Model-driven Development from a Programming Language Perspective

Christiano Braga

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)

`cbraga@ic.uff.br`

Abstract. *Model-driven development techniques and programming languages disciplines such as generative programming, compiler and interpreter construction, static analysis of programs, and program verification are closely coupled. However, lessons learned from such disciplines are often neglected. This paper aims at pointing-out some of these lessons and look at the concept of* transformation contract *from a programming languages perspective.*

1. Introduction

The literature about model-driven development (MDD, e.g. [Kleppe et al. 2003, Kleppe 2008]) clearly presents metamodels as languages and shows how domain-specific languages may be declared as metamodels.

From this perspective, understanding a *model transformation* as a *compiler* is immediate as a relation between two (or more) languages. Similarly, one may build model interpreters. Model interpreters may be quite useful, for instance, in the context of model transformation validation. In [Braga 2010], the present author used an OCL interpreter [Clavel et al. 2008], which applies OCL expressions, defined on a given metamodel, to model descriptions, to validate a model transformation.

Verification techniques, such as logic-based ones, may be directly applied to models when a proper model transformation is defined, that is, a model transformation that produces a sound logical theory from a given model, such as in [Berardi et al. 2005, Braga and Hæusler 2010]. There, the authors use description logic to reason about UML class diagrams and access control models consistency, respectively. These are examples that model transformations may be used not only to refine code out of abstract models but also to generate models at the same level of abstraction and *with different purposes*. A logical theory is one such example. The programming languages (PL) and formal methods (FM) communities are quite familiar with techniques such as these but the software engineering community sometimes overlooks lessons learned from these communities.

The objective of this paper is to highlight many intersections between MDD and PL concepts and suggest how techniques used by PL and FM communities may be used in MDD. In particular, we propose the concept of *transformation contracts* for rigorous model transformations specification and implementation.

This paper is organized as follows. Section 2 highlights many intersections between MDD concepts and PL concepts. Section 3 presents the concept of transformation contracts and relates it to static semantics of programming languages. Section 4 shows an example contract as a static semantics specificaiton. Section 5 concludes this paper with related work and final remarks.

2. Model-driven development and programming languages

The starting point is to convince ourselves that a metamodel is a language and models are sentences in such a language. If we commit to a particular (structural) metamodeling language, such as UML class diagrams, we can relate its (textual) representation to the Backus-Naür form (BNF), a formal notation used to describe the syntax of programming languages. A simple translation[1] is to understand each entity and relationship as a non-terminal in the induced grammar and each model element constructor, such as an XMI tag `<UML:CLASS name=`x`>`, as a terminal. Generalizations produce alternatives in the grammar and multiplicites induce sequences of non-terminals.

The metamodel in Figure 1 is an excerpt of the metamodel for SecureUML [Basin et al. 2006], a domain-specific modeling language for access-control policies. Essentially, the modeling language allows for the specification of actions over controlled resources that may be executed by users within roles iff the authorization constraints of the permissions that relate a given role to the given resource's actions are fulfilled.

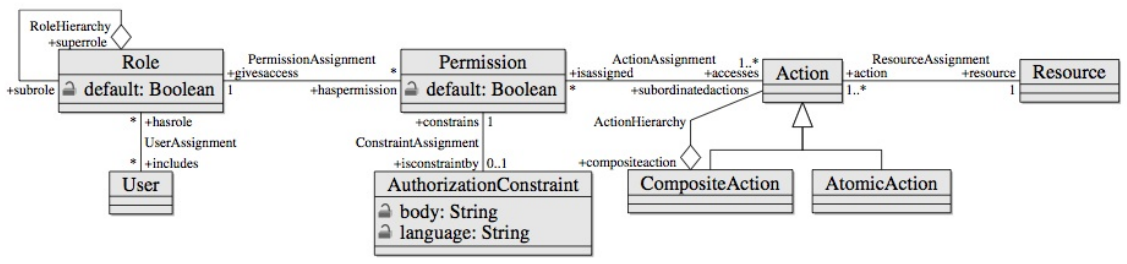

Figure 1. Simplified SecureUML metamodel

The BNF for this language could be written as follows, where the superscripts $*$, $+$ and $?$ denote reflexive closure, transitive closure, and optionality as usual.

POLICY ::= ROLE | PERMISSION | ACTION | RESOURCE |
USER | AUTHORIZATION-CONSTRAINT |
ROLE-HIERARCHY | PERMISSION-ASSIGNMENT* |
ACTION-ASSIGNMENT* | RESOURCE-ASSIGNMENT$^+$ |
USER-ASSIGNMENT* | CONSTRAINT-ASSIGNMENT$^?$ |
ACTION-HIERARCHY

ACTION ::= COMPOSITE-ACTION | ATOMIC-ACTION

It is interesting to note that, even though a model may be a graph, its metamodel may be a tree. An example is the definition of transition systems. The metamodel (that is, the language syntax) is a tree but the models are graphs. Moreover, even when a graph language is required, one may use a *graph grammar* [Rozenberg 1997] to formalize its syntax.

If we are convinced that metamodels are languages, we may now start building on that. Let's see how programming languages' (PL) perspectives may be applied to MDD. Section 2.1 gives a syntactic perspective and Section 2.2 gives a semantic one.

[1]Cardinality constraints require a more careful treatment then what we describe here with regular expressions' wild cards. However, this should be enough to make our point that metamodels are actually languages.

2.1. Syntax

The first PL's perspective that we may apply is the very notion of *concrete* and *abstract* syntax. The concrete syntax of a programming language is the one that programmers write programs with. The abstract syntax is the one that a compiler uses to process a program, which is much simpler from a structural point of view, with fewer nodes then the concrete syntax tree.

We may apply this idea to models as well. If we consider, once again, UML class diagrams as our metamodeling language, a UML *profile* may be seen as *concrete syntax* and a *metamodel* as *abstract syntax* for a modeling language. Given a particular model written in a UML profile, it will have information related with UML model elements and stereotypes, as opposed to a model instance of its metamodel, which will only have instances of model elements of its metamodel. Information in the latter model is the one relevant for model transformation, verification and validation.

As an example, if we consider a modeling language, such as SecureUML (see Figure 1), the XMI representation of a given policy as an instance of SecureUML's profile and the XMI representation of the given policy as an instance of SecureUML's metamodel are quite different. The latter is much simpler, as expected. Due to space constraints we refer to `http://mda.ic.uff.br/bw-dsdm/trc-uml.xmi` and `http://mda.ic.uff.br/bw-dsdm/trc-suml.xmi` for XMI files representing a simple policy as an instance of SecureUML's profile and as an instance of SecureUML's metamodel, respectively. The file `http://mda.ic.uff.br/bw-dsdm/trc.png` shows a graphical rendering of the model.

2.2. Semantics

We may apply another PL's perspective to model transformation rules. A model transformation relates two metamodels, that is, two languages, just like a *compiler*. Model transformation rules act precisely in the same manner as *semantic actions* in a compiler, that is, the behavior that must be applied when a certain node is reached while traversing the parse tree. Similarly, the formal semantics specification for a programming language defines a *semantic equation* for each sentence in a language that specifies the sentence's meaning in terms of a formal framework. In denotational semantics, for instance, such equations are represented by functions whereas in operational semantics as transition rules.

Both a semantic action in a compiler and semantic equation in a formal specification should be *syntax driven*, that is, should be defined by *structural induction* on the syntax of the source language. Similarly, a model transformation rule should *not* be defined in an *ad hoc* way over an XMI file, using example models to guide the model transformation rules, but rather, by structural induction over the metamodel that represents the source language. If the grammar is used, we can actually *prove* that for every sentence in the source language there exists a sentence in the target language, and the other way around.

A lesson from the formal methods community may be applied if one understands a model transformation as a *term rewrite system* [Dershowitz and Jouannaud 1990], that is, a set of rules that specify how terms (sentences) in a language are changed into terms in another language. With such an interpretation, one may automatically verify typical properties for term rewrite systems on model transformations, such as termination and

Church-Rosser. Termination means that all rewrites will eventually end. The Church-Rosser property means that a rewrite system is a *function*, that is, the rewrite system will always produce the same output term for a given input term. For the former case, even though the general problem is undecidable, there are techniques that automatically verify termination in certain cases. The Church-Rosser property may be automatically checked by calculating the so-called critical pairs of the rewriting system, that is, terms (actually, patterns) that make two rules overlap.

Another lesson from the formal methods community may be applied if the semantics of each modeling language, source and target, are formally defined. Then, one can actually prove the *correctness* of the model transformation, that is, show that the model transformation is *semantics preserving*. An example is the work reported in [Berardi et al. 2005]. The authors describe a correct transformation from UML class diagrams to Description Logic (DL) theories (or knowledge bases, using DL terminology). The DL representation of a class diagram allows for reasoning about model consistency, among other properties, that is, to check if a given model has instances. Another example would be, in the context of behavioral models, to prove that a model transformation produces a model with an associated transition system which is equivalent, or *bisimilar*, to the transition system associated with the source model.

Model transformations may also be *validated*, that is, checked on particular applications of the model transformation with respect to a given specification. We now introduce the concept of *transformation contracts* for model transformation validation. Following Bertrand Meyer's Design by Contract (DbC) approach [Meyer 1997], a transformation contract is a set of assertions specifying what a model transformation must do, similarly to a DbC contract for an object-oriented program, which is a set of assertions specifying what an object-oriented program must do. In DbC, the contract is checked on particular executions of a given program. A failing assertion points out a flaw in the program. A transformation contract is checked on each application of a model transformation, making sure that the pair formed by the source and target models respect the invariants that comprise the transformation contract. One particular setting in which transformation contracts may be used is when UML class diagrams are chosen as metamodeling language and the Object Constraint Language is chosen as assertion language. Given an OCL interpreter, such as [Clavel et al. 2008], transformation contracts may be checked in the same way as DbC contracts are. We refer the interested reader to [Braga 2010] for an application of the transformation contract approach to the domain of access control modeling.

We continue in Section 3 discussing an interpretation of transformation contracts as static semantics specifications in programming languages.

3. Transformation contracts and static semantics

Transformation contracts are *invariants* on the metamodel resulting from the join of the source and target metamodels related by a given model transformation. The *joined model* is the disjoint union of two metamodels, source and target, united with a set of relations between source and target metaclasses, which are disjoint from the relations in the source and target metamodels.

Static semantics are usually related with typing rules of programs. One may generalize it by saying that the static semantics specification of a programming language specifies rules for structural well-formedness of programs in a given language, that is, a program is only well-formed if it is syntactically correct, with respect to the language's grammar, and with respect to the language typing rules.

We may apply this idea to models. *We say that a model is* well-formed *when it is an instance of a given metamodel and the invariants of the metamodel hold in the model.* Hence, well-formedness may be understood as the static semantics specification of the language induced by a given metamodel.

Intuitively, a transformation contract produces a well-formed model when the source model is well-formed, the target model is well-formed, and the relations between what was generated and the source fulfills the *invariants* of the contract.

Formally, the conformance relation of a model wrt. a metamodel $\mathcal{L}$ is given by assertions of the form $\Gamma_{\mathcal{L}} \vdash P$, where $\Gamma_{\mathcal{L}}$ is an environment, that is, a set of bindings between model elements of $\mathcal{L}$ and instances of it and P are *patterns* in the language induced by $\mathcal{L}$, that is, P is sentences in $\mathcal{L}$ with *variables*.

Before we define the $\Gamma_{\mathcal{L}}$, let us give some auxiliary definitions first. The class environment $\Gamma_{C_{\mathcal{L}}}$ for a metamodel $\mathcal{L}$ is defined as follows where, $C_{\mathcal{L}}$ is the set of classes of $\mathcal{L}$ and $I_{C_{\mathcal{L}}}$ is a set of instances of $C_{\mathcal{L}}$

$$\Gamma_{C_{\mathcal{L}}} = I_{C_{\mathcal{L}}} \times C_{\mathcal{L}}.$$

The associations environment $\Gamma_{A_{\mathcal{L}}}$ for a metamodel $\mathcal{L}$ is defined as follows

$$\Gamma_{A_{\mathcal{L}}} = I^2_{C_{\mathcal{L}}} \times (I_{E_{\mathcal{L}}}, E_{\mathcal{L}})^2$$

where $E_{\mathcal{L}} = R_{\mathcal{L}} \times C_{\mathcal{L}}$ represents the set of association ends of $\mathcal{L}$ and $R_{\mathcal{L}}$ the set of roles of $\mathcal{L}$. Finally, the environment for a metamodel $\mathcal{L}$ is defined as

$$\Gamma_{\mathcal{L}} = \Gamma_{C_{\mathcal{L}}} \cup \Gamma_{A_{\mathcal{L}}}.$$

Finally, given a model transformation $\theta : \mathcal{S} \rightarrow \mathcal{T}$, where $\mathcal{S}$ and $\mathcal{T}$ are the source and target metamodels, θ produces a well-formed model iff

$$\frac{\Gamma_{\mathcal{S}} \vdash s \quad \Gamma_{\mathcal{T}} \vdash t \quad \Gamma_{\Theta} \vdash r}{\Gamma_{\Theta} \vdash s \uplus t \uplus r}$$

where s is an instance of $\mathcal{S}$; t is an instance of $\mathcal{T}$; $\mathcal{R}_\theta$ is the set of relations between $\mathcal{S}$ metaclasses and $\mathcal{T}$ metaclasses, which is disjoint from $\mathcal{S}$ relations and $\mathcal{T}$ relations; r is a set of links between objects in s and t; $\Theta = \mathcal{S} \biguplus \mathcal{T} \biguplus \mathcal{R}_\theta$, with $\biguplus$ the disjoint union of metamodels and $\uplus$ is the disjoint union of models.

In the next section we show an example of the formalization of transformation contracts as static semantics specifications.

4. Example: A transformation contract for a transformation from SecureUML to AAC

In this section we exemplify how a transformation contract may be seen as a static semantics specification for a transformation from a SecureUML to AAC. SecureUML and

AAC (Aspects for Access Control) [Braga 2010] are modelling languages for access control and aspect concepts to represent access control elements. SecureUML was briefly explained in Section 2. AAC declares abstract classes to encapsulate controled resources, a resource action is represented by a method, and aspects manage access to the controled resources. The aspects' advices implement the access' constraints.

We have defined a transformation from SecureUML to AAC. The metamodels of the two languages and the relations of the joined metamodel are given in Figures 1, 2 and 3, respectively. (Figure 3 refers to SecureUML's metaclasses Entity, Attribute and Method which are not present in Figure 1. They are all subclasses of Resource.) The joined metamodel is specified as $\Theta = SecureUML \uplus AAC \uplus \mathcal{R}$, where $\mathcal{R} = \{(entity, resclass), (entity, aspect), (attribute, resattribute), (attribute, resmethod), (method, resmethod), (advice, constraints)\}$.

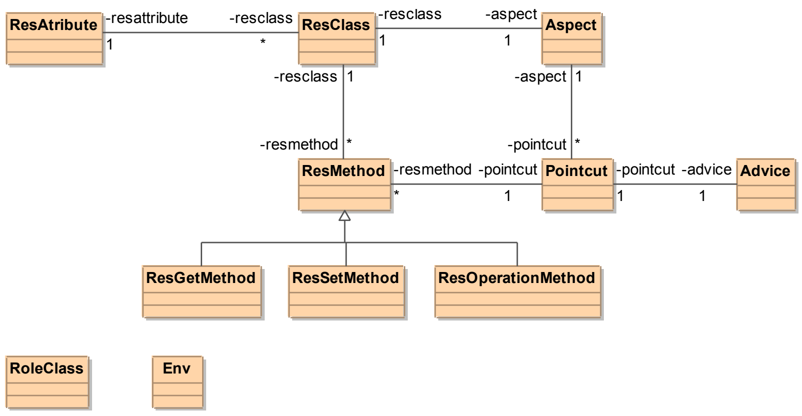

Figure 2. The Metamodel of AAC

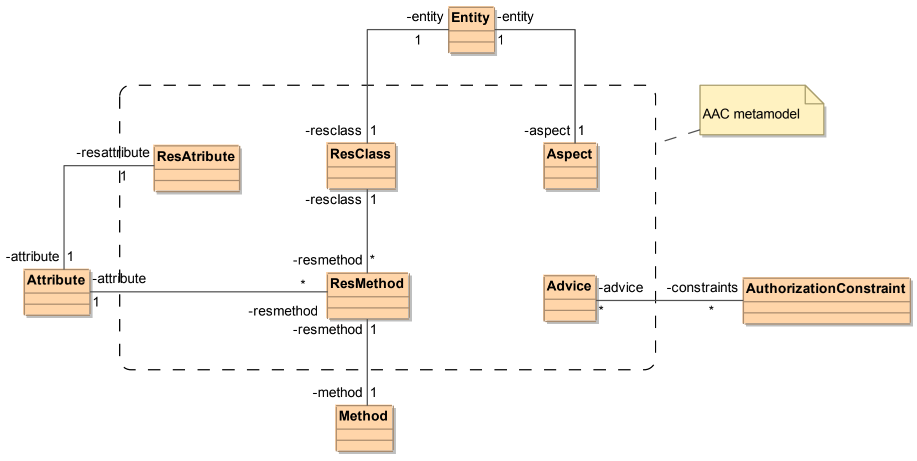

Figure 3. The relationships among SecureUML and AAC metamodels

Next, we show how two invariants of the transformation contract, may be represented as conformance rules. In what follows, we abbreviate (i, c) to $i : c$, with $i \in I_{C_{\mathcal{L}}}$ and $c \in C_{\mathcal{L}}$, and $((i_1, i_2), ((r_1, c_1), (r_2, c_2)))$ to $c_1 : i_1 \overset{r_1 \quad r_2}{\rule{1.5em}{0.4pt}} i_2 : c_2$, with $i_1, i_2 \in I_{E_{\mathcal{L}}}$ and $(r_1, c_1), (r_2, c_2) \in R_{\mathcal{L}}$.

1. *For each Entity there exists a single ResClass.*

This is represented via multiplicity invariant for the relation entity-resclass in the merged metamodel.

$$\frac{\Gamma \vdash e : \mathit{Entity} \quad \Gamma \vdash c : \mathit{ResClass} \quad \Gamma \vdash_{\mathit{Entity}} : e \xrightarrow{\mathit{entity} \quad \mathit{resclass}} c' : \mathit{Class} \quad c = c'}{\Gamma \vdash_{\mathit{Entity}} : e \xrightarrow{\mathit{entity} \quad \mathit{resclass}} c : \mathit{Class}}$$

2. *For each Entity there exists a single Aspect that controls the access over the ResClass associated with the Entity.*
The multiplicities between Entity and Aspect are guaranteed via the multiplicity invariant for the relationship entity-aspect in the merged metamodel. Moreover:

```
context Entity inv:
Entity.allInstances−>forAll(e | e.resclass = e.aspect.resclass)
```

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathit{Entity} \quad \Gamma \vdash a : \mathit{Aspect} \quad \Gamma \vdash c : \mathit{ResClass} \\ \Gamma \vdash_{\mathit{Entity}} : e \xrightarrow{\mathit{entity} \quad \mathit{resclass}} c : \mathit{ResClass} \qquad \Gamma \vdash_{\mathit{Entity}} : e \xrightarrow{\mathit{entity} \quad \mathit{aspect}} a' : \mathit{Aspect} \\ \Gamma \vdash_{\mathit{Aspect}} : a' \xrightarrow{\mathit{aspect} \quad \mathit{resclass}} c : \mathit{ResClass} \quad a = a'\end{array}}{\Gamma \vdash_{\mathit{Entity}} : e \xrightarrow{\mathit{entity} \quad \mathit{aspect}} a : \mathit{Aspect}}$$

5. Related work and final remarks

The idea of a transformation model is central to our work. In [Bézivin et al. 2006] the idea of a transformation model instead of a model transformation is discussed. In [Cariou et al. 2004] the concept of transformation contract is explored specifically for OCL specifications in UML class models.

Some approaches transform models into other languages and formalisms using solvers or theorem provers to validate it. In [Cabot et al. 2008] the authors discuss a way to verify UML/OCL models translating them to Constraint Satisfaction Problem (CSP). After the translation, a constraint solver is used to verify if the problem is satisfiable under a finite search space of possibilities. The advantage of this method is the guarantee that the model and its invariants are well defined. Their approach is similar to ours when we connect the approach described in this paper together with [Braga and Hæusler 2010].

The main difference between the approaches described above and ours is that they do not see metamodels as languages explicitly and do not take advantage of the many techniques that may be applied to models when the perspectives described in this paper are considered.

A quick remark on decidability of transformation contracts. General invariants specified in first-order logic are undecidable. However, they may be validated following the Design by Contract approach. Therefore, transformation contracts may be validated using a similar approach, either interpreting OCL specifications or, as another example, rewriting equations over terms representing class models or other structural models. (The latter approach is actually how the ITP/OCL [Clavel and Egea 2006], an OCL interpreter, works.) The interpretation for transformation contracts as static semantics specifications that we present here may actually be executed and reasoned about in a logic framework.

Future work lies on the application and development of programming languages and formal methods techniques to the context of MDD, in particular, to continue the development of the transformation contract concept.

References

Basin, D., Doser, J., and Lodderstedt, T. (2006). Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91.

Berardi, D., Calvanese, D., and De Giacomo, G. (2005). Reasoning on UML class diagrams. *Artif. Intell.*, 168(1-2):70–118.

Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., and Lindow, A. (2006). Model transformations? transformation models! In *Proceedings of the 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6*, volume 4199, pages 440–453. Springer.

Braga, C. (2010). A transformation contract to generate aspects from access control policies. *Journal of Software and Systems Modeling*. DOI: 10.1007/s10270-010-0156-x.

Braga, C. and Hæusler, E. H. (2010). Lightweight analysis of access control models with description logic. *Innovations in Systems and Software Engineering*, 6:115–123.

Cabot, J., Clarisó, R., and Riera, D. (2008). Verification of UML/OCL class diagrams using constraint programming. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80. IEEE Computer Society.

Cariou, E., Marvie, R., Seinturier, L., and Duchien, L. (2004). OCL for the specification of model transformation contracts. In Patrascoiu, O., editor, *OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal*, pages 69–83.

Clavel, M. and Egea, M. (2006). ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In *Algebraic Methodology and Software Technology, Proceedings of the 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8*, pages 368–373.

Clavel, M., Egea, M., and de Dios, M. A. G. (2008). Building an efficient component for ocl evaluation. *ECEASST*, 15.

Dershowitz, N. and Jouannaud, J.-P. (1990). Rewrite systems. In van Leeuwen, J., editor, *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 243–320. MIT Press, Cambridge, MA, USA.

Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional.

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

Meyer, B. (1997). *Object-Oriented software construction*. Prentice Hall, 2nd. edition.

Rozenberg, G., editor (1997). *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.