Electronic Communications of the EASST
Volume 49 (2012)

Proceedings of the
First International Workshop on
Bidirectional Transformations
(BX 2012)

Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques

Anthony Anjorin, Gergely Varró and Andy Schürr

16 pages

Guest Editors: Frank Hermann, Janis Voigtländer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques

Anthony Anjorin*, **Gergely Varró**† **and Andy Schürr**

anthony.anjorin, gergely.varro, andy.schuerr@es.tu-darmstadt.de
Real-Time Systems Lab,
Technische Universität Darmstadt, Germany

Abstract: Model transformation plays a central role in Model-Driven Engineering (MDE) and providing bidirectional transformation languages is a current challenge with important applications. Triple Graph Grammars (TGGs) are a formally founded, bidirectional model transformation language shown by numerous case studies to be quite promising and successful. Although TGGs provide adequate support for structural aspects via object patterns in TGG rules, support for handling complex relationships between different attributes is still missing in current implementations. For certain applications, such as bidirectional model-to-text transformations, being able to manipulate attributes via string manipulation or arithmetic operations in TGG rules is vital. Our contribution in this paper is to formalize a TGG extension that provides a means for complex attribute manipulation in TGG rules. Our extension is compatible with the existing TGG formalization, and retains the "single specification" philosophy of TGGs.

Keywords: bidirectional model transformation, triple graph grammars, constraint-based programming techniques, pattern matching, complex attribute manipulation

1 Introduction and Motivation

Model-Driven Engineering (MDE) has established itself as a viable means of coping with the increasing complexity of modern software systems. Model-driven techniques promise an increase in productivity and quality of software, as well as support for interoperability and improved communication with domain experts. Model transformation is a fundamental and central task for any successful MDE solution [BG01] and an open challenge is catering for *bidirectionality*. Bidirectional transformations are relevant for a multitude of applications that cut across various technologies and communities [CFH+09]. Many different approaches provide support for bidirectionality and aim to reduce the effort of keeping a pair of unidirectional forward and backward transformations consistent. Important challenges for a bidirectional language, therefore, include increasing productivity by guaranteeing useful properties and well-behavedness of the pair of transformations without compromising expressiveness, and providing an efficient, usable implementation to tackle real-world problems.

* Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.
† Supported by the Postdoctoral Research Fellowship of the Alexander von Humboldt Foundation and associated with the Center for Advanced Security Research Darmstadt.

Strategies include providing reversible languages, working with primitives that preserve bidirectionality under composition, deriving the reverse of a given unidirectional transformation automatically, and exploiting trace information. For a detailed overview of bidirectional languages and approaches we refer to [CFH+09, Ste08].

The Triple Graph Grammar (TGG) approach [Sch94] provides a language for describing the *simultaneous* evolution of two models and a third correspondence model. The specification is in form of a graph grammar: a set of rules which can be used to generate *triple graphs* consisting of related source, correspondence and target graphs from which *operational* rules can be derived for forward and backward transformations. The derivation process and the control algorithm for applying the operational rules are formally founded, guaranteeing properties such as correctness, completeness, termination and an upper bound for runtime complexity [KLKS10]. In addition to a mature formal foundation, numerous implementations for TGGs exist, ranging from an interpreter to a code generator, all under active development by different research groups. The various TGG implementations have been used successfully for different applications [Kön08, Wag09, DG09, LSRS10], and from the experience gained over the years, improving the *expressiveness* of TGGs as a bidirectional transformation language has been identified as a major challenge. A concrete feature stated by various authors [Sch94, KW07, Kön08, Wag09, DG09] as being relevant and important, is the *manipulation of attribute values* in TGG rules. Even though there has been progress in this direction, the solutions have either neither been fully formalized nor implemented [KW07], are restricted to simple attribute manipulation (i.e., attribute assignments) [Kön08, Wag09, DG09], or require the user to specify a pair of functions, one for each direction [GH09]. In all cases, it is unclear how support for reuse and composition of such "attribute constraints" can be provided.

Our contribution in this paper is to formalize an attribute manipulation approach for TGGs, which is compatible with the existing TGG formalization, preserves the TGG philosophy of having a single specification of a bidirectional transformation, supports composition and reuse, and serves as a clear, well-defined interface to Java for user-defined attribute constraints.

The paper is structured as follows: in Sect. 2, a running example is introduced and used to present a formalization of MDE terminology such as models and metamodels according to [EEPT06]. Section 3 discusses a TGG specification for this example, explains the challenge of describing the required attribute manipulation, and presents our extension. Formal results from [EEE+07] are extended in Sect. 4 to show that our extension is compatible with existing TGG theory. Our approach is compared with alternative solutions in Sect. 5, while Sect. 6 concludes with a brief summary and discussion of future work.

2 Running Example and Formalization of Basic MDE Concepts

Various industrial applications have been investigated using TGGs [LSRS10, Kön08, KW07]. In many cases, especially when the application involves a model-to-text transformation [Wag09] or a generic tree-like data structure exported from a tool, the required TGG rules entail not only structural changes but also non-trivial attribute manipulation.

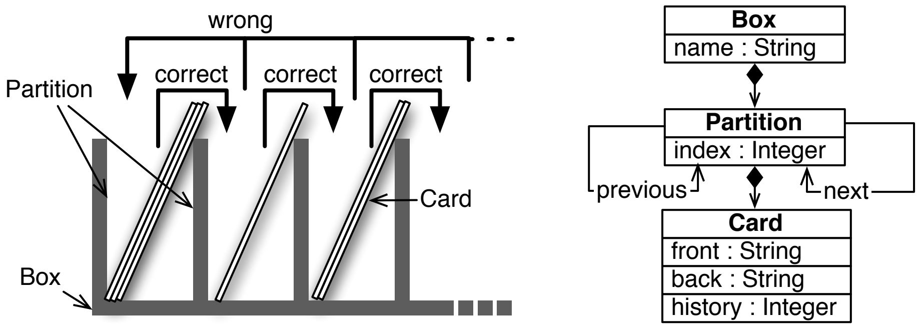


Figure 1: Leitner's learning box with corresponding metamodel

To discuss this in more detail we use a *learning box* according to the Leitner system[1], which mimics the human short-term and long-term memory and optimizes the frequency with which *flashcards* must be repeated for effective learning (Fig. 1). The corresponding *metamodel* for the learning box specifies three concepts: A `Box` represents a single learning box that consists of `Partitions`, which can each contain a number of `Cards`. A box has a name and each partition has an index that specifies the position of the partition in the box. In addition, each partition has a next and previous reference to another partition. Cards have string attributes (front and back) that represent the content to be memorized, e.g., a word in German and its translation in English. Cards also have an integer attribute, history, used to encode how often the card has been repeated. As indicated by the arrows in the schematic representation of the learning box to the left of Fig. 1, if the content of a card is memorized and can be recalled correctly, the card is moved to the next partition, if the content has been forgotten, the card is moved to the *first* partition in the box. These rules (which can also be varied) are the reason why the next and previous references are *not* opposites of each other. The first partition is to be repeated every day while all other partitions should be repeated only when enough cards have reached the partition, hence, less frequently. In the following, we formalize the concept of models and metamodels according to [EEPT06].

Models are formalized as *graphs* consisting of vertices and edges. Additionally, to cater for attribute values of vertices, data vertices and vertex attribute edges are introduced:

Definition 1 (Graph and Graph Morphism)
A *graph* $G = (V_G, E_G, src_G, trg_G, V_D, E_D, src_D, trg_D)$ consists of the sets:
(1) V_G and V_D, called the graph vertices and data vertices, respectively
(2) E_G and E_D called the graph edges and vertex attribute edges, respectively
and the source and target functions:
(3) $src_G : E_G \to V_G$, $trg_G : E_G \to V_G$ for graph edges
(4) $src_D : E_D \to V_G$, $trg_D : E_D \to V_D$ for vertex attribute edges
Let G and G' be graphs. A *graph morphism* $f : G \to G'$ is a tuple $(f_{V_G}, f_{E_G}, f_{V_D}, f_{E_D})$ with $f_{V_G} : V_G \to V_{G'}$, $f_{E_G} : E_G \to E_{G'}$, $f_{V_D} : V_D \to V_{D'}$, $f_{E_D} : E_D \to E_{D'}$ such that f commutes with all source and target functions, e.g., $f_{V_G} \circ src_G = src_{G'} \circ f_{E_G}$.

[1] http://en.wikipedia.org/wiki/Leitner_system

The actual values of attributes are formalized by attributing graphs using *algebras*, which implement an *algebraic signature* specifying types or *sorts* of the attributes (e.g., Integer, String) and *operation symbols* (e.g., length: String $\rightarrow$ Integer):

Definition 2 (Algebraic Signature, Σ-algebra, Homomorphism)
An *algebraic signature* $\Sigma = (S, OP)$ consists of a set S of sorts and a family OP of operation symbols. A Σ-algebra $A = ((A_s)_{s\in S}, (op_A)_{op\in OP})$ is defined by:
(1) For each sort $s \in S$, a set A_s, called the carrier set
(2) For each operation symbol $op : s_1 \dots s_n \rightarrow s \in OP$, a mapping $op_A : A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$
For Σ-algebras A and A', an *algebra homomorphism* $h : A \rightarrow A'$ is a family $h = (h_s)_{s\in S}$ of mappings $h_s : A_s \rightarrow A'_s$, such that $\forall\ op : s_1 \dots s_n \rightarrow s \in OP$, and $\forall x_i \in A_{s_i},\ i \in \{1,\dots,n\}$,
$h_s(op_A(x_1,\dots,x_n)) = op_{A'}(h_{s_1}(x_1),\dots,h_{s_n}(x_n))$.

Graphs can now be attributed by using an algebra to provide the data vertices in the graph:

Definition 3 (Attributed Graph and Attributed Graph Morphism)
Let $\Sigma = (S, OP)$ be a signature. An *attributed graph* $AG = (G, A)$ consists of a graph G together with a Σ-algebra A, such that V_D is the disjoint union of all A_s, i.e., $\biguplus_{s\in S} A_s = V_D$.
Given two attributed graphs $AG = (G, A)$ and $AG' = (G', A')$, an *attributed graph morphism* $f : AG \rightarrow AG'$ is a pair $f = (f_G, f_A)$ with a graph morphism $f_G : G \rightarrow G'$ and an algebra homomorphism $f_A : A \rightarrow A'$ such that f_{G,V_D} restricted to A_s is identical to $f_{A,s}$, i.e., $f_{G,V_D|_{A_s}} = f_{A,s},\ \forall s \in S$.

Metamodels are models and are, therefore, also formalized as attributed graphs. The attribute values in a metamodel, however, are used to specify the allowed *type* and not the concrete values of attributes in models that conform to the metamodel. This is formalized via a *final algebra*:

Definition 4 (Final Algebra)
Let $\Sigma = (S, OP)$ be a signature. The *final Σ-algebra* Z is defined by:
(1) $Z_s = \{s\}$ for each sort $s \in S$
(2) $op_Z : Z_{s_1} \dots Z_{s_n} \rightarrow Z_s,\ (s_1,\dots,s_n) \mapsto s$ for each operation symbol $op : s_1,\dots,s_n \rightarrow s \in OP$

The "conforms to" relationship between metamodels and models can now be formalized as a type morphism between a *type graph* attributed with a final algebra, and attributed *typed graphs*:

Definition 5 (Typed Attributed Graph and Typed Attributed Graph Morphism)
Let $\Sigma = (S, OP)$ be a signature. An *attributed type graph* is an attributed graph $ATG = (TG, Z)$, where Z is the final Σ-algebra.
A *typed attributed graph* $(AG,\ type)$ over ATG consists of an attributed graph AG and an attributed graph morphism $type : AG \rightarrow ATG$.
A *typed attributed graph morphism* $f : (AG, type) \rightarrow (AG', type')$ is an attributed graph morphism $f : AG \rightarrow AG'$ such that $type = type' \circ f$.

A dictionary, the second metamodel for our example, is depicted in Fig. 2 together with a concrete model and the corresponding formalization (using Definitions 1–5) as an attributed type graph and typed graph, respectively[2]. A `Dictionary` has a name and consists of arbitrary many entries. Every `Entry` has a level, a string from the set {"*beginner*", "*advanced*", "*master*"} indicating how difficult the entry is, and a content (what is displayed in the dictionary).

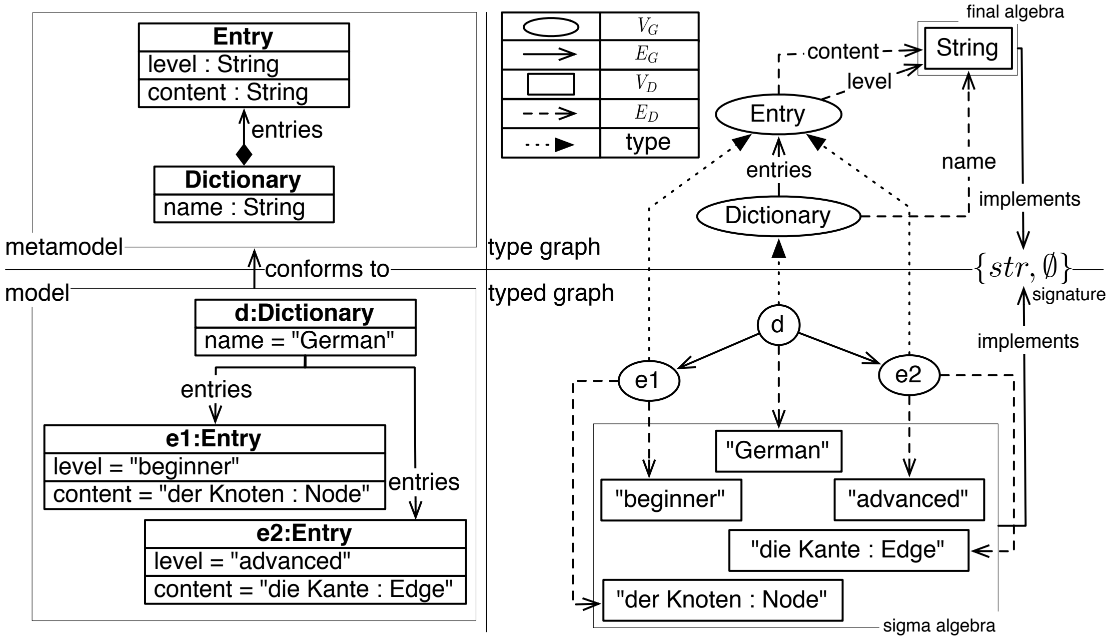


Figure 2: Metamodel and model of a dictionary and corresponding formalization

To learn a new language, one can start with a learning box and a set of cards with words and their translations. When all the cards have been successfully memorized, the vocabulary could be preserved for future reference by transforming the box to a dictionary, which is more suitable for looking up specific words. The difficulty `level` of the entries in the dictionary would be set according to the `history` of the card so as to personalize the dictionary. A dictionary is, however, not an ideal format for actually memorizing its contents, and, if most words have been forgotten, it would be better to transform the dictionary to a suitably "pre-configured" learning box, which can be used to (re)learn the set of words effectively. Pre-configuration could mean taking the difficulty level of each entry in the dictionary into account and already placing each card in a suitable partition, i.e., "easy" entries, from cards that were learnt easily the first time, can probably be also re-learnt faster and do not need to be placed in the first partition. Using a *bidirectional* transformation language for this task could offer various advantages including reducing the effort of specifying the transformation and guaranteeing consistency.

3 A Constraint-Based Attribute Manipulation Approach for TGGs

The basic idea with TGGs is to describe a bidirectional transformation with a single specification from which different unidirectional transformations can be derived. This specification is in form

[2] Please note that only the types of nodes in the typed graph are indicated to simplify the diagram, i.e., edges are also typed by the type morphism but this is not indicated in the diagram.

of a graph grammar, i.e., a set of TGG rules, and can be regarded as inducing a consistency relation on related source, correspondence and target models (graph triples) in the following manner: A triple consisting of source, correspondence and target models is *consistent* with respect to a given TGG if it can be generated using rules in the TGG.

Our approach extends TGGs by adding a *Constraint Satisfaction Problem* (CSP) over attributes for each TGG rule, effectively extending the consistency relationship to encompass not only the graphical structure of the models but also the relative values of their attributes. In the following we present basic definitions based on [Sch94, EEE+07, KLKS10, EEPT06], introducing our new concept of a CSP over attributes in a TGG context, and extending the notion of TGG rules appropriately. After presenting the TGG for our example, we discuss in Sect. 4 how TGG rules with attribute constraints can be decomposed into *operational rules*, which are then used to derive forward/backward unidirectional model transformations.

Model transformation is achieved by applying a sequence of *rules* or *productions* that consist of a precondition and postcondition. The *model fragments* or *patterns* in a rule are formalized as typed graphs, attributed with arbitrary *terms* (e.g., $x+y$) over variables (e.g., $x,\ y$).

Definition 6 (Term Algebra $T_\Sigma(X)$)
Let $\Sigma = (S, OP)$ be a signature and $X = (X_s)_{s\in S}$ a family of pairwise disjoint sets, which are also disjoint with OP. Each X_s is called the set of *variables* of sort s.
The algebra $T_\Sigma(X) = ((T_{\Sigma,s}(X))_{s\in S}, (op_{T_\Sigma(X)})_{op\in OP})$ is called the *term algebra* over Σ and X, where the carrier sets $(T_{\Sigma,s}(X))_{s\in S}$ consist of *terms* with variables, and with operations defined by $op_{T_\Sigma(X)} : T_{\Sigma,s_1}(X) \times \ldots \times T_{\Sigma,s_n}(X) \to T_{\Sigma,s^*}(X)$, $\forall op : s_1 \ldots s_n \to s^* \in OP$.

Constraints are defined as terms in a term algebra that are of type *Bool* (evaluate to true or false):

Definition 7 (Constraint Satisfaction Problem (CSP) over $T_\Sigma(X)$)
Let $\Sigma = (S, OP)$ be a signature with a distinguished sort $Bool \in S$, $T_\Sigma(X)$ a term algebra over Σ and variables X, and A a Σ-algebra with $A_{Bool} = \{true, false\}$.
A *constraint* c is a term in $T_\Sigma(X)$ of sort *Bool*.
A *Constraint Satisfaction Problem (CSP)* over $T_\Sigma(X)$ is a set $\mathscr{C}$ of constraints.
An *assignment* $asgn : X \to A$ is a family of assignment functions $asgn_s : X_s \to A_s$, which, according to [EEPT06], can always be uniquely extended to $\overline{asgn} : T_\Sigma(X) \to A$. An assignment $asgn : X \to A$ *fulfills* a CSP $\mathscr{C}$, denoted as $asgn \models \mathscr{C}$, if $\forall c \in \mathscr{C}$, $\overline{asgn}(c) = true$.

The concepts introduced in Definitions 1–7 can be lifted to *triples* of typed attributed graphs $(G,A) := (G_S,A) \xleftarrow{s_G} (G_C,A) \xrightarrow{t_G} (G_T,A)$ with a common algebra A, typed over a type graph triple $(TG,Z) := (TG_S,Z) \xleftarrow{s_{TG}} (TG_C,Z) \xrightarrow{t_{TG}} (TG_T,Z)$, with common final algebra Z. The typed attributed graph morphisms s_G and t_G connect the correspondence graph G_C with the source graph G_S and the target graph G_T (s_{TG} and t_{TG} analogously). Due to space limitations, we do not treat this extension explicitly and refer to [EEPT06, KLKS10] for further details.

Patterns in TGG rules are formalized as graph triples attributed over a term algebra, which is also used to formalize the constraints (CSP) in TGG rules. TGG rules are *applied* by determining a match for the left hand side pattern in a given graph, and replacing this with the right hand side pattern. In the following, all triple graphs are considered to be typed attributed triple graphs.

Definition 8 (Triple Graph Rule with CSP)
Let Σ be a signature and A a Σ-algebra as in Def. 7.
A *triple graph rule (or production)* $p = (L, R, \mathscr{C})$ consists of triple graphs $L, R : L \subseteq R$ with common term algebra $T_\Sigma(X)$, and a CSP $\mathscr{C}$ over $T_\Sigma(X)$.
A match $m : (L, T_\Sigma(X)) \to (G, A)$ consists of a graph part $m_G : L \to G$ and a data type part $m_A : T_\Sigma(X) \to A$ that is completely determined by an assignment $asgn : X \to A$, such that $asgn \models \mathscr{C}$. In the following this is denoted by $G \xrightarrow{p@m \models \mathscr{C}} G'$.
A triple graph rule p can be *applied* to a triple graph (G, A) at a *match* m to yield G' via a *Push Out (PO)*[3] as depicted in the diagram to the right.

$$\begin{tikzcd} L \arrow[r, "\subseteq"] \arrow[d, "m"'] & R \arrow[d, "m'"] \\ G \arrow[r, "\subseteq"] & G' \end{tikzcd}$$

A consistency relation is induced by the *TGG language* (all models that can be generated using the TGG), i.e., two models are consistent if they can be extended to a triple in the TGG language.

Definition 9 (Triple Graph Grammar and Triple Graph Grammar Language)
A *Triple Graph Grammar* is a pair $TGG = (TG, P)$, where TG is an attributed type triple graph and P is a set of rules. The language $\mathscr{L}(TGG)$ is the set of all triple graphs that can be derived from $G_\emptyset$, the empty triple graph, by applying a finite sequence of triple graph rules in P.

Example: Figure 3 depicts the triple graph rule `CardToEntry` for our example[4]. L and R of the rule are superimposed in a single diagram where required or context elements (L) are black while the green colour and ++ markup indicate $R \setminus L$, i.e., the elements to be created by the rule. A learning box and its corresponding dictionary (created by another rule that establishes the basic structures) are extended with cards and entries, respectively.

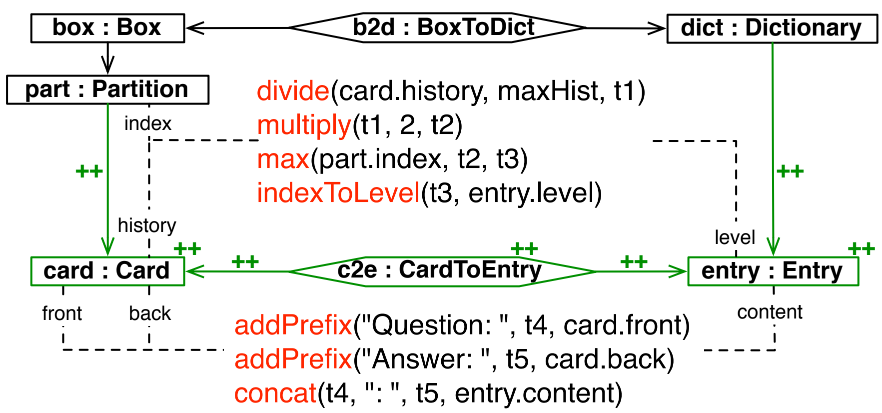


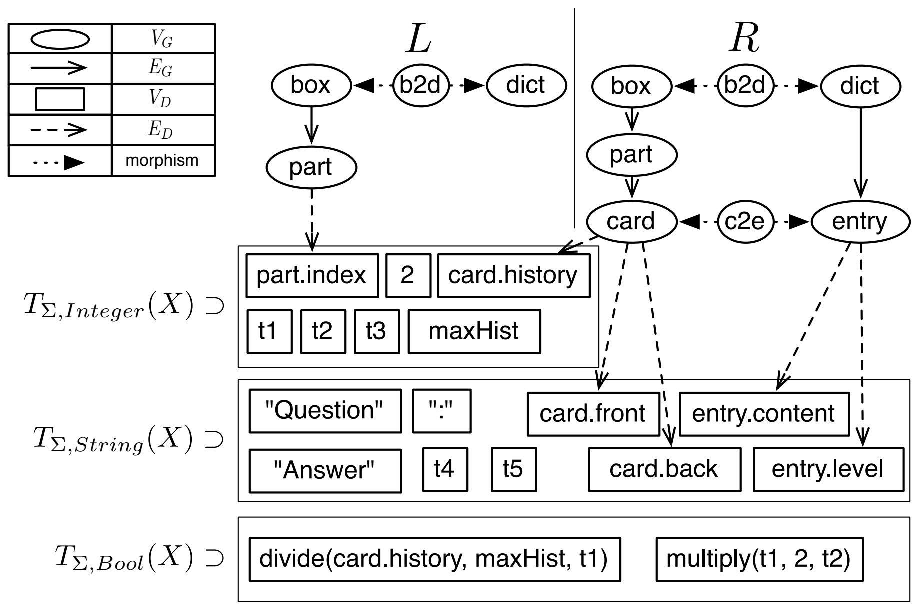
Figure 3: TGG rule for running example: `CardToEntry(maxHist:int)`

The CSP of each rule is specified using a textual syntax. For `CardToEntry` (Fig. 3), the CSP depicted in the rule specifies all dependencies motivated in the informal description of the transformation in Sec. 2. The first four constraints specify the dependencies between the `history` of a card, the `index` of its containing partition and the `level` of the corresponding

[3] Triple graph productions are, therefore, non-deleting.
[4] The metamodel for the correspondence domain (not shown explicitly due to space limitations) consists of only the two link types `BoxToDict` and `CardToEntry` as used in the rule.

dictionary entry. The parameter `maxHist` of the rule is used to normalize the `history` of the current card with `divide`, such that `t1` $\in [0,1]$. As we have three levels for our dictionary, this normalized value is scaled to `t2` $\in [0,2]$ with `multiply`. This is then compared with the actual `index` of the partition using `max`, which ensures that the maximum value is contained in `t3`. Finally, `indexToLevel` is used to convert the float `t3` to a valid level, i.e., a string $\in$ {"*beginner*", "*advanced*", "*master*"}. The remaining three constraints ensure that the `front` and `back` of a card correspond to the `content` of a dictionary entry, adding the prefixes "`Question:`" and "`Answer:`" via `addPrefix`, after splitting the values via `concat` using a colon "`:`" as a separating character.

Figure 4 shows the connection of the concrete syntax of `CardToEntry` to the formalization introduced in Definitions 6–9. Note that variables are trivial terms, constants are formalized as nullary operations, and, to simplify the diagram, not all terms and operations are shown.



$$\Sigma = \{\{String, Integer, Bool\}, \{divide, multiply, 2, \ldots\}\}$$

$$\begin{aligned} X = \{X_{Integer} &= \{part.index, card.history, maxHist, t1, t2, t3\}, \\ X_{String} &= \{card.front, card.back, entry.content, entry.level, t4, t5\}\} \end{aligned}$$

Figure 4: Formalization of TGG rule `cardToEntry`

4 From TGGs to Model Transformations

Although TGGs can be directly used to evolve three models simultaneously, the real potential of TGGs as a bidirectional language lies in the automatic derivation of unidirectional model transformations. To keep the discussion as clear as possible, only the derivation of a forward transformation is discussed. By replacing *source* with *target* and *forward* with *backward* all results can be transferred analogously for the derivation of a backward transformation.

The main idea is to *decompose* every TGG rule into a *source rule* that only transforms the source component of a triple graph, and a *forward rule* that retains the source component and transforms the correspondence and target components (Def. 10). Any source model that can be created with source rules (referred to as *source consistent*) can be inspected to determine an appropriate sequence of forward rules (referred to as a forward graph transformation) that retain the given source model and extend it to a consistent triple [Sch94, EEE+07]. We extend this operationalization process to include our introduced CSPs by regarding each constraint as an atomic unit for which the user must supply corresponding *operations* implemented in Java.

For example, the constraint `indexToLevel` from our running example would have (i) a forward operation that determines the corresponding level for the index given by its first argument, and assigns or *binds* this value to its second argument, and (ii) a backward operation that determines the corresponding index for the level given by its second argument, and assigns this value to its first argument, and (iii) a check operation that ensures both values (index and level) are consistent. Note that not all possible combinations have to be supported for every constraint, e.g., the user can decide if `indexToLevel` makes sense when both arguments are *free* and must be assigned consistent default values, or not.

Such atomic constraints with supplied operations can be reused multiple times in different CSPs, and combined with other constraints in an arbitrary order as shown in our running example. The task of operationalizing the complete CSP is now to determine a correct sequence of corresponding operations (a *search plan*), such that all variables can be assigned values (bound) by executing the operations one after the other. This sorting process has to take the mode (forward, backward) and the operations that each constraint supports into account. In our implementation, we are able to reuse the *very same* search plan algorithm used for graph pattern matching, by swapping the typical graph constraints (type, link) with our user defined attribute constraints (`indexToLevel`, `concat`). In this way, the same transformation engine can be used to realize our extension without adding any further dependencies.

The complete process for each forward rule is performed in the following steps:

1. The left hand side of the forward rule is matched. This determines values for all bound attributes required for solving the CSP.
2. The search plan (determined at compile time) is executed to bind all remaining attributes.
3. The forward rule is now applied using the determined bindings for all attributes.

Although atomic constraints are operationalized by the user, our approach allows for flexible composition and reuse, enabling a well-defined means of integrating user defined functionality in Java with TGGs, and the possibility of establishing reusable *libraries* of constraints.

In the following, we formalize our approach by extending the Decomposition and Composition Theorem [EEE+07], which is the basis for proving correctness and completeness of derived forward graph transformations. The following definition states how a TGG rule can be decomposed into *operational* source and forward rules:

Definition 10 (Derived Triple Rules: Source Rules and Forward Rules)
Given a triple graph $G = (G_S \xleftarrow{s_G} G_C \xrightarrow{t_G} G_T)$, the *projection* to the source is defined as $proj_S(G) := (G_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset)$. For a triple graph morphism $f = (f_S, f_C, f_T)$, $proj_S(f) := f_S$. Projections to the target and correspondence are defined analogously.

From TGG rule $p = (L, R, \mathscr{C})$, the following operational rules can be derived:

1. A *source rule* $p_S = (L_{p_S}, R_{p_S}, \mathscr{C})$ that transforms only the source component $proj_S(G)$ of G. Applying p_S requires a match m_S with an assignment $asgn_S : asgn_S \models \mathscr{C}$ (Def. 8).

2. A *forward rule* $p_F = (L_{p_F}, R_{p_F}, \mathscr{C})$ that transforms the correspondence and target components of G while retaining its source component. Applying p_F requires a match m_F with an assignment $asgn_F : asgn_F \models \mathscr{C}$.

$$\begin{tikzcd}
L_{p_S} = L_S \arrow[d, "p_S"] & \emptyset \arrow[l] \arrow[r] \arrow[d] & \emptyset \arrow[d] \\
R_{p_S} = R_S & \emptyset \arrow[l] \arrow[r] & \emptyset
\end{tikzcd}
\longleftarrow
\begin{tikzcd}
L = L_S \arrow[d, "s"] & L_C \arrow[l, "s_L"'] \arrow[r, "t_L"] \arrow[d, "c"] & L_T \arrow[d, "t"] \\
R = R_S & R_C \arrow[l, "s_R"'] \arrow[r, "t_R"] & R_T
\end{tikzcd}
\longrightarrow
\begin{tikzcd}
L_{p_F} = R_S \arrow[d, "id"] & L_C \arrow[l, "s \circ s_L"'] \arrow[r, "t_L"] \arrow[d, "c"] & L_T \arrow[d, "t"] \\
R_{p_F} = R_S & R_C \arrow[l, "s_R"'] \arrow[r, "t_R"] & R_T
\end{tikzcd}$$

Source Rule — TGG Rule — Forward Rule

The following Theorem 1 proves that it is always possible to take a sequence of TGG rules, decompose each rule in the sequence in operational rules according to Def. 10, *and* reorder the rules until all the source rules can be applied before all the forward rules. This fundamental result allows us to apply forward rules to a given source consistent model (the sequence of source rules is assumed to be already "applied"). We extend this theorem and the proof from [EEE+07] appropriately, to take our CSPs into account and show that all results still hold.

Theorem 1 (Decomposition and Composition of Triple Graph Transformation Sequences)
Given a TGG = (P, $G_\emptyset$), let $p_i = (L_i, R_i, \mathscr{C}_i) \in P$ be the TGG rule with derived operational rules p_{i_S}, p_{i_F} for $i \in \{1 \ldots n\}$ according to Def. 10.

- **Decomposition** : *For each transformation sequence*
  (1) $G_{00} \xrightarrow{p_1 @ m_1 \models \mathscr{C}_1} G_{11} \xrightarrow{p_2 @ m_2 \models \mathscr{C}_2} \ldots \xrightarrow{p_n @ m_n \models \mathscr{C}_n} G_{nn}$
  there is a corresponding match consistent transformation sequence
  (2) $G_{00} \xrightarrow{p_{1_S} @ m_{1_S} \models \mathscr{C}_1} G_{10} \xrightarrow{p_{2_S} @ m_{2_S} \models \mathscr{C}_2} \ldots \xrightarrow{p_{n_S} @ m_{n_S} \models \mathscr{C}_n} G_{n0}$,
  $G_{n0} \xrightarrow{p_{1_F} @ m_{1_F} \models \mathscr{C}_1} G_{n1} \xrightarrow{p_{2_F} @ m_{2_F} \models \mathscr{C}_2} \ldots \xrightarrow{p_{n_F} @ m_{n_F} \models \mathscr{C}_n} G_{nn}$.
  Match consistency *means that the source component of the match of each forward rule $p_{i_F} : G_{n(i-1)} \to G_{ni}$, is completely determined by the co-match of the corresponding source rule $p_{i_S} : G_{(i-1)0} \to G_{i0}$, and the morphism $q_i : G_{i0} \to G_{n(i-1)}$ that maps the triple (G_{i0}) produced by applying the source rule, to the input triple $(G_{n(i-1)})$ for the forward rule, and that the same assignment is used for both the source and forward rule, i.e., $asgn_{i_S} = asgn_{i_F}$.*

- **Composition** : *For each match consistent transformation sequence (2) there is a canonical transformation sequence (1).*

- **Bijective Correspondence** : *Composition and Decomposition are inverse to each other.*

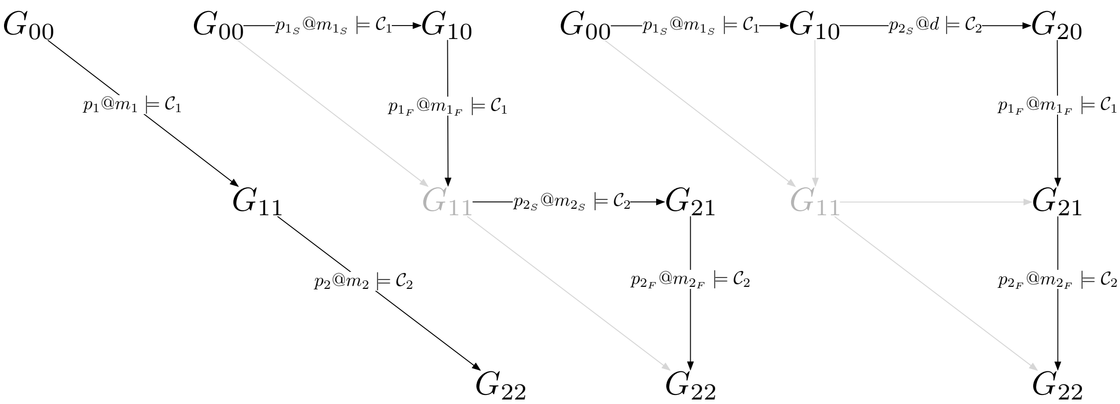
Proof.
Decomposition: Let **TripleAGraphs** be the category consisting of typed attributed triple graphs and typed attributed triple graph morphisms. As shown in [EEE+07], **TripleAGraphs** together

with the class $\mathscr{M}$ of monomorphisms is an adhesive High-Level Replacement (HLR) category, meaning that the general theory of adhesive HLR systems [EEPT06] is applicable.

Using the construction in Def. 10, $G_{00} \xrightarrow{p_1@m_1 \models \mathscr{C}_1} G_{11}$ can be split into two steps: $G_{00} \xrightarrow{p_{1_S}@m_{1_S} \models \mathscr{C}_1} G_{10} \xrightarrow{p_{1_F}@m_{1_F} \models \mathscr{C}_1} G_{11}$. As shown in [EEE+07], the Concurrency Theorem [EEPT06] guarantees that this split is always possible, is unique, and that the resulting steps are match consistent if the assignment $asgn_1$ for m_1 is used for both the source and forward rule, i.e. $asgn_1 = asgn_{1_S} = asgn_{1_F}$ so that $\mathscr{C}_1$ is satisfied in all cases ($\mathscr{C}_1 = \mathscr{C}_{1_S} = \mathscr{C}_{1_F}$). This choice ensures that $asgn_{1_S}$ does not contradict $\mathscr{C}_{1_F}$. As no new data nodes are introduced via the construction, m_1 covers all data nodes required for m_{1_S} and m_{1_F} and, therefore, $asgn_1$ is sufficient to determine the data parts of both m_{1_S} and m_{1_F}. Repeating this process, an intermediate match consistent transformation sequence (1.5) can be derived from (1):

$$(1.5)\ G_{00} \xrightarrow{p_{1_S}@m_{1_S} \models \mathscr{C}_1} G_{10} \xrightarrow{p_{1_F}@m_{1_F} \models \mathscr{C}_1} G_{11} \xrightarrow{p_{2_S}@m_{2_S} \models \mathscr{C}_2} G_{21} \xrightarrow{p_{2_F}@m_{2_F} \models \mathscr{C}_2} G_{22},$$
$$G_{22} \xrightarrow{p_{3_S}@m_{3_S} \models \mathscr{C}_3} \dots \xrightarrow{p_{n_S}@m_{n_S} \models \mathscr{C}_n} G_{n(n-1)} \xrightarrow{p_{n_F}@m_{n_F} \models \mathscr{C}_n} G_{nn}.$$

The steps $G_{10} \xrightarrow{p_{1_F}@m_{1_F} \models \mathscr{C}_1} G_{11} \xrightarrow{p_{2_S}@m_{2_S} \models \mathscr{C}_2} G_{21}$ are *sequentially independent* [EEPT06], as p_{2_S} matches the source component of G_{11} retained from G_{10} by p_{1_F}. The Local Church Rosser Theorem [EEPT06] guarantees the existence of a sequence $G_{10} \xrightarrow{p_{1_S}@d \models \mathscr{C}_2} G_{20} \xrightarrow{p_{1_F}@m_{1_F} \models \mathscr{C}_1} G_{21}$ with $d = (proj_S(m_{2_S}), \emptyset, \emptyset)$. The assignments $asgn_1$ and $asgn_2$ determine both matches completely as m_{1_F} remains unchanged and d is derived from m_{2_S}. Sequential dependency ensures that $asgn_2$ does not contradict $\mathscr{C}_1$. Applying this shift repeatedly to (1.5) leads to match consistent (2).



Composition: Given a match consistent sequence (2):

$$G_{00} \xrightarrow{p_{1_S}@m_{1_S} \models \mathscr{C}_1} G_{10} \xrightarrow{p_{2_S}@m_{2_S} \models \mathscr{C}_2} \dots \xrightarrow{p_{n_S}@m_{n_S} \models \mathscr{C}_n} G_{n0},$$
$$G_{n0} \xrightarrow{p_{1_F}@m_{1_F} \models \mathscr{C}_1} G_{n1} \xrightarrow{p_{2_F}@m_{2_F} \models \mathscr{C}_2} \dots \xrightarrow{p_{n_F}@m_{n_F} \models \mathscr{C}_n} G_{nn},$$ such that $\forall i \in \{1 \dots n\}$, $asgn_{i_S} = asgn_{i_F}$.

Due to sequential dependency and the Local Church Rosser Theorem, we can perform an inverse shift on (2) to obtain (1.5) while retaining match consistency. Match consistency allows us to use the Concurrency Theorem to merge corresponding source and forward rules to obtain (1). Assignments and constraints in (2) are compatible and can also be merged appropriately.

Bijective Correspondence: This is a direct consequence of the bijective correspondence in the Local Church-Rosser Theorem and the Concurrency Theorem [EEE+07]. □

A forward model transformation can be constructed formally by inspecting a given source consistent source model, determining the sequence of source and forward rules as in (2), and applying the forward rules to the source model to result in a consistent triple according to Theorem 1:

Definition 11 (Forward Graph Transformation *FGT*)
Given a *source consistent* input triple graph G_I, i.e., an input triple graph built up with a source rule transformation sequence: $\emptyset = G_{00} \xrightarrow{p_{1_S}@m_{1_S} \models \mathscr{C}_1} G_{10} \xrightarrow{p_{2_S}@m_{2_S} \models \mathscr{C}_2} \ldots \xrightarrow{p_{n_S}@m_{n_S} \models \mathscr{C}_n} G_{n0} = G_I$, the forward graph transformation $FGT : G_I \to G_O$ can be determined using the specified *TGG* as: $G_I = G_{n0} \xrightarrow{p_{1_F}@m_{1_F} \models \mathscr{C}_1} G_{n1} \xrightarrow{p_{2_F}@m_{2_F} \models \mathscr{C}_2} \ldots \xrightarrow{p_{n_F}@m_{n_F} \models \mathscr{C}_n} G_{nn} = G_O$.
A backward graph transformation *BGT* can be defined analogously.

Necessary Restrictions
An efficient implementation that constructs an *FGT* given an input graph and a *TGG* must:

(i) Determine the correct sequence of source rules that builds up the input graph, and
(ii) For each source rule, derive an assignment for all variables, which satisfies the CSP of the TGG rule and can be used for the forward rule.

In our current TGG implementation, we apply the following CSP-related restrictions:

1. We require that a partial assignment, i.e., the connection between terms used in the constraints and attributes in the rule, can be determined by inspecting the attributes of the input graph, for which a solution of the CSP must exist. This is enforced already at compile time during the search plan generation.
2. Attribute constraints of a TGG rule are *local* with respect to this rule, i.e., can only be specified over values of attributes of nodes used in the rule (and not in any other).
3. The actual solving/operationalization of individual constraints must be provided by the user (as Java code). Our solver is used to determine the correct *order* and *choice* of operations for arbitrary sets of constraints in each rule CSP.

5 Related Work

There exist various bidirectional model transformation languages [Ste08, CFH+09] that address the same basic challenges as the TGG approach which, when used for model synchronization, can be regarded as an implementation of symmetric delta-lenses [DXC+11] as shown in [HEO+11]. In the following we discuss three different groups of related approaches:

Other Bidirectional Languages: An approach that gives a nice contrast to TGGs is Janus, a bidirectional programming language [YAG08] that provides basic *reversible* programming primitives. As TGGs are ideal for specifying structural changes in complex graph structures but lack a means for complex attribute manipulation, and Janus excels in the bidirectional manipulation of simple data types (attributes) but faces challenges when dealing with complex data structures (graphs), a combination of both languages would be interesting. Along the same lines, approaches for bidirectional string manipulation such as Boomerang [BFP+08] could be integrated as a sublanguage in TGG rules for attribute manipulation. Combining such full-fledged bidirectional programming languages with TGGs would yield an expressive but quite complex language. It is questionable if one can require users to master two or more non-trivial languages.

Similar to TGGs, *GRoundTram*, a bidirectional framework based on graph transformations [HHI+11], aims to support model transformations in the context of MDE. GRoundTram automatically generates a consistent backward transformation from a given forward transformation specified in UnQL+, which is based on the graph query algebra UnCAL and places strong emphasis on supporting compositionality. In contrast, TGGs provide a rule-based algebraic graph transformation language from which both forward and backward transformations are automatically derived. Both approaches face a different set of non-trivial challenges also with respect to attribute manipulation.

Constraint-Based Approaches: Nentwich et al [NCEF02] show with *xlinkit* that consistency constraints can be used to implement bidirectional transformations. Although the basic idea of using constraints serves as inspiration for our extension, *xlinkit* is geared towards link creation and XML-based technologies and thus cannot be directly used in our TGG context.

The pattern-based model-to-model transformation approach presented in [GLO09, GLO10] is inspired by TGGs but is constraint-based rather than rule-based. This means that the language of consistent triples is defined by specifying a set of constraints that must be fulfilled as opposed to specifying a (triple graph) grammar. The advantages of this approach as compared to TGGs include an easier operationalization for consistency checking and a natural handling of attribute manipulation via attribute constraints. There are, however, also a few weaknesses including a more complex operationalization for forward and backward transformations, and difficulties to guarantee completeness in practice (certain heuristics must be used). Depending on the application scenario, a rule-based specification can also be more compact and intuitive, as a constraint-based specification might require numerous (negative) constraints to define the exact same language. Our approach introduces flexible attribute manipulation to TGGs by combining both approaches: the TGG rule-based approach and a constraint-based approach for attribute handling. Our attribute constraints can be formally regarded as a form of application conditions for TGGs [GEH11], which are only allowed to operate on the attributes in a single TGG rule without manipulating the triple graph structure. In contrast, [GLO09, GLO10] introduce (attribute) constraints for arbitrary graph triples, not only for patterns. Similar to our approach, [GLO09, GLO10] use a constraint solver to compute concrete values if these are required.

A constraint-based handling of attributes for general algebraic graph transformation has been introduced formally in [OL10] via symbolic graphs which might lead to a simpler formalization for our attribute constraints. Moreover, the idea of increasing efficiency via a lazy evaluation of attribute constraints and postponing constraint solving is quite interesting. As our generative approach, however, performs search plan generation already at compile-time, this is probably of greater importance for an interpretative solution.

Existing Solutions for Attribute Manipulation in TGG Rules: The requirement of supporting complex attribute manipulation in TGGs is not new and has already been identified as a major deficit by various authors [KW07, Kön08, Wag09, DG09, GLO09]. As TGGs have been aligned with the QVT specification [OMG05] in [Kön08], the approach taken by [Kön08, KW07] is similar to what is described in the specification. As these approaches have, however, not been sufficiently formalized, it is unclear how constraints that are more complex than simple *expressions* consisting of a single parameter (which are trivially revertible), are to be efficiently handled in an implementation. Furthermore, although *black box operations* can be integrated with *relations* (rules), this does not allow for the same composition and reusability that our approach does by integrating black box constraints in a CSP for each rule. In [DG09], an integration of TGGs with OCL is presented, which allows arbitrary OCL expressions in TGG rules. Currently only trivially reversible (attribute assignments) are supported in the implementation. Our solution of using a constraint solver to support complex expressions and composition can be viewed as a natural and necessary generalization and formalization of ideas from [Kön08, KW07, DG09].

Other approaches [KRW04, GH09], require the user to specify a pair of functions or constraints for each direction that can be implemented in Java or OCL. Although such practical approaches are quite expressive, they go against the TGG philosophy of providing a *single* specification from which different operational rules can be derived. A further problem is that the user is responsible for guaranteeing and maintaining consistency between the pairs of functions.

In our approach, individual constraints only need to be specified once and can then be reused and composed freely in a declarative manner in TGG rules. Furthermore, constraint (library) providers do not need worry about the correct sequence and choice of operations as this is derived automatically by our constraint solver.

6 Conclusion and Future Work

In this paper, we have presented an extension to TGGs to support complex attribute manipulation in TGG rules. Our approach has the following advantages:

1. It works well with the existing formalization of TGGs by [Sch94, EEE$^+$07] and handles the different modi (simultaneous, forward, backward) in a single specification.
2. It is quite flexible, providing a clear interface to user-defined constraints implemented in Java without introducing too much additional complexity in TGG rules.
3. It allows for a composition and reuse of constraints, which can be provided as libraries.
4. Our tool support offers a concise concrete syntax that visually differentiates between variables used for user interaction or scripting (`maxHist` in the running example), and other variables, which are to be resolved in operational rules. Dependencies between attributes are indicated unobtrusively in visual TGG rules while the details of the CSP can be specified with a suitable simple textual DSL.

In the future, we plan to investigate an alternative formalization for handling attributes and terms with variables in graph transformations [OL10], which might lead to a clearer, simpler theory. We shall explore further features such as cost functions and optional constraints, and investigate formal properties of TGGs [KLKS10, EEE$^+$07] to understand how our extension works together with other advanced TGG concepts and theory. Finally, we shall apply our implementation in practice, as a part of our metamodelling tool *eMoflon*[5][ALPS11], and gain experience with various case studies to establish a suitable set of standard libraries of constraints and explore the exact limits of our approach.

[5] www.moflon.org

Bibliography

[ALPS11] A. Anjorin, M. Lauder, S. Patzina, A. Schürr. eMoflon : Leveraging EMF and Professional CASE Tools. In *INFORMATIK 2011*. LNI 192, p. 281. GI, 2011.

[BFP+08] A. Bohannon, J. Foster, B. Pierce, A. Pilkiewicz, A. Schmitt. Boomerang : Resourceful Lenses for String Data. *ACM SIGPLAN Notices* 43(1):407–419, 2008.

[BG01] J. Bézivin, O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proc. of ASE 2001*. Pp. 273–280. IEEE, 2001.

[CFH+09] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Proc. of ICMT 2009*. LNCS 5563, pp. 260–283. Springer, 2009.

[DG09] D. Dang, M. Gogolla. On Integrating OCL and Triple Graph Grammars. In *Models in Software Engineering*. LNCS 5421, pp. 124–137. Springer, 2009.

[DXC+11] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, F. Orejas. From State- to Delta-Based Bidirectional Model Transformations: the Symmetric Case. In *Proc. of MODELS 2011*. Volume 2050685, pp. 304–318. Springer, 2011.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In *Proc. of FASE 2007*. LNCS 4422, pp. 72–86. Springer, 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[GEH11] U. Golas, H. Ehrig, F. Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. In *Proceedings of the International Workshop on Graph Computation Models (GCM'10)*. ECEASST 39, p. 149. European Association of Software Science and Technology, 2011.

[GH09] H. Giese, S. Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical report, Hasso-Plattner Institute at the University of Potsdam, 2009.

[GLO09] E. Guerra, J. D. Lara, F. Orejas. Pattern-Based Model-to-Model Transformation : Handling Attribute Conditions. In *Proc. of ICMT 2009*. LNCS 5563, pp. 83–99. Springer, 2009.

[GLO10] E. Guerra, J. D. Lara, F. Orejas. Controlling Reuse in Pattern-Based Model-to-Model Transformations. In *Graph Transformations and Model Driven Enginering*. LNCS 5765, pp. 175–201. Springer, 2010.

[HEO+11] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In *Proc. of MODELS 2011*. LNCS 6981, pp. 668–682. Springer, 2011.

[HHI+11] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. In Alexander et al. (eds.), *Proc. of ASE 2011*. Pp. 480–483. IEEE, 2011.

[KLKS10] F. Klar, M. Lauder, A. Königs, A. Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In *Graph Transformations and Model Driven Enginering*. LNCS 5765, pp. 141–174. Springer, 2010.

[Kön08] A. Königs. *Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation*. PhD thesis, Technische Universität Darmstadt, 2008.

[KRW04] E. Kindler, V. Rubin, R. Wagner. An Adaptable TGG Interpreter for In-Memory Model Transformations. In *Proceedings of the 2nd International Fujaba Days*. Pp. 35–38. 2004.

[KW07] E. Kindler, R. Wagner. Triple Graph Grammars : Concepts , Extensions , Implementations , and Application Scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[LSRS10] M. Lauder, M. Schlereth, S. Rose, A. Schürr. Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. *Bulletin of the Polish Academy of Sciences, Technical Sciences* 58(3):409–422, 2010.

[NCEF02] C. Nentwich, L. Capra, W. Emmerich, A. Finkelstein. Xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology* 2(2):151–185, 2002.

[OL10] F. Orejas, L. Lambers. Symbolic Attributed Graphs for Attributed Graph Transformation. In *Proc. of GraMot 2010*. Volume 30. ECEASST, 2010.

[OMG05] OMG. MOF QVT Final Adopted Specification. 2005.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of WG 1994*. LNCS 903, pp. 151–163. Springer, 1994.

[Ste08] P. Stevens. A Landscape of Bidirectional Model Transformations. In *Proc. of GTTSE 2007*. LNCS 5235, pp. 408–424. Springer, 2008.

[Ste10] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and Systems Modeling* 9(1):7–20, 2010.

[Wag09] R. Wagner. *Inkrementelle Modellsynchronisation*. PhD thesis, Universität Paderborn, 2009.

[YAG08] T. Yokoyama, H. Axelsen, R. Glück. Principles of a Reversible Programming Language. In *Proc. of CF 2008*. Pp. 43–54. ACM, 2008.