# MoScript: A DSL for querying and manipulating model repositories

Wolfgang Kling[1], Frédéric Jouault[1], Dennis Wagelaar[3*], Marco Brambilla[2], and Jordi Cabot[1]

[1] AtlanMod, INRIA & École des Mines de Nantes
`{wolfgang.kling,frederic.jouault,jordi.cabot}@inria.fr`
[2] Politecnico di Milano, Dipartimento di Elettronica e Informazione
`marco.brambilla@polimi.it`
[3] Vrije Universiteit Brussel, Software Languages Lab
`dennis.wagelaar@vub.ac.be`

**Abstract.** Megamodels has been proposed as a mechanism to describe large number of modelling and non-modelling artefacts (models, transformations, source code, binary files, etc.) and their complex interrelations, typically required in any non-trivial development project. Despite its growing acceptance, there is a lack of adequate tool support to search, inspect, manipulate, and combine, at a glance, the modelling artefacts represented by Megamodels. In this paper we introduce MoScript: a generic and extensible infrastructure and domain-specific language for Megamodelling. With MoScript users can express queries (based on model content, structure, relationships, and behaviour derived through on-the-fly simulation) to retrieve models from heterogeneous model repositories, manipulate them (e.g., by running transformations on sets of models), and store them back in the repository.

**Keywords:** DSL, OCL, Model Management, Megamodel

## 1 Introduction

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

MDE for complex systems [5] is a typical example of this situation. In the model driven development of those systems, every artefact (e.g. requirements specifications, analysis and design documents, implementation artefacts, etc.,) is a model. Apart from being numerous, these artefacts are often large, heterogeneous, interrelated, with complex internal structure, and possibly stored in distributed model repositories.

* The author's work is funded by a postdoctoral research grant provided by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders)

MDE is partly to blame for this complexity, as it introduces new artefacts to deal with, such as models, metamodels, transformation models, and transformations engines. Whereas having special-purpose metamodels allows for reducing model complexity, the interrelations between transformations, models, and metamodels can become very complex. Global Model Management (GMM) aims to address this complexity problem by providing an explicit representation of the modelling artefacts and their interrelations, in a model called *Megamodel* [12].

Megamodels have been used in several domains such as performance engineering [14], ontologies [4], architecture frameworks [24] among others, for understanding the complex arrangements and relations between entities and bridging between their underlying technologies.

However, there is no generic approach for exploiting Megamodels, i.e query Megamodels and manipulate, at a glance, the artefacts it represents (e.g. loading/saving models, executing transformations, bridging between technical spaces, etc.).

In this paper, we propose MoScript a Megamodel agnostic platform and a textual DSL (domain-specific language) for accessing and manipulating modelling artefacts represented in a Megamodel.

MoScript allows to write queries that retrieve models from a repository, inspect them, invoke services on them (e.g. transformations), and to register newly produced models back to the repository. MoScript also facilitates the description and automation of complex modelling, involving several consecutive manipulations on a set of models. As such, the MoScript language can be used for modelling task and/or workflow automation.

The MoScript architecture includes an extensible metadata engine for resolving and accessing modelling artefacts and invoke services from different transformation tools.

The remainder of this paper is structured as follows. Section 2 describes the supporting architecture for MoScript. Section 3 presents the MoScript language. Section 5 describes how we implemented MoScript. Section 6 compares our work with other, related approaches. Finally, section 7 presents our conclusions and future work.

## 2 The MoScript Architecture

Fig. 1 shows an overview of the MoScript architecture, comprising both the basic components and information flows.

### 2.1 Architecture Components

The MoScript architecture is composed of six components: the MoScript DSL, a Megamodel, a metadata engine, model repositories, transformation tools, and external DSLs, editors, and discoverers, as shown in Fig. 1 and described next.

**MoScript:** A DSL, which serves as an interface between the users and the modelling artefacts repositories. Users write and run their MoScript scripts for

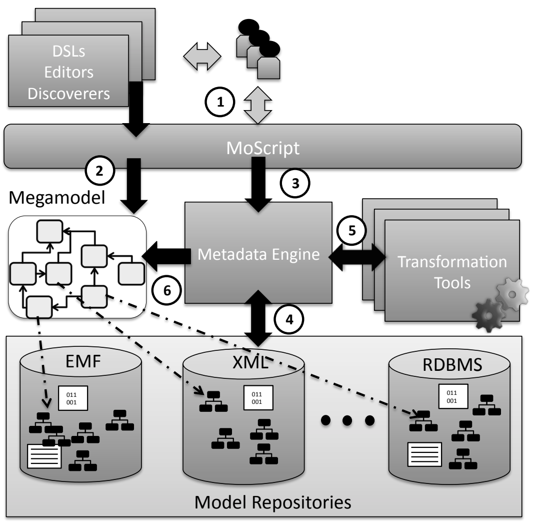


**Fig. 1.** The MoScript architecture.

retrieving modelling artefacts and perform modelling tasks (e.g. inspect, transform, match, etc.) with them. MoScript uses the Megamodel as cartography to navigate the modelling artefacts repositories and to know which kind of models are stored in them. As result of the manipulations, new modelling artefacts may be created in the repositories or existing modelling artefacts may be removed.

**Megamodel:** A model which describes modelling artefacts within repositories (e.g. their location, kind, format, etc.), and how they are interrelated. The Megamodel is a regular model, thus it conforms to a metamodel, which is shown in Fig. 2. Elements in the Megamodel represent modelling artefacts or relationships. For instance the ***Entity*** element represents any MDE (i.e. artefacts that depend on well defined grammars) and non-MDE artefact (such as non structured documents, tools, libraries, etc).

The basic MDE artefacts supported by the Megamodel are: ***MetaMetaModel*s (M3)**, which represent models conforming to themselves; ***Metamodel*s (M2)**, which represent models conforming to metametamodels; and ***TerminalModel*s (M1)**, which represents models conforming to metamodels but no other model conforms to them. Examples of terminal models are Megamodels, transformation models, and weaving models. As MDE artefacts may be translated into models (e.g. XMI files) from this point forward we are going to refer us to MDE artefacts as models.

Relationships between artefacts (MDE and non-MDE) are represented by the ***Relationship*** concept. For instance, a ***Transformation*** is a directed relationship between a ***TransformationModel*** and one or more ***ReferenceModel*s** (metamodels or metametamodels). The *TransformationModel* is the representa-

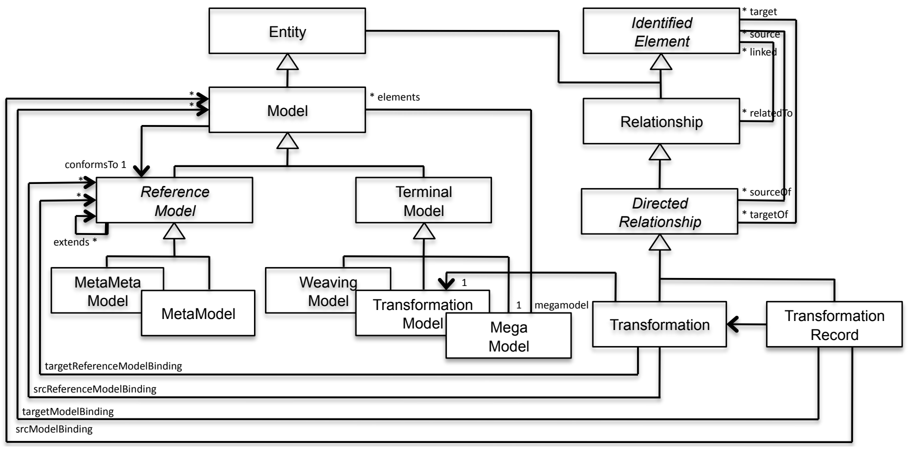


**Fig. 2.** Part of the core metamodel for Megamodels.

tion of the source code of the transformation while the reference models restrict the type of input and output models the transformation may be applied on.

A ***TransformationRecord*** is another kind of directed relationship. A *TransformationRecord* associates a *Transformation* with a set of input and output models. As we will see later, it is useful for rerunning transformations without giving any additional input.

**Metadata Engine:** Provides services to MoScript for retrieving models, executing transformations and (un)register models (from) into the Megamodel. The Metadata Engine exposes a homogeneous interface, which provides location and technology transparency of models and transformation tools. It also protects models from unauthorized access and modifications.

The metadata engine uses the Megamodel for run-time type checking. For instance, the metadata engine can check if the transformations are being applied to the right models. It can also check if the resulting model of a transformation execution, may be used for transforming other models, like when executing a High Order Transformations (HOT). In a previous work [26], we demonstrate the viability of this type checking.

**Model repositories:** Contain models stored in different formats, e.g. XMI, XML, RDBMS, etc. Model repositories may reside in different physical locations, such as a local filesystem, a remote WebDAV server, the cloud, etc.

**Transformation Tools:** Model-to-model (M2M), model-to-text (M2T) or text-to-model (T2M) transformation tools provide transformation services. They implement a generic interface, thus all transformation tools services can be invoked the same way regardless the technology behind. Transformation tools may include QVT [1], ATL [16], EMF Compare[4], JET[5], Xpand[6], etc.). In general, any tool that produces a new view of a modelling artefact (e.g. documentation generators, compilers, file comparison tools, etc.) is considered a transformation tool. If any transformation tool does not fit the generic interface it may extend it along with the metamodel of the Megamodel, for adding new services and concepts.

**DSLs, Editors and Discoverers:** These tools create models outside the MoScript context and need to contribute them to the Megamodel. They can (un)register models (from) into the Megamodel through MoScript, and they can query the Megamodel as well.

## 2.2 Architecture Information Flow

The information flow that takes place between the architecture components when performing models manipulations with MoScript, is denoted by the numbers in Fig. 1. (1) Users write and run a MoScript program. (2) MoScript queries the Megamodel to retrieve the model elements (metadata) describing the models and transformations involved in the process. Then, it (3) asks the Metadata Engine to apply selected transformations on the selected models. (4) The metadata engine retrieves[7] from the repositories the models and transformation definitions (using the information stored in the Megamodel elements, such as location, protocol, access restrictions etc). (5) Then it executes the transformations with the models and (6) registers the resulting models in the Megamodel if necessary. Finally, the metadata engine returns the Megamodel model elements resulting from the program execution to MoScript for further processing.

## 3 The MoScript Language

A Megamodel is a regular model, thus it can be navigated with OCL as any other model. Examples of queries on Megamodels may include the selection of all the transformations of the repository to list their physical location, or more complex queries, such as selecting all the models that conform to a specific kind of metamodel, as shown in the example below.

```
Model::allInstances()->select(m | m.conformsTo.kind = 'Java')
```

The result of the query above is a collection (`Collection(Model)`) of model elements which describe the models of the repository. Although the result is useful to precisely know which models are stored in the repository, it is merely informative. This means that with standard model query languages like OCL, is not possible to use the results of queries to Megamodels to directly manipulate

[4] http://www.eclipse.org/modeling/emf/?project=compare#compare
[5] http://www.eclipse.org/emft/projects/jet/
[6] http://www.eclipse.org/modeling/m2t/?project=xpand
[7] Retrieving the model means that an interface is built and exposed for accessing the model. It does not necessarily means that the whole model traverses the network

(e.g. check, transform, match etc.) the models they describe. This is due to the fact that OCL does not handle models as a bootstrapped concept.

The MoScript language intends to fill this gap. MoScript is an OCL-based [2] scripting language for model-based tasks and workflow automation that works on top of a Megamodel. It proposes three main contributions: **(1) Model dereferencing**, to retrieve models represented by metadata in a Megamodel; **(2) Native library of operations** to perform common model manipulation tasks with the dereferenced models; **(3) Combination of dereferenced artefacts and operations with OCL** to provide powerful expressiveness.

Model dereferencing is applicable to all the Megamodel elements that have a separated physical representation in the system and may be accessed through a locator (e.g., an URI). As a result of the dereferencing, an interface of the artefact is loaded in memory and exposed for being used through an OCL *ModelElement* type. As OCL is working on top of the Megamodel, this OCL *ModelElement* type always corresponds to an element type of the Megamodel (*TerminalModel*, *Metamodel*, *Transformation* etc.).

Furthermore, a set of operations are associated to those model element types for being invoked from OCL and which in turn may be composed to produce more powerful operations.

Next, we will explain MoScript abstract and concrete syntax, as well as the native library's operations.

## 3.1 MoScript Abstract and Concrete Syntax

The MoScript DSL has a semantic model [13] and an abstract and concrete syntax [23].

The MoScript's **semantic model** is the Megamodel. It is the place where the domain concepts are stored and is independent from the language constructs. The core concepts of the Megamodel have been covered in section 1.

The **abstract syntax** as shown in figure 3, is divided in two packages. The OCL package and the MoScript package. As MoScript is OCL-based, the complete OCL abstract syntax (not showed) is included as part of the language.

The `OperationCallExp` from the OCL package has been extended with a set of operations we call *operations without side effects*. These operations are used to perform several modelling tasks that **do not modify the model repository or the Megamodel**.

Operations without side effects are divided in four categories: query operations (`QueryOp`), operations for transformations between same technical spaces (`TransformOp`), operations for transformations between different technical spaces (`ProjectOp`) and operations for checking the models state (`StateCheckOp`). For each category MoScript provide several concrete operations, which will be explained in the next subsection.

Furthermore, the MoScript package provides a set of *operations with side effects* (`SaveOp, RemoveOp and RegisterOp`). These operations allow the modification of the repository or the Megamodel. Side effects operations may embed OCL expression and therefore side effects free operations. This is why the

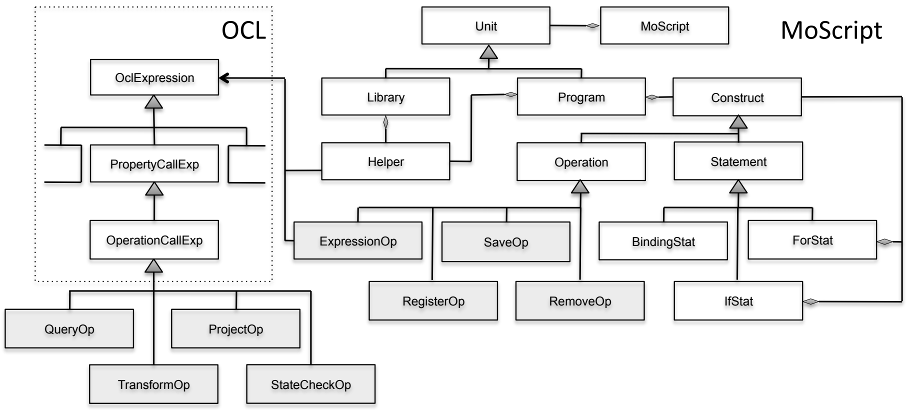


**Fig. 3.** MoScript abstract syntax main concepts

`ExpressionOp` is related to the `OCLExpression`. This relation allows to carry out complex models manipulations before persisting them in the repository and the Megamodel. However, the opposite (embed side effects operations within OCL expression) is not permited. OCL expression do not know side effects operations, thus respecting the OCL side effects free philosophy.

MoScript also provides a statement for variable binding and declaration (`BindingStat`) and **`for`** (`ForStat`) and **`if`** (`IfStat`) statements for control flow.

MoScript has two kinds of modules: **libraries** and **programs**. A library contains **helpers**, which are used to modularise complex OCL expressions. Libraries may be in turn imported by programs or by other libraries.

The **concrete syntax** of MoScript is summarised in the following listing:

```
program program_name

uses library1
uses ...

[ using {
  variable1 : type = OclExpr;
  variable2 ...
}]

do {
  variable1 <- OclExpr;

  model1.save(...);
  model2.remove(OclExpr);
  ...
  megamodel.register(...);

  if...
  for..
}

helper context OclAny def: helper_name(params) : return_type;
...
```

A program has two sections, the `using` and `do` 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, operations with and without side effects are used in combination with the control flow statements and OCL queries to perform modelling artefacts manipulations.

The complete definition of the concrete syntax is expressed in the TCS language [17], and can be found at:
`http://www.emn.fr/z-info/atlanmod/index.php/Moscript`.

In the following subsections, we will discuss in detail the operations with and without side effects provided by MoScript and summarised in table 1.

| **Operations without side effects** |
|---|
| Model :: allContents : Collection(OclAny) |
| Model :: allContentsRoots() : Collection(OclAny) |
| Model :: allContentsInstancesOf(type_name : String) : Collection(OclAny) |
| Model :: allContentsInstancesOf(type : OclAny) : Collection(OclAny) |
| Transformation :: applyTo(inputModels : Sequence(Model)) : TransformationRecord |
| Transformation :: applyTo(inputModels : Map(String, Model)) : TransformationRecord |
| TransformationRecord :: run() : TransformationRecord |
| Model :: inject() : Model |
| Model :: extract() : Model |
| Model :: available() : Boolean |
| Model :: isDirty() : Boolean |
| **Operations with side effects** |
| Model :: save(mm : Megamodel, id : String, locator : String) |
| Model :: remove() |
| MegaModel :: register(id : String, locator : String) : Model |

**Table 1.** MoScript operations summary.

## 3.2 Operations without side effects

This subsection describes in detail the operations without side effects provided by MoScript. As mentioned before, operations without side effects are classified in four categories, queries, transformations of models in a same technical space, transformations of models between different technical spaces and model state checkers.

**Query operations:** The query operations provided by MoScript are `allContents`, `allContentsRoots` and `allContentsInstancesOf`. These operations dereference and load the physical model represented by the `Model` element. Then, they query the model and return a collection of OCL elements. The elements of the resulting collection are used as entry points to the model, from where the rest of the elements may be reached. Subsequent queries to the model are made with standard OCL operations. The following example illustrates how this operations may be used in general:

```
Model::allInstances()
  ->select(m | m.conformsTo.kind = 'Family')
  ->first().allContents()->collect(c | c.name))
```

In the example, we first select all the models of the models repository that conform to the Family metamodel. Then we take the first model and invoke on it the `allContents` operation. The operation dereferences de model and returns an OCL collection with all the elements contained in the model. Next, we iterate on the results, collecting all the element names. Supposing the first model found is a model of the Simpson family, the resulting collection could look like {`'Bart', 'Homer', 'Lisa', 'Maggie', 'Marge'`}.

Note that the `allContents` operation hides complexity from the user. There is no need to specify the metamodels of the models as this information is retrieved from the Megamodel.

The `allContents` facilitates the creation of powerful queries. For instance, it is possible to search models by:

– *Relations between models*, such as models which participate in model weavings [7] or models that were derived from other models (trace models).
– *Internal characteristics*, such as models including an element with a given value, metamodels containing elements of certain types, etc.
– *Computed characteristics*, such as models containing a certain number of model elements or the transformations containing more element mappings than the others, etc.

When working with big models the operation `allContents` may be expensive in terms of memory consumption and processing. So, MoScript includes other operations like `allContentsRoots` and `allContentsInstancesOf` for extracting the models elements with more precision and therefore better performance.

**Model to Model Transformations:** The M2M transformations operations provided by MoScript are the `applyTo` and the `run` operations.

The `applyTo` operations work in the context of the `Transformation` Megamodel element. They input models may be provided as a Map or as a Sequence and the output models are returned as part of a TransformationRecord. When provided as a Map, models are differentiated by their key and when provided as a Sequence, models are differentiated by their order in the Sequence.

The `applyTo` operations are especially useful if we consider transformations that are somehow generic (e.g., a transformation which transforms a Java source code model to a .Net source code model), i.e. there may exist lots of different models that may be transformed with the same transformation. In this case it is very convenient to have a way for varying the input models for each transformation execution. The following example illustrates how these operations may be used:

```
let j2dNet : Transformation =
    Transformation::allInstances()->select(t | t.identifier = 'j2dNet')
in

TerminalModel::allInstances()
    ->select(m | m.conformsTo.kind = 'Java'))
    ->collect(jModel | j2dNet.applyTo(jModel))
```

In the example we first retrieve the transformation “Java to .Net” from the repository and store it as `j2dNet`. Then we apply `j2dNet` to all the Java models found in the repository. Note that behind the scenes, the metadata engine makes several checks before running the transformation. First, it checks if the model is a transformation model, and thus may be executed. Then, it determines which is the right transformation engine for running the transformation. Finally, it checks if the input models conform to the metamodels the transformation supports. To do this, the metadata engine queries the Megamodel.

The `run` operation works in the context of the `TransformationRecord` Megamodel element. The `run` operation executes a transformation based on the information stored in the `TransformationRecord`. Since it stores the last transformation execution parameters, it is useful to rerun transformations without specifying the input models. The operation returns the newly produced models within another `TransformationRecord`.

The following example shows how it is possible to rerun all the transformations of a model repository:

```
TransformationRecord::allInstances()->collect(tr | tr.run())
```

**Projectors:** As we are working with heterogeneous model repositories, we rely on *technical projectors* for non-modelling artefacts (e.g. grammar-based text). There are two kinds of projectors: injectors and extractors. Injectors translate from other technical spaces (e.g. grammarware[19], xmlware, etc) to the modelware technical space and extractors do exactly the opposite. MoScript provides the `inject` peration for injecting models and the `extract` operation for extracting models.

The `inject` operation represents the T2M transformations. It works in the context of the `Model` Megamodel element. The Model element represents a non XMI artefact that depends on a specific grammar. The inject operation applies the transformation to the model and produces an XMI model. The following example shows how is possible to inject the source code of Java programs into Java XMI models:

```
Model::allInstances()
    ->select(m | m.conformsTo.kind = 'JavaGrammar'))
    ->collect(jCode | jCode.inject())
```

In the example, we select all the Java models which conform to the Java grammar and inject them into models conforming to Java metamodels. The result is a collection of Java XMI models. Behind the scenes, the Metadata Engine retrieves from the Megamodel the corresponding parser of the grammar and the tool that uses it, to produce the XMI model.

The `extract` operation represents the M2T transformations and uses the same mechanism as the `inject` operation, but in the opposite direction.

For both operations we follow an approach similar to the one described in [27].

**Models State Checkers:** A set of consistency check utility operations have been included in the language. The `available` operation, which verifies if the modelling artefact is available in the repository (e.g., it could have been removed by an external tool, or its physical location is unreachable), and the `isDirty` operation, which checks if the model has been modified outside MoScript. This is useful to know that is necessary to re-execute the transformations in which the model participate.

## 3.3 Operations with side effects

This subsection describes in detail the operations with side effects provided by MoScript. As said before, this operations allow the modification of the models repository and the Megamodel. This operations are usually combined with the operations without side-effects and OCL queries. The typical usage scenario is to take the models resulting from transformations executions and persist them.

**save.** The `save` operation persists an in-memory model into the repository and registers it in the Megamodel if it is not already registered. The latter step is important for keeping integrity between the Megamodel and the repository. The `save` operation takes as arguments the Megamodel, an identifier and a locator. The Megamodel argument is the Megamodel where the model should be stored. The identifier is self explanatory and the locator is the physical location path where the model should be stored (e.g. a filesystem path or a URI).

Suppose we want to store the .Net models derived from Java models in a previous example. The following example shows how the `save` operation can be used for this purpose:

```
  for(dNetModel in dNetModels) {
    dNetModel.save(
        megamodel,
        dNetModel.getIdentifier(),
        dNetModel.location + '.xmi'
    );
  }
  ...

helper context Model def: getIdentifier(): ...;
```

In the example, we iterate over the collection of .Net models and persist them in the repository. We use a helper operation to produce the identifiers of the models we are going to store.

**register.** The `register` operation allows the registration of models in the Megamodel when the model is already stored in the repository. It takes as arguments the model identifier and the physical location at which it currently exists and returns the newly registered Model instance.

The `register` operation is the operation other tools (e.g. editors, discoverers, DSLs, etc.) use to register the artefacts that are created outside the MoScript context. For instance, manually created models, discovered models, etc. Indeed, in this case the Megamodel becomes incoherent with the repository content and therefore an explicit registration is needed. The following example shows how it is possible with MoScript to register a new metametamodel:

```
megamodel.register(
  'Ecore',
  'http://www.eclipse.org/emf/2002/Ecore')
```

The metadata of a model that has been already registered in the Megamodel can be updated by re-invoking the `register` operation. For instance if another tool changes the location of a model it can be updated by re-invoking the `register` operation with the new location.

**remove.** The `remove` operation allows the removal of elements from the Megamodel. However, the physical files are not removed from the repository, thus they could be in use by other tools.

## 4 Example: Composition of Model Transformations

One of the most useful features of MoScript is the possibility of using it to express complex workflows, such as chains of model transformations and other typical model manipulation procedures. The following example illustrates this aspect.

```
program TransformationChain

using {
  inputModel: Model = Model.allInstances()
    ->select(m|m.name = 'start')->first();
}

do {
  t1: Transformation <- Transformation.allInstances()
    ->select(t| t.name = 'first' and
             t.check(Sequence{inputModel.conformsTo}))
    ->first();

  m1: TransformationRecord <- t1.applyTo(Sequence{inputModel});

  t2: Transformation <- Transformation.allInstances()
    ->select(t| t.name = 'second' and
                t.check(t1.targetReferenceModelBinding))
    ->first();

  m2: TransformationRecord <- t2.applyTo(m1.targetModelBinding);

  t3: Transformation <- Transformation.allInstances()
    ->select(t| t.name = 'third' and
                t.check(t2.targetReferenceModelBinding))
    ->first();

  m3: TransformationRecord <- t3.applyTo(m2.targetModelBinding);

  for (key in m3.targetModelBinding.keySet()) {
    m3.targetModelBinding.get(key).
    save(this, key, thisModule.getDefaultLocator(key));
  }
}

helper context Transformation def: check(metamodels :
  Sequence(ReferenceModel)): Boolean =

  let rm: Sequence(ReferenceModel) = self.sourceReferenceModelBinding in
  metamodels->includesAll(rm);

helper context String def: getDefaultLocator(key :String): String =
  'platform:/resource/project/' + key + '.xmi';
```

The example MoScript program applies three model transformations in sequence: `t1`, `t2`, and `t3`. Each model transformation is retrieved from the Megamodel via its name, and its required input metamodels. In this way, there is an implicit verification at run-time that the model transformation found actually matches the provided input models: `t1` is applicable to `inputModels`, `t2` is applicable to the output models generated by `t1`, and `t3` is applicable to the output models generated by `t2`. This is encoded in the `check` helper operation.

After all model transformations have been executed, the output models of the last transformation are saved. The locator string is derived from the model identifier, in this example.

## 5 Implementation

In this section, we describe our implementation of MoScript. Figure 4 shows how we made the instantiation of the architecture presented in section 2

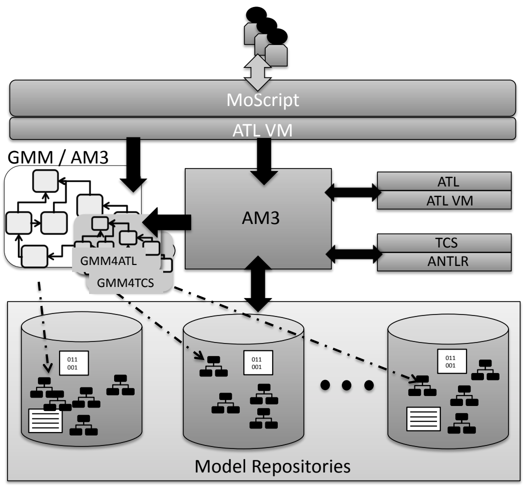


**Fig. 4.** MoScript architecture implementation.

As concrete implementation, we use our previous implementation of the Megamodel included in the AM3 tool[3]. AM3 follows the Megamodel definition as shown in Fig. 2, plus two extensions that support M2M and M2T-T2M transformation in ATL and TCS respectively. The Megamodel extension for ATL is called GMM4ATL and the extension for TCS is called GMM4TCS. As Metadata Engine, we use the AM3 tool metadata layer. As transformation engines we use ATL and TCS. TCS performs T2M transformations by generating an ANTLR[8] grammar and performs M2T using Java-based extractors or ATL OCL queries.

MoScript has been implemented on top of the Eclipse Modeling Platform. We use TCS as well, for defining its abstract and concrete syntax. TCS is in charge of parsing and lexing MoScript to populate an abstract syntax tree (AST) model for its compilation. We built the MoScript compiler with ACG[9], which is the ATL VM Code Generator. It translates the AST model (generated by TCS) into ATL VM assembly code for its execution.

Note that ATL and the ATL VM are two different concepts. ATL is a DSL for transformations which is compiled in ATL VM code. Other DSLs may run on top of the ATL VM as is the case of MoScript.

The concrete architecture uses two instances of the ATL virtual machine. One instance for MoScript and another one for ATL. This guarantees that MoScript operates independently of ATL and other transformation tools.

As model repository we used the ATL Transformations Zoo[10]. It is a repository of ATL transformation projects developed by the Eclipse community. It holds so far 205 metamodels, 275 models, 219 transformations, and more than 400 other artefacts including textual syntaxes, binary code, source code, libraries, etc. We registered these artefacts in a Megamodel by means of an automated model discovery. We have also tested MoScript with a WebML [10] repository, where models are stored in XML.

In fig. 5 we show a screen shot of a running MoScript program in Eclipse. The current implementation of MoScript can be downloaded from `http://www.emn.fr/z-info/atlanmod/index.php/Moscript_downloads`.

## 6 Related work

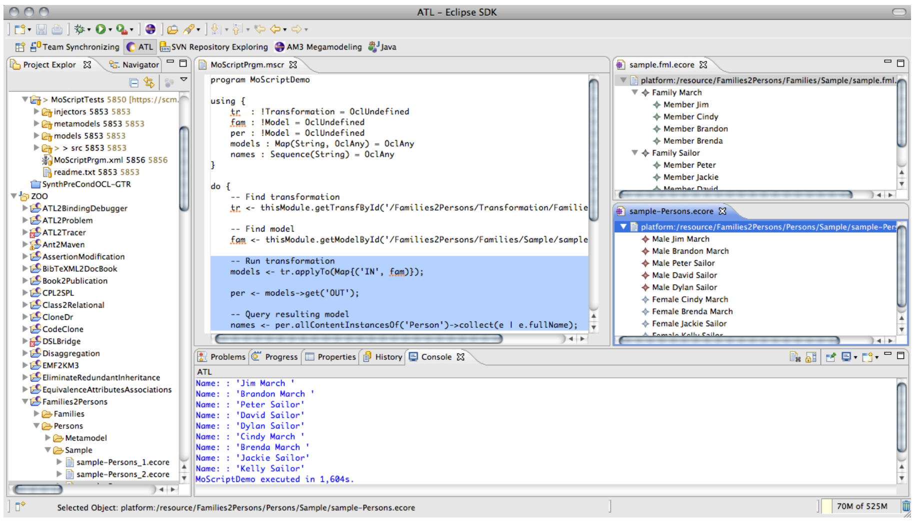
MoScript implements Global Model Management (GMM) by verifying MDE development activities against the explicit metadata inside a Megamodel. Other GMM approaches include Rondo [22], Maudeling[11], Model Bus [8] and Moose [20]. Rondo, Maudeling and Moose translate models to their own internal formats, whereas Model Bus and MoScript work directly on the models via a metadata engine. Rondo represents models as directed labeled graphs. Maudeling represents models in the Maude language [11], which is based on rewriting logic. Moose import models in CDIF or XMI exachange formats conforming to the FAMIX metamodel using third party parsers. Rondo translates between different model representations of the same information, and operates on a lower level than MoScript: it directly manipulates the model artefacts, whereas MoScript relies on

[8] `http://www.antlr.org/`

[9] `http://wiki.eclipse.org/ACG`

[10] `http://www.eclipse.org/m2m/atl/atlTransformations/`

[11] Maudeling: http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Maudeling



```
program MoScriptDemo

using {
    tr  : !Transformation = OclUndefined
    fam : !Model = OclUndefined
    per : !Model = OclUndefined
    models : Map(String, OclAny) = OclAny
    names : Sequence(String) = OclAny
}

do {
    -- Find transformation
    tr <- thisModule.getTransfById('/Families2Persons/Transformation/Familie

    -- Find model
    fam <- thisModule.getModelById('/Families2Persons/Families/Sample/sample

    -- Run transformation
    models <- tr.applyTo(Map{('IN', fam)});

    per <- models->get('OUT');

    -- Query resulting model
    names <- per.allContentInstancesOf('Person')->collect(e | e.fullName);
```

```
ATL
Name: : 'Jim March '
Name: : 'Brandon March '
Name: : 'Peter Sailor'
Name: : 'David Sailor'
Name: : 'Dylan Sailor'
Name: : 'Cindy March '
Name: : 'Brenda March '
Name: : 'Jackie Sailor'
Name: : 'Kelly Sailor'
MoScriptDemo executed in 1,604s.
```

**Fig. 5.** Running MoScript program

the invocations of transformation engines. Maudeling provides advanced querying services on modelling artefacts, and as such, could be an invokable service for MoScript. Moose offers services for navigating and manipulating multiple model versions and uses Pharo [12] (Smaltalk) as scripting language. Model Bus provides a modelling artefact broker service, where registered tools can be applied to registered models. Model Bus does not provide a Megamodel concept to look up model and tool metadata. In a way, the AHEAD tool [6] can also be seen as a GMM solution *avant-la-lettre*. AHEAD allows for the composition of heterogeneous artefacts – called *features* – into a software product. Each feature is represented as an algebraic function, where domain and range define when and where the feature is applicable. MoScript uses a reflective approach, and queries the Megamodel to check if specific modelling artefacts may be used in combination.

Model search engines such as presented in [9] and [21], are also related to GMM in that they can perform large-scale model queries, based on model contents. They differ from our approach in that they do not work on top of a Megamodel, thus the results obtained from a model search cannot be directly used in further querying. The results are usually shown as a list of model names or model fragments which at most can be downloaded.

MoScript is intended to implement MDE workflows, based on the rich contents of a Megamodel. Other MDE workflow approaches are UniTI [25], TraCo [15], the Modeling Workflow Engine (MWE)[13], and MDA Control Center [18]. UniTI composes transformation processes via typed input and output parameters. Com-

[12] http://www.pharo-project.org/home
[13] http://www.eclipse.org/modeling/emft/?project=mwe

positions are validated based on model type information and any additional constraints that can be specified on the models. TraCo uses a component metamodel, with components and ports, where each workflow component is wired to other components via its input and output ports. Ports are typed in order to validate the compositions. MWE is a model-driven version of Ant[14], with several builtin tasks for model querying and transformation. MWE does not perform any validation of the workflow composition. MoScript does not perform a static type check on its workflow compositions either, but checks the validity of the composition at run-time.

## 7 Conclusions and Future Work

In this paper, we presented MoScript: a solution for Global Model Management (GMM), based on the notion of a Megamodel. MoScript consists of a language and supporting architecture. The MoScript architecture provides uniform access to modelling artefacts, such as models, metamodels, and transformations, regardless of their storage format or their physical location. It also provides bindings to several model manipulation tools, such as transformation engines and querying tools, and allows invocation of those tools.

The MoScript language is an OCL-based scripting language for model-based task and workflow automation, based on the metadata contained in a Megamodel. It allows for querying a Megamodel for the required modelling artefacts and tools. The results of such queries can be used in the MoScript language to load and store modelling artefacts, and perform model manipulations, such as the invocation of a model transformation engine. MoScript can use the rich metadata in the Megamodel to validate model manipulations, e.g. to check if a model transformation is applied to a model that conforms to the right metamodel. MoScript is able to perform this validation at run-time, when the model manipulation is invoked.

MoScript has been implemented on top of the Eclipse Modeling Platform, using the AM3, ATL, ACG, and TCS tools. MoScript provides a textual, syntax-highlighting editor, and uses the ATL virtual machine and debugger as its run-time environment. MoScript implementation has been tested against models from the ATL examples repository.

As further work we plan to extend the list of repositories and tools our language can interact with, and increase the number of predefined operations in the language. This may include a querying tool, such as Maudeling, that allows us to validate modelling workflows written in MoScript.

## References

1. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (Apr 2008), http://www.omg.org/spec/QVT/1.0/PDF/, version 1.0, formal/08-04-03

[14] http://ant.apache.org

2. OCL 2.2 Specification (Feb 2010), http://www.omg.org/spec/OCL/2.2/PDF, version 2.2, formal/2010-02-01
3. Allilaire, F., Bézivin, J., Brunelière, H., Jouault, F.: Global model management in Eclipse GMT/AM3. In: Proc. of the Eclipse Technology eXchange (eTX) workshop at ECOOP 2006 (2006)
4. Assmann, U., Zschaler, S., Wagner, G.: Ontologies, Meta-models, and the Model-Driven Paradigm. Ontologies for Software Engineering and Software Technology pp. 249–273 (2006)
5. Barbero, M., Jouault, F., Bézivin, J.: Model driven management of complex systems: Implementing the macroscope's vision. In: Proc. of ECBS 2008, IEEE Computer Society (2008)
6. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (Jun 2004)
7. Bézivin, J.: On the unification power of models. Software and Systems Modelling 4(2), 171–188 (2005)
8. Blanc, X., Gervais, M.P., Sriplakich, P.: Model bus: Towards the interoperability of modelling tools. In: Aßmann, U., Akşit, M., Rensink, A. (eds.) Proc. of MDAFA 2003 and MDAFA 2004. Revised Selected Papers. LNCS, vol. 3599, pp. 17–32. Springer-Verlag (August 2005)
9. Bozzon, A., Brambilla, M., Fraternali, P.: Searching repositories of web application models. In: Proc. of the 10th Int. Conf. on Web Engineering, ICWE 2010. LNCS, vol. 6189, pp. 1–15. Springer-Verlag (2010)
10. Ceri, S., Brambilla, M., Fraternali, P.: The history of webml lessons learned from 10 years of model-driven development of web applications. In: Borgida, A., Chaudhri, V.K., Giorgini, P., Yu, E.S.K. (eds.) Conceptual Modeling: Foundations and Applications. Lecture Notes in Computer Science, vol. 5600, pp. 273–292. Springer (2009)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Proc. Rewriting Techniques and Applications, 2003. LNCS, vol. 2706, pp. 76–87. Springer-Verlag (June 2003)
12. Favre, J.M.: Towards a basic theory to model model driven engineering. 3er UML Workshop in Software Model Engineering (WISME 2004) joint event with UML 2004 (10 2004)
13. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, 1st edn. (October 2010)
14. Fritzsche, M., Bruneliere, H., Vanhooff, B., Berbers, Y., Jouault, F., Gilani, W.: Applying megamodelling to model driven performance engineering. In: Proceedings of the 2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. pp. 244–253. ECBS '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/ECBS.2009.33
15. Heidenreich, F., Kopcsek, J., Assmann, U.: Safe composition of transformations. In: Proc. of ICMT 2010. LNCS, vol. 6142, pp. 108–122. Springer-Verlag (2010)
16. 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
17. Jouault, F., Bézivin, J., Kurtev, I.: Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. In: GPCE '06: Proc of the 5th Int. Conf. on Generative programming and component engineering. pp. 249–254. ACM, New York, NY, USA (2006)

18. Kleppe, A.: Mcc: A model transformation environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer-Verlag (2006)
19. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. 14, 331–380 (July 2005)
20. Laval, J., Denier, S., Ducasse, S., Falleri, J.R.: Supporting Simultaneous Versions for Software Evolution Assessment. Journal of Science of Computer Programming (12 2010), http://hal.inria.fr/inria-00531500/PDF/Lava10a-SCP-Orion.pdf
21. Lucrédio, 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)
22. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: a programming platform for generic model management. In: Proc. of SIGMOD'03. pp. 193–204. ACM (2003)
23. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)
24. Rich Hilliard, Ivano Malavolta, H.M.P.P.: Realizing architecture frameworks through megamodelling techniques. In 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010) (2010)
25. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI: A unified transformation infrastructure. In: Engels, G., Opdyke, B., Schmidt, D., Weil, F. (eds.) Proc. of MoDELS 2007. LNCS, vol. 4735, pp. 31–45. Springer-Verlag (2007)
26. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in model management. In: ICMT '09: Proc. of the 2nd Int.Conf. on Theory and Practice of Model Transformations. pp. 197–212. Springer-Verlag, Berlin, Heidelberg (2009)
27. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Satellite Events at the MoDELS 2005 Conference, pp. 159–168. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2006)