Softw Syst Model (2014) 13:713–734
DOI 10.1007/s10270-012-0270-z

THEME SECTION PAPER

Extracting models from source code in software modernization

Javier Luis Cánovas Izquierdo · Jesús García Molina

Received: 22 March 2011 / Revised: 25 April 2012 / Accepted: 23 July 2012 / Published online: 9 September 2012

Abstract Model-driven software modernization is a discipline in which model-driven development (MDD) techniques are used in the modernization of legacy systems. When existing software artifacts are evolved, they must be transformed into models to apply MDD techniques such as model transformations. Since most modernization scenarios (e.g., application migration) involve dealing with code in general-purpose programming languages (GPL), the extraction of models from GPL code is an essential task in a model-based modernization process. This activity could be performed by tools to bridge *grammarware* and MDD technical spaces, which is normally carried out by dedicated parsers. Grammar-to-Model Transformation Language (Gra2MoL) is a domain-specific language (DSL) tailored to the extraction of models from GPL code. This DSL is actually a text-to-model transformation language which can be applied to any code conforming to a grammar. Gra2MoL aims to reduce the effort needed to implement *grammarware*-MDD bridges, since building dedicated parsers is a complex and time-consuming task. Like ATL and RubyTL languages, Gra2MoL incorporates the binding concept needed to write mappings between grammar elements and metamodel elements in a simple declarative style. The language also provides a powerful query language which eases the retrieval of scattered information in syntax trees. Moreover, it incorporates extensibility and grammar reuse mechanisms. This paper describes Gra2MoL in detail and includes a case study based on the application of the language in the extraction of models from Delphi code.

Keywords Model-driven engineering · Model-driven software development · Domain-specific languages · Software modernization · Model-driven software modernization

Communicated by Dr. D. Tamzalit, B. Schätz, D. Deridder and A. Pierantonio.

J. L. Cánovas Izquierdo (✉) · J. García Molina
University of Murcia, Murcia, Spain
e-mail: jlcanovas@um.es

J. García Molina
Depto. de Informática y Sistemas, Facultad de Informática,
Universidad de Murcia, Campus de Espinardo,
30071 Murcia, Spain
e-mail: jmolina@um.es

J. L. Cánovas Izquierdo
AtlanMod, Ecole des Mines de Nantes, INRIA,
LINA, EMN, Nantes, France
e-mail: javier.canovas@inria.fr

Present Address:
J. L. Cánovas Izquierdo
École des Mines de Nantes, La Chantrerie 4,
rue Alfred Kastler, B.P. 20722, 44307 Nantes, France

1 Introduction

Model-driven software development (MDD) is gaining increasing acceptance, mainly as a result of its ability to raise the level of abstraction and automation in the construction of software. Although the most common MDD approaches (e.g., MDA, software factories or domain-specific development) are aimed at building new software systems, models have also shown the potential to evolve existing systems. MDD techniques, as metamodeling and model transformations, can help to reduce software evolution costs and improve the quality of the artifacts evolved by automating many basic activities in software change processes, such as representing source code at a higher level of abstraction [1] or obtaining information such as metrics [2].

The growing interest in using MDD to manage software evolution is mainly focused on the reengineering or modernization of legacy systems. Several software migration projects have been carried out with model-driven approaches [1,3,4]; modernization tool vendors are offering model-driven solutions (OBEO or Mia Software). And the OMG's Architecture Driven Modernization (ADM) initiative [5] is defining a set of standard metamodels which represent the information normally managed in modernization tasks.

When software artifacts are evolved by applying MDD, it is necessary for them to be represented as models to execute model transformations that generate new evolved artifacts. For example, in a scenario of language-to-language migration, the first step is to extract models from the application code written in the source language. Similarly, in a modernization process to improve the data quality, models should be extracted from data schemas. Once these initial models have been obtained, model transformations can be applied to generate higher level abstraction models and finally the new artifacts (e.g., code in other language or an improved data schema). Other operations on models, such as model comparison or synchronization, may also be applied.

Since most modernization scenarios [6], such as language-to-language conversion or platform migration, involve dealing with code expressed in some general-purpose programming languages (GPL), techniques and tools providing efficient means to extract models from GPL code are essential in model-driven modernization. In these scenarios, models conforming to a target metamodel (e.g., an abstract syntax tree metamodel) should be obtained from source code conforming to the grammar of a GPL. Although a modernization can also involve non-GPL code, which could also conform to a grammar (e.g., code of scripting language), most of the source code to be evolved is GPL code.

The relationship between the pairs of concepts grammar/program and metamodel/model is an example of a bridge between two different technical spaces [7], in particular *grammarware* and MDD (also known as *modelware*). Several tools whose aim is to define textual domain-specific languages (DSLs), such as Xtext [8] or EMFText [9], provide a *grammarware–modelware* bridge which allows models to be extracted from a DSL program. However, these tools are not appropriate for the extraction of models from GPL code, because DSLs have a simpler structure than GPLs, which require an in-depth customization of the tool. Dedicated parsers (also known as model discoverers) [1,3] are,therefore, normally implemented to obtain models from code conforming to a grammar. These parsers perform model-generation tasks in addition to code parsing. Firstly, a syntax tree (i.e., an abstract or concrete syntax tree) is created from the source code, and this syntax tree is then traversed to obtain the information needed to create the model elements. This is a complex task which requires both collecting scattered information and resolving references in the syntax tree.

Since the construction of such dedicated parsers is a time-consuming task, we have defined a DSL, called Grammar To Model Transformation Language (Gra2MoL), which has been specifically designed to extract models from GPL code, although it can be used for any software language conforming to a grammar. Model transformations are classified into three categories [10]: model-to-model transformations whose input and output are models; model-to-text transformations, which generate software artifacts (e.g., GPL code and database schemas) from a source model, and text-to-model transformations which obtain models from existing software artifacts. Gra2MoL would, therefore, be a text-to-model transformation language whose source artifacts must be described by a grammar. While there are a number of transformation languages for model-to-model transformations (e.g., ATL [11], QVT [12] or RubyTL [13]) and model-to-text transformations (e.g., MofScript [14], Xpand [15]), Gra2MoL might be considered the first proposal for a text-to-model language, at least to the best of our knowledge.

When designing a model transformation language, two key design choices are how to express the mappings between source and target elements and how to navigate through the source artifact. Gra2MoL allows mappings to be established between grammar elements and target metamodel elements in a declarative manner that is similar to how mappings are expressed in model-to-model transformation languages such as ATL or RubyTL using the *binding* construct [11]. Furthermore, as a Gra2MoL transformation represents the code in the form of a syntax tree, Gra2MoL provides a powerful query language to ease the navigation and querying of such a tree when writing mappings.

The first version of Gra2MoL, which supported the core features of the language, was presented in refs. [16,17]. Since then, the language has evolved to include new basic features such as: (1) reuse mechanisms at rule-level (i.e., mixin rules), (2) a new kind of rule for dealing with expressions efficiently (i.e., skip rules), and (3) an extensibility mechanism to add new operators. In addition, the development of new features along with the experience gained in using the language in several case studies (model extraction from PL/SQL, Delphi, Bash scripts and more are included in the Gra2MoL website [18]) allowed us to identify new functionalities to improve the expressiveness, usability and performance. The new added extensions are the following: (1) iterators and operators in the query language, (2) copy rules as they exist in ATL and RubyTL, (3) CDO and Morsa model repositories are supported to manage large models efficiently, and (4) support for island grammars. Experiences with Gra2MoL have shown significant advantages in relation to using dedicated parsers: a reduction in development time, the maintenance is facilitated and existing grammars can be reused.

This paper is organized as follows. Section 2 analyzes the difficulties encountered when using existing solutions for model extraction, and the motivation for Gra2MoL is presented. In Sect. 3, we describe the language used to query 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, Sect. 6 presents our conclusions and some future work.

2 Model extraction from source code

This section aims to motivate the approach proposed in this paper. Firstly, model extraction is presented as a task which requires a bridge to be built from *grammarware* to *modelware*, and then, several approaches are contrasted as possible techniques for implementing such a bridge. We identify the main issues to be addressed and discuss the limitations of each approach. Finally, we introduce the Gra2MoL language and indicate how this DSL overcomes the limitations identified previously.

Figure 1 shows the elements involved in the process of extracting models from code conforming to a grammar. This process is a text-to-model transformation T which has as its input a program P along with the grammar definition G to which it conforms. The transformation manages P as either an abstract syntax tree (AST) or a concrete syntax tree (CST). In this paper, we use the term "syntax tree" to refer to both AST and CST. The execution of T generates a target model M_T conforming to a target metamodel MM_T representing the information to be extracted, which is usually more complex than a syntax tree. The extraction process is driven by a specification of the mappings between the grammar elements and the metamodel elements. As we will see, the form of these mappings is different depending on each approach considered for the extraction.

The notion of bridging technical spaces is proposed in ref. [7] to address the integration of MDD with other technologies (e.g., grammar, XML or ontologies). A text-to-model transformation is, therefore, an example of a task which requires a unidirectional bridge to be built between *grammarware* and *modelware* technical spaces as illustrated in Fig. 2, which shows OMG's layered metamodeling architecture as realized for these technical spaces.

With GPL code, creating this bridge requires an efficient mechanism to traverse syntax trees since the model elements

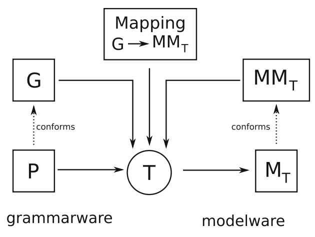


Fig. 1 Process of extracting models from source code

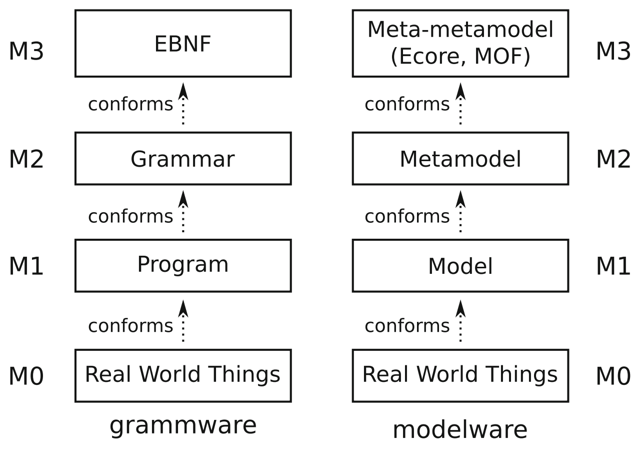


Fig. 2 Bridge between *grammarware* and MDE technical spaces

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 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 (ref. [19] calls this kind of transformations global-to-local transformations). The scattering problem may also appear when the semantic gap between the source code and the target metamodel is high, e.g., a model element representing a metric that counts the number of classes could require the traversal of the source code to count all class declarations in Java.

Two main issues to be tackled by a mechanism for extracting models from GPL code are therefore: (1) establishing the mapping between grammar elements and metamodel elements, and (2) retrieving scattered information from syntax trees. Next, we analyze how dedicated parsers, DSL definition tools, program transformation languages and model-to-model transformation languages could be used to build *grammarware–modelware* bridges in the case of GPL code.

2.1 Approaches for model extraction

2.1.1 Dedicated parsers

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 this syntax tree to generate the target model. For example, both in refs. [1] and [4], dedicated parsers are built to extract models from PL/SQL code. However, dedicated parser development is a time-consuming and expensive task, because the syntax tree traversals must be hardcoded both to collect scattered information and to resolve references. In addition, mappings are also hardcoded, which hinder mantainability. The effort required is usually alleviated by automatically extracting an AST from the source code. This step is performed using an API, which is intended to make the management of this tree easier. An example of such APIs is the JDT Eclipse project [20], 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.

A strategy to help to build dedicated parsers is supported by the MoDisco (Model Discovery) modernization framework [21], which is part of the Eclipse Generative Modeling Technology (GMT) component [22]. This framework is currently under development and its objective is to facilitate the construction of tools to support software modernization use cases. It provides (1) a set of metamodels to describe software systems (e.g., an implementation of the KDM metamodel [23]), (2) tools to understand complex systems (e.g., a model editor specially adapted to deal with huge models) and (3) dedicated parsers ("discoverers" in MoDisco terminology) to obtain models from legacy systems and use them in modernization use cases. The discoverers which are currently available allow models representing the syntax tree to be extracted from XML files and Java source code. The developer must, therefore, traverse the extracted models to obtain models conforming to the target metamodel and model-to-model transformations are still needed. It is also important to note that our approach was developed at the same time that MoDisco was being implemented.

2.1.2 DSL definition tools

The definition of textual DSLs aimed to express models in MDD solutions is another scenario in which a *grammarware–modelware* bridge is needed. Since textual DSL definition tools (also known as language workbenchs [24]) provide the functionality of converting DSL programs into models and vice versa, they must implement one of these bridges. These tools generate a dedicated parser and a DSL editor from the specification of the DSL's abstract and concrete syntaxes. They may, therefore, be considered as an alternative to developing a dedicated parser.

Two approaches are supported by these tools to specify both the abstract and the concrete syntaxes. In grammar-based tools, such as Xtext [8] and TEF [25], the developer uses an EBNF-like notation to specify both the grammar, which include rules intended to specify the mapping for the corresponding metamodel, and the concrete syntax. In some cases, such as in Xtext, the metamodel can also be automatically generated from this specification. On the other hand, metamodel-based tools, such as EMFText [9] and TCS [26], have a metamodel with annotations that specify the concrete syntax as input, and the grammar is automatically generated from this annotated metamodel. Indeed, a tool can support both approaches as in the case of the last version of Xtext.

Metamodel-based definition tools are not well suited to deal with GPLs as stated in ref. [26]: "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). As a DSL has a simpler structure than a GPL, these tools do not address several problems encountered in the management of GPL code. This unsuitability is evidenced when EMFText is used to implement bridges for GPLs. The tool must be customized in depth, mainly to adapt the generated grammar to the GPL one. For instance, the work needed to implement a Java bridge implied so many changes to the tool that a new project called Jamopp [27] had to be created. Moreover, these approaches are not well suited to a model-driven modernization, since a metamodel corresponding to the GPL grammar is not usually available.

With regard to grammar-based approaches, several important limitations arise when they are used to extract models from GPL code. Regarding Xtext, the metamodel generated is of poor quality, because it includes superfluous elements and grammatical aspects, and the semantic gap between this metamodel and the desired target metamodel (e.g., an AST 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 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 this transformation is a complex task, as described below when commenting on model transformation languages. With regard to TEF, although this tool can use any target metamodel, it only provides mechanisms to resolve simple references existing in DSLs (i.e., identifier-based references), for more complex references (e.g., package-based references in GPLs) it would require reference solvers to be hardcoded.

Moreover, neither existing grammar reuse (i.e., the reuse of grammars for well-known parser generators such as ANTLR) nor the reuse of Xtext/TEF grammar specifications is promoted. On the one hand, translating a grammar specification provided by a parser generator into the

EBNF-based specification used 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, these grammar specifications are oriented toward a specific metamodel so they include specific rules for such a metamodel.

Wimmer et al. [28] and Kunert [29] 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.

Prinz et al. [30] presented a metamodel-based approach to define the SDL language which outlines a notation for expressing mappings between grammar and metamodel elements. However, when using this framework with GPLs, the main problem which arises is the lack of support for resolving references. Although the concept of *identifier resolvers* is incorporated to tackle this problem, it is still necessary to hardcode them.

2.1.3 Program transformation languages

Program transformation languages, such as Stratego/XT [31] and TXL [32], 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 a tool for bridging *grammarware* and *modelware* would still, therefore, be 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).

2.1.4 Model transformation languages

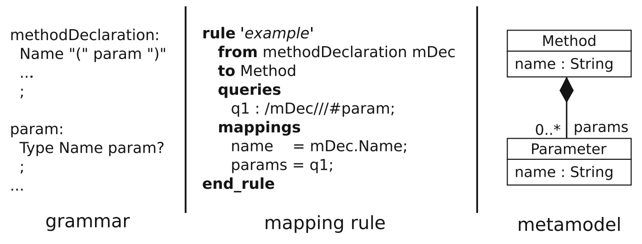
Similarly, model-to-model transformation languages could also be used by first obtaining a simple intermediate model (i.e., a syntax tree model) from the code by means of a dedicated parser. However, defining the transformation would lead to an important problem: the inadequacy of the query language. Most model transformation languages, such as ATL or QVT, provide a variant of the OCL navigation language [33] which allows model graphs to be traversed. Although OCL-like expressions are appropriate for most practical model-to-model transformations, 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, as we illustrate in Sect. 3. Integrating a more suitable query language into 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.

2.2 Our approach for model extraction

In the context of an Oracle Forms migration project, we confronted model extraction from PL/SQL code. We, therefore, considered the definition of a DSL to overcome the limitations of the previously discussed approaches. This DSL had to shorten 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 was necessary to raise the two key-design issues indicated in Sect. 2: 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.

The DSL created, denominated as Gra2MoL, provides constructs to write mappings at a high level of abstraction in a declarative style similar to how mapping are expressed in model-to-model transformation languages such as ATL or RubyTL. With regard to the support of traversing syntax trees, Gra2MoL provides a powerful query language for syntax trees. This query language is introduced in the following section and the DSL is described in detail in Sect. 4.

Figure 3 shows a first example of how Gra2MoL is used. A Gra2MoL definition consists of a set of rules, each one of which express the mapping between a grammar element and a model element. The Gra2MoL definition shown in the example is very simple, and only contains the rule named `example` which transforms a `methodDeclaration` grammar element (see Fig. 3a) into the `Method` metamodel element (see Fig. 3c) according to the *from* and *to* parts of the rule. The *mapping* part expresses how the information of the model element is obtained from the information in the syntax tree. In this example, the `name` attribute of the `Method` model element is first initialized by accessing to the `Name` grammar element of the `methodDeclaration`



```
methodDeclaration:
  Name "(" param ")"
  ...
  ;

param:
  Type Name param?
  ;
...
```

```
rule 'example'
  from methodDeclaration mDec
  to Method
  queries
    q1 : /mDec//#param;
  mappings
    name    = mDec.Name;
    params = q1;
end_rule
```

Fig. 3 Simple example of a Gra2MoL mapping definition

Table 1 Comparison of Gra2MoL with the analyzed approaches

Approach	Syntax tree navigation	Artifacts to be created	Pre processing	Post processing	Existing grammar reuse	Provided grammar reuse	Purpose
Dedicated parser (+API)	GPL code (+primitives)	$MM_T P$	None	None	Yes	NA	Specific model extraction
MoDisco	OCL-like query languages	$MM_T D$	Discoverer (if not dealing with Java or XML)	$m2m$ transf.: $MM_I \rightarrow MM_T$	Yes	NA	General-purpose model extraction
DSL definition tools	Poor support	$G_{xt} MM_T m2m$	None	$m2m$ transf.: $MM_I \rightarrow MM_T$	No	No	DSL creation
Program transf.	Stratego incorporates a query language [34]	$MM_T T_{PT} G_{AS} m2m$	None	Extracting a model from a program conforming to G_{AS}	Limited (a few grammars)	Yes	Program transformation
Model transf.	OCL-like query languages	$MM_T P m2m$	None	None	Yes	NA	Model transformation
Gra2MoL	Structure-shy query language	$MM_T T$	None	None	Yes	NA	General-purpose model extraction

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/TEF grammar, $m2m$ model-to-model transformation definition, G_{AS} abstract syntax grammar, D discoverer

grammar element received by the rule (variable `mDec`). The `params` reference is then initialized using the `q1` query, which collects every `param` grammar element representing the paramteres of the method. Note that mappings are specified explicitly and a specific query language is used to traverse the syntax tree.

Table 1 contrasts Gra2MoL with the approaches analyzed. The columns show the properties which are compared: the ability to navigate the syntax tree; which artifacts must be created; whether pre-processing (only required in MoDisco when there is no discoverer for the GPL at hand) and/or post-processing is necessary (it is normally required to eventually obtain a model conforming to the desired metamodel); whether it is possible to reuse existing (e.g., grammars provided by ANTLR) and provided grammars (i.e., grammars defined by the formalism used in the approach); the main purpose of the approach. The artifacts to be created, and both the pre-processing and post-processing tasks determine the level of effort involved in 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. Both model-to-model and MoDisco approaches requires a great effort to define the model transformation needed to obtain the target model. In addition, MoDisco also requires implementing the discoverer if the language involved is not Java or XML. 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 to the extraction of models. As a consequence, development time is reduced using Gra2MoL.

3 A query language for concrete syntax trees

As stated previously, transforming GPL into models involves the intensive use of traversals through the syntax tree to collect scattered information. A model extraction language must, therefore, provide a powerful query language, which facilitates the access to tree nodes outside the current construct scope (i.e., a rule). Figure 4 illustrates the scattering problem for a simple example of extracting an ASTM model element from a Delphi procedure. Abstract Syntax Tree Metamodel (ASTM) [35] is a metamodel provided by ADM to represent the source code of the software system as ASTs. Since the scattered information problem appears in both AST and CST, and obtaining a CST is easier than obtaining an AST, Gra2MoL uses CSTs to represent the source code. The CST shown in Fig. 4 corresponds with a procedure declaration which includes a variable declaration and an assignment statement initializing the declared vari-

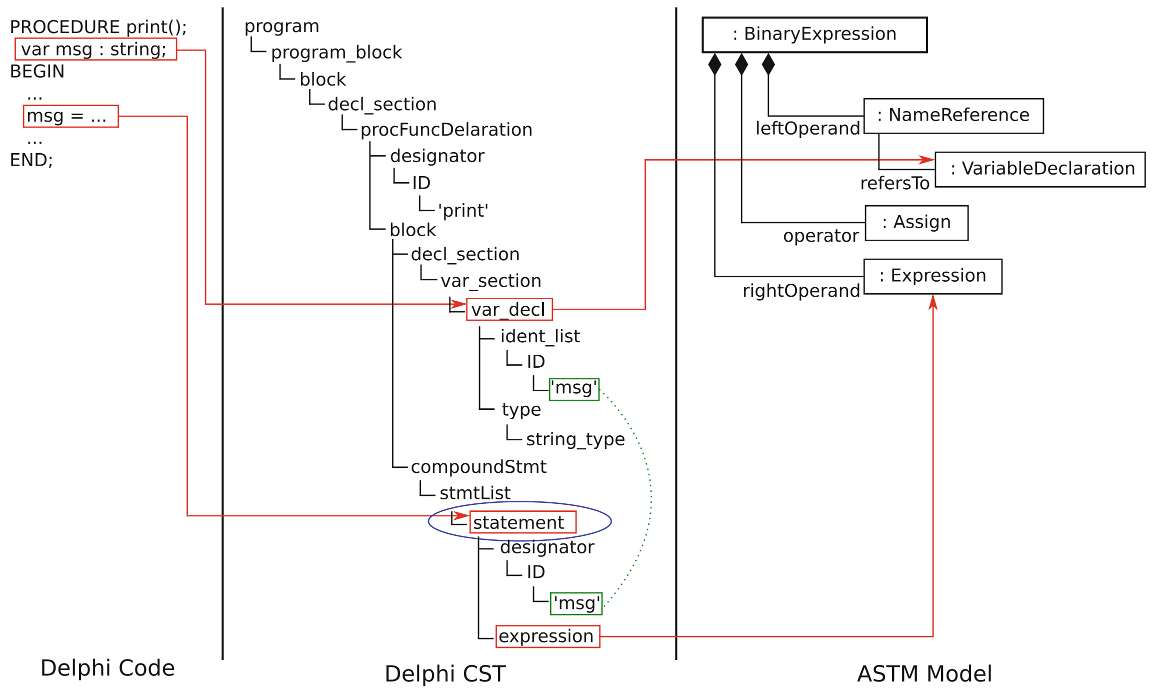


Fig. 4 Example of scattered information. The *oval* indicates the current scope and the *dotted line* indicates an identifier-based reference between tree elements

able. The `BinaryExpression` model element represents binary expressions, which are used to represent assignments in ASTM. This model element has two properties to register the right-hand side and left-hand side expressions of the assignment (i.e., `rightOperand` and `leftOperand` references, respectively) in addition to a property to specify that the binary expression is an assignment (i.e., `operator` reference). As can be observed, whereas all the information needed to initialize the right-hand side attribute (the `expression` grammar element of the assignment) are inside the current scope (depicted as an oval), the information needed to initialize the left-hand side attribute are outside this scope, because the `msg` variable declaration is referenced by an identifier. A query language might, therefore, help to resolve this reference to the `msg` variable, by providing navigation constructs for accessing the corresponding declaration node and retrieving the variables' properties.

We have created a structure-shy query language, inspired by XPath [36], which allows a CST of the source code to be navigated without the need to specify each navigation step. The terms "structure-shy" is often used to refer to behaviour specifications (e.g., queries) which are loosely bounded to the data structures on which operations (e.g., syntax trees) are applied.

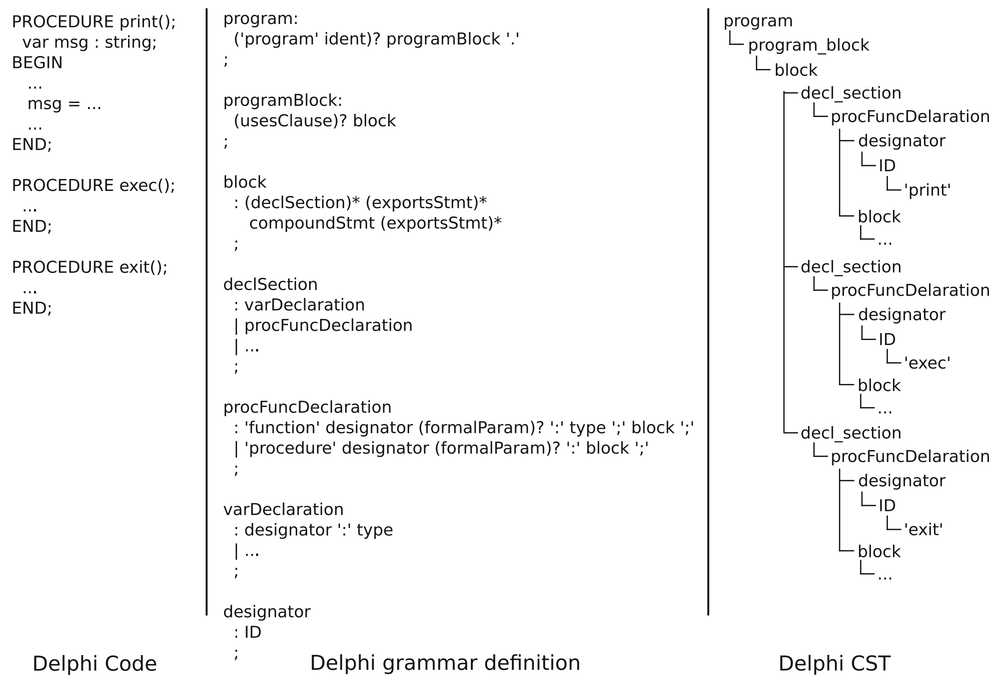
To navigate the CST, the nodes are "typed" using the grammar definition, and each tree node registers the name of the grammar element as its type. Figure 5 illustrates the conformance relationships between the CST and the grammar definition, showing a CST for several Delphi procedures along with the corresponding fragment of Delphi 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 `decl_section` non-terminal element corresponds to the `decl_section` tree node in Figure 5.
– A terminal element corresponds to a leaf. In Fig. 5, the `ID` terminal corresponds to the `ID` 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 Fig. 5, the `decl_section` production rule is represented by the hierarchy whose root is a `decl_section` tree node.

A query consists of a sequence of query operations, each of which includes four elements: an operator, a node type, a filter expression (optional) and an access expression (optional). Moreover, a query can be prefixed by a control statement. The EBNF expression for a query operation is:

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

We have defined three operators to query and navigate over CSTs: `/`, `//` and `///`. The `/` operator returns the immediate children of a node and is similar to dot-notation (e.g., in



```
PROCEDURE print();
  var msg : string;
BEGIN
   ...
   msg = ...
   ...
END;

PROCEDURE exec();
   ...
END;

PROCEDURE exit();
   ...
END;
```

```
program:
  ('program' ident)? programBlock '.'
;

programBlock:
  (usesClause)? block
;

block
  : (declSection)* (exportsStmt)*
     compoundStmt (exportsStmt)*
  ;

declSection
  : varDeclaration
  | procFuncDeclaration
  | ...
  ;

procFuncDeclaration
  : 'function' designator (formalParam)? ':' type ';' block ';'
  | 'procedure' designator (formalParam)? ':' block ';'
  ;

varDeclaration
  : designator ':' type
  | ...
  ;

designator
  : ID
  ;
```

```
program
└ program_block
   └ block
      ┌decl_section
      │  └procFuncDelaration
      │     ├designator
      │     │ └ID
      │     │   └'print'
      │     └block
      │       └...
      ├decl_section
      │  └procFuncDelaration
      │     ├designator
      │     │ └ID
      │     │   └'exec'
      │     └block
      │       └...
      └decl_section
         └procFuncDelaration
            ├designator
            │ └ID
            │   └'exit'
            └block
              └...
```

Fig. 5 CST for an excerpt of the Delphi grammar

OCL). The // and /// operators permit the traversal of all the child nodes (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 `mapCallFunction` rule defined in Sect. 5.3 will illustrate 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 with one and only one query operation of the sequence of operations forming a query.

For instance, to extract all the Delphi variable declarations defined in every procedure of the Delphi CST shown in Fig. 5, the following query could be expressed as `/program//#varDeclaration`. The same query expressed in OCL is shown in Fig. 6. 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

```
Locals(p : program) : Sequence(varDeclaration)
post result =
  if(p.programBlock.block.declSection = oclIsUndefined())
  then
    Sequence {}
  else
   p.programBlock.block.declSection->
     select (pd | ds.oclIsKindOf(procFuncDeclaration))->
     collect(e | e.block.declSection)->flatten()->
     select (vd | ds.oclIsKindOf(varDeclaration))
  endif
```

Fig. 6 OCL query for extracting all the variable declarations of every procedure of the Delphi CST shown in Fig. 5

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 `/program//#varDeclaration/designator{ID.exists} && ID.eq('print')}` will select those procedure grammar elements which include an `ID` leaf whose value is `print` in the Delphi CST shown in Fig. 5.

Finally, query operations can also include an access expression enclosed in square brackets, which is used to access to sibling nodes through indexing. For instance, the query `/program//procFuncDeclaration[0]` will select the first procedure grammar element of the CST in Fig. 5, which is the `print` procedure.

The sequence of query operations that forms a query expression can be prefixed by a control statement which is surrounded by curly braces at the beginning of the query.

```
q1 : //#varDeclaration;
q2 : {for each v in q1} //#statement/designator{ID.eq(v.ID)};
```
(a)

```
q3 : {greatest varDeclaration.Value} //procFuncDeclaration
                                      //#varDeclaration;
```
(b)

Fig. 7 Control statement examples: **a** the use of such statement to parameterize a query, **b** a filtering post-process

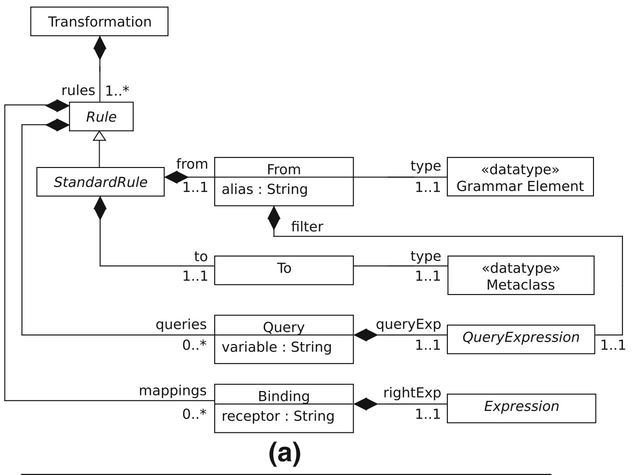
This statement allows the execution of a query to be managed by performing either a pre-process (i.e., query parameterization) or a post-process (i.e., filtering). On the one hand, the query parameterization allows a query to be executed using external information such as each of the result elements of a previous query. Figure 7a shows an example of query parameterization which includes two queries. The query `q1` collects all the variable declarations of a Delphi program and the query `q2` then uses a `for each` iterator which parameterizes the query to obtain all the assignment statements which use such variables. Notice that the control statement is in charge of launching the query `q2` as many times as result elements have `q1`, bounding each element in the `v` variable.

On the other hand, control statements can also be used to filter the result elements once a query has been executed. Figure 7b shows a query which collects all the variable declarations (i.e., `varDeclaration` elements) included in a procedure and then the `greatest` control statement then selects the variable whose `Value` leaf is the greatest. Gra2MoL includes the control statements explained previously along with the `while` and `least` statements, which allow executing a query as many times as the while condition and selecting the leaf whose value is the least, respectively. Moreover, the developer can define new statements using the extension mechanism explained in Sect. 4.5.

4 The Gra2MoL language

Gra2MoL has been designed as a text-to-model transformation language. It is a rule-based language with rules whose structure is similar to that provided in languages such as ATL or RubyTL, with two important differences: (1) the source element of a rule is a grammar element rather than a metamodel element and (2) the navigation through the source code is expressed by the query language presented for CSTs, rather than an OCL-based-language.

An excerpt of the Gra2MoL abstract syntax, expressed as a metamodel, is shown in Fig. 8a and the concrete syntax is illustrated in Fig. 8b. As can be seen, a transformation definition consists of a set of transformation rules (`Rule` element). Gra2MoL includes four types of rules: normal, copy, skip and mixin, as illustrated in Fig. 9. Normal rules are used to express



```
rule '<ruleName>'
  from <sourceGrammarElement> <alias>
  to <targetMetaclass>
  queries
    { variable : queryExpression; }
  mappings
    { receptor = literal | queryResult | expression; }
end_rule
```
(b)

Fig. 8 **a** Excerpt of the abstract syntax of Gra2MoL and **b** a skeleton of its concrete syntax

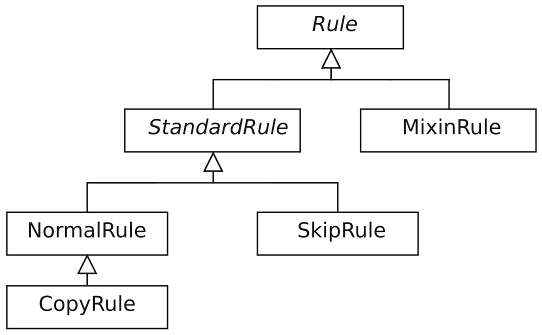


Fig. 9 Gra2MoL rule types

a mapping between a grammar element and a metamodel element, and they are therefore the rules normally required. Copy rules are normal rules but they can transform a source element more than once. On the other hand, skip and mixin rules incorporate special behaviour into Gra2MoL transformations which is explained in Sect. 4.3. Since normal and skip rules have the same syntactical structure, they are categorized as standard rules. A standard rule is composed of the following 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 (i.e., a filter) 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. It is also possible to use control or imperative structures, such as `if` statements or `new` statement, to create instances of metaclases, as occurs in Sect. 5.

4.1 Bindings and rule conformance

To express the relationship between a source grammar element and a target metamodel element, Gra2MoL incorporates the binding construct used in ATL and RubyTL. The syntax and semantics of this construct have been slightly altered to be incorporated into Gra2MoL. 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 rule evaluation is determined by a binding-based scheduling mechanism inspired by the mechanisms of ATL and RubyTL. The definitions of rule conformance and well-formed transformation stated for RubyTL in ref. [13] are applicable to Gra2MoL, with simple changes.

4.1.1 Rule conformance

A rule conforms to a binding if the type in its *from* part conforms to the type in the right-hand side of the binding and the type in its *to* part conforms to the type in the left-hand side of the binding, where the type conformance is defined as follows.

4.1.2 Type conformance

A metaclass A_m conforms to a metaclass B_m if they are the same or A_m is a subtype of B_m, whereas a node type A_n conforms to a node type B_n if they are the same.

4.1.3 Well-formed transformation definition

A transformation definition is well-formed if for each binding involving a non-primitive type as left-hand side type, there exist one or more conforming rules but there is one and only one applicable rule. This means that if two or more conforming rules exist, their filter conditions must be exclusive, since only one of them can be applied.

The application of a binding, therefore, implies that a conforming rule exists which transforms the type of the right-hand side of the binding into the type of the left-hand side of the binding.

4.2 Rule evaluation

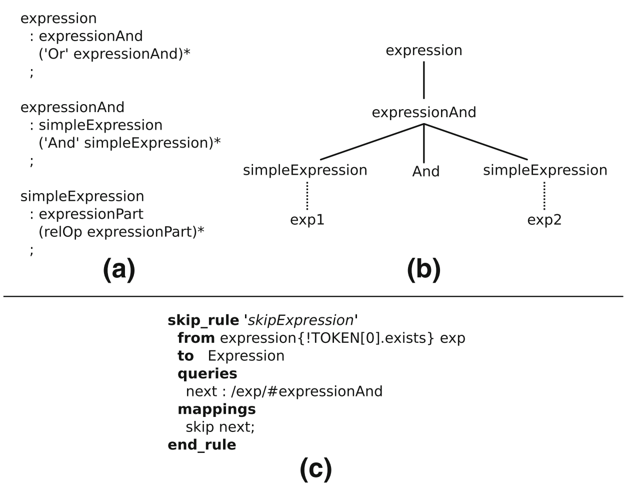
Every Gra2MoL transformation definition must have an entry point to start the transformation execution. The entry point is the first normal rule of the transformation definition and its mappings are in charge of starting the transformation execution. In a Gra2MoL transformation definition, only standard rules (i.e., normal, copy and skip rules) are eligible to be applied by a mapping, whereas mixin rules are applied when the referring rule is executed, as we explain in the following section.

When a rule is applied to a node, the filter located in the *from* part is first checked and then, if the node satisfies the filter, the rule will be executed. If it is a standard rule, an instance of the target metaclass is created. Finally, the rule bindings are executed regardless of the rule type. In the application of a binding, three situations may arise according to the nature of the right-hand side.

– If it is a literal value, the value is directly assigned to the property of the left-hand side.
– If it is a query identifier, the query is executed and a rule conforming to this binding is looked up in the transformation definition. Whenever a conforming rule is found, it is applied using the element of the right-hand side of the binding as the source grammar element.
– 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.

4.3 Skip and mixin rules

Transforming the arithmetic and logical expressions of the source code requires Gra2MoL to provide a special mechanism to deal with the grammar structures usually involved. The use of expressions in a programming language normally causes the addition of a number of grammar rules which creates a new parse tree. These grammar rules are normally defined in a chained manner in which each rule adds a new operator to the expression (see Fig. 10a). Using normal rules, the mappings between the grammar and metamodel elements are usually direct, for instance, an OR expression is normally mapped into a metamodel element which represents



```
expression
  : expressionAnd
    ('Or' expressionAnd)*
  ;

expressionAnd
  : simpleExpression
    ('And' simpleExpression)*
  ;

simpleExpression
  : expressionPart
    (relOp expressionPart)*
  ;
```

```
skip_rule 'skipExpression'
  from expression{!TOKEN[0].exists} exp
  to  Expression
  queries
    next : /exp/#expressionAnd
  mappings
    skip next;
end_rule
```

Fig. 10 **a** Grammar rules to parse both AND and OR expressions and **b** the corresponding syntax tree for the expression `expr1 And expr2`. **c** Skip rule for the `expression` grammar element

OR binary expressions. However, in some cases, parsing a grammar element does not mean creating a model element. For instance, given the grammar in Fig. 10a, if the *expression* grammar element does not contain the operator (i.e., the `OR` token in the example, see Fig. 10b), parsing an `expression` grammar element will not imply creating a metamodel element which represents an OR binary expression.

Gra2MoL, therefore, provides a special type of rule, called skip rules, which are mainly aimed at extracting models from expressions of programming languages. Skips rules allow the creation of the instance of the metaclass specified in the *to* part of the rule to be delayed until some computations to grammar elements have been performed, for instance, the existence of the `OR` token in the example. Depending on the result of such computations, the execution can be transferred to the apropriate rule using the `skip` operator in the *mappings* part. Figure 10c shows the skip rule dealing with those `expression` grammar elements which not contain the `OR` token, tranferring the execution to the rule dealing with the `expressionAnd` token (i.e., the following grammar rule dealing with expressions in the example). Note that skip rules can also be defined in a chained manner, as we illustrate in Sect. 5.

Like other model transformation languages such as RubyTL, Gra2MoL includes a type of rule, called mixin rules, which aims to provide a mechanism for reusing rules. The queries and mappings which are common to several rules can be extracted into a mixin rule. Both normal and skip rules can then import mixin rules to add the queries and mappings they define. A mixin rule has the same syntactical structure as a normal rule except that it does not have *to* part. A normal or skip rule can import a mixin rule only if the *from* part of both rules specifies the same grammar element. To

```
rule 'myNormalRule'                mixin_rule 'myMixinRule'
  from declSection dec               from declSection dec
  to Declaration                     queries
  mixin myMixinRule                    q1 : /dec//#varDeclaration;
  queries                            mappings
    q1 : ...                           vars = q1;
    ...                            end_rule
  mappings
    attr1 = ...;
    ...
end_rule
```

Fig. 11 Mixin rule use

express the importation of a mixin rule, the *mixin* part has been added to the rule structure described above. Figure 11 shows a mixin rule called `myMixinRule` and a normal rule called `myNormalRule` using it. The rule `myMixinRule` will be executed just before executing the `myNormalRule`. Note that the *from* part of both rules is the same and the `dec` variable used in the query `q1` of the mixin rule is bounded to the declaration grammar element received by the normal rule.

4.4 Implementation

The execution of a Gra2MoL transformation is split into three steps. The first step is in charge of building the CST of the source code, the second step obtains the abstract syntax model from the Gra2MoL textual definition, and finally, the third one interprets and executes the transformation definition.

Current implementation of Gra2MoL uses ANTLR grammar definitions. These definitions can be enriched with actions 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. This process also supports the use of island grammars, which is a mechanism applied when the main language contains one or more sublanguages (e.g., the Javadoc language in Java). In this case, the developer must modify the grammar rule of the main language which links to the island grammar to configure the enrichment process (more information on how to tune the grammar to support island grammars is provided in the Gra2MoL website [18]).

Gra2MoL internally uses a metamodel to generically represent CSTs of the parsed source code. This metamodel is shown in Fig. 12. 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.

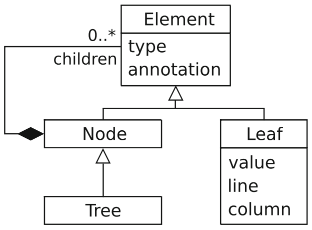


Fig. 12 CST metamodel

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 Sect. 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 Sect. 3.

The execution process of a Gra2MoL transformation is shown in Fig. 13. Figure reffig:10a shows the pre-processing step T to enrich the ANTLR grammar while Fig. 13b shows the step to obtain the abstract syntax model from the textual definition. The latter step is actually a kind of bootstrap process (i.e., Gra2MoL is used to extract a model from the Gra2MoL transformation definition) which has four inputs: the Gra2MoL concrete syntax definition, which is defined by the grammar of the language (G_{Gra2MoL}); the Gra2MoL abstract syntax (MM_{Gra2MoL}); the transformation definition ($G2MM$) and the text input which conforms to the concrete syntax (i.e., the transformation definition of the main process). The result of the bootstrap process is the abstract syntax model (M_{Gra2MoL}), which is later used by the Gra2MoL Engine. The bootstrap process allowed us to implement the DSL without the need to use other DSL definition tools, thus illustrating that our approach might also be used to implement textual DSLs.

Figure 13c shows the Gra2MoL engine, which receives the abstract syntax model, the resulting enriched parser from the pre-processing step and the source code to be transformed. The artifacts generated by the Gra2MoL engine are the model conforming to the target metamodel and a trace model containing the information concerning which target elements have been created, from which source grammar elements and by which rule.

Note that Fig. 13c is same as Fig. 1, except that a parser 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 Fig. 12. Since the CST can become huge when extracting a large number of source files, the Gra2MoL transformation process can be configured to store this tree into a model respositoy (CDO [37] and Morsa [39] model repositories are currently supported), which allows big models to be managed efficiently, thus improving performance (more information about how to set up the process is provided in the Gra2MoL website [18]).

Gra2MoL is distributed as a plugin for the Eclipse IDE which can be downloaded from the Gra2MoL website [18]. The plugin incorporates an editor to manage transformation definitions, which include some assistance mechanisms such as syntax highlighting, auto-completion or code folding. Figure 14 shows a screenshot of the Gra2Mol editor. A launcher to execute transformations from Eclipse is also provided.

4.5 Extension mechanism

Gra2MoL offers extension points to extend some language capabilities. There are two main extension points: the

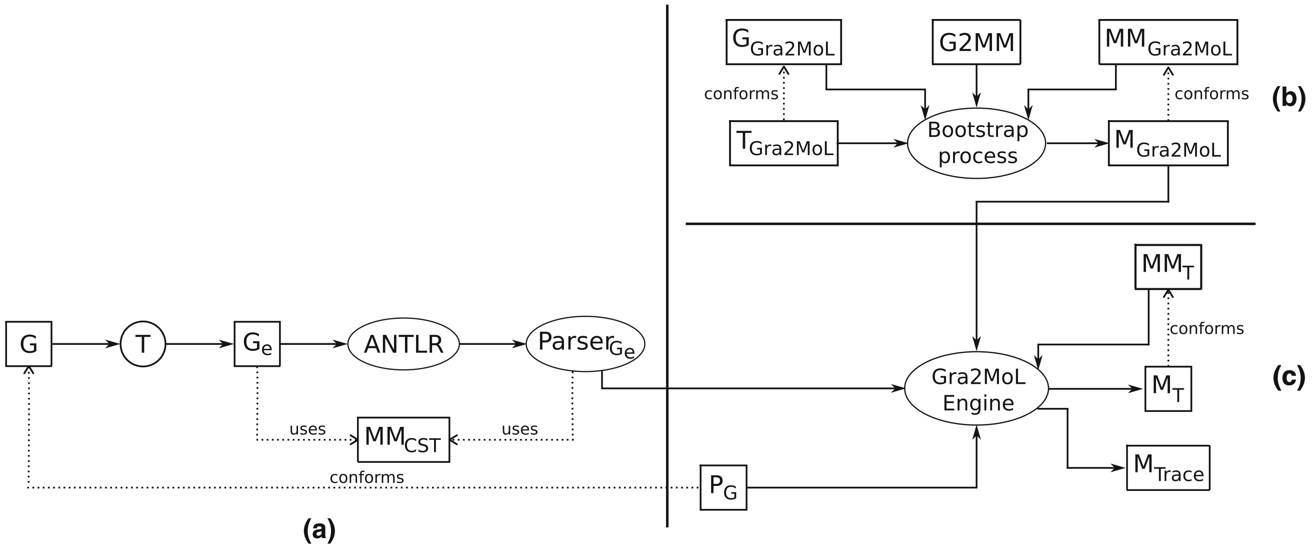
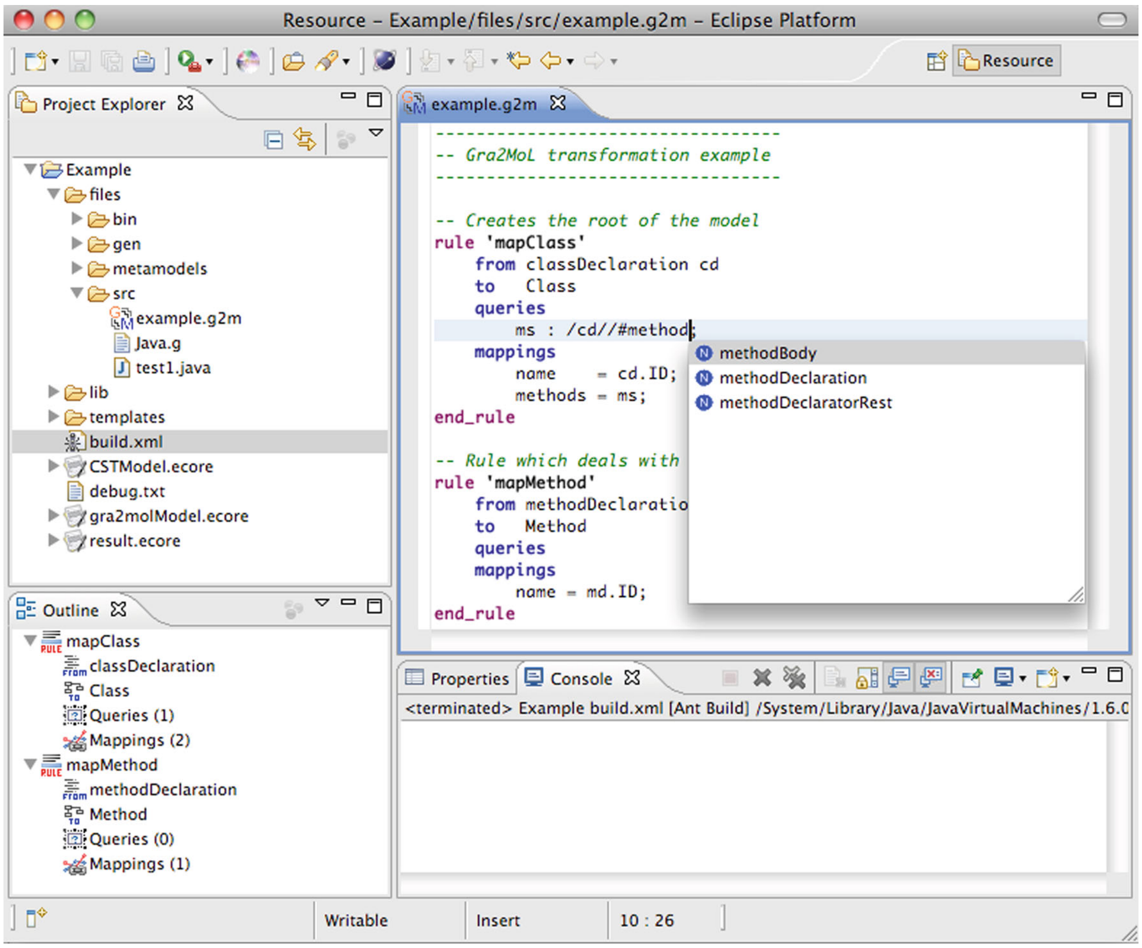


Fig. 13 Gra2MoL transformation process



```
-----------------------------------
-- Gra2MoL transformation example
-----------------------------------

-- Creates the root of the model
rule 'mapClass'
    from classDeclaration cd
    to   Class
    queries
        ms : /cd//#method
    mappings
        name    = cd.ID;
        methods = ms;
end_rule

-- Rule which deals with
rule 'mapMethod'
    from methodDeclaratio
    to   Method
    queries
    mappings
        name = md.ID;
end_rule
```

```
<terminated> Example build.xml [Ant Build] /System/Library/Java/JavaVirtualMachines/1.6.0
```

Fig. 14 Gra2MoL Eclipse plugin

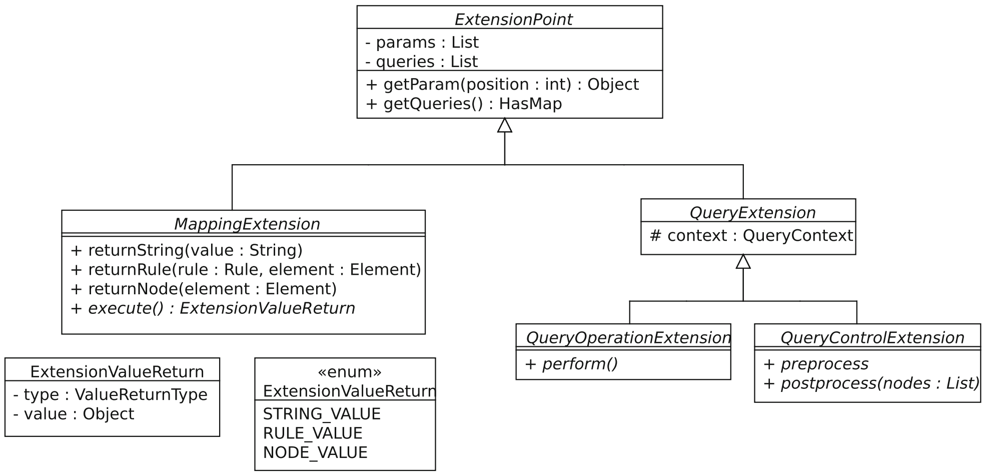


Fig. 15 Extending the Gra2MoL language. Classes composing the extension framework

mapping operators (i.e., operators to be used in the right-hand side of a binding) and the query language. Such extensions can be implemented in and incorporated into the transformation engine using the extension framework provided, which is shown in Fig. 15. Since Gra2MoL has been developed in Java, implementing a new extension is achieved by class inheritance.

When extending Gra2MoL with a new mapping operator, a subclass of the `MappingExtension` abstract class (located in left-hand part of Fig. 15) must be created. This

```
rule 'extensionExample'
  from varDeclaration vd
  to ValuedElement
  queries
  mappings
    value = ext toUpperCase(vd.VALUE);
end_rule
```

(a)

```
public class UpperCaseExtension extends MappingExtension {
  @Override
  public ExtensionValueReturn execute() {
    String value = (String) getParam(0);
    return value.toUpperCase()
  }
  public String[] getKeywords() {
    return new String[] { "toUpperCase" };
  }
}
```

(b)

Fig. 16 New mapping operator example. **a** The use of the ext keyword to call to the new mapping operator. **b** The implementation of the mapping operator

subclass must implement the abstract method `execute`, which performs the specific behavior of this operator, and optionally incorporates a method called `getKeywords` to specify the keywords which identify the new operator (if it is not provided, the keyword must be established by a property file). The `MappingExtension` abstract class also includes several methods to help the developer to build the result of the mapping operator, which can be a string value (`returnString` method), a rule (`returnRule` method) or a node (`returnNode` method). Moreover, `Mapping Extension` inherits from the common root of the hierarchy, the `Extension Point` abstract class, which allows the parameters of the operator (`getParam` method) or the queries included in the rule (`getQueries` method) to be accessed. Gra2MoL supports calls to new mapping operators using the `ext` keyword in the mappings part of a rule. A binding which uses a new operator will, thus, use the `ext` keyword followed by the keyword of the new operator and optionally a list of parameters. For instance, Fig. 16a shows a rule which uses the extension mechanism in the mappings part and calls to `toUpperCase` operator, which receives the `VALUE` string and transforms it to upper case. Figure 16b shows the implementation of the class inheriting from `MappingExtension`.

The query language can also be extended to incorporate new query control statements and operators for filter expressions. They can be added to the language by extending the `QueryControlExtension` and `QueryOperation Extension` abstract classes shown in the right-hand part of the Fig. 15, respectively. Both abstract classes inherit from `QueryExtension`, which allows accessing to the context of the queries part (i.e., accessing to the results of other queries), and in turns, inherits from `ExtensionPoint`.

```
q1 : { ext removeDuplicates } //#varDeclaration;
```

(a)

```
public class TestControlExtension extends QueryControlExtension {
  public void preprocess() { }
  public List<Element> postprocess(List<Element> nodes) {
    List<Element> resultList = removeDuplicates(nodes);
    return resultList;
  }
  private List<Element> removeDuplicates(List<Element> nodes) {
    ...
  }
  public static String[] keywords() {
    return new String[] { "removeDuplicates" };
  }
}
```

(b)

Fig. 17 New query control statement example. **a** The query using the query control statement. **b** An excerpt of the implementation of such statement

In the same way as the mapping operators, the query language extensions can also incorporate a method called `getKeywords` to specify the keywords which identify them.

With regard to new query control statements, the subclass inheriting from the `QueryControlExtension` abstract class must implement both the `preprocess` and the `postprocess` methods, which allow developers to manage the query execution. The `preprocess` method is called before the query execution whereas the `postprocess` method is called after the query execution and receives the list of result nodes. The new query control operators can be called using the `ext` keyword in the control part of a query. For instance, Fig. 17a shows a query control statement called `removeDuplicates` which remove variables whose `VALUE` leaf is the same once the query has been executed. Figure 17b shows an excerpt of the implementation of the corresponding class. Note that it is only necessary to implement the `postprocess` method.

On the other hand, when adding new query operators, the subclass inheriting from `QueryOperationExtension` must implement the `perform` method. This method is applied to the leaf of the node to which the operator is applied and returns a boolean value indicating whether the node satisfies the operator. For instance, Fig. 18a shows a query using an operator called `isSurroundedBy` which checks whether the `VALUE` leaf of the node is surrounded by a character given as parameter. Figure 18b shows an excerpt of the corresponding class implementing the query operator.

5 Example

Delphi is a programming language which is a dialect of Object Pascal. The language has been extensively used to develop business applications, especially in RAD solutions. However, there are a number of applications developed in old versions of Delphi which require adaption or modernization

```
q1 : //#var_decl{VALUE.isSurroundedBy("<");
```

(a)

```
public class testQueryOperation extends QueryOperationExtension {
  ...
  public boolean perform() {
   ExpressionElement element = filter.getElement();
   Leaf leaf = node.getLeaf(element.getName(), element.getPosition());
   return (leaf != null && leaf.isSurroundedBy(getParam(0))) ? true : false;
 }
 private boolean isSourrondedBy(String char) {
   ...
 }
 public static String[] keywords() {
   return new String[] { "isSourrondedBy" };
 }
}
```

(b)

Fig. 18 A new query operator example. **a** The query using the new query operator. **b** An excerpt of the implementation of such operator

(e.g., supporting new language versions or migrating to other platforms). Gra2MoL has been used within the context of a project to migrate Delphi applications to Java platform, to extract models from Delphi source code. This project uses the ASTM to represent the source code of the software system. Once the ASTM models are obtained, MDD techniques (i.e., model-to-model and model-to-text transformations) are applied to obtain the migrated system. In this section, we describe how the Gra2MoL transformation definition was implemented, since the rest of the migration process is not within the scope of this article. In particular, this example covers the transformation of a subset of Delphi statements.

Figures 19 and 20, respectively, show the parts of the Delphi grammar and the ASTM metamodel considered in this example. Both the grammar and the metamodel are explained as follows, and we then go on to describe the Gra2MoL transformation rules for this example.

5.1 The Delphi grammar

The grammar includes the rules needed to parse a subset of Delphi statements. Note that some of them have been simplified or reduced for the sake of simplicity. It therefore includes the `block` grammar rule representing Delphi blocks composed of an optional declaration section (`declSection` grammar rule) and a set of statements (`compoundStmt` grammar rule), which can be surrounded (i.e., including before and/or after) by export statements (`exportsStmt` grammar rule). The `declSection` grammar rule derives into a `varDeclaration` and `procFuncDeclaration` rules, which allows a variable and either a procedure or a function to be declared. Although these declaration grammar rules are used in the example to illustrate implicit references in the code, they have been greatly simplified for the sake of conciseness. The `compoundStmt` grammar rule refers to the `stmtList` rule, which in turn refers to the `statement` grammar rule. The statements considered in this example are

```
block
  : (declSection)* (exportsStmt)*
    compoundStmt (exportsStmt)*
  ;

declSection
  : varDeclaration
  | procFuncDeclaration
  | ...
  ;

varDeclaration
  : designator ':' type
  | ...
  ;

procFuncDeclaration
  : 'function' designator (formalParam)? ':' type ';' block ';'
  | 'procedure' designator (formalParam)? ':' block ';'
  ;

compoundStmt
  : 'begin' stmtList 'end'
  ;

stmtList
  : (statement ';')*
  ;

statement
  | designator ':=' expression
  | designator ('(' param ')')?
  | ...
  ;

param
  : expression (',' param)?
  ;
designator
  : ID
  ;

expression
  : expressionAnd ('Or' expressionAnd)*
  ;

expressionAnd
  : simpleExpression ('And' simpleExpression)*
  ;

simpleExpression
  : expressionPart (relOp expressionPart)*
  ;

expressionPart
  : NUMBER
  | designator
  | ...
  ;

relOp
  : '=' | '>' | '<' | '<=' | '>=' | '<>'
  | ...
 ;
```

Fig. 19 An excerpt of the Delphi grammar used in the example

assignments and function calls (the two alternatives of the `statement` grammar rule, respectively).

The grammar also includes a subset of the grammar rules needed to parse expressions, which will be used to illustrate the use of skip rules. The `expression` grammar rule allows defining optionally an `OR` logical expression where

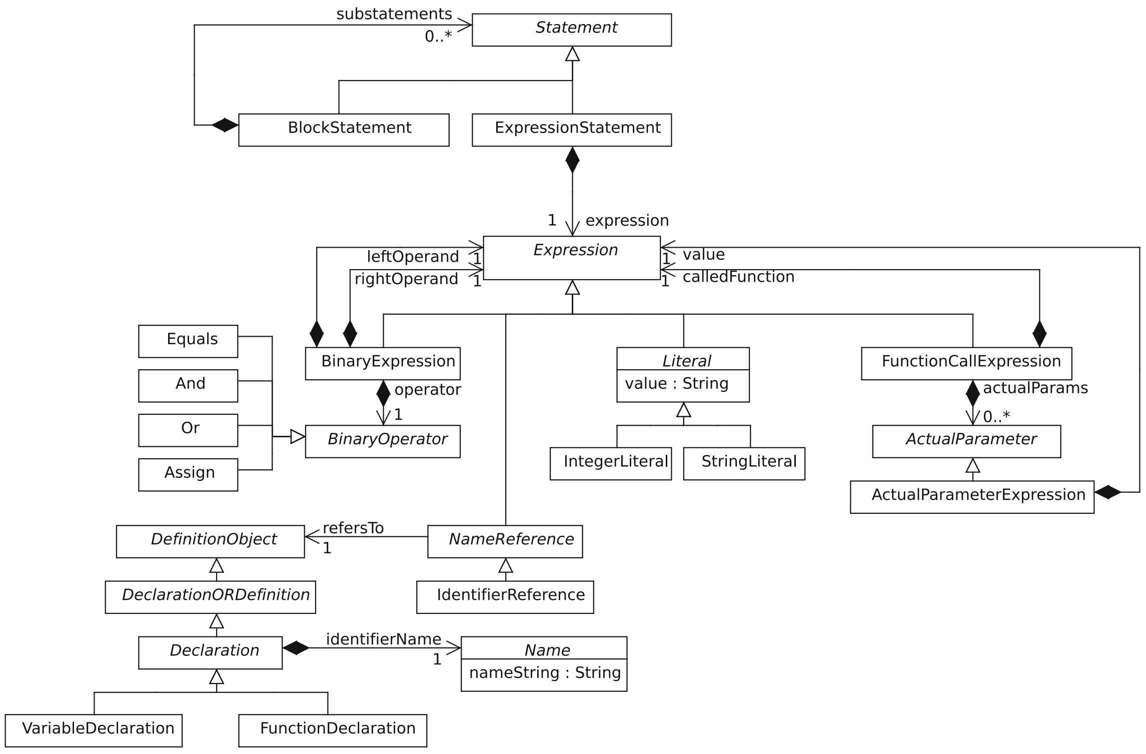


Fig. 20 An excerpt of the ASTM metamodel used in the example

each operand is represented by an `expressionAnd` grammar rule, which in turn allows an `AND` logical expression to be defined. Each operand of an `expressionAnd` grammar rule is represented by a `simple Expression` grammar rule, which uses optionally a logical operator (`relOp` grammar rule). The operands of `simple Expression` are represented by `expressionPart`, which can derive into a number (`NUMBER` token), a string value (`STRING` token) or an element reference (`designator` alternative).

5.2 The ASTM metamodel

The ASTM metamodel excerpt shown in Fig. 20 includes the elements used to represent the subset of statements and expressions considered in the grammar. In ASTM, the `Statement` hierarchy represents the statements of a programming language. The figure includes the `Block Statement` metaclass, which represents blocks of statements (`substatements` reference) and the `Expression Statement` metaclass, which represents an `expression` statement (`expression` reference) and allows assignments, references, literals and function calls to be represented. Note that ASTM models are more complex than AST models, because they incorporate metaclasses which allow cross-references between elements to be represented (i.e., `NameReference` hierarchy explained below). Thus, ASTM models are actually abstract syntax graphs.

In ASTM, the expressions of a programming language are represented by means of the elements of the `Expression` hierarchy. Figure 20 includes the `BinaryExpression` metaclass with which to represent both assignment and logical expressions; the `NameReference` metaclass, which allows elements referring to other abstract syntax tree elements to be represented (e.g., the use of a variable and its declaration or the call to a function and its declaration); the `Literal` metaclass to represent literal values; the `FunctionCallExpression` metaclass to represent function calls. The `BinaryExpression` metaclass refers to the right-hand and left-hand side of the binary expression (`rightOperand` and `leftOperand` references, respectively) along with the operator of such an expression (`operator` reference), which is represented by the metaclasses of the `BinaryOperator` hierarchy. The `Name Reference` metaclass refers to the `Definition Object` (`refersTo` reference), which is the root metaclass of every definition or declaration element in ASTM (e.g., `VariableDeclaration` and `Function Declaration` metaclasses). The `Identifier Refe rence` is subclass of `NameReference` and allows simple references to be represented (e.g., the name of either a variable or a function). The `Literal` metaclass includes the value attribute for storing the literal value and has `IntegerLiteral` and `StringLiteral` as subclasses of representing integer and string values, respectively. Finally, the `FunctionCallExpression` metaclass refers to the declaration of the called function (`calledFunction` reference) and the parameters (`actualParams` reference), which are actually represented by the metaclass `ActualParameterExpression` referring to the expression used as a parameter.

5.3 Transformation rules dealing with statements

The Gra2MoL transformation definition developed consists of 38 rules but owing to lack of space, in this section, we only present the rules involved in the example (the complete definition of the transformation can be downloaded from [18]). We first describe the rules used to transform the statements considered in this example and then the rules dealing with expressions, as they will use the skip rule type.

Figure 21 shows the set of rules used to transform blocks of statements. The `mapBlock` rule starts the transformation execution. This rule has only one binding whose right-hand side is a query identifier (`stats`) and whose left-hand side refers to the `subStatements` reference of the `BlockStatement` metaclass. The query is, therefore, executed and the rules conforming the binding are then looked up and executed for each result element. In this case, `mapCallFunction` and `mapAssignment` rules conform to the binding, but the *from* filter allows the selection of only one result element for each query, depending on the existence of the := token in the `statement` grammar element.

The `mapAssignment` rule defines the mapping between the `statement` grammar element containing the := token and the `ExpressionStatement` metaclass. This rule therefore creates an instance of the `ExpressionStatement` metaclass and its queries obtain both the left-hand side and right-hand side elements of the assignment (`lElem` and `rElem` queries, respectively). The rule includes a set of mappings to initialize the `expression` reference of the instance created. First, a new instance of `BinaryExpression` is created and assigned to this reference. Then, the operator reference is established by creating an `Assign` metaclass instance, which specifies that the `BinaryExpression` created by the rule is an assignment expression. The last two mappings are bindings whose right-hand side is a query identifier (`lElem` and `rElem`, respectively) and left-hand side is a reference (`leftOperand` and `rightOperand`, respectively). Note that the last three mappings use the dot notation in the left-hand part of the binding to access the properties of the model element referred by the `expression` property. The mapping using the `lElem` query identifier will apply the rule `locateFrom Designator`, since the query obtains the `designator` grammar element from the left-hand side of the assignment code statement and it is the only one that conforms to the binding. On the other hand, the `rElem` query obtains the `expression` grammar element of the assignment code statement and the mapping result will, therefore, apply the rule that deals with expressions, which is explained in Sect. 5.4.

The `locateFromDesignator` rule defines the mapping between the `designator` grammar element and the `IdentifierReference` metaclass. The rule therefore creates an instance of the `IdentifierReference`

```
rule 'mapBlock'
  from block b
  to    astm::gastm::BlockStatement
  queries
   stats : /b/compoundStmt//#statement;
  mappings
   subStatements = stats;
end_rule

rule 'mapAssignment'
  from statement{TOKEN[0].eq(":="} st
  to    astm::gastm::ExpressionStatement
  queries
   lElem : /st/#designator;
   rElem : /st/#expression;
  mappings
   expression                = new astm::gastm::BinaryExpression;
   expression.operator       = new astm::gastm::Assign;
   expression.leftOperand    = lElem;
   expression.rightOperand = rElem;
end_rule

rule 'mapCallFunction'
  from statement{!TOKEN[0].eq(":=")} st
  to    astm::gastm::ExpressionStatement
  queries
   dElem : /st/#designator;
   eElem : /st///#expression;
  mappings
   expression                  = new astm::gastm::FunctionCallExpression;
   expression.calledFunction  = dElem;
   expression.actualParams    = eElem;
end_rule
```

```
rule 'locateFromDesignator'
  from designator d
  to    astm::gastm::IdentifierReference
  queries
   varloc : //#varDeclaration//designator{ID.eq(d.ID)};
   metloc : //#procFuncDeclaration//designator{ID.eq(d.ID)};
  mappings
   if(metloc.hasResults) then
    refersTo = metloc;
   else
    refersTo = varloc;
   end_if
end_rule

rule 'mapVariableDeclaration'
  from  varDeclaration varDecl
  to     astm::gastm::VariableDefinition
   ...
end_rule

rule 'mapProcFuncDeclaration'
  from  procFuncDeclaration pfDecl
  to     astm::gastm::FunctionDefinition
   ...
end_rule

rule 'mapParameter'
  from  expression exp
  to     astm::gastm::ActualParameterExpression
  queries
  mappings
   value = exp;
end_rule
```

Fig. 21 Transformation rules dealing with statements

metaclass and its purpose is to locate the referred element through the source code. The queries of this rule thus traverse the CST to locate either the variable or the referred function/procedure declaration (varloc and metloc query identifiers, respectively). These queries use the // operator to find the designator grammar element, which specifies the identifier of either the variable or the function/procedure, facilitating the definition of the traversal of the syntax tree. Note that such a reference is actually a cross-reference between elements of the syntax tree and how easy is to resolve it using the Gra2MoL query language. It is also important to note that the reference format could involve defining some query extension (e.g., if the reference involves dealing with particular scopes). In the mappings section, an if statement checks whether either a variable or function/procedure has been found (i.e., they have result elements) and establishes the refersTo reference. If a function/procedure has been found, the binding involving the metloc query identifier will be applied and the mapProc FuncDeclaration rule will then be executed, which will initialize the refersTo reference to the instance of the FunctionDeclaration metaclass. On the other hand, if a variable has been found, the binding involving the varloc query identifier will be applied, and then, the mapVariableDeclaration rule will be executed, which will initialize the refersTo reference to the instance of the VariableDeclaration metaclass created by this rule. Since the example only covers the transformation rules dealing with some statements, these two rules are not shown in their entirely.

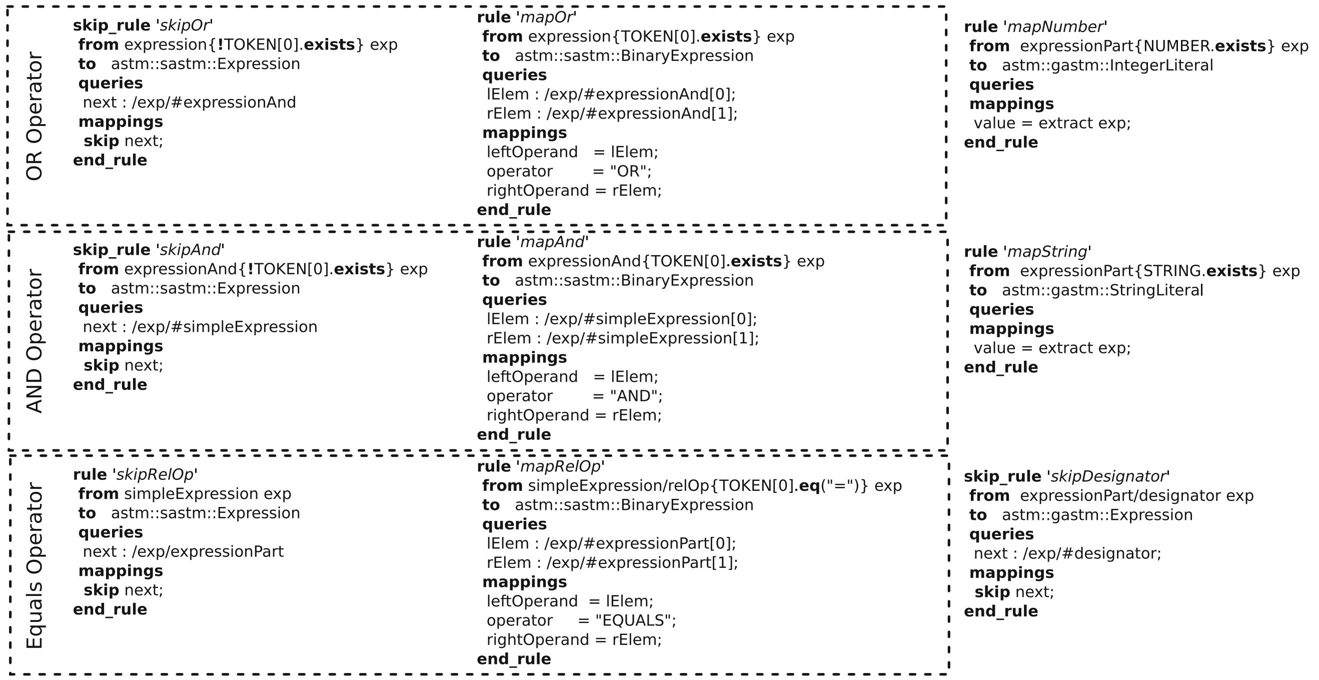
The mapCallFunction rule defines the mapping between the statement grammar element, which does not contain the := token, and the ExpressionStatement metaclass. Like the mapAssignment rule, this rule also creates an instance of ExpressionStatement metaclass, but the expression property must refer to an instance of FunctionCallExpression. The queries contained in this rule obtain the name of the called function/procedure (dElem query) and the set of parameters (eElem query). The last query illustrates the meaning of the /// operator. Since the param grammar rule is defined recursively, the /// allows the CST to be traversed to retrieve every param node. This rule includes a set of mappings to initialize the expression reference of the metaclass created. First, a new instance of FunctionCallExpression is created and assigned to this reference. The next mapping is a binding whose right-hand side is a query identifier (dElem) and left-hand side is the calledFuncion reference. The query is applied and a rule conforming this binding will be executed. In this case, the only rule that can be executed is the locateFromDesignator rule explained above. The last mapping of the mapCallFunction is also a binding whose right-hand side is a query identifier (eElem) and left-hand side is the actualParams reference. In this case, once the query is executed, the mapParameter rule is applied for each query result.

The mapParameter rule defines the mapping between the expression grammar element and the Actual ParameterExpression metaclass. The rule therefore creates an instance of Actual ParameterExpression metaclass and contains only one binding whose right-hand side is the grammar element received by the rule and the left-hand side is the value reference. Since the application of this binding will execute the rules that deal with expressions, it is explained below.

5.4 Transformation rules dealing with expressions

When defining Gra2MoL transformation rules for expression grammar rules, the pattern to be used is the following: for each expression grammar rule, two transformation rules must be added. The former is a skip rule dealing with the grammar element that does not contain the operator token and the latter is a normal rule dealing with the grammar element that contains it. The new skip rule must transfer the execution to the next grammar element dealing with expressions, whereas the new normal rule must transform the current grammar element. For instance, skipOr and mapOr are the rules which deal with the expression grammar element (see Fig. 22). The skipOr rule is a skip rule which transfers the transformation execution to the expressionAnd grammar element whether the expression grammar element does not have the OR token. On the other hand, the mapOr rule is a normal rule which deals with the expression grammar element containing the OR token.

Thus, when an expression grammar element is being evaluated (e.g., the last bindings of mapAssignments and mapParameter), either the skipOr or the mapOr could be executed depending on the existence of the OR token. If it does not exist, the skipOr rule is executed so that the next query locates the next grammar element dealing with expressions (i.e., the expressionAnd element) and executes the skip next statement, which transfers the execution to the rule whose *from* part conforms to expressionAnd and whose *to* part conforms to Expression metaclass. In this case, the candidate rules are skipAnd and mapAnd. On the other hand, if the OR token exists, the mapOr rule is executed so that an instance of the BinaryExpression metaclass is created and both its operator (operator reference) and operands (leftOperand and rightOperand references) are initialized, which causes those rules conforming to this binding to be triggered. In this case, since the type of the queries are expressionAnd grammar elements, the candidate rules also are skipAnd and mapAnd. For the sake of simplicity, the rule only deals with expressions containing two operands.



```
skip_rule 'skipOr'
 from expression{!TOKEN[0].exists} exp
 to  astm::sastm::Expression
 queries
  next : /exp/#expressionAnd
 mappings
  skip next;
end_rule
```

```
rule 'mapOr'
 from expression{TOKEN[0].exists} exp
 to  astm::sastm::BinaryExpression
 queries
  lElem : /exp/#expressionAnd[0];
  rElem : /exp/#expressionAnd[1];
 mappings
  leftOperand   = lElem;
  operator      = "OR";
  rightOperand = rElem;
end_rule
```

```
rule 'mapNumber'
 from  expressionPart{NUMBER.exists} exp
 to  astm::gastm::IntegerLiteral
 queries
 mappings
  value = extract exp;
end_rule
```

```
skip_rule 'skipAnd'
 from expressionAnd{!TOKEN[0].exists} exp
 to  astm::sastm::Expression
 queries
  next : /exp/#simpleExpression
 mappings
  skip next;
end_rule
```

```
rule 'mapAnd'
 from expressionAnd{TOKEN[0].exists} exp
 to  astm::sastm::BinaryExpression
 queries
  lElem : /exp/#simpleExpression[0];
  rElem : /exp/#simpleExpression[1];
 mappings
  leftOperand   = lElem;
  operator      = "AND";
  rightOperand = rElem;
end_rule
```

```
rule 'mapString'
 from  expressionPart{STRING.exists} exp
 to  astm::gastm::StringLiteral
 queries
 mappings
  value = extract exp;
end_rule
```

```
rule 'skipRelOp'
 from simpleExpression exp
 to  astm::sastm::Expression
 queries
  next : /exp/expressionPart
 mappings
  skip next;
end_rule
```

```
rule 'mapRelOp'
 from simpleExpression/relOp{TOKEN[0].eq("=")} exp
 to  astm::sastm::BinaryExpression
 queries
  lElem : /exp/#expressionPart[0];
  rElem : /exp/#expressionPart[1];
 mappings
  leftOperand  = lElem;
  operator     = "EQUALS";
  rightOperand = rElem;
end_rule
```

```
skip_rule 'skipDesignator'
 from  expressionPart/designator exp
 to  astm::gastm::Expression
 queries
  next : /exp/#designator;
 mappings
  skip next;
end_rule
```

Fig. 22 Transformation rules for dealing with expressions in the Delphi example

```
path : string;
procedure deleteFile (path : string, mode : integer);
begin
  ...
end

begin
  path := "debug.txt";
  deleteFile(path, 0);
end
```

Fig. 23 Delphi code example

Notice that the pattern is repeated in the `skipAnd` and `mapAnd` rules for the `expressionAnd` grammar element as well as in the `skipRelOp` and `mapRelOp` rules for the `simple Expression` grammar element. The `mapNumber`, `mapString` and `skipDesignator` rules are the end of the expression transformation. The `mapNumber` and `mapString` rules create an `Integer Literal` or `StringLiteral` that stores the value of the expression element in the `value` attribute. On the other hand, the `skipDesignator` rule transfers the execution to the rule which deals with the `designator` grammar element (i.e., the `locateFromDesignator` rule described before). Note that this use of skip rules differs slightly from the use explained previously. In this case, it is used to transfer the execution to the rule which is in charge of transforming a particular grammar element (i.e., `designator` grammar element in this case), thus allowing developers to control the transformation execution flow.

Figure 23 shows a Delphi code snippet and Fig. 24 shows the model created by means of applying the transformation rules described previously. Note that the transformation starts dealing with the boxed source code.

6 Conclusions and future work

The description about Gra2MoL in this paper has been focused on its usefulness in extracting models from GPL code. However, this DSL is actually a text-to-model transformation which can be used to extract models from any code conforming to a grammar. To the best of our knowledge, Gra2MoL is the first approach for the definition of a text-to-model transformation language.

Figure 25 shows the main Gra2MoL features according to the feature diagram proposed in [38] as a framework for the classification of model transformation languages. Gra2MoL is a unidirectional language, whose source domain is the grammar realm and whose target domain is the MDE realm. A Gra2MoL transformation definition consists of rules which transform grammar elements into model elements by manipulating the CST of the source code. Rules are resolved implicitily and in a deterministic manner, although the developer can alter the rule scheduling using skip rules. The language also incorporates mixin rules as a reuse mechanism, in addition to copy rules which allow a source element to be transformed more than once. With regard to the trace information, the Gra2MoL engine creates a separate trace model automatically. Moreover, a powerful language has been defined to navigate and query a CST in a structure-shy manner. The language has been applied a several case studies (downloadable

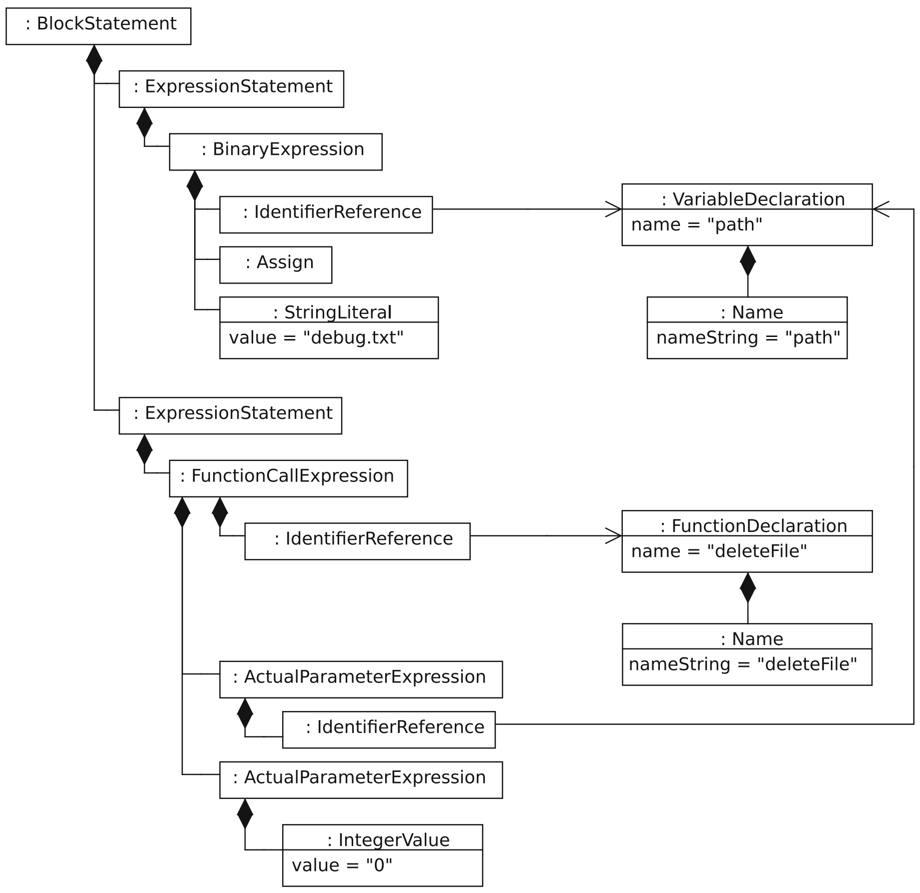


Fig. 24 ASTM model obtained by means of applying the transformation rules to the Delphi code shown in Fig. 23

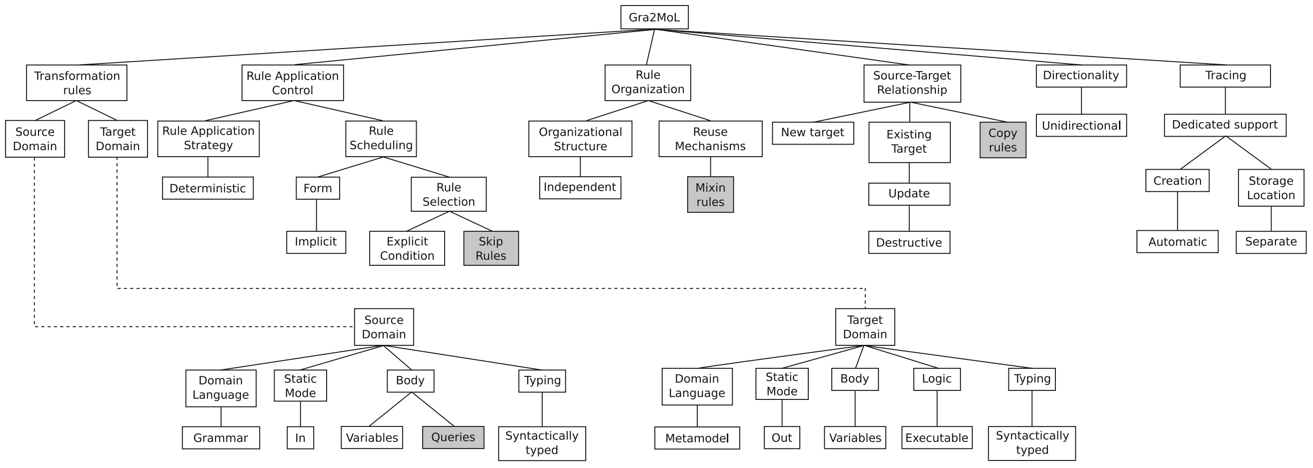


Fig. 25 Feature diagram showing the features of Gra2MoL according to [38]. Gra2MoL specific features are depicted as filled boxes

from [18]), thus allowing us to identify new functionalities to improve its expressiveness, usability and performance.

With regard to future work, we are working on scalability issues such as analyzing the performance impact of managing a CST either in memory or in a model repository. We are additionally studying the incorporation of a phasing mechanism to allow transformation definitions to be organized and modularity to be promoted. We also plan to incorporate a trace query mechanism to improve the transformation control. Finally, we are working on supporting another parser generators to increase the number of existing grammars that can be reused.

Acknowledgments This work has been supported by the Spanish government (TIN2009-11555 project) and the Fundación Séneca of the Regional Government of Murcia (15389/PI/10 grant).

References

1. Heckel, R., Correia, R., Matos, C., El-Ramly, M., Koutsoukos, G., Andrade, L.: Architectural transformations: from legacy to three-tier and services. In: Mens, T., Demeyer, S. (eds.) Software Evolution, p. 170. Springer, Heidelberg (2008)
2. Cánovas Izquierdo, J.L., García Molina, J.: An architecture-driven modernization tool for calculating metrics. IEEE Softw. **27**, 37–43 (2010)
3. Andrade, L.F., Gouveia, J., Antunes, M., El-Ramly, M., Koutsoukos, G.: Forms2Net - migrating oracle forms to Microsoft. NET. In: Generative and Transformational Techniques in Software Engineering, pp. 261–277 (2006)
4. Reus, T., Geers, H., Deursen, A.: Harvesting software systems for MDA-based reengineering. In: European Conference on Model Driven Architecture: Foundations and Applications, LNCS, vol. 4066, pp. 213–225 (2006)
5. ADM initiative website. http://adm.omg.org. Accessed 6 March 2012
6. ADM Task Force: Architecture-driven modernization scenarios. OMG, USA (2006)
7. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: An initial appraisal. In: Cooperative Information Systems, DOA'2002 Federated Conferences, Industrial track (2002)
8. Xtext project. http://www.eclipse.org/Xtext. Accessed 6 March 2012
9. EMFText project. http://emftext.org. Accessed 6 March 2012
10. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. **45**(3), 621–646 (2006)
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comp. Program. **72**(1–2), 31–39 (2008)
12. Heaton, L.: Meta Object Facility (MOF) Query/View/ Transformation Specification. OMG, New York (2005)
13. Sánchez Cuadrado, J., García Molina, J., Mernárguez Tortosa, M.: RubyTL: A practical, extensible transformation language. In: European Conference on Model Driven Architecture: Foundations and Applications, LNCS, vol. 4066, pp. 158–172 (2006)
14. Mofscript project. http://www.eclipse.org/gmt/mofscript. Accessed 6 March 2012
15. Xpand website. http://wiki.eclipse.org/Xpand. Accessed 6 March 2012
16. Cánovas Izquierdo, J.L., Sánchez Cuadrado, J., García Molina, J.: Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization. In: Workshop Model Driven Software Evolution (2008)
17. Cánovas Izquierdo, J.L., García Molina, J.: A domain specific language for extracting models in software modernization. In: European Conference on Model Driven Architecture Foundations and Applications, LNCS, vol. 5562, pp. 82–97 (2009)
18. Gra2MoL website. http://modelum.es/gra2mol. Accessed 6 March 2012
19. Wijngaarden, J., Visser, E.: Program transformation mechanics: a classification of mechanisms for program transformation with a survey of existing transformation systems. Tech. Rep. UU-CS-2003-048. The Department of Information and Computing Sciences, Utrecht University, The Netherlands (2003)
20. JDT Eclipse project. http://www.eclipse.org/jdt. Accessed 6 March 2012
21. MoDisco. http://www.eclipse.org/gmt/modisco. Accessed 6 March 2012
22. GMT Eclipse project. http://www.eclipse.org/gmt. Accessed 6 March 2012
23. KDM metamodel specification. http://www.omg.org/spec/KDM. Accessed 6 March 2012
24. Fowler, M.: Domain-Specific Languages. Addison Wesley, USA (2011)
25. Scheidgen, M.: Textual Modelling Embedded into Graphical Modelling. In: European Conference on Model Driven Architecture Foundations and Applications, LNCS, vol. 4530, pp. 153–168 (2008)
26. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a dsl for the specification of textual concrete syntaxes in model engineering. In: Generative Programming and Component, Engineering, pp. 249–254 (2006)
27. JAMOPP project. http://jamopp.inf.tu-dresden.de. Accessed 6 March 2012
28. Wimmer M., Kramler, G.: Bridging grammarware and modelware. In: Satellite Events at the MoDELS 2005 Conference, pp. 159–168 (2006)
29. Kunert, A.: Semi-automatic generation of metamodels and models from grammars and programs. In: Fifth International Workshop on Graph Transformation and Visual Modeling Techniques. E. N. in Theorical Computer Science, vol. 211, pp. 111–119 (2008)
30. Prinz, A., Scheidgen, M., Tveit, M.S.: A model-based standard for SDL. In: International SDL Forum Conference on Design for Dependable Systems, pp. 1–18 (2007)
31. Stratego/XT. http://strategoxt.org. Accessed 6 March 2012
32. TXL. http://www.txl.ca. Accessed 6 March 2012
33. OCL constraint language. OMG (2006)
34. Wijngaarden, J.: Code Generation from a Domain Specific Language: M.Sc. Thesis (2003)
35. ASTM metamodel specification. http://www.omg.org/spec/ASTM Accessed 6 March 2012
36. Xpath. http://www.w3.org/TR/xpath. Accessed 6 March 2012
37. CDO project. http://www.eclipse.org/cdo. Accessed 6 March 2012
38. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Technique in the Context of the Model Driven, Architecture (2003)
39. Espinazo-Pagán, J., Sánchez Cuadrado, J., García Molina, J.: Morsa: A scalable approach for persisting and accessing large models. In: International Conference on Model Driven Engineering Languages and Systems, pp. 77–92 (2011)

Author Biographies

Javier Luis Cánovas Izquierdo received a Ph.D. in computer science from the University of Murcia, Spain in 2010 and a M.Sc. from the University of Murcia in 2006. Since October 2011, he is a post-doctoral researcher in the AtlanMod team at INRIA Rennes Bretagne Atlantique. His research interests are domain-specific languages, model-driven development and model-driven modernization.

Jesús García Molina is a full professor in the Department of Informatics and Systems at the University of Murcia (Spain), where he leads the Modelum group, an R&D group with focus on Model-Driven Engineering and close partnership with industry. His research interests include model-driven development, domain-specific languages, and model-driven modernization. He received his Ph.D. in physical chemistry from the University of Murcia.