D3.2-Megamodel for Transformations Architecture

MoScript – Models Scripting Language

	NAME	PARTNER	DATE
WRITTEN BY	KLING W.	ATLANMOD	04/02/2011
REVIEWED BY			

RECORD OF REVISIONS

ISSUE	DATE	EFFECT ON		REASONS FOR REVISION
		PAGE	PARA	
1.0	02/11/2010			Document creation

TABLE OF CONTENTS

TABLE OF APPLICABLE DOCUMENTS

N°	TITLE	REFERENCE	ISSUE	DATE	SOURCE	
					SIGLUM	NAME
A1						
A2						
A3						
A4						

TABLE OF REFERENCED DOCUMENTS

N°	TITLE	REFERENCE	ISSUE
R1	Galaxy glossary		
R2			
R3			
R4			

ACRONYMS AND DEFINITIONS

Except if explicitly stated otherwise the definition of all terms and acronyms provided in [R1] is applicable in this document. If any, additional and/or specific definitions applicable only in this document are listed in the two tables below.

Acronymes

ACRONYM	DESCRIPTION

Definitions

TERMS	DESCRIPTION

1. INTRODUCTION

As the Model-Driven Engineering (MDE) paradigm and tools are maturing, the number of modelling artefacts consumed and produced by software or engineering processes (e.g., models, metamodels, transformations, etc) has increased considerably.

MDE development processes for complex systems [MDMCS] are typical examples of this situation. In these systems, every artefact (e.g. requirement specifications, analysis and design documents, implementation artefacts, etc.,) is considered as a model. Apart from being numerous, these artefacts are often large, heterogeneous, interrelated, with a complex internal structure, and possibly distributed. Furthermore, unlike in traditional development processes, in MDE processes systems are built by transforming models (from higher to lower abstraction levels). Then, a change in the upper-level models has a considerable impact in all models derived from it, and requires the re-execution of one or more transformations to propagate changes until the end of the transformation chains [Pragm].

Actually, most of the mentioned characteristics are not inherent to MDE itself but rather to the essential complexity of the domain to be modelled [SilvBul]. However, as any software engineering approach, MDE inevitably introduces additional complexity, which can be referred to as accidental complexity. As a consequence, for MDE to scale up efficiently, both the essential and accidental complexities need to be harnessed.

1.1 GOAL OF THIS DOCUMENT

This document proposes a new strategy based on the Megamodel [GMM] concept, for dealing with large amounts of models. This strategy facilitates the manipulation of Megamodels and the model repositories they represent, to perform common modelling tasks. It allows retrieving information from Megamodels, inspecting models and invoking operations on them, combining models to produce new ones, and automatically evolve Megamodels by registering the newly produced models in the repository. The strategy is presented in terms of a Domain Specific Language (DSL) called MoScript and a supporting metadata platform. Although the supporting metadata platform is not the focus of this work its desired characteristics will be also described in this document to provide the global picture of the solution.

2. PROBLEM STATEMENT

Model repositories may contain hundreds or even millions of models. For instance, after a model driven reverse engineering [MDRE] of a java system; each Java class may have a corresponding XMI model. So, if the system has several hundreds or millions of classes there will be the same amount of models. Furthermore, if we consider other kinds of models such as requirements, analysis, design, architecture models etc., the total amount of models is even higher.

Apart from being numerous, models are usually strongly interrelated. There exist direct interrelations such as the one between a model and its metamodel or as the one between a transformation and the metamodels of the models it transforms etc. There are also indirect relations such as models that weave two or more models, or the transformation that produce target models from input models.

Models are also usually heterogeneous; they can be stored and handled in different formats and by different modelling frameworks. They also may model different aspects of a system or different levels of abstraction. Moreover models can be distributed i.e. they may not reside locally with respect to the tools that use them.

Because of the mentioned characteristics, organizations that deal with complex systems are facing today scalability problems while building such systems or using models following the MDE approach. This scalability problem leads them to build their own “ad-hoc” model management tools or frameworks while they should be spending their resources designing their systems.

The automation of modelling tasks such as the execution of large amounts of transformations and the orchestration of their execution have been done until now mostly with scripting or workflow techniques brought from no MDE approaches, such as Ant[1] Scripts, openArchitectureWare[2] Workflow, etc. The problem of those techniques is that they do not scale automatically with the model repositories, i.e. as soon as a new model enters or goes out from the repository, or change its location scripts and workflows may became outdated. This is because scripts usually have the models hardcoded in the scripts; in the best case they query the storage system to get the list of existing models to do their work.

The mentioned problems are being somehow alleviated with the automatic generation of scripts. However this script generation still requires a certain amount of repetitive work every time a new script must be generated. This approach is also not very effective in collaborative environments where several users create, delete or update models frequently. The execution of those scripts is unsafe and may lead to many inconsistencies without additional verification mechanisms.

Another disadvantage of those approaches is that they do neither hide to the end user, several unnecessary details inherent to the model driven approach and due to the fact of working with several different technologies, nor hide the complexity due to that models may be distributed. Each tool provider has its owns tools or script extensions to process the models and its users must be completely aware of the location of models and technical details of each tool.

In any case those techniques are neither efficient to navigate model repositories because there is missing a global view of the model repository, which describes with accuracy how the models are related and on top of which the scripts or workflows could work.

[1] The apache ant project. //ant.apache.org
[2] openArchitectureWare. http://www.openarchitectureware.org/

To better illustrate the parts of the problem, we describe next some examples of model manipulations over large and heterogeneous model repositories, which are difficult and time consuming to accomplish with current approaches:

- Find a specific kind of model, e.g. models which are not metamodels, metamodels, transformations, requirement specifications, architecture definitions etc.
- Find the models according to how they are related to each other, e.g., models which conform to a specific metamodel, transformations that may be applied to a specific model, etc, models that participate in a transformation chain etc.
- Search for specific models and be able to perform operations with the result of the search, e.g.:
  - Collect metrics from models content, e.g. number of class elements of all the metamodels, models that have element instances of a specific metamodel element, number of new elements produced by a transformation etc.
  - Execute transformations after finding them, e.g. re-execute all the transformations of a repository, find the transformation that produce a given set of models and re-execute them.
  - Run verifications to a given set of models.
  - Perform comparisons between models
  - Match models
- Perform test operations, e.g. select a set of transformations, execute them, and run verifications to the resulting models or query them to check the obtained results

3. PROPOSED STRATEGY

Locate and work with specific models in a small model repository is a simple task. However in large and complex systems, it is difficult to find the rights models to work with, to understand how are they related to other models and therefore, how to correctly apply operations to them, such as transform, query, match etc.

To facilitate the work with large amounts of models, we need ways to apply operations to large sets of models without human intervention. **This is possible if we manage to know which operations may be applied to which models and if we are able to automate the application of these operations without requiring constant human intervention.**

3.1 MEGAMODEL

In order to know which operations may be applied to which models, we propose a metadata layer. It is actually a model that stores the information of each model in the repository, how they are related to other models, as well as the transformations and other operations that may be applied on them, and the tools with which it can be carried out.

Such a model is called a Megamodel. The Megamodel is defined as a model, whose elements represent models and the relations between them, within the scope of a repository or a system. Since we consider that everything is a model, when we talk about models we are not exclusively considering XMI[3] files. Other kinds of artefacts such as source code, binary files or documentation, which may be produced from XMI models or vice versa, are considered as well.

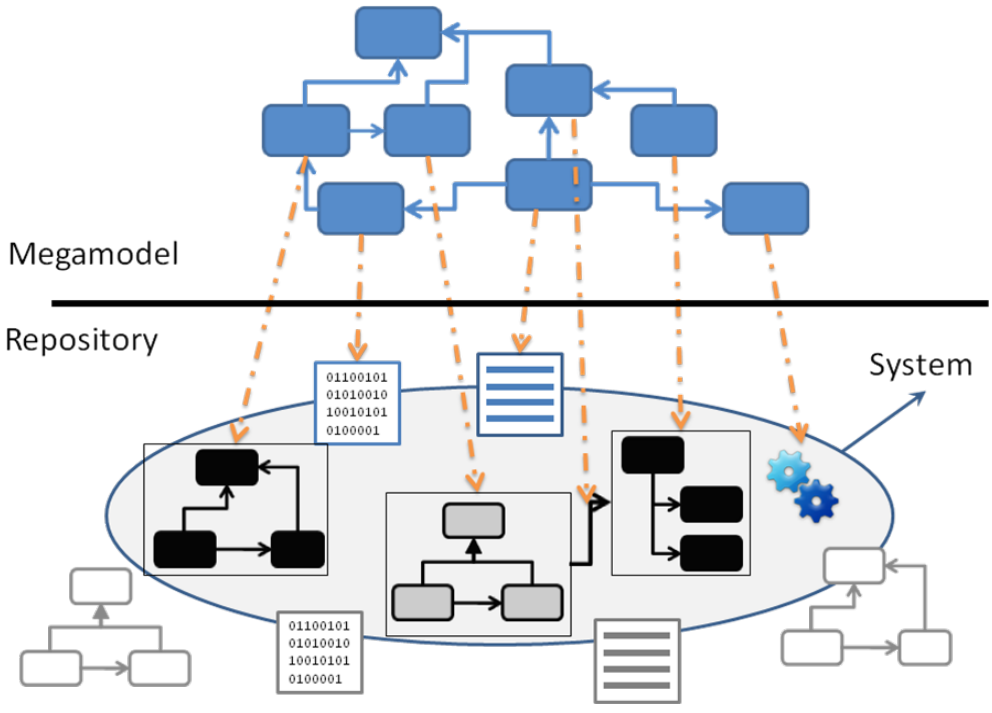


Figure 1: The Megamodel and the system it represents

[3] XML Metadata Interchange (XMI®) - OMG Formally Released Versions Of XMI - http://www.omg.org/spec/XMI/index.htm

Having all the information on models stored in the Megamodel, we are able to query it with query languages such as OCL[4], to find e.g. which models conform to a given metamodel, which models a given transformation can be applied to and so on. Querying the Megamodel is also possible to follow long chains of transformations and discover which models have been derived from a given set of models.

Although the information about the models (that may be obtained from a Megamodel) is very useful for understanding the models repository, and also as a support for taking decisions about the further works with the models, we require additional mechanisms to benefit from this information in order to be able to apply operations (e.g. transformations, matching, querying) to the models obtained as result of a query to the Megamodel.

3.2 MODEL DEREFERENCING

In the field of RDBMS, some database engines have tables that describe what database objects (e.g. tables, procedures, triggers etc) exist within the database. This is known as the database catalogue and may contain information such as names, sizes and number of rows in each table, etc. This information can be used e.g. to find all tables accessible by a user, get a list of stored procedures, and get information about many other types of objects in a database.

Similarly, as explained before, a Megamodel describes models and can be used not only to find models by their name, but also by its kind (e.g. metamodel, metametamodel, transformation etc), by how they are related to others, or any another property provided by the metadata.

RDBMS also enable the users to access the content of tables via SQL queries, for finding tables with specific inner values, or for the application of generic operations to tables e.g. the different kinds of *joins*. This results in powerful data manipulations. These operations can be scripted because they can be applied generically to all the tables.

Considering a Megamodel, a similar mechanism is not directly provided. We cannot, by querying the Megamodel with standard query languages like OCL, e.g. find models by iinner characteristics. We are also not able, as a result of a query to the Megamodel, to apply operations such as transformatons, matchings, comparisons, extractions etc., to a set of models, even if they conform to a same metamodel.

This limitation exists due to the inability of standard query languages to access the physical models described from the metadata stored in a Megamodel. **We need mechanisms to dereference the models pointed at and described by the metadata,** and thus be able to apply operations to them such as query, transform, match etc.

Figure 2 shows the two kinds of relations we find in a Megamodel. On the one hand, we have Element Reference relations, which in the case of the Megamodel, describe how the models are related between them. These are references that can be navigated with a query language such as OCL. On the other hand we have the Model Reference relations, which act as symbolic links to the models but cannot be navigated using usual query languages. Because of that, is not possible for instance to select a model from the Megamodel and then query directly inside of it.

[4] Object Constraint Language (OCL) - OMG Formally Released Versions Of OCL - http://www.omg.org/spec/OCL/

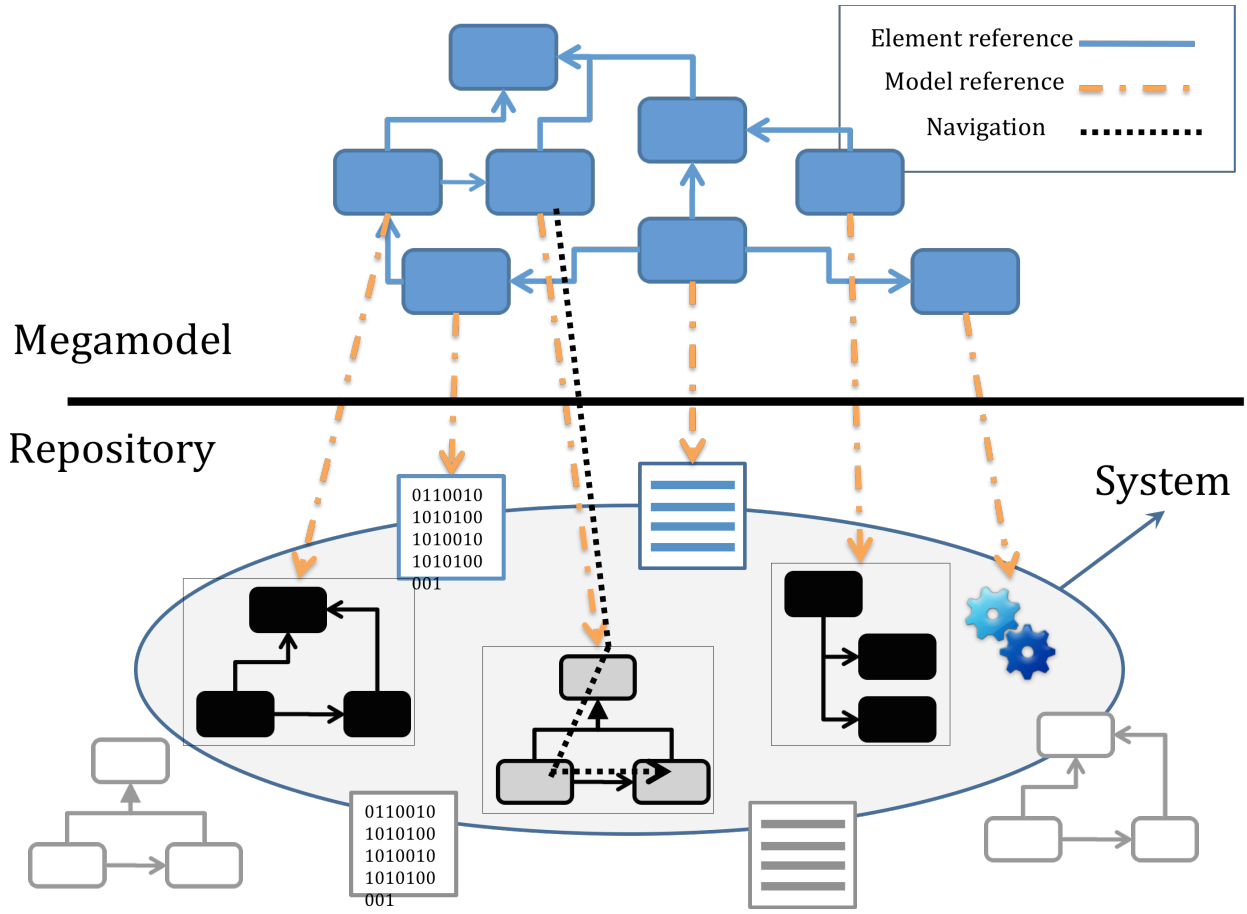


Figure 2: Model dereferencing

Thus, as another part of the strategy, we propose the introduction of a **model dereferencing mechanism**. If the query language is enabled with this mechanism, it means that after finding an element, which represents a model reference, the physical model it points to, may be obtained and operations may be applied to it. Figure 2 shows the kind of navigation it would be possible to do with a query language having a model dereferencing mechanism.

3.3 TRANSFORMATIONS AS MODELS AND OPERATIONS

In the context of MDE, we consider that all the operations that may be applied on models to produce other models or artefacts are transformations. For instance the conventional java file compilation may be seen as a transformation of a java textual syntax model into a java bytecode model. A comparison between two models may be seen as a transformation of two input models into a model of differences, etc.

We propose as another part of the strategy, to represent transformations as transformation models in the Megamodel and to be able to apply them on other models stored in the Megamodel, after retrieving them. To be able to run transformations, we also propose the association of callable operations to model elements. In this case the element that represent a transformation will have associated an operation that may be combined with models to apply transformations on them. As specific tools must execute transformation definitions, we need to have an association between the transformation models and the tools as well.

These strategies will greatly facilitate the management of huge repositories of models and transformations by establishing a homogeneous way to retrieve them and to apply transformations to models and by hiding de details.

We propose two kinds of operations; operations to transform models, operations to query inside models and operations to query the properties of the physical models (e.g. their availability, size, if they have been modified externally or not etc.).

3.3.1 Operations

The operations we propose in this section are side effects free. This means they do not modify the repository. The manipulation of the artefacts performed with these operations is made “in memory”.

The set of operations we propose correspond to the following structure:

```
Model_Type :: operation(arguments) [: Return_Type]
```

`Model_type` corresponds a Megamodel element type, which represents the model e.g. Transformation, Metamodel, Model, etc. The *operation* corresponds to the operation that may be invoked in the context of the model element after its retrieval. The operation may receive arguments, which may also be model elements, and as result return new models or the content of them.

The following are the set of operations we proposed in order to handle large amounts of models in a batch-like way.

```
Model :: allContents() : Collection(OclAny)
```

The invocation of the `allContents` operation on a Model element dereferences the physical model it points to and obtains all its elements as a Collection. The model elements obtained in the collection may be used to reach other model elements. Because the `allContents` operation may be expensive, we propose other operations that enable us to filter the elements to be retrieved from the model.

```
Model :: allContentRoots() : Collection(OclAny)
```

The operation `allContentRoots` permits to retrieve only the elements, which are top containers of other model elements and elements that are not contained by others. From theses elements we gain access to the rest of elements of the model.

```
Model :: allContentInstancesOf(typeName: String) : Collection(OclAny)
```

The operation `allContentInstancesOf` operation retrieves all the contents of the model, which are instances of the type `type_name`. The `type_name` must correspond to a name of a type of the metamodel the model conforms to.

```
Model :: allContentInstancesOf(type: OclAny) : Collection(OclAny)
```

Instead of using the name of the type we may use the instance of the type as well, which should be selected from the metamodel.

The former operations are useful for querying the model contents. For applying transformations to models, we propose the following operations.

```
Transformation :: applyTo(models : Sequence(Model)) :  Sequence(Model)
```

`The Transformation` element represents any kind of transformation, such as model-to-model transformations (m2m), model-to-text transformations (m2t) and text-to-model transformations (t2m). For m2m transformations we propose the `applyTo` operation.

The `applyTo` operation is an operation that may be invoked on a Transformation model element. The `applyTo` operation receives as argument a sequence (ordered collection) of models to which the transformation should be applied on. The order of the models in the sequence does not have any importance unless the transformation receives several models which conform to a same metamodel. In that case the order of the models in the sequence must match the order and type (metamodel) of the transformation.

```
Transformation :: applyTo(models : Map(String, Model)) : Map(String,  Model)
```

The `applyTo` operation with a Map as argument applies also a transformation to one or more models. In this case, its key distinguishes the models one from the other. The models must of course match with the metamodel type specified by the transformation in the Megamodel.

If the transformation execution fails, a model with the problems encountered during execution is returned anyway.

The `applyTo` operation is especially useful when the transformation is in some sense generic. This means that the transformation may be applied to a broad set of different models, e.g. a transformation that transforms UML sequence diagrams into UML class diagrams, or class diagrams into entity relationships diagrams. For these kinds of transformations it is necessary to be able to easily variate their input models.

There are other transformations, which are very specific. This means they take always the same models (e.g. same file names) as input and produce the same models (same file names). For this kind of transformations is not necessary to specify each time which are the models we want the transformation to transform. Instead of that, we store this configuration in an element called `TransformationRecord`. A `TransformationRecord` knows the transformation and the models, which it should be applied on. Of course, if we want to change the model we could use the `applyTo` operation to do so.

```
TransformationRecord :: run()
```

For these kinds of transformation we propose the `run` operation. This operation is associated to the `TransformationRecord` element. This operation runs the transformation with a set of predefined models. We can then, for any transformation, retrieve its `TransformationRecord` (if it exists) and invoke the run method to rerun it. In fact, a `TransformationRecord` record is created each time a transformation is executed storing the configuration of the transformation execution and the status of the result.

For T2M transformations, we propose the `inject` operation. This operation enables us to convert files using concrete textual syntaxes to XMI models. The operation is useful to be then able to apply m2m transformations to the result of the injection (i.e. a model).

```
Transformation :: inject(textual_syntax : Entity) : Model
```

The operation `inject` receives as parameter an element that represents a file using a concrete textual syntax and return its corresponding XMI. Of course, the metamodel, which the textual syntax corresponds to, must exist and must be related to it in the Megamodel. We gave the name `Entity` to the elements in the repository that are not models in XMI format.

Finally, we propose the following operation to check if the physical model exists or is available. Because models may be stored in a file system, a registry or somewhere in the web, and may also be removed or modified externally, we need a way to check their availability before actually manipulating them.

```
Entity :: available(artefact : Entity) : Boolean
```

The `available` operation may be invoked on any Entity, which represents an artefact of the repository. This operation returns “true” if the physical artefact exists or is available. This is one of the many operations we can define to check the state in the artefacts of the repository.

3.4 REPOSITORY EVOLUTION

Systems change with time, so we need ways to evolve the repository, which contains all their artefacts, i.e. ways to register, modify and remove artefacts from it. The operations we presented so far enable us to manipulate existing artefacts and produce new ones, but they do not persist the changes to the repository. These operations are useful when the manipulation of artefacts is made for testing purposes, however when we are sure the manipulations of the artefacts will generate the expected results, we need to persist these changes to evolve the repository.

To this intent, we propose a set of Statements for evolving the repository.

3.4.1 Statements

The first statement is the `save` instruction statement. After running a transformation or injecting a file using textual syntax, we obtain a new model. So if we want it to be part of the Megamodel permanently, we need to serialize and register it in the Megamodel. This is what the save statement does.

```
save <Model> to <locator> as <identifier> in <Megamodel>
```

The `save` statement takes a Model element as argument, which represents the model we want to save. The `locator` is the model URI where the model should be stored. The `identifier` is a name which the Model will be unequivocally identified with, within the Megamodel. The `Megamodel` argument is a model element that represents the Megamodel, which the model should be registered in. Note that a Megamodel is a model, so a Megamodel can contain several megamodels as well as other models.

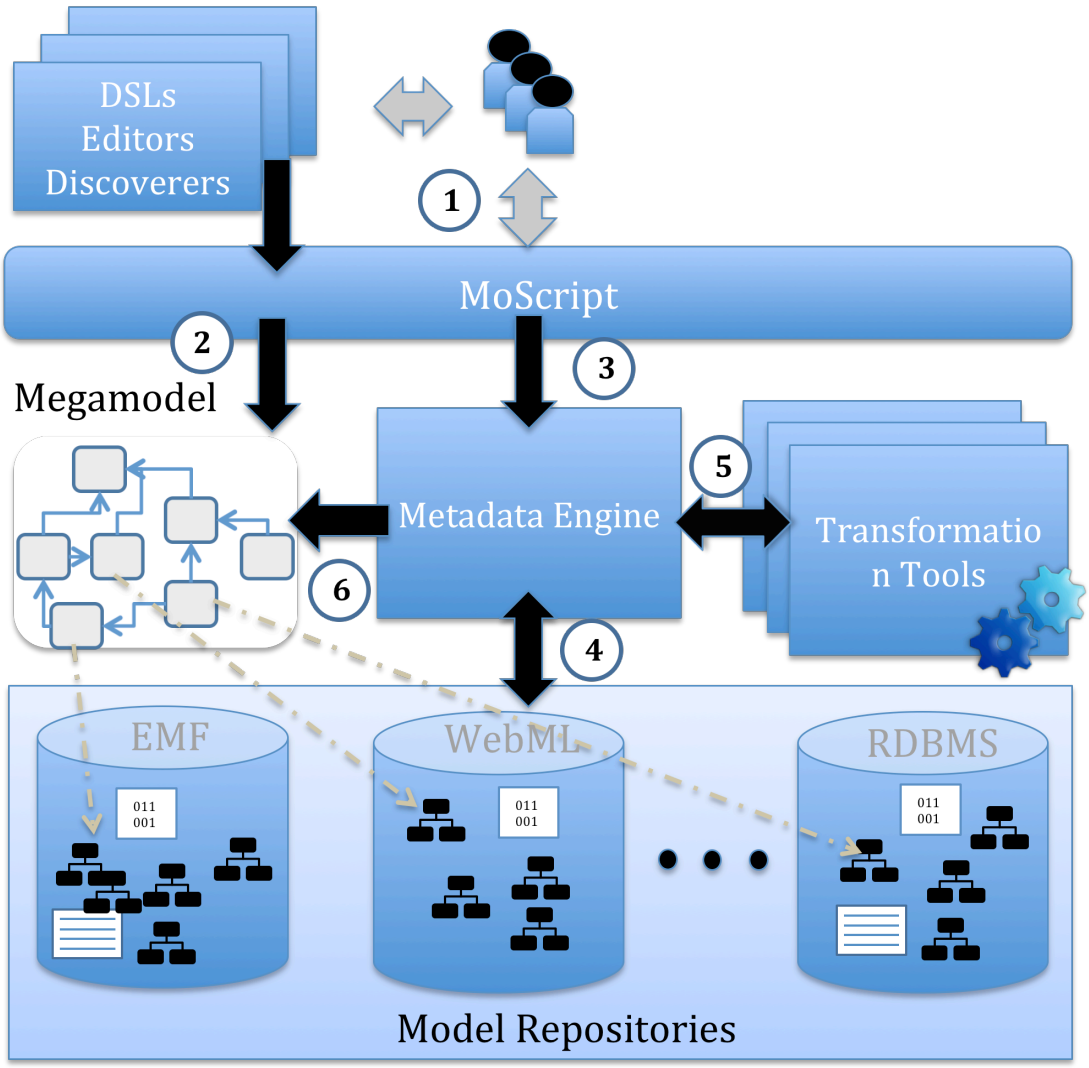
```
remove <identifier> from <Megamodel>
```

The `remove` statement allows removing models from the Megamodel. The `identifier` is the name of the Model we want to remove from the Megamodel. The `Megamodel` argument is a model element that represents the Megamodel, which we want to remove the model from.

4. THE MOSCRIPT LANGUAGE

We already talked about a strategy from a user point of view, which intends to cope with the complexity due to large amounts of models and the complexity introduce by the MDE approach. In this section, we will show how we could materialize this strategy in terms of a DSL called MoScript and a concrete architecture for supporting it.

4.1 ARCHITECTURE



MoScript is supported by architecture with several components: A DSL (MoScript), a metadata engine, the Megamodel, transformations tools, external tools and model repositories. These components are described in detail below.

Megamodel: As mentioned in section 3.1 the Megamodel describes the models repository. MoScript uses the Megamodel for being able to query and browse the model repository in a coherent manner. MoScript uses the Megamodel to know how the artefacts in the repository are interrelated and what kind of artefacts they are.

Metadata Engine: It provides MoScript a simple and homogenous interface for retrieving and storing models, for the application of transformations on models, and for models itself. This component is in charge of:

- Keeping in sync the Megamodel and the models repository when artefacts come in or out from it.
- Providing models and transformation definitions location transparency.
- Providing modelling frameworks technology transparency (model handlers).
- Providing transformation tools technology transparency.
- Ensuring integrity between the Megamodel and the models in the repository, when artefacts come in or out from it.
- Protecting models from unauthorized access.
- Indexing models content for fast retrieval
- Providing models with appropriate model handlers to increase the support for scalability, e.g. small models are retrieved with in memory model handlers, big sized model are retrieved with lazy model handlers, remote models are retrieved with remote model handlers etc.

MoScript DSL: Is a DSL that facilitates the models manipulation. It uses OCL for browsing Megamodels and for retrieving the models metadata. MoScript uses the metadata engine for retrieving physical artefacts, executing transformations and for register new models or removing existing ones. A detail explanation of MoScript will be given in the next section.

DSLs, Editors and Discoverers: In some cases models may not be derived from other models e.g when creating a models by hand. In those cases external tools (e.g. DSLs, Editors, Discoverers etc.) may use MoScript for register or unregister models outiside from MoScript.

Model Repositories represent consistent sets of models, which have a common storage method. A model repository may be a file system based, a database repository, etc. A repository provides interfaces for adding, accessing or deleting models from the repository, which are used by the metadata layer. Most of the model repositories or the models itself are linked to specific model handlers and cannot work with models in other formats (for instance, Teneo[??], Netbeans MetaData Repository[??], Adaptive Metadata Manager[??], etc.). Model Repositories may exist locally with respect to the Metadata Engine or be distributed.

Transformation Tools represent the different transformation tools that may be plugged to the architecture. These tools may be model-to-model (m2m) transformation tools, model-to-text (m2t), text-to-model (t2m) or generally any kind of transformation tools. To this intent the Metadata Engine and the Megamodel are extended to support each transformation tool particularities. According to the information in the Megamodel, the Metadata Engine uses the appropriate transformation tool to run the transformations that should be applied to the models and send the result back to MoScript. Tools for matching, comparing, merging etc., are also considered transformations tools, because they take models as input and produce new views of them.

The information flow that takes place between the architecture components when performing models manipulations with MoScript is denoted by the numbers in figure 4.1. (1) Users write, compile and run a MoScript query or program. (2) MoScript queries the Megamodel for retrieving the models elements (metadata) which describe the models and relations involved in the process. (3) MoScript asks the Metadata engine to apply the transformations to the models providing all the information about them (metadata). (4) MoScript retrieves the models and transformation definitions (using the information stored in the model elements like location, protocol, access restrictions etc). (5) The metadata engine applies the transformation to the retrieved models and (6) registers the new models in the Megamodel if necessary. Finally the metadata engine returns to MoScript the models or model elements, which constitute the result of the program execution.

4.2 MOSCRIPT ABSTRACT SYNTAX

The abstract syntax represents the syntactic structure of the source code of a language. It is considered abstract because it does not represent all the details that appear in the syntax of source code, such as brackets or symbols that delimit blocks of code. Abstract syntax however, represents the data structure of a language by means of data types (Fowler).

The abstract syntax is divided into two main packages: a OCL package and a MoScript package. The OCL package contains OCL expressions, which enable the navigation and querying of the Megamodel. The MoScript package contains language statements for control flow, library declarations and program sections. MoScript statements make use of OCL expressions to obtain the data to work with. The complete lists of abstract syntax elements are shown in Figure 3.

- OCL
  - OclExpression -> LocatedElement
  - VariableExp -> OclExpression
  - SuperExp -> OclExpression
  - PrimitiveExp -> OclExpression
  - StringExp -> PrimitiveExp
  - BooleanExp -> PrimitiveExp
  - NumericExp -> PrimitiveExp
  - RealExp -> NumericExp
  - IntegerExp -> NumericExp
  - CollectionExp -> OclExpression
  - BagExp -> CollectionExp
  - OrderedSetExp -> CollectionExp
  - SequenceExp -> CollectionExp
  - SetExp -> CollectionExp
  - TupleExp -> OclExpression
  - TuplePart -> VariableDeclaration
  - MapExp -> OclExpression
  - MapElement -> LocatedElement
  - EnumLiteralExp -> OclExpression
  - OclUndefinedExp -> OclExpression
  - PropertyCallExp -> OclExpression
  - NavigationOrAttributeCallExp -> PropertyCallExp
  - OperationCallExp -> PropertyCallExp
  - OperatorCallExp -> OperationCallExp
  - CollectionOperationCallExp -> OperationCallExp
  - LoopExp -> PropertyCallExp
  - IterateExp -> LoopExp
  - IteratorExp -> LoopExp
  - LetExp -> OclExpression
  - IfExp -> OclExpression
  - VariableDeclaration -> LocatedElement
  - Iterator -> VariableDeclaration
  - Parameter -> VariableDeclaration
  - CollectionType -> OclType
  - OclType -> OclExpression
  - Primitive -> OclType
  - StringType -> Primitive
  - BooleanType -> Primitive
  - NumericType -> Primitive
  - IntegerType -> NumericType
  - RealType -> NumericType
  - BagType -> CollectionType
  - OrderedSetType -> CollectionType
  - SequenceType -> CollectionType
  - SetType -> CollectionType
  - OclAnyType -> OclType
  - TupleType -> OclType
  - TupleTypeAttribute -> LocatedElement
  - OclModelElement -> OclType
  - MapType -> OclType
  - OclFeatureDefinition -> LocatedElement
  - OclContextDefinition -> LocatedElement
  - OclFeature -> LocatedElement
  - Attribute -> OclFeature
  - Operation -> OclFeature
  - OclModel -> LocatedElement
- MoScript
  - LocatedElement
  - Unit -> LocatedElement
  - Library -> Unit
  - Query -> Unit
  - Program -> Unit
  - ProgramBody -> LocatedElement
  - Helper -> LocatedElement
  - LibraryRef -> LocatedElement
  - Statement -> LocatedElement
  - ExpressionStat -> Statement
  - BindingStat -> Statement
  - IfStat -> Statement
  - ForStat -> Statement
  - SaveStat -> Statement
  - RemoveStat -> Statement

Figure 3: MoScript abstract syntax packages

4.2.1 Program structure

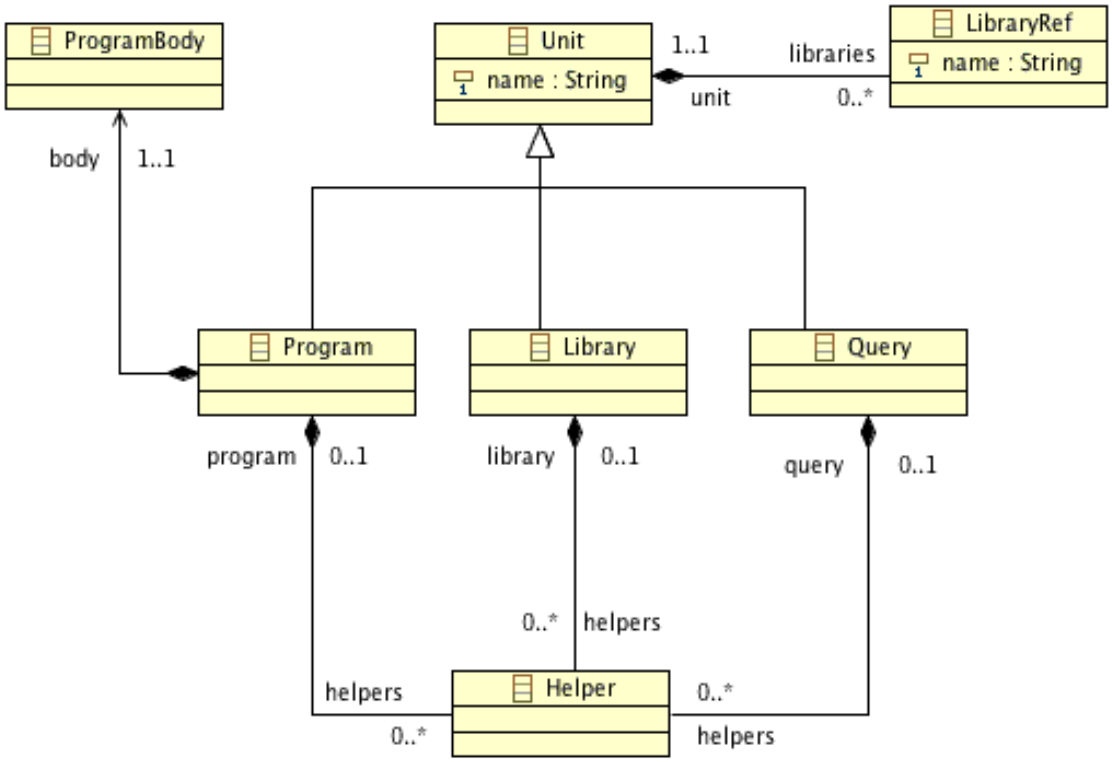


Figure 4: MoScript programs structure

As shown in Figure 4, MoScript provides three kinds of program modules: queries, programs and libraries.

Query modules use OCL expressions exclusively. With these OCL expressions it is possible to query the model that describes the model repository (Megamodel) and to call operations on model elements such as the operations for querying inside models and for applying transformations to them described (section 3.3.1). Keeping in mind the side effects free philosophy of OCL, a query cannot modify the model that describes the repository neither the model repository itself. When a transformation is executed, the resulting models are not persisted and their live end when the query execution ends. Query modules are useful for testing purposes. Using a query, it is possible to navigate the Megamodel, apply transformations to models and check the results of the transformations without modifying the model repository.

Program modules combine OCL expressions with statements. The statements enable MoScript to modify the model that describes the model repository (Megamodel) and the repository itself. It allows creating, modifying or deleting elements, which describe the models of the repository and their interrelationships. It also allows the serialization of newly produced models, the replacement of existing ones or their deletion from the repository.

Program modules are required to evolve the repository. Once a transformation is executed and the resulting model has been validated, they should be persisted in the model repository and also registered in the Megamodel for further use.

Library modules allow us to factorize OCL queries for future reuse. These queries may be grouped in library modules, which in turn may be imported by query or program modules. These query units are called Helpers. The kinds of Helper that may be used frequently are, e.g., the queries for navigating the Megamodel. As model repositories may have a well-defined structure, the ways to navigate across them may be also well defined. Model repository navigations may be represented by OCL queries that navigate the Megamodel to retrieve specific kinds models.

4.2.1 Operations

In section 3.3.1, we proposed a set of operations for inspecting models and applying transformations. As we explained, these operations are related to the model elements of the Megamodel. Because we use OCL to query the Megamodel, these operations correspond to the `OperationCallExp` element of the OCL abstract syntax. As shown in Figure 5, the `OperationCallExp` element has a name, arguments and an OclType (inherited from OclExpression), which corresponds to the type of the result of the expression evaluation. The figure also shows that the expressions may be composed to produce more complex expressions.

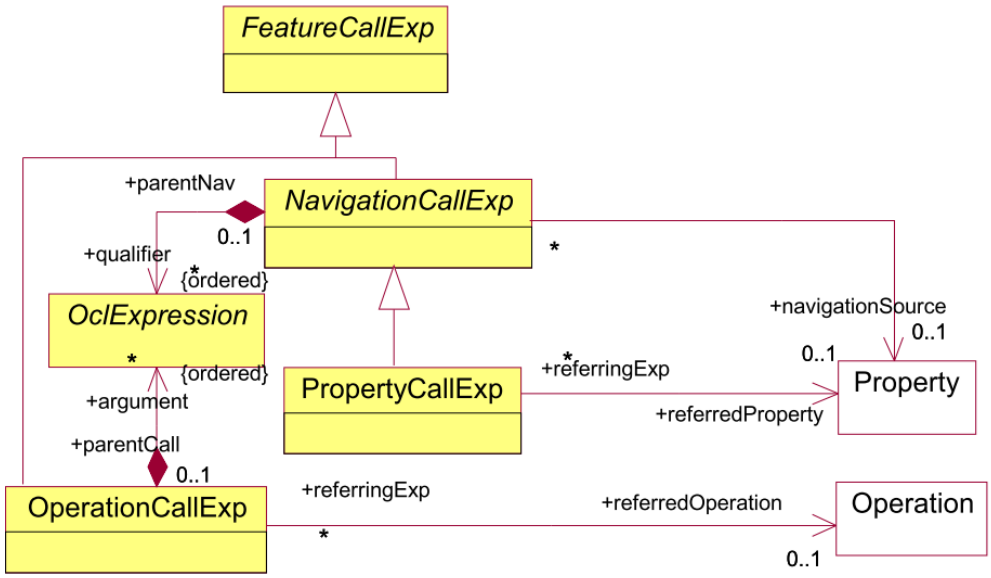


Figure 5 OCL abstract syntax excerpt

4.2.1 Statements

Statements are used exclusively in program modules. We propose statements for control flow, variable definition and Megamodel evolution, as shown in section **Erreur ! Source du renvoi introuvable.**. Statements are executed sequentially and rely on OCL queries to retrieve the data from the Megamodel and work with them.

The set of statements of the language is shown in figure 3.4.1

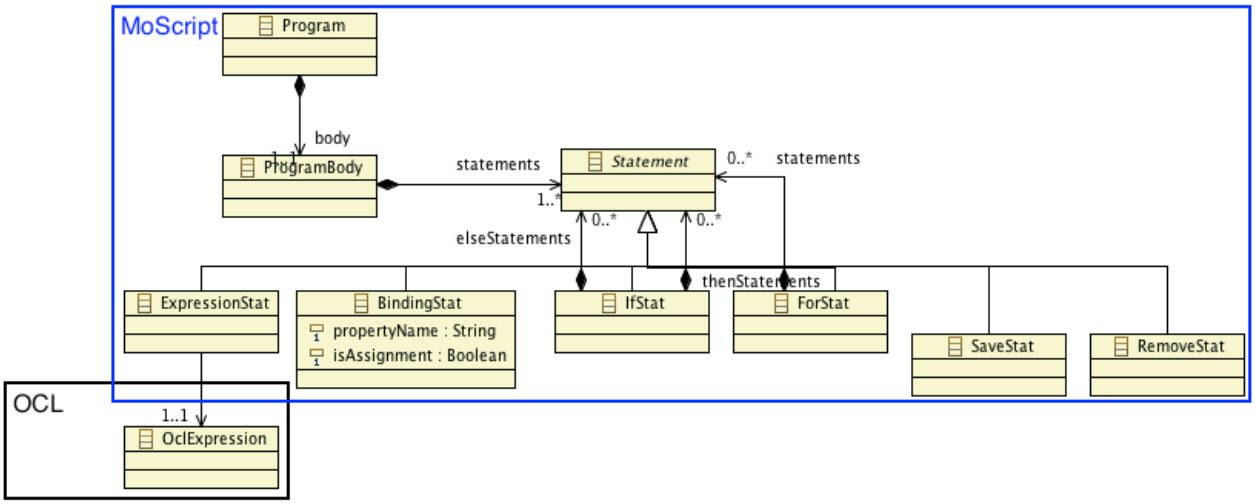


Figure 6 MoScript statements

Note that MoScript statements are linked to OCL through the OCLExpression.

4.3 CONCRETE SYNTAX

4.3.1 Program structure

For the purpose of this document, we will give a briefly description of the concrete syntax in terms of the program modules structure, to give an idea of what the DSL could look like.

As shown in Table 1, a query module must have a name and its result is the result of the evaluation of an OCL expression.

```
1   query query_name = OclExpr;
2
3   uses library1
4   uses library2
5   …
6   uses libraryn
7
8   helper [context context_type] def : helper_name1(parameters) : return_type = OclExpr;
9
10  helper [context context_type] def : helper_name2(parameters) : return_type = OclExpr;
11  …
12  helper [context context_type] def : helper_namen(parameters) : return_type = OclExpr;
```

Table 1 Query module structure

```
1   program program_name
2
3   uses library1
4   uses library2
5   …
6   uses libraryn
7
8   [using {
9       variable1 : type = OclExpr;
```

```
10      variable2 : type = OclExpr;
11      ...
12      variablen : type = OclExpr;
13  }]
14
15  do {
16      …
17      variablen <- OclExpr;
18      …
19      save OclExpr to OclExpr as OclExpr in OclExpr;
20      …
21      remove OclExpr;
22      …
23      if(OclExpr) {
24          …
25          save …
26          …
27          remove …
28      }
29      else {
30          …
31          save …
32          …
33          remove …
34      }
35      …
36      …
37
38      for(variable : OclExpr) {
39          …
40          save …
41          …
42          remove …
43      }
44  }
45
46  helper [context context_type] def : helper_name1(parameters) : return_type = OclExpr;
47
48  helper [context context_type] def : helper_name2(parameters) : return_type = OclExpr;
49  …
50  helper [context context_type] def : helper_namen(parameters) : return_type = OclExpr;
```

Table 2 Program module structure

A program, as shown in Table 2, has two sections, the `using` and `does` sections. The `using` section is optional, and is used for declaring variables and assigning their initial value. The `do` section is mandatory and is the core of the program. In it, the statements with side effects are used in combination with the control flow statements and OCL queries.

```
1   library library_name;
2
3   uses library1
4   uses library2
5   …
6   uses libraryn
7
8   helper [context context_type] def : helper_name1(parameters) : return_type = OclExpr;
9
10  helper [context context_type] def : helper_name2(parameters) : return_type = OclExpr;
11  …
12  helper [context context_type] def : helper_namen(parameters) : return_type = OclExpr;
```

Table 3 Library module structure

A library, as said before, contain helpers and may include other libraries. A Helper may be defined in the context of an element type of the Megamodel or in the context of the program module. Note that we took inspiration from ATL for defining the abstract and concrete syntax of MoScript.

5. QUALITY ATTRIBUTES AND CONSIDERATIONS

In this section we present the set of desirable quality attributes, which MoScript and its supporting platform should have.

5.1 EXTENSIBILITY

One of the main objectives of the global solution is to allow extensibility. The motivation for this is the current explosion of DSLs for model transformation and other modelling tasks, as well as the high diversity of tools and model kinds available in the market. Thus, the solution should be design based on abstractions of them to easily accommodate to its different requirements (e.g. m2m, m2t and t2m), and integrate them in a consistent and uniform way.

5.2 USABILITY

Although the model manipulations that can be made with the solution are far from being simple, the corresponding user interface should stay as simple as possible. It must make less complex for the users to automate specific tasks by using a dedicated focused syntax and a supporting platform that takes care of e.g. errors recovery, transactions, memory management etc. The use of a well-known and accepted query language such as OCL as the base of MoScript, is one concrete strategy for guaranteeing its usability. In any case, for having success in this aspect, it is fundamental to obtain feedback from several external users.

5.3 PERFORMANCE

Handling large numbers of models implies a performance issue specially if models are also large and with complex structure. Memory management when working with many models at the same as well as exhaustive model searches by inspecting them along large and distributed repositories must be treated with special care. The solution must then include mechanisms such as indexing, models lazy loading, garbage collection etc., to be able to locate models into acceptable amounts of time and to avoid enormous memory footprints.

5.4 SECURITY

The platform must ensure models security, especially because it will provide access to several distributed model repositories. Models must be protected against unauthorized access, use, disclosure, corruption, modification, or destruction in order to ensure their integrity, confidentiality and availability. For this purpose, the platform should count with mechanism such as authentication, authorization and transaction management at both, at the model level and the model element level.

6. CONCLUSIONS AND RELATED WORKS

We introduced MoScript a DSL that greatly simplifies the manipulation of large quantities of models (i.e. repositories of software artefacts) based on megamodels. MoScritpt combines both the declarative and imperative approaches. The declarative part uses OCL to query megamodels and may execute side effects free operations. The imperative part combines queries and statements for evolving the megamodel content, i.e. for evolving the corresponding models repositories. MoScript works on top of a metadata engine, which provides location and technology transparency of models and model handlers as well as other services such as concurrency handling and security policies.

This work may be compared with previous works that propose scripting or orchestration workflow facilities for modelling tasks and with other works that propose platforms for (generic and global) model management.

On the one hand almost all transformation languages and tools come with a scripting language for chaining transformations and running these transformation chains in a batch mode. For instance, ATL [ATL] provides scripting by extending an external build tool (Apache Ant) with specific tasks for transformation, injection and extraction of models, as a solution mainly targeted for composing transformations. A similar approach is followed by the Epsilon platform [EPSWF], which also extends Ant for orchestrating model management operations such as model validation, transformation, comparison, merging and model-to-text transformations. Epsilon platform include facilities for transaction management and models disposal. The openArchitectureWare[5] Workflow allows specifying a workflow for chaining m2m and m2t transformations by means of an XML. RubyTL [RubyTL] a Ruby[6] based transformation language, relies on Rake[7] for running custom defined tasks to execute to model-to-model and model-to-code transformations.

We could continue mentioning other several transformation languages that follow the same approach, but the important point here is, that none of them are model oriented neither take advantage of a semantic view of the repositories. This implies that: 1) all necessary metadata (e.g., source and target metamodels of transformations) must be encoded into the scripts, and 2) there is not a way to keep track of the evolution of the repositories e.g., newly produced elements, which are only created as files without attached metadata 3) there is no way to browse the model repositories consistently, 4) compositions defined in such scripts require the specification of many details every time that are used that could be avoided if they where predefined.

On the other hand there are the model management platforms. Model Management [MOMGM] applies operations to models as whole rather than to their individual elements for simplifying the work with models. In [THGLB] a script is proposed for the declarative combination of specific and generic model management operations. Rondo [RONDO] a platform for Generic Model Management supports the execution of model management scripts that are written using high-level operators. The idea is to provide generic operations such as match, merge, extract and compose models. Although we use the same philosophy of handling models as a whole, we do not provide generic operations. Instead we provide a mechanism for new DSLs to provide their own operations that may be generic or specific to a particular DSL. We also have a fundamental difference, which is that we work on top of a model that gives semantics to the repository (megamodel) so we have the possibility to use a query language for first finding models and then apply operations to them. We also provide means to evolve the system and follow its evolution, characteristics that are not mentioned in the former approaches.

[5] http://www.eclipse.org/modeling/emft/?project=mwe
[6] Ruby programming languange: http://www.ruby-lang.org/
[7] RAKE – Ruby Make: http://rake.rubyforge.org/

There are other works such as [SRCHMO] and [MOOGLE] which handle large model repository artefacts. They also use an index as metadata model, which points to each model of the repository. Those works provide means for finding models by their internal characteristics. They differ from our approach in that they do not work on top of a model management tool, thus the results obtained from a model search, cannot be processed or combined with other models. The results are usually shown as a list of model names, which at most can be downloaded.

7. REFERENCES

[AM3] Allilaire, F., Bézivin, J., Brunelière, H., & Jouault, F. (2006). Global Model Management in Eclipse GMT/AM3. *Procedings of the Eclipse Technology eXchange (eTX) workshop at ECOOP*.

[ATL] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming 72 (1-2), 31 – 39 (2008), special Issue on Second issue of experimental software and toolkits.

[EPSWF] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. A Frame- work for Composing Modular and Interoperable Model Management Tasks. In Proc. Workshop on Model Driven Tool and Process Integra- tion (MDTPI), ECMDA, Berlin, Germany, June 2008.

[GMM] Bézivin, J., Jouault, F., and Valduriez, P., On the Need for Megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19 th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.

[GMMDISCO] Mahé, V., Jouault, F., Brunelière, H.: Megamodeling software platforms: Automated discovery of usable cartog-raphy from available metadata. In: Proc. Of REM09 (workshop), Organized in conjunction with WCRE09 (2009).

[GMMPERF] Fritzsche, M., Brunelière, H., Vanhooff, B., Berbers, Y., Jouault, F., Gilani, W.:Applying megamodelling to model driven performance engineering. In: ECBS '09: Proc. of the 2009 16th Annual IEEE Int.Conf. and Workshop on the Engineering of Computer Based Systems. pp. 244–253. IEEE Computer Society, Washington, DC, USA (2009).

[MDRE] Spencer Rugaber, Kurt Stirewalt, "Model-Driven Reverse Engineering," IEEE Software, pp. 45-53, July/August, 2004

[MOMGM] Melnik, S.: Generic Model Management: Concepts And Algorithms (Lecture Notes in Computer Science). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2004) 17. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)

[MOOGLE] Lucredio, D., de M. Fortes, R.P., Whittle, J.: Moogle: A model search engine. In: Proc. of Model Driven Engineering Languages and Systems, 11th Int.Conf., MoDELS 2008. pp. 296–310. Springer-Verlag (2008)

[MDMCS] Barbero, M., Jouault, F., B´ezivin, J.: Model driven management of complex systems: Implementing the macroscopes vision. In: Proc. of ECBS08, IEEE Computer Society (2008)

[OMGXMI] OMG: XML Metadata Interchange (XMI®)

[PRAGM] Selic, B.: The pragmatics of model-driven development. IEEE Softw. 20(5), 19–25 (2003)

[RONDO] Melnik, S., Rahm, E, Bernstein, P.A: Rondo: A Programming Platform for Generic Model Management, ACM SIGMOD Int. Conf., San Diego, Ca, 2003.

[RubyTL] Rensink Arend, Warmer Jos, Cuadrado Jesús, Molina Jesús, Tortosa, Marcos. RubyTL: A Practical, Extensible Transformation Language. Model Driven Architecture – Foundations and Applications. Lecture Notes in Computer Science 2006. Springer Berlin / Heidelberg

[SILVBUL] Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. In: Proc. of the IFIP Tenth World Computing Conf. p. 10691076 (1986)

[SRCHMO] Bozzon, A., Brambilla, M., Fraternali, P.: Searching repositories of web application models (to appear). In: Web Engineering, 10th Int.Conf., ICWE 2010 Proc.

[THGLB] Reiter, T., Altmanninger, K., Retschitzegger, W.: Think global, act local: imple- menting model management with domain-specific integration languages. In: MoDELS'06: Proc. of the 2003 international conference on Models in software engineer- ing. pp. 263–276. Springer-Verlag, Berlin, Heidelberg (2007)

[UniTI] Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI: A Unified Transformation Infrastructure. In: En-gels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)