

The language has been designed as a rule-based transformation language with rules whose structure is similar to those provided in m2m transformation languages such as RubyTL [4] or ATL [5], with two important differences: i) the source element of a rule is a grammar element rather than a metamodel element, and ii) the navigation is expressed by a specially tailored query language rather than an OCL-based language.

A Gra2MoL transformation definition consists of a set of transformation rules. Each rule specifies the mappings between a grammar element and a target metamodel element and is composed of four parts:

- The *from* part specifies a grammar non-terminal symbol, and declares a variable that will be bound to a tree node when the rule is applied. This variable can be used by any expression within the rule. The *from* part can also include query operations to check the structure to be satisfied by the nodes whose type is the non-terminal symbol.
- The *to* part specifies the target element metaclass.
- The *queries* part contains a set of query expressions which allow information to be retrieved from the CST. The result of these queries will be used in the assignments of the *mappings* part.
- Finally, the *mappings* part contains a set of bindings to assign a value to the properties of the target element.

The execution of a Gra2MoL transformation definition is driven by the bindings of the mappings section. Gra2MoL bindings have very similar syntax and semantics to those used in RubyTL and ATL languages but, in this case, the right-hand side can be the variable specified in the *from* part of the rule, a literal value or a query identifier. On the other hand, model extraction from source code requires an intensive use of queries for retrieving scattered information which is necessary to build the target metamodel elements. Thus, Gra2MoL incorporates a *structure-shy* query language inspired in XPath which allows us to traverse syntax trees efficiently in order to obtain such information. A more detailed explanation about the query language and rule evaluation can be found in [2].

Gra2MoL language has incorporated new features to improve its efficiency, extensibility and adaptability to the model extraction problem. Some of these features are the following:

- *Skip* rules for dealing with expressions. Grammar rules for describing expression languages have a particular structure and this new type of rule helps to define transformation rules for extracting models which represent such expressions.
- Support for island grammars. The island grammar mechanism is used when a main language contains one or more sublanguages (e.g. the Javadoc language in Java). Thus, island grammar allows defining complex languages which are composed of languages which conforms to different grammars.
- *Mixin* rules for factorizing the queries and mappings of the rules. In large transformation definitions, there are normally several rules which share some queries or mappings. In this case, *mixin* rules allow us to factorize such transformation code for decreasing the scattering.
- Extensibility mechanism which allows incorporating new operation to both the query language and *mappings* section.
- CDO support for storing large models in databases.

Example

The following example has been used in a migration from Struts to JSF platforms. We used the KDM metamodel for representing the Java source code in order to promote the interoperability. Since the lack of space, the Figure 3 only shows a fragment of the transformation definition (the transformation definition can be downloaded from Gra2MoL website [6]).

```
rule 'mapCodeModel'
  from compilationUnit cu
  to code::CodeModel
  queries
    clases : /cu/#normalClassDecl;
  mappings
    name = "codeModel";
    codeElement = classes;
end_rule

rule 'mapClassUnit'
  from normalClassDecl nc
  to code::ClassUnit
  queries
    elements : /nc/#classBodyDecl;
  mappings
    name = nc.Id;
    codeElement = elements;
end_rule
```

```
rule 'mapField'
  from classBodyDecl//fieldDecl
  to code::MemberUnit
  queries
    ...
  mappings
    ...
end_rule

rule 'mapMethod'
  from classBodyDecl//memDecl{Id.exists}
  to code::MethodUnit
  queries
    ...
  mappings
    ...
end_rule
```

Figure 3. Gra2MoL transformation definition example

The first rule, named `mapCodeModel`, starts the transformation process and creates an instance of `CodeModel` metaclass. Whereas the first binding of this rule assigns a value to the `name` attribute, the second binding calls to the `mapClassUnit` rule, since *from* and *to* parts of such a rule conforms to this binding. The `mapClassUnit` rule also has a binding which initializes the `name` attribute and the second binding calls to the `mapField` and `mapMethod` rules depending on the result nodes of the query `elements`, which have to satisfy the *from* filter of such rules.

Gra2MoL tool overview

Although Gra2MoL has been designed as a tool to be used stand-alone, we are working on integrating it in the AGE platform at this moment. Thus, AGE incorporates a specific editor for Gra2MoL transformations which provides syntax highlighting, outline view and auto-completion for queries, making easier the navigation through the grammar. Figure 4 shows a capture of the editor and the outline view.

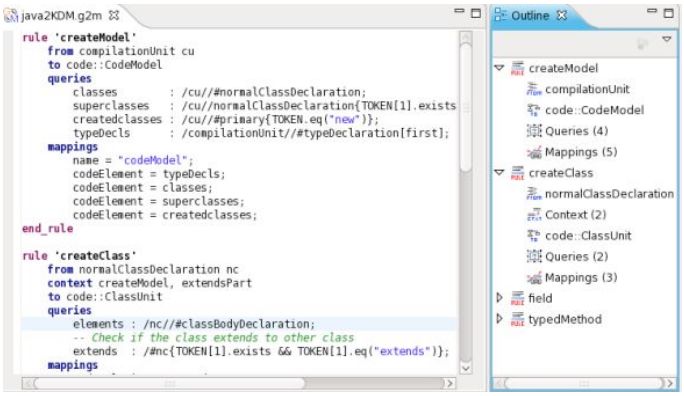

Figure 4. Gra2MoL editor in Eclipse

Gra2MoL transformations can be launched by its Java API but it also provides a pair of Ant tasks which perform the main tasks of the transformation process. One of them is in charge of enriching automatically the source code grammar in order to include computations which will create a syntax tree from the source code. Such enriched grammar is used to generate an ANTLR parser. The other Ant task executes a Gra2MoL transformation definition and has four inputs: (1) the previously generated ANTLR parser from the enriched grammar, (2) the source code, (3) the transformation definition and (4) the target metamodel. The result of this Ant task is a model conforming to the target metamodel. Thus, the whole transformation process can be specified in a

single Ant build file by using these tasks (a skeleton of such a file can be downloaded from the Gra2MoL website [5]).

Using Gra2MoL

Gra2MoL is an academic tool whose first version was available in 2008 [7]. Since then, we have applied it in several real projects, some of them are the following (they are available in the Gra2MoL website [5]):

- Extracting models from Java code in a migration from Struts to JSF applications.
- Extracting models from PL/SQL code in a migration from Oracle Forms to Java platform.
- Extracting models from Maude code.
- Extracting models from *bash*-script configuration files.
- Extracting models from SQL DDL and SQL DML files in an improvement of Wiki systems.

By using Gra2MoL in these projects, we have been able to test the language in different contexts and to check its suitability. Thus, we have improved some quality aspects: the robustness by fixing some errors; the efficiency by optimizing some algorithms and using CDO, and the usability by adding new features which make easy the specification of the transformations (e.g. *mixin* and *skip* rules).

On the other hand, we have applied Gra2MoL for extracting models as part of a modernization process for ADM [8]. In particular, we have obtained ASTM [9] and KDM [10] models from Oracle Forms applications and then checked some metrics [3].

Participation in the tools presentation or only to the exhibition

Gra2MoL could be presented in the tools presentation track with the objective of showing the capabilities of the language and how we are tackled the model extraction in some of the previously presented case studies.

References

[1] R. C. Seacord, Daniel Plakosh and Grace A. Lewis, "*Modernizing Legacy Systems*", Addison-Wesley, 2003.

[2] J. Cánovas and J. García Molina. "*A Domain Specific Language for Extracting Models in Software Modernization*", 5th EC-MDA, LNCS 5562, pp. 82-97, 2009 (downloadable from `http://adm.omg.org/adm_info.htm#white papers`).

[3] J. Cánovas and J. G. Molina, "*Building an ADM-based tool*", IEEE Software – Special Issue in Software Evolution, Jul/Aug 2010. *To be published*.

[4] J. S. Cuadrado, J. G. Molina and M. M. Tortosa, "*Rubytl: A practical, extensible transformation language*" in ECMDA-FA, L. N. in Computer Science, vol. 4066/2006, pp. 158, 172 (2006).

[5] F. Jouault and I. Kurtev, "*Transforming models with atl*" (2005).

[6] Gra2MoL website. `http://modelum.es/gra2mol`.

[7] J. Cánovas, J. S. Cuadrado and J. G. Molina, "*Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization*" in II Workshop Model Driven Software Evolution (MoDSE) (2008).

[8] ADM website. http://adm.omg.org.

[9] ASTM metamodel specification. http://www.omg.org/spec/ASTM/1.0/Beta1

[10] KDM metamodel specification. http://www.omg.org/spec/KDM/1.1/