A domain specific language for extracting models in software modernization

Javier Luis Cánovas Izquierdo and Jesús García Molina

University of Murcia
{jlcanovas,jmolina}@um.es

Abstract. Model-driven engineering techniques can be used both to create new software and to modernize existing software systems. Model-driven software modernization requires a first step for the extraction of models. Most modernization scenarios involve dealing with the GPL source code of the existing system. Techniques and tools providing efficient means to extract models from source code are therefore needed.
In this paper, we analyze the difficulties encountered when using the existing approaches and we propose a language, called Gra2MoL, which is especially tailored to address the problem of model extraction. This provides a powerful query language for concrete syntax trees, and mappings between source grammar elements and target metamodel elements are expressed by rules similar to those found in model transformation languages. Moreover, the approach also allows reusing existing grammars.

1 Introduction

Model-driven engineering (MDE) techniques are not only used to create new systems, but also to evolve or modernize legacy software. The field of model-based software modernization is currently emerging and a great research and development effort will thus be necessary in the years to come. The OMG has recently proposed several modernization standards in its ADM initiative [1], certain tools, such as MoDisco [2], are currently under development, and some research challenges have even been identified for the evolution of systems built by using MDE techniques [3].

Most modernization scenarios [4], such as platform migration or application improvement, involve dealing with source code written in a general purpose programming language (GPL). Techniques and tools providing efficient means to extract models from source code are therefore essential. In this extraction process, models conforming to a target metamodel are generated from source code conforming to the grammar of a GPL. Thus, a bridge between the technical spaces grammarware [5] and MDE must be built. This bridge is normally implemented by dedicated parsers (i.e. tools performing both parsing and model generation tasks), since the use of approaches such as bridging *grammarware* and MDE or transforming programs have certain drawbacks which limit their usefulness. Bridging approaches [6, 7] aim to create textual domain specific languages

(DSL) which have a simpler structure than GPLs; in the case of program transformations [8, 9], the resulting program must still be converted into a model. Since the construction of dedicated parsers is a time-consuming task, we have defined a DSL, denominated as Gra2MoL, which has been specifically designed for extracting models from GPL source code.

Gra2MoL allows mappings to be established between grammar elements and target metamodel elements in a similar way to which model-to-model transformations are expressed in languages as ATL [10] and RubyTL [11]. It provides a powerful query language to ease the navigation and query of syntax trees. We have used Gra2MoL to extract models from Java and PL/SQL code, and this experience has shown a reduction in development time, while maintenance is also improved and existing grammars are reused. In this paper we present Gra2MoL, and compare it with the existing approaches.

This paper is organized as follows. Section 2 analyzes the difficulties encountered when using existing solutions for model extraction, and Gra2MoL is motivated. In Section 3, we describe the language for querying concrete syntax trees provided by Gra2MoL. Section 4 presents the main features of Gra2MoL and explains how it has been implemented, while Section 5 shows an example of the language. Finally, Section 6 presents our conclusions and some future work.

2 Model extraction from source code

In this section we contrast different approaches which could be used to extract models from GPL code, indicating their main limitations. This discussion will motivate the approach proposed in this paper. We will begin by giving a definition of *model extraction* in the context of model-driven modernization, identifying the main issues to be addressed.

Figure 1 shows the elements involved in the process of extracting models from GPL source code. This process is a grammar-to-model transformation T which has as its input a program P along with the grammar definition G to which it conforms. It generates a target model M_T conforming to a target metamodel MM_T which defines the information to be extracted. The extraction process also requires specifying mappings between the grammar elements and the metamodel elements. As we will see, the form of these mappings is different depending on each considered approach. The input program is represented by either an abstract syntax tree (AST) or a concrete syntax tree (CST). Along this paper, we will use the term "syntax tree" to refer both AST and CST.

Therefore extracting models from source code is a scenario which requires a bridge between *grammarware* and MDE techniques (*modelware*), the same as the definition of a textual concrete syntax for an abstract syntax metamodel [6, 12]. With GPL code, creating this bridge requires an efficient mechanism for traversing syntax trees since the model elements to be extracted are usually composed of information that is scattered in such trees. In particular, this scattering is mainly caused by the means used to represent the references between elements. Models are graphs and any model element can directly refer to another, whereas

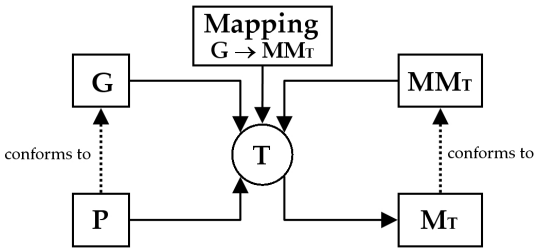

Fig. 1. Process of extracting models from source code.

in a syntax tree that represents certain code which conforms to a GPL grammar, the references between grammar elements are implicitly established by means of identifiers. Transforming an identifier-based reference into an explicit reference involves looking for the "identified" node on the syntax tree. For instance, if a model element is extracted from a "function call" statement where one argument is a global variable, certain necessary information, such as the type of the variable or the function signature, is located outside the current scope ([13] calls this kind of transformations global-to-local transformations).

2.1 Approaches for model extraction

The chosen strategy is normally that of creating dedicated parsers. Given a grammar and a target metamodel, a dedicated parser provides a specific solution which performs both parsing and model generation tasks. The former is in charge of extracting a syntax tree from the source code and the latter traverses such syntax trees in order to generate the target model. For example, in [14] a dedicated parser is built to extract models from PL/SQL code, and another with which to extract models from VB code is presented in [15]. However, dedicated parser development is a time-consuming and very expensive task. The effort required is usually alleviated by automatically extracting an AST from the source code. This step is performed by using an API, which is intended to make the management of such tree easier. An example of such APIs is the JDT Eclipse project [16], which works with Java source code. But APIs do not currently exist for a number of the GPLs widely used in modernization (e.g. PL/SQL language). In addition, although these APIs tackle AST extraction and management, a mechanism for retrieving scattered information must still be hard-coded, so APIs do not considerably shorten the development time.

MoDisco (Model Discovery) [2] is a model extraction framework, which is part of the Eclipse GMT project [17]. This framework is currently under development and provides a model managing infrastructure through which to implement dedicated parsers ("discoverers" in MoDisco terminology). A KDM based metamodel, a metamodel extension mechanism and a methodology for designing such extensions are also planned.

Several approaches for bridging *grammarware* and *modelware* have been defined in the context of creating textual domain specific languages (DSL) for MDE. These approaches can be classified in two groups according to whether they are focused on grammars or metamodels. Grammar-based approaches are oriented towards automatically generating metamodels from grammars, whereas metamodel-based approaches work in the opposite direction. In model-driven modernization, the process starts from existing source code which conforms to the grammar of a GPL. Therefore metamodel-based approaches, such as TCS, are not well suited. As is stated in [12]: "*If the problem at hand is to develop a single, eventually general purpose language then the efforts for developing a dedicated parser are worthwhile*" (rather than using TCS).

XText [6] is an example of a grammar-based approach, which is part of the openArchitectureWare toolkit [18]. An EBNF-based language is provided to specify the input grammar to the xText processor, that is, a specification of the textual concrete syntax including grammar rules intended to guide the generation of the corresponding metamodel. Three artifacts are thus generated: i) a metamodel of the language, ii) a parser to recognize the language syntax and to create models conforming to the generated metamodel, and iii) a language specific editor.

However, several problems arise when xText is used in modernization, since dealing with GPL source code involves problems not addressed by this approach, which is aimed at simpler languages than a GPL. The automatically generated metamodel from the grammar of a GPL is of poor quality because it includes superfluous elements and grammatical aspects, and the semantic gap between this metamodel and the target metamodel is thus very high. A model-to-model transformation is therefore required to convert models generated by xText into models conforming to the desired target metamodel. However, since current model-to-model transformation languages do not offer an efficient mechanism to resolve the problem of gathering scattered information, the definition of such transformation is a complex task. Moreover, neither existing grammar reuse, the reuse of grammars for well-known parser generators (e.g. ANTLR), nor the reuse of xText grammar specifications is promoted. On the one hand, translating a grammar specification provided by a parser generator into a xText EBNF-based specification is extremely complicated since some parser options which are needed to recognize GPLs cannot be specified (e.g. in Java, the use of backtracking or the inclusion of syntactic predicates). On the other hand, xText grammar specifications are oriented to a specific metamodel so they include specific rules for such a metamodel.

Wimmer et al. [7] and Kunert [19] have proposed improving the quality of the generated metamodel by applying heuristics and including manual annotations to the grammar. However, the quality of the metamodel generated from a GPL grammar is still low and it is necessary to additionally define a model-to-model transformation. Moreover, tools supporting these two approaches are not yet available.

Program transformation languages, such as Stratego/XT [8] and TXL [9], could be used to extract models from source code by expressing the abstract syntax as a context-free grammar rather than a metamodel. However, when such languages are used, the following limitations are encountered. Firstly, the result of a program transformation execution is a program conforming to a grammar, and therefore a tool for bridging *grammarware* and *modelware* would be still needed to obtain the model conforming to the target metamodel. Secondly, grammar reuse is not promoted because each toolkit uses its own grammar definition language. Moreover, each toolkit only provides a limited number of GPL grammars (i.e. Java and C in Stratego and TXL).

2.2 Our approach for model extraction

In the context of an Oracle Forms migration project, we faced model extraction from PL/SQL code. Then we considered the definition of a DSL in order to overcome the limitations of the previously discussed approaches. This DSL should decrease the development time, make the maintenance easier and promote the reuse of existing grammars (e.g. ANTLR and JavaCC grammars). To achieve these objectives, it is necessary to raise two key design issues: how can mappings between grammar elements and metamodel elements be expressed in a simple and readable way, and what notation is appropriate when retrieving scattered information from syntax trees.

Model-to-model transformation languages could be used to express the mappings between grammar elements and metamodel elements, but this possibility would require models rather than GPL code as input, and a dedicated parser would therefore have to be implemented to extract a model conforming to an intermediate metamodel (i.e. an abstract syntax tree metamodel) from source code. A model-to-model transformation would then be applicable. However, defining these transformations would lead to an important problem: the inadequacy of the query language. Many current model transformation languages, such as ATL [10] or QVT [20], provide a variant of the OCL navigation language [21] which allows model graphs to be traversed. Although OCL-like expressions are appropriate for most practical model-to-model transformation definitions, they are not convenient for typical global-to-local transformations involved in a model extraction from GPL code: long navigation chains must be written using dot notation. Integrating a more suitable query language in an existing model transformation language would involve important changes if a language supporting two different query mechanisms were to be obtained. For instance, a plugin mechanism could be implemented.

We have therefore created a DSL which has been specially designed to express grammar-to-model mappings, and which provides a powerful query language for syntax trees. In particular, since the scattered information problem appears in both AST and CST, and obtaining a CST is easier than an AST, we use CSTs for representing the source code. This DSL, denominated as Gra2MoL (Grammar To Model Transformation Language) will be described in Section 4 and the query language is introduced in the following section.

Approach	Syntax tree navigation	Artifacts to be created	Post processing	Existing grammar reuse	Built grammar reuse	Purpose
Dedicated parser (+ API)	GPL code (+ primitives)	MM_T P	None	Yes	NA	Specific model extraction
Grammar based bridging (xText)	Poor support	G_{xt} MM_T $m2m$	M2M transformation: generated $MM_I \rightarrow MM_T$	No	No	DSL creation
Program transf.	Stratego incorporates a query language [22]	MM_T T_{PT} G_{AS} $m2m$	Extracting a model from a program conforming to G_{AS}	Limited (a few grammars)	Yes	Program transf.
Gra2MoL	*Structure-shy* query language	MM_T T	None	Yes	NA	General purpose model extraction

Table 1. Comparison of Gra2MoL with the analyzed approaches. NA = Not applicable, G = Grammar, MM_T = Target metamodel, MM_I = Intermediate metamodel, T = Transformation definition, P = Dedicated parser, T_{PT} = Program transformation definition, G_{xt} = xText grammar, $m2m$ = model-to-model transformation definition, G_{AS} = Abstract syntax grammar.

Table 1 contrasts Gra2MoL with the analyzed approaches. The columns show the properties which are compared: the ability to navigate the syntax tree; which artifacts must be created; whether post-processing is necessary; whether it is possible to reuse existing and provided grammars; and the main purpose of the approach. The artifacts to be created and the post-processing tasks determine the effort level of each approach. For instance, we note that bridging and program transformation approaches require more complex tasks than Gra2MoL, such as writing model-to-model transformations or defining a GPL grammar, whereas in Gra2MoL it is only necessary to create the transformation definition and the target metamodel. With regard to the creation of a dedicated parser, Gra2MoL turns a hard-coding task into the writing of a grammar-to-model transformation definition using a language specially tailored for extracting models. As a consequence, development time is reduced by using Gra2MoL.

3 A query language for concrete syntax trees

As stated above, grammar-to-model transformations involve an intensive use of queries to collect scattered information. Therefore, a model extraction approach must provide a powerful query language, which facilitates the access to tree nodes outside the current construct scope (i.e. a rule). Figure 2 illustrates the scattering problem for a simple example of extracting a model element from a PL/SQL procedure. The CST shown corresponds with a procedure declaration which includes a variable declaration and an assignment statement initializing the declared variable. The `AssignmentStatement` model element represents a PL/SQL assignment which has two attributes to register the right-hand side and left-hand side expressions of the assignment. As can be observed, whereas all the information needed to initialize the right-hand side attribute (`insert`

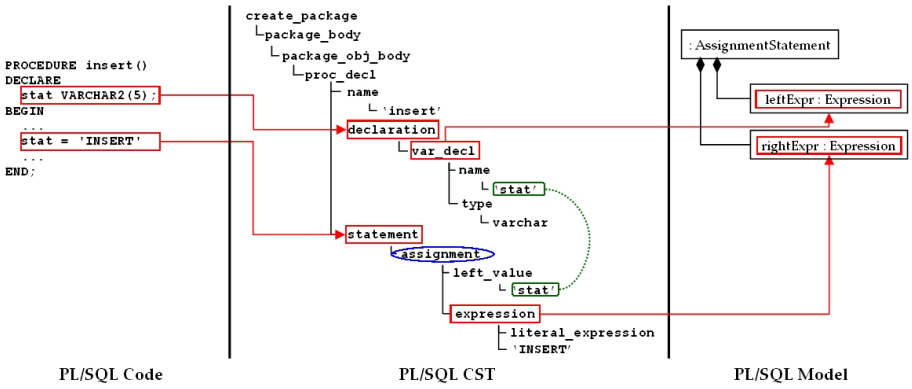

Fig. 2. Example of scattered information. The oval indicates the current scope and the dotted line indicates an identifier-based reference between tree elements.

literal) is inside the current scope (depicted as an oval), the information needed to initialize the left-hand side attribute is outside such a scope, because the `stat` variable declaration is referenced by an identifier. Therefore, a query is necessary to resolve this reference to the `stat` variable, by accessing the corresponding `declaration` node and retrieving the variables properties.

We have developed a *structure-shy* query language, inspired by XPath [23], which allows a CST of the source code to be navigated without the need to specify each navigation step. The term "structure-shy" is often used to refer to behavior specifications which are loosely bounded to the data structures on which operations (i.e. queries) are applied. The term "structure-shy" query is used in this sense.

In order to navigate the CST, the nodes are "typed" using the grammar definition, and each tree node registers the name of the grammar element as its type. Figure 3 illustrates the conformance relationships between the CST and the grammar definition, showing a CST for several PL/SQL procedures along with the corresponding fragment of PL/SQL grammar. The conformance rules are those commonly used to create a tree of this kind:

- A non-terminal element corresponds to a tree node. For instance, the `proc_decl` non-terminal element corresponds to the `proc_decl` tree node in Figure 3.
- A terminal element corresponds to a leaf. In Figure 3, the `Name` terminal corresponds to the `Name` leaf.
- A production rule is represented by a node hierarchy whose parent corresponds to the non-terminal element on the left-hand side of the rule, and a child for each grammar element on the right-hand side by applying the previous rules. In Figure 3, the `proc_decl` production rule is represented by the hierarchy whose root is a `proc_decl` tree node.

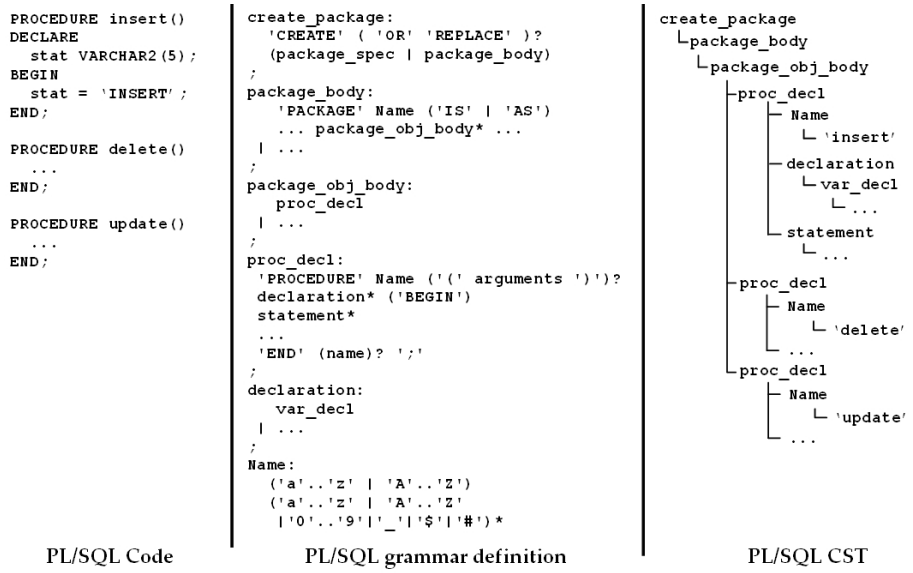

Fig. 3. CST for the PL/SQL grammar.

A query consists of a sequence of query operations in which each operation includes an operator, a node type and optional filter and access expressions. The EBNF expression for a query operation is:

```
{ ('/'|'//'|'///') ('#')? nodeType [filterExpression] [accessExpression] }
```

We have defined three operators for querying and navigating over CSTs: `/`, `//` and `///`. The `/` operator returns the immediate children of a node and is similar to dot-notation (e.g. in OCL). The `//` and `///` operators permit the traversal of all the nodes children (direct and indirect), thus retrieving all nodes of a given type. The `///` operator differs slightly from the `//` operator. Whereas the `///` operator searches the syntax tree in a recursive manner, the `//` operator only matches the nodes whose depth is less than or equal to the depth of the first matched node. The `///` operator is, therefore, only used to extract information from recursive grammar structures. These two operators allow us to ignore intermediate superfluous nodes, thus making the query definition easier, since it specifies what kind of node must be matched, but not how to reach it, in a structure-shy manner. The `createFunctionCallStatement` rule defined in Section 5 will show the difference between both operators.

Since a query could return one or more subtrees, the `#` operator is used to indicate the root node from which the information needed can be accessed. This operator must be associated to one and only one query operation of the sequence of operations forming a query. For instance, in order to extract all the PL/SQL variable declarations defined in every procedure of the PL/SQL CST shown in Figure 3, the following query could be expressed `/create_package//#var_decl`.

```
Locals(cp:create_package) : Sequence(variable_declaration)
post result=if cp.spec.oclIsKind0f(package_body) then
  p.spec.package_obj_body->
    select(pob | pob.oclIsKind0f(procedure_decl))->
    collect(fb | fb.declaration)->flatten()->
    select(ds | ds.oclIsKind0f(variable_declaration))
  else
    Sequence {}
  endif
```

Fig. 4. OCL query for extracting all the variable declarations of every prodecure of the PL/SQL CST shown in Figure 3.

The same query expressed in OCL is shown in Figure 4. It is worth mentioning how the clarity, legibility and conciseness are improved.

Query operations can also include a filter expression, which is enclosed in curly brackets. A filter expression is a logical expression which is applied to the leaves of the node specified in a query operation. Each operand of a filter expression is a boolean function which checks the properties of a leaf, such as its value or whether it exists. Only those nodes that satisfy the filter expression will be selected. For example, the query `/create_package//#proc_decl{Name.exists && Name.eq('insert')}` will select every `procedure` grammar element with a `Name` leaf and the value of such leaf must be `insert` in the PL/SQL CST shown in Figure 3.

Finally, query operators can also include an access expression enclosed in square brackets, which is used to access to sibling nodes through indexing. For instance, the query `/create_package//#proc_decl[0]` will select the first `procedure` grammar element of the CST in Figure 3, which is the `insert` procedure.

The following section outlines the Gra2MoL domain specific language which integrates the described query language.

4 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. According to Figure 1, the input of a Gra2MoL transformation is source code along with the grammar definition it conforms to, a target metamodel and a transformation definition; the output is a model which conforms to the target metamodel.

The language has been designed as a rule-based model transformation language with rules whose structure is similar to those provided in languages such as ATL or RubyTL, 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 the query language described in Section 3, 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.

An example of Gra2MoL is shown and commented upon in Section 5.

4.1 Bindings and rule evaluation

A binding construct is used in the *mappings* part to establish the relationship between a source grammar element and a target metamodel element. This construct has very similar syntax and semantics to the binding construct of the RubyTL [11] and ATL [10] languages. A binding is written as an assignment using the operator "=". The left-hand side must be a property of the target element metaclass. The right-hand side can be the variable specified in the *from* part of the rule, a literal value or a query identifier.

The execution of a transformation definition is driven by the bindings. The definitions of rule conformance and well-formed transformation stated for RubyTL in [11] are applicable to Gra2MoL, with simple changes. A metaclass A_m conforms to a metaclass B_m if they are the same or A_m is subtype of B_m, whereas a node type A_n conforms to a node type B_n if they are the same. Every Gra2MoL transformation definition must have an entry point in order to start the transformation execution. The entry point is the first rule of the transformation definition and its mappings are in charge of starting the transformation execution. When a rule is applied on a node, the filter located in the from part is first checked and then, if the node satisfies the filter, an instance of the target metaclass is created, and the rules bindings are executed. In the execution of a binding, three situations may arise according to the nature of the right-hand side.

1. If it is a literal value, the value is directly assigned to the property of the left-hand side.
2. If it is a query identifier, the query is executed and a rule resolving this binding is looked up in the transformation definition, i.e. a rule whose types of the *from* and *to* parts conform to the types of the right-hand side and left-hand side of the binding, respectively. Whenever a conforming rule is found, it is applied by using the element of the right-hand side of the binding as the source grammar element.

3. If it is an expression, it is evaluated and two situations may arise, depending on whether the result is a node whose type corresponds to a terminal (a leaf) or a non-terminal symbol. If it is a leaf, the result is a primitive type and is directly assigned, otherwise, a rule to resolve the binding is looked up and executed, as was explained in the previous case.

4.2 Implementation

The first step in the execution of a Gra2MoL transformation is to build the CST of the source code. Current implementation of Gra2MoL uses ANTLR grammar definitions. These definitions can be enriched with actions in order to create the CST. However, we are interested in using ANTLR grammar definitions without attached actions for two reasons: (1) to alleviate the grammar developer from the burden of creating the CST programmatically and (2) to promote grammar reuse. We have therefore defined an enrichment process which automatically adds the actions needed to build the CST to the grammar rules.

Gra2MoL uses a metamodel internally to generically represent CSTs of the parsed source code. This metamodel is shown in Figure 5. There are three kinds of elements in a CST model, namely `Leaf`, `Node` and `Tree`. `Leaf` represents a tree node which corresponds to a recognized terminal symbol. `Node` represents a tree node which corresponds to a recognized non-terminal symbol and is composed of one or more children nodes, either of the `Leaf` or `Node` type. The `type` attribute identifies the grammar symbol whose recognition has yielded the tree node creation (this is needed to navigate through the CST, as was explained in Section 3). Finally, `Tree` represents the root node of the tree. The creation of models conforming to this metamodel is driven by the conformance rules explained in Section 3.

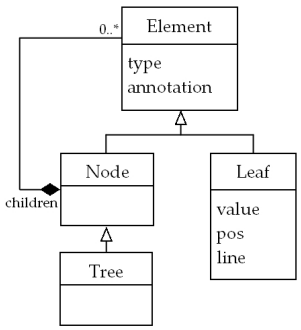

Fig. 5. CST metamodel (MM_{CST}).

The execution process of a Gra2MoL transformation is shown in Figure 6(b), together with a schema of the pre-processing step T to enrich the ANTLR grammar in Figure 6(a). Note that 6(b) is the same as Figure 1, except that a parser

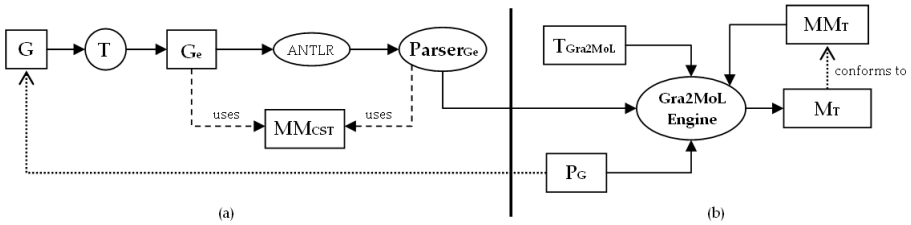

Fig. 6. Gra2MoL implementation

is an input to the Gra2MoL engine to build the CST model. This parser is generated from the grammar (G_e) enriched with actions intended to create CST models conforming to the metamodel MM_{CST} shown in Figure 5.

5 Example

Oracle Forms is an Oracle technology used to design and build enterprise applications in which the business logic and database access is encapsulated in PL/SQL triggers. The source code of such applications is organized in sequences of statements in which each sequence corresponds to a trigger. Gra2MoL has been used within the context of a project to migrate Oracle Forms applications to Java platform, in order to extract models from PL/SQL code. We implemented a Gra2MoL transformation definition to extract models conforming to a metamodel representing a subset of the PL/SQL abstract syntax. This transformation definition consists of 57 rules[1]. An excerpt of this transformation definition will be shown as follows. Figure 7 shows the parts of the PL/SQL grammar and the PL/SQL metamodel considered in this example.

```
rule createPLSQLDefinition
  from create_package cp
  to PLSQLDefinition
  queries
    seqt : /cp//#seq_of_statements;
  mappings
    triggers = seqt;
end_rule

rule createTriggerBlock
  from seq_of_statements seqt
  to TriggerBlock
  queries
    stats : /seqt/#statement;
  mappings
```

[1] The complete transformation definition can be downloaded from http://modelum.es/gra2mol

```
seq_of_statements
  :  statement SEMI ( statement SEMI )*
;
statement
  : return_statement
  | function_call
  | ...
;
function_call
  : user_defined_function LPAREN
    ( call_parameter )? RPAREN )?
;
user_defined_function
  : sql_identifier
;
call_parameter
  : (parameter_name '=>' )? nested_expression
    (call_parameter)?
;
parameter_name
  : identifier
;
return_statement
  : keyRETURN ( plsql_expression )?
;
sql_identifier
  : identifier
;
identifier
    :   ID ( DOT ID )*
;
ID /*options { testLiterals=true; }*/
  : 'A'..'Z'
    ('A'..'Z'|'0'..'9'|'_'|'$'|'#')*
  | DOUBLEQUOTED_STRING
;
```

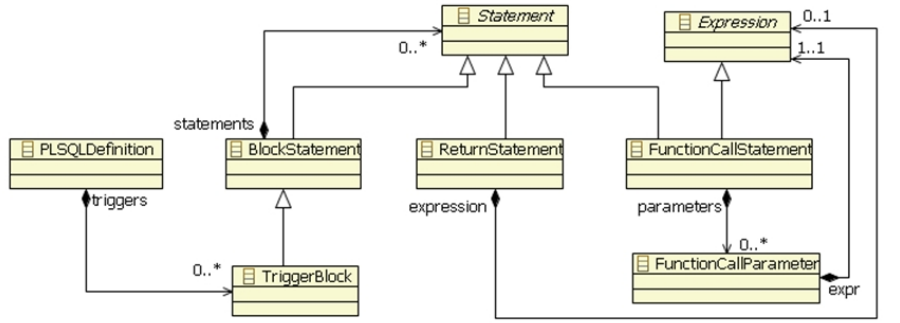

Fig. 7. Excerpt of the PL/SQL grammar and the subset of PL/SQL metamodel used in the example. The non-terminal symbols used in the example are underlined.

```
    statements = stats;
end_rule

rule createReturnStatement
  from statement/return_statement st
  to ReturnStatement
  mappings
end_rule

rule createFunctionCallStatement
  from statement/function_call st
  to FunctionCallStatement
  queries
    fc     : /statement/#function_call;
    iden   : /fc/user_defined_function//#identifier;
    params : /fc///#call_parameter;
  mappings
    name = iden.ID;
    parameters = params;
end_rule
```

```
rule createFunctionCallParamForFunctionCall
  from call_parameter cp
  to FunctionCallParameter
  queries
    iden  : /cp/parameter_name/#identifier;
  mappings
    name = iden.ID;
end_rule
```

The first rule starts the transformation process by creating an instance of PLSQL Definition. This rule has only one binding whose right-hand side is a query identifier and whose left-hand side refers to the triggers attribute of the `PLSQLDefinition` metaclass. The query is therefore executed and the rules conforming the binding are then looked up and executed. In this example, the `createTriggerBlock` rule would create instances of `TriggerBlock` and would apply the `statements = stats` binding, whose right-hand side is a query identifier and whose left-hand side refers to the statements attribute of the `TriggerBlock` metaclass. In this case, both `createReturnStatement` and `createFunctionCallStatement` rules conform to the binding, but the filter of these rules allows the selection of only one, depending on the direct children of the statement grammar element. The `createFunctionCallStatement` rule illustrates the meaning of the `//` and `///` operators. On the one hand, the `//` operator used in the second query avoids the need to specify every navigation step (i.e. `sql_identifier`) to reach the identifier node. On the other hand, since the `call_parameter` production rule is defined recursively, the `///` allows the CST to be traversed in order to retrieve every `call_parameter` node.

It is worth mentioning how clear and legible the transformation shown above is. Both the implicit rule application driven by the bindings and the format of the queries make it a clean language. Moreover, the query format also contributes towards improving the legibility of the CST retrievals. Figure 8 shows the result of an execution of this transformation definition.

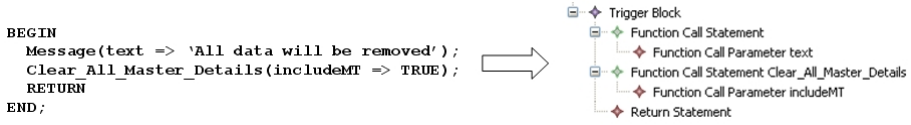

Fig. 8. Result of a Gra2MoL transformation execution

6 Conclusions and future work

In this paper, we have presented Gra2MoL, a DSL for extracting models from GPL source code by means of grammar-to-model transformations. Gra2MoL has

therefore been designed as a rule-based language inspired by languages such as ATL and RubyTL. A Gra2MoL transformation definition consists of rules which transform grammar elements into model elements by manipulating the CST of the source code. A powerful language has been defined to navigate and query a CST in a structure-shy manner. Several benefits will be derived from this new approach, in comparison to existing solutions.

With regard to the implementation of a dedicated parser, our approach considerably reduces the development time, and maintainability is also favored. Mappings and queries are not hard-coded in the code of a programming language, but are specified in a clear, concise and legible manner.

We have also compared Gra2MoL to *grammarware*-MDE bridging and program transformation approaches. Since neither of these approaches was devised for the extraction of models from GPL code, both require the performance of difficult tasks. On the one hand, bridging approaches were designed to create DSL, and are therefore not very practical for dealing with GPL. For instance, xText generates low-level models, and model-to-model transformations have to be defined. On the other hand, a program transformation approach could be used but the developer is required to write a target grammar specification and to define a bridge to convert the program generated into a model; Gra2MoL transformations are, moreover, simpler than those transformations expressed by the more commonly used transformational approaches, which use formalisms such as rewriting techniques.

With regard to future work, we are working on several issues such as a modularity mechanism for Gra2MoL transformations and the identification of query patterns to make the query definition independent from the grammar structure. We are also analyzing how to integrate Gra2MoL in the Modisco framework as a mechanism through which to create discoverers. Moreover, since Gra2MoL deals solely with ANTLR grammars, we would like to support other parser generators in order to increase the number of existing grammars that can be reused.

Acknowledgment

This work has been supported by Consejería de Educación y Cultura (CARM, Spain), grant TICARM-9478. Javier Luis Cánovas Izquierdo enjoys a doctoral grant from the Fundación Séneca.

References

1. Architecture-Driven Modernization Roadmap. OMG (2006).
2. MoDisco. http://www.eclipse.org/gmt/modisco/
3. A. van Deursen, E. Visser, and J. Warmer, "Model-driven software evolution: A research agenda" in Workshop on Model-Driven Software Evolution (2007).
4. ADM Task Force: Architecture-driven modernization scenarios. OMG (2006).
5. P. Klint, R. Lämmel and C. Verhoef, "Toward an engineering discipline for grammarware", in ACM Transactions on Software Engineering Methodology, n. 3, vol. 14, pp. 331-380 (2005).

6. S. Efftinge, "openarchitectureware 4.1 xtext language reference", http://www.eclipse.org/gmt/oaw/doc/4.1/r80 xtextReference.pdf (2006).
7. M. Wimmer and G. Kramler, "Bridging grammarware and modelware", Satellite Events at the MoDELS 2005 Conference, pp. 159168 (2006).
8. Stratego/XT. http://strategoxt.org/
9. TXL. http://www.txl.ca/
10. F. Jouault and I. Kurtev, "Transforming models with atl" (2005).
11. 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).
12. F. Jouault, J. Bézivin, and I. Kurtev, "TCS: a dsl for the specification of textual concrete syntaxes in model engineering", in GPCE, pp. 249-254 (2006).
13. J. van Wijngaarden and E. Visser, "Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems", Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2003-048, (2003).
14. L. F. Andrade, J. Gouveia, M. Antunes, M. El-Ramly and G. Koutsoukos, "Forms2Net - Migrating Oracle Forms to Microsoft .NET", GTTSE, pp. 261-277 (2006).
15. "Migrating Visual Basic Applications to VB.NET using the NewCode extension for Microsoft Visual Studio". Newcode (2008).
16. JDT Eclipse project. http://www.eclipse.org/jdt
17. GMT Eclipse project. http://www.eclipse.org/gmt
18. OpenArchitectureWare toolkit. http://www.openarchitectureware.org
19. A. Kunert, "Semi-automatic generation of metamodels and models from grammars and programs", in Fifth Intl. Workshop on Graph Transformation and Visual Modeling Techniques, E. N. in Theorical Computer Science, vol. 211, pp. 111-119, (2008).
20. Linda Heaton. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG (2005).
21. OCL constraint language. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf. OMG (2006).
22. J. van Wijngaarden. "Code Generation from a Domain Specific Language", MSc Thesis (2003).
23. Xpath. http://www.w3.org/TR/xpath