Applying MDE tools to defining domain specific languages for model management

Gabriela Pérez [1], Jerónimo Irazábal [1,2], Claudia Pons [1,3] y Roxana Giandini [1]
[1]LIFIA, Facultad de Informática, Universidad Nacional de La Plata
[2]CONICET, Consejo Nacional de Investigaciones Científicas y Técnicas
Buenos Aires, Argentina
[3]CIC, Comisión de Investigaciones Científicas
Buenos Aires, Argentina

{gperez , jirazabal, cpons, giandini}@lifia.info.unlp.edu.ar

Abstract. In the model driven engineering (MDE), modeling languages play a central role. They range from the most generic languages such as UML, to more individual ones, called domain-specific modeling languages (DSML). These languages are used to create and manage models and must accompany them throughout their life cycle and evolution.

In this paper we propose a domain-specific language for model management, to facilitate the user's task, developed with techniques and tools used in the MDE paradigm.

Palabras clave: Model Driven Development, DSL, Domain Specific Language. Domain Specific Language for Model Management.

1. Introduction

Modeling is important in order to address the complexity of the systems during the development process and during maintenance. The Model Driven Engineering [1-3], [5, 22] proposes a software development process in which the main elements are the models. From them, engineers can accurately capture relevant aspects of a system from a given perspective and at an appropriate level of abstraction in order to automate its development.

Models can be expressed using different languages: general purpose modeling languages (GPMLs) as UML, or domain specific modeling languages (DSML) [4], such as the Business Process Modeling Notation (BPMN) for business process modeling. The DSMLs are high level languages designed for particular tasks. They allow the specification of a solution directly using problem domain concepts. As the language concepts are already used within the organization, the learning time of the language is significantly reduced. Domain experts can also understand, validate, modify and often develop programs in the DSL. DSMLs have a simpler syntax (i.e., few constructs focused to the particular domain) but their semantics is much more complex (because all the semantics of the particular domain is embedded into the language). Because they are restricted to a particular domain, the code generation is more efficient, enabling significant improvements in productivity, interoperability, maintainability and quality of the products generated. Based on experience, the use of DSMLs can simplify the development of complex software systems by providing domain-specific abstractions for modeling the system and its transformations in a precise but simple and concise way.

The DSMLs increase the expressive power but also increase the complexity. For the end user, building a model means create and connect domain specific low-level elements, which is an error-prone task. These difficulties could be eliminated providing another language to manage the domain-specific models in a more friendly way. In that case, the user will be not longer responsible for the consistently manipulation of the elements. This responsibility will be delegated to the management language by using the operations offered by it.

Having the definition of a domain specific model management language DSMML [28, 32] separated from the domain specific language will allow:

- Independent evolution: The management language and the modeling language can evolve independently. Some aspects of the domain-specific modeling language can be improved without altering the management language interface (signature operations); or even changed radically. Similarly, you might modify, or extend operations of the management language without affecting the specific modeling language.
- Multiple management languages: multiple management languages can be provided for the same specific modeling language. Different languages could be developed with different operations that reflect the responsibilities of each user. For example, a query language for basic users and another language with critical operations, such as editing and deleting for advanced users.
- Friendly interaction: For a standard domain-specific language, the management language will improve the user confidence: defining a model management language on a standard language such as BPMN, enable a more friendly interaction with the standard language.

Figure 1 shows that the definition of a specific language for model management helps users with the effective use of the language. At the top, the figure shows a user who still has some doubts regarding the use of domain specific language (DSML). At the bottom, the figure shows two users using two distinct domain specific model management languages (DSMMLs). Each user is using the appropriate language for each user type (basic or advanced). They can immediately make an effective use of the language.

The problem to be faced consists in implementing this specific management language. Currently there are very powerful frameworks for creating domain-specific languages, as can be seen in [9-11]

This paper describes a proposal to define specific model management languages and analyzes a novel way to define their semantics. Our proposal consists in using MDE tools for the implementation of these languages, improving modularity and reuse. The paper is organized as follows: section 2 introduces the main features of our proposal. Section 3 presents a general approach for defining a model management language on a well known domain, such as the domain of business processes. Section 4 shows an example. Section 5 compares our approach with other related research. And finally we present conclusions and future work.

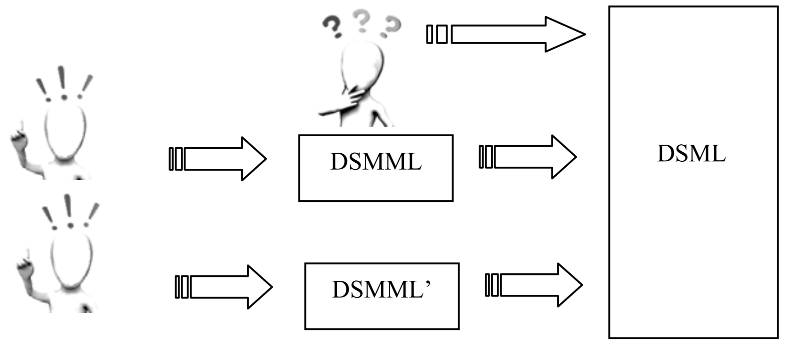

Figure 1: DSMML enable a more friendly interaction with the DSML.

2. DSMML: implementation schema

Any language consists of two main elements: a syntactic notation (syntax) which is a possibly infinite set of elements that can be used in the communication, together with their meaning (semantics). The term “syntax” refers to the notation of the language. Syntactic issues focus purely on the notational aspects of the language, completely disregarding any meaning. On the other hand, the “semantics” assigns an unambiguous meaning to each syntactically allowed phrase in the language. To be useful in the computer engineering discipline, any language must come complete with rigid rules prescribing the allowed form of a syntactically well formed program, and also with formal rules prescribing its semantics.

In programming language theory, semantics is the field concerned with the rigorous mathematical study of the meaning of languages. The formal semantics of a language is given by a mathematical structure that describes the possible computations expressed by the language. There are many approaches to formal semantics, among them the denotational semantics approach is one of the most applied. According to this approach each phrase in the language is translated into a denotation, i.e. a phrase in some other language. Denotational semantics loosely corresponds to compilation, although the "target language" is usually a mathematical formalism rather than another computer language. Formal semantics allows a clear understanding of the meaning of languages but also enables the verification of properties such as program correctness, termination, performance, equivalence between programs, etc.

Technically, a semantic definition for a language consists of two parts a semantic domain and a semantic mapping, denoted μ, from the syntax to the semantic domain.

In particular, our proposal consists in using a well known general propose transformation language as the semantic domain, such as ATL [6-7]. Then, the semantic function μ is defined by a transformation written in a model-to-text transformation language (such as MOFScript [16]). This M2T transformation takes a program written in the DSMML as input, and generates a program written in ATL as output. The language semantics is defined as described in Figure 2.

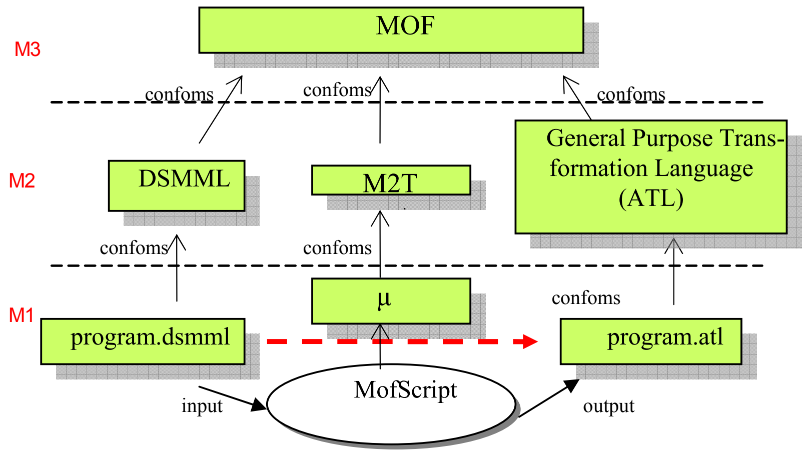

Figure 2: Transformation scenario

The advantage of this technique is that the well-known transformation language has already a well defined semantics and provides an execution environment. So, the semantics of the new language becomes formally described and it is executable. Additionally, the semantic definition is understandable and adaptable because it is expressed in terms of a well known high level language.

In the next sections we present examples of DSMMLs including the definition of their syntax and semantics following the proposed approach.

3. Business Process Model Management Language.

Business Process Management (BPM) is the methodology based on business processes. Its aim is to improve organizational performance through process management. For this, these processes must be designed, modeled, organized, documented and optimized continuously. The proposal of BPM [23] has gained considerable attention recently from both the business management and computer science communities. In this domain, the standard modeling language defined by the OMG [24-25] is Business Process Modeling Notation (BPMN) has become popular. BPMN has been developed to provide a standard notation to business users, similar to how UML has standardized the modeling concepts in the software engineering field.

3.1. Motivation for a Business Process Model Management Language

As we mentioned in the previous section, the BPMN language was developed to provide a standard notation for domain BPM concepts. These concepts, like process, activity or task were already used by business users, so the language learning time is reduced.

A BPMN model might require some modifications along its life cycle. Despite knowing the domain concepts, implementation details of this language are unknown (where the instances should be store or the consequences of deleting an item).

Therefore, to carry out these changes, you must have a detailed knowledge of the metamodel, as well as the relations between the elements. Making these changes by hand, directly on the model, threatens the model integrity, and is an error-prone task.

To preserve the model integrity, it would be desirable to count with a management language. This language must provide specific management operations for the BPMN models and to hide the implementation details. Currently there are a variety of tools to visually edit BPMN diagrams. These tools are mostly focused on the creation, editing and deletion of model elements. However, it would be useful to have more complex operations to facilitate and go together with the evolution of these models. For example, replace a business process by other process previously described is a common modification. With an appropriate management language, these tasks are much simpler. The modifications details are hidden behind the specific management operations.

3.2. BPMML: a DSMML fitting the Business Process Modeling.

In this section we illustrate the definition of a DSMML for specifying modifications on business process models. This language is named Business Process Model Management Language (BPMML). In the design process of BPMML we have considered those management operations that are frequently applied on business process models. In particular, it is useful to count with operations based on commonly used refactorings [26-27], such as the following ones.

- The SubstituteFragment operation that allows process designer to replace a fragment by another one, taking into account the relationships with the first fragment as a source or target.
- The *ExtractFragment* operation that allows extracting a fragment to generate a new process, with the aim of eliminating redundancy, taking into account the relationships with the fragment and creating similar relations with the new process.
- The ReplaceFragmentbyReference operation that replaces a complex activity by a reference.
- In addition to these operations, we have defined further operations that are useful in the domain of business process modeling, such as importing a process, swapping the position between two process (allowing the latter to occur before the first),

breaking the connection between two processes, establishing a new connection between two processes, and so on.

The following code shows the concrete syntax expressed with the EMFText plugin [33].

```
SYNTAXDEF bpmml
FOR <http://bpmml/1.0> <bpmml.genmodel>
START BPMML
RULES {
	BPMML ::= "BPMML" "open" inputModelPath['"','"']
		"{" ( managements : Management)* "}";

	ExtractGroup ::= "Extract" group[] ";";
	RenameActivity ::= "Rename" name[] "to" newName[]";";
	ReplaceSubProcess::= "Replace" oldSubProcess[] "by" new-
SubProcess[]";";
	SubstituteSubProcess ::= "Substitute" oldSubProcess[]
"by" newSubProcess[] "located in" modelPath[]";";
	ImportPool::= "Import pool" pool[] "located in" model-
Path[]";";
	ImportSubProcess::= "Import subprocess" subProcess[] "to"
targetPool[] "located in" modelPath[]";";
	DeletePool::= "Delete pool" pool[]";";
	DeleteElementsFromPool::= "Delete elements from"
pool[]";";
	SwapElements::="Swap" source[]"with" target[]";";
	CreateActivityBetween::= "Create Activity" named[] "be-
tween" source[] "and" target[]";";
	SplitFlow::= "Split Flow add" element[] "between"
source[] "and" target[]";";
	AllActivitiesFirstUpper::= "Format name to all activi-
ties" ";";
  }
```

3.3. BPMML implementation

In this section we present the implementation of our DSMML. Figure 3 explains our translational approach for the definition of the semantics of the domain specific management language. We implement such translation (or compilation) from the domain specific management language to a general purpose transformation language (i.e., ATL). The translation rules are written in the model to text transformation language MOFScript. The generated ATL program is the semantic interpretation of our DSMML.

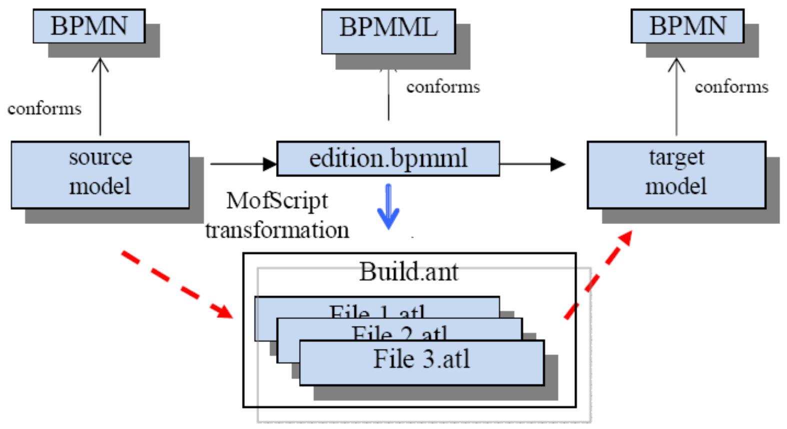

Figure 3: DSMML implementation schema using a translational approach

Due to ATL restrictions, the generation of a separate ATL file for each BPMML program statement was necessary. Then these files needed to be coordinated so that they can run properly. Apache Ant was the tool used for this purpose. Figure 4 shows the files that are generated by the MOFScript program: the sequence of atl files and the build.ant script that coordinates them.

The following code shows the MOFScript transformation, separated into two modules. For one, the main module, it will take each management operation and translate it to their respective ATL code.

```
import ("BPMML_Library.m2t");
 texttransformation BPMML_Semantic (in pmml:"http://bpmml/1.0"){
  bpmml.BPMML::main () {
   self.operations->forEach(operation:bpmml.Operation)
      operation.createATLFile(operation.
                             getFilename(number));
  }
 self.createAntTask(self.inputModelPath, fileList, antLaunchs);
}
```

The following code shows the second module, which translates each operation to ATL code.

```
  texttransformation SemanticaBPMNTL_Library (in
bpmml:"http://bpmml/1.0")
{
bpmml.RenameActivity::printCode(){
println("-- @nsURI BPMN=http://stp.eclipse.org/bpmn");
println("module RenameActivity;");
println("create OUT : BPMN refining IN : BPMN; \n");
….
```

```
bpmml.SplitFlow::printCode(){}
bpmml.SubstituteSubProcess::printCode(){…}
…
```

The following example shows the ATL code that was generated from the Activity Rename operation, which renames an activity called 'Start' with the name 'Experience an unexpected behavior'. It uses the refinement mechanism to write code only for items that will be affected by the transformation, while the rest of the model remains unchanged.

```
  -- @nsURI BPMN=http://stp.eclipse.org/bpmn
  module RenameActivity;
  create OUT : BPMN refining IN : BPMN;

   helper def: activityToRename: BPMN!Activity =
     BPMN!Activity.allInstancesFrom('IN')-> select(a | a.name =
'Start') ->first();

   helper def: notExistsActivityNamed: Boolean =
         BPMN!Activity.allInstancesFrom('IN')->    select(a     |
       a.name = 'Experience an unexpected behavior')
       ->first().oclIsUndefined();

   rule Activity2Activity {
         from activity: BPMN!Activity in IN (activity =
                thisModule.activityToRename     and     thisModu-
              le.notExistsActivityNamed)
       to activityOut: BPMN!Activity (
              name <- 'Experience an unexpected behavior',
              graph <- activity.graph
       )
   }
  }
```

4. An example

In this section we illustrate the applicability of the BPMML management language through an example. Figure 4 shows an initial sketch of the process used to report a bug in Bugzilla. Bugzilla is a web-based tool that allows developers to find bugs, assign bugs to the appropriate developer, maintain progress information in a bug fixing, etc.. This example was presented at EclipseCon 2008.

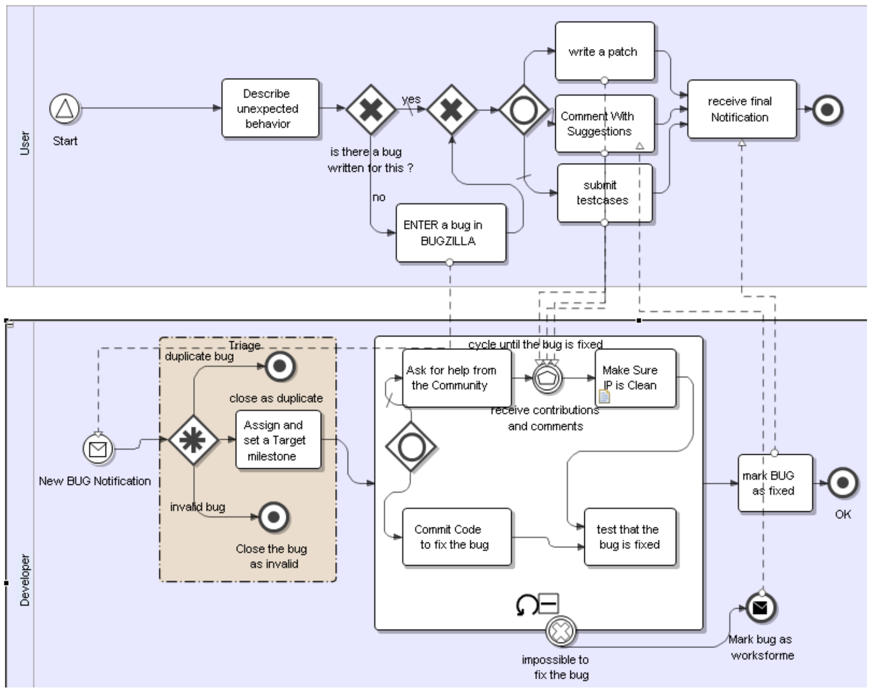
User
Start
Describe unexpected behavior
is there a bug written for this ?
yes
no
ENTER a bug in BUGZILLA
write a patch
Comment With Suggestions
submit testcases
receive final Notification
Developer
New BUG Notification
Triage
duplicate bug
close as duplicate
Assign and set a Target milestone
invalid bug
Close the bug as invalid
cycle until the bug is fixed
Ask for help from the Community
receive contributions and comments
Make Sure IP is Clean
Commit Code to fix the bug
test that the bug is fixed
impossible to fix the bug
mark BUG as fixed
OK
Mark bug as worksforme

Figure 4. First draft of Bugzilla process model.

Looking carefully, we realize that we can make some modifications to this process, including:

- To change the name of the input event by a more meaningful name, e.g. 'An Unexpected Behavior Experience'.

- To add a research activity to the process, with the aim of ensuring that this task is done properly. As this is a common process, already developed by other processes, we can import it and insert it between the input event and the task that describes the unexpected behavior.
- In addition, we could normalize the names of activities, so that all are written in lowercase and begin with the first letter capitalized.

These changes are specified using the following code written in BPMML

```
BPMML open "bugzilla.bpmn" {
 Rename "Start" to "Experience an unexpected behavior";
 Import subprocess "Investigate" to "User" located in "other-
Project.bpmn";
```

```
   Split Flow add "Investigate" between "Experience an unex-
pected behavior" and "Describe unexpected behavior";
   Format name to all activities;
  }
```

Figure 4 presented before shows the model which will be the input for applying these modifications. Then figure 5 shows the output model, after applying the management program.

The coordinated ATL files were applied using the ant file created for that purpose. The model output can be displayed in a graphical editor for bpmn.

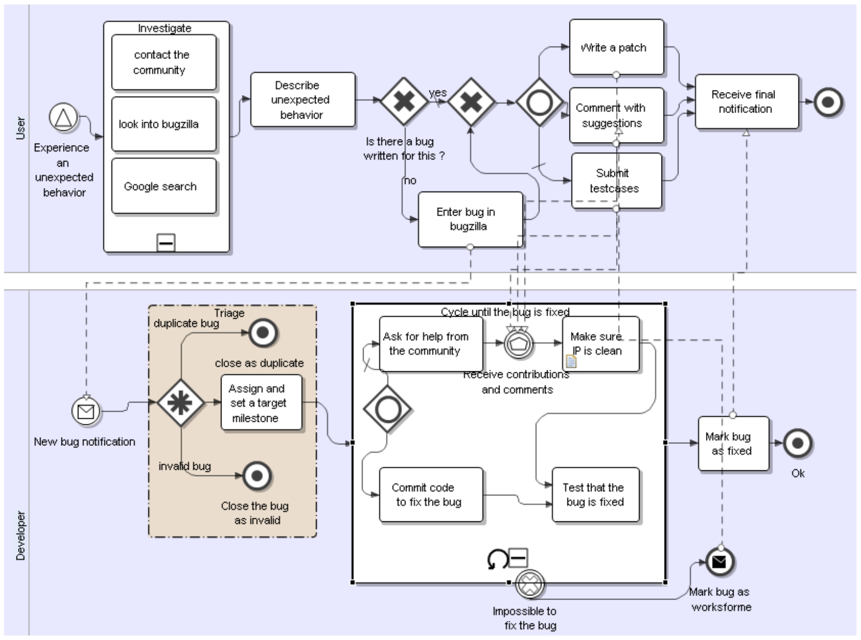
User
Developer
Experience an unexpected behavior
Investigate
contact the community
look into bugzilla
Google search
Describe unexpected behavior
yes
no
Is there a bug written for this ?
Enter bug in bugzilla
Write a patch
Comment with suggestions
Submit testcases
Receive final notification
New bug notification
Triage
duplicate bug
close as duplicate
Assign and set a target milestone
invalid bug
Close the bug as invalid
Cycle until the bug is fixed
Ask for help from the community
Receive contributions and comments
Make sure IP is clean
Commit code to fix the bug
Test that the bug is fixed
Impossible to fix the bug
Mark bug as fixed
Ok
Mark bug as worksforme

Figure 5. Improved Bugzilla process model.

In this example it is evident that the specific language facilitated the management of business model elements. Such management is clearly harder if carried out using an ordinary graphic editor.

5. Related work

The works that have been analyzed are linked to the creation and use of domain specific modeling languages . On one hand Kolovos [30-31] proposed Eugenia annotation language that aims to reduce the learning barrier of GMF and thus make it more accessible to users. GMF is a project that can generate graphical editors in Eclipse.

GMF needs a metamodel and based on this the definition of three models is required. The first one is for the graphical definition, called GMFGraph, which defined shapes, connectors, labels, etc. The second one is for the definition of tools, called GMFTool, in which you specify which items are visible in the palette editor for its creation. And the third one is a model that relates the metamodel elements with the tool and graphic definitions. These three models and the metamodel will be the basis to generate a fully functional graphical editor. Eugenia proposes an alternative to the construction of these three models required by GMF: making annotations directly on the metamodel file and from them creating these models automatically. This proposal is similar to ours since the new language becomes the existing language, in this case, GMF, more user-friendly. Unlike our proposal, this new language is not defined for model management but for generating the GMF required models. Our approach proposes the creation of one or more languages that will be used throughout the entire life cycle of models management.

On the other hand, Kermeta [29] is a meta-modeling language for describing the structure and behavior of models. It is designed to be fully compatible with EMOF language and it provides an actions language to specify the behavior of the models. As a difference, our proposal suggests a separation between the metamodel and specification of their behavior, giving them greater flexibility, allowing by one hand to define and use multiple management languages and by the other hand, the free evolution of both languages.

Our approach can be seen as a technique for abstraction and modularization in that each high level management (written in the DSMML) is associated with a lower level management (written in a more general purpose language), but the users do not need to be aware of the details of the low level management. In this sense, the works that propose techniques to build complex transformations by composing smaller transformation units are related to our proposal. In this category we can mention the composition technique described by A. Kleppe in [17], the Model Bus approach [18], the modeling framework for compound transformations defined by Jon Oldevik in [19] and the module superimposition technique [20], among others. In contrast to these works, our approach generates the composed transformation specification in a simpler way, without introducing any explicit composition machinery.

If we look at languages that abstract from other languages we can mention the MetaBorg language [21]. MetaBorg is a transformation-based approach for the definition of embedded textual DSLs implemented based on the Stratego framework. Similarly to our work, the MetaBorg approach defines new concepts (comparable to our notion of an abstract language) by mapping them to expansions in the host language (comparable to our notion of a concrete language). An important distinction between these works and our work is the application to the MDE field.

The AMMA framework [12] allows us to define the concrete syntax, abstract syntax, and semantics of DSLs. In [13-15] the reader can analyze a number of scenarios where the AMMA framework has been used to define the semantics of DSLs in terms of other languages or in terms of abstract state machines (ASMs). Our proposal is similar to the one of AMMA, but we present a novel alternative, where the language semantics is realized as the interpretation of the DSMML into a general purpose model to text transformation language.

6. Conclusions

In this paper we have explained the concept of domain specific languages for model management to focus on a specific domain. In contrast to well known model management languages such as EOL [8] and ATL, these languages syntax and semantics are directly related to a domain, making management programs easer to write and understand.

Having a domain-specific language for model management has the following advantages:

- It allows users to interact with domain-specific models in a more friendly way. The model modifications are less prone to errors, since the user will not be responsible for generating elements and relate them consistently. This responsibility is delegated to the model management language by using the operations offered by it. Domain experts will feel more comfortable using a management language with constructs reflecting well-known concepts.
- DSMML designer and programmer roles are separated. Programmers do not need to know the general purpose transformation language specification, as this information is encapsulated in the operations offered by the DSMML.
- The management language and the modeling language can evolve independently from each other. Some aspects of the domain-specific modeling language can be improved without altering the management language interface; or even changed radically. Similarly, you might modify, or extend operations of the management language without affecting the specific modeling language.
- For a standard domain-specific language, the management language will make users more confident in the use: defining a model management language on a standard language, such as BPMN, enable a more friendly interaction with the standard language.

Furthermore, we propose that DSMML language semantics to be defined using a general purpose transformation language. We present a proposal where a DSMML instance is not compiled into source code but it is transformed to a general purpose modeling transformation language. In the example we have used the ATL language. This provides several advantages: the language semantics is formally described, and it is executable. The semantics is understandable because it is written in a well known language; it can be easily modified by adding new transformation rules, or radically changing the target language. Although this transformation can be considered as a compiler, the skills needed to create it are lower than if we create a source code compiler.

Going beyond you would think that the language developer is expert only in the domain and not in the general purpose transformation languages. That knowledge is the most important asset and should be enough to define complex operations. These complex operations can be established from basic operations, such as a metaclass instantiation, attribute access, or collection items addition. Our work now focuses on how the language developer could count with these basic operations and use them. In this way the translation of these operations to an existing transformation language becomes transparent.

References

[1] Stahl, T. and Völter, M. Model-Driven Software Development. John Wiley & Sons, Ltd. (2006).

[2] Claudia Pons, Roxana Giandini, Gabriela Pérez. "Model Driven Software Development. Concepts and practical application". Editorial: EDUNLP and McGraw-Hill Education. (2010).

[3] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (2003)

[4] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain specific languages. ACM Computing Surveys, 37(4):316–344, 2005.

[5] MOF QVT Adopted Specification 2.0. OMG Adopted Specification. November 2005. http://www.omg.org

[6] ATLAS team: ATLAS MegaModel Management (AM3) Home page, http://www.eclipse.org/gmt/am3/. (2006)

[7] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science, Springer-Verlag (2006) 128–138

[8] Kolovos, DS, Paige, RF, Polack, FAC: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006).

[9] GME: The Generic Modeling Environment, Reference site, http://www.isis.vanderbilt.edu/Projects/gme. (2006).

[10] Richard C. Gronback. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional. ISBN: 0-321-53407-7, 2009.

[11] Steve Cook, Gareth Jones, Stuart Kent, Alan Cameron Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional. ISBN 0321398203, 2007.

[12] Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-based DSL Frameworks. (2006) OOPSLA Companion 2006:602-616[5].

[13] Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, Fabien Latry: Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD), July 3rd, Nantes, France (2006).

[14] Barbero, M., Bézivin, J., Jouault, F. Building a DSL for Interactive TV Applications with AMMA. In Proceedings of the TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum. Zurich, Switzerland (June 2007).

[15] Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. http://hal.ccsd.cnrs.fr/docs/00/06/61/21/PDF/rr0602.pdf (Downloaded March 2009).

[16] Jon Oldevik. MOFScript User Guide. Version 0.6 (MOFScript v 1.1.11), 2006.

[17] Kleppe, Anneke. MCC: A Model Transformation Environment. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, Spain, June 2006. (2006)

[18] Blanc,X., Gervais, M., Lamari, M. and Sriplakich, P.. Towards an integrated transformation environment (ITE) for model driven development (MDD). In Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2004), USA, July 2004. (2004)

[19] Oldevik, J. Transformation Composition Modeling Framework. DAIS 2005. Lecture Notes in Computer Science 3543, pp. 108-114. (2005)

[20] Wagelaar, Dennis. Composition Techniques for Rule-based Model Transformation Languages. Procs. of ICMT2008 – Int. Conference on Model Transformation. Zurich, Switzerland. July 2008. (2008)

[21] Bravenboer, M., Visser, E.: Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In: Proc. 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM Press (2004) 365-383

[22] OMG/MOF Meta Object Facility (MOF) 2.0. OMG Adopted Specification. October 2003. http://www.omg.org

[23] Weske Mathias, "Business Process Management: Concepts, Languages, Architectures". Springer, Pag 3-67. ISBN 978-3-540-73521-2. 2008

[24] Object Management Group (OMG), http://www.omg.org

[25] Business Process Modeling Notation (BPMN) Version 1.2 OMG, http://www.omg.org/spec/BPMN/1.2

[26] B. Weber and M. Reichert, Refactoring Process Models in Large Process Repositories. Bellahs`ene and L´eonard (Eds.): CAiSE 2008, LNCS 5074, pp. 124–139, 2008. Springer-Verlag Berlin Heidelberg 2008

[27] Fowler, M.: Refactoring-Improving the Design of Existing Code. Addison-Wesley, Reading (2000)

[28] Claudia Pons, Jerónimo Irazábal, Roxana Giandini and Gabriela Pérez. On the semantics of domain specific transformation languages: implementation issues. Chapter 13 of the Book "Software Engineering: Methods, Modelling, and Teaching", prefaced by Ivar Jacobson. (2011).

[29] Kermeta web site - http://www.kermeta.org

[30] Eugenia web site - http://www.eclipse.org/epsilon/doc/eugenia/

[31] Dimitrios S. Kolovos, Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A.C. Polack, Goetz Botterweck. Taming EMF and GMF Using Model Transformation, accepted and to appear in Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS) Oslo, Norway, October 2010

[32] DSMML web site. http://www.lifia.info.unlp.edu.ar/eclipse/DSMML/

[33] EMFText web site - www.emftext.org/