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, LINA
{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. Growing adoption of Model-Driven Engineering has hugely increased the number of modelling artefacts (models, metamodels, transformations, ...) to be managed. Therefore, development teams require appropriate tools to search and manipulate models stored in model repositories, e.g. to find and reuse models or model fragments from previous projects. Unfortunately, current approaches for model management are either ad-hoc (i.e., tied to specific types of repositories and/or models), do not support complex queries (e.g., based on the model structure and its relationship with other modelling artefacts) or do not allow the manipulation of the resulting models (e.g., inspect, transform). This hinders the probability of efficiently reusing existing models or fragments thereof. In this paper we introduce MoScript, a textual domain-specific language for model management. With MoScript, users can write scripts containing queries (based on model content, structure, relationships, and behaviour derived through on-the-fly simulation) to retrieve models from model repositories, manipulate them (e.g., by running transformations on sets of models), and store them back in the repository. MoScript relies on the megamodeling concept to provide a homogeneous model-based interface to heterogeneous repositories.

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

1 Introduction

As Model-Driven Engineering (MDE) methods and tools are maturing and becoming more popular, the number of modelling artefacts consumed and produced

⋆ The author's work is partially supported by the Galaxy (ANR - French National) project.

⋆⋆ 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)

by software engineering processes (e.g., models, metamodels, and transformations) has increased considerably.

MDE for complex systems [4] 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.

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* [10].

However, current GMM solutions only provide passive metadata. It is possible to query a megamodel, but not to access and manipulate the modelling artefacts represented in a megamodel (e.g. loading/saving models, executing transformations, etc.).

In this paper, we propose MoScript a textual DSL (domain-specific language) and megamodel agnostic platform 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 scripts allow the description and automation of complex modelling tasks, 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 explains the motivations of this work. Section 3 describes the supporting architecture for MoScript. Section 4 presents the MoScript language. Section 5 puts everything together in the form of two examples. Section 6 describes how is MoScript implemented. Section 7 compares our work with other, related approaches. Finally, section 8 presents our conclusions and future work.

2 Motivation

Along this section we will present some of the problems that motivated the definition of MoScript. Then, in further sections, will illustrate how MoScript helps us to solve them.

Let's consider repositories of modelling artefacts represented by a megamodel, which are used to develop complex systems. Typically, these repositories would present the following characteristics:

- **Hundreds or thousands of heterogeneous artefacts**. The repositories contain models, metamodels, metametamodels, transformations, source code, file descriptors, data files, etc.
- **Artefacts are related to other artefacts** through predefined (e.g. conformsTo) and ad-hoc (e.g. weaving models) relationships.
- **Many different tools**, each one playing a specific role in the repository, like model to model (M2M) transformation engines, model to text transformation engines (M2T), documentation tools, compilers, script engines etc.
- **Several kind of users participate in the evolution of the repository** (e.g., stakeholders, analysts, developers, etc.).

On this repositories, there are several tasks users may want to accomplish. (1) Finding models, (2) combining modelling artefacts information (3) batch processing, (4) and registering newly generated artefacts in the system.

Finding models may be difficult depending on the search criteria and the size of the repositories. In the simplest case we may be interested in finding a single model whose name (e.g., file name) is known, so a simple search (e.g., with the file system search engine) will suffice. In many other cases models must be searched using more complex criteria, such as:

- **An internal characteristic** such as models with an element with a given value, metamodels containing elements of certain types, etc.
- **A computed characteristic** based on the models size or structure, such as models with more than two hundred elements or transformations that contain more imperative transformation excerpts than others etc.
- **Their relations with other artefacts**, such as models that are related to a given transformation or to other models e.g. by a trace model.

After finding the desired models, we may need to extract and combine their elements. The combination of information from several models is essential for extracting metrics from model repositories. For instance, we may want to compute the number of metamodel elements a transformation uses, or the number of models and elements involved in a model weaving etc.

Another common requirement when working on MDE repositories is to be able to execute batch processes of regular modelling tasks (e.g. transformations, model checks, projections etc.) involving large amounts of models. Furthermore, these batch processes should orchestrate the modelling tasks according to how the models are arranged in the repository. For instance, when a model is modified only the transformations which use the modified model should be executed and then all the transformations that use the output models of the executed transformations and so on.

Finally, since several users manipulate the repositories, they are in constant evolutionary state. Thus, it is important to have an updated view of the repository and to count with mechanisms for easily detect changes in the repository.

For instance suppose there is a batch process that re-executes the transformations in the repository when their input models change. Now, if a user contributes a new transformation to the repository, the process will not be able to re-execute the new transformation if it does not rely on an updated view that reflects the new transformation and also its relation with its input models. Therefore, we require mechanisms for easily register, update and delete artefacts from such a repository view.

In the next sections we will see how MoScript enable us to perform all these tasks.

3 The MoScript Architecture

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

3.1 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.

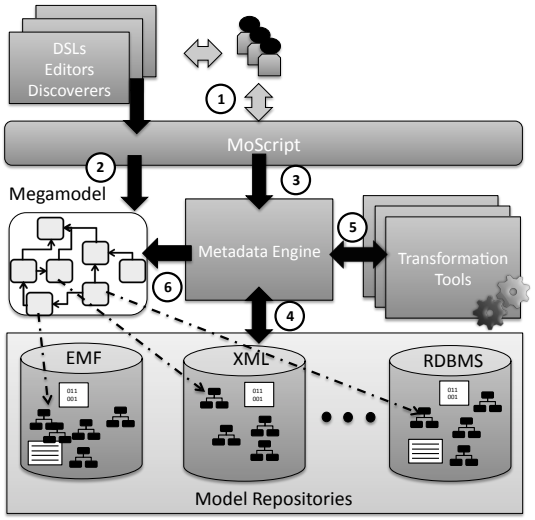

Fig. 1. The MoScript architecture.

– **MoScript:** A textual DSL, which serves as an interface between the users and the modelling artefacts repositories. Users write and run their MoScript scripts for retrieving modelling artefacts and performing modelling tasks (e.g. inspect, transform, match, etc.) with them. MoScript uses the megamodel as cartography to navigate the repositories and select modelling artefacts to

manipulate. As result of the manipulations, new modelling artefacts may be created in the repositories or existing modelling artefacts may be removed.

– **Megamodel[3]:** A model which describes artefacts within repositories (e.g. their location, kind, format, etc.), and how they are interrelated. A megamodel is a regular model, thus it conforms to a metamodel, which is shown in Fig. 2. 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).

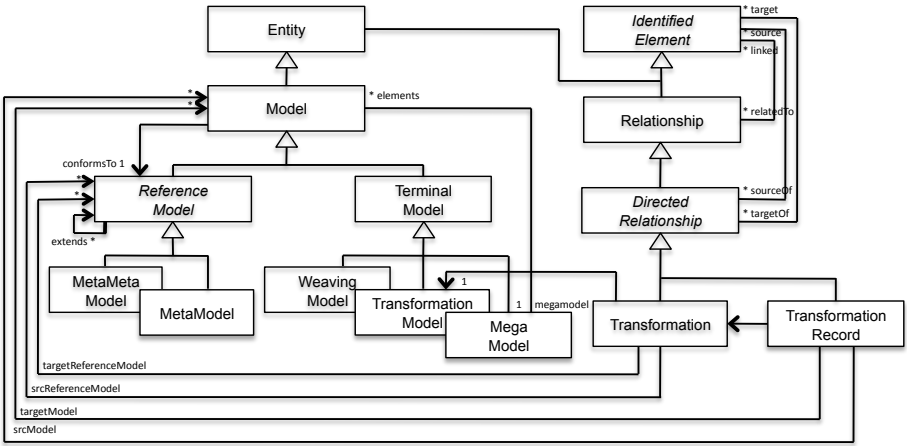

Fig. 2. Part of the core metamodel for megamodels.

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 represent models conforming to metamodels but no other model conforms to them. Examples of *TerminalModel*s are ***TransformationModel*s**, ***WeavingModel*s** and ***Megamodel*s** themselves. 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 representation of the source code of the transformation while the *ReferenceModel*s 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.
Since MDE artefacts may be bridged to models (e.g. XMI files), from this point forward we are going to call them just *models*.

– **Metadata Engine:** Provides services to MoScript for retrieving models, executing tools services 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. In a previous work [24], 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 [14], Kermeta[4], EMF Compare[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.

3.2 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 script. (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 the 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 to MoScript the model elements of the megamodel resulting from the program execution, for further processing.

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

4 The MoScript Language

MoScript is a megamodel-based scripting DSL for modeling tasks and workflow automation that uses OCL [2] as query language.

A megamodel is a regular model and thus can be navigated with standard OCL, however the result of executing an OCL query on it, is merely informative. For instance, consider the following query:

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

The query selects from a megamodel, all the models that conform to a specific kind of metamodel. The result is a collection of elements of type *Model* that cannot be used directly in OCL to access or manipulate (check, match, transform etc.) the physical artefacts they represent. This issue is due to the fact that OCL does not handle models as a bootstrapped concept and does not have multi-model support either.

The MoScript language intends to fill this gap with three main contributions: **(1) Model dereferencing**, to retrieve models represented by metadata in a megamodel; **(2) Extensive library of generic operations** to perform common model manipulation tasks with dereferenced models; **(3) Modelling tasks operation composition** combined with OCL for manipulating dereferenced models with 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 model is loaded in memory and exposed for being used through an OCL *ModelElement* type. Since OCL works on top of the megamodel, the 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 as any other OCL expression, to perform more complex operations.

Next, we will explain in detail MoScript abstract and concrete syntax, as well as its native library of operations and statements.

4.1 MoScript Abstract and Concrete Syntax

The MoScript DSL has a semantic model [11] and an abstract and concrete syntax [22].

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. Since MoScript uses OCL as query language, the complete OCL abstract syntax (not showed) is included as part of the language.

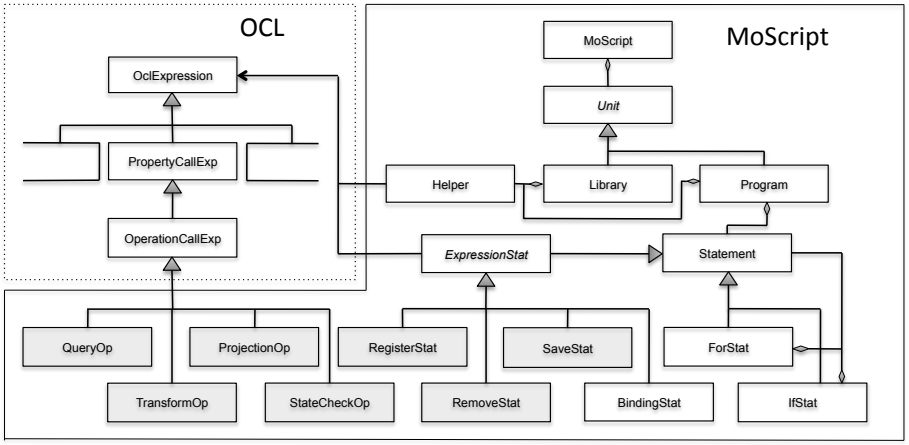

Fig. 3. MoScript abstract syntax main concepts

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 (`ProjectionOp`) and operations for checking the models state (`StateCheckOp`). For each category MoScript provide several concrete operations, which will be explained in the next subsection.

The MoScript package also provides a set of *statements with side effects* (`SaveStat, RemoveStat and RegisterStat`). **These statements allow the modification of the repository or the megamodel.** Side effects statements may embed OCL expression and therefore side effects free operations. This is why `ExpressionStat` is related to `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 statements within OCL expressions) is not permitted. OCL expressions do not know side effects statements, thus respecting the OCL side effects free philosophy.

MoScript also provides a statement for variable declaration and value binding (`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 library
...
[using  {
  variable : type = OclExpr; ...
```

```
}]

do {
  variable <- OclExpr;

  save(...); ...
  remove(OclExpr); ...
  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 without side effects and side effects statements are used in combination with control flow statements and OCL queries to perform modelling artefacts manipulations.

The complete definition of the concrete syntax is expressed in the TCS language [15], 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 without side effects and the statements with 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
Statements with side effects
save(m : Model, mm : Megamodel, id : String, locator : String)
remove(m : Model, mm : Megamodel)
register(mm : Megamodel, id : String, locator : String)

Table 1. MoScript operations and statements summary.

4.2 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 expressions. The following example illustrates how this operations may be used in general:

```
Model::allInstances()->any(m | m.indentifier = 'SimpsonFamily')
  ->allContents()->collect(c | c.name))
```

In the example, we select a model with the "SimpsonFamily" id from the repository, and invoke the `allContents` operation on it. 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. The resulting collection should look like {`'Bart', 'Homer', 'Lisa', 'Maggie', 'Marge'`}.

Note that the `allContents` operation hides complexity from the user. There is no need to specify the metamodel of the model as this information is retrieved from the megamodel.

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()->any(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 checks if the input models conform to the metamodels the transformation supports. Finally, it determines which is the right transformation engine[8] for running the transformation. 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-XMI 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[17], xmlware, etc) to the modelware technical space and extractors do exactly the opposite. MoScript provides the `inject` operation 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* element, which 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[9] 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 [25].

[8] required relations not showed in Fig.2
[9] required relations not showed in Fig.2

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 if it is necessary to re-execute the transformations in which the model participates.

4.3 Statements with side effects

This subsection describes in detail the statements with side effects provided by MoScript. As said before, these statements allow the modification of the models in the repository and the megamodel. These statements are usually combined with OCL expressions and operations without side-effects.

save. The `save` statement 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` statement takes as arguments the *Model* to be persisted, the *megamodel* in which the model should be stored[10], an identifier and a locator. The locator argument is the physical location path where the model should be stored (e.g. a filesystem path or URI).

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

```
... for(dNetModel in dNetModels) {
      save(dNetModel, this, 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 to produce the identifiers of the models. The `this` keyword means that the model will be stored in the root megamodel.

register. The `register` statement allows the registration of models in the megamodel when the model is already stored in the repository. It takes as arguments the *megamodel*, the model identifier, its physical location and creates the corresponding megamodel element.

The `register` statement is the statement other tools (e.g. editors, discoverers, DSLs, etc.) use to register the artefacts created outside the MoScript context. For instance, manually created models, discovered models, etc. The following example shows how it is possible with MoScript to register a new metametamodel:

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

[10] remember a megamodel may contain other megamodels

The metadata of a model already registered in the megamodel can be updated by re-invoking the `register` statement. For instance, when another tool changes the location of a model.

remove. The `remove` statement allows the removal of models from the repository. It also removes the model element from the megamodel in order to maintain consistency between both. It receives as argument the *megamodel* and the *Model* to be eliminated.

5 Putting all together

In this section we provide examples of complete MoScript scripts which demonstrate the power of the language.

5.1 Change propagation

Roughly speaking, Model Driven Development (MDD) consists in transforming models from higher levels to lower levels of abstraction until the generation of code, in order to produce runnable systems.

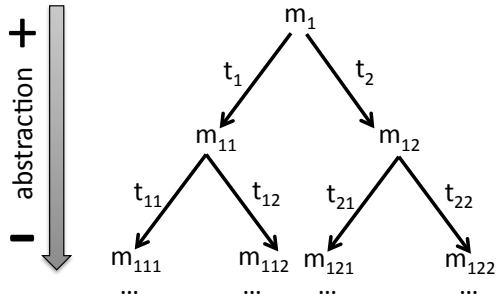

Fig. 4. An MDD system transformation chains

Now, suppose we have an MDD based system, which has a binary tree like arrangement of models and transformations, as shown in Fig. 4. In the figure, models are denoted by the m nodes and transformations by the t directed edges. Now, if model m_1 changes we will have to re-execute all the transformations that are directly or indirectly affected by the change in the model, in order to reflect the change in the models of the lowest level of abstraction (the code). The MoScript program in listing 4 shows how to do it.

Listing 1.1. Change propagation

```
1 program PropagateChanges
2
3 do {
4
5   im : Model = Model::allInstances()->any(m | m.identifier = 'm1');
6
7   for(tr : getTransformations(m)) {
```

```
8       om : Model = tr.run().targetModel->first();
9       save(om, this, om.identifier, om.locator);
10    }
11  }
12
13  helper def: getTransformations(m :Model) :Sequence(TransformationRecord)
        ↪ =
14    trs : Sequence(TransformationRecord) = TransformationRecord::
          ↪allInstances()->select(tr | tr.srcModel->first().identifier = m.
          ↪identifier) in
15    if trs->isEmpty() then
16      Sequence{}
17    else
18      trs->union(trs->collect(tr | getTranformations(tr.targetModel->first
            ↪()))->flatten())
19    endif;
```

The explanation of the code is the following:

I We select from the repository the modified model by its id (line 5).
II We call the helper `getTransformations` (line 7) to return a collection of *TransformationRecord*s in the order they must be executed.
III The `getTransformations` helper selects all the *TransformationRecord*s that use model `m` as input of its transformation (line 14).
IV For each *TransformationRecord*, the output model of its transformation is selected, and a recursive call is made to the `getTransformations` helper, in order to go through the tree in depth, getting the rest of the *TransformationRecord*s (line 18).
V Finally, for each *TransformationRecord* its transformation is executed (line 8), its resulting model saved in the repository and updated in the megamodel (line 9)

Note that the example is an intentional over simplification of real case models and transformations arrangements, in order to keep the code simple. We assumed the transformations have only one input and output model and no cycles between them.

5.2 Inspecting and Combining Models Information

In this example we show how we can combine information from several models and make computations for obtaining measurements from the model repository. We will compute a measure for determining the transformations *naive completeness*[11] of all the transformations in the repository.

A transformation t_1 is *naively complete* if all the elements of its source metamodel mm_1 and its target metamodel mm_2 are matched (used) by at least one rule of the transformation.

To illustrate our definition of naive completeness, suppose we have a transformation and its models $t_1(m_1) = m_2$ where m_1 is the input model and m_2 is the output model. m_1 and m_2 conform to the metamodels mm_1 and mm_2

[11] Checking whether a transformation is actually complete or not is much more complex.

respectively. A transformation is a finite set of rules $t_1 = (r_1, r_2, ..., r_n)$. Each rule has 1 or none input element or pattern and has at least one output element $r() = (op_1, op_2, ..., op_n)$ or $r(ip) = (op_1, op_2, ..., op_n)$. The input and output patterns correspond to elements of the metamodels.

To determine which elements of mm_1 and mm_2 are used in the transformation, we will inspect its transformation rules. For each rule of t_1 we will verify the number of elements from mm_1 (e_{mm1}) present as input pattern (ip_{t1}) in at least one rule of the transformation. Number of elements from mm_2 (e_{mm2}) present as output patterns (op_{t1}) in at least one rule of the transformation.

The result of the measurement is calculated for mm_1 as $Cr_{mm1} = \sum(ip_{t11}, ip_{t12}, \ldots, ip_{t1n}) / \sum(e_{mm11}, e_{mm12}, \ldots, e_{mm1n})$ and the same for mm_2 but with the output patterns and the output metamodel. The transformation is considered naively complete if $Cr_{mm1} = 1$ and $Cr_{mm2} = 1$.

Listing 1.2 shows the OCL query for obtaining the described measures. For the sake of simplicity we inspect only ATL matched rules, which are fully declarative and always have an input element. We also assume that all the metamodels conform to Ecore.

Listing 1.2. Transformation completeness query

```
program TransformationCompleteness

do {
  res : Sequence(OclAny) = getNaiveCompleteness();
  -- Do something with the result
}

helper def getNaiveCompleteness(): Sequence(OclAny) =
  Transformation.allInstances()->collect(tr |
    let trName : String = tr.transformationModel.name in
    let mmIn : Set(String) = tr.srcReferenceModel
      ->collect(e | e.referenceModel.allContentInstancesOf('EClass'))
      ->flatten()->collect(e | e.name).asSet() in
    let mmOut : Set(String) = tr.targetReferenceModel
      ->collect(e | e.referenceModel.allContentInstancesOf('EClass'))
      ->flatten()->collect(e | e.name).asSet() in
    let trIn : Set(String) = tr.transformationModel.inject().
        ↪allContentInstancesOf('MatchedRule')
      ->collect(x | x.inPattern.elements
      ->collect(y | y.type.name))->flatten().asSet() in
    let trOut : Set(String) = tr.transformationModel.inject().
        ↪allContentInstancesOf('MatchedRule')
      ->collect(x | x.outPattern.elements
      ->collect(y | y.type.name))->flatten().asSet() in
    let inRt : Real = trIn->size() / mmIn->size() in
    let outRt : Real = trOut->size() / mmOut->size() in
    Sequence(trName, inRt, outRt)
  );
```

Due to space limitations we do not explain the code in detail, but note that doing these kind of computations without MoScript will demand a lot of work with existing scripting techniques or adhoc codifications.

6 Implementation

In this section, we describe our implementation of MoScript. Figure 5 shows how we made the instantiation of the architecture presented in section 3

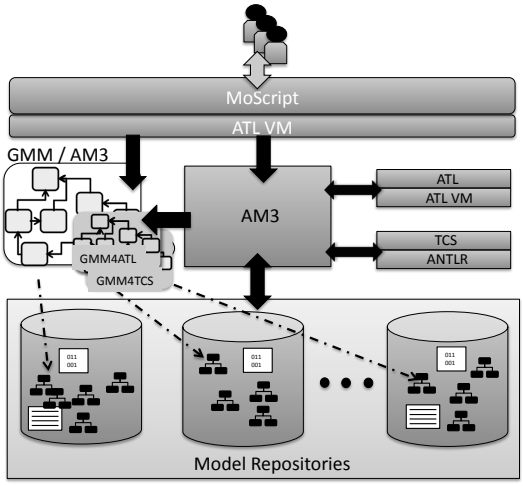

Fig. 5. MoScript architecture implementation.

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[12] 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 ready for compilation. We built the MoScript compiler with ACG[13], 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.

We tested MoScript with the ATL Transformations Zoo[14]. A model 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,

[12] http://www.antlr.org/
[13] http://wiki.eclipse.org/ACG
[14] http://www.eclipse.org/m2m/atl/atlTransformations/

etc. We also tested MoScript with a WebML [8] repository, where models are stored in XML.

In fig. 6 we show a screen shot of a running MoScript script in Eclipse. The current implementation of MoScript can be downloaded from http://www.emn.fr/z-info/atlanmod/index.php/Moscript_downloads.

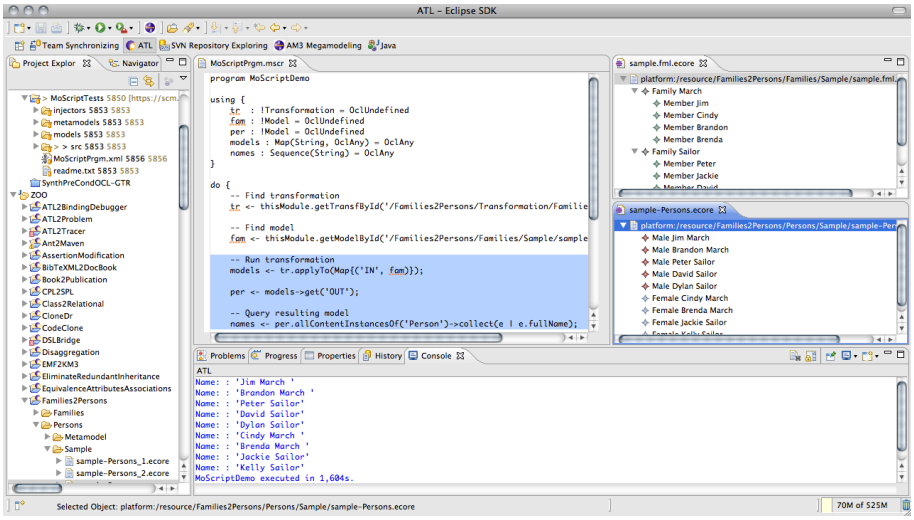

Fig. 6. Running MoScript script

7 Related work

The concept of a megamodel was proposed in [5] and in [10]. In [5] the megamodel is proposed a solution to Global Model Management (GMM) while in [10] it is presented as a metamodel for describing MDE formalized with set theory. Several recently works report use of megamodels or megamodeling techniques. For instance, in [12] a megamodel is used for the representation of all the artefacts and their relationships involved in the model-driven support for the evolution of software architectures. In [20], Megaf an infrastructure for the creation of architecture frameworks formalizes its underlying infrastructure through a megamodel for checking consistency among architectural elements. The 101company[15] project is an effort to create a conceptual framework based on megamodeling techniques for understanding analogies between heterogeneous technologies. In general, all the mentioned works focus on representing high number of different artefacts and technical spaces involved in non trivial software systems and development processes. MoScript uses the megamodel with the same general intend.

As far as we know, there are not many DSLs or approaches for GMM, i.e. they do not use a megamodel as a global view for orchestrating and verifying MDE

[15] http://101companies.uni-koblenz.de/index.php/Main_Page

development activities against it. However, we find similarities with approaches such as Rondo [21], Maudeling[16], Model Bus [6] and Moose [18]. 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 [9], which is based on rewriting logic. Moose represents 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 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 [17] (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. 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 those presented in [7] and [19], 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 the results obtained from a model search cannot be directly used in further modelling operations. 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. The main difference with other MDE workflow approaches is that MoScript relies on a query language that works on the rich contents of a megamodel and orchestrations are based on its queries results. Other MDE workflow approaches are UniTI [23], TraCo [13], the Modeling Workflow Engine (MWE)[18], and MDA Control Center [16]. UniTI composes transformation processes via typed input and output parameters. Compositions 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[19], 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.

[16] Maudeling: http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Maudeling
[17] http://www.pharo-project.org/home
[18] http://www.eclipse.org/modeling/emft/?project=mwe
[19] http://ant.apache.org

8 Conclusions and Future Work

In this paper, we presented MoScript: a scripting DSL and platform for Global Model Management (GMM), based on the notion of a megamodel.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 is an OCL-based scripting language for model-based task and workflow automation, based on the metadata contained in a megamodel. It allows querying a megamodel and use the results of such queries 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 TCS, ACG tools and AM3 metadata engine. It provides a textual and uses the ATL virtual machine and debugger as its run-time environment. MoScript uses ATL as M2M and TCS as M2T-T2M transformation engines. MoScript implementation has been tested against models from the ATL examples repository and a WebML 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 and statements of 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
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., Bezivin, J., Bruneliere, H., Jouault, F.: Global model management in eclipse gmt/am3. In: In Proc. of the Eclipse Technology eXchange workshop (eTX) at ECOOP'06 (2006)
4. 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)
5. Bezivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: In Proc. of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on OOPSLA. (08 2004)
6. 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. LNCS, vol. 3599, pp. 17–32 (August 2005)

7. 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 (2010)
8. Ceri, S., Brambilla, M., Fraternali, P.: The history of webml lessons learned from 10 years of model-driven development of web applications. In: Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 273–292 (2009)
9. 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 (June 2003)
10. 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)
11. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, 1st edn. (October 2010)
12. Graaf, B.: Model-driven evolution of software architectures. European Conference on Software Maintenance and Reengineering (CSMR'07) pp. 357–360 (2007)
13. Heidenreich, F., Kopcsek, J., Assmann, U.: Safe composition of transformations. In: Proc. of ICMT 2010. LNCS, vol. 6142, pp. 108–122. Springer-Verlag (2010)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming 72(1-2), 31 – 39 (2008)
15. 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 (2006)
16. Kleppe, A.: Mcc: A model transformation environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187 (2006)
17. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. 14, 331–380 (July 2005)
18. Laval, J., Denier, S., Ducasse, S., Falleri, J.R.: Supporting Simultaneous Versions for Software Evolution Assessment. Journal of Science of Computer Programming (12 2010)
19. 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)
20. Malavolta, I.: A model-driven approach for managing software architectures with multiple evolving concerns. In: Proc. of European Conference on Software Architecture: Companion Volume. pp. 4–8. ECSA '10, ACM (2010)
21. 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)
22. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)
23. 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 (2007)
24. 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)
25. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Satellite Events at the MoDELS 2005 Conference, pp. 159–168. LNCS (2006)