Metamodeling and Language Engineering

Étienne André

February 15, 2006

Abstract

Language Engineering aims at providing advanced techniques to design, implement and maintain languages or metalanguages. The Meta-Modeling Framework provides a method applying Object-Oriented modeling to the definition of languages, as well as a powerful meta-circular Meta-Modeling Language based on OCL, allowing language engineers to define high-quality languages at low cost. In order to perform a better reusability, Language Engineering also provides language composition mechanisms, as composition of model elements, inspired by Software Engineering.

1 Introduction

Since the development of the first programming languages, needs toward languages development and maintainability have always become higher. With the development of modeling and metamodeling, these needs increase. Thus, the discipline of Language Engineering is required to support the design, the implementation, and the validation of languages with the goal to deliver languages at low cost with high quality. Language Engineering should also provide methods to reuse parts of languages.

For example, let us imagine a compiler transforming a language A into a language B, the language B generally being bytecode or assembly language. This compiler should parse languages sentences, perform conformity checks, and translate these sentences to an interpretable or executable representation. If one decides this compiler should now transform another language (for example C) into the language B, one could rewrite the whole compiler. Language Engineering should also help software engineers to reuse parts of the first compiler for the second one, or even define a generic compiler being then specialized into the first one or the second one.

This document will first define a few concepts, like language, Language Engineering and Domain Specific Languages. Then, it will present the Meta-Modeling Framework and give an example (the Small Modeling Language), before giving an overview of Language Composition techniques.

2 Concepts

2.1 Language

[wiki] gives an interesting definition of a natural language, which can also be used for formal languages: "a formal language can be thought of as a set of formal specifications concerning syntax, vocabulary, and meaning."

A more formal definition is given by [cla01]: "a language consists of models for concrete syntax, abstract syntax and for the semantic domain."

The concrete syntax of a language specifies which arrangements of symbols in a physical representation of a language sentence are considered well-formed. Concrete syntaxes are usually specified with context-free grammars. For example, the concrete syntax in a language for class diagrams represents the allowed layouts of the boxes and the lines.

The abstract syntax of a language describes the structure of each language construction, leaving out the details of concrete representations. A context-free grammar can also be used to specify it. In our example of a language for class diagrams, the abstract syntax is formed by the classes and the associations between them.

The semantic domain of a language specifies the well-formedness rules and the meaning of the language. Thus, it has to define structural constraints on the abstract syntax of the language. In our example of a language for class diagrams, the semantic domain represents the objects and the associations between them.

2.2 Language Engineering

For [wiki], Language Engineering is the creation of natural language processing systems whose cost and outputs are measurable and predictable. Although this definition concerns natural languages, it can be kept for formal languages.

For [bez05], the goals of Language Engineering are the definition of abstract syntax and well-formedness rules, as well as the definition of operational or denotational semantics. Operational semantics is an approach giving a meaning to computer programs in a mathematically rigorous way, whereas denotational semantics is an approach to formalizing the semantics of computer systems by constructing mathematical objects which express the semantics of these systems. Language Engineering is also supposed to define consistency and refinement relations, and model transformations. In other words, Language Engineering must provide methods to check the absence of contradictions as well as refinement and transformations techniques.

2.3 Domain Specific Language

A Domain Specific Language is a language designed to be useful for a specific set of tasks. Contrary to General Purpose Languages (GPL), DSLs tend to be focused on doing only one sort of task, but on doing it well.

DSLs should enhance quality, by using a specific language for a specific problem, as well as maintainability as one only needs to change code linked to our specific problem in case of modifications - and not the whole code structure. DSLs should also enhance portability and reusability, as a code part can be exported to another program, even if this second program doesn't use the same programming language.

Unfortunately, DSLs also have some disadvantages, in particular with respect to their costs: a DSL is expensive to design, to implement, and to maintain. However, this document will explain later how Language Engineering can take part in the costs reduction.

To quote a few examples of DSLs, GraphViz can be used to define directed graphs, Csound to create audio files, whereas the famous YACC is used for parsing and compilers. SQL, the database language, can also be quoted, although this language also fulfills the definition of General Purpose Languages.

2.4 Catalysis Method

Started 1992 by Desmond D'Souza and Alan Wills, who co-authored the first Catalysis Book ([dsou98]), the Catalysis Method is an Object-Oriented approach. Complying with many standards, it helped to define the Unified Modeling Language 1.0 (UML). The key concepts are described in the following paragraphs.

First, the action aims at being as important as the object. The action can even be defined as an object.

Furthermore, a precise vocabulary is defined for collaborations and models, permitting clear specification of responsibilities, while not imposing implementation decisions.

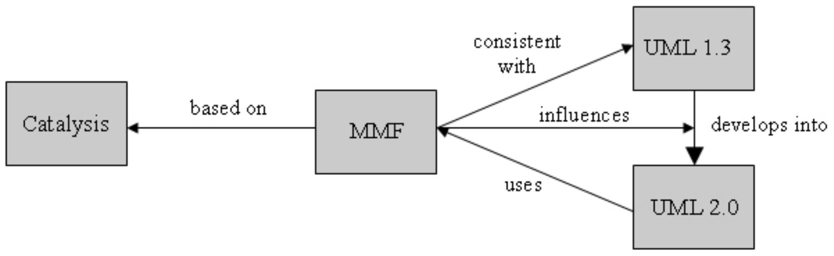

Figure 1: Interactions between MMF, UML and Catalysis

The refinement lets the software engineers describe the interactions between objects at possibly many levels of details. Objects can contain other objects, all these objects interacting with each other at different scales.

The Catalysis Method also provides the software engineers with a package import mechanism, as well as with modeling frameworks defined as templates.

And last, the separation of concerns makes a clear conceptual separation between the decisions of *What* (the combined behavior of a group of objects), *Who* (the responsibilities across participants and the dependencies between them), and *How* (patterns and interactions which provide the service of components).

3 The Meta-Modeling Framework

Described 2002 in [cla01], the Meta-Modeling Framework aimed at improving UML 1.3, where the authors found many deficiencies. They also wanted to support the Model-Driven Architecture (MDA) recently defined by the OMG.

This Meta-Modeling Framework (MMF) was based on the Catalysis Method and should be as consistent as possible with UML 1.3. It also influenced, in some domains strongly, the development of UML 1.3 into UML 2.0, as shown on figure 1.

This framework is split in three parts: a Method for Meta-Modeling, a Language for Meta-Modeling, and a Tool for Meta-Modeling.

3.1 Method for Meta-Modeling

The Method for Meta-Modeling (MMM) is an approach applying Object-Oriented modeling to the definition of languages, especially Object-Oriented modeling languages.

The Object Constraint Language (OCL) is used to define well-formedness

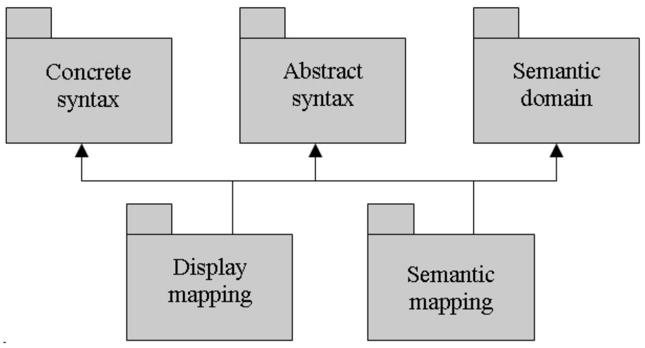

Figure 2: Structure of a language

constraints on the language components, especially for the display mapping between the concrete and the abstract syntax, and the semantic mapping between the abstract syntax and the semantic domain.

One of the key features of MMM is the package specialization. This feature allows partial definitions of model elements in a super package to be then consistently specialized in a sub-package. Although only package specialization has been defined until now, there is no reason why this specialization shouldn't be applied to other elements. This package inheritance can of course be multiple and parametrized through templates.

The templates are parametric model elements. Thus, the template packages are means of representing reusable modeling patterns, by defining a generic template library, and then specializing it for specific needs. The authors of [cla01] wanted UML to become a precisely defined family of languages. That is one reason why the Method for Meta-Modeling provides a framework for defining language families, as a set of template packages can generate many languages — all belonging to the same language family, by using the same package libraries structure.

3.2 Language for Meta-Modeling

The Meta-Modeling Framework also contains a Language for Meta-Modeling. This is a static object-oriented modeling language. Although it is rather small, it is expressive enough to be meta-circular, and to describe itself.

The basic expression language is based on the OCL and supports basic types, including sets and sequences.

3.2.1 Class definition

As one can see in the listing below, a definition in MML is a name and an expression. For example, an attribute is a name (the name of the attribute) and a type, whereas a class definition is a name (the name of the class) and a more complex definition, possibly composed by attributes, constructor, methods and invariant assumptions.

```
class Person
  name : String
  age : Integer
  married : Boolean
  children : Set(Person);
  parents : Set(Person);
  init(s:Seq(Instance)):Person
    self.name := s->at(0)[]
    self.age := s->at(1)[]
    self;
  averageChildAge():Integer
    self.children->iterate(c, a = 0 | a + c.age)
        / self.children->size;
  inv
    IfMarriedThenOver15
      self.married implies self.age >=16;
    OnlyTwoParents
      self.parents->size = 2
end
```

In the example above, the class `Person` is defined with a few attributes, one constructor, one method and two invariants. The constructor is given in the `init` method, with a sequence parameter and, of course, the return type being a `Person`. A sample method to get the average age of the children is also presented, as well as an iteration on the children inside.

Furthermore, a very interesting feature, also presented in this example, is the invariant `inv`, allowing the language engineers to set invariant properties on a language by defining assumptions being always true.

3.2.2 Association definition

One can also easily define associations between language elements. Only binary associations are supported by MML. One has to define the role name of the members of the association, their type, and their multiplicity.

```
association Family
  parents : Person mult: 2
  children : Person mult : *
end
```

In the example presented above, a family is defined as 2 parents (of type `Person`) and children (of type `Person`), the number n of children ranging from 0 to ∞ (* association).

3.2.3 Package definition

Moreover, MML allows to define packages, which are nothing but the composition of classes and associations in the same set.

```
package People
  class Person
    // as given above...
  end;
  association Family
    // as given above
  end
end
```

Above all, MML provides a powerful package specialization mechanism. In other words, a package can inherit from another package, keeping everything that has been defined in the first package, and possibly adding new features or specializing the existing features. Packages can of course inherit many other packages (multiple inheritance).

```
package Employment extends People
  class Person
    yearsInService : Integer
  end;
  class Company
    name : String
  end
  association Works
    company : Company mult : 1
    employees : Person mult : *
  end
end
```

In the example presented above, the class `Person` is specialized, which means it is defined as previously presented, with a new attribute `yearsInService`. The mechanism of package specialization is much more difficult to process than one could first think. Clark, Evans and Kent presented a precise approach for this problem in [cla02].

3.2.4 Template definition

MML also provides a template mechanism, which allows the language engineers to parametrize model elements. The current structure of MML only allows the packages to be defined as templates, but this notion could be extended to other language elements.

```
package Contains(Container,n1,m1,Contained,n2,m2)
  class <<Container>>
    <<n2>>():Set(<<Contained>>)
      self.<<n2>>
    inv
      UniqueNames
        self.<<n2>>->forAll(c1 c2 |
          c1.name = c2.name implies c1 = c2)
  end;
  association <<n1 + n2>>
    <<n1>> : <<Container>> mult: <<m1>>
    <<n2>> : <<Contained>> mult: <<m2>>
  end
end
```

In the example above, a container package is defined. MML allows parameters in templates that can then be used in the definition using << and >>. Thus, the class `<<Container>>` is here extended with a method called `<<n2>>()`, which is a sort of "Get" method giving back the set of contained element, as well as with an invariant checking that every contained element is named differently. We also notice the association between the container and the contained. The multiplicity and the types for this association are defined through the template parameters.

Let us instantiate this template, as shown below.

```
package People extends Container(
    ''Person'', ''children'', *, ''Person'', ''parents'', 2)
  class Person
    // attribute and method definitions
  end
end
```

The result of this instantiation is an extended package `People` where the class `Person` is extended through a `parents()` method, giving back the set of parents, as well as through an invariant checking that the parents are different from each other. The `Childrenparents` association is also created, with 2 parents and n children, $n \in \mathbb{N}$.

This extension mechanism of course occurs internally and the extended package `People` is invisible for the language engineer. However, let us present below what this extended package would be after this template instantiation.

```
 package People
  class Person
    parents():Set(Person)
      self.parents
    inv
      UniqueNames
        self.parents->forAll(c1 c2 |
          c1.name = c2.name implies c1 = c2)
  end;
  association Childrenparents
    children : Person mult: *
    parents : Person mult: 2
  end
end
```

3.3 Tool for Meta-Modeling

The last part of the Meta-Modeling Framework is the Tool for Meta-Modeling (MMT). However, this tool seems not to be public available. The authors described it as a prototype tool written in Java and supporting the MMF approach. This tool should be based on a virtual machine that runs the MML calculus, and should be able to perform some verifications, for example to check well-formedness rules and OCL constraints validity.

3.4 A Small Modeling Language

To better describe the possibilities of this Meta-Modeling Framework, an example will now be introduced.

The Small Modeling Language is a language defined by Clark, Evans and Kent in [cla01] to describe how a language can be engineered using the MMF.

This Small Modeling Language (SML) is a small static modeling language. It contains very simple features, such as packages and classes with attributes, and should be able to model class diagrams. For space reasons, only the abstract syntax, the semantic domain and the semantic mapping between these two domains will be presented in this document.

3.4.1 Template libraries

We first have to define template libraries, which will be instantiated later to concretely model the language. Note that these libraries allow the reusability of the language.

Only one library template will be detailed here, namely the `Named` package, as can be seen below.

```
package Named(Model)
  class <<Model>>
    name : String;
    toString():String
      "<" + self.of.name + self.name + ">"
  end
end
```

As one can easily imagine, a named element shall have a name and a `toString` method.

Other template libraries should be defined. For space reasons, only the name of some of them will be presented below.

```
package NameSpace(Container, Contained)

package Contains(Container, Contained)

package Specializable(Model)

package SpecializableContainer(Container, Contained)
  extends Specializable(Container), Specializable(Contained)

package Relation(Name, Domain, Range)

package RelateAtt(R, Domain, Range, DomainAtt, RangeAtt, Pred)
  extends Relation(R, Domain, Range)

package TypeCorrect(R, Domain, Range)
```

```
  extends RelateAtt(R, Domain, Range, ‘‘type’’, ‘‘value’’, check)

and so on...
```

With these template libraries being defined, we can now concretely define the SML construction.

3.4.2 Abstract syntax

The abstract syntax only consists in the specialization of the different templates presented above. Thus, the package `AbstractSyntax` is defined as inherited from other packages. A small class `Attribute` shall also be added, as shown below.

```
package AbstractSyntax
  extends
    SelfContains("Package"),
    SpecializableContainer("Package","Package"),
    SpecializableContainer("Package","Class"),
    SpecializableContainer("Class","Attribute"),
    Specializable("Attribute"),
    Contains("Package","Class"),
    Contains("Class","Attribute"),
    Clonable("Package","Class"),
    Clonable("Package","Package"),
    Clonable("Class","Attribute"),
    Named("Package"),
    Named("Class"),
    Named("Attribute"),
    NameSpace("Package","Package"),
    NameSpace("Package","Class"),
    NameSpace("Class","Attribute")

  class Attribute
    // some definition
  end
end
```

These template instantiations especially ensure that a package contains classes, and that classes contain attributes.

3.4.3 Semantic Domain

As for the abstract syntax, the `SemanticPackage` only consists of the specialization of different template packages, as presented below.

```
package SemanticDomain
  extends
    SelfContains("Snapshot"),
    Contains("Snapshot","Object"),
    Contains("Object","Slot"),
    Named("Snapshot"),
    Named("Slot")

  class Slot value : Object end
end
```

Here again, these template package specializations especially define containment relations on the elements, and check that they have a name.

3.4.4 Semantic Mapping

Last, the semantic mapping of SML has to be defined in order bind the abstract syntax to the semantic domain, as presented below.

```
package SemanticMapping
  extends
    AbstractSyntax,
    SemanticDomain,
    ContainsInstances1(
      "PackXSnap","Package","Class",
      "ClassXObj","Snapshot","Object"),
    ContainsInstances(
      "ClassXObj","Class","Attribute",
      "AttXSlot","Object","Slot"),
    SameName("AttXSlot","Attribute","Slot")
    TypeCorrect("AttXSlot","Attribute","Slot")
end
```

One can especially note that the classes are bound to objects, and the attributes are bound to slots.

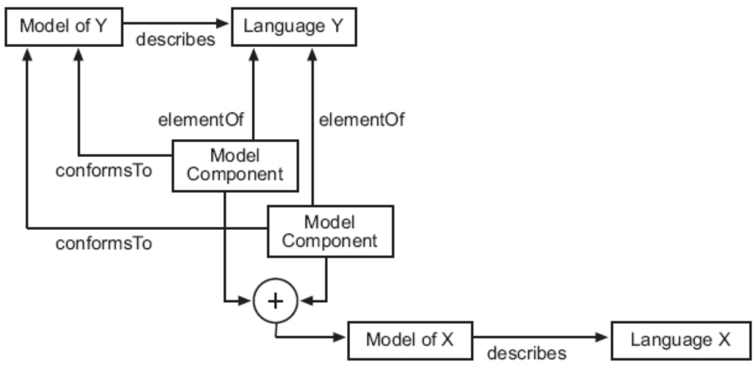

Figure 3: Language composition as composition of model elements

3.4.5 Conclusion on SML

This Small Modeling Language allowed us to see that one can describe a static modeling language very shortly. An important point is the reusability, allowed through the template libraries, which can now be reused for another modeling language. Although the SML is very simple, one can imagine that the Meta-Modeling Framework let us easily build much more complicated languages.

4 Language Composition

In order to perform a better reusability, and of course to reduce development costs, Language Engineering should provide language composition mechanisms. DSL development is the main application field of such a system.

Figure 3 shows the relations between model components and the language, whose sentences are model components. Model components can be defined as an element that might be subject to composition. We can see in figure 3 that the subjects to composition, when realizing language composition, are the models of the language constructs or languages. Indeed, sentences of language Y, which are also conform to the model of Y, are composed together to make a new model. The result of the composition is a model X that describes a new language X.

A parallel can also be seen between Language Engineering and Software Engineering about composition. In Software Engineering, Invasive Software Composition (ISC) is a gray-box composition technique, where components are transformed at previously defined places by composition operators in order to be reused. Consequentially, components are called *fragment boxes*,

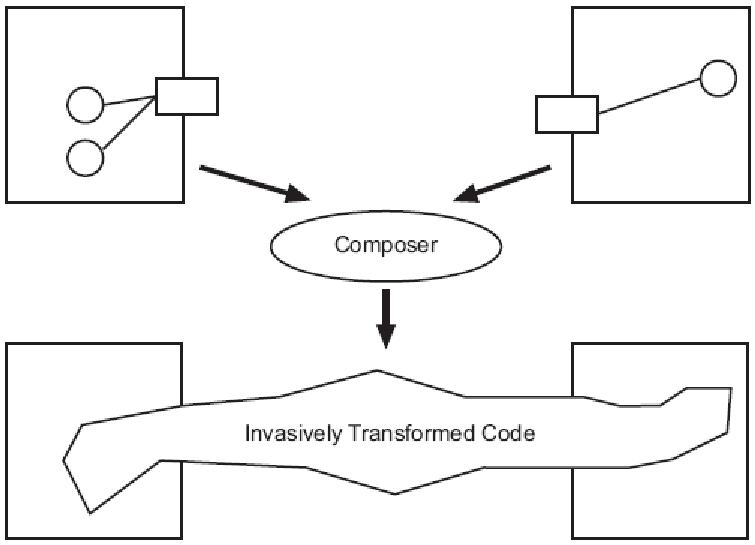

Figure 4: Language composition as composition of model elements

and have a composition interface consisting of a set of *hooks*. A hook is a point of variability of a fragment box, identifying the locations at which variations of a fragment box can occur.

One can see on figure 4 how sets of program fragments are composed to get the transformed code.

This technique can also be applied to Language Engineering, with language fragments. Thus, Invasive Language Composition introduces the concepts of fragment boxes and hooks to a metalanguage by extending a model that describes this metalanguage. Thus, composers need to be identified and implemented for composing sentences of this extended metalanguage.

This mechanism of language composition especially requires package specialization. Clark, Evans and Kent presented a precise approach for this problem in [cla02].

An example of invasive language composition is given in [bur05], where a model that describes the EBNF metalanguage is used as a base and extended to a component model for language specifications called *Comp-EBNF*.

5 Conclusion

The discipline of Language Engineering requires advanced techniques to improve features in the development and the maintainability of languages as well as to reduce costs. This can be done through packages extension, templates specialization, and language composition. Although Language Engineering is an old discipline, these mechanisms shall be improved in order to reach a more practical level, as many of the Language Engineering features are still theoretical.

References

[bez05] Bézivin J., Heckel R. (2005) Language Engineering for Model-driven Software Development

[bur05] Bürger T. (2005) Contributions to language composition using Standard Semantic Web Techniques

[cla00] Clark A., Evans A., Kent S., Brodsky S., Cook S. (2000) A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach

[cla01] Clark T., Evans A., Kent S. (2001) Engineering Modeling Languages: A Precise Meta-Modeling Approach

[cla02] Clark T., Evans A., Kent S. (2002) A Metamodel for Package Extension with Renaming

[dsou98] D'Souza D.F., Wills A.C. (1998) Objects, Components, and Frameworks with UML: The Catalysis Approach

[mda] Model Driven Architecture http://www.omg.org/mda/

[omg] Object Management Group http://www.omg.org/

[wiki] Wikipedia http://en.wikipedia.org/