Gra2MoL: A domain specific transformation language for bridging *grammarware* to *modelware* in software modernization

Javier Luis Cánovas Izquierdo, Jesús Sánchez Cuadrado, Jesús García Molina

***Abstract*—Model-driven engineering (MDE) can be used both to create new software systems and to evolve or modernize existing software systems. In model-driven software evolution, models are extracted from the system and then MDE techniques are applied to make it evolve. Most evolution scenarios involve dealing with existing source code written in some programming languages. Therefore, a bridge from grammarware to MDE must be built to extract models from such a source code.**

In this paper we present an approach for the extraction of models conforming to a target metamodel from source code conforming to the grammar of a programming language. This approach is based on the definition of a grammar-to-model transformation language, named Gra2MoL, which is a language specially tailored to address the grammarware-modelware bridge in modernization scenarios. The language promotes grammar reuse, and provides domain-specific features such as a powerful query language to traverse syntax trees.

***Index Terms*—Software evolution, Model Driven Modernization**

I. Introduction

MODEL-DRIVEN engineering (MDE) techniques are not only used to create new software systems, but also can be useful in software evolution or modernization. The area of model-based software evolution is emerging and a great research and development effort will be needed in the next years. Recently, OMG has proposed several modernization standards in its ADM initiative [1], some tools are currently under development such as MoDisco [2], and even some research challenges have been identified for the evolution of systems built by using MDE techniques [3].

A model-driven evolution process requires models to be extracted from legacy software and most software evolution scenarios involve dealing with source code written in some programming language. Thus, a bridge from *grammarware* to *modelware* technical spaces must be built to extract models from source code. In the last years, several approaches for bridging *grammarware* and *modelware* have been proposed, which aim to define the concrete syntax of textual domain specific languages (DSL). These approaches can be categorized in two groups depending on whether they are focused on generating metamodels from grammars [4] [5], or they are concerned with creating grammars from existing metamodels [6]. Grammar-based approaches could be used to extract models from source code, but they have some drawbacks that may restrict its usefulness in model-driven modernization scenarios, such as the poor quality of the automatically generated metamodel or the fact that they does not permit reuse of existing grammars written for well-known parser generators.

In this paper we present an approach for the extraction of models conforming to a target metamodel from source code conforming to the grammar of a programming language. The target metamodel will represent some kind of knowledge about the source code, as for example a KDM-based metamodel [7] or an abstract syntax tree metamodel. We have defined a transformation language, named Gra2MoL (Grammar-to-Model Language), specifically designed to address the problem of extracting models from source code. Gra2MoL is a rule-based transformation language likes existing model-to-model transformation languages such as ATL or RubyTL, but with the fundamental difference that the source element of a rule is a grammar element instead of a source metamodel element. In this way, the language deals natively with grammars as source artifacts. Another important difference is that the language provides domain-specific features to address specific issues in grammar-to-model transformations. For instance, Gra2MoL provides a query language to ease the traversal of the syntax tree.

This paper is organized as follows. Section 2 shows the motivation of the approach, while Section 3 provides an overview of the proposed approach. In Section 4 the basic features of Gra2MoL are described. Section 5 discusses related work. Finally, in the last section we present our conclusions and outline the future work.

II. Motivation

In model-driven software evolution, MDE techniques (*modelware*) are used to understand and evolve existing software systems. During the evolution process, some models are first extracted from the existing system and then transformations are applied on these models to either obtain a higher level view of the system or to generate a new system. Since software systems are formed by artifacts such as source code or configuration files, different technical spaces [8] such as *grammarware* or XML can be involved in the evolution process. Therefore, a bridge between the MDE technical space *modelware* and these other technical spaces is needed in order to translate specifications expressed in languages of these other domains to models conforming to metamodels and vice versa.

Manuscript received February 25, 2008; revised Month Day, Year.

J. L. Cánovas, J. Sánchez and J. García Molina are with Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; {jlcanovas,jesusc,jmolina}@um.es.

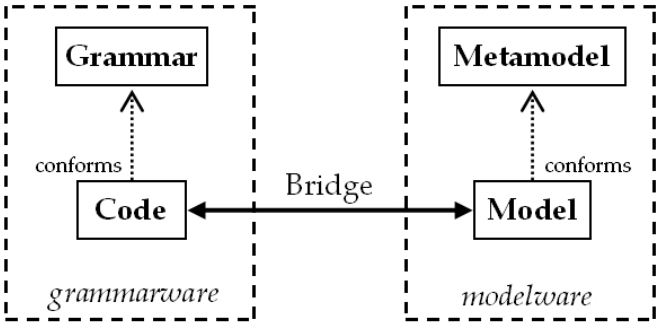


Fig. 1. Bridging *grammarware* to *modelware*

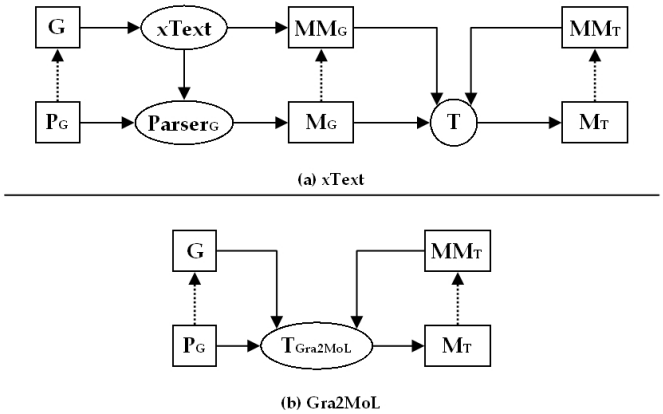


Fig. 2. xText grammar-based approach. A rectangle represents either a metamodel or a model, an ellipse represents a transformation, while dashed arrows represent conformance relationships.

Figure 1 shows a schema for the bridge between *grammarware* and *modelware*.

Most evolution scenarios involve dealing with source code conforming to the grammar of a programming language or textual files whose format can be described by a grammar. Thus, tools providing a bridge between the *grammarware* and *modelware* technical spaces are essential for model-driven software evolution. In this paper we present a proposal for bridging *grammarware* and *modelware*, which is oriented to modernization scenarios, where it is common to deal with a large amount of source code written in existing general purpose languages such as COBOL, C or Java.

Several approaches for bridging *grammarware* and *modelware* have been defined. These approaches can be classified in two groups according to whether they are focused on grammars or metamodels. Grammar-based approaches are oriented to generate metamodels from grammars, whereas metamodel-based approaches work on the opposite direction. In model-driven software evolution, the process starts from existing source code that conforms to the grammar of a programming language. Therefore, metamodel-based approaches are not well suited, but grammar-based approaches have to be considered. xText [4] and the works of Wimmer et al. [5] and Kunert [9] are examples of grammar-based approaches.

xText is a component of the openArchitectureWare toolkit. It allows us to build textual DSLs in the Eclipse platform. The textual concrete syntax of a DSL is specified by means of an EBNF-based language. As can be seen in Figure 2(a), the processing of a concrete syntax specification (G) generates the metamodel of the DSL (MM_G), a parser ($Parser_G$) to recognize the DSL syntax and to instantiate the metamodel, and a DSL specific editor. The generation of metamodels from concrete syntaxes relies on a number of rules that, for instance, are used to identify concept hierarchies or to generalize features in a hierarchy of concepts. Despite these rules, grammatical aspects still remain in the generated metamodels which tend to be quite verbose. Except for simple DSLs, it is usually necessary to define a more suitable metamodel for the DSL, such as a metamodel representing the underlying language abstract syntax. In this way, a model-to-model transformation from the generated xText metamodel to a proper abstract syntax metamodel is required to convert models generated by the parser to models conforming to the abstract syntax metamodel (or any other higher-level metamodel of interest).

Wimmer et al. have proposed a generic framework for bridging *grammarware* and *modelware*. In a first stage, a basic set of rules are applied to generate a raw metamodel from an EBNF grammar, such as the one generated by xText. Next, some heuristics are automatically applied to improve the raw metamodel, but a user-driven stage is needed to obtain the desired metamodel. Similarly, Kunert chooses to add annotations to the grammar in order to drive the generation process, but the annotations are rather limited. It is worth noting that tools supporting these two approaches are not available yet.

The usefulness of existing grammar-based approaches for model-driven modernization scenarios is restricted by some limitations that arise in practical situations. From our experience in several software evolution projects, we have identified four problems:

- The poor quality of the generated metamodels usually makes it compulsory to write a model-to-model transformation to obtain a higher level metamodel, such as an AST or a KDM metamodel (this is illustrated in Figure 2(a)). As we will discuss in this paper, this kind of model-to-model transformations are usually very similar and they involve intensive queries over the whole source model. Current transformation languages do not provide constructs to write them easily.
- As we have previously shown, grammar-based approaches automatically generate a metamodel representing the language grammar and the parser to generate models from the source code. These models are usually stored in XMI files. Since the generated metamodel is very close to the language grammar, the models mimic the source code. In large projects, where hundred of source files can be involved, this may provoke a duplication of information because software artifacts are both represented by source code files and models. This makes such approaches inefficient both from the space and time points of view.
- Parsing information such as filenames, lines, columns, etc. is important in the system evolution. For instance,

the KDM metamodel requires this kind of information. In current grammar-based approaches this information is lost since the generated metamodel does not include it, and therefore it is not propagated to models.

- There exists a considerable catalogue of grammar definitions for existing parser generators, such as ANTLR [10] or JavaCC [11]. Besides, software artifacts are usually programmed in well known programming languages, such as COBOL, C or Java, and writing such grammars from the scratch is a difficult, time-consuming task. Therefore, model-driven modernization approaches should allow grammar definitions for some of the existing parser generators to be reused.

Bearing in mind all the above mentioned problems, we have defined the approach presented in this work.

III. Proposed approach

In this section we present our proposal for bridging the *grammarware* and *modelware* technical spaces in model-driven evolution scenarios. Given a source program (P_G) conforming to the grammar (G) of a programming language, the objective is to generate a model conforming to a target metamodel (MM_T). This metamodel represent some knowledge about the source code, such as a KDM metamodel or an abstract syntax metamodel of a target language.

As we have discussed in the previous section, grammar-based approaches have a series of limitations that make them unsuitable for model-driven modernization scenarios. We propose to use a domain-specific transformation language, named Gra2MoL, to explicitly specifying the relationships between source grammar elements and metamodel elements. To accomplish that, unlike common model transformation languages, our language uses a grammar as the source artifact definition, instead of a metamodel. Gra2Mol treats source code as a model, using the underlying grammar definition as if it were a metamodel. Figure 2(b) shows this schema.

Therefore, the input of a Gra2MoL transformation is some source code along with the grammar definition it conforms to. Then, the source code is parsed to construct a syntax tree. A transformation definition deals with such a syntax tree, using the grammar definition for *typing* the tree nodes. As will be explained in Section IV-B, Gra2MoL transformation rules include query expressions for traversing the syntax tree and retrieving information.

The conformance relationship between the syntax tree and the grammar definition is used to allow navigation over the syntax tree. In this way, the following correspondences between grammar symbols and metamodel elements are identified:

Metamodel	Grammar
Metaclass	No terminal
PrimitiveType	Terminal
Attribute	Terminal in a rule's right hand side
Containment reference	No terminal in a rule's right hand side

It is important to note that a grammar can only represent explicitly containment references (i.e. composite associations). Non-containment references are implicitly represented in the grammar by means of identifiers.

Regarding cardinality of references, two cases need to be distinguished. If the EBNF formalism is used, extended operators such as "?", "*", "+" are enough to find out the cardinality of a no terminal symbol in a rule's right hand side. When BNF is considered, a syntax analysis of the corresponding grammar lambda productions or production recursivity must be done. Our current implementation is able to deal with EBNF extended operators. The following table shows the correspondences:

Cardinality	EBNF Operator
1:1	No operator
0:1	?
0:N	*
1:N	+

Now we will show how the correspondence between a grammar and a metamodel can be used to navigate over a syntax tree, and we will discuss some problems that arise in this approach. We will use the following excerpt of the Java grammar written in EBNF, which includes no-terminal symbols such as `classOrInterfaceDeclaration`, `classDeclaration`, `interfaceDeclaration`, and terminal symbols such as `CLASS` or `IDENTIFIER`.

```
classOrInterfaceDeclaration :
  modifier* (classDeclaration | interfaceDeclaration)

classDeclaration :
  CLASS IDENTIFIER typeParameters
    (EXTENDS type)?
    (IMPLEMENTS typeList)?
    classBody

classBody :
  LBRACE classBodyDeclaration* RBRACE

interfaceDeclaration :
  INTERFACE IDENTIFIER ...
```

Given a syntax tree node `n` of type `N`, a node of type `M` will be reachable from `n` by the expression `n.M` if there exists a grammar rule such that `N` is the left symbol and `M` appears on the rule's right hand side. For instance, given a syntax tree node `n` of type `classDeclaration`, the navigation expression to reach the identifier of the class will be `n.identifier`. In the same way, to access to the set of class' declarations, the expression will be `n.classBody.classBodyDeclaration`. In this expression, `classBody` acts as a reference with cardinality 1:1, while `classBodyDeclaration` acts as reference with cardinality 0:n.

There are, however, some problems regarding production rules with alternatives. In the grammar above, given a node `n` of type `classOrInterfaceDeclaration`, navigating through the alternatives (`classDeclaration | interfaceDeclaration`) is difficult because they do not share a common ancestor, so a conditional check must done to

decide the navigation path. This problem is studied in [5] but it is not solved properly, since it only prescribes introducing an empty anonymous superclass. Thus, each time the node is navigated, the path to follow must be explicitly decided, using a code similar to this:

```
if n.classDeclaration.exist?
  n.classDeclaration.identifier
else
  n.interfaceDeclaration.identifier
end
```

To address this situation, we propose to annotate the grammar (as it is done by xText), so that the alternative expression has a name. In this way, the rule will be rewritten as follows:

```
classOrInterfaceDeclaration :
  modifier*
  decl=(classDeclaration | interfaceDeclaration)
```

Then, the node can be accessed as `n.decl.identifier`. To allow this kind of expressions, the transformation language has been designed to support duck typing [12], that is, a node's property can be accessed if the node is able to return a value. In this way, a form of polymorphism is allowed without the need of inheritance.

Nevertheless, it is not compulsory to annotate the grammar. To avoid the problems of navigating over the syntax tree by means of the dot notation, a structure-shy query language [13] has been incorporated into the transformation language. Section IV-B, will elaborate on this topic.

Finally, in addition to dealing directly with grammars, the language provides some several domain-specific features that ease the work of extracting a model from source code conforming to a grammar. For instance, the structure of the syntax tree (i.e. there is only containment relationships between nodes) has allowed us to define a powerful query language, similar to XPath. On the other hand, primitives to get information such as the line that corresponds to a given node are provided. An outline of these features and a brief explanation of the language will be given in the next section.

IV. GRA2MOL

A. Language Description

Gra2MoL is domain-specific model transformation language specially intended to deal with source code described by a grammar. A Gra2MoL transformation generates a model which conforms to a target metamodel from a source code text which conforms to a grammar. Thus, it considers the source code as a model whose underlying grammar acts as its metamodel, and whose syntax tree nodes are treated as model elements. Gra2MoL is a rule-based transformation language whose rules have a similar nature to that of other model transformation languages such as ATL or RubyTL[1].

A transformation definition consists of a set of transformation rules which specify relationships between grammar elements and metamodel elements. A rule is composed of four parts, namely *from*, *to*, *queries* and *mapping*.

- The *from* part specifies the source grammar no-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 a filter expression that states the conditions to be satisfied by the nodes whose type is the no-terminal symbol.
- The *to* part specifies the target element metaclass.
- The *queries* part contains a set of query expressions which allow us to retrieve information from the syntax tree. The result of these queries will be used in the assignments of the *mapping* part.
- Finally, the *mapping* part contains a set of bindings to assign a value to the properties of the target model element. These bindings are a special kind of assignments, used in model transformations languages such as ATL [14] or RubyTL [15].

Next, we show a simple example of Gra2MoL transformation definition, which extracts a KDM model from Java code. The KDM model will only contain the non-void methods of Java classes to simplify the example. Figure 3 shows the parts of the Java grammar and the KDM metamodel considered in the transformation. We have only shown the grammar rules of interest for this example. Also, we have underlined the no terminal symbols used in the example. Regarding the KDM metamodel, we have considered four metaclasses: *Segment* of the *kdm* package and *codeModel*, *classUnit* and *methodUnit* of the *code* package. According to the KDM specification [7], a code model is contained in a KDM Segment and is composed of classes which are composed of methods.

The following rules specify a Gra2MoL transformation from Java source code to KDM, as explained above. Figure 4 shows the result of an execution of this definition.

```
rule 'createSegment'
  from compilationUnit cu
  to kdm::Segment
  queries
    class : /cu//#normalClassDeclaration;
  mapping
    model = new code::CodeModel;
    model.name = "codeModel";
    model.codeElement = class;
end_rule

rule 'createClass'
  from normalClassDeclaration nc
  to code::ClassUnit
  queries
    ms : /nc//#methodDeclaration[@methodName.exists];
  mapping
    name = nc.classId;
    codeElement = ms;
end_rule

rule 'createMethod'
  from methodDeclaration md
  to code::MethodUnit
  queries
  mapping
    name = md.methodName;
end_rule
```

In the following two sections we introduce the query language and the rule evaluation mechanism.

[1]In the current state of our research, Gra2MoL is a source-driven language such as ATL or RubyTL. However, our experiments are showing that a target-driven language can be well-suited for this kind of transformations. Therefore, in the future the shape of the language may vary.

```
compilationUnit
 : annotations? packageDeclaration?
   importDeclaration* typeDeclaration*
 ;

typeDeclaration
 : classOrInterfaceDeclaration
 ;

classOrInterfaceDeclaration
 : modifier* (classDeclaration | ...)
 ;

classDeclaration
 : normalClassDeclaration
 | ...
 ;

normalClassDeclaration
 : 'class' classId=Identifier (typeParameters)?
   ('extends' type)?
   ('implements' typeList)?
   classBody
 ;
```

```
classBody
 : '{' classBodyDeclaration* '}'
 ;

classBodyDeclaration
 : modifier* memberDecl
 | ...
 ;

memberDecl
 : methodDeclaration
 | ...
 ;

methodDeclaration
 : type methodName=Identifier
   methodDeclaratorRest
 ;

...
```

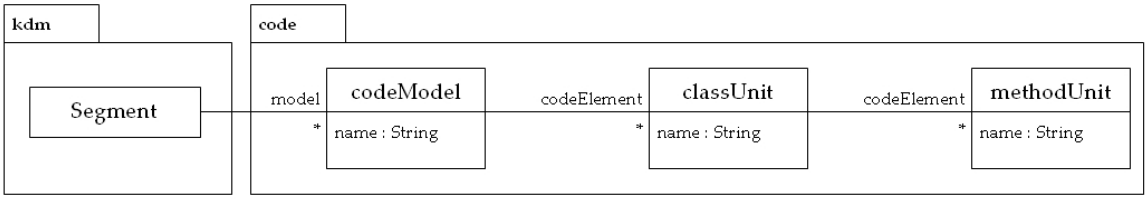
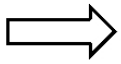


Fig. 3. Excerpt of the Java grammar and the subset of KDM metamodel used in the example.

```
public class BeanBulletinBoard extends ActionForm{

  private String selectedTitle = "";
  private String title = "";
  private String content = "";
  private String author = "";

  public ActionErrors validate(...){...}
  private boolean isNotValidTitle(...){...}
  public void setSelectedTitle(...) {...}
  public String getSelectedTitle() {...}
  public void setTitle(...) {...}
  public String getTitle() {...}
  public void setContent(...) {...}
  public String getContent() {...}
  public void setAuthor(...) {...}
  public String getAuthor() {...}
}
```



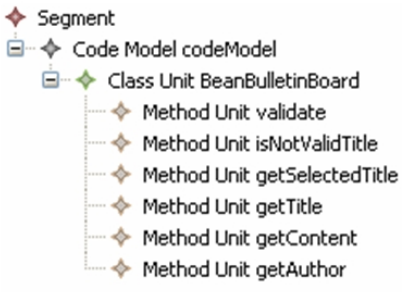


Fig. 4. Result of a Gra2MoL transformation execution.

B. The query language

Model transformation languages usually provide model query facilities in the form of OCL-like language expressions, which use the dot notation to traverse the model graph. This is enough for most practical model-to-model transformation definitions, but the transformations that are addressed in the grammar-to-model extraction step of model-driven modernization have a different nature.

Since programming language grammars produce syntax trees, references between elements must be implicitly established by means of identifiers. On the contrary, models are graphs where references between elements are explicit. Transforming an identifier-based reference into an explicit reference involves finding the "identified" node, which may be out of the scope of the rule performing such a transformation. Therefore, grammar-to-model transformations involve intensive queries over the whole syntax tree to retrieve information that is out of the scope of the current rule. In [16] this kind of transformations is called global-to-local transformations. If

dot-notation is used to write such queries, long navigation chains must be written. Thus, we have developed a structure-shy language, inspired in XPath to allow navigation on the syntax tree without the need of specifying every navigation step (i.e. to avoid specifying long navigation expressions).

A query is applied to the syntax tree, where each node is typed with a grammar symbol, and it traverses the syntax tree to collect a set of nodes that satisfies the query operators conditions. There are two kinds of query operators: `/` and `//`. They both specify the type of the node that must be found. The `/` operator returns the immediate children of a node, while the `//` operator navigates along all node's children (direct and indirect children) retrieving all nodes of a given type. The `//` operator allows us to ignore intermediate superfluous nodes, so making easy the query definition, since it specifies what kind of node must be found, but not how to reach it.

Query operators can include an optional filter expression, enclosed between square brackets. A filter expression is a logical expression which is applied to the leaves of the node specified in the query operator. Each operand of a filter is a boolean function which checks leaves properties, for instance, the existence or the value of a leaf. Only those nodes satisfying the filter expression will be selected. In addition, since a query returns one or more subtrees of the syntax tree, one may be interested in only a certain part of such subtrees. The `#` character is used to indicate the type of root nodes of the query result. It is worth noting that it must only be specified for one of the query operators. In the Java-to-KDM example, the query `/nc//#methodDeclaration[@methodName.exists]` of the second rule returns all `methodDeclaration` nodes, for a given `normalClassDeclaration` node (the query uses the variable defined in from part), which has a `methodName` leaf.

C. Bindings and rule evaluation

A binding construct is used in the *mapping* part to establish the relationship between source grammar elements and target metamodel elements. These bindings have an equivalent syntax and semantics to such bindings of the RubyTL model-to-model transformation language [15]. They are written as assignments using the operator "=". The right side can be the identifier declared in the *from* part of the rule, a literal value or a query identifier, and the left side must be the name of a property of the metaclass declared in the *to* part.

The definitions of rule conformance and well-formed transformation stated in [15] for RubyTL are applicable to Gra2MoL with simple changes. In the current version, the rule evaluation is also driven by bindings. When a rule is applied on a node, first the filter is checked, and then if the node satisfies the conditions, an instance of the target metaclass is created, and the rule's bindings are executed. In the execution of a binding three situations can be found according to the nature of the right side.

1) If it is a literal value, the value is directly assigned to the property of the left side. In the Java-to-KDM example, `model.name = "codeModel"` is an example of this situation.
2) If it is a query identifier, the query is executed and a rule defining the relationship between the query result type (marked with `#`) and the type of the metaelement property is looked up in the transformation definition. This rule is executed for each result element of the query. In the Java-to-KDM example, the assignment `methods = md` means that there exists a rule whose from part is the *methodDeclaration* grammar type and its *to* part conforms to *Java::Method*. It is important to note that conformance between types in the *to* part must take into consideration inheritance between metaclasses. In this case, the *typeMethod* rule conforms to the previous constraints and will be executed,
3) If it is an expression, it is evaluated and there are two situations depending on whether the result is a node whose type corresponds to a terminal (a leaf) or a no terminal symbol. If it is a leaf, the result is a primitive type and is assigned directly (e.g. `name = md.methodName` in the previous Java-to-KDM example), otherwise, a rule to resolve the binding is looked up and executed.

D. Implementation

Current implementation of Gra2MoL uses the ANTLR parser generator to define the grammar of the source code handled by a transformation definition. Parser generators in general, and ANTLR in particular, allow actions to be attached to grammar rules, so that computations can be made during the parsing (e.g. to construct the syntax tree). 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 syntax tree programmatically, (2) to allow grammars to be reused, which usually do not include any actions.

Therefore, each grammar used in a Gra2MoL transformation definition must be enriched with actions, which will be in charge of creating the syntax tree that will be used during the grammar-to-model transformation execution. Gra2MoL uses internally a metamodel to represent generically the concrete syntax tree (CST) of the parsed source code. This metamodel is shown in Figure 5. There are three kinds of element in a

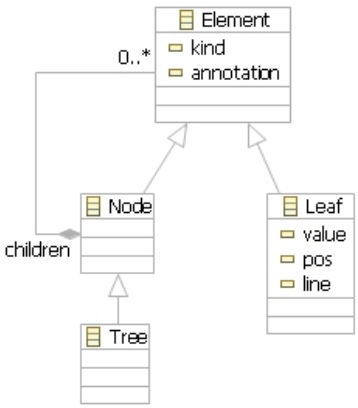


Fig. 5. CST metamodel

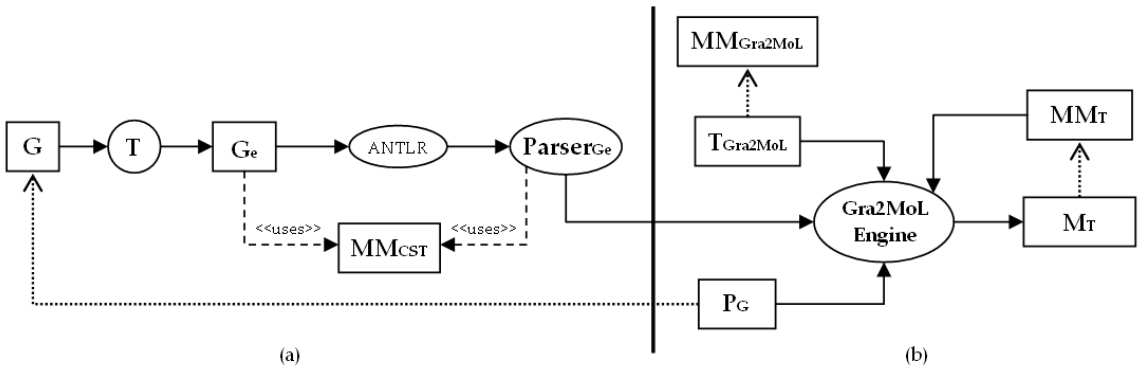


Fig. 6. Proposed approach implementation

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 no terminal symbol, and is composed of one or more children nodes, either of type `Leaf` or `Node`. The `kind` attribute identifies the grammar symbol whose recognition yielded to the tree node creation (this is needed to navigate through the syntax tree as explained in Section III). Finally, `Tree` represents the root node of the syntax tree. It is worth mentioning that the enrichment process is automatic, and can be applied to any ANTLR grammar. In addition, since the developer of the Gra2MoL definition writes the transformation on the basis of the language grammar, the existence of an underlying CST model is transparent.

Once the original grammar (G) has been enriched (G_e), we rely on ANTLR to generate a parser ($Parser_{G_e}$), whose actions will create a model conforming to the CST metamodel. This process is illustrated in Figure 6(a), and can be seen as a preprocessing step to make the ANTLR grammar compatible with Gra2MoL.

The execution process of a Gra2MoL transformation is shown in Figure 6(b). The Gra2MoL engine is fed with the following artifacts:

- One or more source files, P_G, conforming to the grammar G of the source language.
- A reference to the parser generated in the preprocessing step $Parser_{G_e}$ must also be provided to the Gra2MoL engine, so that a CST model can be created from the source files.
- The target metamodel MM_T to which the output model will conform. For instance, the KDM metamodel.
- A Gra2MoL transformation definition.

Before executing the transformation the source files are read by the parser, so that an in-memory CST model is created. When the transformation is executed, transformation rules and queries will traverse the CST model and a target model M_T is constructed.

V. Related work

As explained previously, several approaches for bridging *grammarware* and *modelware* have been defined. xText [4] and the work of Wimmer et al. [5] are the main examples of theses approaches.

The main shortcoming of xText and the work of Wimmer et al. is the need of enhancing the quality of the generated metamodel. In xText, a model-to-model transformation must be defined to obtain a higher level metamodel, and in the work of Wimmer et al. the metamodel must be annotated and there are some manual steps involved. Our proposal allows a higher-level model, such as KDM model or an AST model, to be created by defining a transformation from a grammar to a metamodel using the Gra2MoL transformation language. Since Gra2MoL has been explicitly designed to address this kind of transformation, it is easier to write such transformations than using normal transformation languages. Gra2MoL also allow us to access information such as the line or column a node is located in the source code. Moreover, since Gra2MoL does not use a special language to define the grammar, but the ANTLR one, reuse of existing grammars is promoted.

Several projects aimed to provide tools and methodologies for the model-driven modernization are currently under development [2] [17]. MoDisco (*Model Discovery*) is an extensible approach for model-driven reverse engineering. Its objective is to discover and extract models from legacy systems and it has been defined as part of the Eclipse GMT project [18]. The framework components are: 1) A KDM based metamodel, 2) A metamodel extension's mechanism, 3) Facilities for manipulating models and 4) A methodology for designing extensions. MoDisco is being developed at the moment, and it only offers the infrastructure to manage and create models. MoDisco defines the *discoverer* concept, which is a piece of software in charge of analyzing part of an existing system (for instance, source code written in Java) and extracting a model using the MoDisco's infrastructure. In this way, our proposal can be used to write MoDisco discoverers.

VI. Conclusion and future work

In this paper we have presented an approach for the extraction of models conforming to a target metamodel from source code conforming to the grammar of a programming language. This approach defines Gra2MoL, which is a transformation language specially tailored to address the *grammarware*-

modelware bridge in modernization scenarios. The language promotes grammar reuse and provides domain-specific features such as a powerful query language to traverse syntax trees.

Although Gra2MoL is in prototype state, it is being used in a platform migration modernization scenario. In particular, it is being applied to migrate Struts applications to JSF. Initially we have built a Gra2MoL interpreter in order to evaluate the language, but we are considering writing a Gra2MoL compiler to improve execution performance. Also, it only deals with ANTLR grammars, but we would like to support other parser generators.

Regarding the future work, we are studying whether or not the current rule structure (which is source-driven) is really suitable for this kind of transformations. Also, we will study how to improve the navigation over the syntax tree by recognizing certain grammatical structures (such as rule recursion).

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. Jesús Sánchez enjoys a doctoral grant from the Spanish Ministry of Education and Science.

REFERENCES

[1] OMG, "Architecture-driven modernization roadmap," OMG, Tech. Rep., 2006.
[2] "Modisco." [Online]. Available: 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] S. Efftinge, "openarchitectureware 4.1 xtext language reference," http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtextReference.pdf, August 2006.
[5] M. Wimmer and G. Kramler, "Bridging grammarware and modelware," *Satellite Events at the MoDELS 2005 Conference*, pp. 159–168, 2006.
[6] F. Jouault, J. Bézivin, and I. Kurtev, "Tcs: a dsl for the specification of textual concrete syntaxes in model engineering," in *GPCE*, 2006, pp. 249–254.
[7] ADMTF, "Knowledge discovery meta-model (kdm)," 2007. [Online]. Available: http://www.omg.org/spec/KDM/1.0/
[8] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," in *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002. [Online]. Available: http://www.sciences.univ-nantes.fr/lina/atl/www/papers/PositionPaperKurtev.pdf
[9] 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, Ed., 2006.
[10] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, May 2007.
[11] "Java compiler compiler." [Online]. Available: https://javacc.dev.java.net/
[12] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, October 2004.
[13] T. Cleenewerck and I. Kurtev, "Separation of concerns in translational semantics for dsls in model engineering," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2007, pp. 985–992.
[14] F. Jouault and I. Kurtev, "Transforming models with atl," 2005.
[15] J. S. Cuadrado, J. G. Molina, and M. M. Tortosa, "Rubytl: A practical, extensible transformation language," in *ECMDA-FA*, 2006, pp. 158–172.
[16] 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.
[17] "Momocs - methodology and related tools for fast reengineering of complex systems." [Online]. Available: http://www.momocs.org/
[18] "Gmt project." [Online]. Available: http://www.eclipse.org/gmt/

Javier Luis Cánovas Izquierdo is a PhD candidate at the University of Murcia. His research interests are model-driven development and model-driven modernization. Contact him at the Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; jlcanovas@um.es.

Jesús Sánchez Cuadrado is a PhD candidate at the University of Murcia. His research interests are model-driven development, model transformation languages, and dynamic languages. He received his master's in computer science from the University of Murcia. Contact him at the Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; jesusc@um.es.

Jesús García Molina is a professor of software design at the University of Murcia, where he leads the Software Technology Research Group. His research interests include model-driven development, domain-specific languages, and software processes. He received his PhD in science from the University of Murcia. Contact him at the Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; jmolina@um.es.