GTS Families for the Flexible Composition of Graph Transformation Systems

Steffen Zschaler[1(✉)] and Francisco Durán[2]

[1] Department of Informatics, King's College London, London WC2R 2LS, UK
szschaler@acm.org
[2] Dpto. de Lenguajes y Ciencias de la Computación,
University of Málaga, Málaga, Spain
duran@lcc.uma.es

Abstract. Morphisms between graph-transformation systems (GTSs) have been successfully used for the refinement, reuse, and composition of GTSs. All these uses share a fundamental problem: to be able to define a morphism, source and target GTSs need to be quite similar in their structure (in terms of both the type graphs and the set of rules and their respective structures). This limits the applicability of these approaches by excluding a wide range of mappings that would intuitively be accepted as meaningful, but that cannot be captured formally as a morphism. Some researchers have attempted to introduce some flexibility, but these attempts either focus only on the type graphs (e.g., Kleisli morphisms between type graphs) or only support specific forms of deviation (e.g., supporting sub-typing in type graphs through clan morphisms). In this work, we introduce the notion of GTS families, which provide a general mechanism for explicitly expressing the amount of acceptable adaptability of the involved GTSs so that the intended morphisms can be defined. On this basis, we demonstrate how GTS families that are extension preserving can be used to enable flexible GTS amalgamation.

1 Introduction

Graph transformation systems (GTSs) were proposed in the late seventies as a formal technique for the rule-based specification of the dynamic behaviour of systems [1]. Since then, GTSs have been used in different contexts in computer science, including the formalisation of systems, programming languages and model-driven engineering.

In the many contexts in which GTSs have been used, a key ingredient for exploiting their power is that of GTS morphisms. GTS morphisms have been used for different purposes in system specification, for instance, to characterise the relationship between a system and views of it [2], for expressing refinements [3,4], or for modelling import and export interfaces of modules [5]. Recent uses of GTSs in the context of Model-Driven Engineering (MDE) have gone one step further, proposing practical uses of different forms of parametric GTSs for reusing model transformations, and reusing and composing domain specific language

© Springer-Verlag GmbH Germany 2017
M. Huisman and J. Rubin (Eds.): FASE 2017, LNCS 10202, pp. 208–225, 2017.
DOI: 10.1007/978-3-662-54494-5_12

(DSL) definitions. In [6], de Lara and Guerra proposed the use of transformation templates, typed by a graph, that they call *concept*, which can then be instantiated by binding the concept to a concrete graph. Durán et al. proposed in [7,8] what they call *parameterised GTSs*, where the parameter is not just a type graph, but a complete GTS, and where composition of GTSs is based on a GTS amalgamation construction. In the same way concepts gather the structural requirements, the set of rules of parameter GTSs are behavioural requirements over the concrete GTSs used in the instantiation. In each of these cases, using GTS morphisms enables useful syntactic or semantic guarantees to be given. For example, in [6] the use of morphisms means that transformations can be guaranteed to be syntactically reusable. In the case of [8], the use of suitable morphisms enables guarantees on behaviour protection of amalgamated GTSs.

For reuse and composition, the main difficulty is the flexibility of the mechanisms available. In specific domains, *ad hoc* definitions of GTS morphisms have been proposed. For example, an alternative notion of refinement relation is given for transactional graph transformation systems in [9], where a graph typing mechanism induces a distinction between stable and unstable graph items and where implementation morphisms map single productions to whole transactions so that morphisms define simulations. Taentzer [10] uses a subset of UML extended by reconfiguration and import/export view facilities, represented as embedding morphisms, to propose a formal framework for visual modeling of distributed object systems. Component composition is defined by only allowing embedding morphisms between import and export rules where each two rules connected by an embedding morphism are named equally. These specialised solutions do not easily extend to the general setting. The use of GTSs in MDE introduces even more challenges, as we need to consider more complex graphs including attributes and node-type inheritance [11].

The need for a mechanism for relating different GTSs, or their type graphs, that is more flexible than direct morphisms—to broaden opportunities for GTS reuse—has been recognised before: In the case of models, represented as graphs, this has been resolved more or less pragmatically by supporting a specific, fixed set of adaptations to be applied prior to applying the morphism (see, *e.g.,* [6,11–13]). For example, Diskin *et al.* [12] propose using Kleisli categories for relating models. De Lara and Guerra extended their work on concepts in [13] by using adapters to allow heterogeneities between the concept and the concrete graph. Each of these works "hard wires" a specific set of flexibilities.

To support complete GTSs, rules must also be related in a flexible manner. In [4,14], Große-Rhode *et al.* introduce temporal and spatial refinement relations. In a spatial refinement, each rule is refined by an amalgamation (*i.e.,* a parallel composition with sharing) of rules, while in a temporal refinement it is refined by a sequential composition. Engels *et al.* [5] present a framework for classifying and systematically defining GTS morphisms. Different types of morphisms are characterised by their relationship between the behaviours of source and target GTSs. For instance, refinements are a case of behaviour-preserving morphisms, while views are a case of behaviour-reflecting morphisms. Durán *et al.* [8] similarly introduce different behaviour-aware GTS morphisms.

These solutions are, however, far from satisfactory. Even though the introduction of derived attributes and links as in [12] or [13], and the behavioural relations provided for GTS morphisms as in [8] improve the chances of defining the required morphisms, structural mismatch remains a problem. Often even where there is an intuitive match, no morphism can be established. This substantially limits the reuse potential of these approaches. In many cases, a simple restructuring of the GTSs involved could easily allow a valid mapping to be established. However, there is currently no support for capturing such restructurings, and in particular for capturing exactly the set of restructurings the designer of a GTS would consider valid and meaningful.

In this work, we propose the use of GTS transformers to refactor GTSs with the goal of resolving the structural mismatches between source and target GTSs so that GTS morphisms can be defined. In fact, GTS transformers may be seen as re-factoring mechanisms, which provide a general setting for defining adaptations. GTS transformers are functions, and can successively be applied to our source GTS to find the one on which the morphism can be defined. To systematise this, we introduce the notion of *GTS families*. Given a set of transformers T, the T-family of a GTS GTS_0 is the set of GTSs reachable from GTS_0 using the transformers in T. The problem of defining a mapping morphism between a GTS GTS_0 and a target GTS GTS_1 then amounts to finding a GTS in the family of GTS_0 from which the morphism can be defined. This way, the problem becomes a *model-based search problem* [15].

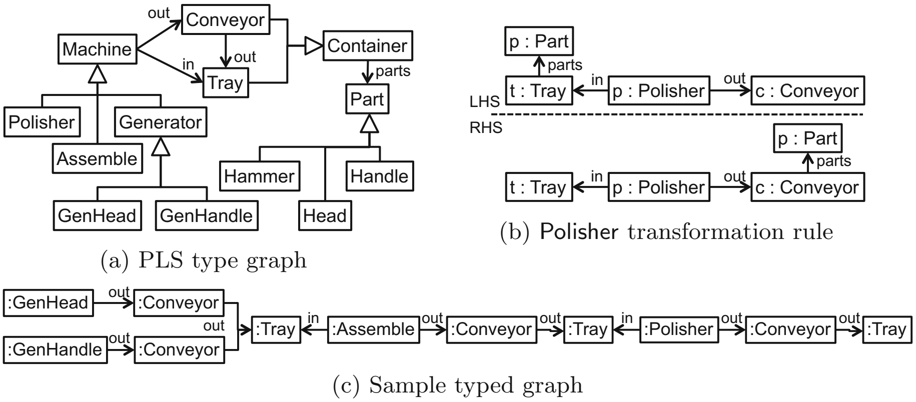
Of course, any mechanisms enabling more flexibility must balance this against the required level of control so that suitable semantic properties can be guaranteed. We provide different transformers and prove that they preserve *extensions* [16] between GTSs. As a result, we show how these transformers can be used to enable flexible composition of GTSs.

The remainder of this paper is structured as follows. In Sect. 2, we introduce a running example and motivate the limitations of GTS morphisms and the need for more flexibility. After providing some formal background on typed attributed graphs, GTSs, and their morphisms in Sect. 3, Sect. 4 introduces the notions of GTS transformers and GTS families as well as three example GTS transformers. In Sect. 4.3, we show that these transformers are *extension preserving* and can, thus, be used to compose GTSs using the mechanism from [16]. We wrap up in Sect. 5 with some conclusions and lines of future research.

2 Running Example

Let us consider a simple production-line system (PLS) GTS, part of which is depicted in Fig. 1. This GTS models a PLS for making hammers out of hammer heads and handles. In the type graph TG_{PLS} in Fig. 1a we find different types of machines, different types of parts, and different containers of parts.[1] The behaviour of such systems is defined through a number of graph-transformation rules, like the one in Fig. 1b, which models the polishing of a part by a polishing

[1] We use the hollow-arrow notation from UML to denote inheritance relationships.

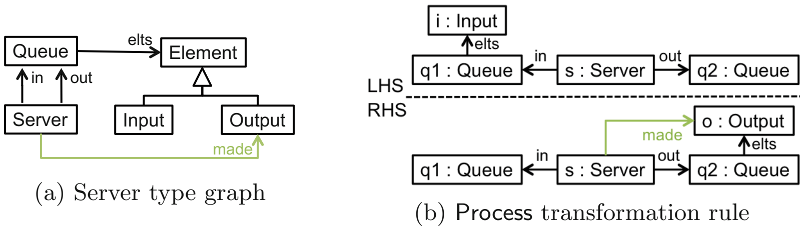


(a) PLS type graph

(b) Polisher transformation rule

(c) Sample typed graph

Fig. 1. Production line system



(a) Server type graph

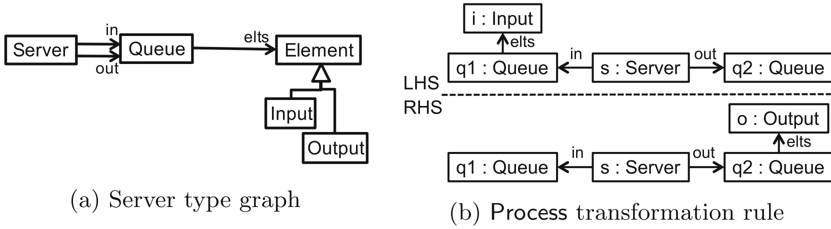
(b) Process transformation rule

Fig. 2. Server tracking system

machine. Other actions, like the generation of head and handle parts in generator machines, the assembling of hammers out of hammer heads and handles, the moving of parts along conveyors, or the collection of parts from final trays is modelled by corresponding rules. A sample graph conforming to such type graph, providing an instance of the system, is shown in Fig. 1c. In it, we can see how machines take parts from input trays and put their outputs in corresponding conveyors, which move parts towards trays.

Let us suppose now that we wanted to keep track of the elements polished in our production line system. Instead of modifying our PLS GTS, we may use the mechanism in [16] to compose it with a generic tracker system, the Tracker GTS defined in Fig. 2. Its type graph $TG_{Tracker}$ is depicted in Fig. 2a. It describes the concepts related to servers that process elements taken from input queues into resulting elements that are placed in output queues. The action of processing an input element is then modelled by the transformation rule in Fig. 2b. The made association allows servers to keep track of all processed instances.

To compose the Tracker and the PLS GTSs following the construction in [16], we take the Tracker GTS as a parameterized GTS. The Server GTS, shown in Fig. 3, defines a generic behaviour, the structural and behavioural requirements the Tracker GTS builds on. Note that we can easily establish an inclusion g to



(a) Server type graph (b) Process transformation rule

Fig. 3. Server system parameter GTS

the Tracker GTS. Then, we can compose the Tracker GTS with the PLS GTS so that polisher machines would keep track of the parts it processes:

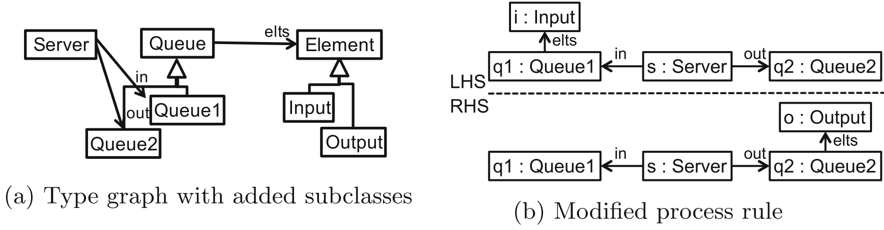
$$\begin{array}{ccc} GTS_0 & \xrightarrow{f} & GTS_2 \\ \Big\downarrow g & & \Big\downarrow \widehat{g} \\ GTS_1 & \dashrightarrow & \widehat{GTS} \end{array}$$

where GTS_0 is the parameter GTS Server, GTS_1 is the parameterized GTS Tracker, and GTS_2 is the PLS GTS used in the instantiation.

For this to work, we need to establish a GTS morphism between the Server GTS (the parameter) and the PLS GTS. Intuitively, the PLS GTS might be seen as a concrete interpretation of the Server GTS—e.g., polishing of parts can be seen as particular case of the server processing. A morphism between the two GTSs would be the formal expression of this. Let us first focus on the type graphs. At first sight, it may seem quite reasonable to define a binding between TG_{Server} and TG_{PLS} by mapping *Server* into Polisher, and Queue into Container,[2] with the in and out associations going to the corresponding ones in the PLS system type graph. However, in TG_{PLS} input and output "queues" are represented by two different types: Tray and Conveyor, respectively. As a result, we cannot establish a valid morphism. Whether source and target queues are of the same type is not actually relevant to our specification of *Server* nor to the definition of tracking. We would like to be able to express the intuition that this particular mapping should be considered valid. Even where we can establish a morphism between the type graphs, there may still be problems establishing morphisms between the rule sets.

In this paper, we introduce the notions of GTS transformers and GTS families and show how they can be used to automatically rewrite the Server and Tracker GTSs in sync to find the GTS depicted in Fig. 4, for which a morphism to the PLS GTS can straightforwardly be defined. Thus, GTS families enable expressing the above intuition about what GTS mappings we would like to allow.

[2] Notice that although we use Queue to name the device in which input and output elements are placed, no specific order is assumed on its elements in the server GTS.



(a) Type graph with added subclasses

(b) Modified process rule

Fig. 4. Modified GTS

3 Preliminaries on GTSs, Clans, and Clan Morphisms

We focus on the double pushout (DPO) approach to graph transformation [3]. In this section, we introduce some of the basic definitions concerning typed graphs and the algebraic approaches to the rewriting of typed graphs, and provide background definitions underpinning our discussion throughout this paper. The notation and most of the definitions in this section follow very closely those in, *e.g.,* [3,8,13,17].

3.1 Graph Transformation Systems

Given some category of graphs and graph morphisms **Graph**, and given a distinguished graph TG, called *type graph*, a TG-*typed graph* (G, g_G), or simply *typed graph* if TG is known, consists of a graph G and a typing homomorphism $g_G : G \to TG$ associating with each vertex and edge of G its type in TG. To enhance readability, we will use simply g_G to denote a typed graph (G, g_G), and when the typing morphism g_G can be considered implicit, we will often refer to a typed graph (G, g_G) just as G. A TG-typed graph morphism between TG-typed graphs $(G_i, g_i : G_i \to TG)$, with $i = 1, 2$, denoted $f : (G_1, g_1) \to (G_2, g_2)$, is a graph morphism $f : G_1 \to G_2$ which preserves types; that is, $g_2 \circ f = g_1$.

A graph transformation rule[3] p is of the form $L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ with graphs L, K, and R, called, resp., left-hand side, interface, and right-hand side, and some kind of monomorphisms (typically inclusions) l and r.

In the DPO approach to graph transformation, the application of a transformation rule $p = L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ to a graph G via a match $m : L \to G$ is constructed as two gluings (1) and (2), which are pushouts in the corresponding graph category, leading to a direct transformation $G \stackrel{p,m}{\Longrightarrow} H$.

$$\begin{array}{ccccc} L & \stackrel{l}{\longleftarrow} & K & \stackrel{r}{\longrightarrow} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

[3] As a simplification, we do not consider application conditions (*cf., e.g.,* [8]).

A graph transformation system (GTS) over a type graph TG is a triple (TG, P, π) where P is a set of rule names and π is a function mapping each rule name p into a rule $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ typed over TG.

Since we are interested in relating GTSs over different type graphs, we need to move graphs and graph morphisms along morphisms. Assuming $\mathbf{Graph}_{TG}$ the category of TG-typed graphs and TG-typed graph morphisms, a graph morphism $f\colon TG \to TG'$ induces *forward and backward retyping* functors $f^{>}\colon \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$ and $f^{<}\colon \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$. Since, as said above, we refer to a TG-typed graph $G \to TG$ just by its typed graph G, leaving TG implicit, given a morphism $f\colon TG \to TG'$, we may refer to the corresponding TG'-typed graph by $f^{>}(G)$. Since we can retype graphs and graph morphisms, we can retype rules. Given a rule p over a type graph TG and a graph morphism $f\colon TG \to TG'$, we will write things like $f^{<}(p)$ and $f^{>}(p)$ denoting, respectively, the backward and forward retyping of rule p.

3.2 Morphisms Between Graph Transformation Systems

Although the mechanisms presented in the following sections may be applicable to most notions of GTS morphisms defined in the literature, to simplify the presentation we will focus on a specific type of rule morphism and GTS morphism. We begin with rule morphisms, relating two graph-transformation rules.[4]

Definition 1. *Given rules $p_i = L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i$, for $i = 0, 1$, a rule morphism $f\colon p_0 \to p_1$ is a tuple $f = (f_L, f_K, f_R)$ of graph monomorphisms $f_L\colon L_0 \to L_1$, $f_K\colon K_0 \to K_1$, and $f_R\colon R_0 \to R_1$ such that the squares with the span morphisms l_0, l_1, r_0, and r_1 are pullbacks, as in the diagram below.*

$$\begin{array}{ccccccc} p_0 & : & L_0 & \xleftarrow{l_0} & K_0 & \xrightarrow{r_0} & R_0 \\ {\scriptstyle f}\downarrow & & {\scriptstyle f_L}\downarrow & pb & {\scriptstyle f_K}\downarrow & pb & \downarrow{\scriptstyle f_R} \\ p_1 & : & L_1 & \xleftarrow[l_1]{} & K_1 & \xrightarrow[r_1]{} & R_1 \end{array}$$

We are now ready to introduce GTS morphisms.[5]

Definition 2. *Given GTSs $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f\colon GTS_0 \to GTS_1$, with $f = (f_{TG}, f_P, f_r)$, is given by a morphism $f_{TG}\colon TG_0 \to TG_1$, a surjective mapping $f_P\colon P_1 \to P_0$ between the sets of rule names, and a family of rule morphisms $f_r = \{f^p\colon f^{>}_{TG}(\pi_0(f_P(p))) \to \pi_1(p)\}_{p \in P_1}$.*

A special kind of GTS morphism is a GTS extension, which is essentially an inclusion such that everything being added to the rules of the extended GTS is typed by elements also added to the type graph.

[4] Similar definitions of rule morphisms can be found in the literature where the squares are pushouts instead of pullbacks, or simply commuting squares (*e.g.,* [18]), or where the relations are between a single rule and a collection of rules (*e.g.,* spatial and temporal refinements [4]). Requiring pullbacks is quite natural though: the intuition of morphisms is that they should preserve the "structure" of objects.

[5] See [5] for a systematic classification of other definitions of GTS morphisms.

Definition 3 ***(GTS Extension*** [16]*). Given GTSs $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$, with $f = (f_{TG}, f_P, f_r)$, is an extension morphism if f_{TG} is a monomorphism and for each $p \in P_1$, $\pi_0(f_P(p)) = f^{<}_{TG}(\pi_1(p))$.*

3.3 Typed Attributed Graphs and Clan Morphisms

Our underlying graphs are attributed graphs typed over attributed type graphs with inheritance [3,11]. In these graphs, attributes are represented as edges between graph nodes and data nodes (captured by the notion of *E-graphs* in [3]). We use symbolic graphs [19,20] to enrich graphs with a set Φ of formulas over a signature $\Sigma = (S, \Omega)$, with S a set of sorts and Ω a set of operations. We assume that each formula is an equality between a variable and its value (grounded symbolic graphs in [19]). For simplicity, we assume attributed graphs on the same signature and omit a treatment of cardinalities and composition relations, which could be given as constraints as discussed in [21]. We refer the interested reader to [19,20] for a more general presentation of symbolic attributed graphs.

An attributed graph $ATG = (TG, \Phi)$ may be used as a type graph. As for any type of graph, a *typed attributed graph* (AG, t) over an attributed type graph ATG consists of an attributed graph AG together with an attributed morphism $t : AG \to ATG$. A typed attributed graph morphism $f : (AG_1, t_1) \to (AG_2, t_2)$ is an attributed graph morphism $f : AG_1 \to AG_2$ such that $t_2 \circ f = t_1$.

To deal with object-oriented systems we need some additional machinery. We follow [11] in defining attributed type graphs with inheritance.

Definition 4. *An attributed type graph with inheritance $ATGI = (ATG, I, Ab)$ consists of an attributed type graph $ATG = (TG, \Phi)$, with an E-graph $TG = (V, E, A, D, s^E, t^E, s^A, t^A)$, a set $I \subseteq V \times V$ of inheritance relations, and a set $Ab \subseteq V$ of abstract nodes.*

The typing of an object diagram with respect to a class diagram is typically represented as a clan morphism [11]. Intuitively, a clan morphism $f : AG \to ATGI$ from an attributed graph AG to an attributed type graph with inheritance $ATGI$ is an attributed graph morphism that takes into account the inheritance relation and abstraction definitions of the target $ATGI$.

Definition 5. *Let $ATG_i = (TG_i, \Phi_{ATG_i})$, with $i = 1, 2$, be attributed type graphs, with $TG_i = (V_{TG_i}, E_{TG_i}, A_{TG_i}, D_{TG_i}, s^E_{TG_i}, t^E_{TG_i}, s^A_{TG_i}, t^A_{TG_i})$, and let $ATGI_2 = (ATG_2, I, Ab)$ be an attributed type graph with inheritance. For each node v in V_{TG_2}, $clan(v) = \{v' \in V_{TG_2} \mid (v', v) \in I^*\}$, with I^* the reflexive and transitive closure of I. Then, given an algebra $\mathcal{A}$, a clan morphism $f : ATG_1 \to ATGI_2$ is an attributed graph morphism $(f_V, f_E, f_A, f_D) : ATG_1 \to ATG_2$ such that*

1. *$\forall e \in E_{TG_1}, f_V(s^E_{TG_1}(e)) \in clan(s^E_{TG_2}(f_E(e)))$ and $f_V(t^E_{TG_1}(e)) \in clan(t^E_{TG_2}(f_E(e)))$, and*

2. $\forall a \in A_{TG_1}, f_A(s^A_{TG_1}(a)) \in clan(s^A_{TG_2}(f_A(a)))$ *and* $f_A(t^A_{TG_1}(a)) = t^A_{TG_2}(f_A(a))$.

Definition 6. *Given* $ATGI_i = (ATG_i, I_i, Ab_i)$*, for* $i = 1, 2$*, attributed type graphs with inheritance, and an algebra* $\mathcal{A}$*, a morphism* $f : ATGI_1 \rightarrow ATGI_2$ *is a clan morphism* $f = (f_V, f_E, f_A, f_D) : ATG_1 \rightarrow ATGI_2$ *that*

1. *preserves the inheritance relation, i.e., if* $(a, b) \in I_1$ *then* $(f_V(a), f_V(b)) \in I_2^*$*,*
2. *reflects subtyping, that is, for each* $(a, b) \in I_2^*$ *with some* $a' \in V_1$ *such that* $f_V(a') = a$*, there must be a* $b' \in V_1$ *such that* $f_V(b') = b$ *and* $(a', b') \in I_1^*$*, where* V_1 *is the node set of* ATG_1*, and*
3. *preserves the abstraction definitions, that is,* $u \in Ab_1 \Leftrightarrow f_D(u) \in Ab_2$*.*

Example 1. The mapping in Sect. 2 between TG_{Server} and TG_{PLS} does not satisfy the conditions to be part of a clan morphism. Specifically, the mapping for the in association fails condition 1 in Definition 5: $f_V(t^E_{Server}(\mathsf{in})) = \mathsf{Container} \notin clan(t^E_{PLS}(f_E(\mathsf{in}))) = \{\mathsf{Tray}\}$.

4 GTS Transformers and Families

Intuitively, a GTS family is a set of GTSs inductively defined from a source GTS GTS_0, capturing exactly the kind of flexibility we would like to permit when mapping GTS_0 to another GTS GTS_1. Given a set of transformers T, that model the different alterations that may be applied on GTSs, we denote by $[GTS_0]_T$ the family of GTS_0 using T. Mappings are then formally defined by selecting one GTS from the GTS family of GTS_0, written $[GTS_0]_T \rightsquigarrow GTS'_0$, and establishing a morphism between GTS'_0 and GTS_1. We first introduce the notion of GTS transformers, before using them to formally define GTS families. We then show how extension preserving transformers can be used to enable the flexible composition of GTSs and how individual members of a GTS family can be identified based on a given target GTS for a mapping.

4.1 GTS Transformers

GTS transformers, and the GTS families we generate with them, generalise the idea of adapters over transformations (also called adaptations in [13,22,23]). We start by defining GTS transformers as transformations between GTSs.[6]

Definition 7 (GTS transformer). *A GTS transformer* t *is a triple of three inter-related transformations* $t = (t_{TG}, t_P, t_\pi)$*:*

t_{TG} *takes GTSs to type graphs;*
t_P *takes GTSs to sets of rule names;*
t_π *takes GTSs to functions mapping rule names to rules.*

[6] In effect, GTS transformers are a form of higher-order transformation [24].

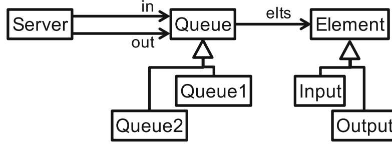


Fig. 5. *IntroSC*-modified type graph

GTS transformers define functions over the set of all ATGI-typed GTSs. Given a GTS $GTS_0 = (TG_0, P_0, \pi_0)$, $t(GTS_0) = (TG_1, P_1, \pi_1)$ *such that:* $TG_1 = t_{TG}(GTS_0)$, $P_1 = t_P(GTS_0)$, $\pi_1 = t_\pi(GTS_0)$, *and for all* $p \in P_1$, $\pi_1(p)$ *is a rule typed over* TG_1.

Note that while the three component functions are defined to range over the entire GTS, they each only transform one aspect of the GTS. For example, t_{TG} will only transform the type graph of the given GTS. However, we define them to range over the entire GTS so that they can ensure consistency of the result.

Remark 1. By definition, given a valid GTS as input, a well-defined GTS transformer will always produce a valid GTS as output.

To make this definition more concrete, we now introduce three examples of transformers, namely $t_{IntroSC}$, $t_{MvAssoc}$ and $t_{InhUnfld}$, which, respectively, add subclasses in the inheritance hierarchy, move associations in its type graph down in the inheritance hierarchy, and specialise rules to particular subclasses.

Definition 8. *The* $t_{IntroSC}$ *transformer modifies the type graph of a GTS by introducing a subclass to a class. It non-deterministically chooses a class from the type graph of the original GTS and adds a subclass with no attributes nor associations. All other classes, attributes, and associations are maintained. The set of rules is not changed.*

Example 2. By repeatedly applying $t_{IntroSC}$ to the Server GTS in Fig. 3 we could obtain, *e.g.*, a GTS Server v1 with the type graph shown in Fig. 5. The set of rules in the new GTS are identical to the rules in the original GTS.

Definition 9. *Given a GTS* $GTS_0 = (TG_0, P_0, \pi_0)$, *the* $t_{InhUnfld}$ *transformer produces a new GTS* GTS_1 *with the same type graph and a rule set resulting of modifying its rules as follows:*

1. *Non-deterministically picks a class* $C \in TG_0$ *that has a number of subclasses* $SC_i \in TG_0$, $i = 1, \ldots, n$;
2. *Non-deterministically picks a rule name* $p \in P_0$ *s.t.* $\pi_0(p)$ *contains objects that are typed by* C;
3. *Non-deterministically picks one subclass of* C *for every free object in* $\pi_0(p)$ *typed by* C *(different subclasses may be picked for different objects);*

4. *Generates a new rule $\pi_1(p)$ using the chosen subclasses to type the corresponding objects;*
5. *Copies all other rules as they are.*

Example 3. Given the Server v1 GTS resulting from the application of the $t_{IntroSC}$ transformer as in Example 2, the application of the $t_{InhUnfld}$ transformer on it may result in the GTS Server v2 with the same type graph and a rule as in Fig. 4b. All other rules remain as in the original GTS.

Definition 10. *Given a GTS, the $t_{MvAssoc}$ transformer produces a GTS with all rules as in the original GTS and where the type graph is modified as follows:*

1. *Non-deterministically picks a class C.*
2. *Non-deterministically picks an association assoc that ends in C.*
3. *Non-deterministically picks a set SC of subclasses of C. At least every subclass S of C for which there is a rule in π_0 where assoc refers to an object typed as S will be included in SC.*
4. *If C and assoc are such that there are no rules that use assoc to refer to an object typed as C, then assoc is removed from the type graph, and a new replica of assoc with the same source as the original is created and defined to point to each $S \in SC$.*

If the type graph modification is not possible, because the condition in Step 4 fails, $t_{MvAssoc}$ returns the original GTS.

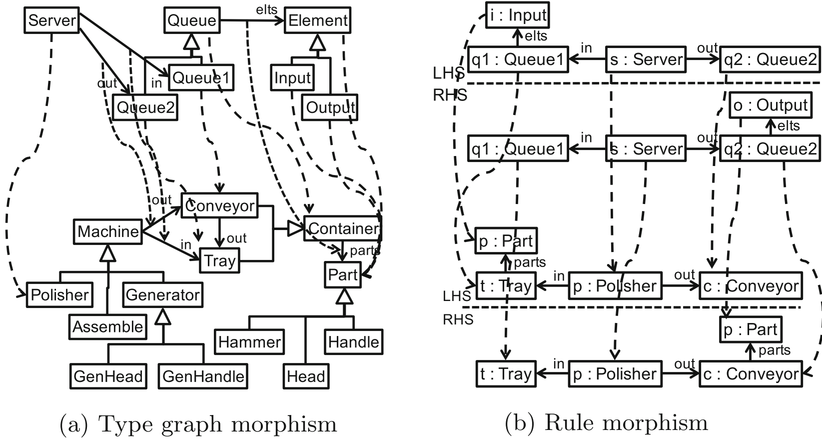
The condition in Step 4 as well as the specific construction of the set SC are required to ensure that $t_{MvAssoc}$ produces a valid and well-typed GTS.

Example 4. Repeatedly applying $t_{MvAssoc}$ to the Server v2 GTS produced in Example 3, may result in a GTS Server v3 with the type graph in Fig. 4a and the rules as those of Server v2 GTS.

The three introduced transformers are just a sample of the kind of transformations we can define on GTSs. For our running example, we have used these transformers to reflect our intuition that the specific types of input and output queues are not important for the behaviour we want to abstract in the Server GTS. As we have mentioned at the beginning of this section, the mismatches may be both in the structure or in the transformation rules, and therefore more sophisticated transformers operating on the type graph and on the rules may be necessary in other cases. In the next subsection we introduce GTS families as a way of packaging the transformers representing our intuition about the behaviour we want to capture.

4.2 GTS Families

Given a set of GTS transformers, new GTSs can be derived from a GTS.



(a) Type graph morphism (b) Rule morphism

Fig. 6. Sketch of the morphism between GTSs Server v3 and PLS

Definition 11. *Given a GTS GTS_0 and a GTS transformer t, a single-step GTS derivation $GTS_0 \Rightarrow_t GTS_1$ is induced iff $GTS_1 = t(GTS_0)$. Given a set $T = \{t_i | i = 1, \ldots, n\}$ of GTS transformers, a GTS derivation of GTS_0 over T ($GTS_0 \Rightarrow_T^* GTS_m$) is given by a, possibly empty, sequence of single-step derivations $GTS_j \Rightarrow_{t_j} GTS_{j+1}$, $j = 0, \ldots, m-1$, $t_j \in T$.*

Example 5. After the application of transformers $t_{IntroSC}$, $t_{InhUnfld}$, and $t_{MvAssoc}$ as in Examples. 2–4, the morphism between GTSs Server v3 and PLS can now be defined, as sketched in Fig. 6.

We call the (possibly infinite) set of all GTSs derivable from GTS_0 over a set of transformers T the T-family of GTS_0.

Definition 12. *Given a set of GTS transformers $T = \{t_i | i = 1, \ldots, n\}$, and a GTS GTS_0, the T-family of GTS_0, denoted by $[GTS_0]_T$, is the (possibly infinite) reflexive-transitive closure of GTS derivations of GTS_0 over T: $GTS'_0 \in [GTS_0]_T \Leftrightarrow GTS_0 \Rightarrow_T^* GTS'_0$.*

We will write $[GTS_0]_T \rightsquigarrow GTS'_0$ to denote the *selection* of some GTS'_0 from the GTSs in $[GTS_0]_T$. Note that $GTS_0 \in [GTS_0]_T$.

The GTS transformers defined above, non-deterministically select elements of the type graph or rules to be modified. In analogy to graph-transformation rules, we allow transformer applications to be guided by providing a (partial) match for these elements. We will refer to the combination of a transformer and a complete match for each of the elements it would otherwise select non-deterministically as a *specific application of a transformer*.

Example 6. Consider the GTS Server v1 in Example 2. It results from the repeated application of the $t_{IntroSC}$ transformer (Definition 9). Specifically, given

the Server GTS we may invoke the application of $t_{IntroSC}$ on class Queue, introducing subclass Queue1, followed again by its application on class Queue to introduce the subclass Queue2.

4.3 Extension Preserving Transformers and GTS Amalgamation

An interesting case, with direct application to parameterized GTSs, is the case of *extensions* (see Definition 3). Parametrization of GTSs establishes an inclusion between the parameter GTS (GTS_0) and the full, parametrized GTS (GTS_1)—for example, see [8]. Typically, these inclusions are extensions.

To improve the possibilities of instantiating a parameterized GTS, we would like to be able to consider as parameter GTS any of the GTSs we can reach using a given set of transformers. In other words, we would like to be able to consider as parameter, not a single GTS, but its entire family. To make this safe, we need to ensure that for any path of transformers $GTS_0 \Rightarrow_T^* GTS_0'$ we can find a corresponding path $GTS_1 \Rightarrow_T^* GTS_1'$ such that the extension $GTS_0 \hookrightarrow GTS_1$ leads to the extension $GTS_0' \hookrightarrow GTS_1'$; that is, a new parametrized GTS preserving the extension. The easiest way of finding such a corresponding path is by constructing it from the same transformer applications in both cases. Transformers for which this can be done, we will call *extension-preserving transformers.*

Definition 13. *A transformer t preserves extensions if for any GTS extension $GTS_0 \hookrightarrow GTS_1$, the exact same specific application of t to GTS_0 and GTS_1 results in an extension as depicted in the following diagram, where the dotted arrow means that the application of the transformer on GTS_1 is exactly the same as the one on GTS_0.*

$$\begin{array}{ccc} GTS_0 & \xrightarrow[t]{} & GTS_0' \\ \downarrow & \vdots & \downarrow \\ GTS_1 & \xrightarrow[t]{} & GTS_1' \end{array}$$

Note that it is enough to prove that each individual transformer is extension preserving for any combination of these transformers to be extension preserving, too. In particular, to be extension preserving, a transformer needs to be applicable both on GTS_0 and on GTS_1 without changes.

Proposition 1. *The $t_{IntroSC}$ transformer preserves extensions.*

Proof. Given GTSs $GTS_i = (TG_i, P_i, \pi_i)$, with $i = 0, 1$, and $\iota = (\iota_{TG}, \iota_P, \iota_r) : GTS_0 \hookrightarrow GTS_1$ an inclusion morphism, the first observation is that if the $t_{IntroSC}$ transformer is applicable to GTS_0, then it can also be applied to GTS_1 in exactly the same way. All classes in TG_0 are in TG_1, and specifically the class C to which the new subclass is added. Assume that the application of $t_{IntroSC}$ on GTS_0 results in a new GTS $GTS_0' = \left(TG_0', P_0, \pi_0\right)$, with rules as in GTS_0 and a type graph TG_0' as TG_0 but with a new class C' added, and declared subclass of C. Applying $t_{IntroSC}$ to GTS_1 results in the introduction of the new

class C'' as a subclass of $\iota_{TG}(C)$, with no attributes or links. This produces a new GTS $GTS'_1 = \left(TG'_1, P_1, \pi_1\right)$, with the same rules as in GTS_1. The inclusion morphism $\iota' = (\iota'_{TG}, \iota'_P, \iota'_r) : GTS'_0 \hookrightarrow GTS'_1$ can trivially be defined by defining ι'_{TG} by extending ι_{TG} for the new class and subclass relation, mapping C' to C'', with the same components for the rules. Notice that if ι_{TG} is an attribute type graph morphism, then ι'_{TG} is as well, since it preserves the inheritance relation, reflects subtyping, and preserves the abstraction definitions. Since rules are not changed, if ι is an extension, then ι' is also an extension.

Proposition 2. *The $t_{InhUnfld}$ transformer preserves extensions.*

Proof. Let GTSs GTS_0 and GTS_1 and inclusion GTS morphism $\iota : GTS_0 \hookrightarrow GTS_1$ as in Proposition 1, and let us assume it is an extension. Since the type graph is not changed, if the transformer is applicable on GTS_0, it is obviously applicable on GTS_1 for same class C, rule p and subclasses of C. We can define the inclusion morphism $\iota' = (\iota'_{TG}, \iota'_P, \iota'_r) : GTS'_0 \hookrightarrow GTS'_1$ by taking the same type graph morphism and morphisms for non-modified rules. Since the transformer replaces rules $\pi_0(p)$ and $\pi_1(p)$ with rules $\pi'_0(p)$ and $\pi'_1(p)$, respectively in GTS_0 and GTS_1, with these new rules generated in exactly the same way, given $\iota^p : \iota^{>}_{TG}(\pi_0(\iota_P(p))) \rightarrow \pi_1(p)$ we have $\iota'^p : \iota^{>}_{TG}(\pi'_0(\iota'_P(p))) \rightarrow \pi'_1(p)$. Since ι is an extension, to have that ι' is also an extension we just need to check that $\pi'_0(f'_P(p)) = f^{<}_{TG}(\pi'_1(p))$. But this is the case, since the transformer is exactly making the same changes.

Proposition 3. *The $t_{MvAssoc}$ transformer preserves extensions.*

Proof. Let GTSs GTS_0 and GTS_1 and inclusion GTS morphism $\iota : GTS_0 \hookrightarrow GTS_1$ as in Proposition 1. The transformer just moves an association *assoc* from class C to one or more of its subclasses, requiring that there is no rule in the source GTS using *assoc* on instances of C. Assuming ι is an extension, then, if $t_{MvAssoc}$ is applicable on GTS_0 for some C, some subclass(es) of C and rule p, there cannot be in GTS_1 a new rule, or an extension of a rule in GTS_0, with such a link (to C or any of its subclasses), and therefore the same application of the transformer is possible on GTS_1. Since the transformer is changing associations consistently in TG_0 and TG_1, the definition of ι'_{TG} follows quite closely that of ι_{TG}. Rules are left unchanged, so the definitions of ι'_P and ι'_r are as those in ι. Thus, we can conclude that ι' is also an extension.

Let us go back to our example in Sect. 2. First, notice that Tracker is an extension of the Server GTS. A new association is added to its type graph, and its rules are modified just by adding instances of the new elements. Then, to apply the composition scheme from [16], we need a morphism f from the parameter GTS Server to the PLS GTS. However, as we have seen in Sect. 2, the morphism f cannot be established. To support this composition scenario, we can express our server GTS as a GTS family and follow the scheme below:

$$\begin{array}{ccccc} [GTS_0]_T & \overset{}{\underset{T}{\rightsquigarrow}} & GTS'_0 & \xrightarrow{f} & GTS_2 \\ \downarrow g & & \downarrow g' & & \downarrow \widehat{g} \\ [GTS_1]_T & \underset{T}{\rightsquigarrow} & GTS'_1 & \dashrightarrow & \widehat{GTS} \end{array}$$

In other words, we explicitly encode the variability we find acceptable by extending the parameter GTS GTS_0 into a GTS family $[GTS_0]_T$, providing transformers $t_{IntroSC}$, $t_{InhUnfld}$, and $t_{MvAssoc}$ in T. By using these transformers, we can derive a GTS GTS'_0 (see Fig. 4) for which a GTS morphism f can be established to the PLS described in GTS_2. Because all the transformers in T are extension-preserving, we can derive a corresponding GTS'_1 and the extension $g' : GTS'_0 \hookrightarrow GTS'_1$. With these, we can finally apply the amalgamation scheme from [16] to produce the composed GTS $\widehat{GTS}$.

4.4 Finding GTS-Family Members

Finding the appropriate representative of a GTS family for a given composition problem is not trivial. Essentially, this requires searching through the space of GTSs spanned by the GTS family, looking for a GTS with the right structure, if any exists. Search-based problems have long been the subject of intense research interest [15]. More recently, there have been proposals for tools solving search problems in an MDE context [25–27]. Of particular interest in the context of GTS families is the work on MoMOT by Fleck *et al.* [27]. Here, new candidate solutions are found by applying a sequence of transformations to an initial model (*e.g.,* a GTS). The search is guided by appropriate fitness criteria (*e.g.,* the number of matching elements that could be used to construct a suitable morphism). Their approach keeps track of the transformation sequence at all time and thus guarantees that it will only find solutions for which there is a transformation sequence—a key criteria in finding representatives of GTS families.

Based on similar ideas, we have developed a basic automated search algorithm in Maude [28]. This prototype demonstrates that automated search of suitable GTSs in a GTS family is possible, but the prototype still suffers from inefficiencies. As part of our future work, we are exploring improving the implementation based on MoMOT or similar tools.

5 Conclusions

In this paper, we have presented GTS families as a mechanism for encoding controlled flexibility for morphisms between GTSs. This is achieved by extending a GTS GTS_0 to a set of GTSs that can be derived from GTS_0 given a set of GTS transformers T. This set is called the GTS T-family of GTS_0 and is taken to encode the full intent of what is expected to be preserved by any morphism from GTS_0. Then a direct morphism between GTS_0 and GTS_1 is replaced by selecting a suitable representative from the GTS family and defining the morphism from

that representative. Thus, instead of direct morphisms $GTS_0 \to GTS_1$ we will use the construction $[GTS_0]_T \rightsquigarrow GTS'_0 \to GTS_1$.

In addition to providing an explicit design mechanism for transformation developers, GTS families as a formal concept also open up a new research agenda: rather than relying on the pragmatic approaches taken to the definition of valid adaptations so far, we can now begin to study the fundamental properties of different types of GTS transformers, identifying different classes of GTS families that can be most appropriately used in different scenarios (*e.g.,* are there easily checked conditions that will guarantee extension preservation?). In this paper, we have shown how extension-preserving transformers can be used to construct GTS families that enable flexible GTS amalgamation. As part of our future work, we plan to study the properties required of GTS transformers to allow flexible reuse of transformations, extending the work by de Lara *et al.* to semantic transformation reuse.

Acknowledgements. This work has been partially supported by Spanish MINECO/FEDER project TIN2014-52034-R and Univ. Málaga, Campus de Excelencia Internacional Andalucía Tech.

References

1. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) 1st Graph Grammar Workshop, vol. 73, LNCS, pp. 1–69. Springer, Heidelberg (1979)
2. Engels, G., Heckel, R., Taentzer, G., Ehrig, H.: A combined reference model- and view-based approach to system specification. Int. J. Software Eng. Knowl. Eng. **7**(4), 457–477 (1997)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
4. Große-Rhode, M., Parisi-Presicce, F., Simeoni, M.: Spatial and temporal refinement of typed graph transformation systems. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 553–561. Springer, Heidelberg (1998). doi:10.1007/BFb0055805
5. Engels, G., Heckel, R., Cherchago, A.: Flexible interconnection of graph transformation modules. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 38–63. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31847-7_3
6. de Lara, J., Guerra, E.: From types to type requirements: Genericity for model-driven engineering. SoSyM **12**(3), 453–474 (2013)
7. Durán, F., Zschaler, S., Troya, J.: On the reusable specification of non-functional properties in DSLs. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 332–351. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36089-3_19
8. Durán, F., Moreno-Delgado, A., Orejas, F., Zschaler, S.: Amalgamation of domain specific languages with behaviour. J. Log. Algebraic Methods Program. (2015)
9. Baldan, P., Corradini, A., Dotti, F.L., Foss, L., Gadducci, F., Ribeiro, L.: Towards a notion of transaction in graph rewriting. Electr. Notes Theor. Comput. Sci. **211**, 39–50 (2008)

10. Taentzer, G.: A visual modeling framework for distributed object computing. In: Jacobs, B., Rensink, A. (eds.) FMOODS 2002. IFIP, vol. 81, pp. 263–278. Springer, Boston, MA (2002). doi:10.1007/978-0-387-35496-5_18
11. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theoret. Comput. Sci. **376**, 139–163 (2007)
12. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and kleisli categories. In: Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 163–177. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28872-2_12
13. de Lara, J., Guerra, E.: Towards the flexible reuse of model transformations: A formal approach based on graph transformation. J. Log. Algebraic Methods Program. **83**(5–6), 427–458 (2014)
14. Große-Rhode, M., Parisi Presicce, F., Simeoni, M.: Refinements of graph transformation systems via rule expressions. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 368–382. Springer, Heidelberg (2000). doi:10.1007/978-3-540-46464-8_26
15. Harman, M.: The current state and future of search based software engineering. In: Briand, L.C., Wolf, A.L. (eds.) International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, 23–25 May, Minneapolis, MN, USA, 342–357. IEEE Computer Society (2007)
16. Durán, F., Orejas, F., Zschaler, S.: Behaviour protection in modular rule-based system specifications. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 24–49. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37635-1_2
17. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1: Foundations, World Scientific (1997)
18. Parisi-Presicce, F.: Transformations of graph grammars. In: Cuny, J., Ehrig, H., Engels, G., Rozenberg, G. (eds.) Graph Grammars 1994. LNCS, vol. 1073, pp. 428–442. Springer, Heidelberg (1996). doi:10.1007/3-540-61228-9_103
19. Orejas, F., Lambers, L.: Symbolic attributed graphs for attributed graph transformation. ECEASST **30** (2010)
20. Orejas, F.: Symbolic graphs for attributed graph constraints. J. Symbolic Comput. **46**(3), 294–315 (2011)
21. Taentzer, G., Rensink, A.: Ensuring structural constraints in graph-based models with type inheritance. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 64–79. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31984-9_6
22. Cuadrado, J.S., Guerra, E., de Lara, J.: Flexible model-to-model transformation templates: an application to ATL. J. Object Technol. **11**(2), 4:1–4:28 (2012)
23. Guy, C., Combemale, B., Derrien, S., Steel, J.R.H., Jézéquel, J.-M.: On model subtyping. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 400–415. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31491-9_30
24. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02674-4_3
25. Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: A model-driven framework for guided design space exploration. In: Proceedings of the 26th IEEE/ACM International Conference Automated Software Engineering (ASE 2011), pp. 173–182, November 2011

26. Zschaler, S., Mandow, L.: Towards model-based optimisation: Using domain knowledge explicitly. In: Proceedings of Workshop on Model-Driven Engineering, Logic and Optimization (MELO 2016) (2016)
27. Fleck, M., Troya, J., Wimmer, M.: Marrying search-based optimization and model transformation technology. In: Proceedings of the 1st North American Search Based Software Engineering Symposium (NasBASE 2015) (2015) (Preprint). http://martin-fleck.github.io/momot/downloads/NasBASE_MOMoT.pdf
28. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, vol. 4350. LNCS. Springer, Heidelberg (2007)
29. Brim, L., Gruska, J., Zlatuška, J. (eds.): MFCS 1998. LNCS, vol. 1450. Springer, Heidelberg (1998)