Flexible Interconnection of Graph Transformation Modules

A Systematic Approach

Gregor Engels, Reiko Heckel, and Alexey Cherchago

University of Paderborn, Germany
{engels,reiko,cherchago}@upb.de

Abstract. Modularization is a well-known concept to structure software systems as well as their specifications. Modules are equipped with export and import interfaces and thus can be connected with other modules requesting or providing certain features.
In this paper, we study modules the interfaces of which consist of behavioral specifications given by typed graph transformation systems. We introduce a framework for classifying and systematically defining relations between typed graph transformation systems. The framework comprises a number of standard ingredients, like homomorphisms between type graphs and mappings between sets of graph transformation rules.
The framework is applied to develop a novel concept of substitution morphism by separating preconditions and effects in the specification of rules. This substitution morphism is suited to define the semantic relation between export and import interfaces of requesting and providing modules.

1 Introduction

One of the most successful principles of software engineering is *encapsulation*, i.e., the containment of implementations in classes, modules, or components accessible through well-defined interfaces only. This reduces possible dependencies of clients to those functions provided in the interface and allows to replace implementations without affecting the client.

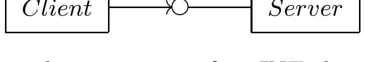

Fig. 1. Server component implementing interface INT that is used by Client component.

As it is obvious from Fig. 1, the developer of the Client component requires knowledge about the interface of the Server. That means, the development of the two components can not easily be decoupled and the architectural dependencies have to be known at design time.

In the service-oriented paradigm, but also in more advanced component models, this picture is extended by distinguishing between *provided* and *required* interfaces. While provided interfaces describe existing implementations, required interfaces are specifications of *virtual components* whose existence is assumed at design time to capture the context dependencies of the components under development.

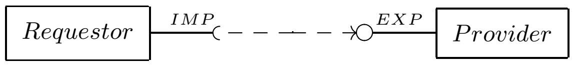

Fig. 2. Requestor and provider components.

As shown in Fig. 2, composing two components (or services) now means to connect their required and provided interfaces. This is possible if the operations asked for in the required interface are guaranteed by the provided interface. In programming languages like Java and component models like Corba such a relation between interfaces is verified by the compiler, matching the signatures (names and parameter types) of these operations.

However, in a truly open scenario, as it is typical for Web services or, more generally, service-oriented architectures, we cannot assume that, e.g., the name of an operation has any global meaning or that the types of the parameters convey enough information about the purpose and usage of the operation. In such case, it is inevitable that both required and provided interfaces contain *behavioral specifications* which are taken into account when interfaces are matched.

1.1 Module and Component Models with Behavioral Interfaces

The first steps in this direction have been made in the context of algebraic and logic specifications. An algebraic specification module MOD (see, e.g., [7]) consists of a body BOD providing the implementation and of interfaces IMP for import and EXP for export describing, respectively, required and provided functionality. (In addition, a parameter PAR is provided to allow for generic modules, but this feature will not be relevant for our purposes.) All specifications are connected through algebraic specification morphisms.

The composition of modules MOD and MOD' is based on morphisms, too, connecting the import (required) interface of IMP of MOD with the export (provided) interface of EXP' of MOD'. Hence, algebraic specification modules realize the idea illustrated in Fig. 2 that components are connected indirectly through the matching of required and provided interfaces.

In [7] the relation between IMP and EXP' is described by standard morphisms of algebraic specifications. That means, for example, that matching operations are required to have the same number of parameters of corresponding types. A more flexible approach to the connection of required and provided interfaces is presented by Zaremski and Wing in [25] and [26], who have developed sophisticated matching procedures at the level of both signatures and specifications.

In object-oriented programming, the extension of interfaces with behavioral information became known under the name of *Design by Contract* in the context

$$\begin{tikzcd}
 & & EXP \arrow[d, "exp"] \\
 & IMP \arrow[r, "imp"] \arrow[d, "subst"] & BOD \\
 & EXP' \arrow[d, "exp'"] & \\
IMP' \arrow[r, "imp'"] & BOD' &
\end{tikzcd}$$

Fig. 3. Conformation of the TGTS-modules.

of the Eiffel language [22]. Here, preconditions and effects of operations are specified by means of logic predicates, and object-oriented subtyping rules are extended to capture the compatibility of, for example, the contracts of a newly introduced subclass with that of a superclass.

1.2 Modules of Graph Transformation Systems

Since the mid nineties [5], there is an increasing interest in the transfer of modularity concepts from algebraic specifications and programming languages to graph transformation systems [21, 24, 10] (see also the survey [19]). Modules of typed graph transformation systems (TGTS modules) [10] follow the structure of algebraic specification modules, replacing the specifications BOD, IMP, and EXP by graph transformation systems related by different kinds of morphisms. In particular, IMP and BOD are related by a simple inclusion morphism ($IMP \subseteq BOD$), whereas EXP and BOD are connected by a refinement morphism, allowing a sequential or parallel decomposition of rules.

In [1, 15] it has been observed that graph transformation rules could provide a more abstract, visual representation of contracts, specifying preconditions and effects of operations. In order to check the desired behavioral compatibility of contracts between required and provided interfaces, a matching relation has been defined which can be syntactically verified. Roughly speaking, the semantic idea of this compatibility was the *substitution principle*, i.e., it should be safe to replace the required rule by a matching provided rule: The applicability of the first should imply the applicability of the latter, and the effect of applying the latter should satisfy the expectations of the first.

However, it has been assumed that the two rules to be compared are defined over the same types, based on the assumption that the matching is performed by a central discovery agency which represents both provided and required contracts over a common ontology. This assumption, which is satisfied in service-oriented architectures, is not in general true for modules or components.

It is the purpose of this paper to define a flexible matching relation enabling retyping as a morphism of typed graph transformation systems representing the required and provided (import and export) interfaces of modules.

1.3 Morphisms of Graph Transformation Systems

A survey of the literature reveals at least five fundamentally different proposals for morphisms of graph transformation system [2, 23, 13, 17, 20]. They represent different objectives, like inclusions, refinements, or views and enjoy different semantic properties.

So far there has been no general and systematic approach for comparing and relating different notions. Hence, before going on to extend the list by a new proposal, so-called substitution morphisms, we will survey possible definitions and provide a four-step recipe for deriving the appropriate definition from given semantic requirements.

We will apply this recipe to derive a notion of morphism between graph transformation systems with application conditions which, as it turns out, are essential for a flexible and yet semantically meaningful relation between required and provided interfaces.

1.4 Outline of the Paper

The rest of this paper is organized as follows. After recalling in the next section the basic concepts of the DPO and DPB approaches to graph transformation, in Section 3 we introduce a framework enabling to classify and systematically define morphisms of typed graph transformation systems. In particular, two examples of morphisms existing in the literature will be discussed informally in Section 3.1. After that, the constituents of the framework will be identified in Section 3.2, and aggregated in the definitions of the sample morphisms in Section 3.3. Then, a new concept of a substitution morphism playing a role of an inter-connector between two modules requiring and offering a specific service will be considered in Section 3.4.

The flexibility of the substitution morphism depends on the rule structure which is refined in Section 4 via separating preconditions and effects. Here, we will revise all the necessary definitions on the graph transformation rules with application conditions in Section 4.1, and formally define the substitution morphism in Section 4.2. In Section 5, the substitution morphism as well as the other sample morphisms are illustrated via their application as intra- and inter-connectors of the modules. We conclude with the summary of our work in Section 6.

2 Basic Definitions

In this section we review some of the basic notions of the *double-pushout* (DPO) [8] and *double-pullback* (DPB) [18] approaches to graph transformation. The DPB approach represents a loose version of the classical DPO, assuming that rules may be incomplete specifications of the transformations to be performed and thus allowing additional, unspecified effects. Both approaches are presented using typed graphs [3].

By *graphs* we mean directed unlabeled graphs $G = \langle G_V, G_E, src^G, tar^G \rangle$ with set of vertices G_V, set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating with each edge its source and target vertex. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ preserving source and target, that is, $src^H \circ f_E = f_V \circ src^G$ and $tar^H \circ f_E = f_V \circ tar^G$. With componentwise identities and composition this defines the category **Graph**.

Given a graph TG, called *type graph*, a *TG-typed (instance) graph* consists of a graph G together with a typing homomorphism $g : G \to TG$ (cf. Fig. 4 on the left) associating with each vertex and edge x of G its type $g(x) = t$ in TG. In this case, we also write $x : t \in G$. A TG-typed graph morphism between two TG-typed instance graphs $\langle G, g \rangle$ and $\langle H, h \rangle$ is a graph morphism $f : G \to H$ which preserves types, that is, $h \circ f = g$. With composition and identities this defines the category $\mathbf{Graph}_{TG}$, which is the comma category **Graph** over TG.

$$\begin{tikzcd}
G \arrow[d, "g"] \\
TG
\end{tikzcd}
\qquad
\begin{tikzcd}
G \arrow[rr, bend left, "f"] \arrow[dr, "g"'] & & H \arrow[dl, "h"] \\
& TG &
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d, "d_L"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d_K"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "d_R"] \\
G & D \arrow[l, "g"] \arrow[r, "h"'] & H
\end{tikzcd}$$

Fig. 4. Typed graph and graph morphism (left) and double-pushout (or -pullback) diagram (right).

Definition 1 (typed graph transformation system). *A TG-typed graph* transformation rule *is specified by a span $(L \xleftarrow{l} K \xrightarrow{r} R)$ of injective TG-typed graph morphisms (cf. Fig. 4 on the right).*

Given TG-typed graph transformation rules $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and $q = (L' \xleftarrow{l'} K' \xrightarrow{r'} R')$, a typed rule morphism *$f : p \to q$ is a tuple (f_L, f_K, f_R) of TG-typed graph morphisms commuting with the span morphisms l, l', r and r' (cf. Fig. 5). With componentwise identities and composition this defines the category $\mathbf{Rule}_{TG}$, which is the comma category* **Rule** *over TG.*

$$\begin{tikzcd}
L \arrow[d, "f_L"'] \arrow[dr, phantom, "(=)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "f_K"] \arrow[dr, phantom, "(=)"] & R \arrow[d, "f_R"] \\
L' & K' \arrow[l, "l'"] \arrow[r, "r'"'] & R'
\end{tikzcd}$$

Fig. 5. Typed rule morphism.

A typed graph transformation system $GTS = (TG, P, \pi)$ *consists of a* type graph *TG, a set of rule names P, and a mapping $\pi : P \to |\mathbf{Rule}_{TG}|$ associating with each rule name p a TG-typed rule $\pi(p)$.*

The *left-hand side* L of a rule contains the items that must be present for an application of the rule, the *right-hand side* R those that are present afterwards, and the *interface graph* K specifies the "gluing items", i.e., the objects which are read during application, but are not consumed.

As running example, a specification of a mutual exclusion algorithm with deadlock detection [16] is developed throughout the paper.

Example 1 (MUTEX). The typed graph transformation system in Fig. 6 models a distributed algorithm for mutual exclusion (MUTEX). This example is derived from a small case study [16] and tailored for our presentation. Two basic types, processes P (drawn as black nodes) and resources R (drawn as light boxes), constitute the type graph shown in the upper-left corner. A *request* is modeled by an edge going from a process to a resource. The fact that the resource is currently *held_by* the process is shown by an edge in the opposite direction. A token ring algorithm implements the mutual exclusion. The processes in the token ring are arranged in a cycle. Two neighbor processes are connected by an edge running from the antecedent to the *next* process. This edge is given by a loop in the type graph. A default position for introducing new processes and resources is marked by a pointer *head.*

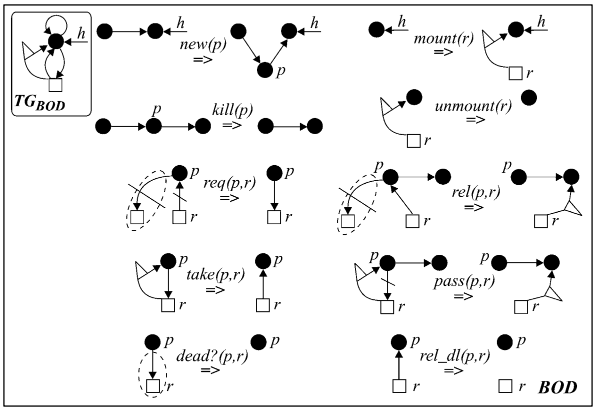

Fig. 6. Graph transformation system modeling MUTEX algorithm.

An edge with a white flag denotes a token which is passed from process to process along the ring. In order to get an access to a resource a process waits for the corresponding token. Mutual exclusion is achieved by uniqueness of the token for each resource in the system.

Now we discuss the rules of the graph transformation system. The first four rules are used for creating and killing processes (*new* and *kill*), and for mounting and unmounting resources (*mount* and *unmount*). The rules *req*, *take*, and *rel* allow processes to issue requests, take resources, and release them upon *regular* completion of their task. The negative application conditions [14] for *req* ensure that a process can not issue more then one request at a time. The negative application condition for *rel* prevents the release of a resource r while the process requests another resource, since r may still be required to complete the given task.

The last two rules are intended for application in possibly deadlock situations resulting from competition of processes for non-sharable resources. The MUTEX algorithm does not know how to detect deadlocks, therefore the rule *dead*? represents external features, to be imported from another module. The dotted part of this rule is a positive application condition (cf. Section 4.1) representing items that must be present for the rule application, but are not consumed. This condition restricts the applicability of rule *dead*? to situations where the process has a pending request for a resource.

In general, positive application conditions can be encoded by extending both the left- and the right-hand side of a rule by the required elements: they become part of the context. That means rules with positive application conditions can easily be transformed into ordinary rules. The use of positive application conditions makes a difference, however, when we consider relations between different systems, as shall be demonstrated in Section 4.2.

Rule *rel_dl* finally implements the resolution of detected deadlocks by forcing the release of the resource held by the involved process.

In the DPO approach, transformation of graphs is defined by a pair of pushout diagrams, a so-called double-pushout construction. Operationally speaking that means: the elements of G matched by $L \setminus l(K)$ are removed, and a copy of $R \setminus r(K)$ is added to D.

A *double-pushout (DPO) diagram d* is a diagram as in Fig. 4 on the right, where (1) and (2) are pushouts. Gluing the graphs L and D over their common part K yields again the given graph G, i.e., D is a so-called *pushout complement* and the left-hand square (1) is a pushout square. Only in this case the application is permitted. Similarly, the derived graph H is the gluing of D and R over K, which forms the right-hand side pushout square (2).

This formalization implies that only vertices that are preserved can be merged or connected to edges in the context. It is reflected in the *identification* and the *dangling conditions* of the DPO approach. The *identification condition* states that objects from the left-hand side may only be identified by the match if they also belong to the interface (and are thus preserved). The *dangling condition* ensures that the structure D obtained by removing from G all objects that are to be deleted is indeed a graph, that is, no edges are left "dangling" without source or target node.

Definition 2 (DPO graph transformation). *Given a typed graph transformation system* $GTS = (TG, P, \pi)$*, a* (DPO) transformation step *in* GTS *from* G *to* H *via* p *is denoted by* $G \overset{p/d}{\Longrightarrow} H$*, or simply by* $G \overset{p}{\Longrightarrow} H$ *if the DPO diagram* d *is understood.*

A transformation sequence $\rho = \rho_1 \ldots \rho_n : G \Rightarrow^* H$ *in* GTS *via* $p_1, \ldots, p_n$ *is a sequence of transformation steps* $\rho_i = (G_i \overset{p_i/d_i}{\Longrightarrow} H_i)$ *such that* $G_1 = G, H_n = H$ *and consecutive steps are composable, that is,* $G_{i+1} = H_i$ *for all* $1 \leq i < n$*.*

The category of transformation sequences over *GTS denoted by* $\mathbf{Trf}(GTS)$ *has all graphs* $G \in \mathbf{Graph}_{TG}$ *as objects and all transformation sequences in GTS as arrows.*

A sample transformation step is shown in Fig. 7. It applies the rule *new* inserting a new process in the token ring.

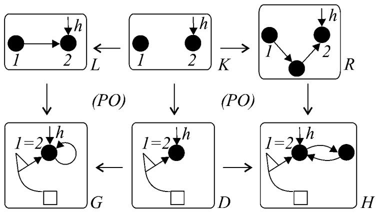

Fig. 7. A sample DPO transformation step.

The DPO approach ensures that the changes to the given graph H are exactly those specified by the rule. A more liberal notion of rule application is provided by the double-pullback (DPB) approach to graph transformation [18], where *at least* the elements of G matched by $L \setminus l(K)$ are removed, and *at least* the elements matched by $R \setminus r(K)$ are added. The DPB approach introduces *graph transitions* and generalizes DPO by allowing additional, unspecified changes. Formally, graph transitions are defined by replacing the double-pushout diagram of a transformation step with a double-*pullback*.

Definition 3 (DPB graph transitions). *Given a typed graph transformation system $GTS = (TG, P, \pi)$, a* transition *in GTS from G to H via p, denoted by $G \overset{p/d}{\rightsquigarrow} H$, is a diagram like in the right of Fig. 4, where both (1) and (2) are pullback squares. A transition is called* injective *if both g and h are injective graph morphisms. It is called* faithful *if it is injective, and the morphisms d_L and d_R satisfy the following* identification condition [4] *with respect to l and r: for all $x, y \in L$, $y \notin l(K)$ implies $d_L(x) \neq d_L(y)$, and analogously for d_R.*

A transition sequence *$\rho = \rho_1 \dots \rho_n : G \rightsquigarrow^* H$ in GTS via $p_1, \dots, p_n$ is a sequence of faithful transitions $\rho_i = G_i \overset{p_i/d_i}{\rightsquigarrow} H_i$ such that $G_1 = G, H_n = H$ and consecutive steps are composable, that is, $G_{i+1} = H_i$ for all $1 \leq i < n$.*

The category of transitions over *GTS, denoted by* $\mathbf{Trs}(GTS)$*, has all graphs* $G \in \mathbf{Graph}_{TG}$ *as objects and all transition sequences in GTS as arrows.*

A sample transition is shown in Fig. 8. It also demonstrates an application of the rule *new*. Note that during application of the rule a token is deleted that is unspecified by *new*. Here the left-hand square is not a pushout: the graph G obtaining by the gluing of L and D additionally contains the *token* which is "spontaneously deleted".

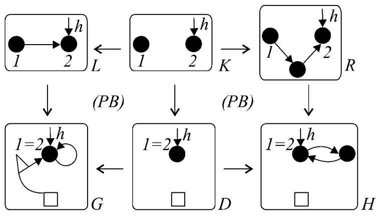

Fig. 8. A sample DPB graph transition.

3 TGTS Morphisms, Systematically

In this section, we provide a framework for classifying and systematically defining morphisms of typed graph transformation systems based on a number of standard “ingredients”, like homomorphisms between type graphs and mappings between sets of rules. First, two examples of morphisms will be discussed informally in the context of TGTS modules. Then, in Section 3.2, the constituents of the framework are presented and combined, yielding definitions of the sample morphisms in Section 3.3. In Section 3.4, a novel concept of *substitution morphism* is considered.

3.1 TGTS Morphisms as Intra-connectors of Modules

Each TGTS describes a specific behavior in terms of the transformation or transition sequences obtained via application of its rules. A TGTS morphism $f : GTS \to GTS'$ defines a relation between the behaviors of GTS and GTS' through an association of their type graphs and rules. Thus, a systematic approach should always start by identifying the kind of semantic relation that shall be expressed.

First, we consider an example of *behavior-preserving* morphisms providing a first attempt at describing the relation between the export interface EXP of a module MOD with its body BOD. The export interface EXP specifies the features offered for import by other modules. The specification of these features should be consistent with their implementation in the body. That means, applicability of EXP rules should imply applicability of the corresponding BOD rules. Behavior-preserving morphisms shall ensure this property.

Example 2 (behavior-preserving morphism). The body of the module MUTEX is given in Fig. 6. One service provided by this module is deadlock resolution described by the rule *rel_dl* in the export interface EXP (cf. Fig. 9 on the right). (It shall be imported by an external deadlock detection module to break up detected deadlocks.) The embedding of EXP into BOD preserves behavior: Each transformation sequence in EXP implies a corresponding sequence in BOD.

The type graph TG_{EXP} of the export is a subgraph of TG_{BOD} of the body containing all the types relevant for deadlock resolution. More generally, a homomorphism between type graphs ensures that all types of the source (TG_{EXP} in

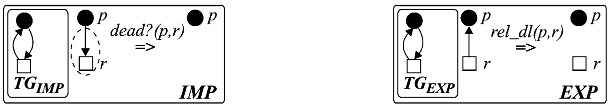

Fig. 9. TGTSs IMP (left) and EXP (right) of the module MOD modeling the MUTEX algorithm.

this case) have a correspondence in the target (TG_{BOD}). If the homomorphism is not an inclusion, as in our example, a type in the target may have a different name than its source or two different types in the source may be mapped to the same target type.

Based on the homomorphism, graphs, rules, and also transformations typed over the source can be converted into such typed over the target by a simple renaming of their types. This gives us the opportunity to compare two systems by translating the rules of the source system into ones typed over the target.

Due to the subgraph relation between TG_{EXP} and TG_{BOD} the translation of the EXP rule to the BOD type graph does not change anything in this rule. The comparison reveals that the rule identical to *rel_dl* is already present in TG_{BOD}, even with the same name. In the general case, we might consider a mapping of rule names as well to use different names for corresponding rules in the two systems.

The behavior-preserving morphisms as discussed above are originally introduced in [9, 11]. In our example, the export interface EXP is just a subsystem of BOD. More general situations are considered in [11] where the relation between export interface and body may be, e.g., *spatial* or *temporal* refinements. In spatial refinements, a rule of the source system may be associated with an amalgamation of rules of the target system, in temporal refinements with a sequential composition.

The requirements and definitions about behavior-preserving morphisms are presented in Section 3.3. Here, we proceed with an example of *behavior-reflecting* morphisms, determining the relation between the import interface IMP and the body BOD of a module MOD. The idea is that the rules required at IMP have at least the effect of the rules specified at BOD. Otherwise, the body could not use the imported rules for the internal implementations. This can be expressed as a reflection of the BOD transformations by IMP transitions.

Example 3 (behavior-reflecting morphism). As mentioned already, deadlock detection represents an external feature abstractly represented in the MUTEX module by the rule *dead?* in the import interface IMP (cf. Fig. 9 on the left).

Reflection of BOD behavior by IMP means that for each transformation in BOD we require a corresponding transition in IMP. As with behavior-preserving morphisms, we have to specify the relation between the type graphs and rules of the two systems.

For type graphs, a homomorphism from TG_{IMP} to TG_{BOD} ensures that BOD has at least the same types as IMP. In order to check that transformations in BOD are reflected by transitions in IMP, we have to compare the rules of the two systems. In this case, since we are interested in reflection rather than preservation of steps, we translate the rules of BOD to IMP *against* the direction of the type graph morphism. That means, beside the renaming of types, elements of the rules are removed if their type in BOD does not have a pre-image in IMP under the type graph homomorphism.

Then, the BOD behavior is reflected by IMP if each rule, after the translation, turns out to be a super-rule of the corresponding rule in IMP. In our case, the rule *dead*? of BOD coincides with the one in IMP after the translation.

Morphisms that reflect transformation in the target by transitions in the source have been introduced in [16] to specify the relation between different views of a system model.

Below, we formally introduce the different components of TGTS morphisms.

3.2 Definitions of Ingredients

In this section we define the two main ingredients of TGTS morphisms, i.e., translations between type graphs and subrule relations. We start with forward and backward retyping using the notation of [12].

Definition 4 (retyping). *A graph morphism $f_{TG} : TG \to TG'$ induces a* forward retyping functor $f^{>}_{TG}$: $\mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$*, $f^{>}(g) = f \circ g$ and $f^{>}(k : g \to h) = k$ by composition as shown in the diagram below,*

$$\begin{tikzcd}
 & H \arrow[dd, "h"] & \\
G \arrow[ur, "k"] \arrow[dr, "g"'] & & \\
 & TG \arrow[r, "f"'] & TG'
\end{tikzcd}$$

as well as a backward retyping functor $f^{<}_{TG}$: $\mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$*, $f^{<}(g') = g^*$ and $f^{<}(k' : g' \to h') = k^* : g^* \to h^*$ by pullbacks and mediating morphisms as shown in the diagram below.*

$$\begin{tikzcd}
 & H^* \arrow[rr] \arrow[dd, "h^*"] & & H' \arrow[dd, "h'"] \\
G^* \arrow[ur, "k^*"] \arrow[rr] \arrow[dr, "g^*"'] & & G' \arrow[ur, "k'"] \arrow[dr, "g'"'] & \\
 & TG \arrow[rr, "f"'] & & TG'
\end{tikzcd}$$

We proceed by listing a number of relations between rules typed over the same type graph.

Definition 5 (subrule relations). *Given TG-typed graph transformation rules $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, $q : (L' \xleftarrow{l'} K' \xrightarrow{r'} R')$, and a* typed rule morphism *$f : p \to q$ (cf. Fig. 5), we say that*

- *p is* identical *to q, in symbols $p = q$, if f is an identity in $\mathbf{Rule}_{TG}$,*
- *p is a* DPO-subrule *of q, in symbols $p \sqsubseteq_{DPO} q$, if the diagrams (1) and (2) in Fig. 5 are pushouts in $\mathbf{Graph}_{TG}$,*
- *p is a* DPB-subrule *of q, in symbols $p \sqsubseteq_{DPB} q$, if the diagrams (1) and (2) in Fig. 5 are pullbacks in $\mathbf{Graph}_{TG}$ and construct a faithful transition.*

This list could be further extended by relations between a single rule and a collection of rules such as the *spatial* and *temporal* refinements, but this is beyond the scope of this paper.

Having defined retyping and rule relations we are now in a position to combine these ingredients into definitions of TGTS morphisms.

3.3 Recipes for TGTS Morphism

We have already discussed by means of the two examples in Section 3.1 how semantic requirements determine the definition of morphisms of graph transformation systems. In this section, we are going to make this explicit in terms of a four-step recipe. In each step we introduce a number of options and motivate possible choices based on the semantic requirements. First of all we formulate an initial assumption to simplify the presentation.

Assumption: Without loss of generality we assume that the TGTS morphism $f : GTS \to GTS'$ and the type graph morphism $f_{TG} : TG \to TG'$ have the same direction. That means, the target system has at least the types like the ones of the source system, but possibly more.

Step 1. The first variation point is the *relation between the sets of rule names* of GTS and GTS'. Here it is most convenient to use total functions, rather than general relations. For example, a mapping from P to P' designates for each $p \in P$ one corresponding $p' \in P'$: the relation is *left total* and *right unique*. This option should be used for behavior-preserving morphisms, where each transformation of the source system has to be associated with a transformation of the target system. Dually, a mapping in the opposite direction provides for each $p' \in P'$ one $p \in P$: a *left unique* and *right total* relation which is suitable for the behavior-reflecting morphisms.

Step 2. The next alternative is introduced by the *context of comparison*, i.e., where the corresponding rules of the two systems are compared. This can be done either in the context of GTS' using the forward retyping $f^{>}_{TG} : \mathbf{Graph_{TG}} \to \mathbf{Graph_{TG'}}$ of the rules in GTS, or in the context of GTS using the backward retyping $f^{<}_{TG} : \mathbf{Graph_{TG'}} \to \mathbf{Graph_{TG}}$ of the rules in GTS'. The forward retyping is appropriate for behavior-preserving morphism, the objective in this case being the construction of transformations in the target from existing ones in the source system. By analogy backward is used for behavior-reflecting morphisms.

We continue with the specification of the subrule relation required between the rules of the two systems. For pairs of corresponding rules as defined in *Step 1* and modulo the retyping functor selected in *Step 2* this means to decide for the direction of the relation in *Step 3* and its kind in *Step 4*.

Step 3. The *direction of the subrule relation*, i.e., if p is required to be a subrule of p', or vice versa, depends on the desired relation between the sets of transformations or transitions of the two systems. If p is a subrule of p' then each transformation step via p' implies a transition or transformation step via p. Thus, behavior-preserving morphisms generally require that the GTS' rules are subrules of the GTS rule, while behavior-reflecting morphisms specify the dual requirement.

Remark 1. Note that it may be the case that a subrule relation between p and p' holds when considered over the larger type graph GTS' using forward retyping, but not if compared via backward retyping (projection) over the smaller GTS type graph. The converse is also true, i.e. a subrule relation may hold over GTS, but not over GTS'.

This motivates why the comparison of rules is always done over the system where the existence of transformations or transitions should be ensured, i.e., the target system if behavior shall be preserved and the source system if behavior shall be reflected.

Step 4. Finally, we have to select the *kind of subrule relation* that the comparison shall be based upon. The identity of p and p' ensures that all transformations via p are also transformations via p'. If p is a DPO or DPB subrule of p', respectively, then each transformation step via p' implies a transformation ($\sqsubseteq_{DPO}$) or transition ($\sqsubseteq_{DPB}$) via p. (The dual holds if we replace p and p'.)

The relation between the different choices and the implied semantic properties is summarized in Table 1. Combinatorially, we obtain eight different notions. Numbers 4 and 5 represent, respectively, the behavior-reflecting and preserving morphisms discussed above.

Next we introduce formally the semantic requirements of behavior-preservation and -reflection and, subsequently, the actual definitions of the morphisms.

Table 1. Ingredients of TGTS-morphisms.

	forward retyping $f^{>}_{TG}$		backward retyping $f^{<}_{TG}$	
	left-total right-unique relation	left-unique right-total relation	left-total right-unique relation	left-unique right-total relation
$\sqsubseteq$	– 1	– 2	DPO/DPB_{ϵ} 3	DPO,DPB,[=] 4
$\sqsupseteq$	=,[DPO/DPB] 5	– 6	– 7	– 8

Definition 6 (preservation of behavior). *Given typed graph transformation systems $GTS = (TG, P, \pi)$ called the source system and $GTS' = (TG', P', \pi')$ called the target system. We say that the target system preserves the behavior of the source system if there exists a functor $F : \mathbf{Trf}(GTS) \to \mathbf{Trf}(GTS')$.*

The existence of a functor between two categories of sequences requires that each individual step in GTS is mapped to a sequence in GTS'. By induction, this mapping extends to sequences in GTS. However, we will deal with the simpler case where a step in GTS is actually mapped to a single step in GTS'.

As discussed above, this requires that each rule $p \in P$ has a corresponding rule $p' \in P'$. Hence, a mapping $f : P \to P'$ is chosen in *Step 1*. To ensure the preservation of sequences in GTS', the comparison of rules is done in the context of GTS' and, therefore, forward retyping is applied at *Step 2*.

The mapping in *Step 1* must guarantee the desired relation between the transformations in the two systems. This is achieved if in *Step 3* the rules in GTS' are subrules of those in GTS. The choices in *Step 4* ensuring behavior preservation range from identity to DPB relations. The identity is the most common one because it results in an embedding of GTS into GTS', while any true subrule relations would mean that the rules of GTS are reduced in GTS'.

The behavior-preserving morphism is specified in cell 5 of Table 1 and formally defined below.

Definition 7 (behavior-preserving morphism). *Given typed graph transformation systems $GTS = (TG, P, \pi)$ and $GTS' = (TG', P', \pi')$, a* behavior-preserving TGTS morphism *$f^{pres} = (f_{TG}, f_P)$ is given by a type graph morphism $f_{TG} : TG \to TG'$ and a mapping $f_P : P \to P'$ between the sets of rule names such that for each $p \in P$, $f^{>}_{TG}(\pi(p)) = \pi'((f_P(p)))$.*

The justifications for the following claim can be found in [9, 11].

Fact 1 *Behavior-preserving morphisms $f^{pres} : GTS \to GTS'$ satisfy the requirements of Def. 6.*

Just to consider another example, the candidate in cell 6 differs from the one above in the direction of the mapping between rule names. That means, to each $p' \in P'$ a $p \in P$ is associated. If we require the existence of the subrule relation for all pairs of rules thus associated, this guarantees a partial preservation of behavior only, i.e., for those transformations in GTS via rules with corresponding rules in GTS'.

To continue on the right-hand side of the table, the semantic requirements for behavior-reflecting morphisms are given below.

Definition 8 (reflection of behavior). *Given typed graph transformation systems $GTS = (TG, P, \pi)$ called the source system and $GTS' = (TG', P', \pi')$ called the target system, we say that the first reflects the behavior of the second if there exists a functor $F : \mathbf{Trf}(GTS') \to \mathbf{Trs}(GTS)$.*

That means, each transformation step in GTS' implies a *transition* in GTS, a liberal requirement compared to reflecting transformations in *transformations*.

By the same arguments as above, in *Step 1* we assume a mapping of rule names from P' to P. The context of comparison is the source system, leading to the use of backward retyping is selected at *Step 2*. To fulfill the semantic requirement, rules in P' are subrules of corresponding rules in P in *Step 3*. Both DPO or DPB subrule relations are reasonable at *Step 4*. The first would, in fact, guarantee the stronger reflection property based on transformations only.

This morphism specified in cell 4 of Table 1 is formally defined below.

Definition 9 (behavior-reflecting morphism). *Given typed graph transformation systems* $GTS = (TG, P, \pi)$ *and* $GTS' = (TG', P', \pi')$, *a* behavior-reflecting morphism $f^{refl} = (f_{TG}, f_P)$ *is given by a type graph morphism* $f_{TG} : TG \to TG'$ *and a mapping* $f_P : P' \to P$ *between rule names such that for each* $p' \in P'$, $\pi(f_P(p')) \sqsubseteq_{DPO/DPB} f_{TG}^{<}(\pi'(p'))$.

The proof of following is obvious.

Fact 2 *Behavior-reflecting morphisms* $f^{refl} : GTS \to GTS'$ *satisfy the requirements of Def. 8.*

In [16] a variant of the above has been used represented by cell 3. The difference from 4 is the direction of the mapping of rule names from P to P', i.e., in the same direction like the mapping of types. Using DPB subrules and assuming in each GTS an empty ϵ-rule, each step in GTS' using a rule without a corresponding rule in GTS is associated with an ϵ-transition. In this way, the behavior is indeed reflected by GTS.

If we consider, instead, DPO subrules we obtain a partial reflection of the target transformations by the source ones.

It turns out that none of the other alternatives in Table 1 preserve or reflect behavior. Variants 1 and 2 are not behavior-preserving, because the subrule relation allows rules in the target system to be larger than in the source. Hence, additional preconditions may be introduced which make rules in GTS' applicable in less situations.

Similarly, variants 7 and 8 are inadequate for the behavior reflection since, due to Remark 1, subrule rule relations are not in general preserved by the retyping.

The preservation properties for subrule relations between the rules of the two systems are detailed in Fig. 10.

In the next section we discuss another kind of semantic relation, called substitutability: Abstract operation specifications in the source system (e.g., the import interface) shall by substituted by their implementations in the target system (e.g., the body of another module).

3.4 Towards Substitution Morphisms

Let us come back to the discussion of the connector between the import interface IMP of the requestor module MOD and the export interface EXP' of the

$$\begin{tikzcd}
f^{>}_{TG}(\pi(p)) \sqsubseteq \pi'(p') \arrow[r, Rightarrow, shift left, "/" marking] \arrow[d, Rightarrow, shift left, "/" marking] & \pi(p) \sqsubseteq f^{<}_{TG}(\pi'(p')) \arrow[l, Rightarrow, shift left] \arrow[d, Rightarrow, shift left, "/" marking] \\
f^{>}_{TG}(\pi(p)) \sqsupseteq \pi'(p') \arrow[r, Rightarrow, shift left] \arrow[u, Rightarrow, shift left, "/" marking] & \pi(p) \sqsupseteq f^{<}_{TG}(\pi'(p') \arrow[l, Rightarrow, shift left, "/" marking] \arrow[u, Rightarrow, shift left, "/" marking]
\end{tikzcd}$$

Fig. 10. Relation between the different alternatives for the TGTS-morphisms.

provider module MOD'. Since IMP and EXP' are TGTS, the desired relation between them should be described via a TGTS morphism. To construct an appropriate recipe for the morphism between IMP and EXP', it is necessary to understand what are the semantic requirements behind this a relation.

IMP contains the abstract specifications of the required features, so it is natural to interpret its rules as incomplete, with DPB semantics. The provider offers the concrete implementations of the operations, therefore the corresponding rules should be complete, having DPO interpretation.

The concrete rules can be safely substituted for the abstract rules if the following two conditions are true: First, the effect of applying a rule of EXP' should satisfy the expectations described in the rule of IMP for which it was substituted. This is the case if IMP reflects the EXP' behavior. Second, applicability of the IMP rule should imply applicability of the corresponding EXP' rule, i.e., applicability must be preserved from IMP to EXP'. These two requirements are formally given in the following definition.

Definition 10 (substitutability). *Given typed graph transformation systems $GTS = (TG, P, \pi)$, the source system, and $GTS' = (TG', P', \pi')$, the target system, the second is substitutable for the source if there exists a functor F : $\mathbf{Trf}(GTS') \to \mathbf{Trs}(GTS)$ such that for all graphs $G' \in |\mathbf{Trf}(GTS')|$ and for all transition sequences $\rho : F(G') \to \_ \in \mathbf{Trs}(GTS)$ there exists a transformation sequence $\rho' : G' \to \_ \in \mathbf{Trf}(GTS')$ with $F(\rho') = \rho$.*

Let us give an operational interpretation of what happens when the abstract rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ are substituted for the concrete rules $p'_i = (L'_i \xleftarrow{l'_i} K'_i \xrightarrow{r'_i} R'_i)$. This assumes that requestor and provider are actual components which communicate at runtime.

$$\begin{tikzcd}
G_0 \arrow[r, rightsquigarrow, "p_1/d_1"] & H_0 = G_1 \arrow[r, rightsquigarrow, "p_2/d_2"] & H_1 = G_2 & \cdots \\
G'_0 \arrow[u, mapsto, "f^{<}_{TG}"] \arrow[r, Rightarrow, "p'_1/d'_1"] & H'_0 = G'_1 \arrow[u, mapsto, "f^{<}_{TG}"] \arrow[r, Rightarrow, "p'_2/d'_2"] & H'_1 = G'_2 \arrow[u, mapsto, "f^{<}_{TG}"] & \cdots
\end{tikzcd}$$

Fig. 11. Substitution in detail.

The starting point is a graph $G'_0 \in \mathbf{Graph_{TG'}}$, representing the state of the provider component (cf. Fig. 11).

The substitution consists of the following steps:

- G'_0 is projected to $G_0 \in \mathbf{Graph_{TG}}$ via backward retyping, modeling the requestors incomplete knowledge about the provider.
- If a rule p_1 is applicable to G_0 on the requestor side, the same holds for the corresponding provider rule $p'_1 = f_P(p_1)$.
- A transformation step $G' \overset{p'_1/d'_1}{\Longrightarrow} H'$ is performed by the provider which projects to a transition $G_0 \overset{p_1/d_1}{\rightsquigarrow} H$ via the corresponding rule in the requestor view.

Thus, the requestor receives an update to its local view of the state of the provider, and the cycle can start anew.

After this operational motivation, let us understand the consequences of the semantic requirements of Def. 10, i.e., reflection of behavior and preservation of rule applicability. The first requires that rules are compared over the GTS (backward retyping for *Step 2*) with $\pi(p) \sqsubseteq_{DPB} f^{<}_{TG}(\pi'(p'))$ (DPB subrule from p to p' for *Steps 3 and 4*). The second is guaranteed if the left-hand sides of the rules p' is contained in that of p (*Steps 3 and 4*), compared over GTS' (forward retyping for *Step 2*). Thus, modulo retyping, p is contained in p', but L' is contained in L, i.e., the rules must be essentially identical.

It is clear that this is not a satisfactory result because it means that, again, requestor and provider components have to be developed in close coordination. We will see in the next section that the solution consists in a separation of the *preconditions* for the application of the rule from the description of the *effects* of the transformation. Indeed, the problem occurs because the left-hand side of a rule mixes up items restricting the applicability with items needed to specify the actual transformations.

4 Separating Preconditions and Effects

As discussed in the previous section, the separate specifications of application conditions and transformations allows for a more flexible notion of substitution morphisms. The desired separation is achieved by extending rules with positive and negative application conditions as introduced below. In Section 4.2, substitution morphisms will be introduced formally. It will be illustrated by an example in Section 5, as well as the other morphisms discussed so far.

4.1 Application Conditions

Negative conditions are well-known to increase the expressive power of rules [14]. This is not the case for positive application conditions which are easily encoded in the left-hand side of a rule (more precisely: in both the left- and the right-hand side if the elements are to be preserved).

However, this encoding, while leading to an identical operational behavior, is not compatible with the semantic requirements for substitution morphisms.

For example, by strengthening the precondition of an operation in the import we should preserve legal substitution relations because the overall requirements towards existing implementations are weakened. Yet, due to the encoding we are enlarging the rule itself, which reduces the collection of legal substitution morphisms outgoing from the import interface.

Therefore, we consider in the following definition negative as well as positive application conditions.

Definition 11 (rules with application conditions). *An* application condition $A(p) = (AP(p), AN(p))$ *for a graph transformation rule* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ *consists of two sets of typed graph morphisms* $AP(p), AN(p)$ *outgoing from* L *which contain positive and negative constraints, respectively.* $A(p)$ *is called positive (negative) if* $AN(p)$ $(AP(p))$ *is empty.*

Let $L \xrightarrow{\hat{l}} \hat{L}$ *be a positive or negative constraint and* $L \xrightarrow{d_L} G$ *be a typed graph morphism (cf. Fig. 12). Then* d_L P-satisfies $\hat{l}$, *if there exists a typed graph morphism* $\hat{L} \xrightarrow{d_{\hat{L}}} G$ *such that* $d_{\hat{L}} \circ \hat{l} = d_L$. d_L N-satisfies $\hat{l}$, *if it does not* P-satisfy $\hat{l}$.

$$\begin{tikzcd}
\hat{L} \arrow[dr, "d_{\hat{L}}"'] & L \arrow[l, "\hat{l}"'] \arrow[d, "d_L"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d_K"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "d_R"] \\
 & G & D \arrow[l, "g"] \arrow[r, "h"'] & H
\end{tikzcd}$$

Fig. 12. DPB graph transition and rule with application condition.

Let $A(p) = (AP(p), AN(p))$ *be an application condition and* $L \xrightarrow{d_L} G$ *be a typed graph morphism. Then* d_L satisfies $A(p)$, *if it* P–satisfies *at least one positive constraint and* N-satisfies *all negative constraints from* $A(p)$.

A graph transformation rule with application condition *is a pair* $\hat{p} = (p, A(p))$ *consisting of a graph transformation rule* $p : s = (L \xleftarrow{l} K \xrightarrow{r} R)$ *and an application condition* $A(p)$ *for* p. *It is* applicable *to a graph* G *via* $L \xrightarrow{d_L} G$ *if* d_L *satisfies* $A(p)$.

Let $\hat{p} = (p : (L \xleftarrow{l} K \xrightarrow{r} R), A(p))$ *be a graph transformation rule with application condition. A* graph transition *from* G *to* H *via the rule* $\hat{p}$, *denoted by* $G \overset{\hat{p}/d}{\rightsquigarrow} H$, *is a graph transition via a rule* p, *such that* $d_L \in d$ *satisfies the application condition of* $\hat{p}$.

Note that positive application conditions consist of a disjunction of positive constraints, in contrast with the conjunction in [14]. That means, $L \xrightarrow{d_L} G$ satisfies $AP(p)$ if it satisfies *at least one* positive constraints. So, positive and negative conditions are, in fact, dual to each other.

As an example of a rule with positive and negative constraints let us consider the rule *req* in Fig. 6. Constraints are represented in the left-hand side of the rule where they are distinguished by dotted borders. If a positive constraint coincides with L, we omit this border. All nodes and edges outside these borders form the left-hand side L while $\hat{L}$ is given by the left-hand side plus one of the bordered parts and $\hat{l}$ or $\hat{k}$ by the corresponding embedding. Two negative constraints and one positive, being identical to the left-hand side, constitute the application condition of the rule *req*.

4.2 Substitution Morphism

We proceed with the definition of substitution morphisms, consisting of two parts. The first one ensures that the applicability of the requestor rule implies the applicability of the associated provider rule. This is similar to behavior-preserving morphisms (cf. cell 5 in Table 1) except that the application condition is considered instead of the actual rule.

The second part of the definition ensures the reflection of effects. Thus, behavior-reflecting morphisms are appropriate here, but only for those rules of EXP' which are associated to rules of IMP, cf. cell 3 of Table 1.

Below, we first deal with reflection of effects and then with preservation of applicability.

Definition 12 (substitution morphism). *Given typed graph transformation systems $GTS = (TG, P, \pi)$ and $GTS' = (TG', P', \pi')$ containing graph transformation rules with application conditions. A* substitution morphism *$f^{sub} = (f_{TG}, f_P)$ is given by a type graph morphism $f_{TG} : TG \rightarrow TG'$ and a mapping $f_P : P \rightarrow P'$ between the sets of rule names, such that for each $p \in P$ we have*

1. *$\pi(p) \sqsubseteq_{DPB} f^{<}_{TG}(\pi'(p'))$ (cf. Fig. 13 on the right)*
2. *applicability of p implies that of $f_P(p) = p'$, i.e.*
   (a) *for each $f^{>}_{TG}(\hat{l} : L \rightarrow \hat{L}) \in f^{>}_{TG}(AP(p))$ there exist $\hat{l}' : L' \rightarrow \hat{L}' \in AP(p')$ and a graph homomorphisms $h_{\hat{L}'_P} : \hat{L}' \rightarrow f^{>}_{TG}(\hat{L})$ such that the corresponding square in Fig. 13 on the left commutes;*
   (b) *for each $\hat{k}' : L' \rightarrow \hat{L}' \in AN(p')$ there exist $f^{>}_{TG}(\hat{k} : L \rightarrow \hat{L}) \in f^{>}_{TG}(AN(p))$ and a graph homomorphism $h_{\hat{L}'_N} : f^{>}_{TG}(\hat{L}) \rightarrow \hat{L}'$ such that the corresponding square in Fig. 13 on the left commutes.*

The justification for the definition of the substitution morphism is presented in the following theorem.

Theorem 1. *The substitution morphism $f^{sub} = (f_{TG}, f_P)$ satisfies the semantic requirements of Def. 10.*

Proof Sketch. It is necessary to show that Def. 12 implies Def. 10, i.e. (1) transformation steps via a GTS' rule can be considered as transitions via the corresponding GTS rule, and (2) the applicability of this GTS rule implies the

$$\begin{tikzcd}
f^{>}_{TG}(& \hat{L} \arrow[d, bend right, "h_{\hat{L}'_N}"'] & L \arrow[l, "\hat{l}/\hat{k}"'] \arrow[d, "f^{>}_{TG}(f_L)"] &) \\
 & \hat{L}' \arrow[u, bend right, "h_{\hat{L}'_P}"'] \arrow[r, phantom, "="] & L' \arrow[l, "\hat{l}'/\hat{k}'"]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & L \arrow[d, "f_L"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "f_K"] & R \arrow[d, "f_R"] & \\
f^{<}_{TG}(& L' & K' \arrow[l, "l'"] \arrow[r, "r'"'] & R' &)
\end{tikzcd}$$

Fig. 13. Substitution morphism of graph transformation rules (the functors $f^{>}_{TG}$ and $f^{<}_{TG}$ are applied to the entire constraint of p in the left part of figure and to the entire bottom span in the right part of figure correspondingly).

applicability of the GTS' rule. Assume two graph transformation rules with application conditions $\hat{p} = (\pi(p), A(p))$ in GTS and $\hat{p}' = (\pi'(p'), A(p'))$ in GTS' such that $p' = f_P(p)$.

1. It is necessary to demonstrate that each transformation step via the GTS' rule can be reflected by a transition via the GTS rule. By assumption, for each backward retyped rule $f^{<}_{TG}(\hat{p}')$ there is a DPO-/DPB-subrule $\hat{p}$ in GTS, i.e. there exist graph homomorphisms between the first and the second rule (f_L, f_K, f_R), forming a faithful transition (cf. Fig. 14 on the right). Now, both transitions can be vertically composed using the composition of the underlying pushout/pullback squares. The faithfulness of the composed transition follows from the preservation of the identification condition under the composition of pushout/pullback squares. Obviously, both transitions have the same underlying span $G \xleftarrow{g} D \xrightarrow{h} H$.
2. We have to show that if $f^{>}_{TG}(d_L)$ satisfies the application condition of $f^{>}_{TG}(\hat{p})$, then $d_{L'}$ satisfies the application condition of $\hat{p}'$. This induces two problems:
   (a) $d_{L'}$ (cf. Fig. 14 on the left) must *N-satisfy all* negative constraints of $\hat{p}'$, i.e., there must not exist $d_{\hat{L}'} : \hat{L}' \to G'$. This can be proved by assuming existence of $d_{\hat{L}'}$ and showing a contradiction. The full proof of this can be found in [1].
   (b) $d_{L'}$ (cf. Fig. 14 on the left) must *P-satisfy some* positive constraint of $\hat{p}'$. Since the satisfiability of the positive constraints is defined dually to the negative case, the proof is analogous.

   Combining (a) and (b), we obtain that $d_{L'}$ satisfies the application condition of $\hat{p}'$.

Further we discuss the application of the introduced TGTS morphisms.

5 Application of TGTS Morphisms

In this section we revise the *intra-connectors* relating the import/export interfaces and the body of a module and introduce a new concept of *inter-connector* employing the substitution morphism defined in the previous section. The inter-connectors determine a relation between the import and export interfaces of two modules being requestor and provider of a specific service. To illustrate the

$$\begin{tikzcd}
f^{>}_{TG}(\,\hat{L} \arrow[dr, "f^{>}_{TG}(d_{\hat{L}})"] \arrow[dd, bend right, "h_{\hat{L}'_N}"'] & & L\,) \arrow[ll, "\hat{l}/\hat{k}"'] \arrow[dl, "f^{>}_{TG}(d_L)"'] \arrow[dd, "f^{>}_{TG}(f_L)"] \\
 & G' & \\
\hat{L}' \arrow[uu, "h_{\hat{L}'_P}"] \arrow[ur, "d_{\hat{L}'}"] & & L' \arrow[ll, "\hat{l}'/\hat{k}'"] \arrow[ul, "d_{L'}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d, "f_L"'] \arrow[dd, bend right=60, "d_L"'] \arrow[dr, phantom, "PO/PB"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "f_K"'] \arrow[dd, bend left=40, "d_K"] \arrow[dr, phantom, "PO/PB"] & R \arrow[d, "f_R"] \arrow[dd, bend left=60, "d_R"] \\
f^{<}_{TG}(\,L' \arrow[d, "d_{L'}"'] \arrow[dr, phantom, "PO"] & K' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d, "d_{K'}"'] \arrow[dr, phantom, "PO"] & R'\,) \arrow[d, "d_{R'}"] \\
G & D \arrow[l, "g"] \arrow[r, "h"'] & H
\end{tikzcd}$$

Fig. 14. Substitution morphisms satisfy the semantic requirements (the functors $f^{>}_{TG}$ and $f^{<}_{TG}$ are applied to the entire application constraint of p in the left part of figure and to the entire bottom span in the right part of figure correspondingly).

application of the substitution morphism as the inter-connector a module implementing the algorithm for distributed deadlock detection (DDD) is introduced.

5.1 Extended Scenario

The algorithm for distributed deadlock detection is specified by the module MOD' depicted in the lower part of Fig. 15. The upper part of this figure shows the module MOD modeling the algorithm for mutual exclusion discussed in Section 3.1.

The module MOD' offers a deadlock detection service at the export interface EXP' asked for by the module MOD at the import interface IMP (cf. IMP and EXP' in Fig. 15). At the same time, the module MOD' lacks deadlock resolution capabilities provided, in turn, by the module MOD through the export interface EXP (cf. IMP' and EXP in Fig. 15). In general, such a relation between module interfaces, called *cyclic import*, might be problematic for practical realization. However, it properly illustrates different kinds of module connectors.

Example 4 (distributed deadlock detection). The main purpose of MOD' is to observe processes and resources and to detect a deadlock if asked to do so. In a graph representing a system state, a deadlock appears as a cycle of *request* and *held_by* edges, where one process requests a resource held by another process and simultaneously holds a resource requested by it. The distributed deadlock detection uses *blocked* messages, represented by edges with a black flag, in order to detect such cyclic dependencies.

The algorithm is invoked by a process p waiting for a resource r. The process uses rule *dead?* to send a *blocked*-message to r. This feature is offered by MOD' at EXP' for external use, e.g., by MOD. If the resource is held by another process which itself is waiting for a resource, the message is passed on using *waiting*. If this is not the case, which is checked by a negative application condition, the message is deleted by rule *ignore*. Thanks to the mutual exclusion, each resource is held by only one process. Hence, if the message arrives at a resource which is held by the original sender, a cycle has been detected.

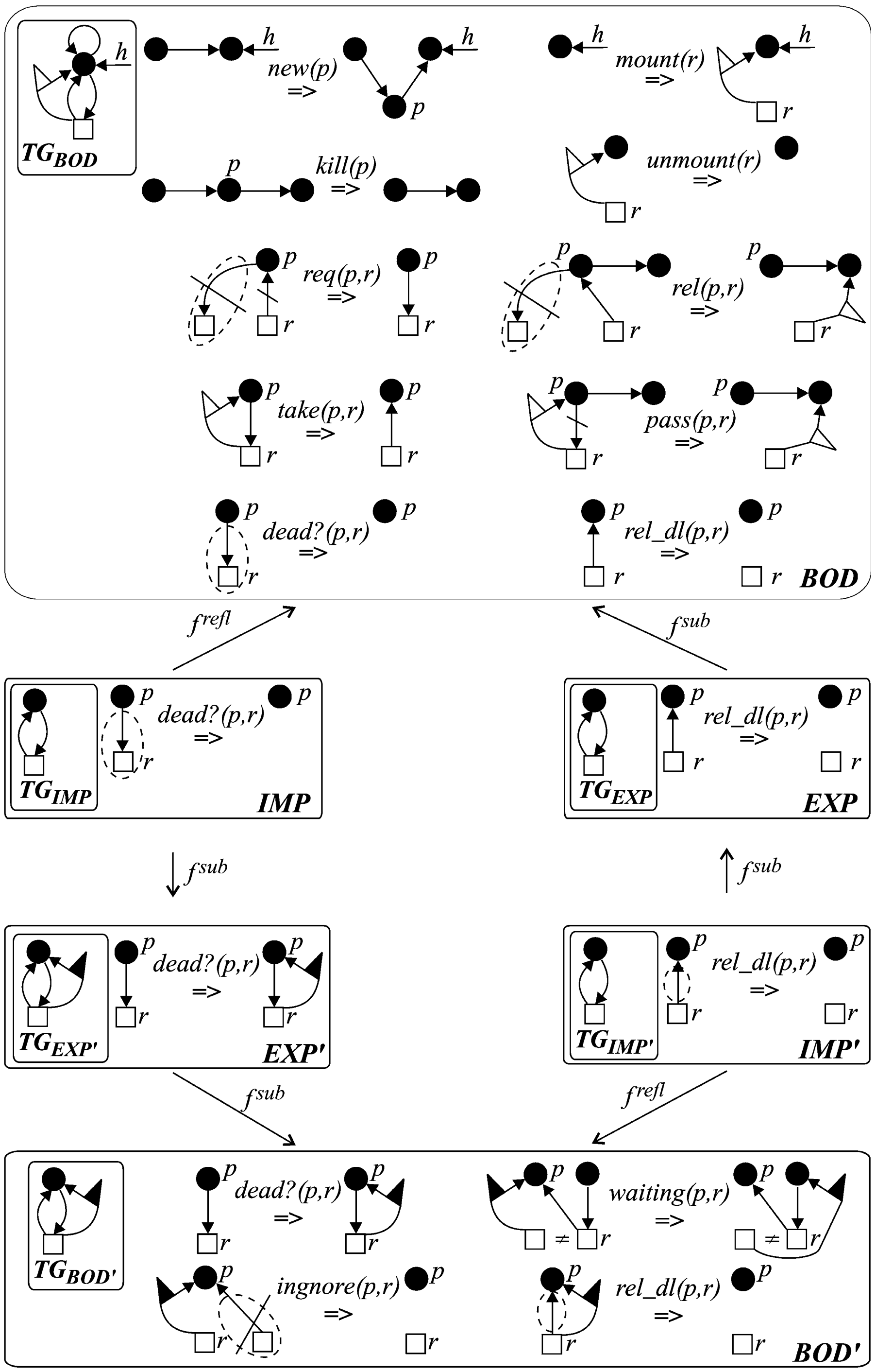

Fig. 15. Modules implementing the algorithms for mutual exclusion (upper) and distributed deadlock detection (lower).

Since MOD' is only destined for deadlock detection, deadlock resolution is described only abstractly by the rule *rel_dl*, which deletes the *blocked*-message, but does not decide how the deadlock is actually resolved. This rule in the import interface IMP' needs to be replaced by the rule of MOD with the same name. The positive application condition of *rel_dl* restricts the rule applicability to the system states where a resource is held by the process, i.e. to the situations being meaningful for the deadlock resolution.

5.2 TGTS Morphisms as Intra-connectors of Modules (Revised)

We proceed with the discussion of intra-connectors relating an import interface and a body of a module. In Section 3.1 behavior-reflecting morphisms were proposed for this purpose. We shall verify whether the requirements of Def. 9 are fulfilled for the corresponding constituents of the modules MOD and MOD' in Fig. 15.

First of all, we establish a type graph morphism f_{TG} and a mapping f_P between the sets of rule names. In the module MOD the type graph TG_{IMP} of the import is a subgraph of TG_{BOD} of the body. Similarly, the type graph $TG_{IMP'}$ is a subgraph of $TG_{BOD'}$ in the module MOD'. The type graph morphisms in both cases are given by inclusions. The corresponding rules in the source and target systems are identified by their names, i.e. *dead?* for IMP and BOD, and *rel_dl* for IMP' and BOD'.

The rule *dead?* in IMP is a subrule of the BOD one, simply because the two rules are identical (cf. Def. 9). The rule *rel_dl* in IMP' is a subrule of the BOD' rule, because the latter becomes identical to the IMP' rule after the backward retyping. Hence, the specifications at IMP and IMP' conform with BOD and BOD' correspondingly.

In contrast with the import-body connector, the requirements towards the connector between export interface and body shall be strengthened. Behavior-preservation guarantees that the applicability of rules in the export interface implies the applicability of corresponding body rules. However, this property would be satisfied even for empty body rules. In fact, we also require that the effect achieved by the body rules is at least the one promised by the rules in the export interface. Hence, we "upgrade" behavior-preserving morphisms to substitution morphisms. Next we shall demonstrate that the relations between exports and bodies of the modules in Fig 15 are indeed substitution morphism.

To show this one should check preservation of applicability from the export interface to the body and reflection of the effects between the rules in the body and the export interface. The rules *rel_dl* in the export interface EXP of MOD and *dead?* in EXP' of MOD' are identical to the body rules of the modules, and so the properties required by Def 12 obviously hold.

5.3 TGTS Morphisms as Inter-connectors of Modules

The ultimate aim of matching import and export interfaces of requestor and provider modules is to check whether the corresponding rules in the body of the former can be safely substituted for the rules in the body of the later. That means the obvious choice of morphism for this inter-connector is the substitution morphism.

Let us first discuss the relation between the import interface IMP and the export interface EXP' of the modules in Fig 15. The type graph morphism f_{TG} from TG_{IMP} to $TG_{EXP'}$ is given by an inclusion. The mapping f_P between the sets of rule names is unique, because only one rule *dead?* is contained in each of the interfaces.

After that, one should check the preservation of applicability from IMP to EXP' (cf. Def 12). Each of the rules has one positive application condition being the union of the left-hand side and the dotted part for the IMP rule, and coinciding with the left-hand side for the EXP' rule. The application conditions of the two rules are the same because the forward retyping of the EXP' rule does not introduce any changes. Applicability is thus preserved.

The last step is the reflection of effects. While the backward retyping of the EXP' rule gets rid of the *blocked*-message, it is still bigger in context and effect then the IMP rule. This is allowed by the DPB-subrule relation which can be established between the two rules. Thereby, the import interface IMP is associated with the export interface EXP' by a substitution morphism.

Now we discuss the relation between the import interface IMP' and the export interface EXP. The type graphs $TG_{IMP'}$ and TG_{EXP} of the two systems are the same, consequently the retyping does not change the rules. The positive application conditions of the rules *rel_dl* coincide, that means preservation of applicability from IMP' to EXP. The reflection of effects is ensured by the DPO-subrule relation between the two rules in spite of the bigger context of the EXP' rule additionally containing the *held_by* edge. Hence, the import interface IMP' and the export interface EXP are also connected by a substitution morphism.

The fact that import-export and export-body relations are both described by substitution morphisms allows us, by means of their composition, to consider the body of the provider module as a replacement for the export of the requestor. This is the first prerequisite for a composition of modules, i.e., the actual substitution of the import by the body. The detailed analysis of this construction is, however, beyond the scope of this paper.

The final section summarizes the main results of our work.

6 Conclusion

The contributions of this paper can be summarized in two points: a systematic presentation of morphisms of graph transformation systems along with a recipe of how to define new variants, if needed, in a generic framework; and a novel notion of substitution morphism between graph transformation systems with application conditions being uniformly introduced in the context of this framework.

The latter has been motivated by the need to connect import and export interfaces of modules in a flexible way, i.e., such that they can be developed independently of each other. The first result is a reaction to the multitude of proposals and variants that exist in the literature.

Future work will include the further analysis of modules based on the connectors introduced here, in particular their composition, as well as possible generalizations towards refinements of both the general framework and the notion of substitution morphism.

References

1. A. Cherchago and R. Heckel. Specification matching of web services using conditional graph transformation rules. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, volume 3256 of *LNCS*. Springer-Verlag, 2004.
2. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 56–74. Springer-Verlag, 1996.
3. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
4. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997. Preprint available as Tech. Rep. 96/17, Univ. of Pisa, `http://www.di.unipi.it/TR/TRengl.html`.
5. H. Ehrig and G. Engels. Pragmatic and semantic aspects of a module concept for graph transformation systems. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, LNCS, pages 137–154. Springer-Verlag, 1996.
6. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
7. H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 2: module specifications and constraints*. Springer-Verlag, 1990.
8. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
9. M. Große–Rhode, F. Parisi Presicce, and M. Simeoni. Concrete spatial refinement construction for graph transformation systems. Technical Report SI 97/10, Università di Roma La Sapienza, Dip. Scienze dell'Informazione, 1997.
10. M. Große–Rhode, M. Simeoni, and F. Parisi Presicce. Refinements and modules for typed graph transformation systems. In J.L.Fiadeiro, editor, *Proc. WADT'98 (Workshop on Algebraic Development Techniques), at ETAPS'98, Lisbon, April*, number 1589 in LNCS, pages 138 – 151. Springer, 1999.
11. M. Grosse-Rhode, F. Parisi-Presicce, and M. Simeoni. Spatial and temporal refinement of graph transformation systems. In *Proc. of Mathematical Foundations of Computer Science 1998*, volume 1450 of *LNCS*, pages 553–561. Springer-Verlag, 1998.

12. M. Grosse-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinements and modules for typed graph transformation systems. In J.L. Fiadeiro, editor, *Proc. Workshop on Algebraic Development Techniques (WADT'98), at ETAPS'98, Lisbon, April 1998*, volume 1589 of *LNCS*, pages 138–151. Springer-Verlag, 1999.
13. M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*, pages 368–382. Springer-Verlag, 2000.
14. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287 – 313, 1996.
15. J.H. Hausmann, R. Heckel, and M. Lohmann. Model-based discovery of web services. In *Proc. International Conference on Web Services*, San Diego, USA, July 2004.
16. R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems.* PhD thesis, TU Berlin, 1998.
17. R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996.
18. R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, 9(1), January 2001. See also TR 97-07 at `http://www.cs.tu-berlin.de/cs/ifb/TechnBerichteListe.html`.
19. R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. Classification and comparison of modularity concepts for graph transformation systems. In Ehrig et al. [6], pages 669 – 690.
20. R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. A view-based approach to system modelling based on open graph transformation systems. In Ehrig et al. [6], pages 639 – 667.
21. H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units - a step into GRACE. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 89 – 106. Springer-Verlag, 1996.
22. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall International, 1988.
23. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, TU Berlin, 1996.
24. A. Schürr and A.J. Winter. UML packages for PROgrammed Graph REwrite Systems. In *Selected Papers of 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98), Paderborn, Germany*, volume 1764 of *LNCS*, pages 396–409. Springer-Verlag, 1999.
25. A.M. Zaremski and J.M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(2):146 – 170, April 1995.
26. A.M. Zaremski and J.M. Wing. Specification matching of software components. In *Proc. SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 20(4) of *ACM SIGSOFT Software Engineering Notes*, pages 6–17, October 1995. Also CMU-CS-95-127, March, 1995.