DSLTrans: A Turing Incomplete Transformation Language

Bruno Barroca[†], Levi Lúcio[‡], Vasco Amaral[†], Roberto Félix[†], and Vasco Sousa[†]

[†]CITI, Departamento de Informática, Faculdade de Ciencias e Tecnologia
Universidade Nova de Lisboa, Portugal
[‡]LASSY, University of Luxembourg, Luxembourg
Levi.Lucio@uni.lu
{Bruno.Barroca,Vasco.Amaral,Roberto.Felix,Vasco.Sousa}@di.fct.unl.pt

Abstract. In this paper we present DSLTrans: a visual language and a tool for model transformations [1]. We aim at tackling a couple of important challenges in model transformation languages — transformation termination and confluence. The contribution of this paper is the proposition of a transformation language where all possible transformations are guaranteed to be terminating and confluent by construction. The resulting transformation language is simple, turing incomplete and includes transformation abstractions to support transformations in a software language engineering context. Our explanation of DSLTrans includes a complete formal description of our visual language and its properties.

Keywords: Model Transformations, Turing Incompleteness, Termination, Confluence.

1 Introduction

A problem in modern model transformation languages that has recently received some attention is to how to guarantee that a transformation terminates. Because the semantics of transformation languages are usually based on graph grammars, the termination problem is in general undecidable [7]. Termination has been described in [6] as one of the '*quality requirements for a transformation language or tool*'. The problem has been approached by several authors [2–4] who have proposed criteria that can be applied to decide about the termination of transformations under particular conditions. The EMFTrans tool [1] presents a complete formalization of all concepts involved in a transformation and it is possible under certain conditions to decide if a given transformation is locally confluent and terminates.

In this paper we propose an 'egg of Columbus' approach to the termination problem by building a visual transformation language called DSLTrans which

[1] This work has been developed in the context of project BATIC3S partially funded by the Portuguese FCT/MCTES ref. PTDC/EIA/65798/2006, the doctoral grant ref. SFRH/BD/38123/2007, the post doctoral grant ref. SFRH/BPD/65394/2009 and the Luxembourguese FNR/CORE project MOVERE ref. C09/IS/02

guarantees that the number of steps in a model transformation is always finite. As a consequence, any transformation expressed in our language will always end. We also guarantee by construction the confluence of any model transformation written in DSLTrans, which is an important correctness property of model transformations as mentioned in [3]. DSLTrans is, by construction, a turing incomplete language. This is due to the fact that our language is free of loop or recursion constructs. The work presented in this paper provides the basis for the work we present in [5], where a technique for proving properties of the type '*if a structural relation between some elements of the source model holds, then another structural relation between some elements of the target model should also hold*' is presented. By proposing such a technique we are able to provide additional '*success criteria for a transformation language or tool*' [6], which is the ability to verify our model transformations.

The rest of this paper is organized as follows. In section 2, we informally describe the syntax and semantics of DSLTrans; In section 3 we describe the mathematical underpinnings of our transformation language; Section 4 concludes.

2 Language Overview

Let us present the DSLTrans running example we will use throughout this paper.

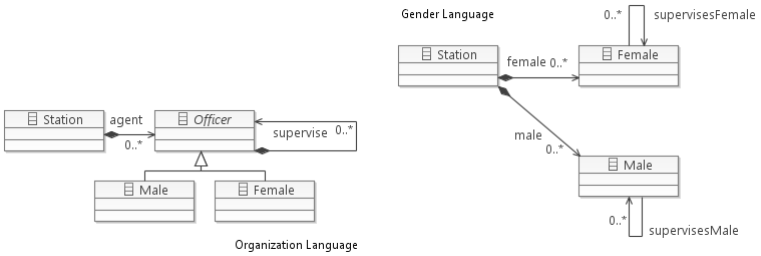

Fig. 1. Metamodels of a squad of agents(left) and a squad organized by gender(right).

Figure 1 presents two metamodels of languages for describing views over the organization of a police station. The metamodel annotated with 'Organization Language' represents a language for describing the chain of command in a police station, which includes male (*Male* class) and female officers (*Female* class). The metamodel annotated with 'Gender Language' represents a language for describing a different view over the chain of command, where the officers working at the police station are classified by gender. In figure 2 we present a transformation written in DSLTrans[2] between models of both languages. The purpose of

[2] DSLTrans was deployed as an Eclipse plug-in [8]. The example shown in figure 2, was expressed using a concrete visual syntax of an Eclipse diagrammatic editor.

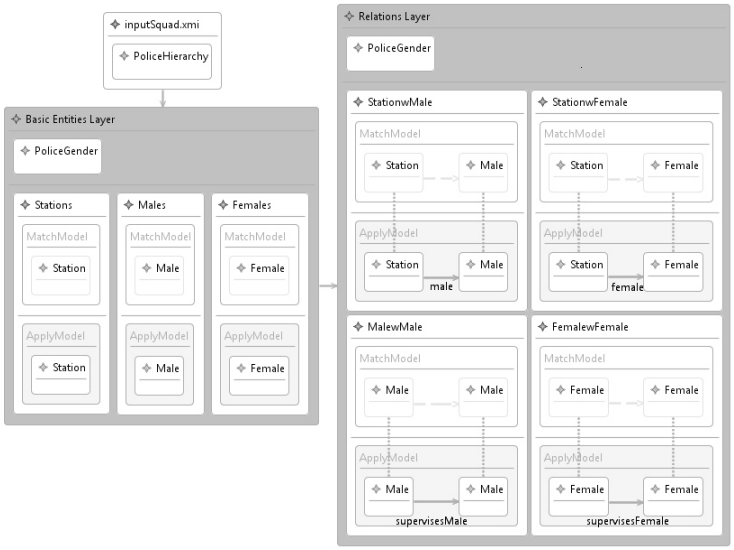

Fig. 2. A model transformation expressed in DSLTrans.

this transformation is to flatten a chain of command given in language 'Organization Language' into two independent sets of male and female officers. Within each of those sets the command relations are kept, i.e. a female officer will be directly related to all her female subordinates and likewise for male officers.

An example of an instance of this transformation can be observed in figure 3, where the original model is on the left and the transformed one on the right. Notice that the elements s, m_k and f_k in the figure on the left are instances of the source metamodel elements $Station$, $Male$ and $Female$ respectively (in figure 1). The primed elements in the figure on the right are their instance counterparts in the target metamodel.

A transformation in DSLTrans is formed by a set of input model sources called *file-ports* ('inputSquad.xmi' in figure 2) and a list of *layers* ('Basic entities' and 'Relations' layers in figure 2). Both layers and file-ports are typed according to metamodels. DSLTrans executes sequentially the list of layers of a transformation specification. A layer is a set of transformation rules, which executes in a non-deterministic fashion. Each transformation rule is a pair $(match, apply)$ where *match* is a pattern holding elements from the source metamodel, and *apply* is a pattern holding elements of the target metamodel. For example, in the transformation rule 'Stations' in the 'Basic entities' layer (in figure 2) the *match* pattern holds one 'Station' class from the 'Squad Organization Language' metamodel — the source metamodel; the *apply* pattern holds one 'Station' class from the 'Squad Gender Language' metamodel — the target metamodel. This means that all elements in the input source which are of type 'Station' of the source metamodel will be transformed into elements of type 'Station' of the target metamodel.

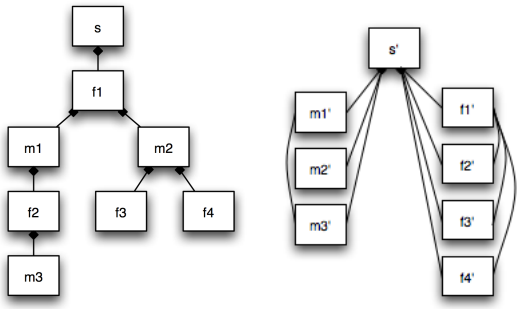

Fig. 3. Original model (left) and transformed model (right).

Let us first define the constructs available for building transformation rules' match patterns. We will illustrate the constructs by referring to the transformation in figure 2.

- *Match Elements*: are variables typed by elements of the source metamodel which can assume as values elements of that type (or subtype) in the input model. In our example, a match element is the 'Station' element in the 'Stations' transformation rule of layer 'Basic Entities' layer;
- *Attribute Conditions*: conditions over the attributes of a *match* element;
- *Direct Match Links*: are variables typed by labelled relations of the source metamodel. These variables can assume as values relations having the same label in the input model. A direct match link is always expressed between two match elements;
- *Indirect Match Links*: indirect match links are similar to direct match links, but there may exist a path of containment associations between the matched instances[3]. In our example, indirect match links are represented in all the transformation rules of layer 'Relations' as dashed arrows between elements of the match models;
- *Backward Links*: backward links connect elements of the match and the apply models. They exist in our example in all transformation rules in the 'Relations' layer, depicted as dashed vertical lines. Backward links are used to refer to elements created in a previous layer in order to use them in the current one. An important characteristic of DSLTrans is that throughout all the layers the source model remains intact as a match source. Therefore, the only possibility to reuse elements created from a previous layer is to reference them using backward links;
- *Negative Conditions*: it is possible to express negative conditions over match elements, backward, direct and indirect match links.

The constructs for building transformation rules' apply patterns are:

[3] In the implementation the notion of indirect links only captures EMF containment associations in order to avoid cycles.

– *Apply Elements and Apply Links*: apply elements, as match elements, are variables typed by elements of the source metamodel. Apply elements in a given transformation rule that are not connected to backward links will create elements of the same type in the transformation output. A similar mechanism is used for apply links. These output elements and links will be created as many times as the match model of the transformation rule is instantiated in the input model. In our example, the 'StationwMale' transformation rule of layer 'Relations Layer' takes instances of *Station* and $Male$ (of the 'Gender Language' metamodel) which were created in a previous layer from instances of *Station* and $Male$ (of the 'Organization Language' metamodel), and connects them using a 'male' relation;
– *Apply Attributes*: DSLTrans includes a small attribute language allowing the composition of attributes of apply model elements from references to one or more match model element attributes.

3 Formal Syntax and Semantics

In this section we build a formal definition of DSLTrans in order to provide a clear specification of our language and a basis for studying and proving properties about it. In the mathematical theory we disregard the formalization of: class attributes; negative conditions; class inheritance at the metamodel level. We present a light formalization of the relations at the metamodel and model levels which deals only with the difference between reference and containment relations between classes. These non formalized — but implemented in [8] — features of the language do not affect the termination or confluence properties of our language.

3.1 Transformation Language Syntax

Definition 1. *Typed Graph and Indirect Typed Graph*

A typed graph is a triple $\langle V, E, \tau \rangle$ *where* V *is a finite set of vertices,* $E \subseteq V \times V$ *is a finite set of directed edges connecting the vertices and* $\tau : \{V \cup E\} \rightarrow Type \cup \{containment, reference\}$ *is a typing function for the elements of* V *and* E *such that* $\tau(v) \in Type$ *if* $v \in V$ *and* $\tau(e) \in \{containment, reference\}$[4] *if* $e \in E$*. Edges* $(v, v') \in E$ *are noted* $v \rightarrow v'$*. We furthermore impose that the graph* $\langle V, \{v \rightarrow v' \in E | \tau(v \rightarrow v') = containment\}\rangle$ *is acyclic. The set of all typed graphs is called* TG*.*

An indirect typed graph is a 4-tuple $\langle V, E, T, Il\rangle$*, where* $\langle V, E, T\rangle$ *is a typed graph and* $Il \subseteq E$ *is a set of edges called* indirect links*. The set of all indirect typed graphs is called* ITG*.*

[4] By using *containment* and *reference* as types for edges we allow modeling the different types of associations between the elements of a metamodel or a model. In particular, the fact that the subgraph of containment relations in a typed graph is acyclic models EMF containment associations.

Definition 2. *Typed Graph Union*

Let $\langle V, E, \tau\rangle, \langle V', E'\tau'\rangle \in TG$ be typed graphs. The typed graph union is the function $\sqcup : TG \times TG \to TG$ defined as:

$$\langle V, E, \tau\rangle \ \sqcup \ \langle V', E', \tau'\rangle = \langle V \cup V', E \cup E', \tau \cup \tau'\rangle$$

Definition 3. *Typed Subgraph and Indirect Typed Subgraph*

Let $\langle V, E, \tau\rangle = g, \langle V', E', \tau'\rangle = g' \in TG$ be typed graphs. g' is a typed subgraph of g, written $g' \blacktriangleleft g$, iff $V' \subseteq V$, $E' \subseteq E$ and $\tau' = \tau|_{V'}$.

An indirect typed graph $\langle V', E', \tau', Il\rangle \in ITG$ is an indirect typed subgraph of a typed graph $\langle V, E, \tau\rangle \in TG$, written $\langle V', E', \tau', Il\rangle \lhd \langle V, E, \tau\rangle$ iff:

1. $\langle V', E' \setminus Il, \tau'\rangle \blacktriangleleft \langle V, E, \tau\rangle$
2. *if $v_i \to v_i' \in Il$ then there exists $v \to v' \in E_c^*$ where $\tau(v_i) = \tau(v)$, $\tau(v_i') = \tau(v')$ and E_c^* is obtained by the transitive closure of $E_c = \{v \to v' \in E | \tau(v \to v') = containment\}$.*

Definition 4. *Typed Graph Equivalence*

Let $\langle V, E, \tau\rangle = g, \langle V', E', \tau'\rangle = g' \in TG$ be typed graphs. g and g' are equivalent, written $g \cong g'$, iff there is a graph isomorphism $f : V \to V'$ of graphs $\langle V, E\rangle$ and $\langle V', E'\rangle$ such that $\forall x \in V \cup E \ . \ \tau(x) = \tau'(f(x))$ and $\forall x' \in V' \cup E' \ . \ \tau'(x') = \tau(f^{-1}(x'))$

More informally, two typed graphs are defined equivalent if they have the same shape and related vertices and edges have the same type.

Definition 5. *Typed Graph Instance*

Let $\langle V, E, \tau\rangle = g, \langle V', E', \tau\rangle = g' \in TG$ be typed graphs. g' is a typed graph instance of g, written $g' \Vdash g$, iff for all $v_1' \to v_2' \in E'$ there is a $v_1 \to v_2 \in E$ such that $\tau(v_1') = \tau(v_1)$, $\tau(v_2') = \tau(v_2)$ and $\tau(v_1' \to v_2') = \tau(v_1 \to v_2)$.

Notice that we only enforce that connections between vertices of g' must exist also in g and have the same type.

Definition 6. *Metamodel and Model*

A metamodel $\langle V, E, \tau\rangle \in TG$ is a typed graph where τ is a bijective typing function. The set of all metamodels is called $META$.

A model is a 4-tuple $\langle V, E, \tau, M\rangle$ where $\langle V, E, \tau\rangle$ is a typed graph. Moreover $M = \langle V', E', \tau'\rangle \in META$ is a Metamodel and the codomain of τ equals the codomain of τ'. Finally $\langle V, E, \tau\rangle \Vdash M$, which means $\langle V, E, \tau\rangle$ is an instance of a metamodel M. The set of all models for a metamodel M is called $MODEL^M$.

Definition 7. *Match-Apply Model*

A Match-Apply Model is a 6-tuple $\langle V, E, \tau, Match, Apply, Bl\rangle$, where $Match = \langle V', E', \tau', s\rangle$ and $Apply = \langle V'', E'', \tau'', t\rangle$ are models and $\langle V, E, \tau\rangle = \langle V', E', \tau'\rangle \sqcup \langle V'', E'', \tau''\rangle$. Edges $Bl \subseteq V' \times V'' \subseteq E$ are called backward links. *s is called the* source *metamodel and t the target metamodel. The set of all Match-Apply models for a source metamodel s and a target metamodel t is called MAM_t^s. Vertices in*

the Apply model which are not connected to backward links *are called* free vertices. *The* $back : MAM_t^s \rightarrow MAM_t^s$ *function connects all vertices in the* $Match$ *model to all* free *vertices with* backward link *edges.*

The $Match$ part of a match-apply model is used to hold the immutable source model during a transformation. The *Apply* part is used to hold the intermediate results of the transformation.

Definition 8. *Transformation Rule*

A Transformation Rule is a 7-tuple $\langle V, E, \tau, Match, Apply, Bl, Il \rangle$, *where* $\langle V, E, \tau, Match, Apply, Bl \rangle \in MAM_t^s$ *is a match-apply model.* $Match = \langle V, E, \tau, M \rangle$ *and the edges* $Il \subseteq E$ *are called* indirect links *(see definition 3). The set of all transformation rules is called* TR_t^s. *The* $strip : TR_t^s \rightarrow TR_t^s$ *function removes from a transformation rule all free vertices and associated edges.*

We define a *transformation rule* as a kind of match-apply model which allows *indirect links* in the *match* pattern.

Definition 9. *Layer, Transformation*

A layer is a finite set of transformation rules $tr \subseteq TR_t^s$. *The set of all layers for a source metamodel* s *and a target metamodel* t *is called* $Layer_t^s$. *A transformation is a finite list of layers denoted* $[l_1 :: l_2 :: \ldots :: l_n]$ *where* $l_k \in Layer_t^s$ *and* $1 \leq k \leq n$. *The set of all transformations for a source metamodel* s *and a target metamodel* t *is called* $Transformation_t^s$.

We naturally extend the notion of union (definition 2) to models (definition 6), match-apply models (definition 7) and transformation rules (definition 8). We also extend the notion of indirect typed subgraph (definition 3) to transformation rules (definition 8) and match-apply models (definition 7). Finally, we extend the notion of typed graph equivalence (definition 4) to transformation rules (definition 8).

3.2 Transformation Language Semantics

Definition 10. *Match Function*

Let $m \in MAM_t^s$ *be a model and* $tr \in TR_t^s$ *be a transformation rule. The* $match : MAM_t^s \times TR_t^s \rightarrow \mathcal{P}(TR_t^s)$ *is defined as follows:*

$$match_{tr}(m) = remove\big(\{g \mid g \lhd m \wedge g \cong strip(tr)\}\big)$$

Due to the fact that the $\cong$ *relation is based on the notion of graph isomorphism, permutations of the same match result may exist in the* $\{g \mid g \lhd m \wedge g \cong strip(tr)\}$ *set. The — undefined — remove* $: \mathcal{P}(TR_t^s) \rightarrow \mathcal{P}(TR_t^s)$ *function is such that it removes such undesired permutations.*

Definition 11. *Apply Function*

Let $m \in MAM_t^s$ *be a match-apply model and* $tr \in TR_t^s$ *a transformation. The* $apply : MAM_t^s \times TR_t^s \rightarrow MAM_t^s$ *is defined as follows:*

$$apply_{tr}(m) = \bigsqcup_{g \in match_{tr}(m)} back(g \sqcup g_\Delta)$$

where g_Δ *is such that* $g \sqcup g_\Delta \cong tr$

The freshly created vertices of g_Δ *in the flattened* $apply_{tr}(m)$ *set are disjoint.*

Definitions 10 and 11 are complementary: the former gathers all subgraphs of a match-apply graph which match a transformation rule; the latter builds the new instances which are created by applying that transformation rule as many times as the number of subgraphs found by the *match* function. The *strip* function is used to enable matching over backward links but not elements to be created by the transformation rule. The *back* function connects all newly created vertices to the elements of the source model that originated them.

Definition 12. *Layer Step Semantics*

Let $l \in Layer$ *be a Layer. The* layer step relation $\overset{layerstep}{\rightarrow} \subseteq MAM_t^s \times TR_t^s \times MAM_t^s$ *is defined as follows:*

$$\frac{}{\langle m, m', \emptyset \rangle \xrightarrow{layerstep} m \sqcup m'}$$

$$\frac{tr \in l,\ apply_{tr}(m) = m''', \quad \langle m, m'' \sqcup m''', l \backslash \{tr\} \rangle \xrightarrow{layerstep} m'}{\langle m, m'', l \rangle \xrightarrow{layerstep} m'}$$

where $\{m, m', m''\} \subseteq MAM_t^s$ *are match-apply models.*

The freshly created vertices in m''' *are disjoint from those in* m''*.*

For each layer we go through all the transformation rules and build for each one of them the set of new instances created by their application. These instances are built using the *apply* function in the second rule of definition 12. The new instance results of the *apply* function for each transformation rule are accumulated until all transformation rules are treated. Then, the first rule of definition 12 will merge all the new instances with the starting match-apply model. The merge is performed by uniting (using the non-disjoint $\sqcup$ union) match-apply graphs including the new instances with the starting match-apply model.

Definition 13. *Transformation Step Semantics*

Let $[l :: R] \in Transformation_t^s$ *be a Transformation, where* $l \in Layer_t^s$ *is a Layer and* R *a list. The* transformation step relation $\overset{trstep}{\rightarrow} \subseteq MAM_t^s \times TR_t^s \times MAM_t^s$ *is defined as follows:*

Let $[l :: R] \in Transformation_t^s$ *be a Transformation, where* $l \in Layer_t^s$ *is a Layer and* R *a list. The* transformation step relation $\overset{trstep}{\rightarrow} \subseteq MAM_t^s \times TR_t^s \times MAM_t^s$ *is defined as follows:*

$$\frac{}{\langle m, [] \rangle \xrightarrow{trstep} m}$$

$$\frac{\langle m, \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle, l \rangle \xrightarrow{layerstep} m'',\ \langle m'', R \rangle \xrightarrow{trstep} m'}{\langle m, [l :: R] \rangle \xrightarrow{trstep} m'}$$

where $\{m, m', m''\} \subseteq MAM_t^s$ *are match-apply models.*

A model transformation is a sequential application of transformation layers to a match-apply model containing the source model and an empty apply model. The transformation output is the apply part of the resulting match-apply model.

Definition 14. *Model Transformation*

Let $m_s \in MODEL^s$ *and* $m_t \in MODEL^t$ *be models and* $tr \in Transformation_t^s$ *be a transformation. A model transformation* $\overset{transf}{\rightarrow} \subseteq MODEL^s \times Transformation_t^s \times MODEL^t$ *is defined as follows:*

$$m_s, tr \xrightarrow{transf} m_t \Leftrightarrow \langle V, E, \tau, m_s, \emptyset, \emptyset \rangle, tr \xrightarrow{trstep} \langle V, E, \tau, m_s, m_t, Bl \rangle$$

We now prove two important properties about DSLTrans' transformations.

Proposition 1. *Confluence*

Every model transformation is confluent regarding typed graph equivalence.

Proof. (Sketch) We want to prove that for every model transformation $tr \in Transformation_t^s$ having as input a model $m_s \in MODEL^s$, if $m_s, tr \xrightarrow{transf} m_t$ and $m_s, tr \xrightarrow{transf} m_t'$ then $m_t \cong m_t'$. Note that we only have to prove typed graph equivalence between m_t and m_t' because the identifiers of the objects produced by a model transformation are irrelevant.
If we assume $\neg(m_t \cong m_t')$ then this should happen because of non-determinism points in the rules defining the semantics of a transformation: 1) in definition 11 g_Δ is non-deterministic up to typed graph equivalence, which does not contradict the proposition; 2) in definition 12 transformation rule tr is chosen non-deterministically from layer l. Thus, the order in which the transformation rules are treated is non-deterministic. However, the increments to the transformation by each rule of a layer are united using $\sqcup$, which is commutative and thus renders the transformation result of each layer deterministic. Since there are no other possibilities of non-determinism points in the semantics of a transformation, $\neg(m_t \cong m_t')$ provokes a contradiction and thus the proposition is proved. □

Proposition 2. *Termination*

Every model transformation terminates.

Proof. (Sketch) Let us assume that there is a transformation which does not terminate. In order for this to happen there must exist a section of the semantics of that transformation which induces an algorithm with an infinite amount of steps. We identify three points of a transformation's semantics where this can happen: 1) if definition 13 induces an infinite amount of steps. The only possibility for this to happen is if the transformation has an infinite amount of layers,

which is a contradiction with definition 9; 2) if definition 12 induces an infinite amount of steps. The only possibility for this to happen is if a layer has an infinite amount of transformation rules, which is a contradiction with definition 8; 3) if the result of the $match_{tr}(m)$ function in definition 10 is an infinite set of match-apply graphs. The match-apply graph m is by definition finite, thus the number of isomorphic subgraphs of m is infinite only if the transitive closure of containment edges of m is infinite. The only possibility for this to happen is if the graph induced by the containment edges of m has cycles, which contradicts definition 1. Since there are no more points in the semantics of a transformation that can induce an infinite amount of steps, the proposition is proved. □

4 Conclusions

We have presented DSLTrans, a turing incomplete transformation language with a mathematical underpinning which guarantees transformation termination and confluence by construction. With this language, we have introduced interesting abstractions such as layers, backward and indirect links. An important side effect of DSLTrans not being a turing complete language is the fact that verification of properties about our transformations are possible.

References

1. Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *MODELS'08: Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, pages 53–67, Berlin, Germany, 2008. Springer.
2. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, 2006.
3. Hartmut Ehrig, Karsten Ehrig, Juan De Lara, Gabriele Taentzer, Dniel Varr, and Szilvia Varr-gyapay. Termination criteria for model transformation. In *Proc. Fundamental Approaches to Software Engineering (FASE*, pages 49–63. Springer, 2005.
4. Tihamér Levendovszky, Ulrike Prange, and Hartmut Ehrig. Termination criteria for dpo transformations with injective matches. *Electron. Notes Theor. Comput. Sci.*, 175(4):87–100, 2007.
5. Levi Lúcio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In *To appear in the Proceedings of MODELS 2010.* Springer, 2010.
6. Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
7. D. Plumpf. Termination of graph rewriting is undecidable. *Fundam. Inf.*, 33(2):201–209, 1998.
8. Solar Group. Dsltrans plug-in. `http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ReleaseFiles/dsltrans.october.2010.zip`.