On the Use of Higher-Order Model Transformations

Massimo Tisi[1], Frédéric Jouault[2], Piero Fraternali[1],
Stefano Ceri[1], and Jean Bézivin[2]

[1] Politecnico di Milano, Dipartimento di Elettronica e Informazione
Milano - Italy
{massimo.tisi, piero.fraternali, stefano.ceri}@polimi.it
[2] INRIA, Centre Rennes - Bretagne Atlantique
Nantes, France
{frederic.jouault, jean.bezivin}@inria.fr

Abstract. The level of maturity that has been reached by model transformation technologies is proved by the growing literature on transformation libraries that address an increasingly wide spectrum of applications. With the success of the modeling and transformation paradigm, the need arises to address more complex applications that require a direct manipulation of model transformations.
The uniformity and flexibility of the model-driven paradigm allows this class of applications to make use of the same transformation infrastructure. This is possible because transformations can be translated into transformation models and given as objects to a different class of model transformations, called Higher-Order Transformations (HOT).
This paper provides an introduction to HOTs and a survey of the several application cases where their use is relevant. A number of possible future applications of HOTs is also proposed.

1 Introduction

The popularity of Model-Driven Engineering (MDE) is continuously growing and the reason behind this increasing success is mainly technological: a set of automation frameworks, built around model transformation technologies, are reaching a good level of maturity. This maturity is related to two important technological drivers. At first, common recognized formalisms (e.g. MOF, Ecore, KM3[23]) have allowed the explicit characterization of several metamodels; then more and more libraries of transformations have started to gather reusable model transformations expressed in declarative rule-based languages (e.g. QVT, ATL[24]).

The evolution of model-driven environments since its first steps can be outlined distinguishing three phases. In a first phase, early MDE tools were limited to assistance in drawing models, sometimes reverse-engineering them from existing code, and to generation of structural skeleton of programs from diagrams. Such limited approaches were relegating models to the role of mere program documentation, difficult to maintain, with a cost that was not clearly justified by a tangible improvement in software quality.

The second phase saw an increasing success of model transformation technologies to automate forward engineering and several other software development activities. Automation is among the most appealing means to reduce costs and increase productivity. While the idea of automating the generation of a significant part of the program code accompanies MDE since its first steps, the success of code generation only started when it was able to produce code whose efficiency was comparable to hand-crafted code. Since that moment, the use of models in software development started to become much wider than their role as drivers of the implementation. Once models become an integral part of the software engineering process, there is no reason not to exploit the same transformation infrastructure to automate other tasks. This lead to the development of a vast library of transformations to accomplish several activities automatically, such as metrics evaluation, testing, generation of documentation.

Finally in a third phase, while models and transformations are still a central part of the software development process, they start to become also an integral part of the developed system. Model-based software systems appear, where models and model transformations are first-class elements of the runtime architecture, together with data structures and programs. In these systems, models are used to represent several heterogeneous types of information. They can be handled natively at runtime, and system logic can be represented by complex transformation workflows.

It is especially in this third phase that the idea of transformation manipulation naturally arises. As part of the developed system, transformations can be themselves generated and handled by model-driven development, exactly like traditional programs. While transformation manipulation can be performed by means of an independent methodology (e.g., program transformation, aspect orientation), the elegance of the model-driven paradigm allows again the reuse of the same transformation infrastructure. To achieve this objective, the concept of model transformation needs to be extended with that of transformation model [11]. The transformation is represented by a transformation model that has to conform to a transformation metamodel. Just as a normal model can be created, modified, augmented through a transformation, a transformation model can itself be instantiated, modified and so on. This uniformity is beneficial in several ways: especially it allows reusing tools and methods, and it creates a framework that can in theory be applied recursively (since transformations of transformations can be transformed themselves).

Once the boundary between development-time transformations and execution-time transformations is weakened, a wide set of application patterns appear that involve transformation models in the roles of both manipulation program and manipulated object. This paper provides a first survey and classification of these applications and the related transformation patterns.

The rest of the paper is organized as follows: Section 2 introduces definition and structure of higher-order transformations, i.e. transformations of transformations; Section 3 presents the outline of the classification and the main transformation types; Section 4, 5, 6 and 7 describe the four main areas in which

the survey is divided, namely transformation synthesis, analysis, composition and modification; Section 8 suggests other applications that can be addressed in future by higher-order transformations; Section 9 draws the conclusions.

2 Higher-Order Transformations

An essential prerequisite for fully exploiting the power of transformations is their treatment as objects. This demands the representation of the transformation as a model conforming to a transformation metamodel.

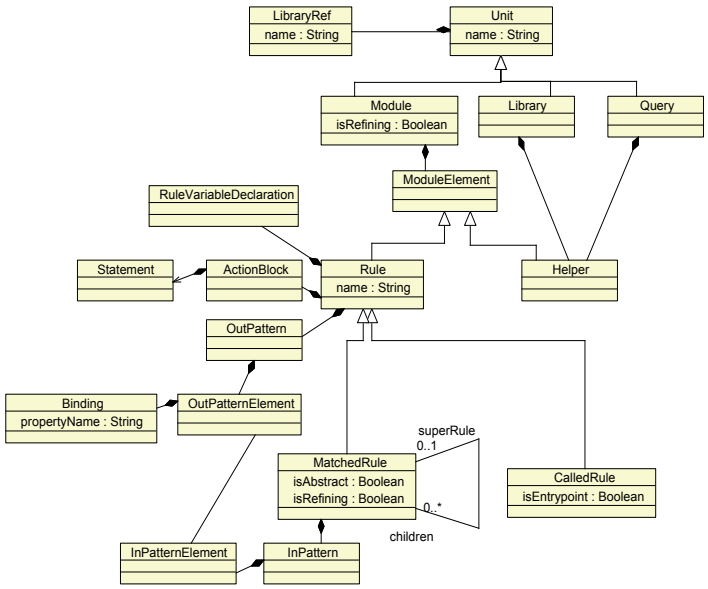

Fig. 1. Simplified version of the ATL Metamodel.

Not all transformation frameworks provide a transformation metamodel. In this work we will mainly refer to the AmmA framework [27] that contains a mature implementation of the ATL transformation language. Within AmmA an ATL transformation is itself a model, conforming to the ATL metamodel. Figure 1 shows the main classes of the ATL metamodel. Besides the shown elements like the central classes of *Rule*, *Helper*, *InPattern*, and *OutPattern*, the ATL metamodel also incorporates the whole OCL metamodel to represent expressions on the manipulated models.

Once the representation of a transformation as a transformation model is available, a HOT can be defined as follows:

Definition 1 (Higher-order transformation). *A higher-order transformation is a model transformation such that its input and/or output models are themselves transformation models.*

According to this definition HOTs either take a transformation model as input, produce a transformation model as output, or both. An example of a HOT in the AmmA framework is shown in Figure 2. This example reads and writes a transformation, e.g. with the purpose of performing a refactoring. The three operations shown as large arrows at level M1 (Models) are:

- *TCS Injection.* The textual representation of the transformation rules is read and translated into a model representation. This example uses for this step a generic program that is parametrized with a model representing the concrete syntax of the ATL language. The Textual Concrete Syntax is described in AmmA by means of the TCS formalism [22]. The generated model is an instance of the ATL metamodel (Figure 1).
- *HOT.* The transformation model is the input of a model transformation that produces another transformation model. The input, output and HOT transformation models are all conforming to the same ATL metamodel.
- *TCS Extraction.* Finally an extraction is performed to serialize the output transformation model into a textual transformation program.

Note that the injection and extraction operations are not always used during a HOT. For instance, the source transformation model may come from a previous step, and already be in the form of a model. Similarly, the target transformation model is sometimes reused as a model without an immediate serialization.

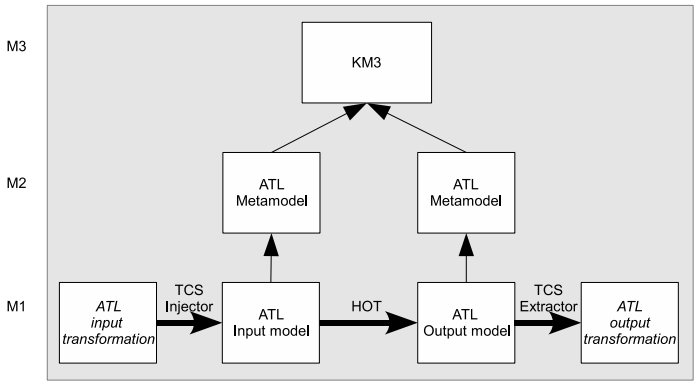

Fig. 2. A typical example of Higher-Order Transformation.

3 A survey of Higher-Order Transformations

The next sections are an overview of the literature on model transformations that involve the use of HOTs for specific tasks.

This survey comprises all publications known to us that are related to HOTs in the ATL language. ATL appears to be the preferred language for HOTs development to date, and we were able to gather a set of 44 transformations from

previous work. We add to this set other notable examples of HOTs in other frameworks. Most of the described model transformations are freely available and highly reusable. They constitute a first comprehensive library of HOTs.

The survey is organized in a two-level hierarchy. The first level is focused on the identification of base *transformation patterns*, each of them representing the usage pattern of a given class of HOTs. In the second level, each one of the base groups is further divided by considering different variants of the patterns.

We identified four transformation patterns that are shown in Figure 3. These patterns include the following models: the HOT, its input and output models and the input and output models of the transformations that are handled by the HOT. Models are included together with their respective metamodel and they are linked by the following associations: the *conforms-to* relationship between a model and its metamodel (represented as a thin arrow), and the *transforms* relationship between a transformation and its output and input models (represented as thick arrows). The *transforms* arrows of the HOTs are shown in a darker shade of gray. The four base patterns are:

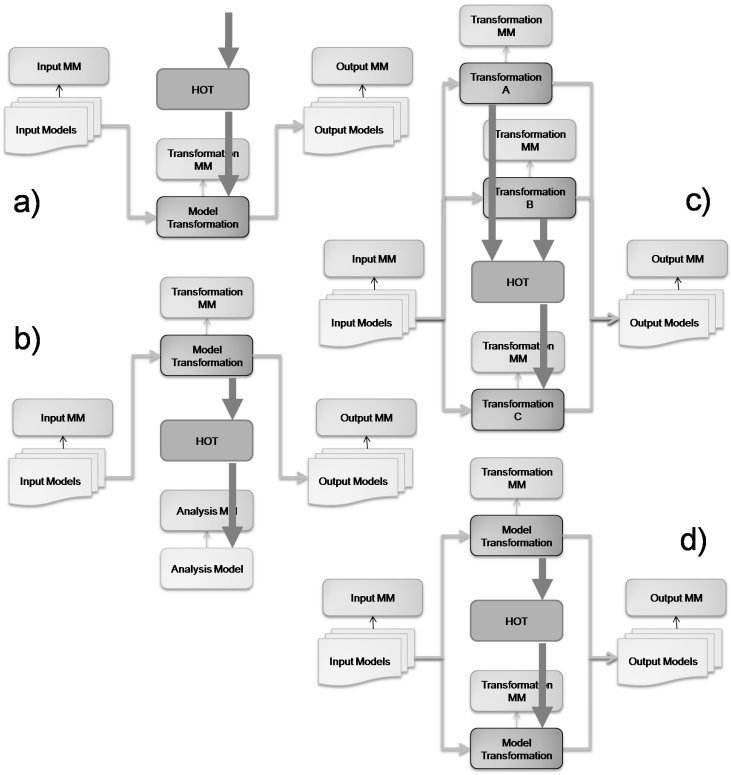

Fig. 3. Base HOT patterns: a) Transformation synthesis, b) Transformation analysis, c) Transformation composition (decomposition not shown), d) Transformation modification.

Transformation Synthesis (Section 4) is the common pattern for HOTs that generate transformations from different information sources. These HOTs are defined by two conditions: 1) the output model is a transformation; 2) the input models, if present, are not transformations;

Transformation Analysis (Section 5) is the pattern for HOTs that take transformations as input, to generate different kinds of data, in the form of output models. More precisely: 1) they have a transformation as input model; 2) they do not have transformations as output models.

Transformation (De)composition (Section 6) is the pattern for HOTs that take multiple transformations as input (composition) and/or output (decomposition). Three conditions define these HOTs: 1) at least one of the input models must be a transformation; 2) at least one of the output models must be a transformation; 3) the input and/or output models must contain more than one transformation.

Transformation Modification (detailed in Section 7) is the pattern for HOTs that take a transformation as input and generate a modified version of the same transformation. These HOTs must have: 1) one transformation as input; 2) one transformation as output.

Additional input or output models are allowed for all the previous HOTs, with the condition that they are not transformation models.

4 Transformation Synthesis

All the cases of transformation synthesis in our survey can be inserted in one of the following sub-classes: 1) mapping implementation, i.e. the generation of an executable version of an abstract mapping; 2) generic metamodel, i.e. the construction of transformations that are generic with respect to the input or output metamodel.

4.1 Mapping implementation

The semantics of a transformation language, while representing an abstract mapping between metamodels, needs to be concrete enough to allow a direct execution in a transformation engine. For a set of practical applications this level of abstraction is not sufficient and a higher-level specification is preferred. A more abstract mapping has advantages in terms of readability and manageability and can provide useful features that may not be in the transformation language, such as bi-directionality. However the non-executable representation needs to be translated into an executable transformation. This is a typical application of HOTs. We refer to this class of HOTs with the term *mapping implementation*, since the abstract representation can be considered as the higher-level specification of a mapping that needs to be implemented as a transformation.

In [15] a practical use case for mapping implementation HOTs is deeply studied. The motivating scenario is enabling tool interoperability between different bug-tracking systems. Two autonomous software development companies

that need to collaborate rely on different bug-tracking systems and they need an import/export mechanism between their bug-tracking metadata. A declarative correspondence model is semi-automatically generated by the analysis of the metamodels used by the two systems. The correspondence model represents a set of relationships between model elements. To address the problem of the possibly infinite kinds of relationships between source and target elements, the authors use an extension mechanism. The supported links are classified in three major groups: *similarity expressions* (e.g., equivalence of model elements), *mapping expressions* (e.g., one-to-many, many-to-one or many-to-many relationships between source and target model elements) or *data-value expressions* that involve values of related attributes (e.g., the relationship between the values of the *status* attribute for a bug request in the two systems). Finally a HOT translates this representation in an executable transformation.

The same authors provide another example of this approach in [7]. Using AMW (i.e. a model that represents generic correspondences between models [29]) and an ATL HOT it is possible to generate two model transformations that translate a KM3 metamodel in the SQL DDL and vice versa. In this example the HOT takes as input a AMW weaving model (i.e. a correspondence model) between a KM3 metamodel and a SQL DDL metamodel. The weaving model defines a correspondence between metamodel classes and tables on a database. This model is then used to produce the two implementation transformations for bidirectional translation.

HOTs to translate an AMW correspondence model into an implementation are provided also in [1]. This example contains an extension of the core weaving metamodel to specify correspondences from a metamodel with flat structures and foreign keys relationships (as in relational databases) to a metamodel that contains nested structures (as in XML). The two proposed HOTs generate respectively an ATL and an XSLT executable version of the AMW specification.

The same general schema is followed by:

- [25] that addresses the issue of adapting a model to an evolving metamodel;
- DUALLy [28], an automated framework that allows architectural languages and tools interoperability;
- the MML2MMR transformation in [20] for studying the integration of DoDAF [5] metamodels and models.

In all these cases an AMW mapping that correlates two different metamodels is translated into two transformations at model level (one for each direction of the mapping). Each AMW link corresponds to one or more rules or helpers in the generated transformations. [16] provides an abstract specification of several transformations. Among them, the *patchgen* transformation is the specification of the previous transformation group.

[14] presents an alternate representation for differences between models that conform to the same metamodel. In this proposal the difference model conforms to a new metamodel that is derived from the input metamodel using a model transformation. This transformation specializes the metaclasses of the input metamodel in three subclasses: for each class *ClassName* an 1) *AddedClass-*

Name, a 2) *DeletedClassName* and a 3) *ModifiedClassName* are introduced. The instances of these metaclasses represent all the differences between the analyzed models. In this case too, starting from the difference representation, the authors use a HOT to derive an ATL implementation (they call this implementation *difference animation*).

Ecore2RDF [19] is a bridge between the EMF and SemanticWeb technical spaces. A custom mapping metamodel, which extends the whole ATL metamodel by adding novel constructs, is the core formalism of this solution. Two HOTs generate the two directions of the mapping.

Mapping implementations are common also outside the ATL environment. For instance WebRatio[8] is a modeling environment for the WebML domain-specific language that allows the generation of a Web application from structure and navigation models. WebRatio includes a tool called EasyStyle to generate the presentation of Web pages. EasyStyle is based on a XSLT HOT which takes as input an HTML template annotated with custom tags, i.e. placeholders for the content elements of the page. The XSLT HOT translates this HTML file into another XSLT transformation that generate pages conforming to that template. This application is not different from the previous ones, since the HTML template can be considered as a mapping that relates the elements of the input model (the Web application model) with the elements of the output model (the page layout model).

4.2 Generic metamodels

Several application cases require the development of generic model transformations that need to take as input or output a metamodel that is not known a priori. The typical solution in these cases is using HOTs for the on-the-fly generation of those model transformations that could not be easily developed manually with a sufficient generality.

For instance the proposal in [18] includes a HOT that takes a KM3 metamodel as source model and generates a transformation for conflict detection. The output transformation takes two source models (an implementation and an architecture model) that conform to the same metamodel and generates a target model that is the union of the two source models with each of the model elements labeled convergent, divergent or absent depending on their occurrence in one or both of the architecture and implementation models.

The need to use a HOT in this case is related to the limitations of the ATL language. The expressing power of ATL (at least in its declarative form) does not allow the direct development of a generic conformance checking transformation. For instance a specific input and output metamodel needs to be specified at development time. The reflectivity features of declarative ATL are not enough to outflank the problem (an alternative using imperative ATL is possible but not painless). This limitation is shared by several other transformation languages.

An analogous problem is presented in [6], where the HOT is used to generate a generic copier for models conforming to a metamodel that is only known at runtime. [12] presents an alternative implementation of the model copier as a

HOT based on the transformations in [36] and applied to product lines. In this case a so-called *configuration* is obtained by a copier transformation that selects only a subset of the elements of an extremely generic default model. To keep the genericity with respect to the configuration metamodel, a HOT dynamically generates the copier.

In [10] the authors develop a set of transformations to bridge models expressed in the Microsoft DSL framework to the Eclipse Modeling Framework. A first sequence of transformations performs the bridging at M2 level, translating metamodels between the two frameworks. A second set deals with the models. The last step of this process is a transformation that needs to build models conforming to the metamodel generated in the first step. Knowledge about this metamodel is available only at runtime and can not be embedded into the last transformation. In the proposed solution the last transformation is dynamically generated from a HOT that *instantiates* some general ATL rules for the elements of the input metamodels.

Generating test cases for model transformations is a task that can be easily performed by exploiting the syntactic description of the input and output domains of the transformation, given by the input and output metamodels. This kind of approach to test set generation is referred to as black-box generation, since the process does not involve an analysis of the internal structure of the transformation under testing. In [9] the last step of the test data generation process is the *MM2TM* transformation for the generation of test models according to a test criterion. In [9] the generation of models is guided by *model fragments* that are particular model chunks identified in previous steps: each model fragment should appear at least once in generated test models. To be implemented as a generic model transformation, applicable to any input metamodel, the *MM2TM* transformation has to be dynamically generated for each input metamodel. The generating HOT contains the logic of the coverage criteria (i.e., a different HOT has to be developed if we want to use a different criteria).

5 Transformation Analysis

The generation of an output model that represents a particular analysis of an input transformation is inherently a HOT.

An example of these HOTs is given in [4]. The ATL to Problem use case describes a transformation from an ATL model into a Problem model. The generated Problem model conforms to a single-class Problem metamodel and contains the list of non-structural errors and warnings that have been identified within the input ATL model. The transformation assumes the input ATL model is structurally correct, i.e. it conforms to the ATL metamodel.

[26] shows a complex example of transformation analysis in the GReAT framework. The HOT in this case reads a model transformation to derive a variability metamodel: for each mapping defined in the transformation, the types of the input and output elements are extracted and gathered into a model of the possible variabilities of the system.

A HOT included in the Topcased project [33] analyzes a transformation to address the problem of conformance checking. In [33] the authors describe a framework that translates abstract SimplePDL models into Petri nets. The SimplePDL2PetriNet transformation is strongly dependent on a set of implementation choices made by the user. When the author faces the task of checking the correctness of the Petri nets starting from SimplePDL specifications, they need to analyze also the SimplePDL2PetriNet transformations to retrieve the translation choices. The corresponding ATL HOT takes as input the SimplePDL2PetriNet transformation, the Petri net to check, and the high-level specifications. As output it returns a boolean value expressing the result of the conformance test.

6 Transformation Composition

There are two mechanisms to perform the composition of model transformations. External composition consists in chaining separate model transformations and in passing models from one transformation to another. Internal composition composes two model transformation definitions into one new model transformation, with a typically complex merge of the transformation rules. Internal composition, when performed by a model transformation, is a higher-order problem.

[36] provides an example of internal composition performed by a HOT. The paper uses *superimposition* as the composition mechanism to merge two ATL transformation modules into a single output transformation. Superimposition is a simple kind of internal composition in which a transformation module A is superimposed to a transformation module B obtaining a transformation module C, such that: 1) C contains the union of the sets of transformation rules and helpers of A and B; 2) C does not contain any rule or helper of B, for which A contains a rule or helper with the same *name* and the same *context*.

In [36] the HOT is split up in the external composition of two HOTs: ATLCopy.atl that is a simple copying transformation, and Superimpose.atl that provides the special transformation rules for superimposition.

7 Transformation Modification

In our survey the most common use of HOTs is related to the modification of existing transformations. The HOTs of this kind can be more precisely classified in one of the sub-classes described in the following sections.

7.1 Transformation variants

The transformation variants approach is particularly useful when developing product lines, as in [32]. In this work HOTs are programmed using the model-to-text transformation language MOFScript. Two sets of HOTs are used to generate two kinds of variability: 1) *Platform Variability* is implemented by generic HOTs

that are developed once and can be applied on every input transformation; *Intra-domain Variability* is implemented by ad-hoc HOTs that hard-code some domain knowledge to change the internal structure of the original transformation.

A particular kind of variants is produced during mutation analysis. Mutation analysis consists in systematically creating faulty versions of a program (called mutants) and in checking the efficiency of a test dataset to reveal the faults in these erroneous programs. The main interest of mutation analysis is to provide an estimate of the quality of a test dataset with the proportion of faulty programs it detects. To be effective, mutation analysis must create mutant programs that correspond to realistic faults. The faults are injected into the correct program by means of a set of *mutation operators*. The problem to identify a set of realistic mutation operators for model transformations, independently from a particular transformation language has already been studied in [30]. The authors distinguish four error classes, correspondent to the four main operations performed by model transformations, i.e navigation, filtering, output creation, input modification. For each one of the error classes, a set of mutation operators is defined to represent the most common mistakes in transformation development. For instance one of the simplest mutation operators is *Collection Filtering Change with Deletion (CFCD)* that represent the mistake of forgetting a needed filter from the left hand side of a transformation rule. [9] proposes a formalization of mutation operators as a set of HOTs and provides an implementation of the mutation framework in the EMF platform.

7.2 Feature weaving

HOTs can be easily used to weave cross-cutting concerns into a model transformation. Examples of these concerns are related to debugging, traceability, program tracing. A HOT for adding a cross-cutting concern can usually be programmed with extreme generality, and complete independence from the logic of the original transformation. In this way the same HOT can be used as a general means to add that specific feature to any transformation. As a further consequence several features can be added to the same transformation by sequentially applying the corresponding HOTs.

The use of HOTs in this application case is especially convenient when the transformation language does not provide a native implementation of aspect orientation. An aspect oriented mechanism can be in fact considered as a particular type of higher-order transformation that is performed on-the-fly when the original transformation is executed. A further solution to attach cross-cutting concerns to existing rules could be based on rule inheritance. Such a solution is already programmable in the current ATL but it lacks generality, having a tighter coupling with the program logic. Finally a more general alternative could be implemented by using reflection, letting the developer dynamically plug-in code to existing rules. The coupling between the feature code and the program logic could be relatively loose in this case.

There are several cases in literature that exploit ATL HOTs for cross-cutting concerns. [3] describes an ATL2BindingDebugger HOT that adds a debug in-

struction to each attribute binding in an input ATL transformation. Each time that a value is assigned to an attribute using a binding in the target element of a rule, a line is printed in the logfile containing the names of the rule, of the target element and of the assigned value. This feature is easily implemented augmenting each binding in the form $targetAttribute \leftarrow value$ with a print on a logfile using the syntax $targetAttribute \leftarrow value.debug('ruleName.targetName.value')$. The required ATL2BindingDebugger HOT is very concise and shows how nicely HOTs can address cross-cutting concerns.

[21] and [2] provide two different implementations for another application case, in which the addressed cross-cutting concern is traceability, i.e. the maintenance of a set of links between corresponding source and target model elements. In [21] traceability is implemented by adding to each original transformation rule the production of a traceability link in an external ad-hoc traceability model (conforming to a small traceability metamodel). The solution presented in [2] is analogous, with a slightly higher complexity, due to the fact that the traceability link is represented by an ad-hoc extension of the *core weaving metamodel*.

7.3 Changing the engine execution mode

A higher-order transformation can be used to modify the execution mode of the transformation engine for particular model transformations. Some practical problems, in fact, would find a more natural solution if the transformation engine had a different execution semantics. This would allow for more readable and concise transformation code.

For example in [9] transformations are used to implement mutation operators. The most natural way to express a mutation operator is a single transformation rule, whose intended application would require the generation of a different output model (i.e.; a mutation) each time that the rule is applied to a single mutation point. In the chosen solution a HOT translates the simple mutation specification with the mutation semantics in an equivalent, much more verbose, version with the standard execution semantics.

As a side note the solution in [9] has another reason of interest in being the only second order HOT in this survey. It is a transformation that takes as input and output other HOTs, i.e. the mutation operators.

7.4 Transformation language extension

One of the means to simplify the specification of a transformation is the addition of new language features, e.g., a new operator. However the direct extension of a transformation language and engine may not be the optimal solution. Often the new feature would be useful in only a limited set of cases, not enough to justify the cost of making the engine and notation heavier.

HOTs provide a good alternative for language extension. The transformation language can be easily extended by adding new metaclasses to the transformation metamodel. Then a HOT can be developed to convert the new elements into a set of rules in the existing transformation language. In this way the extension

does not require to change the actual engine and can be easily applied only to transformations that need it. Examples of this kind of transformations are described in [31]. [17] uses this approach to develop an extended version of ATL to address the specification of model matching strategies.

7.5 Parametric transformation

Being faithful to the "everything is a model" ideal, ATL does not support an explicit parameter initialization mechanism but it requires to define a metamodel for the parameters and to add it as a further input to the transformation.

A common alternative among ATL developers is the use of HOTs to overwrite a parameter value directly defined within the original transformation. This solution is sometimes more concise (thanks to the ATL refining mode) and flexible of the standard one.

7.6 Transformation adaptation

Within a model-driven system it is possible that a set of properties for a specific transformation is known only at runtime. Often such a problem is addressed by representing these properties as a *configuration model*. The configuration model is given as input to the transformation, guiding a set of choices that are hard-coded into a limited set of rules. This approach, however, requires to define, at development time, the kinds of variations that will be possible at runtime.

HOTs are the natural means to remove any constraint on the possible runtime adaptations of the base transformation. [34] describes an example of this kind, and provides a simple implementation that chooses at runtime among different variants of rule, based on runtime statistics.

8 Other applications for HOTs

This section lists application areas for HOTs that have not yet been explored by works in literature. The list does not want to be exhaustive but it has the purpose of providing some directions for future works.

Transformation metrics. One of the most natural applications of HOTs is the analysis of model transformations for deriving metric values. Several works implement measurement transformations for generic models (e.g., [35]) and these transformations are of course applicable also to the analysis of transformation models. However, current research lacks a set of specifically higher-order transformations for the generation of transformation metrics.

Transformation refactoring. The building of model transformations could benefit from a set of transformation refactorings, encapsulating best practices for transformation development. These refactorings could be implemented as HOTs and executed by the users during the editing of the transformation. A theory of model transformation refactorings has not been investigated and refactoring HOTs have not been developed yet.

Transformation optimization. The execution of a model transformation could be improved by preceding the execution phase with a preprocessing step. In this step the transformation could be analyzed to detect common patterns and translated into an equivalent version that trades a speed or memory improvement with a loss in readability and manageability. This topic has not been addressed by previous work.

Partial Evaluation. While a model transformation has often several input models, it is very common the case in which some of the input models are not subject to frequent changes. At every execution, the transformation needs to process all the input models, including the stable ones. Often it is possible to obtain a remarkable performance gain by removing the stable input models and hard-coding their information inside the transformation. This process, called *partial evaluation* has two benefits: 1) the new version of the transformation needs to process only the input models that actually change between different executions; 2) the hard-coding of some input models can allow ad hoc optimizations and a simplification of the transformation structure. The implementation of a general HOT for the partial evaluation of any transformation, with respect to any input model has never been addressed.

9 Conclusions

In this paper we have presented a categorized survey of HOTs. The categorization is based at its first level on the concept of transformation pattern, for which we have given an informal definition. Four base patterns are identified. Then HOTs are further divided based on some variants of the base patterns. Table 1 summarizes this list of HOTs showing their input and output metamodels and the transformation areas they belong to.

The main contributions of this paper are: 1) to provide an index of the previous work on HOTs, and to identify some important unexplored areas; 2) to provide a list of applications for HOTs proving the practical value of this approach and the necessity of a deeper research in this direction; 3) to identify a limited set of common transformation patterns that involve HOTs; 4) to provide a first categorization of existing and future HOTs based on their role within transformation patterns.

References

1. AMW to ATL. `http://www.eclipse.org/gmt/amw/examples/#AMW_2ATL_XSLT`.
2. AMW Traceability. `http://www.eclipse.org/gmt/amw/usecases/traceability`.
3. ATL to BindingDebugger. `http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2BindingDebugger`.
4. ATL to problem. `http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2Problem`.
5. DoDAF 2004 volume II: product description, 4/2/2004. `http://www.defenselink.mil/cio-nii/global_Info_grid.html`.

Name (# of cases)	Language	Source MM	Target MM	Type	Ref.
AMWtoATL_KM32SQL	ATL	AMW	ATL	Implementation	[7]
AMWtoATL_MantisBug	ATL	AMW	ATL	Implementation	[15]
AMWtoATL	ATL	AMW	ATL	Implementation	[1]
AMWtoXSLT	ATL	AMW	XSLT	Implementation	[1]
ATL2BindingDebugger	ATL	ATL	ATL	Weaving	[3]
ATL2Tracer	ATL	ATL	ATL	Weaving	[21]
ATL2WTracer	ATL	ATL	ATL	Weaving	[2]
ATL2Problem	ATL	ATL	Problem	Analysis	[4]
KM32CONFATL	ATL	KM3	ATL	Generic	[18]
KM32ATLCopier	ATL	KM3	ATL	Generic	[6]
AMWtoATL_Kelly	ATL	AMW	ATL	Implementation	[25]
MMD2ATL	ATL	KM3	ATL	Implementation	[14]
MMTtoMT	ATL	ATL, Ecore	ATL	Execution	[13]
MSDSL2EMF	ATL	KM3	ATL	Generic	[10]
Superimpose	ATL	ATL, ATL	ATL	Composition	[36]
ATLCopy	ATL	ATL, ATL	ATL	Composition	[36]
HITransform	MOFScript	MOFScript	MOFScript	Variants	[32]
Topcased	ATL	ATL	ATL	Analysis	[33]
Metamodel2Derivation	ATL	Ecore	ATL	Generic	[12]
DUALLyLeft2Right (2)	ATL	AMW, Ecore, Ecore	ATL	Implementation	[28]
Easystyle	XSLT	HTML	XSLT	Implementation	[8]
HOT4Tests	ATL	Ecore	ATL	Testing	[9]
Mutators (11)	ATL	ATL, Trace	ATL	Mutation	[9]
SingleApplication	ATL	ATL	ATL	Execution	[9]
patchgen	ATL	AMW	ATL	Implementation	[16]
propagate	ATL	ATL,INMM,INMM,AMW	ATL	Implementation	[16]
VariabilityMM_HOT	GReAT	GME, GReAT	GReAT	Analysis	[26]
MML2MMR (2)	ATL	AMW, Ecore, Ecore	ATL	Implementation	[20]
AML2ATL	ATL	AML	ATL	Extension	[17]
UITransReconfig	ATL	ATL, USRMM	ATL	Adaptation	[34]
Ecore2RDF (2)	ATL	Meo, Ecore, OWL	ATL	Implementation	[19]

Table 1. Summary of HOTs.

6. KM3 to ATL copier. `http://www.eclipse.org/m2m/atl/atlTransformations/#KM32ATLCopier`.
7. Translating KM3 into SQL using AMW and ATL. `http://www.eclipse.org/gmt/amw/examples/#AMW_KM32SQL`.
8. WebRatio. `http://www.webratio.com/`.
9. WebRatio MD Framework. `http://home.dei.polimi.it/mbrambil/legacytomda`.
10. J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA*, 2005.
11. Jean Bézivin, Fabian Büttner, Martin Gogolla, Frederic Jouault, Ivan Kurtev, and Arne Lindow. *Model Transformations? Transformation Models!*, volume Model Driven Engineering Languages and Systems, pages 440–453. 2006.
12. G. Botterweck, L. O'Brien, and S. Thiel. Model-driven derivation of product architectures. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 469–472. ACM, NY, USA, 2007.
13. M. Brambilla, P. Fraternali, and M. Tisi. A metamodel transformation framework for the migration of WebML models to MDA. MDWE at Models2008, 2008.
14. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6:165–185, 2007.
15. M. D. Del Fabro, J. Bezivin, and P. Valduriez. *Model-Driven Tool Interoperability: An Application in Bug Tracking*, page 863. LNCS, 2006.

16. M. Didonet Del Fabro and J. Bézivin. Generic model management: from theory to practice. First Intl. Workshop on Towers of Models, 2007.
17. Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. A domain specific language for expressing model matching. In *Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09)*, Nancy, France, 2009.
18. B. Graaf, A. van Deursen, B. Baudry, A. Faivrea, S. Ghosh, and A. Pretschner. Using MDE for generic comparison of views. In *Proceedings of the 4 thInternational Workshop on Model Design, Verification and Validation (MoDeVVa 2007)*, 2007.
19. G. Hillairet, F. Bertrand, and J. Y Lafaye. MDE for publishing data on the semantic web. *Transf. and Weaving Ontologies in MDE (TWOMDE) at MODELS'2008*.
20. A. Jossic, M. D. Del Fabro, J. P. Lerat, J. Bézivin, F. Jouault, and S. A. S. Sodius. Model integration with model weaving: a case study in system architecture. In *International Conference on Systems Engineering and Modeling ICSEM'07.*, 2007.
21. F. Jouault. Loosely coupled traceability for ATL. In *Workshop on Traceability at ECMDA 2005, Nuremberg, Germany*, 2005.
22. F. Jouault, J. Bézivin, and I. Kurtev. TCS:: a DSL for the specification of textual concrete syntaxes in model engineering. *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006.
23. Frédéric Jouault and Jean Bézivin. *KM3: A DSL for Metamodel Specification*, volume Formal Methods for Open Object-Based Distributed Systems. LNCS, 2006.
24. Frédéric Jouault and Ivan Kurtev. *Transforming Models with ATL*, volume Satellite Events at the MoDELS 2005 Conference, pages 128–138. 2006.
25. P. Cointe K. Garcés, F. Jouault and J. Bézivin. Adaptation of models to evolving metamodels. Technical report, 2008.
26. A. Kavimandan, R. Klemm, and A. Gokhale. Automated Context-Sensitive dialog synthesis for enterprise workflows using templatized model transformations. In *EDOC '08.*, pages 159–168, 2008.
27. I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL frameworks. In *21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 602–616, Portland, Oregon, USA, 2006. ACM.
28. H. M. Malavolta, P. Pelliccione, and D. A. Tamburri. Providing architectural languages and tools interoperability through model transformation technologies. Technical report, TR 004-2008. Available at the DUALLY site, 2008.
29. D. D. F. Marcos, B. Jean, J. Frdric, B. Erwan, and G. Guillaume. AMW: a generic model weaver. In *1res Journes sur l'Ingnierie Dirige par les Modles*, 2005.
30. Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. *Mutation Analysis Testing for Model Transformations*, pages 376–390. 2006.
31. O. Muliawan. Extending a model transformation language using higher order transformations. In *15th Working Conf. on Reverse Engineering, WCRE*, 2008.
32. J. Oldevik and O. Haugen. Higher-Order transformations for product lines. In *Proceedings of the 11th International Software Product Line Conference (SPLC 2007*, pages 243–254. IEEE Computer Society Washington, DC, USA, 2007.
33. M. Pantel, ACADIE team, OLC team, and TOPCASED team. The TOPCASED project. *Int. Conf. on Embedded Real Time Software*, 2006.
34. J.S. Sottet, V. Ganneau, G. Calvary, J. Coutaz, J.M. Favre, and R. Demumieux. Model-Driven adaptation for plastic user interfaces. In *Interact 2007*, 2007.
35. É Vépa, J. Bézivin, H. Brunelière, and F. Jouault. Measuring model repositories. In *Proceedings of the Model Size Metrics Workshop at the MoDELS/UML 2006 conference, Genova, Italy*, 2006.
36. D. Wagelaar. Composition techniques for Rule-Based model transformation languages. In *Icmt 2008, Eth, Switzerland, July 1-2*, page 152. Springer, 2008.