An Implementation Approach to Achieve Metamodel Independence in Domain Specific Model Manipulation Languages

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

Keywords: Model Driven Engineering, Model Transformation Language, Domain Specific Language.

Abstract: Model Driven Engineering proposes a software development process in which the key notions are models that allow engineers to precisely capture relevant aspects of a system from a given perspective and at an appropriate level of abstraction. Then, models are manipulated with the goal of enabling the automated development of a system from its corresponding models. Unlike general-purpose modeling languages, domain-specific modeling languages 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. In this work we elaborate on the notion of domain specific model manipulation language, that is to say a model manipulation language tailored to a specific domain. In contrast to well-known model manipulation languages, such as EOL or ATL, the language syntax and semantics are directly related to a specific domain and/or kind of manipulation, making manipulation easier to write and understand. We present an implementation approach achieving complete platform-independence. We illustrate the proposal through a practical example.

1 INTRODUCTION

Model Driven Engineering (MDE) (Stahl T. and Völter, 2006) (Pons C. et. al., 2010) (Kleppe G. et. al., 2003) proposes a software development process in which the key notions are models that allow engineers to precisely capture relevant aspects of a system from a given perspective and at an appropriate level of abstraction. Then, the automated development of a system from its corresponding models is realized by manipulating them. Model manipulation consists of a number of operations on the models, such as verifications, views, queries, transformations from model to model, transformations from model to code, etc.

Models can be expressed using different languages. Unlike general-purpose modeling languages (GPMLs), such us the UML, Domain-specific modeling languages (DSMLs), such as the Business Process Modeling Notation (BPMN) (Weske M., 2008), can simplify the development of complex software systems by providing domain-specific abstractions for modeling the system in a precise but simple and concise way. DSMLs have a simpler syntax (few constructs focused to the particular domain) but its semantics is much more complex (all the semantics of the particular domain is embedded into the language).

In a model-driven process, software is built by constructing one or more models, and successively manipulating them and transforming them into other models, until reaching an executable program code. A model manipulation program is a set of rules that together describe how a model can be checked (e.g. for consistency) and how a model written in the source language is mapped to a model written in the target language. Model manipulations are specified using a model manipulation language. There are already several proposals for model manipulation specification, implementation, and execution, which are being used by MDE practitioners (Czarnecki H., 2006). The term "model manipulation language" comprises all sorts of artificial languages used in model manipulation development including general-purpose programming languages, domain-specific languages (DSLs) (Mernik M., 2005), modeling and meta-modeling languages and ontologies. Examples include languages such as the standard QVT (QVT,

Irazábal J., Pérez G., Pons C. and Giandini R..
An Implementation Approach to Achieve Metamodel Independence in Domain Specific Model Manipulation Languages.
DOI: 10.5220/0004082800620069
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 62-69
ISBN: 978-989-8565-19-8
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

2005), ATL (ATL, 2006) (Jouault F., 2005) and EOL (Kolovos D. et. al., 2006).

These languages are specific for defining model manipulations but they are independent of any modeling domain; so they contain complex constructs referring to pattern matching mechanisms, control structures, etc. This can eventually compromise the primary aims for which the DSML was built: domain focus and conciseness. Consequently, an extra level of specialization should be achieved on them; we can define a manipulation language specifically addressed to a given domain, that is to say, a Domain Specific Model Manipulation Language (DSMML). For example, we can create a language dedicated to the definition of transformations between data-base models or a language addressed to the definition of transformations between business process models.

In this context, if we would like to take advantage of a very specific manipulation language we face the problem of implementing such a new language. There exist powerful frameworks for the definition of domain specific languages, such as Eclipse (GME, 2006) (Gronback R., 2009) and Microsoft DSL Tools (Cook S. et. al., 2007) (Greenfield J. et. al., 2004).

In the present work we describe a proposal for defining domain specific model manipulation languages and also we analyze a novel way to define their semantics. Our proposal consists in using MDE tools themselves for the implementation of such languages, which improves modularity and reuse. The article is organized as follows. Section 2 presents the main features of our proposal to define domain specific manipulation languages using MDE tools. Section 3 illustrates the use of the approach by the definition of a new DSMML. Section 4 compares our approach with related research and finally Section 5 presents the conclusions.

2 DSMML SEMANTICS: IMPLEMENTATION SCHEMA

Any language consists of two main elements: a syntactic notation (syntax) which is a 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 manipulation language as the semantic domain for the definition of the new DSMML´s semantics. Then, the semantic function μ is defined by a transformation written in a model-to-text transformation language (such as MOFScript (Oldevik J., 2006)). This M2T transformation takes a program written in the DSMML as input, and generates a program written in a general purpose manipulation language (such as EOL) as output. This schema is described in Figure 1.

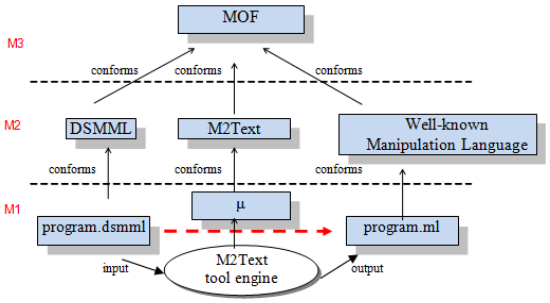

Figure 1: Transformation scenario.

The advantage of this technique is that the well-known manipulation 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.

3 USE CASE

In this section we present a new DSMML using the proposed approach. This section is organized as follows; first we introduce the domain, then we propose different meta-models for a simplified version of the domain. Next, we present the new DSMML trough some examples. And finally we describe the most relevant issues of its implementation.

3.1 Workout Plan Domain

In websites related to running we frequently see tables such as the one showed in Figure 2. Such tables describe workout plans to help people to reach their fitness goals. The workout plan usually has a duration expressed in weeks and each day of the week contains a list of exercises that must be done with specific requirements, such as intensity and duration. Given that we are considering this domain just to exemplify our approach, we will restrict its functionality by giving to the user the possibility to specify only the time for each exercise, but without considering intensity or complex exercises.

	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY	SUNDAY
1	Run: 50	Gym: 45	Run: 50	Gym: 45	Run: 50		Run: 75
2	Run: 50	Gym: 45	Run: 50	Gym: 45	Run: 50		Run: 75
3	Run: 60	Gym: 45	Run: 60	Gym: 45	Run: 60		Run: 90
4	Run: 60	Gym: 45	Run: 60	Gym: 45	Run: 60		Run: 90

Figure 2: A Workout Plan.

As we said before, the DSMML is independent of the underlying meta-model. That is to say, the language syntax will remain unchanged even if we use a different but equivalent meta-model for the domain. In order to provide concrete evidence about this feature, we will present two meta-models for this domain, which are displayed in Figure 3 and Figure 4 respectively.

It is worth to mention that if we add or remove information from the meta-model, the manipulation language may get affected by these changes. For example, if we add the possibility to specify the intensity at which the exercises should be done, we might change the language to support this new feature. This fact does not mean that the language depends on the underlying meta-model; on the contrary the language just depends on the available information while how that information was represented in the meta-model is completely irrelevant.

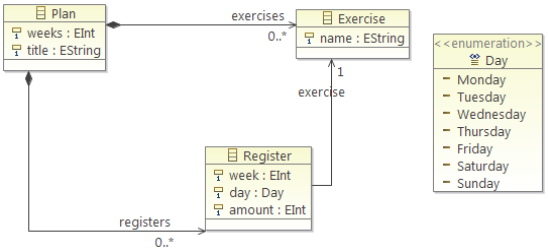

Figure 3: Workout Plan Meta-model, version 1.

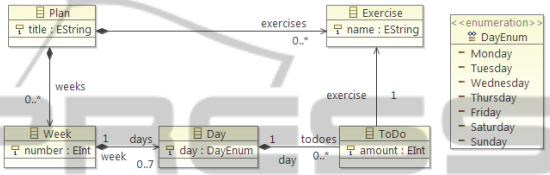

Figure 4: Workout Plan Meta-model, version 2.

3.2 WPML: A DSMML fitting the Workout Plan Domain

In this section we introduce WPML (Workout Plan Manipulation Language). Given the high level of abstraction of WPML we consider that the code is self-explanatory. You can find detailed information about the language in (DSMML, 2011). The following WPML code creates the model showed in Figure 2:

```
create plan "myplan.plan"

set title "My plan"
set weeks 4

add exercise Run
add exercise Gym

on weeks 1 and 2 {
 on days Monday and Wednesday and
      Friday {
  do Run as much as 50 minutes
 }

 on days Tuesday and Thursday {
  do Gym as much as 45 minutes
 }
 on days Sunday {
  do Run as much as 150% of Run
   on day Monday of week same week
 }
}

from week 3 to 4 {
```

```
  on all days {
  do Run as much as 120% of Run
         on day same day of week 1
  do Gym as much as 100% of Gym
         on day same day of week 1
  }
}
```

The code exhibited above generates a new model. Additionally, WPML allows us to make changes to an existent model. Obviously, in a real situation if you have the WPML code that generates the plan you would prefer to change the code, but this may not always be the case, e.g. the model could be generated by a tool or another language. So, for example, given the model presented above, suppose we would like to increment the Running time by a 10% on the entire plan and also we would like to establish Sunday as the recovering day (day without exercises) instead of Saturday. The new plan is illustrated in Figure 5.

	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY	SUNDAY
1	Run: 55	Gym: 45	Run: 55	Gym: 45	Run: 55	Run: 82	
2	Run: 55	Gym: 45	Run: 55	Gym: 45	Run: 55	Run: 82	
3	Run: 66	Gym: 45	Run: 66	Gym: 45	Run: 66	Run: 99	
4	Run: 66	Gym: 45	Run: 66	Gym: 45	Run: 66	Run: 99	

Figure 5: Modified Workout Plan.

The WPML code to make those changes on the original model could be:

```
use plan "myplan.plan"

on all weeks {
   on all days {
     increase Run by 10%
  }
  swap Saturday and Sunday
}
```

3.3 WPML: Implementation

This section covers the key aspects in the implementation of WPML. The organization of this section is as follows. First, the overall implementation schema is showed; then the functions and operations that are defined in the specific domain are implemented emphasizing their meta-model independence; finally, the WPML compiler is partially presented and the compilation results for the WPML are illustrated.

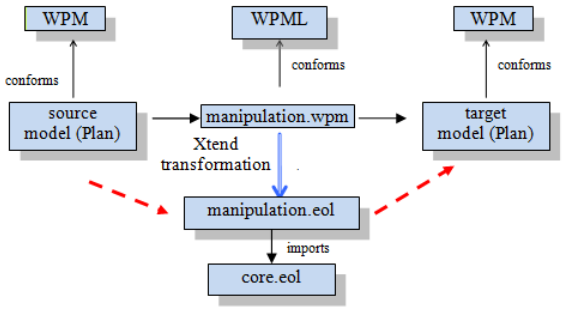

Figure 6: DSMML implementation schema using a translational approach.

Figure 6 shows an overview of the implementation schema where our domain specific manipulation language is translated to a general purpose manipulation language, in this case EOL. The EOL code generated from the WPML code imports a file named “core.eol”. This file contains the implementation of all the functionality provided by the specific manipulation language, such as setting the number of weeks of the plan, adding exercises, setting the duration of each exercise per week, swapping the schedule between two days, etc.

The following code is a fragment of the file “core.eol”; it uses the meta-model showed in Figure 3:

```
operation Plan doExerciseOnDayOfWeek
(ex:String,amount:Integer,
 day:Integer,week:Integer) {
 if (amount = 0) {
 self.removeExerciseInDayOfWeek(ex,
                              day,week);
 } else {
 self.getOrCreateRegister(ex,day,
             week).amount := amount;
 }
}

operation Plan
increaseExerciseByPercentOnDayOfWeek
 (ex:String,percent:Integer,
  day:Integer,week:Integer) {
 var r : Register =
      self.getRegister(ex,day,week);

 if (r<>null) {
 r.amount = r.amount +
           r.amount * percent / 100;
 }
}


operation Plan swapDaysOnWeek
 (day1:Integer,day2:Integer,
  week:Integer) {
for (r:Register in self.registers){
```

```
    if (r.week = week) {
    if (r.day.value = day1) {
        r.setDay(day2);
    } else {
         if (r.day.value = day2) {
        r.setDay(day1);
        }
      }
    }
  }
}
```

With the aim of showing more evidence about meta-model independence we have also implemented the language using a different meta-model. Next we present a fragment of the code contained in the file named "core.eol" adapted to the meta-model showed in Figure 4.

```
operation Plan doExerciseOnDayOfWeek
(ex:String,amount:Integer,day:Integer,
 week:Integer) {
 if (amount = 0) {
  self.removeExerciseInDayOfWeek(ex,
                              day,week);
 } else {
  self.getOrCreateToDo(ex,day,
               week).amount := amount;
 }
}

operation Plan
increaseExerciseByPercentOnDayOfWeek
(ex:String,percent:Integer,
 day:Integer,week:Integer) {
 var toDo : ToDo =
           self.getToDo(ex,day,week);

 if (toDo<>null) {
   toDo.amount = toDo.amount +
          toDo.amount * percent / 100;
 }
}

operation Plan swapDaysOnWeek
 (d1:Integer,d2:Integer,w:Integer) {
 for (d:Day in self.getWeek(w).days) {
   if (d.day.value = d1) {
   d.setDay(d2);
    } else {
   if (d.day.value = d2) {
     d.setDay(d1);
   }
    }
 }
}
```

Afterward, the compiler written with XTend (XText, 2011) creates an EOL file from a WPML file. This file imports the core.eol file and invokes its functions according to the WPML code. The following code is a fragment of the compiler:

```
def compile(Manipulation m) '''
import "../src/core.eol";
var p : Plan = getPlan();
«FOR c:m.metaChanges»
   «c.compileMetaChange»
«ENDFOR»
«FOR c:m.changes»
   «c.compileWeekChange»
«ENDFOR»
'''
…
def compileMetaChangeSetTitle(
MetaChangeSetTitle c) '''
p.setTitle("«c.title»");
'''
…
def compileWeekChangeForAllWeeks(
WeekChangeForAllWeeks c) '''
for (w in Sequence{1..p.getWeeks()})
{
  «FOR dc:c.changes»
    «dc.compileDayChange»
  «ENDFOR»
}
'''
…
def compileDayChangeSwapDays(
DayChangeSwapDays c) '''
 p.swapDaysOnWeek(«c.day1.value»,
                  «c.day2.value»,w);
```

The EOL code that we show next was generated by the compiler with the WPML code given before for the creation and manipulation of a plan respectively.

```
import "../src/core.eol";
var p : Plan = getPlan();

p.setTitle("My plan");
p.setWeeks(4);
p.addExercise("Run");
p.addExercise("Gym");

for (w in Sequence{ 1,  2 }) {
 for (d in Sequence{0,2,4}) {
   p.doExerciseOnDayOfWeek("Run",
                           50,d,w);
 }
 for (d in Sequence{1,3}) {
   p.doExerciseOnDayOfWeek("Gym",
                           45,d,w);
 }
 for (d in Sequence{6}) {
   p.doExerciseOnDayOfWeek("Run",
((p.getAmountOfExerciseOnDayOfWeek("
Run",0,w))*150/100),d,w);
 }
}
for (w in Sequence{3..4}) {
 for (d in Sequence{0..6}) {
```

SCIENCE AND TECHNOLOGY PUBLICATIONS

```
    p.doExerciseOnDayOfWeek("Run",((p.g
etAmountOfExerciseOnDayOfWeek("Run",d,1
))*120/100),d,w);
    p.doExerciseOnDayOfWeek("Gym",((p.g
etAmountOfExerciseOnDayOfWeek("Gym",d,1
))*100/100),d,w);
    }
   }
```

The EOL code showed next is generated by the compiler with the WPML code showed before for the modification of a previously created plan.

```
import "../src/core.eol";
var p : Plan = getPlan();

for (w in Sequence{1..p.getWeeks()})
{
 for (d in Sequence{0..6}) {
 p.increaseExerciseByPercentOnDayOf
                Week("Run",10,d,w);
 }
 p.swapDaysOnWeek(5,6,w);
}
```

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4 RELATED WORK

There are a number of features of our work that can be contrasted to previous works:

- The schema presented in this work could be considered as an evolution of the implementation schemas presented in (Irazábal et al., 2010), where the first approach covered consisted of writing a transformation in a general transformation language (e.g. ATL) taking two models as input, one with the model to by manipulated and the other with the statements to be executed, and building a model as the result of applying those statements to the model given as input; the other schema consisting in a two step transformation scenario, the first transformation (a model to text transformation) takes a model conforming the new DSMML and translates it to a general transformation language (e.g. ATL). Then, the generated transformation when executed over a model of the domain of interest performs the desired changes to it. In our current work, the transformation is written in a general transformation language (e.g. EOL) with the characteristic of being parameterised code. This way, the statements written in the new DSMML are translated (with a model to text transformation) to invocations to the previously written transformations, setting the parameters according to the elements to be manipulated. This way, the transformations are simpler and modularized.
- Abstraction and modularization of model transformations: Our approach can be seen as a technique for abstraction and modularization in that each high level manipulation (written in the DSMML) is associated with a lower level manipulation (written in a more general purpose language), but the users do not need to be aware of the details of the low level manipulations. 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 in (Kleppe A., 2006), the Model Bus approach (Blanc X., et. al., 2004), the modeling framework for compound transformations defined in (Oldevik J., 2005) and the module superimposition technique (Wagelaar D., 2008), among others. In contrast to these works, our approach generates the composed transformation specification in a simpler way, without introducing any explicit composition machinery.
- Creating languages that abstract from other more abstract languages: This subject has been intensely discussed in the literature on DSLs. For example, the MetaBorg (Bravenboer M. and Visser E., 2004) 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). Johannes shows how to develop DSLs as abstractions of other DSLs by transferring translational approaches for textual DSLs into the domain of modelling languages (Johannes J. et. al., 2009). The underlying notion of an embedded DSL has been discussed in (Hudak P., 1998). The idea of forwarding has been introduced in (Van Wyk E. et. al., 2002). An important distinction between these works and our work is the application to the MDE field. The AMMA framework (Kurtev I. et. al., 2006) allows us to define the concrete syntax, abstract syntax, and semantics of DSLs. In (Jouault F. et. al., 2006) (Barbero M. et. al., 2007) (Di Ruscio D. et. al., 2009) 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 manipulation language, by means of a

transformation written in a M2T transformation language.

- Concrete-syntax-based transformations: Contrary to traditional approaches to model transformation, our approach, such as the one presented in (Baar T. and Whittle J., 2007), uses the concrete syntax of a language for expressing transformation rules. The claim is that this simplifies the development of model transformations, as transformation designers do not need deep knowledge of the language's metamodel. In our approach, we use the abstract DSMML with a similar purpose: users do not need to count with any knowledge of the abstract syntax of the involved modeling languages; they just use the simple syntax of the DSMML.

5 CONCLUSIONS

In this article we have explained the concept of domain specific model manipulation language, that is to say model manipulation languages tailored to a specific domain. In contrast to well-known model manipulation languages, such as EOL and ATL, the language syntax and semantics are directly related to a specific domain and/or kind of manipulation, making manipulation easer to write and understand.

In contrast to an approach where a general purpose model manipulation language is used, our approach provides the following benefits: the complexity of model manipulation programs gets reduced. A program is composed by few lines of high expressive commands. Domain experts will feel more comfortable using a specific language with constructs reflecting well-known concepts (such as, exercise and week in our example); consequently it is predictable that they will be able to write more understandable and reusable manipulation programs in a shorter time. Manipulation developers do not need to know the intricate details of the model manipulation languages, as these are encapsulated in the DSL constructs. This leads to a natural separation into a language designer and a manipulation programmer role, with a reduced learning effort for the later.

Also, we have proposed an implementation schema in which the transformation that compiles the DSMML sentences consists of invocations to previous defined operations written in a well known transformation language (e.g. EOL). This fact provides several advantages: the language semantics is formally described; it is executable; the semantics is understandable because it is written in a well-known language; the semantics can be easily modified by adding new transformation rules or even by radically changing the target language. Although this transformation may be considered as a compiler, the amount of programming skills required to create it is smaller than for creating a compiler to source code.

As an experimental example in this article we have reported the definition of a DSMML in the domain of workout plans and we have described its implementation using MDE tools. The experience was successful; showing the advantages of defining DSMML for model transformations within the same language, that is to say, transformations that locally change an existent model producing a new model that conforms to the same metamodel. Currently we are working in the definition of other DSMMLs in other domains.

It is also important to take the benefits coming from the platform-independence of the model manipulation language into account; on one hand the language is independent of the underlying metamodel and on the other hand we are able to transform and execute the manipulation programs onto different model manipulation platforms, in the examples we have used EOL and ATL, but any other manipulation language can be used.

REFERENCES

ATLAS MegaModel Management. (2006). http:// www.eclipse.org/gmt/am3/.

Baar, T., and Whittle, J. (2007). On the Usage of Concrete Syntax in Model Transformation Rules. In Book: Perspectives of Systems Informatics. *LNCS 4378*, Springer Heidelberg, Berlin.

Barbero, M., Bézivin, J., and Jouault, F. (2007). 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.

Blanc,X., Gervais, M., Lamari, M. and Sriplakich, P. (2004). 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.

Bravenboer, M., and Visser, E. (2004). Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. *In OOPSLA'04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press. pp. 365–383.

Cook Steve, Gareth Jones, Stuart Kent, and Alan Cameron Wills. (2007). Domain-Specific Development with Visual Studio DSL Tools. *Addison-Wesley Professional.* ISBN 0321398203.

Czarnecki, Helsen. (2006). Feature-based survey of model transformation approaches. *IBM System Journal*, v.45, n.3.

DSMML (2011) http://www.lifia.info.unlp.edu.ar/eclipse/DSMML/.

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

Jouault Frédéric and Ivan Kurtev. (2005).Transforming Models with ATL. In: Jean-Michel Bruel (Ed.), Satellite Events at the MoDELS 2005 Conference, LNCS 3844. Springer Berlin / Heidelberg., Montego Bay, Jamaica (pp. 128-138) .

Jouault Frédéric, Jean Bézivin, Charles Consel, Ivan Kurtev, and Fabien Latry. (2006). Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages in *Proceedings of the First ECOOP Workshop on Domain-Specific Program Development.* Nantes, France.

GME (2006). http://www.isis.vanderbilt.edu/Projects/gme.

Greenfield, J., Short, K., Cook, S., Kent, S., and Crupi, J. (2004). Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools (1st ed.): Wiley.

Gronback R. C. (2009). Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. *Addison-Wesley Professional*. ISBN: 0-321-53407-7.

Hudak, P. (1998). Modular domain specific languages and tools. In ICSR'98: Proceedings of the 5th International Conference on Software Reuse, *IEEE Computer Society Press.* pp. 134–142. June. Victoria, B.C., Canada.

Irazábal J., Pons C., Neil C. (2010). Model transformation as a mechanism for the implementation of domain specific transformation languages. *SADIO Electronic Journal of Informatics and Operations Research*. vol. 9, no.1.

Johannes, J., Zschaler, S., Fernandez, M., Castillo, A., Kolovos, D., and Paige, R. (2009). Abstracting Complex Languages through Transformation and Composition. *In MoDELS'09: Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems*. USA, LNCS, Springer. October. Denver, Colorado, USA.

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

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

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2006). The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128-142. *Springer Heidelberg*.

Kurtev, I. and Bézivin, J. and Jouault, F. and Valduriez, P. (2006) Model-based DSL frameworks. *In: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Portland, Oregon, USA. (pp. 602-616). ACM Press. ISBN 1-59593-491-X.

Mernik Marjan, Heering Jan, and Sloane Anthony M. (2005). When and how to develop domain specific languages. *ACM Computing Surveys*, v.37 n.4, p.316-344.

Meta Object Facility (MOF) 2.0. (2003). http://www.omg.org.

OCL. (2006). http://www.omg.org/spec/OCL/2.0.

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

Oldevik Jon. (2006). MOFScript User Guide. http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf.

OMG. (2011). http://www.omg.org.

Pons Claudia, Giandini Roxana, and Pérez Gabriela. (2010). "Model Driven Software Development. Concepts and practical application". Buenos Aires, Agentina. EDUNLP and McGraw-Hill Education.

Pons Claudia, Irazábal Jerónimo, Giandini Roxana and Pérez Gabriela. (2011). On the semantics of domain specific transformation languages: implementation issues. *Software Engineering: Methods, Modeling, and Teaching*, Chapter 13. ISBN: 9789588692326.

QVT Adopted Specification 2.0. (2005). http://www.omg.org.

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

Van Wyk, E., de Moor, O., Backhouse, K., and Kwiatkowski, P. (2002). Forwarding in attribute grammars for modular language design. *In Horspool, R.N., ed.: Int Conf. on Compiler Construction*. LNCS 2304, Springer, Berlin / Heidelberg pp. 128–142.

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

Weske Mathias. (2008). Business Process Management: Concepts, Languages, Architectures. Springer, (pp. 3-67). ISBN 978-3-540-73521-2.

XTend. (2011). http://www.eclipse.org/Xtext/#xtend2

XText. (2011). http://www.eclipse.org/Xtext/