On Executable Meta-Languages applied to Model Transformations

Pierre-Alain Muller[1], Franck Fleurey[1], Zoé Drey[1], Damien Pollet[1],
Frédéric Fondement[2], Philippe Studer[3]

[1] IRISA/INRIA, France {pierre-alain.muller, franck.fleurey, zoe.drey, damien.pollet}@irisa.fr
[2] EPFL/IC/UP-LGL, INJ, Switzerland, frederic.fondement@epfl.ch
[3] MIPS, Université de Haute-Alsace, France, philippe.studer@uha.fr

Abstract. Domain specific languages for model transformation have recently generated significant interest in the model-driven engineering community. The adopted QVT specification has normalized some scheme of model transformation language; however several different model transformation language paradigms are likely to co-exist in the near future, ranging from imperative to declarative (including hybrid). It remains nevertheless questionable how model transformation specific languages compare to more general purpose languages, in terms of applicability, scalability and robustness. In this paper we report on our specific experience in applying an executable meta-language to the model transformation field.

1 Introduction

A DSL (Domain-Specific Language) is a specification or programming language which offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Model transformation is a key facet of model-engineering, by which models which conform to some metamodels are translated into models which conform to some other metamodels. Technologies to perform model transformations range from conventional programming languages to specific transformation languages.

Domain specific languages for model transformation have recently generated significant interest in the model-driven engineering community. The OMG has adopted the QVT (Query, View, Transformation) specification, which normalizes some scheme of model transformation language.

However, many open issues about transformation languages still remain, and it is likely that several alternative paradigms (such as imperative, declarative or hybrid) will co-exist in the foreseeable future of model transformation languages.

From a software engineering point of view, it is highly desirable to gain a better understanding of how these various kinds of model transformation languages address issues such as applicability, scalability and robustness. Indeed, there is not yet much practical experience with dedicated model transformation languages versus general purpose languages with respect to model transformations, and the optimal scope of domain specific transformation languages remains unclear. For instance, should these model transformation languages be limited to model manipulation in general and be associated to specific libraries, or should they emphasize a specific activity performed with models, such as transformation?

In this paper we report on our experience with executable meta-languages associated to specific frameworks to develop model transformations. Our experience is based on the development and use of three different imperative object-oriented languages for model manipulation, respectively: MTL a transformation language, Xion an action language and Kermeta an executable meta-language.

Although it has made this paper lengthy, we have decided to give substantial excerpts of the source code of the transformations, because we found it relevant in the context of a workshop dedicated to language comparison. Hopefully, the impatient reader may not have to browse through all these examples to understand our approach to model transformation, as we have motivated and summarized our position in the first sections of the paper. The complete sources of the examples can be found on http://www.irisa.fr/triskell/Softwares/kermeta/examples/mtip, from where the Kermeta Workbench can also be downloaded.

This paper is organized as follows: after this introduction, section 1 highlights the rationales for our work and examines some related works, section 2 presents the Xion, MTL and Kermeta languages, section 3 discusses model transformation design options with executable meta-languages, section 4 implements and discusses the workshop case-study, and finally section 5 (the conclusion) summarizes our position and outlines future directions.

2 Rationales for object-oriented executable meta-languages

In the model-driven engineering community, meta-languages such as MOF[1] are widely used to specify meta-models. The issue of persistence is well understood, and is achieved either via a serialization in XML (via XMI, for XML Metadata Interchange), or via direct storage in some database (e. g. MDR[2]).

Yet, existing meta-data languages (including MOF, EMOF[1], ECore[3], MetaGME[4]), as their generic name suggests, are languages for defining data about data. Such data-driven languages focus on structural specifications and have no built-in support for the definition of behavior about these structures. There are mainly two options to work with the metamodels (and models) stored in the repositories:

- Using conventional programming languages, such as Java, via specific libraries which provide facilities to navigate, create, read or delete models and model elements.
- Using domain specific languages, such as action languages, constraint languages and transformation languages.

We believe that the rationale for the current separation between data and behavior specifications at the meta-meta level is mainly coincidental and results from the fact that research works in the field of model-driven engineering have been initially conducted by technical domains; mainly driven by functional user requirements, such as definition of actions, constraints and transformations, with little sharing beyond the data persistence level.

In our opinion, there is now enough understanding of these functional domains to initiate a convergence under the shape of some kind of common denominator of the fundamental current model-driven technologies, i.e. languages for meta-data definitions (such as MOF, EMOF, Ecore), model transformations (including MTL[5] and ATL[6], all more or less QVT[7] compliant), constraint and query expressions (such as OCL[8]) and action specifications (such as the Action Semantics[9], now integrated in UML 2.0).

We claim in this paper that a common kernel of language constructs can be defined to serve all purposes of model manipulations such as definition of metamodels and models, actions, queries, views and transformations. Moreover, making this kernel executable provides direct support to express the operational semantics of metamodels.

This kernel should contain basic instructions to define model structure and elements, manipulate the models and the model elements via fundamental *create*, *read*, *update* and *delete* operations, as well as iterators to navigate models or sets of model elements.

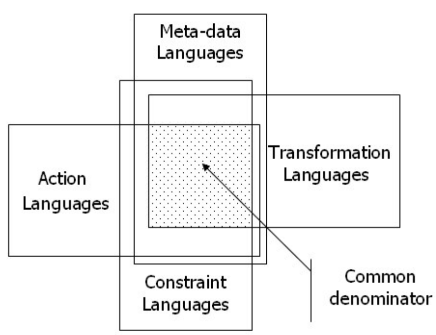

Figure 1: Meta-data, action, transformation and constraint languages share a common subset of language constructs.

2.1 Related works

Our work is related to many other works, and can be considered as some kind of synthesis of these works, in the specific context of model-driven engineering applied to language definition. The sections below include the major areas of related works.

Grammars, graphs and generic environment generators. Much of the concepts behind our work take their roots in the seminal work conducted in the late sixties on grammars and graphs and in the early eighties in the field of generic environment generators (such as Centaur[10]) that, when given the formal specification of a programming language (syntax and semantics), produce a language-specific environment. The generic environment generators sub-category has recently received significant industrial interest; this includes approaches such as Xactium[11], or Software Factories[12]. Among these efforts, it is Xactium which comes closer to our work. The major difference is our adherence to OMG standards (such as EMOF) and the fact that we have a fully static type system.

Generative programming and domain-specific languages. Generative programming aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in a domain-specific language. This includes multi-purpose model-aware languages such as Xion or MTL[5], or model transformation languages such as QVT[7].

We share the vision of generative programming, and we use models to generate fully executable code which can be compiled. The Xion and MTL languages have had a direct impact on our work.

QVT is different as it addresses mappings between models. QVT works on structures, by specifying how one structure is mapped into another one; for instance translating a UML class diagram into a RDBMS schema. QVT is not suitable for the definition of the behavior of metamodels.

2 Overview of Xion, MTL and Kermeta

Xion and MTL are the ancestors of the Kermeta language. Xion is a platform independent action language which has been originally developed in the context of the Netsilon environment, for model-driven development of Web information systems[13]. MTL is an object-oriented model transformation language, which has been developed with software engineering concerns in mind, such as robustness, modularity and scalability. Interestingly, the two teams which have developed Xion and MTL independently have come to the same kind of conclusions. They have both developed a general-purpose, imperative, object-oriented language, with model-navigation capabilities based on OCL, control structures such as found in Java or Eiffel, and with model management capabilities.

The lessons learned with Xion and MTL have shaped the requirements of Kermeta. Kermeta is a multi-faceted language. Kermeta is a small imperative object-oriented language, which is both an executable meta-langage and a kernel upon which to build other languages (such as Xion or MTL which could be re-expressed in Kermeta). Although Kermeta is a small language, it provides high-level mechanisms such as static-typing, genericity and exceptions.

2.1 Xion

Xion is a general-purpose object-oriented action language, with special support for model manipulation, and automatic persistence of model elements. Xion is a platform-independent action language which abstracts away the details of data access, while being translatable into different target languages (such as PHP or Java).

Xion provides modeling concepts such as classes (with attributes, operations and methods), associations and aggregations, class-associations, and simple generalizations. Xion is a semi-graphical language, classes, attributes, operations and relations are defined via class diagrams; add- and remove-link operations are generated automatically. User-defined methods are specified in text.

In the context of this case study, Xion is used as an executable meta-langage, in which case the metamodels are expressed in terms of classes and relations, in a manner very similar to what is done with MOF or ECore.

Xion provides support to query models and to express methods and state changes via an extension of the OCL *query expressions*. This means adding side-effects capability to OCL, and providing imperative constructs, such as blocks and control flows. Supporting side-effects means:

- create and delete an object,
- change an attribute value,
- create and delete links,
- change a variable value,
- call non-query operations.

It was also necessary to remove some constructs of the OCL, which are out of the scope of our approach:

- context declaration, only useful for defining constraints,
- @pre operator and message management, only meaningful in the context of an operation post-condition,
- state machine querying.

Since most developers are already familiar with the Java language, we re-used part of its concrete syntax. Constructs we took from Java are:

- instruction blocks, i.e. sequences of expressions,
- control flow (if, while, do, for),
- return statement for exiting an operation possibly sending a value,
- "super" initializer for constructors.

Moreover, for Xion to look like Java as much as possible we decided to keep Java variable declaration, and operators (==, !=, +=, >>, ? ternary operator, etc.) rather than those defined by OCL. The standard OCL library was also slightly extended, by adding the `Double`, `Float`, `Long`, `Int`, `Short` and `Byte` primitive types, whose size is clearly defined unlike the OCL `Integer` or `Real`. As applications often deal with time, we have also added the `Date` and `Time` predefined types. An exhaustive presentation of the language is given in the help of the Netsilon tool.

2.2 MTL

MTL (Model Transformation Language) shares much of the description of Xion given beyond, in terms of abstract (and even concrete) syntax. A major difference is the requirement of MTL to be tool independent. Whereas Xion was emphasizing automatic object persistence, MTL seeks model (and metamodel) representation independence, and can interact in a unified way with various model repositories such as MDR (MetaData Repository, Sun), ModFact[14] (Lip6) and EMF (Eclipse Modeling Framework, IBM). For instance, a MTL transformation written for a given model, can then be applied to that model independently of the language (say MOF or ECore) used to express the metamodel of that given model.

For practical reasons (vacations), a separate submission to the MTIP workshop (by Didier Vojtisek and Jean-Marc Jézéquel) presents MTL in more details.

2.3 Kermeta

Kermeta has been designed to be the core language of a model-oriented platform[15]. Kermeta, as shown in Figure 1, can be considered as a common denominator of several model-oriented technologies.

Kermeta consists of an extension to the Essential Meta-Object Facilities (EMOF) 2.0**Erreur ! Signet non défini.** to support behavior definition. It provides an action language to specify the body of operations in metamodels. The action language of Kermeta is imperative and object-oriented.

Kermeta has been defined based on the experience of two existing languages Xion and MTL. Xion is an action language for UML class diagrams, it is used to provide a high level platform independent implementation of operations and methods. The Netsilon tool is used to generate either java or PHP code from Xion code.

MTL (Model Transformation Language) is an object-oriented model transformation language. It provides APIs to allow manipulating models from various repository (Eclipse EMF, Netbeans MDR...) in an unified manner.

Xion and MTL have been designed for different purposes but they share many constructions such as expression for querying model or CRUD operations on objects. Kermeta is mainly an object-oriented language which includes features such as multiple inheritance, operation redefinition, class genericity and dynamic binding. However, for model processing, specific construction were added to handle model specific features such as associations and object containment. In addition to these, and for usability purposes, convenient model navigation expressions such as OCL iterators (select, collect, reject…) have been added. The resulting language is fully statically typed to ensure strong reliability concerns.

The Kermeta platform is developed at INRIA as an open-source project[16]. It currently includes a parser, a type-checker and an interpreter. It is distributed as en Eclipse plug-in which includes an editor for Kermeta programs with syntax coloring and code completion capabilities. In addition the platform includes libraries to load and store models from the Eclipse Modeling Framework and to import ECore metamodels.

3 Discussion

In this section, we will cover the points of discussion that were listed in the call for papers, and some others that we find relevant in the context of Kermeta.

Object-orientation

Kermeta is an object-oriented language, which provide support for classes and relations, multiple inheritance, late binding, static typing, class genericity, exception, typed function objects… The object-oriented nature of Kermeta has a double origin. First, as for Xion which is based on UML, the structural part of the Kermeta language is based on an object-oriented meta-data language (EMOF). Next, as for MTL, there is a strong requirement for Kermeta to support software engineering good-practices such as modularity, testability and reuse, which are well supported by object-orientation.

The refactorings optional part of the case study is a good example of how object-oriented techniques, such as patterns, may be applied to model transformations. The Kermeta implementation shows the use of a command patterns to apply a transformation. Interestingly, this also provide an example of doing and undoing a transformation.

Finally, object-orientation eases the learning of the new language, as many developers are used to that paradigm, and can immediately apply their programming skills in the context of model transformation.

Composition of transformations

Kermeta provides packages, classes, operations and methods, inheritance and late bindings. All these features can be used to encapsulate transformations. Composition of transformations can be achieved in several ways, for instance by operations calls or method overloading. Rule recursivity is handled by function recursivity.

Robustness and error handling.

Reliability is a major concern in the design of Kermeta. The language is statically typed, and the code can be fully checked for correctness at compilation time. For unexpected behavior at runtime, the language provides exception handling.

Debugging support

The debugger is under development. This is not part of the language itself, but of the Kermeta Workbench. In the meantime, traces are used to help solving problems in the transformation code.

Flexibility, overall usability and power of the chosen approach

To our opinion, Kermeta is currently a very good compromise between a general-purpose language such as Java, and a specific model transformation language such as specified by QVT. The implementation of the workshop case study, given below, proofs that the language is usable for model transformation, and also that is can handle all kinds of transformations (either specified via mappings as for the class to RDBMS example, or via algorithms as for the determinization/minimization of automata).

Whether the approach can express bidirectional and / or incremental (sometimes known as change propagating) transformations

This is really a matter of programming. The Arabic and Roman numbers, and the refactorings, are examples of bidirectional, and change propagating transformations.

Technical aspects such as the ability to deal with model exchange formats, modeling tool APIs, and layout updates

Kermeta is fully compatible with ECore. The structural part of Kermeta is compliant to ECore, and the behavioral part is expressed as an ECore metamodel. Thus, any tool compatible with ECore is compatible with Kermeta. Kermeta can read and write ECore files to load and save models. The Kermeta workbench is available as an Eclipse plugin.

General purpose language vs. model transformation language

Kermeta has nothing specific to model transformation, but being quite general purpose, it can be used to implement mechanisms to support model transformations. As with general-purpose languages such as Java, specific support (for instance for transformations triggering, trace and debugging) is added via libraries and frameworks. This gives extensive expression freedom to the developer to write the transformations.

For instance, in the class to RDBMS example, information has to be stored in one pass to be used in another one. The Xion example also shows how part of the transformation are queued and triggered later on by other parts, for instance to query the target model when transforming the associations.

Design variations, libraries vs. DSLs

Writing a transformation can be done is many different ways. A final design reflects a set of tradeoffs made by the developer. The variation of the designs may be more or less constraint by the amount of pre-design and reuse provided by the language environment. Libraries and frameworks can be used to provide specific capabilities, such as traceability or information storage about the transformation process itself.

Such pre-design decisions can be captured either in libraries or in the language itself, and this is the difference between libraries for general purpose languages, and DSLs.

Kermeta has precisely been built to facilitate such transition, when the domain knowledge is such that it is worth to embed this knowledge directly in a dedicated language[17]. Kermeta can be used to represent the abstract syntax of languages, under the shape of metamodels. Then, Kermeta being executable, the operational semantic of these languages can be further specified in Kermeta, reusing the library code which was previously developed.

Software engineering concerns

Since model transformation may become quite complex, we believe that transformation developers need to re-use popular know-how and best practices in software engineering. Kermeta provides language support for modularity in the small (classes) and the large (packages), reliability (static typing, typed function objects and exception handling), extensibility and reuse (inheritance, late binding and genericity).

4 MTIP workshop case studies

This section presents the transformations that we have written to implement the requirements of the MTIP workshop case study. Sub-sections 4.1 and 4.2 present the mandatory transformations, respectively in Xion and Kermeta (notice that a MTL implementation is submitted to the MTIP workshop, in a separate paper), then section 4.3 shows the optional example of Roman and Arabic numbers in Kermeta, section 4.4 gives an example of refactoring written in Kermeta, and finally section 4.5 presents the determinization and minimization of automata in Kermeta.

4.1 Mapping classes to tables in Xion

The following metamodels are part of the MTIP case study. They have been represented visually with the Netsilon class diagram editor. Figure 2 and Figure 3 show respectively the source and target metamodels of the MTIP workshop case study. Notice that various operations have been added to the classes. Adding these operations directly in the classes is questionable. On one hand, it promotes encapsulation and gives an object-oriented flavor to the transformation; on the other hand it establishes some coupling between the static aspect of the metamodel and a specific transformation. Whether this is good or bad depends heavily on the context; in this example in Xion we chose the former option. In the following example in Kermeta we will choose the latter.

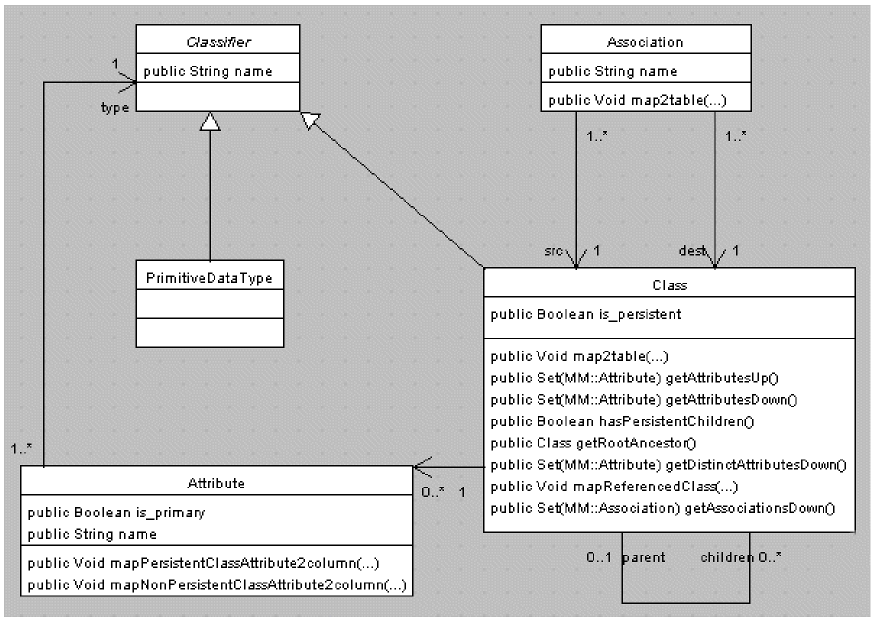

Figure 2: Source metamodel expressed in Xion.

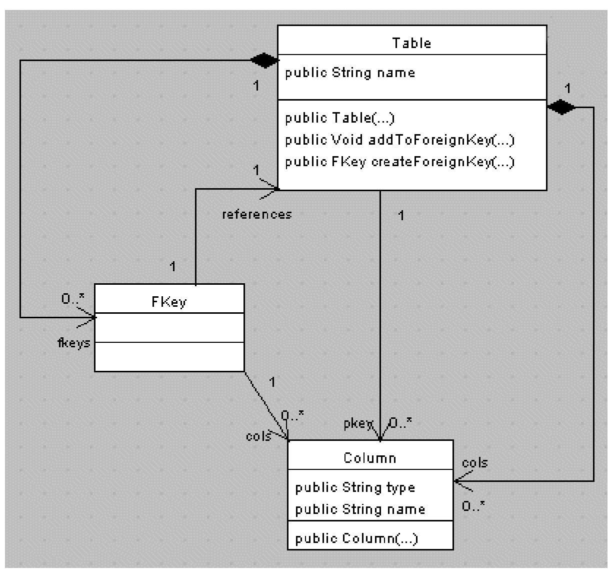

Figure 3: Target metamodel expressed in Xion.

In addition to the classes defined in the case study, we have also defined two utility classes `Transformation` and `ClassTransformation`, to store information about the transformation process itself.

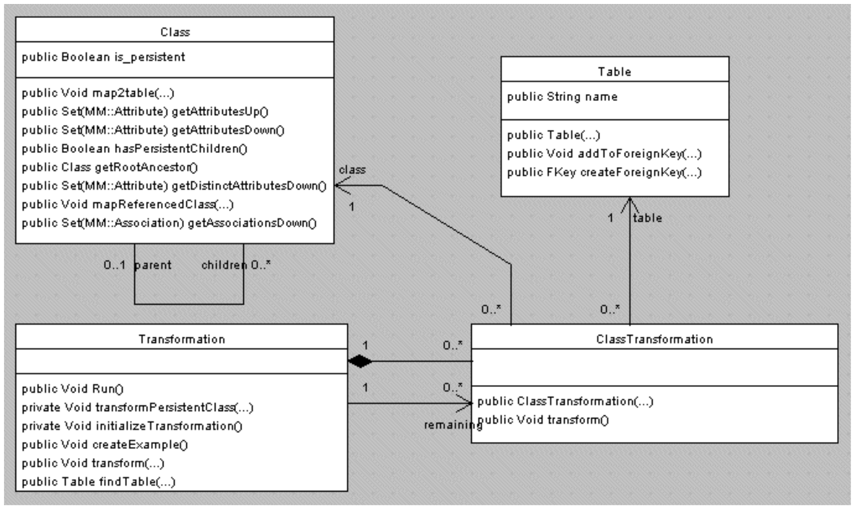

Figure 4: Specific Xion classes to support transformations.

The transformation process (initiated when the `Run` method of the `Transformation` class is invoked) first traverses all the classes in the source model, and creates a `ClassTransformation` instance (a pair `Class`, `Table`) for each class which has to be made persistent. In a second step, the `Transform` method of class `ClassTransformation` creates the columns required to ensure persistence of all the attributes and relations present in the source model.

Transformation Class

The `Transformation` class contains the top-level transformation methods. The process of transformation starts when the `Run` method is invoked.

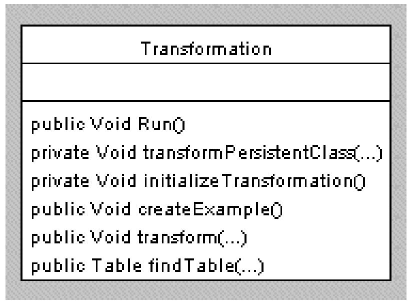

Figure 5: Specification of the Transformation Class.

For all class hierarchy which contain persistent classes, a table is created for the root class. The `allInstances` operation retrieves the collection of all the instances of the `Class` class. Xion is a persistent language; all the instances are automatically stored in permanent storage.

The following Xion expression, creates a table for all topmost classes which are either persistent, or have at least one persistent subclass. All the metamodel classes are defined in a package named MM (for metamodel).

```
public Void Transformation::Run (Class class, Table table) {
   MM::Class.allInstances()
      ->select(parent == null)
      ->select(c : c.is_persistent || c.hasPersistentChildren())
      ->collect(c : transformPersistentClass(c));

   this.classTransformation->collect(t : t.transform());
}
```

with

```
private Void Transformation::transformPersistentClass (Class class) {

   MM::ClassTransformation t =
      new MM::ClassTransformation(class, new MM::Table(class.name));

   this.addclassTransformation(t);
   this.addremaining(t);
}
```

Class Class

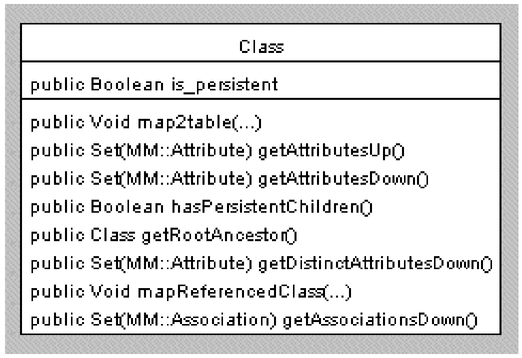

Figure 6: Specification of the Class Class.

Rule 6 states that attributes in subclasses with the same name as an attribute in a parent class are considered to override the parent attribute. The operation `getDistinctAttributesDown` builds the set of attributes within a class hierarchy, while removing those overridden in subclasses. The operation starts from a root class, and then collects recursively down the attributes defined in subclasses.

```
public Set(MM::Attribute) Class::getDistinctAttributesDown ()
{

   if (this.children == null)
      return this.attribute;

   Set(MM::Attribute) children =
      this.children.getDistinctAttributesDown()->asSet();

   return this.attribute
      ->select(a : !children->exists(c : c.name == a.name))
      ->union(children);
}
```

According to the CFP, there is not association overriding.

```
Public Set(MM::Association) Class::getAssociationsDown()
{
   Set(MM::Association) associations =
      MM::Association.allInstances()->select(a : a.src == this);

   if (this.children == null)
      return associations;
   else
      return associations
         ->union(this.children.getAssociationsDown()->asSet());
}
```

Rule 6 states that when transforming a class, all attributes of its parent class (which must be recursively calculated), and all associations which have such classes as a src, should be considered.

Attributes in subclasses with the same name as an attribute in a parent class are considered to override the parent attribute.

```
public Void Class::map2table (Table t, Transformation transformation)
{

  self.getDistinctAttributesDown().mapPersistentClassAttribute2column
                                              (this, t, transformation);

  this.getAssociationsDown().map2table(this, t, transformation);

}


public Void Class::mapReferencedClass( Class sourceClass,
                                      Table sourceTable,
                                      String namespace,
                                      Transformation transformation)
{
  // Be sure that the attributes have already been transformed
  transformation.transform(this);

  // 2. Classes that are marked as non-persistent should not be
  // transformed at the top level.
  if (!this.is_persistent)
  {
     this.attribute
        ->collect(a : a.mapNonPersistentClassAttribute2Column
                        (sourceClass, sourceTable, namespace));
  }
  else
  {
     // Retrieve primary attributes
     MM::Table table = transformation.findTable(this);

     Sequence(MM::Column) columns = table.pkey->asSequence();

     if (columns->size() > 0)
     {
        MM::FKey fkey = sourceTable.createForeignKey(table);
        for (Integer i = 0; i < columns->size(); i++)
        {
           sourceTable.addToForeignKey(fkey,
                                   new MM::Column
                                     (namespace + "_" +
                                         columns->at(i).name,
                                      columns->at(i).type));
        }
     }
  }
}
```

Attribute class

Two operations have been added to the Attribute class. These operations are responsible for mapping an attribute to a column.

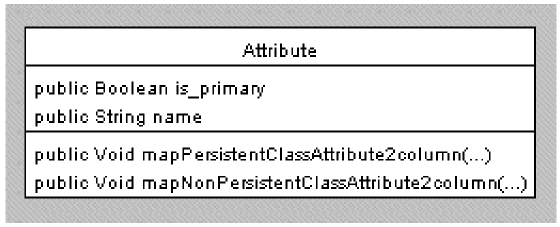

Figure 7: Specification of Attribute class.

The `mapNonPersistentAttribute2Column` operation is partially implemented. In the case of a primitive type, a column is created, with renaming, and primary key property if required. Other cases would to store information about attributes and associated columns, and were not implemented, for lack of time ressource.

```
Void Attribute::mapNonPersistentAttribute2Column(Class sourceClass,
Table sourceTable, String namespace)
{
   if (self.type.oclIsTypeOf(MM::PrimitiveDataType))
     {
     // Rule 3
     // Attributes whose type is a primitive type should be transformed
     // to a single column whose type is the same as the primitive type

     MM::Column column = new MM::Column (namespace + "_" + self.name,
                                         self.type.name);

     sourceTable.addcols(column);

     if (self.is_primary)
        sourceTable.addpkey(column);
  }

  else if (self.type.oclAsType(MM::Class).is_persistent)
  {
     // To Do
  }
  else if (self.type.oclAsType(MM::Class).is_persistent == false)
  {
     // To Do
  }
  else
  {
     // should raise an error
  }
}
```

```
Void Attribute::mapPersistentClassAttribute2Column
                                     (Class class,
                                      Table t,
                                      Tranformation transformation)
{
  if (self.type.oclIsTypeOf(MM::PrimitiveDataType))
  {

     // Rule 3
     // Attributes whose type is a primitive type should be transformed
     // to a single column whose type is the same as the primitive type

     MM::Column column = new MM::Column (self.name, self.type.name);
     t.addcols(column);
     if (self.is_primary)
        t.addpkey(column);
  }

  else if (self.type.oclAsType(MM::Class).is_persistent)
  {
     // Rule 4
     // Attributes whose type is a persistent class should be
     // transformed to one or more columns, which should be created
     // from the persistent classes' primary key attributes.
     // The column should ne named name_transformed_attr where name
     // is the attributes' name.
     // The resultant columns should be marked as constituting a
     // foreign key; the FKey element created should refer to the table
     // created from the persistent class

     MM::FKey fk = t.createForeignKey(MM::Table.allInstances()
       ->select
        this.type.oclAsType(MM::Class).getRootAncestor().name == name)
           ->getOne());

     self.type.oclAsType(MM::Class).getAttributesUp()
        ->select(is_primary)
        ->collect(pk : t.addToForeignKey(fk,
                                          new MM::Column
                                           (pk.name+"_transformed_attr",
                                            pk.type.name)));
  }

  else if (self.type.oclAsType(MM::Class).is_persistent == false)
  {
     // Rule 5
     // Attributes whose type is a non-persistent class should be
     // transformed to one or more columns, as per rule 2.
     // Note that primary keys and foreign keys of the translated
     // non-persistent class need to be merged into the appropriate
     // table

     self.type.oclAsType(MM::Class).mapReferencedClass(class,
                                                        t,
                                                        name,
                                                        transformation);

  }

  else
  {
     // should raise an error
  }
}
```

4.2 Mapping classes to tables in Kermeta

This section details step by step the implementation of the mandatory model transformation for the workshop in Kermeta.

The metamodels

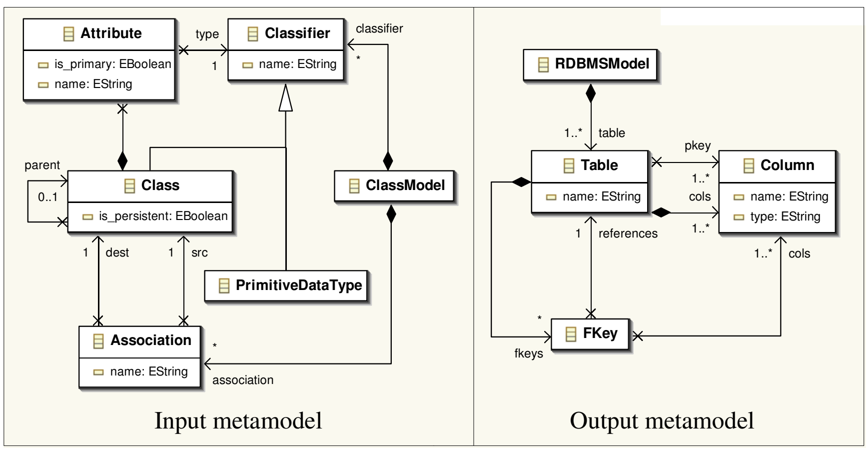

Figure 8: Input and output metamodels expressed in Kermeta visual syntax

Within the Kermeta environment, the first step for implementing a model transformation is to provide the input and output metamodels. As Kermeta relies on the Eclipse Modeling Framework (EMF) for model storage, regular EMF metamodels cans be used: ECore files. These metamodels can be created and edited using the generic model editor provided with the EMF. The Omondo UML tool[18] provides a graphical editor for ECore metamodels. Figure 1 displays the Class metamodel and the Database metamodel as it has been defined using Omondo UML.

There are two different ways of using an ECore metamodel in a Kermeta program.

- This first is to import directly the ECore metamodel in the Kermeta program. This is the simplest manner as it provides the ability to manipulate in Kermeta instances of the classes of the metamodel.
- The second possibility is to generate from the ECore file the metamodel in Kermeta. This is especially useful to add behaviors to the metamodel, which is one of the key features of Kermeta.

For the implementation of the `Class2RDBMS` transformation we have chosen, in order to present the two approaches, to use the input metamodel directly and to generate the Kermeta metamodel for the RDBMS metamodel. Figure 2 presents the Kermeta code generated from the RDBMS metamodel.

```
package RDBMSMM;

require kermeta
using kermeta::standard

class Table
{
	attribute name : String
	attribute cols : Column[1..*]
	reference pkey : Column[1..*]
	attribute fkeys : FKey[0..*]
}
class FKey
{
	reference references : Table
	reference cols : Column[1..*]
}
class Column
{
	attribute name : String
	attribute type : String
}
class RDBMSModel
{
	attribute table : Table[1..*]
}
```

Figure 9: RDBMS metamodels expressed in Kermeta text.

Design of the transformation

Once the metamodels for the inputs and outputs of the transformation are provided, the next step is, as for any software development, to design the transformation. The design stage is more important here, as we use a general-purpose language, than if we were using a language dedicated to model transformation.

In effect, a formalism dedicated to model transformation would provide specific ways of writing transformation where as with Kermeta the transformation is simply an object-oriented program that manipulates model elements.

The proposed transformation generates tables from classes marked persistent. For attributes and associations in these classes it generates columns and foreign keys in the tables. The transformation can be implemented in tree steps:

- Create tables. Tables are created from each class marked persistent in the input model.
- Create columns. For each persistent class process all attributes and outgoing associations to create corresponding columns. The foreign keys are created but the *cols* property cannot be filled and the corresponding columns cannot be created because primary keys of *references* table cannot be known before it has been processed.

- Update foreign-keys. The foreign-key columns are created in the table that contains the foreign-key and the property *cols* of foreign-keys is updated.

Between step 1 and 2 a trace information should be kept between persistent classes and created tables. Keep mappings from source objects and target objects is a general model transformation need. In Kermeta this can be done using several techniques. The first is to use an ad-hoc data structure such as a *Map* to store the correspondence. This would be the simplest solution but as traceability is a common feature to model transformations, it might be interesting to design a reusable solution to handle management of trace information. Kermeta provide for such purposes facilities such as generic classes and operations in order to make the reuse of generic frameworks safe and easy.

Figure 3 presents the implementation of a very simple reusable trace utility. It can be used to represent a one to one mapping between two types of objects. In the implementation of the Class2RDBMS transformation we will use it to store the mapping between persistent classes and generated tables. The *Trace* class is defined a a generic class with two type parameters *SRC* and *DST* that should be bound with the type of the source and target objects. The implementation of the class is very simple and consists of using a *Hashtable* (available from the Kermeta standard library) to store the mapping between objects.

This simple traceability capability is enough for the implementation of the `class2RDBMS` case study but in practice the traceability framework should be enriched to adapt to most model transformation issues. We have already identified several needs for such a framework such as the ability to store bi-directional mappings, on to many or many to many mappings. In addition to the problem of storing object mappings, traces for transformations from model to text or text to model should also be taken into account.

```
package trace;

require kermeta
using kermeta::utils

/**
 * This class represents a simple one to one
 * unidirectional mapping
 */
class Trace<SRC, TGT>
{
        /** Mapping between source and target objects */
        reference src2tgt : Hashtable<SRC, TGT>

        operation create() is do
                src2tgt := Hashtable<SRC, TGT>.new
        end

        /** get a target element */
        operation getTargetElem(src : SRC) : TGT is do
                result := src2tgt.getValue(src)
        end

        /** Store a trace */
         operation storeTrace(src : SRC, tgt : TGT) is do
                src2tgt.put(src, tgt)
         end
}
```

Figure 10: Very simple traceability framework.

Another design issues when writing a model transformation in Kermeta is how the code of the transformation is encapsulated. One of the important feature of Kermeta is to allow adding code directly in metamodels. This is especially interesting to allow sharing queries and common behaviors between several transformations that operates on the same metamodel. However, non-reusable transformation specific features should not be added to the metamodel. To do so, transformation specific classes can be defined directly in Kermeta to contain the implementation of the transformation. Traditional OO design techniques should be used to design these classes.

As the Class2RDBMS example is pretty simple, we have chosen to implement it in a Class2RDBMS class and to include a helper method in the RDBMS metamodel to handle the proper creation of foreign-keys (step 3 of the algorithm). As a result, the Class2RDBMS contains a few query methods on class model that could have been integrated in the Class metamodel to be reused by other transformations. The next section details the implementation of the transformation.

Implementation of the transformation

The transformation has been implemented in a class named *Class2RDBMS*. This class provides a method *transform* that takes the input model as a parameter and returns the corresponding output model. Figure 4 presents an excerpt of the Kermeta code of the transformation.

```
package Class2RDBMS;

require kermeta     // The kermerta standard library
require "trace.kmt"                  // The trace framework
require "../metamodels/ClassMM.ecore"  // Input metamodel in ecore
require "../metamodels/RDBMSMM.kmt"    // Output metamodel in kermeta

[...]

class Class2RDBMS
{
  /** The trace of the transformation */
  reference class2table : Trace<Class, Table>
  /** Set of keys of the output model */
  reference fkeys : Collection<FKey>

  operation transform(inputModel : ClassModel) : RDBMSModel is do
    // Initialize the trace
    class2table := Trace<Class, Table>.new
    class2table.create
    fkeys := Set<FKey>.new
    result := RDBMSModel.new
    // Create tables
    getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
      var table : Table init Table.new
      table.name := c.name
      class2table.storeTrace(c, table)
      result.table.add(table)
    }
    // Create columns
    getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
      createColumns(class2table.getTargetElem(c), c, "")
    }
    // Create foreign keys
    fkeys.each{ k | k.createFKeyColumns }

  end

 [...]
}
```

Figure 11: The three steps of the transformation

The three steps of the transformation clearly appear in the body of operation transform. First, classes are created for each persistent class, second columns are created in the tables and finaly the foreign keys are updated. The mapping between classes and tables is represented by the reference *class2table* in class *Class2RDBMS*. The *fkeys* reference is used to store all created foreign keys during step 2 in order to be able to update them at step 3.

```
operation createColumns(table : Table, cls : Class, prefix : String) is
do
  // add all attributes
  getAllAttributes(cls).each{ att |
    createColumnsForAttribute(table, att, prefix)
  }
  // add all associations
  getAllAssociation(cls).each{ asso |
    createColumnsForAssociation(table, asso, prefix)
  }
end

operation createColumnsForAttribute(table : Table, att : Attribute, prefix : String) is
do
  // The type is primitive : create a simple column
  if PrimitiveDataType.isInstance(att.type) then
    var c : Column init Column.new
    c.name := prefix + att.name
    c.type := att.type.name
    table.cols.add(c)
    if att.is_primary then table.pkey.add(c) end
  else
    var type : Class  type ?= att.type
    // The type is persitant
    if isPersistentClass(type) then
      // Create a FKey
      var fk : FKey init FKey.new
      fk.prefix := prefix + att.name
      table.fkeys.add(fk)
      fk.references:=class2table.getTargetElem(getPersistentClass(type))
      fkeys.add(fk)
    else
     // Recusively add all attrs and asso of the non persistent table
     createColumns(table, type, prefix + att.name)
    end
  end
end
```

Figure 12: Implementation of step 2, columns creation.

Figure 12 presents the implementation of method *createColumns* and *createcolumnsForAttribute*. The *createColumns* operation creates the columns in a table by adding columns for all attributes of the class and all outgoing association from the class. The operations `getAllAttribute(Class)` and `getAllAssociation(Class)` are defined to get all the attributes and outgoing association of a class and all its subclasses.

The operation *createcolumnsForAttribute* handles the creation of columns corresponding to an attribute. Three cases have to be considered:

- If the type of the attribute is simple a single column is created.
- If type of the attribute is persistent, a foreign key is created and the columns in the table will be created at step 3 when all table have been processed.
- If type of the attribute is non-persistent then columns corresponding to attribute in the non-persistent type are added in the table. This is done by a recursive call to method *createColumns.*

The operation *createcolumnsForAssociation* handles the creation of columns corresponding to an association. This operation is not detailed on the figure as it is very similar to *createcolumnsForAttribute* except that the destination type of association cannot be a simple type.

```
class FKey
{
 reference references : Table
 reference cols : Column[1..*]

 /**
  * prefix for the name of the columns
  * used by the createFKeyColumns method
  */
 attribute prefix : String

 /**
  * Create the FKey columns in the table
  */
 operation createFKeyColumns() is do
  var src_table : Table
  src_table ?= container
  // add columns
  references.pkey.each{ k |
   var c : Column init Column.new
   c.name := prefix + k.name
   c.type := k.type
   self.cols.add(c)
   src_table.cols.add(c)
  }
 end
}
```

Figure 13: Implementation of step 3, updating foreign keys.

Figure 13 presents the implementation of step 3. The code has been directly added to the RDBMS metamodel in the class *FKey*. An attribute *prefix* has been added to the class to store the name prefix of the columns to create. When a Kermeta metamodel is generated from an ECore metamodel, any property or operation can be added. The added properties can be considered as “non-persistent” because as they are not in the ECore metamodel the will not be saved when a model is serialized using EMF.

Testing / using the transformation

This section briefly presents how the transformation can be practically used within the Kermeta environment. As Kermeta if fully compatible with the EMF, models can be created, modified and visualized using EMF generic tools. Figure 14 is a screen-shot of an input model for the transformation. Figure 15 displays the Kermeta workbench which has been developed as an eclipse plug-in. finally Figure 16 displays the output model obtained by running the transformation.

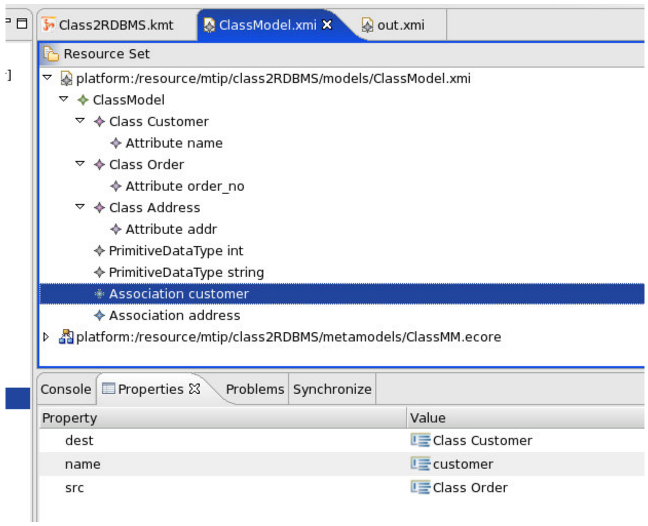

Figure 14: An input model.

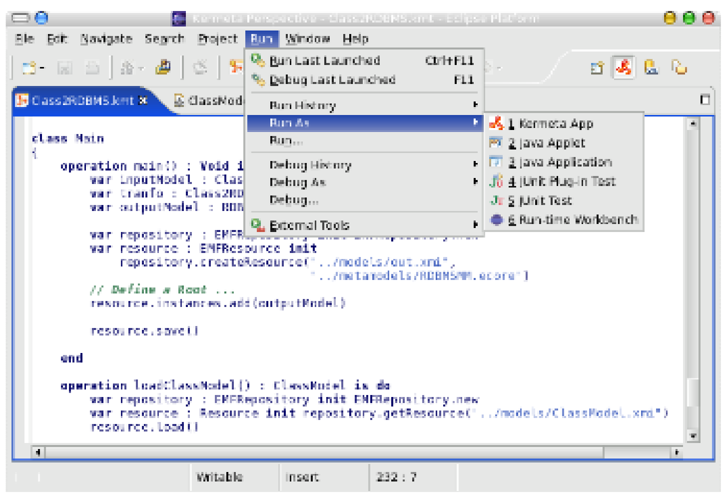

Figure 15: Execution of the transformation.

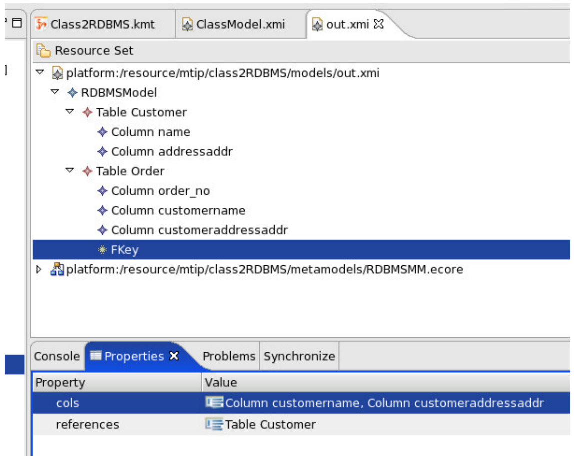

Figure 16: Generated output model.

4.3 Conversion of Roman to/from Arabic numbers in Kermeta

This section presents the implementation of the optional model transformation from Arabic to Roman number and vice-versa. The transformation has been implemented in Kermeta in both directions. The following presents the metamodel that has been used to represent Arabic and Roman numbers, and then details the implementation of the transformation itself.

The metamodels

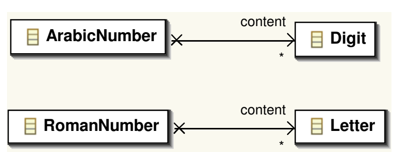

Figure 18: Visual representation of the metamodels

An Arabic number simply consists of a collection of digits and a Roman number in a collection of letters.

```
class ArabicNumber
{
 reference content : Digit[0..*]
 operation toString() : String is do
   result := “”
   content.each{digit |
     result := result + digit.~value.toString()
   }
 end

 operation getValue() : Integer is do
   result := 0
   content.each{n | result := result*10 + n.~value}
 end

 // precondition newValue < 10000
 operation setValue(newValue : Integer) is do
   [...]
 end
}

class Digit
{
 attribute ~value : Integer
}
```

Figure 19: Executable metamodel of arabic numbers in Kermeta.

```
class RomanNumber
{
 reference content : Letter[0..*]
 operation toString() : String is do
     result := String.new()
   content.each{letter | result := result + letter.~value}
 end
 operation getValue() : Integer is do
   [...]
 end
}

class Letter
{
 attribute ~value : String
 operation getValue() : Integer is do
   if value == "I" then result := 1
   else if ~value == "V" then result := 5
   else if ~value == "X" then result := 10
   else if ~value == "L" then result := 50
   else if ~value == "C" then result := 100
   else if ~value == "D" then result := 500
   else if ~value == "M" then result := 1000 end
   end end end end end end
 end
}
```

Figure 20: Executable metamodel of roman numbers in Kermeta.

The transformation

Figure 21 presents the implementation of the transformation in Kermeta. The method *roman2arab* is very straightforward as the roman number metamodel contains already a method *getValue* which computes the integer value of a roman number. The method *arab2roman* is designed to transform arabic numbers lower than 3999.

```
class Main
{
 // convert a roman number to an arabic one
 operation roman2arab(r : RomanNumber) : ArabicNumber is do
     result := ArabicNumber.new()
   result.setValue(r.getValue())
 end
 // convert an arab number to a roman one
 // precondition : a < 3999
 operation arab2roman(a : ArabicNumber) : RomanNumber is do
      result := RomanNumber.new()
      var position : Integer init a.content.size
      // assertion: position <= 4
      if position == 4 then
      addDigit2roman(result,a.content.elementAt(0), "M"," overflow"," overflow")
      position := position - 1
    end
      if position == 3 then
      addDigit2roman(result,a.content.elementAt(a.content.size-3), "C","D","M")
      position := position - 1
    end
      if position == 2 then
      addDigit2roman(result,a.content.elementAt(a.content.size-2), "X","L","C")
      position := position - 1
    end
      if position == 1 then
      addDigit2roman(result,a.content.elementAt(a.content.size-1), "I","V","X")
    end
 end
 // convert a single digit to roman style, depending on its position
 operation addDigit2roman(r: RomanNumber, d : Digit, unit : String, five : String, ten : String) is do
   if d.~value < 4 then addLetters(r,d.~value, unit)
   else if d.~value == 4 then addLetters(r,1,unit)
     addLetters(r,1,five)
   else if d.~value < 9 then do
     addLetters(r,1,five)
     addLetters(r,d.~value-5, unit)
     end
   else if d.~value == 9 then do addLetters(r,1,unit)
     addLetters(r,1,ten) end
   end end end end
 end
 // add letter 'l' 'times' times
 operation addLetters(r: RomanNumber, times : Integer, l : String) is  do
   var aLetter : Letter init Letter.new()
   aLetter.~value := l
   from var i : Integer init 0
   until i >= times
   loop
     r.content.add(aLetter)
     i := i + 1
   end
 end
}
```

Figure 21: Implementation of the transformation in Kermeta.

4.4 Refactorings in Kermeta

This section shows how we can use well-known OO design techniques such as design patterns to develop model transformations. In contrast with more generative transformations (such as the class to RDBMS example) which map a whole model to another, model refactorings[19] are typically used as model edition primitives. A given refactoring can be used multiple times, in different contexts, and each application takes the development process a small step forward. The transformation tool should be as interactive as possible to ease this process by allowing the developer to experiment and progress through trial and error.
We don't put the emphasis on the refactoring presented here itself, which is really simple (moving a given method up to a superclass) but on the application of the *Command* design pattern to specify refactorings.

Refactorings as Commands

Firstly, a refactoring is a generic transformation which has to be parameterized; in our case, we have to specify which method we want to move, and to which superclass we want to move it. Then the tool needs to check the refactoring preconditions; if they are respected it will proceed to transform the model. The interface for refactorings thus defines three methods as follows:

```
abstract class RefactoringCommand
{
    operation check() : Boolean is abstract
    operation transform() : Void is abstract
    operation revert() : Void is abstract
}
```

This interface specifies three operations playing the Execute() role in the GoF pattern description:

- check() is called to evaluate preconditions on the model before transformation; when these preconditions are satisfied the transformation is guaranteed to be a refactoring so it can be applied safely.
- transform() applies the transformation.
- revert() should return the refactored model to it's previous state.

Concrete refactorings must subclass `RefactoringCommand` and provide methods for its three operations:

```
class MoveUpMethod inherits RefactoringCommand
{
   reference methodToMove : ClassHierarchyMM::Method
   reference destinationClass : ClassHierarchyMM::Class
   reference originClass : ClassHierarchyMM::Class

   method check() : Boolean is do

      // assert destinationClass is a superclass of methodToMove's owner
      if not destinationClass.isSuperclass
                                (methodToMove.owner.asClass) then
         raise Exception.new
      end

      // assert new methodToMove won't conflict
      if not conflicts(methodToMove, destinationClass).empty then
         raise Exception.new
      end
   end


    /** Apply the transformation : Move method to destination class */
    method transform() : Void boolean
       // could return a "successfully applied"
    is do
       // memorize current owner for revert()
       originClass := methodToMove.owner.asClass
       // move method to destination
       methodToMove.owner := destinationClass
    end


    /** undo the transformation : Move method back to original owner */
        method revert() : Void
    is do
        methodToMove.owner := originClass
    end


    /** list conflicts that would appear if the transformation is
    applied */
    operation conflicts(m : ClassHierarchyMM::Method,
                        someClass : ClassHierarchyMM::Class) :
                                  Collection<ClassHierarchyMM::Method>
    is do
        result := Set<ClassHierarchyMM::Method>.new
        result := someClass.subclasses.collect{ c |
            c.features }.select{ f | (f.name == m.name) and (f != m) }
    end
}
```

The metamodel

For this example we will use the small subset of the UML metamodel shown below, which defines classes, inheritance hierarchy and methods. This Kermeta metamodel also defines utility methods in metaclasses:

```
package ClassHierarchyMM;

require kermeta

using kermeta::standard
```

```
abstract class Classifier inherits GeneralizableElement
{
    attribute feature : seq Feature[0..*]#owner
    operation asClass() : Class is do
        result ?= self
    end
}

class Class inherits Classifier
{
    operation superclasses() : Collection<Class> is do
        result := generalization.collect{ g |
            var p : Class
            p ?= g.parent
        }
    end

    operation isSuperclass(child : Class) : Boolean is do
        result := child.superclasses().contains(self)
    end

    operation subclasses() : Collection<Class> is do
        result := specalization.collect{ g |
            var p : Class
            p ?= g.child
        }
    end

    operation isSubclass(child : Class) : Boolean is do
        result := child.subclasses().contains(self)
    end
}

class Feature inherits ModelElement
{
    reference owner : Classifier[1..1]#feature
    attribute visibility : String
}

abstract class GeneralizableElement inherits ModelElement
{
    reference specialization : Generalization[0..*]#parent
    reference generalization : Generalization[0..*]#child
}

class Generalization inherits ModelElement
{
    reference parent : GeneralizableElement[1..1]#specialization
    reference child : GeneralizableElement[1..1]#generalization
}

class Method inherits Feature
{
    attribute body : String
}

class Model inherits ModelElement
{
    attribute ownedElement : ModelElement[0..*]#namespace
}

abstract class ModelElement
{
    reference namespace : Model#ownedElement
    attribute name : String
}
```

Using the Transformation

```
@mainClass "Refactoring::Main"
@mainOperation "main"


package Refactoring;

require kermeta
require "../models/ClassHierarchyMM.kmt"

using kermeta::standard
using kermeta::utils
using kermeta::persistence
using kermeta::exceptions

// definition of RefactoringCommand

// definition of MoveUpMethod

class Main
{
   reference resource : Resource
   reference inputModel : ClassHierarchyMM::Model


   operation main() : Void
   is do
      loadResource("../models/SampleModel.xmi")
      inputModel ?= findElement(ClassHierarchyMM::Model, "root")

      var transfo : MoveUpMethod init MoveUpMethod.new
      transfo.methodToMove ?= findElement(ClassHierarchyMM::Method, "m")
      transfo.destinationClass  ?=  findElement(ClassHierarchyMM::Class,
"AncestorClass")

      if transfo.check() then
         transfo.transform()
         stdio.println("transformation applied")
      else
         stdio.println("precondition not satisfied")
      end

      resource.saveWithNewURI("../models/SampleModel-out.xmi")
    end


    operation loadResource(filename : String)
    is do
      var repository : EMFRepository init EMFRepository.new
      resource := repository.getResource(filename)
      resource.load()
    end
```

```
   operation findElement(metaclass : kermeta::reflection::Class,
                         name : String) :
                               ClassHierarchyMM::ModelElement
   is do
      var found : Boolean init false
      from var it : Iterator<Object> init resource.instances.iterator
      until found or it.isOff
      loop
         var next : Object init it.next
         if (metaclass.isInstance(next)) then
            var n : ClassHierarchyMM::ModelElement
            n ?= next
            if n.name == name then
               result ?= next
               found := true
            end
         end
      end
   end
}
```

4.5 Determinization and minimization of automata in Kermeta

These transformations show the power of OCL-like constructs to manipulate collections.

Formal definition of non-determinist finite automaton metamodel

Formally, a non-determinist finite automaton is a A = (∑, Q, T, q0, F), where :

- ∑ is an alphabet
- Q is a finite set of states
- T is a set of transitions rules, such as *si* X *a* --> *sj* where *si*, *sj* Є Q^2 and *a* Є ∑È{e}
- q0 is the initial state
- F is the set of final states

Automaton metamodel

We considered as simple case of finite automaton: an automaton that as an initial state, (*initialState*), a set of available states (*stateSet*), a set of transitions (*transitionSet*), an alphabet, an a set of final states (*finalStateSet*). We chose that our automatons are all e-free (no e-transitions).

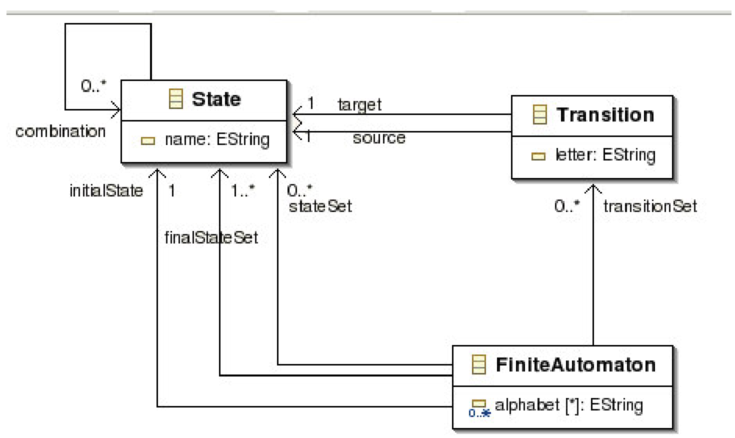

Figure 22: Automaton metamodel.

combination is an attribute of State that is used for two purposes :

- For the determinisation implementation : it represents each new group of state that is created for the determinist automaton
- For the minimisation implementation : it is used as a marker for states of the input automaton that have already been added in the equivalence classes of the output-minimal automaton. This is a light optimisation of the computation of the equivalence classes that constitute the new states of the minimalized automaton.

alphabet is, in Kermeta implementation, a derived property that is computed from the automaton instance (we get all the letters defined on the transitions)

Kermeta representation

The following figure shows the Kermeta textual representation of the automaton metamodel.

```
class FiniteAutomaton
{
   reference stateSet : set State[0..*]//Set<State> //#owningFA
   reference initialState : State
   reference finalStateSet : set State[0..*]
   reference transitionSet : set Transition[0..*]
   property readonly alphabet : Set<String>
   getter is do
      result := self.seqToSet(self.transitionSet.collect{e|e.letter})
   end

   /** Initialize a new automaton from an existing one */
   operation initialize(initState : State) is do
      stateSet.add(initState)
      initialState := initState
      initialState.combination := Set<State>.new
   end
}

class State
{
   reference combination : Set<State>
   reference name : String
}

class Transition
{
   reference source : State
   reference target : State
   reference letter : String
}
```

Formal definitions of the determinization algorithm

Determinist automaton

A finite automaton is determinist if and only if the relation t is a transition function such as:

t : Q X ∑ --> Q (e-transition no more allowed, but in our case we don't work with it)

from a state, there is at most one possible transition with the same letter

Algorithm

The algorithm for determinizing a non-determinist finite automaton is a classical problem for the automata such as defined in our metamodel. An introduction of it can be found in[20]. Here it is:

Initialisation of A' = (∑, Q', T', q0', F'), determinist version of an automaton A :

- T' initialized to Æ
- q0' initialized to { q0 }
- Q' initialized to { q0' }

q' is a "new" state that is a part of Q (q' Î P(Q))

```
for each state q' of Q' not considered yet do
  for each letter a of ∑ do
    q'' ¬ { y /  x Î q' and y Î Q / (x, a, y) Î T }
    T'  ¬  T  È { (q', a, q'') }
           Q' ¬ Q' È {q''}
  rof
rof
```

F' ¬ { q'Î Q' / q'Ç F ¹ Æ}

Implementation

```
class Determinization
{
   reference processed_states : Set<State>

   operation main() : Void is do

      // Input automaton (non-determinist)
      var input : FiniteAutomaton init Sampler.new.createSample1()
      var output : FiniteAutomaton init FiniteAutomaton.new
      // Control variables
      processed_states := Set<State>.new
      // Initialize output automaton with input.initialState
      output.initialize(input.initialState)
      // Apply the determinisation
      determinize(input, output, output.initialState)

      // Define the final states : q' intersection initial
      // Final states is void
      output.finalStateSet.addAll
      (
        output.seqToSet( output.stateSet.select {
          e | e.combination.detect{ a |
            input.finalStateSet.contains(a) } != void  } )
      )
   end


// THE DETERMINISATION ALGORITHM
   //input : initial automaton
   // output : final determinized automaton
   // output_state : the next state to consider
   operation determinize( input : FiniteAutomaton,
                          output : FiniteAutomaton,
                          output_state : State)
   is do
     // for each state not considered yet
     if not processed_states.contains(output_state) then
        processed_states.add(output_state)
        var newq : State init State.new
        // For each letter of the alphabet
        from var lit : Iterator<String> init input.alphabet.iterator
        until lit.isOff
        loop
          // There exists a state x of q'
          //(where q' is a P(Q)) and a state y from Q
          // such that: x --l--> y belongs to input.transitionSet
          var nextl : String init lit.next
          newq.combination := Set<State>.new
          newq.combination.addAll(
            input.seqToSet(
               input.transitionSet.
                 select { e | e.letter.equals(nextl) }.
                   select { a | output_state.equals(a.source)
                     or output_state.combination.contains(a.source)}.
                       collect { b | b.target })
            )
          newq.name := join(newq.combination.collect{ a | a.name })
          // Add the state to the output automaton if we found one
          if (newq.combination.size > 0) then
             // Add the new transition
             var newt : Transition init Transition.new
             newt.initialize(output_state, newq, nextl)
```

```
            output.transitionSet.add(newt)
            // Add the new state if it is not already added
            if (output.stateSet.
                        detect { e | newq.name == e.name } == void)
            then
                  output.stateSet.add(newq)
            end
              self.determinize(input, output, newq)
           end
        end // End of loop
      end // End of processing of not considered state

   end

   // Create a special name for new states with their combination :
 // the state -> { q0, q1 } is name "q0q1"
   operation join( str_seq : Collection<String>) : String is do
        result := ""
        from var it : Iterator<String> init str_seq.iterator
        until it.isOff
        loop
            result.append(it.next)
        end
   end

}
```

Formal definition of the minimization algorithm

A minimal automaton is an optimized (pre-determinized) automaton that has the minimum number of states that performs the same function (i.e. produces the same language in a language automaton) of its equivalent automaton. The reader can find a formal definition of minimal automaton in[21].

We chose to implement a simple algorithm provided by[21]. It is called a **layerwise** computation of the equivalence relations. A better implementation should be provided, but would need a few optimizations for the list handling in Kermeta. However, the language was quite ergonomic, thanks to the implementation of OCL constraints, and made the implementation easier to write.

The algorithm

The algorithm finds, incrementally, the pair of states (p, q) such as p and q ra

```
AFD M = (Q, A, q0, F, d)
H : ( Q -F )² U Q²
Hold : Q*Q
Begin
  do
    Hold := H
    for each (p, q) in Q*Q do
      for each letter a in A do
         s = d(p,a)
         t = d(q,a)
         if (s,t) is not in H then remove (p,q) from H
  until Hold == H
end
```

Implementation

The implementation of the minimization was a bit more complicated, since we had to construct also, from the pairs of states given by the algorithm execution, the new states (which are the equivalence classes of the pairs of equivalent states (*p,q* in the algorithm), and the new transitions. We show here only the relevant parts of the implementation of minimization algorithm in Kermeta.

```
class Minimization
{
  reference equivalent_pairs : set Pair[0..*]
  reference all_input_pairs : Set<Pair>
  reference helper : AutomatonHelper

  operation main() : Void is do

       helper := AutomatonHelper.new
       // Input automaton (non-determinist)
       var input : FiniteAutomaton init Sampler.new.createSampleM1()
       var output : FiniteAutomaton init FiniteAutomaton.new
       all_input_pairs := Set<Pair>.new

       // Initialize the complete set of
       // possible pairs: all_input_pairs = Q x Q (Q is the stateSet)
       // Initialize Eo : equivalent_pairs
       // = { F \ Q }Â² ^ FÂ²  (states that accept
       // the {e} transition or empty word
       input.stateSet.each { p | input.stateSet.each { b |
       // Check : (p,q) is in Eo, i.e either both are final
       // states or both are NOT final states
       var isFinalLeft : Boolean init input.finalStateSet.
                                 detect { e | p.name == e.name }!=void
       var isFinalB    : Boolean init input.finalStateSet.
                                 detect { e | b.name == e.name }!=void
       // Also fill the all input pairs
       if find_one(all_input_pairs, p, b) == void
          then all_input_pairs.add(createPair(p, b)) end
       if  ((isFinalLeft and isFinalB)
                or (not isFinalLeft and not isFinalB))
           and
             find_one(equivalent_pairs, p, b) == void
       then
             equivalent_pairs.add(createPair(p, b))
       end    } }

       // Minimalize
       //var output_pairs : Set<Pair> init
       minimalize(input, output)
       output.prettyprint()
  end

  operation minimalize(input : FiniteAutomaton,
                       output : FiniteAutomaton) :   Set<Pair> is do
    result := equivalent_pairs
    var old_equivalent_pairs : Set<Pair> init all_input_pairs
    from var it : Iterator<Pair> init old_equivalent_pairs.iterator
    until old_equivalent_pairs == result
    loop
        old_equivalent_pairs := result
        // For each pair
```

```
      old_equivalent_pairs.each { eqPair |
      // For each letter of
      if (isNotOwnedTransition(input,
                               eqPair,
                               old_equivalent_pairs) == true)
      then
         // remove this pair from eq. pairs (H)
         result := old_equivalent_pairs
         var fp : Pair
                   init find_one(result, eqPair.left, eqPair.right)
         if (fp!=void) then
             result.remove(fp)
         end
      end
   }

  end
  // Set the result
  result := Set<Pair>.new
  result.addAll(old_equivalent_pairs)
  // Create the equivalent classes, which become the new states
  var classSet : Set<Set<State>> init Set<Set<State>>.new
  createEquivalenceClasses(output,
                           input.stateSet,
                           old_equivalent_pairs)

  output.stateSet.each
 { s | s.name := helper.join(s.combination.collect{ a | a.name }) }

  // Create the transition between the new states
  // inputStates contains the links to their eq.class
  createEquivalentTransitions(output.stateSet,
                              input.stateSet,
                              input.transitionSet)
end

  // Equivalence relation  xRy == yRx :
  operation find_one(pairSet : Set<Pair>,
                     left : State,
                     right : State) : Pair is do
    result := pairSet.detect { p |
    (p.left.name == left.name and p.right.name == right.name) or
    (p.right.name == left.name and p.left.name == right.name) }
  end


// Returns true if for each letter of the input automaton,
// a pair (p,q) does not satisfy the "T(p, a), T(q, a)
// belongs_to the equivalent_pairs" condition
operation isNotOwnedTransition(automaton : FiniteAutomaton,
                               pair : Pair,
                               equivalent_pairs : Set<Pair>) :
                                            Boolean
is do
  // if there exists a letter a in the automaton such as
  // T(pair.left, a), T(pair.right, a) belongs to distinct_pairs
  // "void" pair is allowed!
  result := false
  from var it : Iterator<String> init automaton.alphabet.iterator
  until it.isOff or result == true
  loop
     var letter : String init it.next
     var tleft : Transition init automaton.
       transitionSet.detect { t |  t.source.name==pair.left.name
           and t.letter == letter }
     var tright : Transition init automaton.
       transitionSet.detect { t | t.source.name==pair.right.name
           and t.letter == letter }
```

```
        if (tleft!=void and tright!=void) then
           // empty word belongs to accepted words
           if find_one(equivalent_pairs,
                       tleft.target,
                       tright.target) == void then
               result := true
           end
        end
     end
   end
```

```
// Create the equivalenceClasses that will constitutes
// the states of the minimal output automaton.
operation createEquivalenceClasses(output : FiniteAutomaton,
                                   stateSet : Set<State>,
                                   equivalent_pairs : Set<Pair>) :
                                             Set<Set<State>> is do
  var eqClass : Set<State> init Set<State>.new
  result := Set<Set<State>>.new
  from var it : Iterator<State> init stateSet.iterator
  until it.isOff
  loop
     var state : State init it.next
     var news : State
     equivalent_pairs.select
       { pair | pair.left == state }.each
     { pair |
     // combination becomes a "marker" for classed states
     // if it is void, it means that it does not
     // belong to a eqclass yet
     if (state.combination == void) then
          // create the eq. class and the state
          eqClass := Set<State>.new
          eqClass.add(pair.left)
          news := helper.createState(state.name)
          news.combination.add(eqClass.one)
          helper.join(eqClass.collect{ a | a.name })
          output.stateSet.add(news)
          result.add(eqClass)
          // Mark state that is already added
          // we use combination to ease the transition computation
          state.combination := Set<State>.new
          state.combination.add(news)
      end
      // Process the right element of the pair :
      // add it to the eq.class of the left element!
      var sright : State init stateSet.
           detect { s | pair.right == s and s.combination == void }
      if (sright != void) then
          sright.combination := Set<State>.new
          result.detect{ c | c.contains(state)}.
             add(State.clone(pair.right))
          output.stateSet.
             collect { s | s.combination}.
                detect{ c | c.contains(state) }.
                   add(State.clone(pair.right))
      end
}
      // Set the eq-class of current state in its "combination"
      // attribute if it was skipped because already processed
      // through the left element selection
      if (state.combination.size ==0) then
          state.combination.add(output.stateSet.
             detect{ s | s.combination.contains(state)})
      end
  end
end
```

```
  // eqClassStateSet : the minimal automaton set of states
  // stateSet : the input automaton set of states
  // transitionSet : the intput automaton set of transitions
  operation createEquivalentTransitions
                  ( eqClassStateSet : Set<State>,
                    stateSet : Set<State>,
                    transitionSet : Set<Transition>) :
                                     Set<Transition> is do
    result := Set<Transition>.new
    // for each eq-class
    from var it : Iterator<State> init stateSet.iterator
    until it.isOff
    loop
        var nextInputState : State init it.next
        // Get the eq.class to which the current state belongs
        var nextEqClassState : State init
                                 nextInputState.combination.one
        // For each letter, Get the transition for which
        // the current state is a source
        var nextTransitionSet : Sequence<Transition> init
             transitionSet.select { t | t.source == nextInputState }
        // The target combination is the eq. class target of
        // the new transition!
        nextTransitionSet.each { t |
        // Add this transition
        if result.
           detect {  rt | rt.source == nextEqClassState and
                          rt.letter == t.letter } == void
        then
            var newt : Transition init Transition.new
            var nextEqClassStateTarget : State init
                eqClassStateSet.
                    detect { s | s.combination.contains(t.target) }
            newt.initialize(nextEqClassState,
                            nextEqClassStateTarget,
                            t.letter)
            result.add(newt)
        end
        }
    end
    // Print the transition
    stdio.writeln("transitions : " + result.size.toString)
    result.each { t | stdio.writeln(t.toString) }
  end
}
```

5 Conclusion

In this paper we have shown how an executable meta-language could be used to express model transformations.

We have explained the rationales for building object-oriented executable meta-languages, and then discussed the perceived benefits of these languages applied to the model transformation field.

We are currently in favor of a level of language support for model transformation which is between totally general purpose languages (such as Java) and model transformation domain specific languages such as specified by QVT. Our approach could be described as model domain specific languages.

Nevertheless, Kermeta is first and foremost an executable meta-language, which can be used to for a wide range of purposes, including model transformation but also to specify the abstract syntax of languages under the shape of metamodels. As Kermeta is executable, the operational semantic of these languages can then be further specified, and even implemented by reusing domain specific libraries. Hence, Kermeta is a language development environment, where domain specific experimentations can be conducted via libraries, and then injected into Kermeta metamodels, which in turn model domain specific languages (for instance for model transformation).

Interestingly, we have found that it was more difficult to understand the description of the required transformations than to write the transformations. This leads us to believe that it would be useful to find more precise ways to specify the transformations.

The work presented in this paper may be viewed as an experimentation in applying executable meta-languages to model transformations. Our work is obviously far from bringing definitive answers to the complex problems addressed by the MTIP workshop. However the presented material may contribute, with many other ongoing research works on similar topics, to a better understanding of language requirements with respect to model transformations and software engineering.

References

[1] OMG. MOF 2.0 Core Final Adopted Specification, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2004.
[2] Netbeans MDR, web site http://mdr.netbeans.org/
[3] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R. and Grose, T. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.
[4] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, "*The Generic Modeling Environment*", Proceedings of the IEEE International Workshop on Intelligent Signal Processing, WISP'2001, Budapest, Hungary, may 24-25, 2001
[5] Vojtisek, D. and Jézéquel, J.-M. MTL and Umlaut NG: Engine and Framework for Model Transformation. *ERCIM News*, *58*.
[6] Bézivin, J, Dupé, G, Jouault , F, Pitette, G, and Rougui, EJ : First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: OOPSLA 2003 Workshop, Anaheim, California.
[7] OMG. Revised submission for MOF 2.0 Query/View/Transformation, Object Management Group (QVT-Merge Group), http://www.omg.org/cgi-bin/apps/doc?ad/2005-03-02, 2005.
[8] OMG. UML 2.0 Object Constraint Language (OCL) Final Adopted specification, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, 2003.
[9] Mellor, S., Tockey, S., Arthaud, R. and Leblanc, P. Action Language for UML: Proposal for a Precise Execution Semantics. *Proceedings of UML 98 (LNCS1618)*. 307-318.
[10] Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V. Centaur: the system. *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments*, *13* (5). 14 - 24.
[11] Clark, T., Evans, A., Sammut, P. and Willans, J. Applied Metamodelling: A Foundation for Language Driven Development, http://albini.xactium.com, 2004.
[12] Greenfield, J., Short, K., Cook, S., Kent, S. and Crupi, J. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
[13] Muller, P.-A., Studer, P., Fondement, F. and Bezivin, J. Platform independent Web Application Modeling and Development with Netsilon. *Accepted for publication in Journal on Software and Systems Modelling (SoSym)*. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/netsilon_sosym.pdf.
[14] X. Blanc, S. Bouzitouna and M.P. Gervais, A Critical Analysis of MDA Standards through an Implementation: the ModFact Tool, First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, March 17-18, 2004, Enshede, The Netherlands.
[15] Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In L. Briand, S.K., ed.: Proceedings of MODELS/UML'2005. Volume to be published of LNCS., Montego Bay, Jamaica, Springer (2005)
[16] http://www.kermeta.org
[17] D. Roberts and R. Johnson. Evolve frameworks into domain-specific languages. In *3rd International Conference on*
Pattern Languages, Allerton Park, Ill., September 1996.
[18] http://www.omondo.com/
[19] Gerson Sunye, Damien Pollet, Ives Le Traon, and Jean-Marc Jezequel. Refactoring UML models. In Proceedings of UML 2001, pp. 134-148
[20] Julia, Automates finis, University Course, Université de Nice Sofia-Antipolis, France
[21] Watson, B.W, A Taxonomy of finite automata minimization algorithms, Technical Report, Faculty of Mathematic and Computing Science, Eindhoven University of Technology, the Netherlands (Sep. 1994)