Modularizing Triple Graph Grammars Using Rule Refinement

Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr

Technische Universität Darmstadt,
Real-Time Systems Lab, Germany
`surname@es.tu-darmstadt.de`

Abstract. Model transformation plays a central role in Model-Driven Engineering. In application scenarios such as tool integration or view specification, bidirectionality is a crucial requirement. Triple Graph Grammars (TGGs) are a formally founded, bidirectional transformation language, which has been used successfully in various case studies from different applications domains.

In practice, supporting the maintainability of TGGs is a current challenge and existing modularity concepts, e.g., to avoid pattern duplication in TGG rules, are still inadequate. Existing TGG tools either provide no support at all for modularity, or provide limited support with restrictions that are often not applicable.

In this paper, we present and formalize a novel modularity concept for TGGs: *Rule refinement*, which generalizes existing modularity concepts, solves the problem of pattern duplication, and enables concise, maintainable specifications.

Keywords: model transformation, triple graph grammars, modularity.

1 Introduction and Motivation

Model-Driven Engineering (MDE) is an established, viable means of coping with the increasing complexity of modern software systems, promising an increase in productivity, interoperability and a reduced gap between problem and solution domains. *Model transformation* plays a central role in MDE and *bidirectionality* is often a crucial requirement especially in application scenarios that require model synchronization such as tool integration and view specification [3].

Triple Graph Grammars (TGGs) [9] are a rule-based, formally founded technique of specifying a consistency relation between models in a source and target domain, which allows for bidirectional model transformation. TGG rules consist of patterns representing the precondition and postcondition of a change to a model and are fully declarative, i.e., no control flow or similar constructs can be used to specify exactly *how* the change should be realized. In contrast to, e.g., *programmed* graph transformations, TGGs, therefore, require a *rule* structuring mechanism to avoid redundancy, i.e., identical patterns in multiple rules. When TGGs with a considerable number of rules are required, supporting productivity and maintainability becomes crucial.

As initially presented by Klar et al. [7], a viable means of addressing these challenges is to avoid pattern duplication in TGG rules by reusing rule fragments. Existing modularity concepts [7,5], however, pose strong restrictions on the way rules can be reused. Examples for such restrictions include: (i) that the context of a basis rule (the rule to be reused by refining it appropriately) can only be extended but not changed, and (ii) a lack of support for multiple basis rules. Our observation is that these restrictions are too strong and thus prevent reuse in many cases, especially in combination with the limited support for modularity on the metamodel level provided by EMF/Ecore.

Our contribution in this paper is to:

1. Present a novel flexible concept of *rule refinement* for TGG rules as a generalization of previous work by Klar et al. [7] and Greenyer et al. [5]. This is done intuitively in Sect. 2 with a running example.
2. Compare our approach with [7,5] and explain in detail why the generalizations we suggest are necessary. This is done in Sect. 3, where we discuss related existing modularity concepts for TGGs and graph transformations.
3. Provide a comprehensive formalization of rule refinement in Sect. 4.

We conclude with a summary and an overview of areas of future work in Sect. 5.

2 Rule Refinements for TGGs

Our running example is inspired by the families to persons transformation example in the ATL transformation zoo[1]. It represents a tool integration scenario, e.g., between the residents registration office and the tax office of a city.

Figure 1 depicts the triple of *source*, *correspondence*, and *target* metamodels for the transformation, referred to as a *TGG schema*. The source metamodel (left of Fig. 1) comprises a FamilyRegister, which contains multiple Families. A Family consists of Members, which play the role of a son, father, mother, or daughter in the family as indicated by the *references* connecting Family with Member.

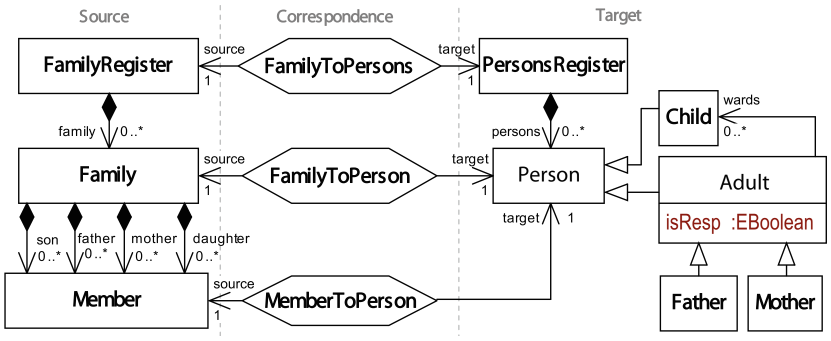


Fig. 1. TGG schema for the running example

[1] `http://www.eclipse.org/atl/atlTransformations/#Families2Persons`

The target metamodel (right of Fig. 1) comprises a PersonsRegister containing Persons. A Person is either a Child, or an Adult, i.e., a Father or Mother. Although the concept of a family is not explicitly present in the target metamodel, an Adult can be responsible for a number of children (isResp attribute in Adult and wards reference to Child) as this is relevant for tax calculation. The source and target metamodels are connected by a correspondence metamodel (hexagonal elements in Fig. 1) specifying which source and target types correspond to each other.

In addition to a TGG schema, a TGG consists of *TGG rules* that describe how triples of source, correspondence and target models are built-up simultaneously. Figure 2 depicts three of the seven rules implementing the families to persons transformation. A TGG rule consists of *elements* (nodes and edges). Nodes are depicted as label:Type, e.g., family:Family, while edges are depicted as Type without labels. Elements *created* by a TGG rule are depicted as green nodes/edges with a "++" markup, while *context* elements are depicted as black nodes/edges without any markup and must be present for the rule to be applied.

The TGG rule r_1: FamilyToPersonsRule creates a family register and a persons register simultaneously and connects them appropriately with a correspondence link. The TGG rules r_2: FamilyToFatherRule and r_3: MemberToFatherRule specify how fathers are handled: According to r_2, a family with a father Member corresponds to a Father in a PersonsRegister, if the Father is responsible for children (isResp := true in the node person:Father). Note that the created father and family are connected with a FamilyToPerson correspondence. In contrast, r_3 creates a Father that is not responsible for any children and requires, therefore, an adult who corresponds to the family as context.
The remaining rules of the TGG for the running example are:
r_4*: FamilyToMotherRule*, which is identical to r_2 but creates a Mother instead of a Father in the target model and connects the created Member via the mother reference instead of father.
r_5*: MemberToMotherRule*, identical to r_3 in an analogous manner as r_4 to r_2.
$r_{6/7}$*: MemberToSonRule* and MemberToDaughterRule, which are both identical to r_3 but connect the created Member to the Family via the son/daughter reference instead of the father reference in the source model. Furthermore, the rules create a Child instead of a Father in the target model, connecting the Child to the responsible adult via the wards reference.

Looking closer at the rules r_2 and r_3, one can observe that r_3 is a copy of r_2 with an additional element in the target domain and a few changes (some elements are required as context instead of being created and the attribute assignment is adjusted). Similar to code duplication in programs, such *pattern duplication* in the rules of a TGG has an averse effect on productivity and maintainability. For our running example, pattern duplication increases for the remaining rules $r_4 - r_7$ turning the rule specification process into an error-prone copy-and-paste task. Changing the transformations now implies multiple changes in different rules resulting in a maintenance nightmare, which gets worse with time as the relationships between rules is not explicit, i.e., new developers cannot know what must be adjusted. To avoid pattern duplication, a means of *reusing*

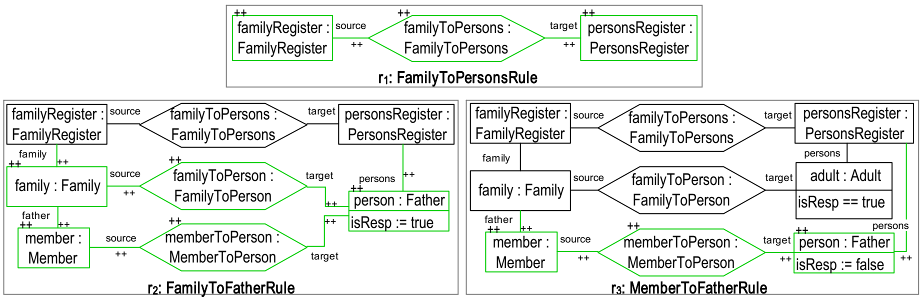


Fig. 2. TGG rules for handling fathers without rule refinements

common patterns in multiple rules is required. In addition, the reuse mechanism must be flexible enough to handle cases where the common pattern is not exactly the same but is only slightly changed. Our concept of rule refinements addresses this challenge by providing a concise pattern language with which higher-order transformations (using rule patterns to transform rule patterns) can be specified.

Figure 3 depicts the complete TGG for the running example using *rule refinements*. The TGG is now represented as an acyclic network of rules, with a

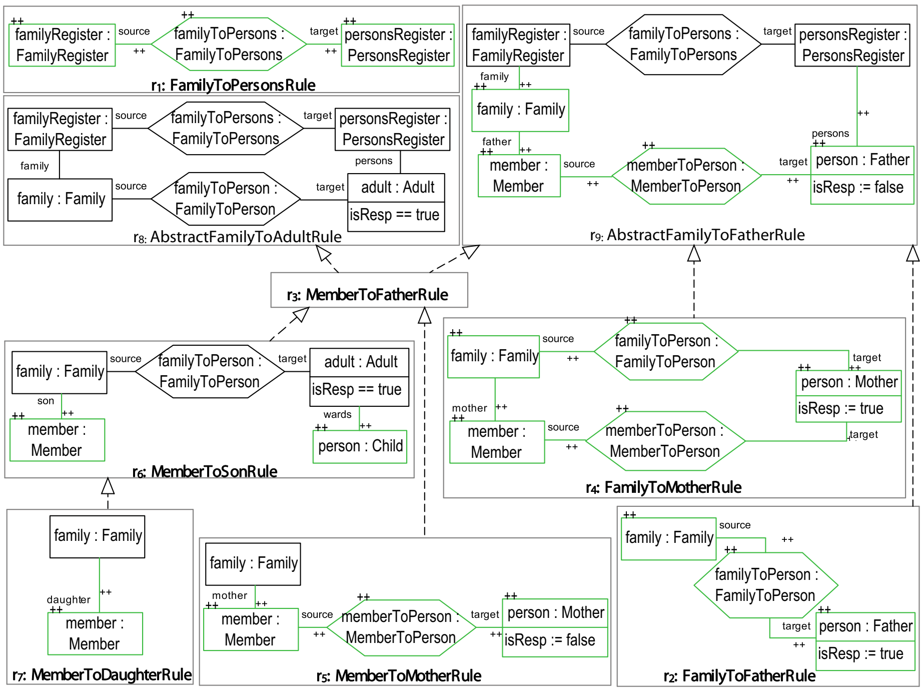


Fig. 3. Refinement network for complete TGG from running example

refinement relation depicted as dashed arrows between rules, e.g., from r_2 to r_9. The rule r_9 is referred to as the *basis rule* of r_2, which in turn *refines* r_9. The refinement network depicted in Fig. 3 is *resolved* to a TGG as follows. A rule r without a basis rule is trivially resolved to $r()$ with exactly the same elements as r. Parentheses indicate that the rule is "resolved" by refining its basis rules. This is the case for r_1, r_8 and r_9, which are resolved to $r_1(), r_8()$ and $r_9()$ in this manner.

The rule r_2 now has a resolved basis $r_9()$ and is further resolved to $r_2(r_9())$ by adding all elements from r_2 to $r_9()$, replacing nodes with the same label. In a similar manner, r_4 is resolved to $r_4(r_9())$. The following three points are to be noted here: Firstly, person: Father in $r_9()$ is replaced by person: Mother, showing that types of elements can be changed when refining, if all edges in the resolved basis rule can be reconnected to the new element. Secondly, the father edge between family and member in $r_9()$ is deleted as it is not in r_4. Finally, a mother edge between family and member is created as it is in r_4 but not in $r_9()$.

Resolving r_3 involves *multiple refinement* as it refines both r_8 and r_9. This is accomplished by *merging* all resolved basis rules to a single basis rule, which is then refined as usual. In this case, $\oplus(r_8(), r_9())$ is constructed by merging elements with the same label together, e.g., familyRegister from $r_8()$ is merged with familyRegister in $r_9()$ to form the same element in $\oplus(r_8(), r_9())$. Note that infix notation is not used here as the merge operator is n-ary. Elements such as adult and person that cannot be identified with a counterpart are added directly to $\oplus(r_8(), r_9())$ as new elements. Note that (i) merging is only possible if the types of identified elements are exactly the same and (ii) context elements have priority over created elements, i.e., family in $r_8()$ and family in $r_9()$ are identified with each other and merged to a context variable family in $\oplus(r_8(), r_9())$. This means that the stronger precondition is taken for the merged rule. The resolved rule $r_3(\oplus(r_8(), r_9()))$ is then constructed as usual with r_3 as refining rule and $\oplus(r_8(), r_9())$ as its resolved basis rule. As r_3 contains no elements, there is nothing else to be done.

Rules r_5, r_6, and r_7 are resolved to $r_5(r_3(\oplus(r_8(), r_9()))), r_6(r_3(\oplus(r_8(), r_9())))$, and $r_7(r_6(r_3(\oplus(r_8(), r_9()))))$ by replacing and creating elements as already described above. The final TGG consists of resolved rules $r_1(), r_2(r_9()), \ldots, r_7(\ldots)$, excluding the *abstract* rules r_8 : AbstractFamilyToAdultRule and r_9 : AbstractFamilyToFatherRule (italicized in Fig. 3).

Our concept of rule refinement helps to avoid pattern duplication in rules by enabling a flexible composition and reuse of (sub)patterns. For our running example, although 9 rules are now required instead of 7, due to 2 extra abstract rules (r_8 and r_9), the total sum of elements in the rules is reduced from 117 to 65, i.e., almost a 50% reduction of the required number of elements. Current industrial projects with $50 - 100$ TGG rules and an average of $15 - 20$ elements per rule would hardly be tractable without using refinements.

3 Related Work

In the following, we compare our approach to existing modularity concepts for TGGs in particular and graph transformation in general. We refer to [11] for a broad survey of modularity concepts for model transformation languages.

Modularity Concepts for TGGs: Klar et al. [7] introduce a reuse mechanism for TGGs, which avoids pattern duplication by allowing rules to *refine* a basis rule. Greenyer et al. [5] extend this idea by introducing *reusable nodes*, i.e., nodes in TGG rules that can be created or parsed as context as required. As this can be simulated with our rule refinement concept, our approach can be viewed as a generalization of [7,5] with the following extensions:

1. We support and formalize *multiple* basis rules, i.e., multiple refinement, which is crucial for a flexible composition of modular TGG rules.
2. In the approach of [7], every rule can only create a single distinct correspondence type. This leads to a confusing mix of two different and orthogonal concepts: (i) Support for inheritance and abstract types in the metamodels (especially the correspondence metamodel) according to [4], and (ii) Refinement of TGG rules. In our approach, this restriction is removed completely; both reuse concepts are clearly separated and can be combined freely.
3. Rather strong restrictions are posed in [7,5] to guarantee the property that a basis TGG rule is always applicable when its refining rules are. We have decided to lift these restrictions as: (i) TGGs are usually *operationalized* to derive, e.g., forward and backward transformations. The mentioned property does *not* apply to these operational scenarios in general and is thus of questionable use in practice. (ii) The approach in [7] is formulated for MOF2 which supports advanced modularity concepts such as inheritance on *edge types*. The *de facto* standard EMF/Ecore is simpler in this respect and, as a consequence, requires a more flexible modularity concept for rules.
4. Both approaches use some form of rule priorities to resolve ambiguities caused by conflicts between basis and refining rules. As neither approach employs backtracking due to efficiency reasons, this can either lead to wrong decisions [5], or requires the user to constantly adjust priorities as rules are added and changed [7]. To resolve such conflicts, we utilize instead a *look-ahead* [8] as a form of application condition, which simulates rule application to detect obvious dead-ends in the transformation. We are thus able to handle a well-defined class of TGGs without backtracking or user intervention.

Modularity Concepts for Graph Transformations: There are numerous modularity concepts in the mature field of graph transformation. The concept of *variable nodes* in rules [6], which can be expanded to instantiate concrete rules, leads to "template" rules and requires separate, explicit expansion rules. Compared to our approach, this increases flexibility but also complexity. A related approach is amalgamation [2], where fragments of a rule can be denoted as being allowed to be matched arbitrarily many times. In this manner, a single rule can be also expanded at runtime by matching such fragments as *often as necessary.*

4 Formalization of Rule Refinements

The basic idea is to establish a suitable and compact language for describing *rule refinements*, i.e., the changes required to produce a new rule from a set of basic rules. We first of all define the syntax of the language, which is chosen to fit to the existing TGG syntax for rules, and specify how a *rule refinement* is decomposed into a set of *primitive transformation steps.* The semantics of rule refinement is then given by executing these primitive (atomic) transformations in a certain sequence to yield the corresponding higher-order (refinement can be seen as rewriting of triple rules) model transformation. Furthermore, refinements can be composed into complex *networks* with support for multiple refinement and abstract rules. For presentation purposes and due to space limitations, we focus in the following discussion on formal details necessary for rule refinement for TGGs, omitting details concerning, e.g., attribute manipulation, inheritance, and negative application conditions. We refer to [1,4,8] for further details.

4.1 Preliminaries: Models, Metamodels and Model Transformation

Models and metamodels are formalized as graphs, with a *conforms to* relationship between a model and its metamodel represented by a structure preserving map, i.e., a graph morphism *type* from a *typed graph* to its *type graph.*

Definition 1 (Typed Graph and Typed Graph Morphism)
A graph $G = (V, E, s, t)$ *consists of a finite set* V *of nodes and a finite set* E *of edges, and two functions* $s, t : E \to V$ *that assign to each edge source and target nodes, respectively.*

A graph morphism $f : G \to G'$*, with* $G' = (V', E', s', t')$*, is defined as a pair of functions* $f := (f_V, f_E)$ *where* $f_V : V \to V'$*,* $f_E : E \to E'$ *and*
$\forall e \in E : f_V(s(e)) = s'(f_E(e)) \ \wedge \ f_V(t(e)) = t'(f_E(e))$.

A type graph *is a distinguished graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.
A typed graph *is a pair* $(G, type)$ *of a graph* G *and a graph morphism* type*:* $G \to TG$.

Given $(G, type)$ *and* $(G', type')$*,* $f : G \to G'$ *is a* typed graph morphism *iff* $type = type' \circ f$.[2] *The set of all graphs of type* TG *is denoted as* $\mathcal{L}(TG)$.

The following definition provides a rule-based, declarative formalization for model transformation. Changes to a model are represented as a *rule*, i.e., a pair of graphs representing the state of the model before and after the transformation.

Definition 2 (Monotonic Creating Rule, Graph Grammar). *Given a type graph* TG*, a* monotonic creating rule $r = (L, R)$ *consists of a pair of typed graphs* $L, R \in \mathcal{L}(TG)$*, with* $L \subseteq R$.[3] *A* graph grammar $GG := (TG, \mathcal{R})$ *consists of a type graph* TG *and a set* $\mathcal{R}$ *of monotonic creating rules.*

As TGG rules describe the simultaneous evolution of *triples of typed graphs*, all concepts are generalized accordingly. In the following, plain letters such as G denote *typed triple graphs*, whereas letters with a subscript such as G_S denote single *typed graphs.*

[2] $f \circ g$ denotes the morphism obtained by composing f and g and reads “f after g”.
[3] $L \subseteq R$ denotes $L \xrightarrow{r} R$, where r is an *injective* typed graph morphism.

Definition 3 (Typed Triple Graph, Typed Triple Graph Morphism) *A* triple graph $G := G_S \xleftarrow{\gamma_S} G_C \xrightarrow{\gamma_T} G_T$ *consists of typed graphs* $G_X \in \mathcal{L}(TG_X)$*,* $X \in \{S, C, T\}$*, and morphisms* $\gamma_S : G_C \to G_S$ *and* $\gamma_T : G_C \to G_T$*.*

Given a triple graph $H = H_S \xleftarrow{\gamma'_S} H_C \xrightarrow{\gamma'_T} H_T$*, a* triple morphism $f := (f_S, f_C, f_T) : G \to H$*, is a triple of typed morphisms* $f_X : G_X \to H_X$*,* $X \in \{S, C, T\}$*, s.t.* $f_S \circ \gamma_S = \gamma'_S \circ f_C$ *and* $f_T \circ \gamma_T = \gamma'_T \circ f_C$*.*

A type triple graph *is a triple graph* $TG = TG_S \xleftarrow{\Gamma_S} TG_C \xrightarrow{\Gamma_T} TG_T$*.*

A typed triple graph *is a pair* $(G, type)$ *of a triple graph* G *and triple morphism* $type : G \to TG$*.*

Given $(G, type)$ *and* $(G', type')$*,* $f : G \to G'$ *is a* typed triple graph morphism *iff* $type = type' \circ f$*.* $\mathcal{L}(TG)$ *denotes the set of all triple graphs of type TG.*

Definition 4 (Triple Rules, Triple Graph Grammar (TGG)) *Given a type triple graph TG, a* triple rule $r = (L, R)$ *is a monotonic creating rule, where* $L, R \in \mathcal{L}(TG)$*, and* $L \subseteq R$*.*

A triple graph grammar $TGG := (TG, \mathcal{R})$ *is a pair consisting of a type triple graph TG and a finite set* $\mathcal{R}$ *of triple rules.*

Example 1. The TGG schema for our running example depicted in Fig. 1 is, according to our formalization, a type triple graph. The TGG rule r_2 depicted in Fig. 2 is a triple rule, i.e., a pair of typed triple graphs (L_{r_2}, R_{r_2}) where L_{r_2} consists of all black elements and R_{r_2} of all black *and* green ("++") elements.

Although TGGs can be used directly to generate triples of consistent models, e.g., for test generation, TGGs are often *operationalized* in practice to derive a pair of unidirectional forward and backward transformations for bidirectional model transformation. As our concept of rule refinement is completely resolved at compile time, details of TGG operationalization are not necessary to understand our formalization and are omitted. We refer to [8] for further details.

4.2 Syntax of Refinements

We now formalize the syntactic structure of a *refinement*, which consists of two triple rules connected in such a manner that it is clear which elements are to be deleted, replaced, or newly created. We take a compositional approach and define a series of refinement *primitives*, representing executable atomic modifications to the basis rule. Complex refinements are composed by combining these primitives.

Definition 5 (Refinement). *A* refinement $\Delta(r^*, r)$ *consists of two triple rules* $r^* = (L^*, R^*)$ *and* $r = (L, R)$*, connected by triple morphisms* $\delta_L, \delta_{L^*}, \delta_R, \delta_{R^*}$ *and typed triple graphs* Δ_L, Δ_R*, with* $\Delta_L \subseteq \Delta_R$*, such that the diagram depicted on the right commutes. The rule* r^* refines *its* basis rule r*. Note that* $\delta_L, \delta_{L^*}, \delta_R, \delta_{R^*}$ *are not necessarily typed.*

$$\begin{tikzcd}
L \arrow[r] & R \\
\Delta_L \arrow[u, "\delta_L"] \arrow[r] \arrow[d, "\delta_{L^*}"] & \Delta_R \arrow[u, "\delta_R"] \arrow[d, "\delta_{R^*}"] \\
L^* \arrow[r] & R^*
\end{tikzcd}$$

Definition 6 (Refinement Network). *A* Refinement Network *is an acyclic graph $\mathcal{N}(V, E, s, t)$ where each node $n \in V$ in the network is a triple rule and each edge $e \in E$ indicates that $s(e)$* refines *$t(e)$ in the sense of Def. 5.*

Definition 7 (DeleteEdge). *A* DeleteEdge *source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams depicted in Fig. 4 below.* DeleteEdge *target refinements are defined analogously, i.e., with non-trivial components only in the target components of $L, R, \Delta_L, \Delta_R, L^*$, and R^*.*

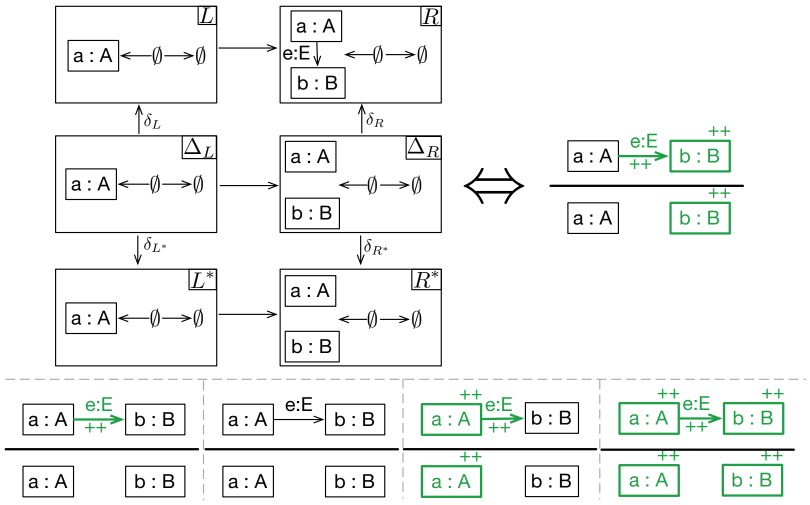


Fig. 4. DeleteEdge source refinements

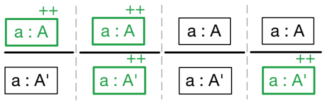
The first *DeleteEdge* diagram is depicted in both a detailed syntax to the left, and an equivalent compact syntax to the right. In the detailed syntax, elements in the typed graphs are denoted by label:type giving a label for the element and its type. The graph morphisms $\delta_{L_S}, \delta_{L_S^*}, \delta_{R_S}, \delta_{R_S^*}$, depicted as arrows, are given by requiring all element labels to be unique in each graph and mapping equally labelled nodes (not necessarily of the same type) to each other, and equally labelled edges *of the same type* to each other. In the compact syntax, only non-trivial graphs are shown (in this case only the source components). The basis rule is placed above the black horizontal line, while the refining rule is placed below. Elements in $R_S \setminus L_S$ are annotated with a "++" markup[4] to differentiate them from elements in L_S. This allows for a compact notation, which is used for all other cases. Fig. 4 depicts in sum five different diagrams for *DeleteEdge*.

Definition 8 (CreateEdge). *A* CreateEdge *source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams depicted in Fig. 4 but with the roles of L/L^* and R/R^* exchanged.* CreateEdge *target refinements are defined analogously.*

[4] Additionally emphasized by depicting them in green instead of black.

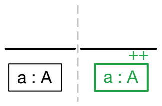
Example 2. Consider the refinement $\Delta(r_4, r_9())$ in Fig. 3. In this case, the edge *father* in r_9 is removed via a *DeleteEdge* primitive, while the edge *mother* in r_4 is added via a *CreateEdge*. We denote this in the following as *DeleteEdge(father)* and *CreateEdge(mother)*, respectively.

Definition 9 (ReplaceNode)
A ReplaceNode *source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the four diagrams depicted to the right.* ReplaceNode *target refinements are defined analogously.*



Note that the type of the replaced node can be changed in general, i.e., the graph morphisms $\delta_{L_S}, \delta_{L^*_S}, \delta_{R_S}, \delta_{R^*_S}$ are not necessarily type preserving (cf. Def. 5).

Definition 10 (CreateNode). *A* CreateNode *source refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the two diagrams depicted to the right.* CreateNode *target refinements are defined analogously.*



Definition 11 (DeleteCorr). *A* DeleteCorr *refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams in Fig. 5.*

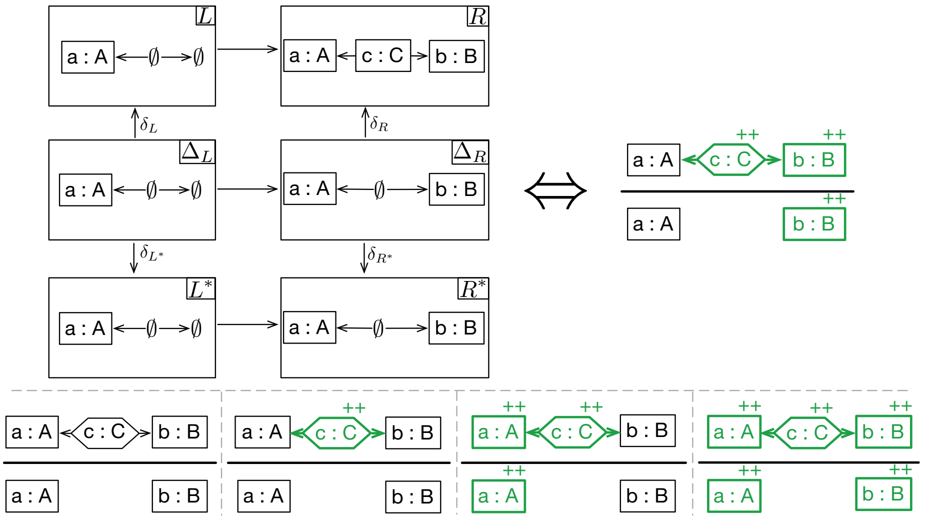


Fig. 5. DeleteCorr refinements

Note that, analogously to Fig. 4, the first *DeleteCorr* refinement is depicted in a detailed formal syntax to the left and a compact syntax to the right. Due to space limitations, the latter is used for the rest of the refinements.

Definition 12 (CreateCorr)
A CreateCorr *refinement is a refinement $\Delta(r^*, r)$, which is isomorphic to one of the five diagrams in Fig. 5 but with the role of L/L^* and R/R^* exchanged.*

Definition 13 (Refinement Primitive)
A refinement primitive *is a* DeleteEdge*,* ReplaceNode*,* CreateNode*,* CreateEdge*,* CreateCorr *or* DeleteCorr *refinement.*

4.3 Semantics of Refinement

To formalize the semantics of our rule refinement concept, we start by defining how a given refinement can be decomposed into primitives:

Algorithm 1. Refinement Decomposition

```
A refinement $\Delta(r^* = (L^*, R^*), r = (L, R))$ can be decomposed into sets $P_S, P_C, P_T$ of
refinement primitives as follows:
  (i) For all nodes $n$ in $V_{R^*_S}$, if $n \notin range(\delta_{R^*_S,V})$ then add a corresponding CreateNode
      to $P_S$, else add a ReplaceNode to $P_S$.
 (ii) For all edges $e^*$ in $E_{R^*_S}$, if $e^* \notin range(\delta_{R^*_S,E})$ then add a CreateEdge to $P_S$.
(iii) For all edges $e$ in $E_{R_S}$, if $e \notin range(\delta_{R_S,E})$ and $s(e) \in range(\delta_{R_S,V})$ and
      $t(e) \in range(\delta_{R_S,V})$, then add a DeleteEdge to $P_S$.
 (iv) Perform steps (i) - (iii) for target components $V_{R^*_T}$, $E_{R^*_S}$, $E_{R_T}$, and $P_T$.
  (v) For all correspondence nodes $c^*$ in $V_{R^*_C}$, if $c^* \notin range(\delta_{R^*_C,V})$, then add a
      CreateCorr to $P_C$.
 (vi) For all correspondence nodes $c$ in $V_{R_C}$, if $c \notin range(\delta_{R_C,V})$ and
      $\gamma_S(c) \in range(\delta_{R_S,V})$ and $\gamma_T(c) \in range(\delta_{R_T,V})$, then add a DeleteCorr to $P_C$.
```

Example 3. From our running example, using the same notation to represent primitives as introduced in Ex. 2, $\Delta(r_4, r_9())$ is decomposed (Alg. 1) to:
$P_S = \{ReplaceNode(family), ReplaceNode(member), DeleteEdge(father),$
$CreateEdge(mother)\}$, $P_C = \{CreateCorr(familyToPerson)\}$,
$P_T = \{ReplaceNode(person)\}$.

Theorem 1 (Completeness of Refinement Decomposition). *Given an arbitrary refinement $\Delta(r^*, r)$, decomposition in sets of refinement primitives P_S, P_C, P_T according to Alg. 1 is possible and unique.*

Proof. (Sketch) Algorithm 1 and induction over sets of nodes/edges in $\Delta(r^*, r)$.

Note that Def. 5 only fixes the *syntax* for $\Delta(r^*, r)$, which is then interpreted (i.e., assigned *semantics*) according to Alg. 1. This is the reason why omitting a node n in r^* does *not* mean it should be deleted, but rather that it is not to be refined in any way and does not induce any refinement primitive.

Algorithm 2 specifies the executable, atomic higher-order transformation each primitive represents. Based on this, we are now able to define the transformation a refinement represents via decomposition in primitives and execution of the primitives in a fixed order (given by the dependencies between primitives).

Algorithm 2. Refinement Primitive Resolution

Given a triple rule $r = (L, R)$, a refinement primitive $\Delta(r^*, r)$ is *resolved* to yield a new rule $r^*(r)$ from r by executing the corresponding higher-order model transformation given in pseudo code as follows (target primitives are handled analogously):

DeleteEdge(e): Remove e from E_{R_S} and, if $e \in E_{L^*_S}$, from E_{L_S}. Adjust source and target functions appropriately by removing entries for e.

CreateEdge(e): Add e to E_{R_S} and, if $e \in E_{L^*_S}$, also to E_{L_S}. Adjust source and target functions appropriately by adding entries for e.

ReplaceNode(n, m): If all incident edges to n can be transferred to m whilst retaining type conformity, remove n from and add m to V_{R_S} (repeat for V_{L_S} if $m \in V_{L^*_S}$). Transfer all incident edges. If this violates type conformity abort (primitive can not be resolved).

DeleteCorr(c): Remove c from V_{R_C} and, if $c \in V_{L^*_C}$, from V_{L_C}. Adjust graph morphisms between source/target and correspondence components appropriately by removing entries for c.

CreateCorr(c): Add c to V_{R_C} and, if $c \in V_{L^*_C}$, also to V_{L_C}. Adjust graph morphisms between source/target and correspondence components appropriately by adding entries for c.

Definition 14 (Refinement Resolution). *A refinement $\Delta(r^*, r)$ is* resolved *to yield a new rule $r^*(r)$ by decomposing it into sets of primitives P_S, P_C, P_T according to Alg. 1 and resolving the primitives (Alg. 2) in the following order:*
(i) All DeleteCorrs *in P_C, (ii) all* DeleteEdges *in P_S and P_T,*
(iii) all ReplaceNodes *in P_S and P_T, (iv) all* CreateNodes *in P_S and P_T,*
(v) all CreateEdges *in P_S and P_T, and finally, (vi) all* CreateCorrs *in P_C.*

The next step on the way to formalizing a network of refinements is to specify how multiple refinement is handled via a merge operator defined on rules.

Definition 15 (Merge Operator $\oplus$)
Given a finite set $\{r_1, r_2, \ldots, r_n\}$ of rules $r_i = (L_i, R_i)$, $r = (L, R) = \oplus(r_1, r_2, \ldots, r_n)$ can be constructed as depicted in the diagram to the right.
$\{L_{1,2,\ldots,n}, \rho_{l_1}, \rho_{l_2}, \ldots, \rho_{l_n}\}$ and $\{R_{1,2,\ldots,n}, \rho_{r_1}, \rho_{r_2}, \ldots, \rho_{r_n}\}$ are constructed as the co-products *of $L_1, L_2, \ldots, L_n$ and $R_1, R_2, \ldots, R_n$, respectively. The typed triple morphism $\mu_R : R_{1,2,\ldots,n} \to R$ must be provided (e.g., via a labelling function) and represents the decision which elements are to be regarded as equal and, therefore, merged in R. L, μ_{L_e} and μ_{L_m} are uniquely fixed by the choice of μ_R.*

$$\begin{tikzcd}
 & & L_{1,2,\cdots,n} \arrow[rr, "\mu_{L_e}"] & & L \arrow[ddd, "\mu_{L_m}"] \\
L_1 \arrow[urr, "\rho_{L_1}"] \arrow[d] & L_2 \arrow[ur, "\rho_{L_2}"] \arrow[d] & \cdots & L_n \arrow[ul, "\rho_{L_n}"'] \arrow[d] & \\
R_1 \arrow[drr, "\rho_{R_1}"'] & R_2 \arrow[dr, "\rho_{R_2}"] & \cdots & R_n \arrow[dl, "\rho_{R_n}"] & \\
 & & R_{1,2,\cdots,n} \arrow[rr, "\mu_R"] & & R
\end{tikzcd}$$

Example 4. From our running example, $\oplus(r_8(), r_9())$ is constructed by building the co-product (disjoint union of edges and nodes) of the left-hand sides of the rules (the black elements). This means $L_{8,9}$ consists of all elements in $r_8()$ (of interest is the node $family$!) and all black elements from $r_9()$. $R_{8,9}$ is constructed analogously and consists of all elements in both rules. The merging morphism μ_R (and thus L, μ_{L_e} and μ_{L_m}) is given implicitly by merging all elements with the same label together. Note that both family nodes are glued together to a single family node, i.e., family is now a node in L as well as R and is, therefore, a context node in the merged rule.

Theorem 2 (Merge Operator is Sound). *The merge operator is commutative w.r.t. its arguments and uniquely defined for a given μ_R (Def. 15).*

Proof. (Sketch) The co-product construction is basically a disjoint union defined for graphs and is commutative. As $L_i \subseteq R_i$ for all rules r_i, it is also easy to show that the choice of μ_R fixes L, μ_{L_e} and μ_{L_m} with standard arguments.

Using the merge operator and refinement resolution, we can now provide an algorithm for resolving a refinement network to a TGG (without refinements):

Algorithm 3. Refinement Network Resolution

A refinement network $\mathcal{N}(V, E, s, t)$ is resolved as follows:

1. Every node r without outgoing edges is regarded as a *resolved* triple rule $r()$.
2. Every node r^* with a single outgoing edge e to a resolved rule $r()$ is regarded as a refinement: $\Delta(s(e), t(e))$.
3. Every node r^* with multiple outgoing edges $e_1, e_2, \ldots, e_k$ to resolved rules $r_1(), r_2(), \ldots, r_k()$ respectively, is regarded as a refinement over the result of merging all rules: $\Delta(r^*, r = \oplus(r_1, r_2, \ldots, r_k))$.
4. Every refinement $\Delta(r^*, r(\ldots))$ is resolved according to Def. 14, transforming the refinement network $\mathcal{N}$ in $\mathcal{N}'$ by removing all from r^* outgoing edges $e_1, e_2, \ldots, e_k$, and replacing r^* with the resolved rule $r^*(r(\ldots))$ in the network.
5. As $\mathcal{N}$ is acyclic, there exists a partial order $k_0, k_1, \ldots, k_l$ in which the network can be transformed with steps (1) – (4) until there are no edges left, i.e., $\mathcal{N} \overset{k_1}{\Rightarrow} \mathcal{N}_1 \overset{k_2}{\Rightarrow} \ldots \overset{k_l}{\Rightarrow} \mathcal{N}_l = (V_{\mathcal{N}_l}, \emptyset)$.
6. A refinement network without any edges is *resolved* and consists only of TGG rules. The final TGG is constructed from a resolved refinement network by excluding all rules that are tagged by the user as being *abstract*.

Theorem 3 (Completeness of Refinement). *A refinement network $\mathcal{N}(V, E, s, t)$ can be resolved to a TGG if all induced* ReplaceNode *primitives are restricted to using type preserving morphisms. If the refinement network can be resolved, the resulting TGG is unique up to isomorphism.*

Proof. The refinement network is acyclic so there exists at least one linearization in which the network can be resolved according to Alg. 3 (decomposition is always possible by Thm. 1). Demanding that all *ReplaceNode* primitives are restricted to using type preserving morphisms ensures that all refinement primitives can be resolved. There might be multiple sortings of the network but the resolution process for a rule r only depends on its transitive dependencies, which are *before* r in any valid sorting. The merge operator is commutative (Thm. 2), so the resulting TGG is independent of the order in which basis rules are resolved.

Example 5. A valid sorting for the refinement network of our running example is: $r_1, r_8, r_9, r_2, r_4, r_3, r_5, r_6, r_7$. The rules r_1, r_8 and r_9 can be resolved to $r_1(), r_8(), r_9()$ with Alg. 3.1. Resulting refinements are $\Delta(r_2, r_9())$, and $\Delta(r_4, r_9())$ according to Alg. 3.2, and $\Delta(r_3, \oplus(r_8(), r_9()))$ according to Alg. 3.3. With Alg. 3.4, these four refinements can be resolved to yield the new nodes $r_2(r_9())$, $r_4(r_9())$, and $r_3(\oplus(r_8(), r_9()))$ removing all outgoing edges from r_2, r_4 and r_3 and replacing r_2, r_4 and r_3 with their resolved versions (Alg. 3.4). The remaining network r_3, r_5, r_6, r_7 is resolved analogously.

4.4 Design Choices vs. Simplifications

In practice, correspondence graphs are often constructed as simple sets of correspondence nodes without any edges. To simplify the discussion in this paper, this common simplification is assumed, i.e., there are no *CreateCorrEdge* primitives. *DeleteNode* and *ReplaceEdge* primitives, however, are omitted on purpose as one could construct confusing refinement networks by introducing and removing nodes arbitrarily in the refinement network via *CreateNode* and *DeleteNode*.

The merge operator requires a typed triple morphism μ_R that decides which elements in the basis rules are to be merged together to result in a single element in the resulting rule. There are different ways to specify this morphism in practice. A user could provide the mapping explicitly by choosing the elements to be merged (in a dialogue or with a textual specification), or the mapping can be indicated implicitly by using equal labels for elements to be merged.

The readability of refinement networks has a considerable effect on usability. Although tool support can provide a "preview" of the complete rules, experience indicates that users actually appreciate the focus on a small section of the rule that is changed with respect to the basis rule. Concerning debugging of refinements, the resulting TGG can already be pretty printed in our textual concrete syntax and an import in our visual modelling environment is in development.

According to the classification of modularization concepts according to [11], our rule refinement is *flattened*, i.e., resolved at compile time. This means that the dynamic semantics of TGGs with respect to the resulting TGG is neither changed nor affected by using refinements. At first sight this might seem inefficient, why not use the information concerning rule similarities to control the choice of rules and possibly reduce unnecessary pattern matching? A similar challenge is also relevant in the context of incremental pattern matching and has already been analyzed in detail. We plan to employ the algorithm of [10] to detect rule similarities and enable efficient pattern matching even in cases where the extra information from refinements is not available or is insufficient (e.g., for weakly typed metamodels).

5 Conclusion and Future Work

In this paper we have introduced and formalized *rule refinement* as a pragmatic modularization concept for TGGs. Our approach generalizes existing work providing support for multiple refinement and increased flexibility as required for

EMF/Ecore. Although we focus in this paper on TGGs, our approach can be transferred to (transformation) languages with rules consisting of graph patterns.

An implementation of rule refinement as proposed in this paper is integrated in the current version of our metamodelling and model transformation tool eMoflon.[5] As future work we plan to improve readability by providing a visualization of the flattened TGG, which can be produced on demand. We plan to analyze our existing collection of TGGs to develop a catalogue of *bad smells*, and a set of systematic *refactorings*, which can be used to introduce refinement and reduce pattern duplication in existing TGG rules. A further important extension is to generalize our concept to refinement between complete TGGs, i.e., with primitives such as *AddRule* or *ReplaceRule*.

References

1. Anjorin, A., Varró, G., Schürr, A.: Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. In: Hermann, F., Voigtländer, J. (eds.) BX 2012. ECEASST, vol. 49. EASST (2012)
2. Biermann, E., Ehrig, H., Ermel, C., Golas, U., Taentzer, G.: Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 121–140. Springer, Heidelberg (2010)
3. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
5. Greenyer, J., Rieke, J.: Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 222–237. Springer, Heidelberg (2012)
6. Hoffmann, B., Janssens, D., Van Eetvelde, N.: Cloning and Expanding Graph Transformation Rules for Refactoring. In: ENTCS, vol. 152, pp. 53–67 (2006)
7. Klar, F., Königs, A., Schürr, A.: Model Transformation in the Large. In: Crnkovic, I., Bertolino, A. (eds.) FSE 2007, pp. 285–294, No. 594074. ACM (2007)
8. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
9. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
10. Varró, G., Deckwerth, F.: A Rete Network Construction Algorithm for Incremental Pattern Matching. In: Duddy, K., Kappel, G. (eds.) ICMT 2013. LNCS, vol. 7909, pp. 125–140. Springer, Heidelberg (2013)
11. Wimmer, M., et al.: A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 31–46. Springer, Heidelberg (2011)

[5] `www.emoflon.org`