

Think Global, Act Local: Implementing Model Management with Domain-Specific Integration Languages^{4,5}

Thomas Reiter¹, Werner Retschitzegger²

*Information Systems Group (IFS)
Johannes Kepler University Linz, Austria*

Kerstin Altmanninger³

*Department of Telecooperation (TK)
Johannes Kepler University Linz, Austria*

Abstract

In recent years a number of model transformation languages have emerged that deal with fine-grained, local transformation specifications, commonly known as *programming in the small*. To be able to develop complex transformation systems in a scalable way, mechanisms to work directly on the global model level are desirable, referred to as *programming in the large*. In this paper we show how domain specific model integration languages can be defined, and how they can be composed in order to achieve complex model management tasks. Thereby, we base our approach on the definition of declarative model integration languages, of which implementing transformations are derived. We give a categorization of these transformations and rely on an object-oriented mechanism allowing to realize complex model management tasks.

Keywords: model integration, model transformation, model management, domain-specific languages.

1 Introduction

Model-driven Development (MDD) in general aims at raising the productivity and quality of software development by automatically deriving code artifacts from models. Even though an immediate model-to-code mechanism can yield tremendous benefits, it is commonly accepted that working model-to-model mechanisms are necessary [21] to achieve integration among multiple models describing a system and to make models first-class-citizens in MDD.

¹ Email: reiter@ifs.uni-linz.ac.at

² Email: werner@ifs.uni-linz.ac.at

³ Email: kerstin@tk.uni-linz.ac.at

⁴ We thank Elisabeth Kapsammer, Wieland Schwinger and Manuel Wimmer for their comments.

⁵ This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.

In recent years, therefore, a number of model transformation languages (MTLs) have emerged, which allow to specify transformations between metamodels. Such transformations are defined on a fine-grained, *local* level, upon elements of these metamodels. Albeit the advantages that MTLs bring in terms of manipulating models, it is quite clear that defining model transformations on a local level, only, can pose substantial scalability problems.

There are already first approaches trying to alleviate the above mentioned problem from two different angles (cf. also Section 5). The first category adheres to a bottom-up approach, meaning that existing general purpose MTLs are extended for special tasks like model merging [12] or model comparison [19]. Furthermore, mappings carrying special semantics can be established between metamodels and further on be derived into executable model transformations [6].

The second category of approaches is more top-down-oriented and falls into the area of model management, where relationships between models are expressed on a coarse-grained, *global* level through a set of generic model management operators. The aim of model management is to ease and speed up the development of meta-data intensive applications, by factoring out common tasks in various application scenarios and by providing generic model management operators for these tasks. The generality of these operators allows to make assumptions about, e.g., algebraic properties of model management operations, but does not necessarily make any specific assumptions about the actual implementations of these model management operators. For instance, Rondo [5] is an actual implementation of such a system, oriented towards managing relational and XML schemata.

It is our opinion that both, bottom-up and top-down approaches are valuable contributions and should be considered as potentially complementing each other, as opposed to be thought of as two sides of a coin. One of model management's main contributions is to provide a conceptually well-founded framework guiding the actual implementation of model management operators, for which the capabilities of increasingly more powerful MTLs can be leveraged.

Therefore, this paper represents early work in drafting an approach that tries to build on the strengths of both paradigms. On the one hand, the model management rationale to make models first-class-citizens and to achieve complex model management tasks by assembling global operations on models, is followed. On the other hand, our approach relies on domain-specific languages (DSLs) developed atop general-purpose MTLs for locally handling fine-grained relationships between metamodels.

The proposed approach resides in the context of the ModelCVS [16][15] tool integration project, which aims at integrating various modeling tools via metamodels representing their respective modeling language. Concretely, the problems that need to be solved are finding efficient ways to integrate various metamodels on a local level, and solve common problems, e.g., metamodel evolution, on a global level.

The remainder of this paper is structured as follows. Section 2 discusses the rationale behind our approach. Section 3 deals with the composition of model management operators and classifies different kinds of transformations. Section 4 goes into detail about how domain specific integration languages can be defined. Section 5 discusses related work and Section 6 summarizes our approach.

2 Rationale for our Approach

To better motivate the rationale underlying our approach, this section starts with an analogy referring to the definition of primitive recursive functions. Table 1 shows the various abstraction layers our approach is built on and introduces terms and concepts used throughout this paper. Referring to computability theory, using only the constant, successor, and projection functions, all primitive recursive functions, such as addition or subtraction operators, can be defined. Analogous to that, on top of existing *model transformation languages* residing on the local level, we define *integration operators* on the local composite level for handling fine-grained relationships between model elements. Algebraic as well as integration operators are then bundled up into sets representing algebras or *integration languages*, respectively. We refer to this level as intermediate, because the elements of algebras and integration languages act upon the local level, but are used to define transformations acting upon the global level. Hence, on the global level, complex functions and concrete realizations of model management operators are found. These algebras and languages are at a suitable level of abstraction and are commonly used to assemble algebraic terms or model management scripts [4]. After establishing a view across the abstraction layers, ranging from bottom-level MTLs to top-level model management scripts, we illustrate our approach in a top-down fashion in more detail.

Level	Natural Numbers	Example	Proposed Approach	Example
Global Composite	Terms	$\text{power}(\max(x,y))$	Model Mgmt. Scripts	<code>m''=translate(m.merge(m'))</code>
Global	Complex Functions	$\text{power}(z), \max(x,y)$	Model Mgmt. Operators	Translation, PackageMerge
Intermediate	Algebras	$\{+,-,\mathbb{N}\}, \{*,./,\mathbb{N}\}$	Integration Languages	FullEquivLang, MergeLang
Local Composite	Operators	$+, -, *$	Integration Operators	FullEquivClass, MergeClass
Local	Base Functions	$\text{succ}(x), \text{null}()$	MTL Expressions	ATLRule, OCLEExpression

Table 1
Analogy referring to the definition of primitive recursive functions.

Global and Global Composite. As depicted in Figure 1, we believe it is helpful to view the composition of complex model management operations as an object-oriented (OO) meta-programming task [2], where models are understood as objects and transformations as methods acting upon these “objects”. Consequently, we think that an integral part of defining a metamodel should be to specify *integration behavior* in the form of transformations (1) that are tied to that metamodel (e.g., merging state-machines). The composition of transformations can then be facilitated by writing model management scripts in an OO-style notation, which invokes transformations on models (2) just like methods on objects. Transformations representing actual realizations of model management operators are defined by languages (3) which we refer to as *domain specific integration languages* (DSIL).

Intermediate. A DSIL consists of operators that enable to locally handle fine-grained relationships between metamodels and is formalized as a weaving metamodel [8]. The domain specificity of a DSIL stems from the fact that a DSIL can only be applied to certain kinds of metamodels (4). For instance, a *MergeLang* may be used to specify a merge for metamodels representing structures (e.g., class diagrams). As behavioral integration poses a very different challenge than structural integration [23], a merge on a metamodel representing some kind of behavior

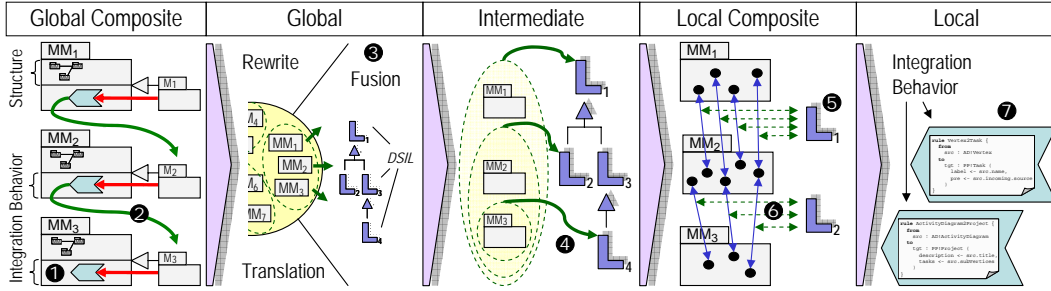


Fig. 1. Illustration of our approach’s abstraction layers.

(e.g., business process), would have to be specified in a *FlowMergeLang*, whose operators are specifically aimed towards metamodels representing flows [22]. Efforts to formalize a metamodel’s domain (e.g., by mapping metamodels onto ontologies [17]), could help to check whether a metamodel falls into the domain of a certain DSIL. From our point of view, this still poses an open research question and the applicability of a DSIL on a metamodel ultimately requires a user’s judgement.

Local and Local Composite. An *integration specification* in a DSIL is a *weaving* model that conforms to its *weaving metamodel*, which is a certain DSIL’s metamodel (5). A weaving consists of a set of typed links between elements of a model or a metamodel. The types of links represent different kinds of *integration operators* (6), whose execution semantics are defined through a mapping towards an executable MTL. Thus, an integration specification is finally derived into an executable model transformation (7).

Notably, our approach focuses on specifying integration between metamodels in a purely declarative way, as such a specification (which abstracts imperative implementations) is the basis for reasoning tasks like analysis or optimization.

3 Managing Models on a Global Level

This section discusses the two top-most layers of abstraction which have been previously introduced as *global* and *global composite*. The following subsection exemplifies transformation composition on the global composite layer through a model management script. Based on observations gained in the example, the global level is elaborated on in more detail by laying out a useful classification of transformations.

3.1 Model Management Scripts on the Global Composite Level

The following example deals with the merging of two domains represented by two metamodels, as depicted in Fig. 2. When these metamodels are merged, however, also their conforming models should be merged. We refer to such a model management task as an *exogenous merge*. A concrete application would be to merge previously modularized metamodels (e.g., a BPEL metamodel split into a structural and a behavioral part) or to extend a metamodel with a certain aspect (e.g., add “Marks” to a Petri-net metamodel) [18]. Throughout the example, however, for simplicity reasons and to emphasize the global perspective at this abstraction layer

we will not go into detail about the makeup of the metamodels, which are simply referred to as A and B and their conforming models as a and as b , respectively.

There may be multiple ways to describe an *exogenous merge*. A straightforward way would be to program the whole task as one monolithic transformation in a general purpose transformation language. As already argued before, such ad-hoc approaches suffer poor scalability and reuse potential. Instead, a description of such complex tasks as a composition of global model management operations favors scalability and reuse: Firstly, one is not concerned with handling fine-grained relationships on the local model element level, and secondly, model management operations can be easily reused in order to assemble scripts for different tasks. Thinking of model management scripts as OO programs, as we propose to do, furthermore has the advantage that the code for this model management script does not need to be changed in order to work with other metamodels, as the actual transformations that are invoked, are *dynamically* bound depending on a model’s metamodel.

Fig. 2 depicts the described setting and gives a listing of the according *exogenous merge* model management script. Details of the various steps in that script are discussed in the following.

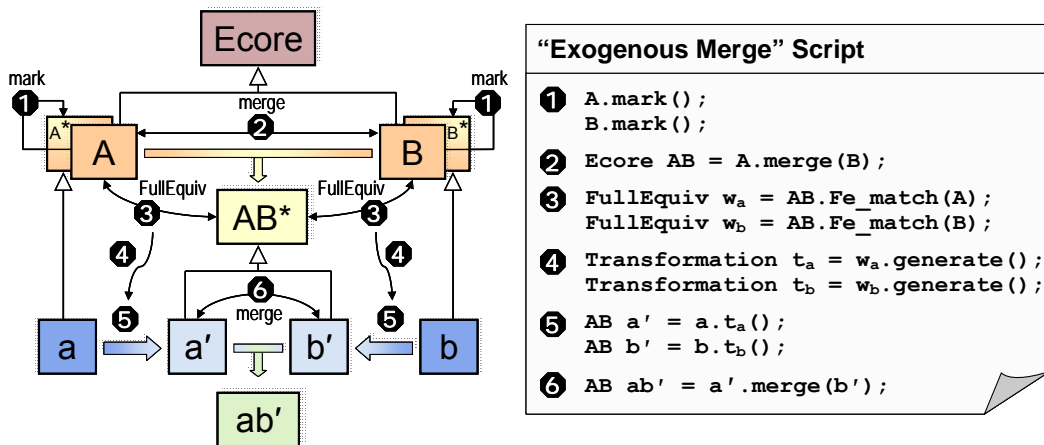


Fig. 2. Model management script for exogenous merge.

In the first step (1) a *mark* transformation is run that tags all metamodel elements with a unique id by adding annotations. In the second step (2) a *merge* transformation is executed that unites the metamodels A and B as specified in the *merge* integration specification, for instance through overlapping the two metamodels on certain join points. This results in a new metamodel AB , which also contains the initially introduced markings. In the third step (3) a transformation creates a weaving between each of the original A and B metamodels and the newly created AB metamodel. A transformation creating such a weaving does a relatively easy job, as it can rely on the previously introduced traceability annotations to match model elements. The weavings created in our example comprise a certain integration specification, which in step (4) is derived into executable transformations, which are executed in (5) and migrate the models a and b towards models a' and b' that conform to the AB metamodel. Since these models now conform to the same metamodel, they can

be overlapped in a *merge* transformation (6). We would like to mention, that also other ways of realizing traceability mechanisms exist, for instance through weaving a traceability aspect into a base transformation in an aspect-oriented fashion [14]. Embedding traceability information into a model through annotations, in our opinion has the advantage that a transformation producing a weaving can relatively easily create a trace weaving model. For further processing, the annotations could be easily pruned from the model.

3.2 Categorizing Transformations on the Global Level

After having discussed the composition of global model management operations, the following section will establish a better understanding of the transformations that were used in the previous example. However, this will not be done by discussing the behavior of these transformations in terms of how model elements are manipulated, as this is transparent on the global level and would differ for different kinds of metamodels. Rather, the global level requires to put thought on what *kinds of transformations* are being employed.

Hence, we classify our approach’s DSILs used to define actual transformations, into certain categories. These categories reflect recurring kinds of transformations prevalent in model engineering. Such a categorization favors the definition of modular and comprehensible transformations and creates a mindset where one can think of solving complex model management tasks through composition of such modular transformations, as exemplified in the previous subsection. Another advantage of this approach is that for every category a generic toolset can be built that allows to manipulate languages falling into a certain category. Transformations producing weavings can all share a tool like the Atlas Model Weaver [8], whereas translating transformations, for instance, can benefit from tooling to capture execution traces.

A similar distinction is made in the area of generic model management [4]. However, we allow the distinction between different categories according to the kind of input (IMM) and output metamodels (OMM) (cf. Table 2) that the transformations act upon, as opposed to focus on making assumptions about the behavior or algebraic properties of transformations.

Table 2 gives an overview by showing a category’s input/output characteristics, example transformations, a reference to similar operators proposed in literature, and a function signature being representative for a category’s transformations. To put each of the example transformations in a concrete context, we refer to the previously used traceability mechanism in more detail now. First, the *containsAnnotations* transformation is called to check whether a model is free of traceability annotations. If so, with *addTraceAnnotations* traceability annotations are added to all model elements. Next, *translateWithAnnotations* or *mergeWithAnnotations* is called that produces an output model in which the traceability annotations are migrated from source to target model elements. Then, *matchByAnnotations* is invoked which establishes a weaving model representing traceability links according to the annotations contained in source and target model. In a final step, this traceability weaving is input to the *createReverseTranslation* transformation which produces a round-tripping translation transformation.

Category	Arity	Output	Function Signatur	Example	Operators in Lit.
Check	1	Prim. Type	$P\ p = \text{check}(M\ m);$	containsAnnotations	Check-property [11]
Rewrite	1	OMM==IMM	$M\ m' = \text{rewrite}(M\ m);$	addTraceAnnotations	Refactorings [15]
Translation	1	OMM!=IMM	$Mb\ mb = \text{translate}(Ma\ ma);$	translateWithAnnotations	ModelGen [3]
Fusion	2	OMM==IMM	$M\ m = \text{fuse}(M\ ma, M\ mb);$	mergeWithAnnotations	Merge [11]
Relation	2	Weaving	$W\ w = \text{relate}(Ma\ ma, Mb\ mb);$	matchByAnnotations	Match [3]
Generation	1	Transform.	$T\ t = \text{generate}(W\ w);$	createReverseTranslation	GlueCodeGen [10]

Table 2
Categories of transformations on the global level.

Check. The first category deals with transformations that map models onto primitive value ranges, like booleans or natural numbers. This kind of functions allow to determine whether certain properties hold for models (consistency checks), or to evaluate certain criteria (e.g., number of inheritance relationships) of models.

Rewrite. This category encompasses transformations that modify a model but do not transform it into a model of another metamodel. This kind of transformations can be associated with editing or specialized refactoring operations [15], that do not require input from another model. An example language discussed later on is a language that allows to mark elements in a model with certain annotations.

Translation. A translating function maps concepts of one metamodel onto concepts of another metamodel and henceforth transforms a model conforming to one metamodel into a model conforming to another metamodel. A special case of a translating transformation would be if the source and target metamodels are the same, but nevertheless concepts are translated into other concepts. This would especially be the case when using UML, which, by means of stereotypes or tagged values offers a somewhat weaker mechanism than DSLs to represent concepts. Still we consider such transformations as part of this class, as the same translation language constructs can be of use, even though binding these needs some special effort.

Fusion. We classify a transformation as a fusion, if it takes two models as input and produces an output model taking into account each of the inputs. The input and output models thereby conform to the same metamodel. For instance, this class includes transformations that are usually associated with a merge or a diff [11], although domain specific realizations may potentially blend these two behaviors, by overlapping and clipping certain parts of the source models.

Relation. Transformations of this kind produce special kinds of models, which relate two other models. These models are referred to as weaving models [8] and consist of typed links between elements of left-hand side (LHS) and right-hand side (RHS) models. An example for a transformation creating a weaving could be carried out through a matcher, which heuristically establishes weaving links. Therefore, the creation of a weaving is often a task involving manual effort.

Generation. This kind of transformations generates other transformations. More precisely, they function as a compiler which turns weaving models into executable transformations. Typically this is either accomplished through a transformation whose target metamodel is the abstract syntax of a model transformation language or through a templating mechanism. It is important to note, that our view of a weaving is that a weaving model implicitly references its LHS and its RHS model, hence we omit these models in the above signature. Thus, we can still assume that the generation function has access to read the LHS and RHS models.

4 Integrating Models on the Local Level

The previous section has detailed the global composite and the global level. Hence, this subsection focuses on the remaining abstraction layers. As integration languages reside on the intermediate layer and this section makes use of a concrete example DSIL, the first subsection is dedicated to the *intermediate* level and to introducing the example language. The second subsection discusses the *local composite* level and discusses integration operators for the example DSIL. The *local* level is dealt with in the third subsection and focuses on the definition and extension of execution semantics for integration operators through a mapping towards MTL code.

4.1 An Example DSIL on the Intermediate Level

The abstract syntax of a DSIL is defined in a weaving metamodel [8], which is basically made up of meta-classes for the languages' integration operators. Furthermore, constraints are specified that enable to check whether a certain integration specification is valid. Such an analysis is comparable to static compile-time checking in traditional programming languages. In the following we will give an example for a basic language for the *translation* category. Due to space limitations we will not go into detail about languages of other categories, just as we are not claiming that the described integration operators are complete, as a precise definition is out of scope of this paper.

The setting for our example is depicted in Fig. 3, which shows a simple metamodel for activity diagrams (AD) as the LHS metamodel, and a Gantt-chart project plan (PP) metamodel as the RHS metamodel. An activity diagram consists of vertices and transitions in-between. A project consists of a number of tasks and every task has a reference to its previous task.

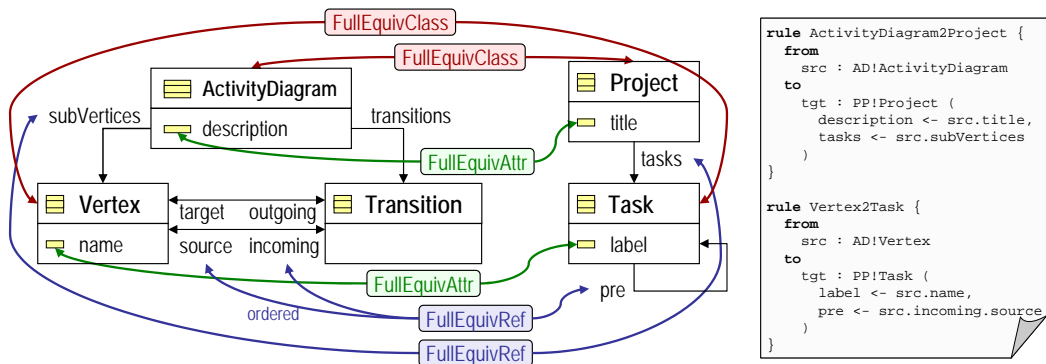


Fig. 3. Example integration specification in the *FullEquiv* language.

The intention is to transform ADs into PPs in a semantics preserving way. Instead of programming the transformation directly, a DSIL is used to specify a mapping that denotes the translation of concepts of the AD metamodel onto concepts of the PP metamodel. The code snippet on the right side of Fig. 3 shows the final transformation code that should be generated in an ATL-like⁶ notation.

⁶ For simplicity reasons code snippets use simplified ATL syntax.

4.2 Integration Operators on the Local Composite Level

The DSIL used is the so called *FullEquivalence* language, which can be seen as a basic language for the *translation* category. It consists of three operators, namely *FullEquivClass*, *FullEquivAttr*, and *FullEquivRef*, which in a pair-wise manner link classes, attributes, and references, respectively. During the definition of a DSIL, it is important to define how its operators relate to each other. In our example, for instance, the *FullEquivAttr* and the *FullEquivRef* operators have to stand in the context of the *FullEquivClass* operator, as the assignment of values and the setting of references needs to happen in the context of the model elements which these attributes and references belong to. Such a relationship is defined through containment in the metamodel of the *FullEquivalence* language by making the *FullEquivAttr* and the *FullEquivRef* operators children of the *FullEquivClass* parent. Relationships not inferable from structure (e.g., precedence rules) can be specified in a constraint language. An example for a constraint that should be enforced is that an attribute in a target model element cannot be referenced by more than one *FullEquivAttr* operator having the same *FullEquivClass* parent, as this would lead to ambiguity concerning which source attribute should be used to set the target attribute.

4.3 Mapping Integration Operators onto the Local Level

After describing the operators, in the following example it is shown how a generating function can derive an implementation in the form of MTL code. Furthermore, we will exemplify the extension of an existing operator’s semantics. The execution semantics are expressed through a function, mapping integration specifications expressed as weaving models onto executable transformations. This is either achieved through a template producing MTL code, or through a transformation creating a transformation program encoded as a model (higher-order transformation). However, writing transformations that produce transformation programs can be a daunting task. Thus, for better understandability, our explanation uses an example template language, which allows to see the output in bits of concrete syntax more intuitively.

Depending on what kind of transformation engine is used, the semantics of the resulting transformations are for instance formalized as abstract state machines [13] or as graph-based formalisms, such as triple-graph-grammars [20].

Continuing the above example, the subsequent paragraphs concentrate on the execution semantics for each of the operators given in Fig. 3, by using ATL-like code templates. At compile-time, each operator is derived into a fragment of ATL-code, only. A weaving in a certain language, though, stands for a complete ATL transformation. The generator, therefore, needs to integrate all these fragments into a complete ATL transformation as shown in Fig. 3.

Fig. 4 depicts pseudo-template code to show how semantics of operators can be specified. The template code consists of target code (ATL) in plain text, and template code in angle brackets which is bound at compile-time against LHS and RHS model elements. Square brackets contain control-flow instructions for the generator. In the template body of the parenting *FullEquivClass* operator for instance, templates of children operators are invoked.

```

template FullEquivClass {
  rule <smodel>2<tmodel> {
    from
      <sname>:<smodel>!<sclass>
      [extensionpoint: precondition,
       requiredType: BooleanExpression]
    to
      <tname>:<tmodel>!<tclass> (
        [applyTemplates(this.children)]
      )
  }
}

template CondEquivClass extends FullEquivClass{
  [extension FullEquivClass::precondition] {
    ( [applyTemplates(this.condition)] )
  }
}

template FullEquivRef {
  <tref> <- <sname>.<sref>
}

template FullEquivAttr {
  <tattr> <- <sname>.<sattr>
}

```

Fig. 4. Template code for integration operators.

To enable the extension of existing operators, a plugin-mechanism can be used. Thereby, templates can offer extension-points, into which templates of more specialized operators can plug-in their contributions. In Fig. 4, the *FullEquivClass* template declares an extension point that requires the contribution of a boolean expression. An example for an extension is given by the template of the *CondEquivClass* operator, which itself invokes a template that returns a boolean expression bound to the operator’s context. Through this inheritance-based reuse, a *CondEquivClass* iterator can inherit all of *FullEquivClass*’ behavior and additionally denote that a model element should be transformed if a certain condition holds, only.

5 Related Work

In this paper we have laid out an approach stretching across various abstraction layers, from global model management to local MTLs. As shown in Table 3, existing work typically focuses on certain abstraction levels, but, in our opinion, have not established a common understanding of how bottom-up approaches can be utilized for the implementation of top-down approaches in a scalable way. Furthermore, we compare related works on basis of certain key characteristics of our approach, like the employment of DSILs, OO-style model management scripts, the extensibility of operators and the explicit use of declarative integration specifications.

Related Work	Key Characteristics				Abstraction Levels				
	DSIL	OO	Extensible	Declarative	Glob. Comp.	Glob.	Intermed.	Loc. Comp.	Loc.
MMgmt.	-	-	-	+	+	+	-	-	-
MOMENT	-	-	~	+	-	+	-	-	+
GGT	+	-	-	+	-	+	+	+	~
AMW	+	-	+	+	-	-	+	~	-
EOL	+	-	+	~	-	-	+	+	+
ATL	-	-	-	~	-	-	-	-	+

Table 3
Comparison of related work.

Model management as proposed by Bernstein et al. aims at applying operators on the model level [3] [11]. In [4] a language-independent semantics is established to guide the implementation of model management operators. Although our work embraces the ideas of model management operators, e.g., by categorizing transformations, we also extend the notion of model management scripts with OO-mechanisms and explicitly focus on providing for scalable implementations through DSILs.

MOMENT [9] realizes model management operators by defining their semantics in QVT relations [21] that are mapped onto the algebraic specification language Maude, which, through term rewriting, executes the defined transformations. Although we focus on supporting the implementation of model management operators, the justified intention behind *MOMENT* to study formal properties of transformations could complement our approach in the future. However, our approach could potentially do this on the more abstract level of basically language independent integration operators and DSILs, as opposed to *MOMENT*, where Maude doubles as an execution environment as well as a testbed for proving formal properties.

The *Glue Generator Tool* (GGT) [7] aims at the reuse of existing MDA applications by specifying composition relationships between platform-independent models (PIMs), of which glue code for the integration of platform-specific models (PSMs) can be derived. Although rules similar to our integration operators are offered, our approach seems to be more flexible as we allow to extend the semantics of integration operators. Furthermore, the integration scenario described in GGT could be realized as a model management script carrying out the necessary transformations, which could allow for better modularity and maintainability of the overall approach.

The *Atlas Model Weaver* (AMW) [8] is a generic, extensible tool that aims at supporting modelers to establish semantic links between elements of arbitrary models or metamodels. The links are referred to as weavings and are formalized in a weaving metamodel, which can be extended to denote link types with special semantics. This extension mechanism is the basis for defining the syntax of integration operators and DSILs in our approach. Created weavings can then be subject to further processing like derivation of MTL code.

The *Epsilon Object Language* (EOL) is a language for managing models of arbitrary metamodels [19]. It can either be used as a standalone language for model navigation and comparison, or also as an infrastructure on which task-specific languages such as the *Epsilon Merging Language* (EML) or the *Epsilon Comparison Language* (ECL) can be built. Similarly, the *Atlas Transformation Language* (ATL) [1] is a hybrid (imperative/declarative) MTL based on the Eclipse Modeling Framework. In our opinion, both efforts present themselves as possible execution environments for our approach. Especially the definition of execution semantics for DSILs falling into categories like *Check* or *Fusion* could be conveniently accomplished relying on the expressiveness of languages like ECL or EML.

6 Conclusion and Future Work

In this paper we have proposed a conceptual approach which allows to define declarative model integration languages to implement model management operators, and to compose these into model management scripts. The distinction between local and global transformations fosters reuse of existing integration operators, and allows for sound composition of transformation functions. We have given a description of transformation categories and exemplified the composition of transformations into model management scripts. According to the understanding of transformations defining the *integration behavior* of metamodels, these scripts rely on an OO mechanism to invoke transformations which are dynamically bound depending on

a metamodel's type. Furthermore, we discussed the syntax and the semantics of an example integration language and described a way to extend integration operators.

We think of the approach described in this paper as a step towards the realization of future transformation systems which operate on the global model level, as opposed to the local model-element level, only. To raise the level of abstraction, domain specific languages in the form of declarative integration specifications play a key part in our approach. These are built on existing general-purpose transformation languages and are basically technology neutral. We have experimented with the implementation of various weaving languages which consist of operators that form the language kernels for the proposed transformation categories. Current work deals with building a technical framework based on existing model engineering infrastructure supporting our approach and a generically reusable toolset for various transformation categories.

In the context of ModelCVS, besides the integration of modeling tools, a crucial issue is the support for language evolution through metamodel modification. Future work will investigate to what extent such metamodel extensions can have characteristics analogous to traditional OO sub-classing, which would allow transformations to be inherited towards extended versions of metamodels.

References

- [1] ATL Homepage, <http://www.eclipse.org/gmt/atl/>, 2006.
- [2] Batory, D., *Multilevel models in model-driven engineering, product lines, and metaprogramming*. IBM Systems Journal, VOL 45, NO 3, 2006.
- [3] Bernstein, P.A., *Applying Model Management to Classical Meta Data Problems*. In Proceedings of the Conference on Innovative Data Systems Research (CIDR), Asilomar, California, January 2003.
- [4] Bernstein, P.A., A.Y. Halevy, S. Melnik, and E. Rahm, *A Semantics for Model Management Operators*. Microsoft Technical Report , June 2004.
- [5] Bernstein, P.A., S. Melnik, and E. Rahm, *Rondo: A Programming Platform for Generic Model Management*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2003.
- [6] Bézivin et al., *Combining Preoccupations with Models*. 1st Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSO at the 19th ECOOP, July 2005.
- [7] Bézivin, J., F. Jouault, D. Kolovos, I. Kurtev, and R.F. Paige, *A Canonical Scheme for Model Composition*. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 346–360, 2006.
- [8] Bézivin, J., E. Breton, M. Didonet Del Fabro, G. Gueltas, and F. Jouault, *AMW: A Generic Model Weaver*. In Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles, Paris, France, 2005.
- [9] Boronat, A., J.Á. Carsí, and I. Ramos, *Algebraic Specification of a Model Transformation Engine*. European Joint Conferences on Theory and Practice of Software (ETAPS06), Vienna, March 2006.
- [10] Bouzitouna, S., M.P. Gervais, and X. Blanc, *Models Reuse in MDA*. In Proceedings of the International Conference on Software Engineering Research and Practice (SERP05), Las Vegas, USA, June 2005.
- [11] Brunet et al., *A Manifesto for Model Merging*. In Proceedings of the 1st International Workshop on Global Integrated Model Management (GaMMA2006), Shanghai, May 2006.
- [12] Engel, K.-D., D.S. Kolovos, and R.F. Paige, *Using a Model Merging Language for Reconciling Model Versions*. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 143–157, 2006.
- [13] Gurevich, Y., P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications*. LNCS VOL 1912, Springer-Verlag, 2000.
- [14] Jouault, F., *Loosely Coupled Traceability for ATL*. In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany, 2005.

- [15] Kappel et al., *Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages*. In Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML), Genova, Italy, October 2006.
- [16] Kappel et al., *On Models and Ontologies - A Semantic Infrastructure Supporting Model Integration*. In Proceedings of Modellierung, Innsbruck, Tirol, Austria, March 2006.
- [17] Kappel et al., *Towards A Semantic Infrastructure Supporting Model-based Tool Integration*. In Proc. of the 1st Int. Workshop on Global integrated Model Management (GaMMA2006), Shanghai, May 2006.
- [18] Kapsammer, E., T. Reiter, W. Retschitzegger, and W. Schwinger, *Model Integration Through Mega Operations*. In Proc. of the Int. Workshop on Model-driven Web Engineering (MDWE), Sydney, July 2005.
- [19] Kolovos, D.S., R.F. Paige, and F.A.C. Polack, *The Epsilon Object Language (EOL)*. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 128–142, 2006.
- [20] Königs, A., and A. Schürr *Specification of Graph Translators with Triple Graph Grammars*. In Proc. of Graph-Theoretic Concepts in Computer Science, 20th Int. Workshop, Herrsching, Germany, 1994.
- [21] Object Management Group (OMG), *MOF QVT Final Adopted Specification*. November 2005.
- [22] Reiter, T., W. Retschitzegger, W. Schwinger, and M. Stumptner, *A Generator Framework for Domain-Specific Model Transformation Languages*. In Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS), Paphos, Cyprus, May 2006.
- [23] Stumptner, M., M. Schrefl, and G. Grossmann, *On the Road to Behavior-Based Integration*. In Proceedings of Conceptual Modelling, First Asia-Pacific Conference on Conceptual Modelling (APCCM2004), Dunedin, New Zealand, January 2004.