Chaining Model Transformations

Anne Etien and Vincent Aranega
LIFL CNRS UMR 8022
Université Lille 1
France
firstname.lastname@lifl.fr

Xavier Blanc
LaBri CNRS UMR 5800,
Université Bordeaux 1
France
Xavier.Blanc@labri.fr

Richard F. Paige
Department of Computer Science
University of York
UK
richard.paige@york.ac.uk

ABSTRACT

Model transformation is one of the key practices of Model-Driven Engineering. Building very large model transformations may benefit from the construction of small transformations, in order to manage complexity and enhance reusability, maintainability and modularity. The decomposition of transformations into smaller ones raises the issue of assuring the validity of a composition: if two or more transformations are *chained* together, are the results of executing the chain the expected ones?

This paper addresses the challenge of determining if two transformations are conflicting. Transformations can conflict in numerous ways, e.g., in terms of preconditions, post-conditions, behaviour of individual rules. In this paper, we demonstrate a strong notion of conflict, via commutativity: two transformations do not conflict if they can be chained in either order, and in doing so produce identical results. We propose an approach to detecting such potential conflicts based on static analysis, exploiting an intermediate representation of transformations independent of any concrete language.

1. INTRODUCTION

Model Driven Engineering (MDE) advocates the principle of *separation of concerns*, through the extensive use of models in all steps of the software development cycle [17]. A number of approaches [14, 18] recommend to adopt this principle for designing and implementing transformations. Thus, it is desirable to *decompose* transformations, much as engineers decompose other artefacts (like architectures and object-oriented designs). If large transformations are suitably decomposed, their scalability, maintainability and reusability may be improved [14, 20], and it thereafter becomes tractable for transformations to be systematically engineered [3].

The decomposition of transformations into smaller ones raises a new issue: assuring the validity of the composition — *i.e.* the *chaining* — of these transformations. Understanding what constitutes a *valid* chain of transformations, and providing support for checking validity, is the focus of this paper.

This paper deals with transformations where the intersection of the input and output metamodels is not empty. In [5], we proposed an analysis based on the metamodels involved in the transformations to identify when transformations can be chained. These constraints effectively allow us to prune some chains that are invalid. However, this analysis had a limitation: in some situations where two transformations f and g were analysed, it was determined that these transformations could be chained in either order (*e.g.* $f \circ g$ or $g \circ f$), though each ordering produced different models. A finer grained analysis is thus required.

This paper addresses the challenge of identifying conflicts between transformations and thus determining if any two arbitrary transformations are *commutative*, *i.e.* if they can be composed in either order and produce the same results no matter what the input model. The main contribution of the paper is to propose a *static analysis* of the transformations that relies on the following intuition: a transformation reading some instances of a metaclass is in conflict with another transformation creating, deleting or modifying instances of the same metaclass.

The paper is organized as follows. Section 2 presents a motivating example. Section 3 provides a set of formal definitions, in Alloy, that allow us to define what constitutes a valid chain of transformations, and thereafter derive associated chaining constraints. Section 4 details how the checking of the chaining constraints can be implemented. Section 5 illustrates the advantages of the proposed static analysis. Section 6 describes related work on transformation composition and reusability. Section 7 draws some conclusions and additional perspectives.

2. MOTIVATING EXAMPLE

Consider a scenario where we want to automatically generate an information system from a UML model. The generated system should satisfy established practices like adopting the 3-tier Model View Controller architecture. We have some reusable transformations available, including: a transformation T_1 applying the MVC framework and a transformation T_2 introducing an identifier to each class. Concretely, transformation T_1 handles UML concepts and introduces *View*

and *Control* and their relationships with *Classes* from UML in the output model, while transformation T_2 adds an identifier property to each *Class*. Building a valid chain from these two transformations is not so simple and the task becomes harder when the number of transformations increases.

Intuitively, two transformations can be composed in sequence if the concepts used by the second transformation are not deleted by the first one. Applying this constraint in either order establishes if the two transformations can be inverted. These chaining constraints have been identified and formally defined in [5]. Thus, checking the combination in both orders, according to this approach, is equivalent to showing that the difference between, the intersection of the input metamodels of both transformations and the intersection of the output metamodels, is empty. More formally, if $(In_{T_1} \cap In_{T_2}) \setminus (Out_{T_1} \cap Out_{T_2}) = \varnothing$ then T_1 and T_2 can be chained in both order (with In_{T_1} (resp. In_{T_2}) the input metamodel of T_1 (resp. T_2) and Out_{T_1} (resp. Out_{T_2}) its output one). On the motivating example, we lead to the conclusion that T_1 and T_2 can be chained in both orders; however, the chaining constraints do not take into account whether the transformations are conflicting and thus commutative *i.e.* that the chaining T_1 and T_2 or T_2 and T_1 leads to the same results. In our scenario, it is clear that T_1 and T_2 are not commutative. In applying T_1 then T_2, all the classes (including the *Control*) will have an ID, whereas in the opposite case, only the classes initially designed. The two transformations are conflicting. The resulting system will be different. The two orders are valid, but only one leads to the expected result. The purpose of this paper is to provide constraints to automatically prevent the chain designer that a verification of the chaining order is required since the opposite order is valid but does not lead to the same result.

3. CHAINING OF TRANSFORMATIONS BASED ON STATIC ANALYSIS

In this section, we present a static analysis based on formal definitions that allow us to determine whether a chain of transformations is valid. The definitions are given in Alloy, a lightweight specification language based on first order logic [6]. Our definitions are based on those in [9], which consider and encode a subset of MOF [12].

3.1 Formalization of the concept of a model transformation

In [9], the authors consider models to be a finite set of model elements where each model element is typed by a metaclass and can own values for properties and references (complex concepts such as opposite references, derived property and subset values are not supported). In this section, we also use the syntactic domains defined in [9] and represented as sets of atoms: ME for model elements, MC for metaclasses, P for properties, R for references and V for property values. Listing 1 presents the Alloy signature of the considered domains. In Alloy, a set of atoms is specified by a signature.

Listing 1 alludes to the Alloy specification of models proposed in [9] where `me` represents the set of model elements owned by the model (the keyword `set` means more than one), `class` the function that assigns one metaclass to each element, `valueP` the function that assigns one value to each property owned by model elements and `valueR` the function that assigns reference values. Based on these definitions, we define a metamodel (Listing 1) as a set of metaclasses `mc`, a set of properties `p` and a set of references `r`.

```
1  sig ME {} //Model Elmt
2  sig MC {} //Metaclass
3  sig R{} //Reference
4  sig P {} //Property
5  sig V {} //Value
6  sig Model {
7   me: set ME,
8   class: me -> one MC,
9   valueP: me ->P ->V,
10  valueR: me ->R ->me,
11 }
12 sig MetaModel {
13  mc: set MC,
14  p: set P,
15  r:set R,
16 }
```

Listing 1: Signature of Model and Metamodel.

Concerning the conformance relationship between model and metamodel, Listing 2 gives only the signature of the Alloy predicate. The body can be replaced by any definition provided in literature, such as the one specified in [13].

```
1  pred conformsTo[m:Model, mm:MetaModel] {}
```

Listing 2: Conform Relationship.

3.2 Introduction of an intermediary representation

A transformation is defined in Listing 3 with its single input metamodel (`mmin`) and single output metamodel (`mmout`) (lines 2 and 3). To use alloy as a SAT solver, we consider that the execution of a transformation corresponds to the production of a single output model from one single input model (line 6). To that extent, we have specified within the transformation all of its input models (`min`) (line 4) and all of its output one (`mout`) (line 5); each input (resp. output) model conforming the input (resp. output) metamodel (line 9) (resp. (line 10)).

```
1  sig Transformation {
2   mmin: one MetaModel,
3   mmout: one MetaModel,
4   min: set Model,
5   mout: set Model,
6   execution: min one -> one mout,
7   prop: MC -> TProp
8  }{
9   all mi:min | conformsTo[mi,mmin]
10  all mo:mout | conformsTo[mo,mmout]
11  (prop).TImpacted in mmout.mc
12  (prop).TRead + TImpacted in mmin.mc
13 }
```

Listing 3: Transformation.

Furthermore, we consider that the execution of a transformation might either have an impact (*i.e.*, create, or modify, or delete), or read some elements of the input or output models. From these considerations, we introduce an intermediary representation of model transformation relying on the following properties: `TImpacted`, `TCreated`, `TDeleted`, `TModified` and `TRead`. Listing 4 introduces the corresponding domains that extend the abstract domain `TProp`.

```
1  abstract sig TProp {}
2  sig TImpacted, TCreated, TModified, TDeleted, TRead
       extends TProp {}
```

Listing 4: Properties associated to transformation.

To each transformation is associated a set of properties (line 7 of Listing 3) specifying for which metaclasses, model elements can be impacted, (*i.e.*, created, modified or deleted) or read. The metaclasses whose elements can be impacted are included in the set of the output metamodel metaclasses (line 11), whereas the metaclasses whose elements can be read, or impacted are included in the set of the input metamodel metaclasses (line 12). From these two lines it can be noticed that some metaclasses are present both in the input and the output metamodel.

Let us now precisely specify to what these properties correspond. A metaclass belongs to the `TCreated` set, if there exists some execution of the transformation that creates a model element instance of this metaclass. The TCreated property is specified by the TransformationCreatedCompliant predicate (Listing 5 lines 11 to 19) that uses the predicates `ModelElementInstanceOfMcHasBeenCreated`, which defines the meaning of instance creation.

```
1  //An instance has been created (Used to define TCreated)
2  pred ModelElementInstanceOfMcHasBeenCreated[mi:Model , mo
       :Model , mc:MC] {
3   some melt:mo.me {
4    melt.(mo.class) = mc
5    melt !in mi.me
6   }
7  }
8
9  //TCreated
10 pred TransformationCreatedCompliant[t:Transformation] {
11     all cmc: (t.prop).TCreated {
12         some mi,mo: Model {
13             (mi->mo) in t.execution
14             ModelElementInstanceOfMcHasBeenCreated[mi,mo,
                   cmc]
15         }
16     }
17 }
18
19 pred TransformationCreatedConstrained[t:Transformation] {
20     some (t.prop).TCreated
21 }
```

Listing 5: Details of the TCreated property.

A metaclass `mc` is considered read only if each element of the input model whose `mc` is the metaclass, is unchanged (Listing 6 lines 11 to 15). A model element is unchanged if it belongs to the input and the output models and if its metaclass, its property values and its reference values are the same in both models (lines 2 to 8).

```
1  //The model element remains unchanged (Used to define
       TRead)
2  pred ModelElementIsUnchanged[mi:Model, mo:Model,melt:ME]
       {
3   melt in mi.me
4   melt in mo.me
5   melt.(mi.class) = melt.(mo.class)
6   melt.(mi.valueP) = melt.(mo.valueP)
7   melt.(mi.valueR) = melt.(mo.valueR)
8  }
9
10 //All instances are unchanged (Used to define TRead)
11 pred ModelElementAreReadOnly[mi:Model, mo:Model , mc:MC]
       {
12  all melt:mi.me {
13   melt.(mi.class) = mc implies ModelElementIsUnchanged[mi,
         mo,melt]
14  }
15 }
```

Listing 6: Definition of the used predicates.

The other properties are similarly built[1]. They use predicates that define how elements are either deleted or modified. An instance of a metaclass `mc` has been deleted by the execution of the transformation if there exists, in the input model, some element whose metaclass is `mc` that belongs to the input model and not to the output model. A metaclass `mc` is considered modified if it exists at least one element of the input model whose `mc` is the metaclass that is changed *i.e.*, its values or references in the input model are different from those in the output model.

3.3 Chaining Properties

Before specifying chaining constraints, let us review two definitions: chaining and commutativity (Listing 7). Two transformations `t1` and `t2` can be chained if the set of `t1` output models is included in the set of `t2` input models (lines 2 to 4). Two transformations `t1` and `t2` are commutative if `t1` can be chained with `t2`, `t2` can be chained with `t1` and if each execution of the chain in one order or the other leads to the same result (lines 6 to 10).

```
1  //Chaining
2  pred ChainingTransformation[t1,t2:Transformation] {
3   t1.mout in t2.min
4  }
5
6  //Commutativity
7  pred Commutativy[t1,t2:Transformation] {
8   ChainingTransformation[t1,t2]
9   ChainingTransformation[t2,t1]
10  all m:t1.min+t2.min | t2.execution.(t1.execution.m) = t1
        .execution.(t2.execution.m)
11 }
```

Listing 7: Chaining and Commutativity properties.

Thanks to Alloy and its SAT solver, we can state that two transformations `t1` and `t2` can be chained in either order, and lead to the same results on any input model, if the elements read or modified by one transformation are not impacted by the other, and if neither of the two transformations is the identity. Indeed, Alloy cannot find any example where the `CommutativeTransformation` predicate is true *i.e.*, when some elements of the metamodels involved in `t1` and in `t2` are read or modified by one of the transformation and impacted by the other (line 7 of Listing 8); and none of the two transformations is the identity; and `t1` and `t2` are commutative.

```
1  pred NonEmptyIntersect[t1,t2:Transformation] {
2   some ((t1.prop).TRead + (t1.prop).TModified )& ((t2.prop
        ).TImpacted)
3  }
4
5  pred CommutativeTransformation{
6   some t1,t2:Transformation {
7    NonEmptyIntersect[t1,t2] || NonEmptyIntersect[t2,t1]
8    TransformationCompliant[t1]
9    TransformationCompliant[t2]
10   IsNotIdentity[t1]
11   IsNotIdentity[t2]
12   Commutativy[t1,t2]
13  }
14 }
```

Listing 8: Commutative Transformations checking properties.

[1]The complete code can be found at http://www.lifl.fr/∼etien/commutativity

The static analysis based on Alloy leads to the following property (its symmetric is obviously true, $t1$ and $t2$ could be intertwined):

Property 1 $(TRead_{t1} \cup TModified_{t1}) \cap TImpacted_{t2} \neq \emptyset \Leftrightarrow$ *t1 and t2 are not commutable.*

Consider two transformations $t1$ and $t2$ where one or more metaclasses of the input metamodel also belong to the output metamodel of the transformation. If the intersection of the concepts read or modified by one transformation and the concepts impacted (*i.e.*, whose some instances are created, deleted or modified) by the other is not empty, the two transformations are not commutable. In the other case, they are considered commutable.

4. IMPLEMENTATION

Alloy and its SAT solver enables us to formally define the commutative property. However, transformations are typically written in task-specific transformation languages such as QVTo or ATL, not Alloy, and so adaptations have to be performed. There are at least two alternatives: either encode the transformations in Alloy or adapt the formal static analysis to existing transformation languages. We chose the second alternative in order to preserve the transformations in the language used by the transformation developers. For this purpose, we have defined a metamodel corresponding to the intermediary representation and relying on the properties defined on Listings 5 and 6 in Alloy. The example transformations that we make use of in this paper are written in QVTo. In this section, we illustrate our static analysis using a transformation chain written in QVTo, to show the relationship between the intermediary representation and the languages that chain designers would apply.

4.1 Principle

Figure 1 sketches the approach. The intermediary representation is produced from a model representation [2] of the transformation to analyse. This step is itself implemented by a model transformation. Because it effectively takes a transformation as input, this is called a higher-order transformation (HOT) [1]. Obviously, the choice of the language used to define this HOT is independent of the languages used to write the transformations that are to be chained.

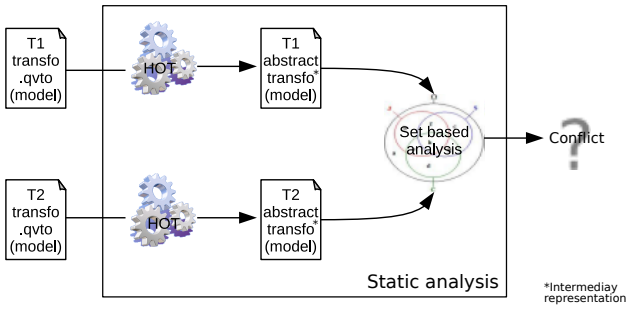

Figure 1: Sketch of the approach

[2]Most transformation tools such as QVTo or ATL provide such a representation.

The static analysis we have presented is applied to models in the automatically generated intermediary representation. Following the analysis returns, a flag is raised to the chain designer. The chain designer can then verify if a chosen order is the one desired.

4.2 Creating the intermediary representation from a transformation

In the following paragraphs, we detail the way the `TCreated`, and `TRead` elements are produced from a QVTo transformation following the Alloy specifications of Section 3. The `TDeleted` and `TModified` are built similarly.

4.2.1 *Creating elements*

Three different ways are commonly used to create elements in the output model through a QVTo transformation: the *mapping* operation, the *constructor* operation and the *object* keyword. Each of these operations create a new instance (except *inout mapping* operation). This clearly respects the definition of `TCreated` and of the predicate `ModelElementInstanceOfMcHasBeenCreated` specified in Section 3. Thus, for each *mapping*, *constructor* operation or *object* keyword contained in the transformation, a `TCreated` element referring the output metaclass is created.

4.2.2 *Reading elements*

In QVTo, read an instance is to access to it. Therefore, each time an existing instance is accessed and not modified (in order to conform to the `TRead` definition), a `TRead` referring to the involved metaclass is created. An instance can be accessed from different ways such as: from a *mapping* operation input, from successive navigations, from *mapping* operation body, *etc.*

4.2.3 *Deleting and modifying elements*

In QVTo, only the *removeElement* operation removes model elements instance of a given metaclass, from a model. Thus, for each *removeElement* operation, a `TDeleted` element, referring to the metaclass of the deleted element, is created. In QVTo, the *inout mapping* operation and the *inout* operation parameter may alter a property value or a reference of an element existing in the input and output models. Consequently, for each *inout mapping* or each *inout* parameter, a `TModified` element referring to the involved metaclass is created.

4.3 Set-based analysis

The set-based analysis checks if the elements read or modified by a transformation may be impacted (*i.e.* created, modified or deleted) by a second transformation. It corresponds to the implementation of the Alloy `NonEmptyIntersect` predicate and is concretely implemented through a QVTo transformation composed of a single query that checks the inclusion and returns a Boolean. This transformation also manages type inheritance. If in a metamodel, a metaclass X inherits from another metaclass Y and if an instance of X is created/modified/deleted by the transformation, then it also implicitly means than an instance of Y has been created/modified/deleted.

Similarly to what was done in Alloy in the previous section, this query is called twice. If one of the two intersections is

not empty, the return value is `true` meaning that the two transformations are not commutative and thus conflicting. A flag is raised; the chain designer must verify that the chosen order is the one expected since we are in the case where both orders are valid (result from the type based analysis as described in [5]). In the opposite, if the returned value is `false` for both query calls, then whatever the models, the execution of t_1 then t_2 or t_2 then t_1 will lead to the same results. Until now the returned value (`true` or `false`) is printed in a log and is manually handled.

5. EVALUATION

In order to evaluate the efficiency of the proposed analysis (which we call TAction), we compared it with the input-output metamodels based analysis presented in [5].

For the experimentation, we considered three transformations sets. The first set is composed of 6 toy transformations, which correspond to elementary test cases. The second set is composed of 22 real transformations which are used to generate information systems. The 30 transformations of the third set also aim to generate information systems. The latter two sets have been provided by two different partner organisations, whereas the toy transformations were produced by the authors. For each set, we computed the number of commutable transformation pairs using the input-output metamodels based analysis of [5] and our TAction-based analysis.

Set	#Transfo.	Pair	in.out. MM	TActions
1	6	21	13 (61.90%)	6 (38.06%)
2	22	253	33 (13.04%)	26 (10.27%)
3	30	465	45 (9.68%)	40 (8.60%)

Table 1: Commutativity results

Table 1 summarises, for each set, the number of transformations it contains; the number of transformation pairs and the number of commutable pairs identified for each of the two anslysis. For example, the second set can contain up to 253 commutable pairs. The input-output metamodels based analysis identified 33 pairs whereas our analysis only 26. For each set, the difference between the two analyses is very small (ranging from seven to five pairs).

We developed the TAction-based analysis to reduce *false positives* (*i.e.* pairs erroneously identified as commutative by the input-output metamodels based analysis). A deeper study of the results presented in Table 1 is required. The pairs considered commutable by the type based analysis were compared to the ones highlighted by the analysis proposed in this paper. Three cases occur: some pairs are identified by both analysis; some pairs only by the input-output metamodels based analysis (*false positive*), some pairs by only the TAction-based analysis (*false negative*). Some pairs are considered non commutative by the input-output metamodels based analysis, whereas the TAction-based analysis marked them as commutative. They result from the fact that the input-output metamodels based analysis takes into account the whole input and output metamodels associated to a transformation even if only a subset of concepts is used. The *false negatives* disappear when the metamodels only contain metaclasses involved in the transformation (*i.e.* read or impacted). Furthermore, because of the over approximation due to the TAction-based analysis (*e.g.* dead code), and the used level of detail (*i.e.* based on metaclasses rather than properties) doubts persist concerning the results. In our examples, manual checking allows us to ensure that the obtained results are correct.

Set	Common	False positive	False negative
1	5 (38.46%)	8 (61.53%)	1 (7.69%)
2	23 (69.69%)	10 (30.30%)	3 (9.09%)
3	30 (66.66%)	15 (33.33%)	10 (22.22%)

Table 2: Numbers of errors and approximations using TAction-based analysis

Table 2 gathers the figures for each set computed by the TAction-based analysis. Once again, set 1 has been developed for test purposes and the resulting figures are not trustworthy. In contrast, in the two other sets, around 30% of the commutative pairs identified by the input-output metamodels based analysis are in fact not commutative (false positive). Furthermore, some pairs in these two cases were not considered as commutative whereas they are. The difference observed between set 2 and set 3, concerning false negative, arises from the adequacy of the metamodels to the transformations: the input-output metamodels based analysis uses metamodels with significant irrelevant concepts for the transformation. This evaluation highlights the added-value of our analysis based on a TAction-based analysis to check the commutativity between transformations: we can produce fewer false negatives.

6. RELATED WORK

Several authors promote the decomposition of transformation into smaller ones in order to enhance their reusability and their maintainability. The way the chains are built from these small transformations varies according to the proposed approaches. Different operators exist: chaining [10, 11], composition [16], that can eventually be conditional or parallel [15], and loop [15]. Mostly such operators can be applied when transformations are traditional heterogeneous ones and the chaining conditions relative to the inclusion of the output and input metamodels also [10, 11, 15]. In [16], the transformations can be composed to create a new one but only if they have the same input metamodel; they may lead to a completely different output metamodel. In some approaches, for each small transformation, the required and provided concepts are explicitly manually identified by the chain designer using a profile [19] or automatically identified based on the distinction between concepts copied and those mutated [2] or more generally using critical pair analysis [7, 4, 8]. However, in these latter cases, they only deal with endogenous transformations (even if they suggest that an extension to heterogeneous transformations is possible) and do not care about commutativity.

In [7] and [4], the authors also use graph theory to precisely specify when two transformation rules of pure refactoring are conflicting. If the critical pair analysis is adapted to the detection of conflicts between rules, it cannot be adapted between transformations. Critical pair analysis is based on the

single pushout semantics where rules have a left-hand side, a right-hand side, and eventually a NAC. Such a semantics is not general enough to considered a whole transformation expressed in different transformation languages. Our relies on the same intuition than the one underlying the parallel and sequential independence described in [4]. Nevertheless it takes into account the transformation in its whole and not each of its rules. To that extent, it proposes the introduction of an intermediary representation to detect conflict between transformations.

In this paper we provide a static analysis based on an intermediary representation where concepts created, modified or read are distinguished. The designer has nothing to specify by hand, everything is automatically computed. Our approach does not impose any constraint on the metamodels of the transformation. It takes all its benefits when the input and output of the transformations are different but contain some metaclasses in common.

7. CONCLUSION

In this paper, we have proposed an approach to verify the chaining of model transformations. Our mechanism aims at checking if two transformations are commutative, *i.e.* if they can be chained in either order and deliver the same results no matter the input. This mechanism relaxes the constraints previously established by distinguishing the read and impacted (*i.e.* created, deleted or modified) concepts.

We formally proposed a static analysis relying on an intermediary representation independent from any transformation language. We have implemented our approach on top of QVTo in order to check the commutativity of transformations written with this language. If the two studied transformations are not commutative, a flag is raised to the chain developer in order she verifies if the chosen order is the expected one. As any other static analysis, our approach suffers from over approximations due to dead code for example. We also foresee to take into account not only the metaclasses but also the properties. With our approach two transformations modifying two different properties of the same metaclass are not currently considered as commutative whereas they can be.

As a conclusion, our investigations have been validated using QVTo but do not depend on any model transformation technique. The model transformations are handled as functions having one input and one output metamodel possibly sharing some concepts. This approach refines the existing chaining constraints and will help to design chains from existing transformations.

8. REFERENCES

[1] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! 2006.

[2] R. Chenouard and F. Jouault. Automatically discovering hidden transformation chaining constraints. In *Model Driven Engineering Languages and Systems*. 2009.

[3] J. Cordy. Eating our own dog food: Dsls for generative and transformational engineering. In *GPCE*, 2009.

[4] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. 2006.

[5] A. Etien, A. Muller, T. Legrand, and X. Blanc. Combining independent model transformations. In *Proceedings of the ACM SAC*, 2010.

[6] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[7] L. Lambers, H. Ehrig, and G. Taentzer. Sufficient criteria for applicability and non-applicability of rule sequences. *ECEASST*, 2008.

[8] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science*, Apr. 2005.

[9] A. Mougenot, X. Blanc, and M.-P. Gervais. D-praxis: A peer-to-peer collaborative model editing framework. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS '09, 2009.

[10] J. Oldevik. Transformation composition modelling framework. In *Proceedings of the Distributed Applications and Interoperable Systems Conference*, 2005.

[11] G. Olsen, J. Aagedal, and J. Oldevik. Aspects of reusable model transformations. In *Proceedings of the ECMDA Composition of Model Transformations Workshop*, 2006.

[12] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, July 2007.

[13] R. Paige, P. Brooke, and J. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, 2007.

[14] J. Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and visualizing transformation chains. In *Proceedings of the European conference on Model Driven Architecture*, 2008.

[15] J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo. Orchestrating ATL model transformations. In *Proc. of MtATL*, Nantes, France, July 2009.

[16] J. Sanchez Cuadrado and Garcia Molina. Approaches for model transformation reuse: Factorization and composition. In *Proceedings of the International Conference on Model Transformation*, 2008.

[17] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 2003.

[18] B. Vanhooff, D. Ayed, and Y. Berbers. A framework for transformation chain development processes. In *Proceedings of the ECMDA Composition of Model Transformations Workshop*, 2006.

[19] B. Vanhooff and Y. Berbers. Breaking up the transformation chain. In *Proceedings of the Best Practices for Model-Driven Software Development at OOPSLA*, 2005.

[20] D. Wagelaar, R. Van Der Straeten, and D. Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *Software and Systems Modeling*, 2009.