Orchestrating ATL Model Transformations

J.E. Rivera, D. Ruiz-González, F. López-Romero, J. Bautista, and A. Vallecillo

GISUM/Atenea Research Group. Universidad de Málaga (Spain).
{rivera,daniruiz,fernando,jmbautista,av}@lcc.uma.es

Abstract. The design and development of any complex applications using a Model-driven Engineering approach involve not only the use of many models, but also the use of many model transformations between them. These transformations should be chained together, using Model Transformation Orchestration notations and tools. This paper introduces Wires*, a graphical executable language for the orchestration of ATL transformations, which provides appropriate mechanisms to enable the modular and compositional specification and execution of complex model transformations chains.

1 Introduction

Model-Driven Engineering (MDE) advocates the use of models as the key artifacts in all phases of development, from system specification and analysis, to design and implementation. Each model usually addresses one concern, independently from the rest of the issues involved in the construction of the system. Thus, the basic functionality of the system can be separated from its final implementation; the business logic can be separated from the underlying platform technology, etc. The implementation of a system is then obtained by applying a set of transformations to the different models defined for it. These model transformations are usually interconnected, i.e., the output models of some transformations are the input models of others.

Model Transformation Orchestration (MTO) aims at supporting the construction of complex model transformations from other transformations already defined. MTO is especially relevant in the context of Global Model Management [1, 2], in which megamodels are kinds of registries for resources available from a given model-driven platform, recording all accessible entities like models, metamodels, transformations, tools, and the various relations between them. In this setting, we expect that model transformations become available off-the-shelf and that can be reused to build new tools, processes and, of course, new transformations.

Eclipse provides an ideal infrastructure for building tools to support the use of models. While there is a large selection of tools available for working with individual models and with individual transformations, there is less support for working with collections of models and with collections of model transformations. When we want to chain model transformations, we normally end up using a set of Ant tasks. Although this solution works, it does not provide a high level

specification of chains of transformations, i.e., it remains at quite a low level. Furthermore, the analysis capabilities of the Ant task specifications are limited.

In this paper we present Wires*, a Domain Specific Language that enables the high-level orchestration of model transformations. It provides a visual notation for defining chains of model transformation in a modular and compositional manner, and it is supported by a graphical framework and an execution engine that loads the appropriate models and execute the transformations along the pre-defined path.

The structure of this document is as follows. After this introduction, Section 2 presents the Wires* language. Then, Section 3 describes the execution engine that interprets Wires* models and executes the model transformations. Finally, Section 4 compares our work with other related proposals, and Section 5 draws some conclusions and outlines some future research activities.

2   The Wires* Language

Wires* is a graphical executable language for orchestrating ATL transformations. Fig. 2 shows the metamodel of the language. Basically, Wires* assumes a data-flow process, in which a set of input models (conforming to their corresponding metamodels) are processed by a chain of ATL transformations until a set of output models is produced. The chain is composed of transformations, which act as processing nodes. Parameters represent the consumed and produced data by transformations. Transformations are wired together by directed connectors (called *Dataflows* or, simply, *Wires*) that indicate how the outputs of the transformations are linked to the inputs of the next ones. Fig. 1 shows a simple example, where an input model m1 is transformed by two transformations in a row to produce an output model m3.
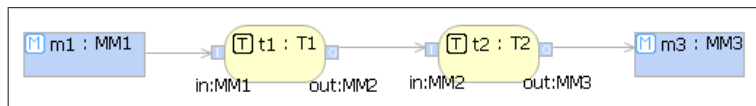
Fig. 1. A simple chain of 2 model transformations.

Although our notation may resemble at first simple UML 2 activities (transformations are represented by activity nodes, models are represented by object nodes, and wires by object flows), there is a significant difference in the semantics. Basically, Wires* is a data-flow oriented language, whilst UML 2 activities can be either data- or process-flow oriented, or both.

The following paragraphs describe the main concepts of the language with more detail.

Models. In our approach we consider that models and transformations are course-grained building blocks of the MDE process. Models are typed by the
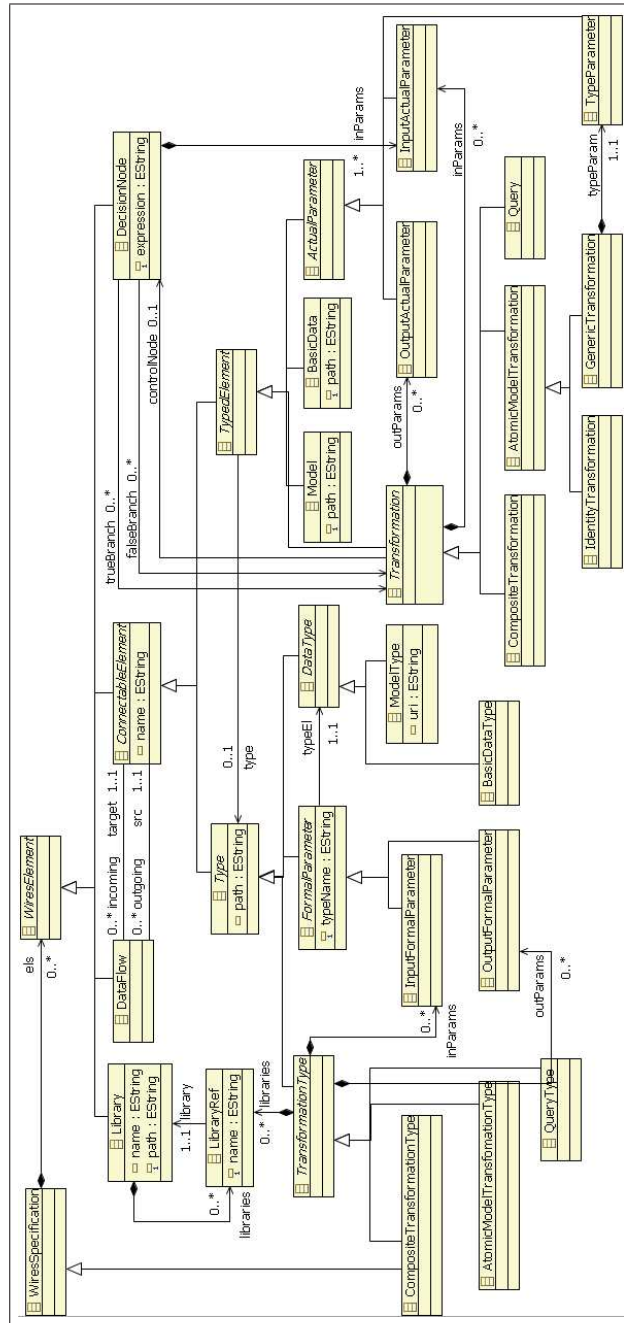
Fig. 2. The Wires* Metamodel

initial metamodel (*ModelType*) they conform to [3]. Models without incoming *Dataflows* represent the input models of the process (i.e., the models that are initially loaded), while models with incoming *Dataflows* represent output models (i.e., models that are saved). Primitive data types (*BasicDataType*) are also supported.

Transformations. We contemplate the three different kinds of units defined by ATL: modules, libraries and queries. ATL modules and queries are specified by instances of *AtomicModelTransformation* and *Query* metaclasses, respectively. They are separated from their corresponding specifications (*AtomicModelTransformationType* and *QueryType*, respectively), on which we specify the data file where they are stored (the *path* attribute), their input and output formal parameters, as well as the auxiliary *Libraries* they make use of. *AtomicModelTransformations* may have multiple input and multiple output models. *Queries* have one single output, which is a primitive type value; one common use of a *Query* is the generation of a textual output (encoded in a string value), which can be stored in a text file (using *BasicData* instances).

In addition to *Atomic* transformations, which represent basic ATL transformations, we also allow the definition of *Composite* transformations, which represent chains of transformations which can be later used as a single (atomic) transformation, i.e., as building blocks to specify further transformations. Fig. 3 shows an example of the specification of a composite model transformation, CT1, made of t1 followed by t2.
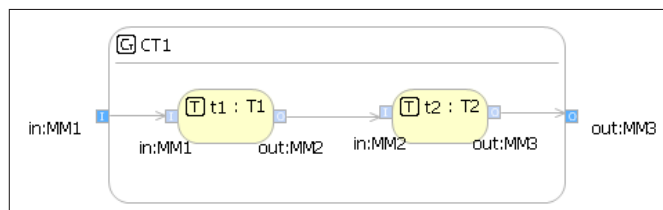
Fig. 3. The specification of a *Composite* transformation

We have also identified two special kinds of atomic transformations: *Identity* and *Generic*. Identity transformations are model transformations that do not alter the data which flow through them (they are very useful for defining conditional compositions of transformations, as we shall later see). Generic transformation are those used to load and execute ATL transformations produced by high-order transformations. They have a special input parameter (*typeParam*) that should conform to the ATL metamodel.

Fig. 4 depicts an example of the specification of a *Generic* transformation. It shows a high-order transformation HOT that produces a transformation out-Transf, which is passed as input to the generic transformation gt. Then, gt builds the ATL code corresponding to the model outTransf, compiles it and executes it

on the input model m1 to produce model m3. The special input parameter *type-Param* of a *generic* transformation is represented by another pin, of the same color as the transformation, to differentiate it from the normal input parameters of the transformation (which are colored as models).
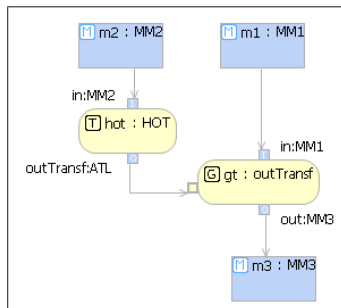

Fig. 4. The specification of a *Generic* transformation

Conditional compositions. Of course, not all chains of model transformations follow linear paths. A *DecisionNode* is an element whose input parameters are the outputs of *Queries* (i.e., primitive data types), and that contains an OCL expression that will be evaluated to decide which one of its two branches (*trueBranch* or *falseBranch*) is enabled.
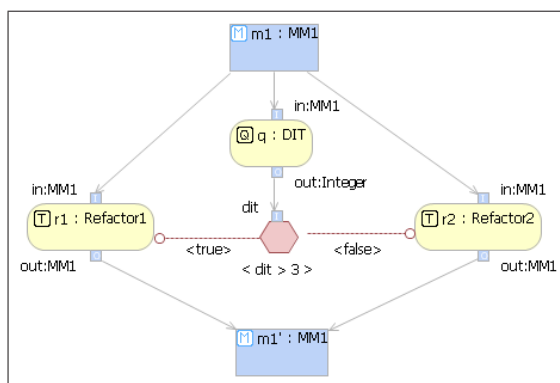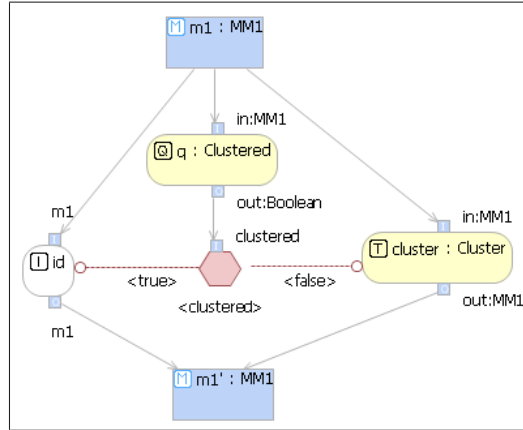

Fig. 5. A conditional composition

Fig. 5 shows an example of a conditional composition of transformations, using *Queries* and *DecisionNodes*. Assuming that we have two different refactoring algorithms (Refactor1 and Refactor2, implemented by two model trans-

formations), the composition applies one or the other to an input model m1 depending on the deep inheritance tree (implemented by *Query* DIT) of m1.

In Wires* every transformation can be connected to a *DecisionNode* using *trueBranch* or a *falseBranch*. If unconnected, the transformation will always be enabled. When connected to a *DecisionNode*, the transformation will be enabled only if the corresponding branch is activated after evaluating the *DecisionNode* expression.


Fig. 6. Another conditional composition

Fig. 6 shows another example of a conditional composition of transformations, this time using an *Identity* transformation. The diagram represents a situation in which a *Query* (q) checks whether a model that represent a component-based system is already organized in efficient clusters or not. If so, the model is left unchanged; otherwise the model is changed by applying the cluster transformation.

Parallel composition. As shown in figures 5 and 6, several *Dataflows* can come out of a *model* (m1), meaning that the model can be used in parallel as input of the corresponding transformations. In case of conditional compositions, only one of these transformations will be active. However, branches without decision nodes are also allowed, i.e., *models* can have several outgoing *Dataflows*. They enable implementing parallel composition of transformations. For example, Fig. 7 shows how a model m1 serves as input of two transformations t1 and t1' which are executed concurrently to produce models m2 and m3.

In case a model is connected as the output of two or more transformations, and they are executed in parallel, the result is undefined. Please notice that this does not happen with model m1' in the conditional composition shown in Fig. 5, because only one of the transformations r1 and r2 will be executed. The same happens to model m1' in Fig. 6, which can be produced by either the *Identity*
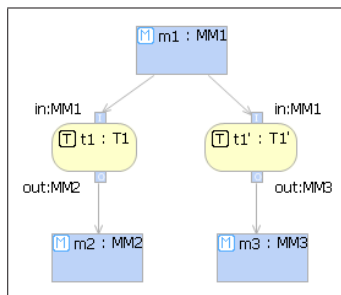
Fig. 7. A parallel composition

or the cluster transformations, but not by both during the same execution of the chain.

Loops. In Wires*, loops are simply expressed using *DecisionNodes* and *Dataflows* to connect one of their branches to the beginning of the loop. For instance, Fig. 8 shows the Wires* specification of a process that, given an endogenous model transformation b and an initial model in, iteratively applies b until it reaches a fixed point (i.e., its input model coincides with its output model). Equals is a *Query* that checks whether its two input models are equal or not, returning a Boolean value with the result.

Notice that if b is an endogenous transformation defined by a set of rules, which are used to specify the behavior of metamodel MM1 (see [4, 5]), and provided that ATL supports this kind of in-place semantics, then the process depicted in Fig. 8 implements the simulation of the behavior of the system, from an initial state defined by model in. If required, one additional model transformation could take the intermediate output models m1' and visualize them, hence enabling the visualization of the successive states of the system during the simulation, and therefore producing a motion picture show.

Temporary vs. persistent models. Finally, models can appear in a Wires* specification not only as inputs or outputs of the whole process, but also in-between. For instance, Fig. 9 is similar to Fig. 1 but storing the intermediate result in a model m2. In this way such an intermediate model is created and made persistent, while in the chain shown in Fig. 1 that model is temporarily created by the Wires* execution engine, and deleted after model m3 is created.

3 The Wires* Execution Engine

Apart from a graphical notation, Wires* is supported by an execution engine that interprets Wires* models and execute them.

Basically, the way in which the engine works is as follows. Given a Wires* model, it starts by reading and loading the input *models*, which are those with no incoming *Dataflows*. For each of these models the execution engine looks for
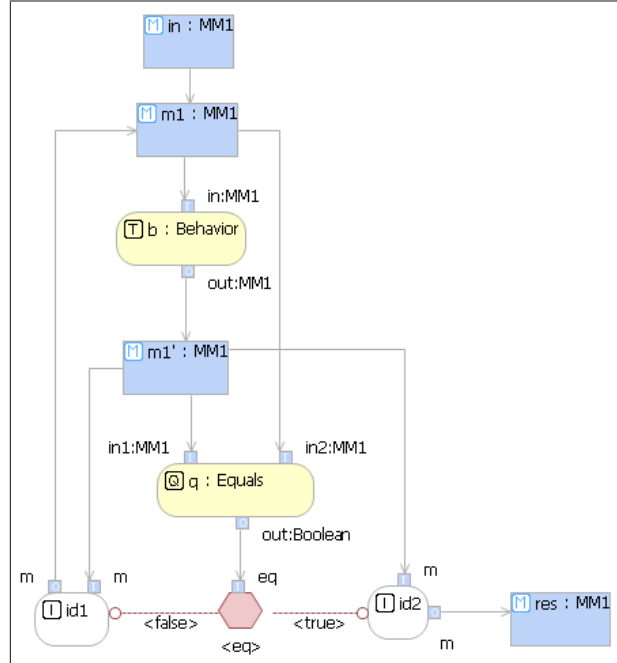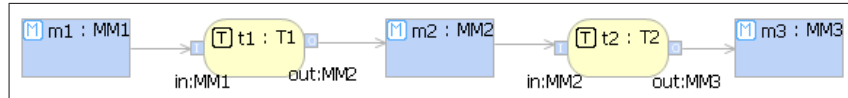
Fig. 8. A loop with Wires*

Fig. 9. A chain of transformations with persistent storage of intermediate result.

those transformations which are enabled (i.e., transformations with every input models loaded and with their branch enabled if they are connected to a decision node) and whose input models are already loaded. Once these transformations are identified, they are loaded and executed, producing the corresponding output models. The process continues until no enabled transformation is left unexecuted.

The pseudocode shown in Fig. 10 describes the structure and basic behavior of the algorithm. To illustrate this behavior, let us describe the execution of the chain depicted in Fig. 6. Firstly the execution engine identifies the models without incoming data flows (input models), in this case model m1. The algorithm determines its outgoing elements, which are the id identity transformation, the query q and the cluster transformation. Although the execution order is not pre-determined, let us suppose a left-to-right order. Then, the algorithm checks whether id is enabled, but it is not because the decision node has not activated any branch yet. Secondly, the engine tries to execute the query q, which is possible because its input model is loaded and it is enabled (not connected to any

```
1.  elements = searchInputElements();
2.  void execute(elements) {
3.    for each e in elements {
4.        targetElements = searchTargetElements(e);
5.        for each targetElement in targetElements {
6.           if ( enabled(targetElement) ) {
7.              executeOneElement(targetElement);
8.              targetOutgoings = searchTargetElements(targetElement);
9.              execute(targetOutgoings)
10.          }
11.       }
12.    }
13. }
```

Fig. 10. The Wires* engine algorithm.

control branch). Hence the query is executed and the result is obtained, storing it in a temporary variable. Since the algorithm uses an in-depth traversal of the spanning tree, the engine continues by following that path, which leads to a decision node whose expression is just the Boolean output of the query. Suppose that the value is true. Then the corresponding transformation is enabled (in this case, id) and then executed because its input model is already loaded. Being an *Identity* transformation, it just copies its input model into its output model m1'. The engine continues by exploring the last outgoing possibility of input model m1, trying to execute the cluster transformation. However, it is not possible because its incoming branch is not active. And the process ends. In case the result of the clustered query was false, the algorithm would have followed the other branch of the *DecisionNode*, executing the cluster transformation instead.

It is important to remark that the information about the actual input and output parameters of the transformations is used to build the invocation of the corresponding ATL transformations.

The current version of the execution engine is sequential, i.e., in case of possible parallel execution of transformations it executes one after the other (but the order is not pre-determined).

The tool (including the graphical editor and the execution engine) can be downloaded from `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Wires*`. Although still in alpha version, it provides all the functionality described here and is fully operative. Further extensions are described later in Section 5.

4   Related Work

This proposal was inspired by the needs of a real project for visualizing component-based systems in order to detect design anomalies. The architecture of the project is based on a set of chained ATL model transformations, which progressively extract the model of the code from the C++ files and DLLs; organize the classes
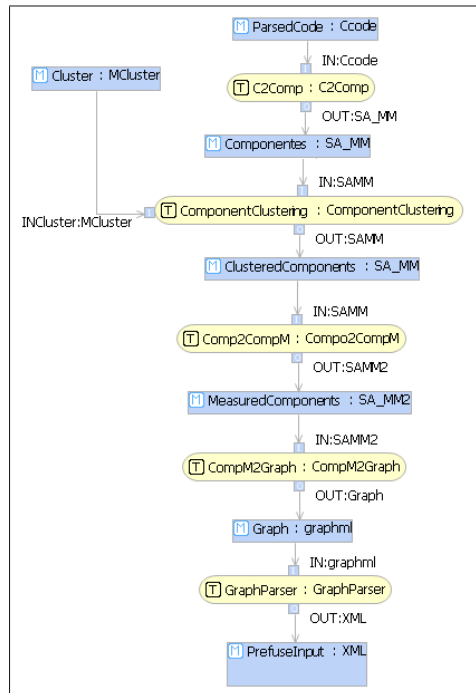
Fig. 11. Fragment of the chained transformations of the VIASCO process

into components; cluster them to build the software architecture; measure the components and their connectors; and finally visualize the system. This approach provided a very clean and modular way to design, architect and develop the tool, and is shown in Fig. 11.

There are already some approaches for chaining model transformations, some of them graphical and very powerful such as UniTI [6], other textual such as MCC [7], and others that use UML activity diagrams for specifying the flow of control (e.g. [8]). In general, they allow model transformations written in different languages to be composed to form a chained process that can be executed. The Sensoria Development Environment (SDE) [9, 10] provides a very powerful graphical notation and environment for the orchestration of software tools (and in particular of model transformations) using a MDE approach. However, when we tried to use these tools with ATL in our project we ended up using none of them, but a set of Ant tasks [11]. Basically, we wanted a quick and very simple way to chain several ATL transformations in a row. In this sense, ATLFLow [12] is a graphical editor for transformation processes with ATL. It describes the structure of a transformation flow, and it has the ability to execute both transformations and generators. However, it does not provide support for conditional branches nor for composite transformations.

The QVT language [13] can also be used to specify complex transformations. In theory it supports composition of transformations through various reuse mechanisms, such as extension or black-box reuse of transformation libraries through its access mechanism. Combined with control constructs, structured transformation compositions can be created. The graphical QVT syntax might be used for similar constructions, although the current graphical notation is more apt for specifying relations between the concepts of a single transformation, than for composing transformations. As mentioned in [8], a UML2-oriented approach with composite activities (or even composite structures) can leverage specification of higher-order transformations. However, we found no readily available tool for supporting what we pursued in a simple way.

Other sources of inspiration for our work include those from the CBSD and Software Architecture communities with their ADLs, the Workflow community (that uses UML activity diagrams and other process-based notations such as BPMN or SPEM), and the SOA community, with orchestration languages such as BPEL, for instance. Again, they all provide notations and tools very useful within their contexts, but not directly applicable to our concrete problem.

In this sense, our approach is much more modest than those previously mentioned. It focuses just on ATL transformations, and provides a notation and an execution engine for orchestrating this kind of model transformations. On the other hand, being specific means that it realizes some concrete concepts and mechanisms which are particular of this context, e.g., naturally dealing with high-order transformations.

Here we have focused on the composition of complete transformations. Other works discuss the composition of model transformations at a lower level, i.e., at the level of their constituent rules [14–16]. In general, it is difficult to make a meaningful comparison between both approaches, and it is not clear whether they can be easily combined or not. In any case, this issue falls outside the scope of this paper. For a more detailed evaluation of the composition possibilities offered by rule-based transformation languages we refer the reader to [14].

5 Conclusions and Future Work

This paper has presented Wires*, a graphical executable language for the orchestration of ATL transformations, which provides appropriate mechanisms to enable the modular and compositional construction of complex model transformations chains. The language is supported by an execution engine that provides not only a proof-of-concept for the approach, but also a tool that effectively realizes the orchestration of the execution of the model transformations that compose a given chain.

There are many different lines of work that we plan to explore now that we have a tool for orchestrating ATL transformations. Firstly, we want to define and use a type system that allows the static validation of the Wires* models, for instance that the *Dataflows* connect output models with valid input models. In other words, we would like to be able to check the type substitutability between

the output of a transformation and the input of another, in such a way that type safety is guaranteed. Thus, we are currently adding a metamodel subtyping algorithm [3, 17] to our execution engine.

Secondly, we also want to connect our tool with other kinds of model transformations, such as model injectors and extractors (defined using, e.g., TCS).

Finally, as future extensions we'd also like to explore the possibility of integrating and executing model transformations defined in other languages, as well as incorporating new ATL features as they are released.

References

1. Allilaire, F., Bézivin, J., Brunelière, H., Jouault, F.: Global Model Management in Eclipse GMT/AM3. In: Proc. of the Eclipse Technology eXchange (eTX) workshop at ECOOP 2006, Nantes, France (2006) `http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/AM3/`.
2. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In Aßmann, U., Aksit, M., Rensink, A., eds.: Model Driven Architecture, European MDA Workshops: Foundations and Applications (MDA-FA 2003/2004). Volume 3599 of LNCS., Springer (2005) 33–46
3. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and tool support for model driven engineering with Maude. Journal of Object Technology **6**(9) (2007) 187–207
4. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In: Proc. of the first International Conference on Software Language Engineering (SLE'08). LNCS, Toulouse, France, Springer (2008)
5. Rivera, J.E., Vicente-Chicote, C., Vallecillo, A.: Extending visual modeling languages with timed behavioral specifications. In: Proc. of IDEAS 2009, Medellín, Colombia (2009)
6. Vanhooff, B., Ayed, D., Baelen, S.V., Joosen, W., Berbers, Y.: UniTI: A Unified Transformation Infrastructure. In: Proc. of MoDELS 2007. Number 4735 in LNCS, Springer-Verlag (2007) 31–45
7. Kleppe, A.: MCC: A Model Transformation Environment. In: Proc. of ECMDA-FA 2006. Number 4066 in LNCS, Springer-Verlag (2006) 173–187
8. Oldevik, J.: Transformation composition modelling framework. In: Proc. of DAIS 2005. Number 3543 in LNCS, Springer-Verlag (2005) 108–114
9. Foster, H., Mayer, P.: Leveraging Integrated Tools for Model-Based Analysis of Service Compositions. In: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services (ICIW 2008), IEEE Computer Society (2008) 72–77
10. Mayer, P., Ráth, I., Horváth, A.: Report on the Sensoria Development Environment (second version). Project deliverable D7.4.c, SENSORIA EU-IST Project (2008) `http://www.pst.ifi.lmu.de/projekte/Sensoria/del_36/D7.4.c.pdf`.
11. ANT: The Apache Ant Project (2008) `http://ant.apache.org/`.
12. Zeidler, U.: (ATLFlow) `http://opensource.urszeidler.de/ATLflow/`.

13. OMG: MOF QVT Final Adopted Specification. Object Management Group. (2005) OMG doc. ptc/05-11-01.
14. Kurtev, I., van den Berg, K., Jouault, F.: Evaluation of rule-based modularization in model transformation languages illustrated with ATL. In: Proc. of the Model Tranformation track at ACM SAC 2006 (MT 2006), ACM Press (2006) 1202–1209
15. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In: Proc. of ICMT 2008. Number 5063 in LNCS, Springer-Verlag (2008) 152–167
16. Sánchez-Cuadrado, J., García-Molina, J.: Approaches for model transformation reuse: Factorization and composition. In: Proc. of ICMT 2008. Number 5063 in LNCS, Springer-Verlag (2008) 168–182
17. Steel, J., Jézéquel, J.M.: On model typing. Software and Systems Modeling **6**(4) (2007) 401–413