Behaviour Protection in Modular Rule-Based System Specifications

Francisco Durán[1], Fernando Orejas[2], and Steffen Zschaler[3]

[1] Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
duran@lcc.uma.es
[2] Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
orejas@lsi.upc.edu
[3] Department of Informatics
King's College London
szschaler@acm.org

Abstract. Model-driven engineering (MDE) and, in particular, the notion of domain-specific modelling languages (DSMLs) is an increasingly popular approach to systems development. DSMLs are particularly interesting because they allow encoding domain-knowledge into a modelling language and enable full code generation and analysis based on high-level models. However, as a result of the domain-specificity of DSMLs, there is a need for many such languages. This means that their use only becomes economically viable if the development of new DSMLs can be made efficient. One way to achieve this is by reusing functionality across DSMLs. On this background, we are working on techniques for modularising DSMLs into reusable units. Specifically, we focus on DSMLs whose semantics are defined through in-place model transformations. In this paper, we present a formal framework of morphisms between graph-transformation systems (GTSs) that allow us to define a novel technique for conservative extensions of such DSMLs. In particular, we define different behaviour-aware GTS morphisms and prove that they can be used to define conservative extensions of a GTS.

1 Introduction

Model-Driven Engineering (MDE) [41] has raised the level of abstraction at which systems are developed, moving development focus from the programming-language level to the development of software models. Models and specifications of systems have been around the software industry from its very beginning, but MDE articulates them so that the development of information systems can be at least partially automated. Thus models are being used not only to specify systems, but also to simulate, analyze, modify and generate code of such systems. A particularly useful concept in MDE are domain-specific modelling languages (DSMLs) [7]. These languages offer concepts specifically targeted at a particular domain. On the one hand this makes it easier for domain experts to express their problems and requirements. On the other hand, the higher amount of knowledge embedded in each concept allows for much more complete generation of

executable solution code from a DSML model [27] as compared to a model expressed in a general-purpose modelling language.

DSMLs can only be as effective as they are specific for a particular domain. This implies that there is a need for a large number of such languages to be developed. However, development of a DSML takes additional effort in a software-development project. DSMLs are only viable if their development can be made efficient. One way of achieving this is by allowing them to be built largely from reusable components. Consequently, there has been substantial research on how to modularise language specifications. DSMLs are often defined by specifying their syntax (often separated into concrete and abstract syntax) and their semantics. While we have reasonably good knowledge of how to modularise DSML syntax, the modularisation of language semantics is an as yet unsolved issue.

DSML semantics can be represented in a range of different ways—for example using UML behavioural models [17, 20], abstract state machines [8, 2], Kermeta [34], or in-place model transformations [33, 37]. In the context of MDE it seems natural to describe the semantics by means of models, so that they may be integrated with the rest of the MDE environment and tools. We focus on the use of in-place model transformations.

Graph transformation systems (GTSs) were proposed as a formal specification technique for the rule-based specification of the dynamic behaviour of systems [10]. Different approaches exist for modularisation in the context of the graph-grammar formalism [5, 40, 12]. All of them have followed the tradition of modules inspired by the notion of algebraic specification module [15]. A module is thus typically considered as given by an export and an import interface, and an implementation body that realises what is offered in the export interface, using the specification to be imported from other modules via the import interface. For example, Große-Rhode, Parisi-Presicce, and Simeoni introduce in [22] a notion of *module* for typed graph transformation systems, with interfaces and implementation bodies; they propose operations for union, composition, and refinement of modules. Other approaches to modularisation of graph transformation systems include PROGRES Packages [42], GRACE Graph Transformation Units and Modules [31], and DIEGO Modules [43]. See [26] for a discussion on these proposals.

For the kind of systems we deal with, the type of module we need is much simpler. For us, a module is just the specification of a system, a GTS, without import and export interfaces. Then, we build on GTS morphisms to compose these modules, and specifically we define parametrised GTSs. The instantiation of such parameterized GTS is then provided by an amalgamation construction. We present formal results about graph-transformation systems and morphisms between them. Specifically, we provide definitions for behaviour-reflecting and -protecting GTS morphisms and show that they can be used to infer semantic properties of these morphisms. We give a construction for the amalgamation of GTSs, as a base for the composition of GTSs, and we prove it to protect behaviour under certain circumstances. Although we motivate and illustrate our approach using the *e-Motions* language [38, 39], our proposal is language-independent, and all the results are presented for GTSs and adhesive HLR systems [32, 14].

Different forms of GTS morphisms have been used in the literature, taking one form or another depending on their concrete application. Thus, we find proposals cen-

tered on refinements (see., e.g., [25, 21, 22]), views (see, e.g., [19]), and substitutability (see [18]). See [18] for a first attempt to a systematic comparison of the different proposals and notations. None of these notions fit our needs, and none of them coincide with our behaviour-aware GTS morphisms.

Moreover, as far as we know, parameterised GTSs and GTS morphisms, as we discuss them, have not been studied before. Heckel and Cherchago introduce parameterised GTSs in [24], but their notion has little to do with our parameterised GTSs. In their case, the parameter is a signature, intended to match service descriptions. They however use a double-pullback semantics, and have a notion of substitution morphism which is related to our behaviour preserving morphism.

Our work is originally motivated by the specification of non-functional properties (NFPs), such as performance or throughput, in DSMLs. We have been looking for ways in which to encapsulate the ability to specify non-functional properties into reusable DSML modules. Troya et al. used the concept of observers in [45, 46] to model non-functional properties of systems described by GTSs in a way that could be analysed by simulation. In [9], we have built on this work and ideas from [48] to allow the modular encapsulation of such observer definitions in a way that can be reused in different DSML specifications. In this paper, we present a full formal framework of such language extensions. Nevertheless, this framework is independent of the specific example of non-functional property specifications, but instead applies to any conservative extension of a base GTS.

The way in which we think about composition of reusable DSML modules has been inspired by work in aspect-oriented modelling (AOM). In particular, our ideas for expressing parametrised metamodels are based on the proposals in [3, 29]. Most AOM approaches use syntactic notions to automate the establishment of mappings between different models to be composed, often focusing primarily on the structural parts of a model. While our mapping specifications are syntactic in nature, we focus on composition of behaviours and provide semantic guarantees. In this sense, our work is perhaps most closely related to the work on MATA [47] or semantic-based weaving of scenarios [30].

The rest of the paper begins with a presentation of a motivating example expressed in the *e-Motions* language in Section 2. Section 3 introduces a brief summary of graph transformation and adhesive HLR categories. Section 4 introduces behaviour-reflecting GTS morphisms, the construction of amalgamations in the category of GTSs and GTS morphisms, and several results on these amalgamations, including the one stating that the morphisms induced by these amalgamations protect behaviour, given appropriate conditions. The paper finishes with some conclusions and future work in Section 5.

2 NFP specification with *e-Motions*

In this section, we use *e-Motions* [38, 39] to provide a motivating example, adapted from [45], as well as intuitions for the formal framework developed. However, as stated in the previous section, the framework itself is independent of such a language.

e-Motions is a Domain Specific Modeling Language (DSML) and graphical framework developed for Eclipse that supports the specification, simulation, and formal anal-

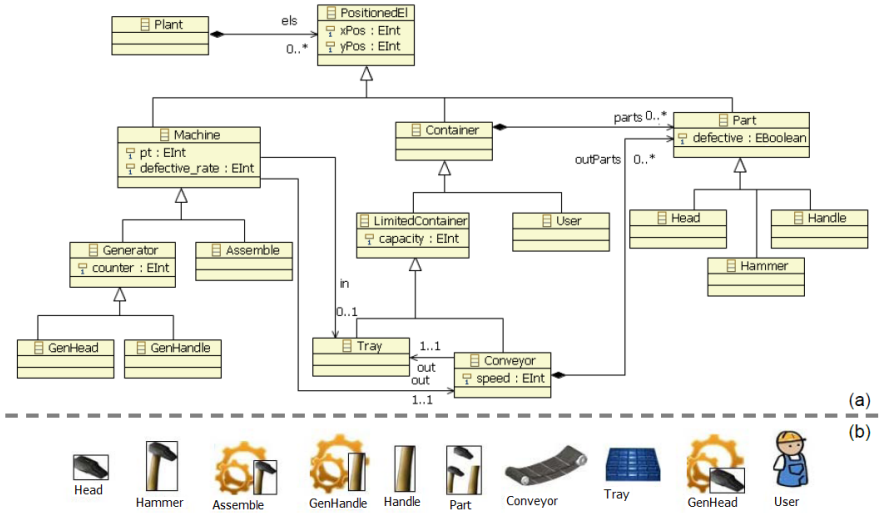


Fig. 1. Production Line (a) metamodel and (b) concrete syntax (from [45]).

ysis of DSMLs. Given a MOF metamodel (abstract syntax) and a GCS model (a graphical concrete syntax) for it, the behaviour of a DSML is defined by in-place graph transformation rules. Although we briefly introduce the language here, we omit all those details not relevant to this paper. We refer the interested reader to [36, 39] or `http://atenea.lcc.uma.es/e-Motions` for additional details.

Figure 1(a) shows the metamodel of a DSML for specifying Production Line systems for producing hammers out of hammer heads and handles, which are generated in respective machines, transported along the production line via conveyors, and temporarily stored in trays. As usual in MDE-based DSMLs, this metamodel defines all the concepts of the language and their interconnections; in short, it provides the language's *abstract* syntax. In addition, a *concrete* syntax is provided. In the case of our example, this is sufficiently well defined by providing icons for each concept (see Figure 1(b)); connections between concepts are indicated through arrows connecting the corresponding icons. Figure 2 shows a model conforming to the metamodel in Figure 1(a) using the graphical notation introduced in the GCS model in Figure 1(b).

The behavioural semantics of the DSML is then given by providing transformation rules specifying how models can evolve. Figure 3 shows an example of such a rule. The rule consists of a left-hand side matching a situation before the execution of the rule and a right-hand side showing the result of applying the rule. Specifically, this rule shows how a new hammer is assembled: a hammer generator a has an incoming tray of parts and is connected to an outgoing conveyor belt. Whenever there is a handle and a head available, and there is space in the conveyor for at least one part, the hammer generator can assemble them into a hammer. The new hammer is added to the parts set of the outgoing conveyor belt in time T, with T some value in the range [a.pt - 3, a.pt + 3], and where pt is an attribute representing the production time of a machine. The complete semantics of our production-line DSML is constructed from a number

of such rules covering all kinds of atomic steps that can occur, e.g., generating new pieces, moving pieces from a conveyor to a tray, etc. The complete specification of a Production Line example using *e-Motions* can be found at `http://atenea.lcc.uma.es/E-motions/PLSExample`.

For a Production Line system like this one, we may be interested in a number of non-functional properties. For example, we would like to assess the throughput of a production line, or how long it takes for a hammer to be produced. Figure 4(a) shows

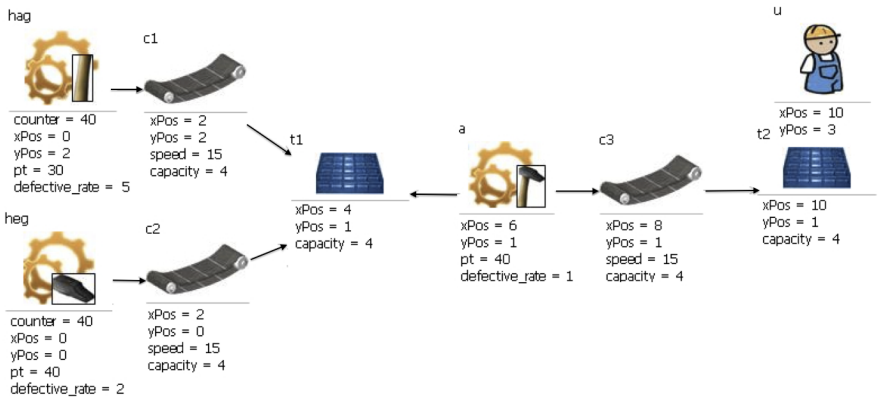


Fig. 2. Example of production line configuration.

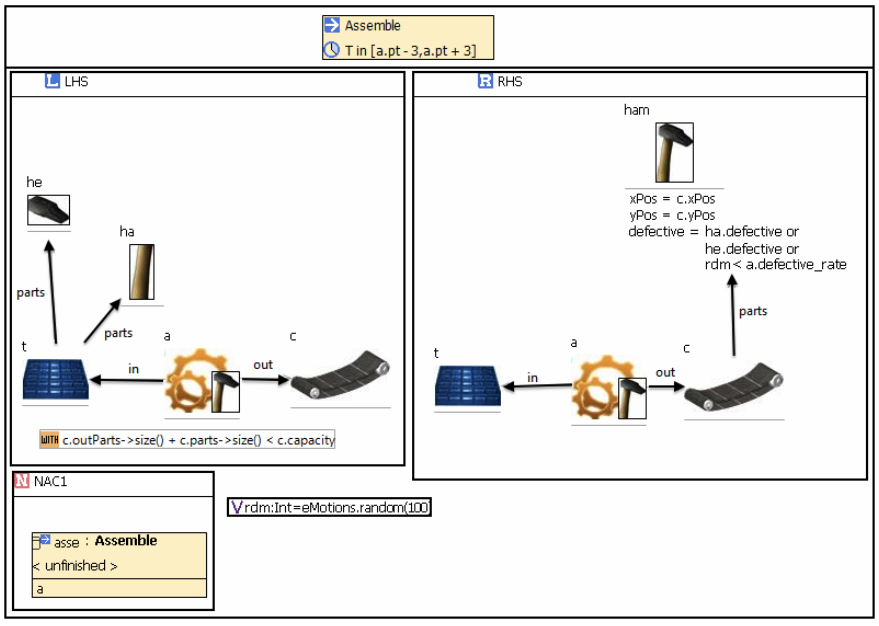
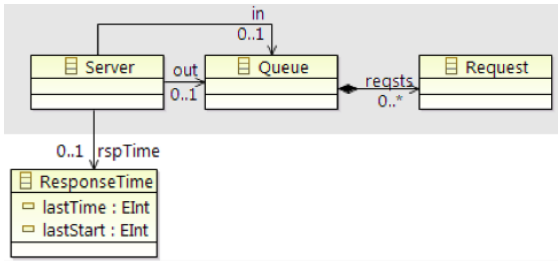
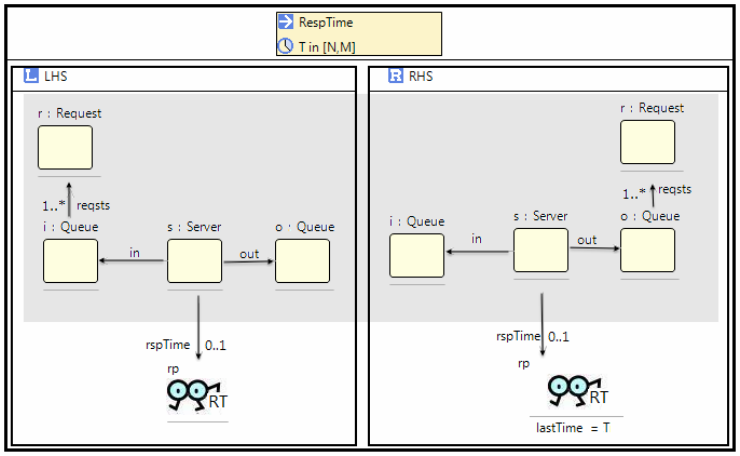


Fig. 3. Assemble rule indicating how a new hammer is assembled (from [45]).



(a) Metamodel.

(b) Concrete syntax.



(c) Sample response time rule.

Fig. 4. Generic model of response time observer.

the metamodel for a DSML for specifying production time. It is defined as a parametric model (i.e., a model template), defined independently of the Production Line system. It uses the notion of response time, which can be applied to different systems with different meanings. The concepts of Server, Queue, and Request and their interconnections are parameters of the metamodel, and they are shaded in grey for illustration purposes. Figure 4(b) shows the concrete syntax for the response time observer object. Whenever that observer appears in a behavioural rule, it will be represented by that graphical symbol.

Figure 4(c) shows one of the transformation rules defining the semantics of the response time observer. It states that if there is a server with an in queue and an out queue and there initially are some requests (at least one) in the in queue, and the out queue contains some requests after rule execution, the last response time should be recorded to have been equal to the time it took the rule to execute. Similar rules need to be written to capture other situations in which response time needs to be measured, for example, where a request stays at a server for some time, or where a server does not have an explicit in or out queue.

Note that, as in the metamodel in Figure 4(a), part of the rule in Figure 4(c) has been shaded in grey. Intuitively, the shaded part represents a pattern describing transformation rules that need to be extended to include response-time accounting.[4] The lower part of the rule describes the extensions that are required. So, in addition to reading Figure 4(c) as a 'normal' transformation rule (as we have done above), we can also read it as a *rule transformation*, stating: "Find all rules that match the shaded pattern and add ResponseTime objects to their left- and right-hand sides as described." In effect, observer models become higher-order transformations [44].

To use our response-time language to allow specification of production time of hammers in our Production Line DSML, we need to weave the two languages together. For this, we need to provide a binding from the parameters of the response-time metamodel (Figure 4(a)) to concepts in the Production Line metamodel (Figure 1(a)). In this case, assuming that we are interested in measuring the response time of the Assemble machine, the binding might be as follows:

- Server to Assemble;
- Queue to LimitedContainer as the Assemble machine is to be connected to an arbitrary LimitedContainer for queuing incoming and outgoing parts;
- Request to Part as Assemble only does something when there are Parts to be processed; and
- Associations:
  - The in and out associations from Server to Queue are bound to the corresponding in and out associations from Machine to Tray and Conveyor, respectively; and
  - The association from Queue to Request is bound to the association from Container to Part.

As we will see in Section 4, given DSMLs defined by a metamodel plus a behaviour, the weaving of DSMLs will correspond to amalgamation in the category of DSMLs and DSML morphisms. Figure 5 shows the amalgamation of an inclusion morphism between the model of an observer DSML, M_{Obs}, and its parameter sub-model M_{Par}, and the binding morphism from M_{Par} to the DSML of the system at hand, M_{DSML}, the Production Line DSML in our example. The amalgamation object $M_{\widehat{DSML}}$ is obtained by the construction of the amalgamation of the corresponding metamodel morphisms and the amalgamation of the rules describing the behaviour of the different DSMLs.

In our example, the amalgamation of the metamodel corresponding morphisms is shown in Figure 6 (note that the binding is only partially depicted). The weaving process has added the ResponseTime concept to the metamodel. Notice that the weaving process also ensures that only sensible woven metamodels can be produced: for a given binding of parameters, there needs to be a match between the constraints expressed in the observer metamodel and the DSML metamodel. We will discuss this issue in more formal detail in Section 4.

The binding also enables us to execute the rule transformations specified in the observer language. For example, the rule in Figure 3 matches the pattern in Figure 4(c),

[4] Please, notice the use of the cardinality constraint $1..*$ in the rule in Figure 4(c). It is out of the scope of this paper to discuss the syntactical facilities of the *e-Motions* system.

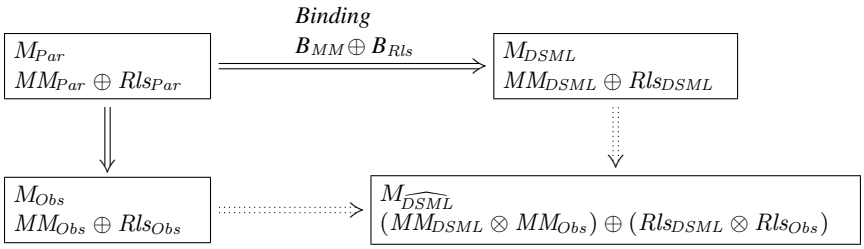


Fig. 5. Amalgamation in the category of DSMLs and DSML morphisms.

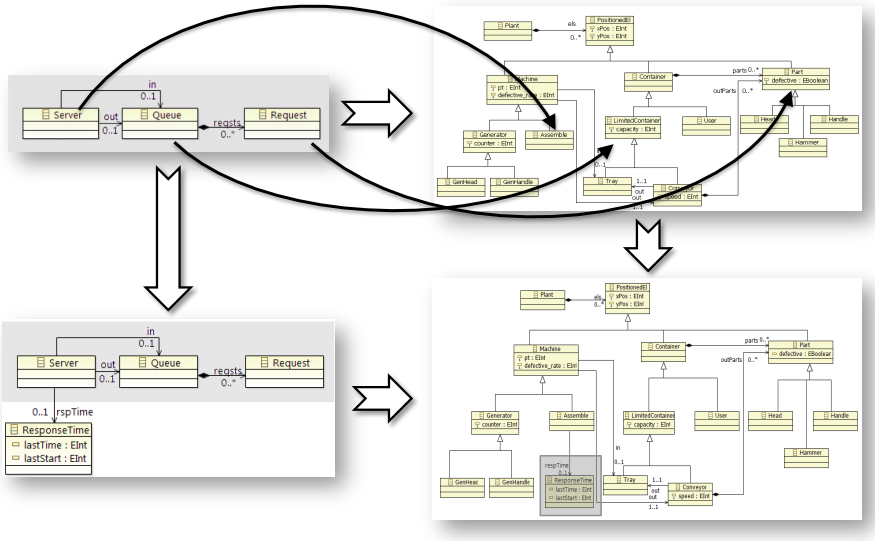


Fig. 6. Weaving of metamodels (highlighting added for illustration purposes).

given this binding: In the left-hand side, there is a Server (Assemble) with an in-Queue (Tray) that holds two Requests (Handle and Head) and an out-Queue (Conveyor). In the right-hand side, there is a Server (Assemble) with an in-Queue (Tray) and an out-Queue (Conveyor) that holds one Request (Hammer). Consequently, we can apply the rule transformation from the rule in Figure 4(c). As we will explain in Section 4, the semantics of this rule transformation is provided by the rule amalgamation illustrated in Figure 7, where we can see how the obtained amalgamated rule is similar to the Assemble rule but with the observers in the RespTime rule appropriately introduced.

Clearly, such a separation of concerns between a specification of the base DSML and specifications of languages for non-functional properties is desirable. We have used the response-time property as an example here. Other properties can be defined easily

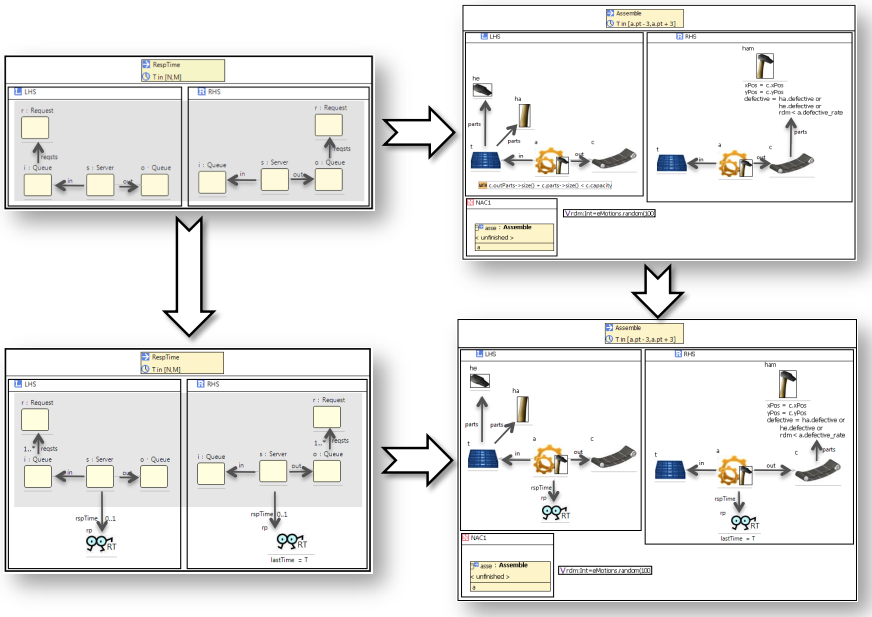


Fig. 7. Amalgamation of the Assemble and RespTime rules.

in a similar vein as shown in [45] and at `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions/PLSObExample`. In the following sections, we discuss the formal framework required for this and how we can distinguish safe bindings from unsafe ones.

The *e-Motions* models thus obtained are automatically transformed into Maude [4] specifications [39]. See [36] for a detailed presentation of how Maude provides an accurate way of specifying both the abstract syntax and the behavioral semantics of models and metamodels, and offers good tool support both for simulating and for reasoning about them.

3 Graph transformation and adhesive HLR categories

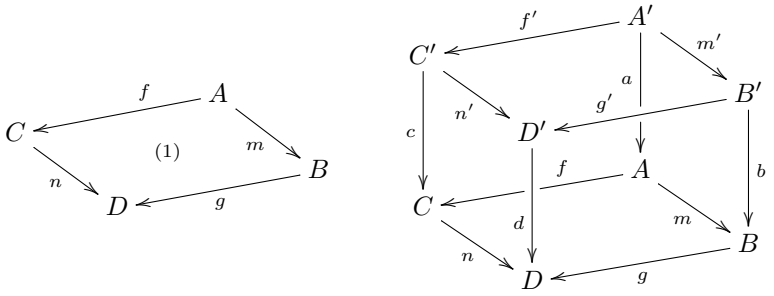
Graph transformation [40] is a formal, graphical and natural way of expressing graph manipulation based on rewriting rules. In graph-based modelling (and meta-modelling), graphs are used to define the static structures, such as class and object ones, which represent visual alphabets and sentences over them. We formalise our approach using the typed graph transformation approach, specifically the Double Pushout (DPO) algebraic approach, with positive and negative (nested) application conditions [11, 23]. We however carry on our formalisation for weak adhesive high-level replacement (HLR) categories [12]. Some of the proofs in this paper assume that the category of graphs at hand

is adhesive HLR. Thus, in the rest of the paper, when we talk about graphs or typed graphs, keep in mind that we actually mean some type of graph whose corresponding category is adhesive HLR. Specifically, the category of typed attributed graphs, the one of interest to us, was proved to be adhesive HLR in [16].

3.1 Generic notions

The concepts of adhesive and (weak) adhesive HLR categories abstract the foundations of a general class of models, and come together with a collection of general semantic techniques [32, 14]. Thus, e.g., given proofs for adhesive HLR categories of general results such as the Local Church-Rosser, or the Parallelism and Concurrency Theorem, they are automatically valid for any category which is proved an adhesive HLR category. This framework has been a break-through for the DPO approach of algebraic graph transformation, for which most main results can be proved in these categorical frameworks, and then instantiated to any HLR system.

Definition 1. *(**Van Kampen square**) Pushout (1) is a van Kampen square if, for any commutative cube with (1) in the bottom and where the back faces are pullbacks, we have that the top face is a pushout if and only if the front faces are pullbacks.*



Definition 2. *(**Adhesive HLR category**) A category **C** with a morphism class M is called adhesive HLR category if*

- *M is a class of monomorphisms closed under isomorphisms and closed under composition and decomposition,*
- ***C** has pushouts and pullbacks along M-morphisms, i.e., if one of the given morphisms is in M, then also the opposite one is in M, and M-morphisms are closed under pushouts and pullbacks, and*
- *pushouts in **C** along M-morphisms are van Kampen squares.*

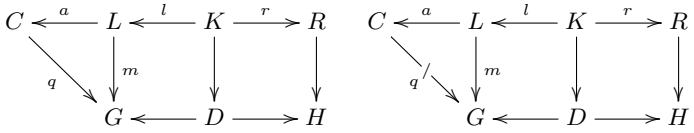
In the DPO approach to graph transformation, a rule p is given by a span $(L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ with graphs L, K, and R, called, respectively, left-hand side, interface, and right-hand side, and some kind of monomorphisms (typically, inclusions) l and r. A graph transformation system (GTS) is a pair (P, π) where P is a set of rule names and π is a function mapping each rule name p into a rule $L \stackrel{l}{\longleftarrow} K \stackrel{r}{\longrightarrow} R$.

An application of a rule $p : L \xleftarrow{l} K \xrightarrow{r} R$ 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 $G \overset{p,m}{\Longrightarrow} H$.

$$
\begin{array}{ccccc}
p & : & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 & & {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\
 & & G & \xleftarrow{} & D & \xrightarrow{} & H
\end{array}
$$

We only consider injective matches, that is, monomorphisms. If the matching m is understood, a DPO transformation step $G \overset{p,m}{\Longrightarrow} H$ will be simply written $G \overset{p}{\Longrightarrow} H$. A transformation sequence $\rho = \rho_1 \dots \rho_n : G \Rightarrow^* H$ via rules $p_1, \dots, p_n$ is a sequence of transformation steps $\rho_i = (G_i \overset{p_i,m_i}{\Longrightarrow} H_i)$ 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 an adhesive category **C**, denoted by **Trf**(**C**), has all graphs in $|\mathbf{C}|$ as objects and all transformation sequences as arrows.

Transformation rules may have application conditions. We consider rules of the form $(L \xleftarrow{l} K \xrightarrow{r} R, ac)$, where $(L \xleftarrow{l} K \xrightarrow{r} R)$ is a normal rule and ac is a *(nested) application condition* on L. Application conditions may be positive or negative (see Figure 8). Positive application conditions have the form $\exists a$, for a monomorphism $a : L \to C$, and demand a certain structure in addition to L. Negative application conditions of the form $\nexists a$ forbid such a structure. A match $m : L \to G$ satisfies a positive application condition $\exists a$ if there is a monomorphism $q : C \to G$ satisfying $q \circ a = m$. A matching m satisfies a negative application condition $\nexists a$ if there is no such monomorphism.



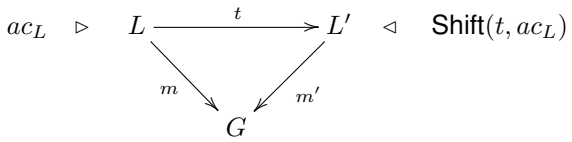
(a) Positive application condition (b) Negative application condition

Fig. 8. Positive and negative application conditions.

Given an application condition $\exists a$ or $\nexists a$, for a monomorphism $a : L \to C$, another application condition ac can be established on C, giving place to nested application conditions [23]. For a basic application condition $\exists(a, ac_C)$ on L with an application condition ac_C on C, in addition to the existence of q it is required that q satisfies ac_C. We write $m \models ac$ if m satisfies ac. $ac_C \cong ac'_C$ denotes the semantical equivalence of ac_C and ac'_C on C.

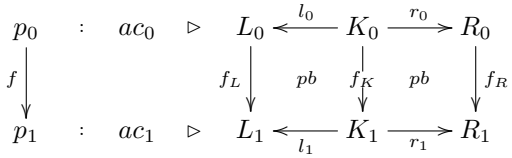
To improve readability, we assume projection functions *ac*, *lhs* and *rhs*, returning, respectively, the application condition, the left-hand side and the right-hand side of a rule. Thus, given a rule $r = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$, $ac(r) = ac$, $lhs(r) = L$, and $rhs(r) = R$.

Given an application condition ac_L on L and a monomorphism $t : L \to L'$, then there is an application condition $\mathsf{Shift}(t, ac_L)$ on L' such that for all $m' : L' \to G$, $m' \models \mathsf{Shift}(t, ac_L) \leftrightarrow m = m' \circ t \models ac_L$.



Parisi-Presicce proposed in [35] a notion of rule morphism very similar to the one below, although we consider rules with application conditions, and require the commuting squares to be pullbacks.

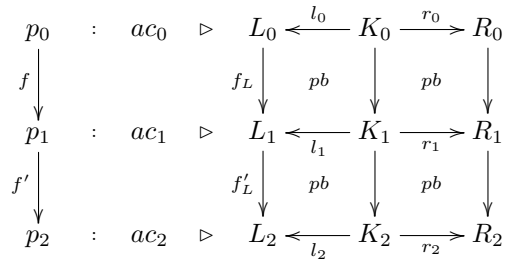
Definition 3. ***(Rule morphism)*** *Given transformation rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$, for $i = 0, 1$, a rule morphism $f : p_0 \to p_1$ is a tuple $f = (f_L, f_K, f_R)$ of graph monomorphisms $f_L : L_0 \to L_1$, $f_K : K_0 \to K_1$, and $f_R : R_0 \to R_1$ such that the squares with the span morphisms l_0, l_1, r_0, and r_1 are pullbacks, as in the diagram below, and such that $ac_1 \Rightarrow \mathsf{Shift}(f_L, ac_0)$.*



The requirement that the commuting squares are pullbacks is quite natural from an intuitive point of view: the intuition of morphisms is that they should preserve the "structure" of objects. If we think of rules not as a span of monomorphisms, but in terms of their intuitive semantics (i.e., L\K is what should be deleted from a given graph, R\K is what should be added to a given graph and K is what should be preserved), then asking that the two squares are pullbacks means, precisely, to preserve that structure. I.e., we preserve what should be deleted, what should be added and what must remain invariant. Of course, pushouts also preserve the created and deleted parts. But they reflect this structure as well, which we do not want in general.

Fact 1 *With componentwise identities and composition, rule morphisms define the category* $\mathbf{Rule}$.

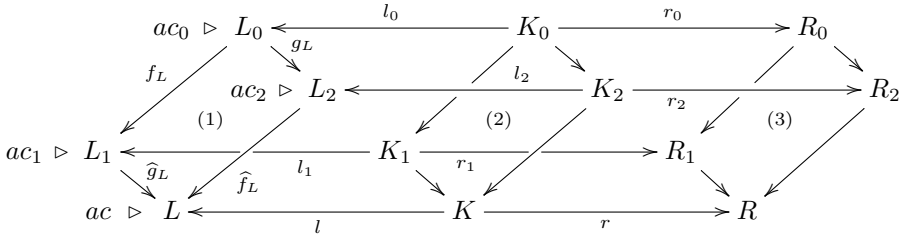
Proof Sketch. Follows trivially from the fact that $ac \cong \mathsf{Shift}(id_L, ac)$, pullback composition, and that given morphisms $f' \circ f$ such that



then we have $\mathsf{Shift}(f'_L, \mathsf{Shift}(f_L, ac_0)) \cong \mathsf{Shift}(f'_L \circ f_L, ac_0)$. □

A key concept in the constructions in the following section is that of *rule amalgamation* [1, 12]. The amalgamation of two rules p_1 and p_2 glues them together into a single rule $\tilde{p}$ to obtain the effect of the original rules. I.e., the simultaneous application of p_1 and p_2 yields the same successor graph as the application of the amalgamated rule $\tilde{p}$. The possible overlapping of rules p_1 and p_2 is captured by a rule p_0 and rule morphisms $f : p_0 \to p_1$ and $g : p_0 \to p_2$.

Definition 4. ***(Rule amalgamation)*** *Given transformation rules $p_i : (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$, for $i = 0, 1, 2$, and rule morphisms $f : p_0 \to p_1$ and $g : p_0 \to p_2$, the amalgamated production $p_1 +_{p_0} p_2$ is the production $(L \xleftarrow{l} K \xrightarrow{r} R, ac)$ in the diagram below, where subdiagrams (1), (2) and (3) are pushouts, l and r are induced by the universal property of (2) so that all subdiagrams commute, and $ac = \mathsf{Shift}(\widehat{f}_L, ac_2) \wedge \mathsf{Shift}(\widehat{g}_L, ac_1)$.*



Notice that in the above diagram all squares are either pushouts or pullbacks (by the van Kampen property) which means that all their arrows are monomorphisms (by being an adhesive HLR category).

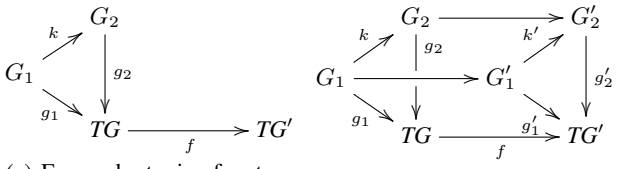
We end this section by introducing the notion of rule identity.

Definition 5. ***(Rule-identity morphism)*** *Given graph transformation rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$, for $i = 0, 1$, and a rule morphism $f : p_0 \to p_1$, with $f = (f_L, f_K, f_R)$, p_0 and p_1 are said to be identical, denoted $p_0 \equiv p_1$, if f_L, f_K, and f_R are identity morphisms and $ac_0 \cong ac_1$.*

3.2 Typed graph transformation systems

A (directed unlabeled) *graph* $G = (V, E, s, t)$ is given by a set of nodes (or vertices) V, a set of edges E, and source and target functions $s, t : E \to V$. Given graphs $G_i = (V_i, E_i, s_i, t_i)$, with $i = 1, 2$, a graph homomorphism $f : G_1 \to G_2$ is a pair of functions $(f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. With componentwise identities and composition this defines the category **Graph**.

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*. However, to enhance readability, we will use simply g_G to denote a typed graph (G, g_G), and when



(a) Forward retyping functor. (b) Backward retyping functor.

Fig. 9. Forward and backward retyping functors.

the typing morphism g_G can be considered implicit, we will often refer to it 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 $f : (G_1, g_1) \to (G_2, g_2)$ (or simply $f : g_1 \to g_2$), is a graph morphism $f : G_1 \to G_2$ which preserves types, i.e., $g_2 \circ f = g_1$. $\mathbf{Graph}_{TG}$ is the category of *TG*-typed graphs and *TG*-typed graph morphisms, which is the comma category **Graph** over *TG*.

If the underlying graph category is adhesive (resp., adhesive HLR, weakly adhesive) then so are the associated typed categories [12], and therefore all definitions in Section 3.1 apply to them. A *TG*-typed graph transformation rule is a span $p = L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ of injective *TG*-typed graph morphisms and a (nested) application condition on L. Given *TG*-typed graph transformation rules $p_i = (L_i \stackrel{l_i}{\leftarrow} K_i \stackrel{r_i}{\rightarrow} R_i, ac_i)$, with $i = 1, 2$, a typed rule morphism $f : p_1 \to p_2$ is a tuple (f_L, f_K, f_R) of *TG*-typed graph monomorphisms such that the squares with the span monomorphisms l_i and r_i, for $i = 1, 2$, are pullbacks, and such that $ac_2 \Rightarrow \mathsf{Shift}(f_L, ac_1)$. *TG*-typed graph transformation rules and typed rule morphisms define the category $\mathbf{Rule}_{TG}$, which is the comma category **Rule** over *TG*.

Following [5, 22], we use forward and backward retyping functors to deal with graphs over different type graphs. A graph morphism $f : TG \to TG'$ induces a forward retyping functor $f^{>} : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$, with $f^{>}(g_1) = f \circ g_1$ and $f^{>}(k\colon g_1 \to g_2) = k$ by composition, as shown in the diagram in Figure 9(a). Similarly, such a morphism f induces a backward retyping functor $f^{<} : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$, with $f^{<}(g_1') = g_1$ and $f^{<}(k' : g_1' \to g_2') = k : g_1 \to g_2$ by pullbacks and mediating morphisms as shown in the diagram in Figure 9(b). Retyping functors also extends to application conditions and rules, so we will write things like $f^{>}(ac)$ or $f^{<}(p)$ for some application condition ac and production p. Notice, for example, that given a graph morphism $f : TG \to TG'$, the forward retyping of a production $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R, ac)$ over *TG* is a production $f^{>}_{TG}(p) = (f^{>}_{TG}(L) \stackrel{f^{>}_{TG}(l)}{\longleftarrow} f^{>}_{TG}(K) \stackrel{f^{>}_{TG}(r)}{\longrightarrow} f^{>}_{TG}(R), f^{>}_{TG}(ac))$ over *TG*′, defining an induced morphism $f^p : p \to f^{>}_{TG}(p)$ in **Rule**. Since f^p is a morphism between rules in $|\mathbf{Rule}_{TG}|$ and $|\mathbf{Rule}_{TG'}|$, it is defined in **Rule**, forgetting the typing. Notice also that $f^{>}_{TG}(ac) \cong \mathsf{Shift}(f^p_L, ac)$.

As said above, to improve readability, if $G \to TG$ is a *TG*-typed graph, we sometimes refer to it just by its typed graph G, leaving *TG* implicit. As a consequence, if $f : TG \to TG'$ is a morphism, we may refer to the *TG*′-typed graph $f^{>}(G)$, even if this may be considered an abuse of notation.

The following results will be used in the proofs in the following section.

Proposition 1. ***(From [22]) (Adjunction)*** *Forward and backward retyping functors are left and right adjoints; i.e., for each $f : TG \to TG'$ we have $f^{>} \dashv f^{<} : TG \to TG'$.*

Remark 1. Given a graph monomorphism $f : TG \to TG'$, for all $k' : G'_1 \to G'_2$ in $\mathbf{Graph_{TG'}}$, the following diagram is a pullback:

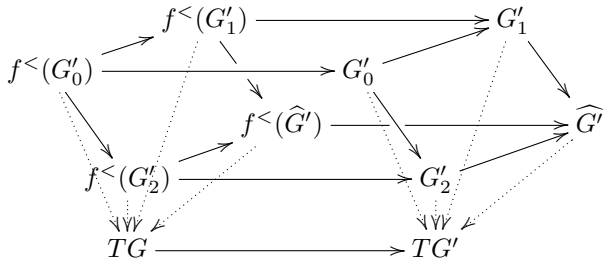
$$\begin{array}{ccc} f^{<}(G'_1) & \xrightarrow{f^{<}(k')} & f^{<}(G'_2) \\ \downarrow & pb & \downarrow \\ G'_1 & \xrightarrow[k']{} & G'_2 \end{array}$$

This is true just by pullback decomposition.

Remark 2. Given a graph monomorphism $f : TG \to TG'$, and given monomorphisms $k' : G'_0 \to G'_1$ and $h' : G'_0 \to G'_2$ in $\mathbf{Graph_{TG'}}$, if the following diagram on the left is a pushout then the diagram on the right is also a pushout:

$$\begin{array}{ccc} G'_0 & \xrightarrow{k'} & G'_1 \\ {\scriptstyle h'}\downarrow & po & \downarrow{\scriptstyle \widehat{h'}} \\ G'_2 & \xrightarrow[\widehat{k'}]{} & \widehat{G} \end{array} \qquad \begin{array}{ccc} f^{<}(G'_0) & \xrightarrow{f^{<}(k')} & f^{<}(G'_1) \\ {\scriptstyle f^{<}(h')}\downarrow & po & \downarrow{\scriptstyle f^{<}(\widehat{h'})} \\ f^{<}(G'_2) & \xrightarrow[f^{<}(\widehat{k'})]{} & f^{<}(\widehat{G}) \end{array}$$

Notice that since in an adhesive HLR category all pushouts along M-morphisms are van Kampen squares, the commutative square created by the pullbacks and induced morphisms by the backward retyping functor imply the second pushout.



Remark 3. Given a graph monomorphism $f : TG \to TG'$, and given monomorphisms $k : G_0 \to G_1$ and $h : G_0 \to G_2$ in $\mathbf{Graph_{TG}}$, if the diagram on the left is a pushout (resp., a pullback) then the diagram on the right is also a pushout (resp., a pullback):

$$\begin{array}{ccc} G_0 & \xrightarrow{k} & G_1 \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \widehat{h}} \\ G_2 & \xrightarrow[\widehat{k}]{} & \widehat{G} \end{array} \qquad \begin{array}{ccc} f^{>}(G_0) & \xrightarrow{f^{>}(k)} & f^{>}(G_1) \\ {\scriptstyle f^{>}(h)}\downarrow & & \downarrow{\scriptstyle f^{>}(\widehat{h})} \\ f^{>}(G_2) & \xrightarrow[f^{>}(\widehat{k})]{} & f^{>}(\widehat{G}) \end{array}$$

Remark 4. Given a graph monomorphism $f : TG \to TG'$, and a TG'-typed graph transformation rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$, if a matching $m : L \to C$ satisfies ac, that is, $m \models ac$, then, $f^{<}(m) \models f^{<}(ac)$.

4 GTS morphisms and preservation of behaviour

A typed graph transformation system over a type graph *TG*, is a graph transformation system where the given graph transformation rules are defined over the category of *TG*-typed graphs. Since in this paper we deal with GTSs over different type graphs, we will make explicit the given type graph. This means that, from now on, a typed GTS is a triple (TG, P, π) where *TG* is a type graph, P is a set of rule names and π is a function mapping each rule name p into a rule $(L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R, ac)$ typed over *TG*.

The set of transformation rules of each GTS specifies a behaviour in terms of the derivations obtained via such rules. A GTS morphism defines then a relation between its source and target GTSs by providing an association between their type graphs and rules.

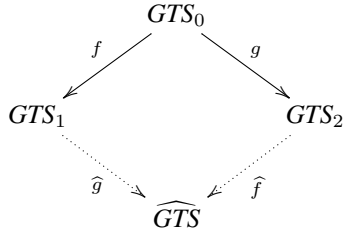
Definition 6. ***(GTS morphism)*** *Given typed graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$, with $f = (f_{TG}, f_P, f_r)$, is given by a morphism $f_{TG} : TG_0 \to TG_1$, a surjective mapping $f_P : P_1 \to P_0$ between the sets of rule names, and a family of rule morphisms $f_r = \{f^p : f^{>}_{TG}(\pi_0(f_P(p))) \to \pi_1(p)\}_{p \in P_1}$.*

Given a GTS morphism $f : GTS_0 \to GTS_1$, each rule in GTS_1 extends a rule in GTS_0. However if there are internal computation rules in GTS_1 that do not extend any rule in GTS_0, we can always consider that the empty rule is included in GTS_0, and assume that those rules extend the empty rule.

Please note that rule morphisms are defined on rules over the same type graph (see Definition 3). To deal with rules over different type graphs we retype one of the rules to make them be defined over the same type graph.

Typed GTSs and GTS morphisms define the category $\mathbf{GTS}$. The GTS amalgamation construction provides a very convenient way of composing GTSs.

Definition 7. ***(GTS Amalgamation)*** *Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1, 2$, and GTS morphisms $f : GTS_0 \to GTS_1$ and $g : GTS_0 \to GTS_2$, the amalgamated GTS $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ is the GTS $(\widehat{TG}, \widehat{P}, \widehat{\pi})$ constructed as follows. We first construct the pushout of typing graph morphisms $f_{TG} : TG_0 \to TG_1$ and $g_{TG} : TG_0 \to TG_2$, obtaining morphisms $\widehat{f}_{TG} : TG_2 \to \widehat{TG}$ and $\widehat{g}_{TG} : TG_1 \to \widehat{TG}$. The pullback of set morphisms $f_P : P_1 \to P_0$ and $g_P : P_2 \to P_0$ defines morphisms $\widehat{f}_P : \widehat{P} \to P_2$ and $\widehat{g}_P : \widehat{P} \to P_1$. Then, for each rule p in $\widehat{P}$, the rule $\widehat{\pi}(p)$ is defined as the amalgamation of rules $\widehat{f}^{>}_{TG}(\pi_2(\widehat{f}_P(p)))$ and $\widehat{g}^{>}_{TG}(\pi_1(\widehat{g}_P(p)))$ with respect to the kernel rule $\widehat{f}^{>}_{TG}(g^{>}_{TG}(\pi_0(g_P(\widehat{f}_P(p)))))$.*



Among the different types of GTS morphisms, let us now focus on those that reflect behaviour. Given a GTS morphism $f : GTS_0 \to GTS_1$, we say that it reflects behaviour if for any derivation that may happen in GTS_1 there exists a corresponding derivation in GTS_0.

Definition 8. ***(Behaviour-reflecting GTS morphism)*** *Given graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$ is behaviour-reflecting if for all graphs G, H in $|\mathbf{Graph}_{TG_1}|$, all rules p in P_1, and all matches $m : lhs(\pi_1(p)) \to G$ such that $G \overset{p,m}{\Longrightarrow} H$, then $f^<_{TG}(G) \overset{f_P(p), f^<_{TG}(m)}{\Longrightarrow} f^<_{TG}(H)$ in GTS_0.*

Morphisms between GTSs that only add to the transformation rules elements not in their source type graph are behaviour-reflecting. We call them *extension morphisms*.

Definition 9. ***(Extension GTS morphism)*** *Given graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, 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 $p \in P_1$, $\pi_0(f_P(p)) \equiv f^<_{TG}(\pi_1(p))$.*

That an extension morphism is indeed a behaviour-reflecting morphism is shown by the following lemma.

Lemma 1. *All extension GTS morphisms are behaviour-reflecting.*

Proof Sketch. Given graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, let a GTS morphism $f : GTS_0 \to GTS_1$ be an extension morphism. Then, we have to prove that for all graphs G, H in $|\mathbf{Graph}_{TG_1}|$, all rules p in P_1, and all matches $m : lhs(\pi_1(p)) \to G$, if $G \overset{p,m}{\Longrightarrow} H$ then $f^<_{TG}(G) \overset{f_P(p), f^<_{TG}(m)}{\Longrightarrow} f^<_{TG}(H)$.

Assuming transformation rules $\pi_1(p) = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1, ac_1)$ and $\pi_0(f_P(p)) = (L_0 \xleftarrow{l_0} K_0 \xrightarrow{r_0} R_0, ac_0)$, and given the derivation

$$\begin{array}{ccccccc} ac_1 & \triangleright & L_1 & \xleftarrow{l_1} & K_1 & \xrightarrow{r_1} & R_1 \\ & & {\scriptstyle m}\downarrow & po & \downarrow & po & \downarrow \\ & & G & \longleftarrow & D & \longrightarrow & H \end{array}$$

since f is an extension morphism, and therefore f_{TG} is a monomorphism, and l_1 and m are also monomorphisms, by Remark 2 and Definition 8, we have the diagram

$$\begin{array}{ccccccc} ac_0 & \triangleright & L_0 & \xleftarrow{l_1} & K_0 & \xrightarrow{r_1} & R_0 \\ \cong & & \| & & \| & & \| \\ f^<_{TG}(ac_1) & \triangleright & f^<_{TG}(L_1) & \xleftarrow{f^<_{TG}(l_1)} & f^<_{TG}(K_1) & \xrightarrow{f^<_{TG}(r_1)} & f^<_{TG}(R_1) \\ & & {\scriptstyle f^<_{TG}(m)}\downarrow & po & \downarrow & po & \downarrow \\ & & f^<_{TG}(G) & \longleftarrow & f^<_{TG}(D) & \longrightarrow & f^<_{TG}(H) \end{array}$$

Then, given the pushouts in the above diagram and Remark 4, we have the derivation $f^{<}_{TG}(G) \xRightarrow{f_P(p), f^{<}_{TG}(m)} f^{<}_{TG}(H)$. □

Notice that Definition 9 provides specific checks on individual rules. In the concrete case we presented in Section 2, the inclusion morphism between the model of an observer DSML, M_{Obs}, and its parameter sub-model M_{Par}, may be very easily checked to be an extension, by making sure that the features "added" in the rules will be removed by the backward retyping functor. In this case the check is particularly simple because of the subgraph relation between the type graphs, but for a morphism as the binding morphism between M_{Par} and the DSML of the system at hand, M_{DSML}, the check would also be relatively simple. Basically, the backward retyping of each rule in M_{DSML}, i.e., the rule resulting from removing all elements not target of the binding map, must coincide with the corresponding rule, and the application conditions must be equivalent.

Since the amalgamation of GTSs is the basic construction for combining them, it is very important to know whether the reflection of behaviour remains invariant under amalgamations.

Proposition 2. *Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1, 2$, and the amalgamation $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ of GTS morphisms $f : GTS_0 \to GTS_1$ and $g : GTS_0 \to GTS_2$, if f_{TG} is a monomorphism and g is an extension morphism, then $\widehat{g}$ is also an extension morphism.*

$$\begin{array}{ccc} GTS_0 & \xrightarrow{\quad f \quad} & GTS_1 \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \widehat{g}} \\ GTS_2 & \xrightarrow[\quad \widehat{f} \quad]{} & \widehat{GTS} \end{array}$$

Proof Sketch. Let it be $\widehat{GTS} = (\widehat{TG}, \widehat{P}, \widehat{\pi})$. We have to prove that for each $p \in \widehat{P}$, $\pi_1(\widehat{g}_P(p)) \equiv \widehat{g}^{<}_{TG}(\widehat{\pi}(p))$. By construction, rule $\widehat{\pi}(p)$ is obtained from the amalgamation of rules $\widehat{g}^{>}_{TG}(\pi_1(\widehat{g}_P(\widehat{p})))$ and $\widehat{f}^{>}_{TG}(\pi_2(\widehat{f}_P(\widehat{p})))$. More specifically, without considering application conditions by now, the amalgamation of such rules is accomplished by constructing the pushouts of the morphisms for the left-hand sides, for the kernel graphs, and for the right-hand sides.

By Remark 2, we know that if the diagram

$$\begin{array}{ccc} \widehat{f}^{>}_{TG}(g^{>}_{TG}(L_0)) \cong \widehat{g}^{>}_{TG}(f^{>}_{TG}(L_0)) & \xrightarrow{\widehat{g}^{>}_{TG}(f_L^{\widehat{g}_P(p)})} & \widehat{g}^{>}_{TG}(L_1) \\ {\scriptstyle \widehat{f}^{>}_{TG}(g_L^{\widehat{f}_P(p)})}\downarrow & & \downarrow{\scriptstyle \widehat{g}^{p}_{L}} \\ \widehat{f}^{>}_{TG}(L_2) & \xrightarrow[\quad \widehat{f}^{p}_{L} \quad]{} & \widehat{L} \end{array}$$

is a pushout, then if we apply the backward retyping functor $\widehat{g}^{<}_{TG}$ to all its components (graphs) and morphisms, the resulting diagram is also a pushout.

$$\begin{array}{ccc} \widehat{g}^{<}_{TG}(\widehat{g}^{>}_{TG}(f^{>}_{TG}(L_0))) & \xrightarrow{\widehat{g}^{<}_{TG}(\widehat{g}^{>}_{TG}(f_L^{\widehat{g}_P(p)}))} & \widehat{g}^{<}_{TG}(\widehat{g}^{>}_{TG}(L_1)) \\ \downarrow{\scriptstyle \widehat{g}^{<}_{TG}(\widehat{f}^{>}_{TG}(g_L^{\widehat{f}_P(p)}))} & & \downarrow{\scriptstyle \widehat{g}^{<}_{TG}(\widehat{g}^{p}_{L})} \\ \widehat{g}^{<}_{TG}(\widehat{f}^{>}_{TG}(L_2)) & \xrightarrow[\widehat{g}^{<}_{TG}(\widehat{f}^{p}_{L})]{} & \widehat{g}^{<}_{TG}(\widehat{L}) \end{array}$$

Because, by Proposition 1, for every $f : TG \to TG'$ and every TG-type graph G and morphism g, since f_{TG} is assumed to be a monomorphism, $f^{<}(f^{>}(G)) = G$ and $f^{<}(f^{>}(g)) = g$, we have $\widehat{g}^{<}_{TG}(\widehat{g}^{>}_{TG}(f^{>}_{TG}(L_0)) = f^{>}_{TG}(L_0)$, $\widehat{g}^{<}_{TG}(\widehat{g}^{>}_{TG}(f_L^{\widehat{g}_P(p)})) = f_L^{\widehat{g}_P(p)}$, and $\widehat{g}^{<}_{TG}(\widehat{g}^{>}_{TG}(L_1)) = L_1$. By pullback decomposition in the corresponding retyping diagram, $\widehat{g}^{<}_{TG}(\widehat{f}^{>}_{TG}(L_2)) = f^{>}_{TG}(g^{<}_{TG}(L_2))$.

Thus, we are left with this other pushout:

$$\begin{array}{ccc} f^{>}_{TG}(L_0) & \xrightarrow{f_L^{\widehat{g}_P(p)}} & L_1 \\ \downarrow{\scriptstyle f^{>}_{TG}(g^{<}_{TG}(g_L^{\widehat{f}_P(p)}))} & & \downarrow{\scriptstyle \widehat{g}^{<}_{TG}(\widehat{g}^{p}_{L})} \\ f^{>}_{TG}(g^{<}_{TG}(L_2)) & \xrightarrow[\widehat{g}^{<}_{TG}(\widehat{f}^{p}_{L})]{} & \widehat{g}^{<}_{TG}(\widehat{L}) \end{array}$$

Since g is an extension, $L_0 \cong g^{<}_{TG}(L_2)$, which, because f_{TG} is a monomorphism, implies $f^{>}_{TG}(L_0) \cong f^{>}_{TG}(g^{<}_{TG}(L_2))$. This implies that $\widehat{g}^{<}_{TG}(\widehat{L}) \cong L_1$.

Similar diagrams for kernel objects and right-hand sides lead to similar identity morphisms for them. It only remains to see that $ac(\pi_1(\widehat{g}_P(p))) \cong ac(\widehat{g}^{<}_{TG}(\widehat{\pi}(p)))$.

By the rule amalgamation construction, $\widehat{ac} = \widehat{f}^{>}_{TG}(ac_2) \wedge \widehat{g}^{>}_{TG}(ac_1)$. Since g is an extension morphism, $ac_2 \cong g^{>}_{TG}(ac_0)$. Then, $\widehat{ac} \cong \widehat{f}^{>}_{TG}(g^{>}_{TG}(ac_0)) \wedge \widehat{g}^{>}_{TG}(ac_1)$. For f, as for any other rule morphism, we have $ac_1 \Rightarrow f^{>}_{TG}(ac_0)$. By the Shift construction, for any match $m_1 : L_1 \to C_1$, $m_1 \models ac_1$ iff $\widehat{g}^{>}_{TG}(m_1) \models \widehat{g}^{>}_{TG}(ac_1)$ and, similarly, for any match $m_0 : L_0 \to C_0$, $m_0 \models ac_0$ iff $f^{>}_{TG}(m_0) \models f^{>}_{TG}(ac_0)$. Then, $ac_1 \Rightarrow f^{>}_{TG}(ac_0) \cong \widehat{g}^{>}_{TG}(ac_1) \Rightarrow \widehat{g}^{>}_{TG}(f^{>}_{TG}(ac_0)) \cong \widehat{g}^{>}_{TG}(ac_1) \Rightarrow \widehat{f}^{>}_{TG}(g^{>}_{TG}(ac_0))$. And therefore, since $\widehat{ac} = \widehat{f}^{>}(g^{>}_{TG}(ac_0)) \wedge \widehat{g}^{>}_{TG}(ac_1)$ and $\widehat{g}^{>}_{TG}(ac_1) \Rightarrow \widehat{f}^{>}_{TG}(g^{>}_{TG}(ac_0))$, we conclude $\widehat{ac} \cong \widehat{g}^{>}_{TG}(ac_1)$. □

When a DSL is extended with observers and other alien elements whose goal is to measure some property, or to verify certain invariant property, we need to guarantee that such an extension does not change the semantics of the original DSL. Specifically, we need to guarantee that the behaviour of the resulting system is exactly the same, that is, that any derivation in the source system also happens in the target one (behaviour preservation), and any derivation in the target system was also possible in the source one (behaviour reflection). The following definition of behaviour-protecting GTS morphism captures the intuition of a morphism that both reflects and preserves behaviour, that is, that establishes a bidirectional correspondence between derivations in the source and target GTSs.

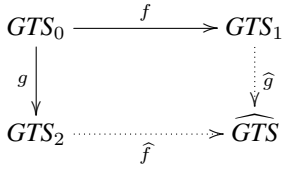
Definition 10. *(**Behaviour-protecting GTS morphism**) Given typed graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$ is behaviour-protecting if for all graphs G and H in $|\mathbf{Graph}_{TG_1}|$, all rules p in P_1, and all matches $m : lhs(\pi_1(p)) \to G$,*

$$g^<_{TG}(G) \overset{g_P(p), g^<_{TG}(m)}{\Longrightarrow} g^<_{TG}(H) \iff G \overset{p,m}{\Longrightarrow} H$$

We find in the literature definitions of behaviour-preserving morphisms as morphisms in which the rules in the source GTS are included in the set of rules of the target GTS. Although these morphisms trivially preserve behaviour, they are not useful for our purposes. Works like [25] or [22], mainly dealing with refinements of GTSs, only consider cases in which GTSs are extended by adding new transformation rules. In our case, in addition to adding new rules, we are enriching the rules themselves.

The main result in this paper is related to the protection of behaviour, and more precisely on the behaviour-related guarantees on the induced morphisms.

Theorem 1. *Given typed transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1, 2$, and the amalgamation $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ of GTS morphisms $f : GTS_0 \to GTS_1$ and $g : GTS_0 \to GTS_2$, if f is a behaviour-reflecting GTS morphism, f_{TG} is a monomorphism, and g is an extension and behaviour-protecting morphism, then $\widehat{g}$ is behaviour-protecting as well.*



Proof Sketch. Since g is an extension morphism and f_{TG} is a monomorphism, by Proposition 2, $\widehat{g}$ is also an extension morphism, and therefore, by Lemma 1, also behaviour-reflecting. We are then left with the proof of behaviour preservation.

Given a derivation $G_1 \overset{p_1, m_1}{\Longrightarrow} H_1$ in GTS_1, with $\pi_1(p_1) = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1, ac_1)$, since $f : GTS_0 \to GTS_1$ is a behaviour-reflecting morphism, there is a corresponding derivation in GTS_0. Specifically, the rule $f_P(p_1)$ can be applied on $f^<_{TG}(G_1)$ with match $f^<_{TG}(m_1)$ satisfying the application condition of production $\pi_0(f_P(p_1))$, and resulting in a graph $f^<_{TG}(H_1)$.

$$f^<_{TG}(G_1) \overset{f_P(p_1), f^<_{TG}(m_1)}{\Longrightarrow} f^<_{TG}(H_1)$$

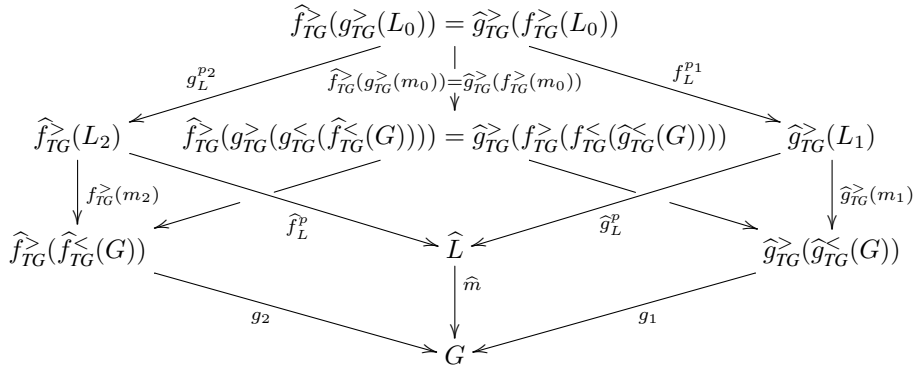
Moreover, since g is a behaviour-protecting morphism, this derivation implies a corresponding derivation in GTS_2.

By the amalgamation construction in Definition 7, the set of rules of $\widehat{GTS}$ includes, for each p in $\widehat{P}$, the amalgamation of (the forward retyping of) the rules $\pi_1(\widehat{g}_P(p)) = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1, ac_1)$ and $\pi_2(\widehat{f}_P(p)) = (L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2, ac_2)$, with kernel rule $\pi_0(f_P(\widehat{g}_P(p))) = \pi_0(g_P(\widehat{f}_P(p))) = (L_0 \xleftarrow{l_0} K_0 \xrightarrow{r_0} R_0, ac_0)$.

First, notice that for any $\widehat{TG}$ graph G, G is the pushout of the graphs $\widehat{g}^<_{TG}(G)$, $\widehat{f}^<_{TG}(G)$ and $f^<_{TG}(\widehat{g}^<_{TG}(G))$ (with the obvious morphisms). This can be proved using a van Kampen square, where in the bottom we have the pushout of the type graphs, the vertical

faces are the pullbacks defining the backward retyping functors and on top we have that pushout.

Thus, for each graph G in $\widehat{GTS}$, if a transformation rule in GTS_1 can be applied on $\widehat{g^<_{TG}}(G)$, the corresponding transformation rule should be applicable on G in $\widehat{GTS}$. The following diagram focus on the lefthand sides of the involved rules.



As we have seen above, rules $\widehat{g}_P(p)$, $\widehat{f}_P(p)$, and $\widehat{f}_P(g_P(p)) = \widehat{g}_P(f_P(p))$ are applicable on their respective graphs using the matchings depicted in the above diagram. Since, by the amalgamation construction, the top square is a pushout, and $g_1 \circ \widehat{g^>_{TG}}(m_1) \circ f_L^{p_1} = g_2 \circ \widehat{f^>_{TG}}(m_2) \circ g_L^{p_2}$, then there is a unique morphism $\widehat{m} : \widehat{L} \to G$ making $g_1 \circ \widehat{g^>_{TG}}(m_1) = \widehat{m} \circ \widehat{g}^p_L$ and $g_2 \circ \widehat{f^>_{TG}}(m_2) = \widehat{m} \circ \widehat{f}^p_L$. This $\widehat{m}$ will be used as matching morphism in the derivation we seek.

By construction, the application condition $\widehat{ac}$ of the amalgamated rule p is the conjunction of the shiftings of the application conditions of $\widehat{g}_P(p)$ and $\widehat{f}_P(p)$. Then, since

$$m_1 \models ac_1 \iff \widehat{m} \models \mathsf{Shift}(\widehat{g}^p_L, ac_1)$$

and

$$m_2 \models ac_2 \iff \widehat{m} \models \mathsf{Shift}(\widehat{f}^p_L, ac_2),$$

and therefore

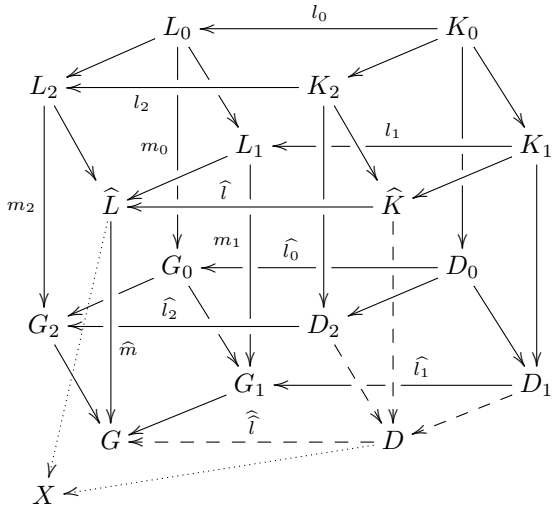
$$m_1 \models ac_1 \wedge m_2 \models ac_2 \iff \widehat{m} \models \widehat{ac}.$$

We can then conclude that rule p is applicable on graph G with match $\widehat{m}$ satisfying its application condition $\widehat{ac}$. Indeed, given the rule $\pi(p) = (\widehat{L} \xleftarrow{\widehat{l}} \widehat{K} \xrightarrow{\widehat{r}} \widehat{R}, \widehat{ac})$ we have the following derivation:

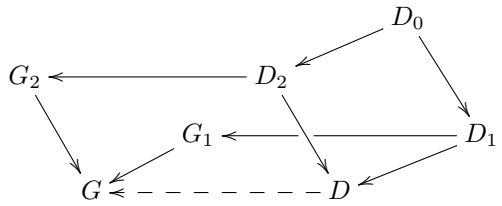
$$\begin{array}{ccccccc} \widehat{ac} & \triangleright & \widehat{L} & \xleftarrow{l_1} & \widehat{K} & \xrightarrow{r_1} & \widehat{R} \\ & & \downarrow{\scriptstyle \widehat{m}} & po & \downarrow & po & \downarrow \\ & & G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Let us finally check then that D and H are as expected. To improve readability, in the following diagrams we eliminate the retyping functors. For instance, for the rest of the theorem L_0 denotes $\widehat{f^>_{TG}}(g^>_{TG}(L_0)) = \widehat{g^>_{TG}}(f^>_{TG}(L_0))$, L_1 denotes $\widehat{g^>_{TG}}(L_1)$, etc.

First, let us focus on the pushout complement of $\widehat{l} : \widehat{K} \to \widehat{L}$ and $\widehat{m} : \widehat{L} \to G$. Given rules $\widehat{g}_P(p)$, $\widehat{f}_P(p)$, and $\widehat{f}_P(g_P(p)) = \widehat{g}_P(f_P(p))$ and rule morphisms between them as above, the following diagram shows both the construction by amalgamation of the morphism $\widehat{l} : \widehat{K} \to \widehat{L}$, and the construction of the pushout complements for morphisms l_i and m_i, for $i = 0 \ldots 2$.

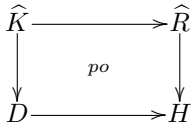


By the pushout of $D_0 \to D_1$ and $D_0 \to D_2$, and given the commuting subdiagram



there exists a unique morphism $D \to G$ making the diagram commute. This D is indeed the object of the pushout complement we were looking for. By the pushout of $K_0 \to K_1$ and $K_0 \to K_2$, there is a unique morphism from $\widehat{K}$ to D making the diagram commute. We claim that these morphisms $\widehat{K} \to D$ and $D \to G$ are the pushout complement of $\widehat{K} \to \widehat{L}$ and $\widehat{L} \to G$. Suppose that the pushout of $\widehat{K} \to \widehat{L}$ and $\widehat{K} \to D$ were $\widehat{L} \to X$ and $D \to X$ for some graph X different from G. By the pushout of $K_1 \to D_1$ and $K_1 \to L_1$ there is a unique morphism $G_1 \to X$ making the diagram commute. By the pushout of $K_2 \to D_2$ and $K_2 \to L_2$ there is a unique morphism $G_2 \to X$ making the diagram commute. By the pushout of $G_0 \to G_1$ and $G_0 \to G_2$, there is a unique morphism $G \to X$. But since $\widehat{L} \to X$ and $D \to X$ are the pushout of $\widehat{K} \to \widehat{L}$ and $\widehat{K} \to D$, there is a unique morphism $X \to G$ making the diagram commute. Therefore, we can conclude that X and G are isomorphic.

By a similar construction for the righthand sides we get the pushout



and therefore the derivation $\widehat{G} \overset{\widehat{p},\widehat{m}}{\Longrightarrow} \widehat{H}$. □

Theorem 1 provides a checkable condition for verifying the conservative nature of an extension in our example, namely the monomorphism $M_{Par} \rightarrow M_{Obs}$ being a behaviour-protecting and extension morphism, $M_{Par} \rightarrow M_{DSML}$ a behaviour-reflecting morphism, and $MM_{Par} \rightarrow MM_{DSML}$ a monomorphism.

In the concrete application domain we presented in Section 2 this result is very important. Notice that the parameter specification is a sub-specification of the observers DSL, making it particularly simple to verify that the inclusion morphism is an extension and also that it is behaviour-protecting. The check may possibly be reduced to checking that the extended system has no terminal states not in its parameter sub-specification. Application conditions should also be checked equivalent. Forbidding the specification of application conditions in rules in the observers DSL may be a practical shortcut.

The morphism binding the parameter specification to the system to be analysed can very easily be verified behaviour-reflecting. Once the morphism is checked to be a monomorphism, we just need to check that the rules after applying the backward retyping morphism exactly coincide with the rules in the source GTS. Checking the equivalence of the application conditions may require human intervention. Notice that with appropriate tools and restrictions, most of these restrictions, if not all, can be automatically verified. We may even be able to restrict editing capabilities so that only correct bindings can be specified.

Once the observers DSL are defined and checked, they can be used as many times as wished. Once they are to be used, we just need to provide the morphism binding the parameter DSL and the target system. As depicted in Figures 6 for the metamodels the binding is just a set of pairs, which may be easily supported by appropriate graphical tools. The binding must be completed by similar correspondences for each of the rules. Notice that once the binding is defined for the metamodels, most of the rule bindings can be inferred automatically.

Finally, given the appropriate morphisms, the specifications may be merged in accordance to the amalgamation construction in Definition 7. The resulting system is guaranteed to both reflect and preserve the original behaviour by Theorem 1.

5 Conclusions and future work

In this paper, we have presented formal notions of morphisms between graph transformation systems (GTSs) and a construction of amalgamations in the category of GTSs and GTS morphisms. We have shown that, given certain conditions on the morphisms involved, such amalgamations reflect and protect behaviour across the GTSs. This result is useful because it can be applied to define a notion of conservative extensions

of GTSs, which allow adding spectative behaviour (cf. [28]) without affecting the core transformation behaviour expressed in a GTS.

There are of course a number of further research steps to be taken—both in applying the formal framework to particular domains and in further development of the framework itself. In terms of application, we need to provide methods to check the preconditions of Theorem 1, and if possible automatically checkable conditions that imply these, so that behaviour protection of an extension can be checked effectively. This will enable the development of tooling to support the validation of language or transformation compositions. On the part of the formal framework, we need to study relaxations of our definitions so as to allow cases where there is a less than perfect match between the base DSML and the DSML to be woven in. Inspired by [28], we are also planning to study different categories of extensions, which do not necessarily need to be spectative (conservative), and whether syntactic characterisations exist for them, too.

Acknowledgments

We are thankful to Andrea Corradini for his very helpful comments. We would also like to thank Javier Troya and Antonio Vallecillo for fruitful discussions and previous and on-going collaborations this work relies on. This work has been partially supported by CICYT projects TIN2011-23795 and TIN2007-66523, and by the AGAUR grant to the research group ALBCOM (ref. 00516).

References

1. Boehm, P., Fonio, H.R., Habel, A.: Amalgamation of graph transformations with applications to synchronization. In: Ehrig, H., Floyd, C., Nivat, M., Thatcher, J.W. (eds.) TAPSOFT, Vol.1. Lecture Notes in Computer Science, vol. 185, pp. 267–283. Springer (1985)
2. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. In: Proc. of the 1st European Conference on Model Driven Architecture: Fondations and Applications (ECMDA-FA'05). Lecture Notes in Computer Science, vol. 3748. Springer (2005)
3. Clarke, S., Walker, R.J.: Generic aspect-oriented design with Theme/UML. In: Aspect-Oriented Software Development, pp. 425–458. Addison-Wesley (2005)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
5. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: The category of typed graph grammars and its adjunctions with categories of derivations. In: Cuny et al. [6], pp. 56–74
6. Cuny, J., Ehrig, H., Engels, G., Rozenberg, G. (eds.): Proc. 5th International Workshop on Graph Grammars and their Applications to Computer Science (GraGra 1994), Lecture Notes in Computer Science, vol. 1073. Springer (1996)
7. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Not. 35(6), 26–36 (2000)
8. Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Tech. Rep. 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA) (Apr 2006)
9. Durán, F., Zschaler, S., Troya, J.: On the reusable specification of non-functional properties in DSLs. In: Proc. 5th Int'l Conf. on Software Language Engineering (SLE 2012) (2012)

10. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) 1st Graph Grammar Workshop. Lecture Notes in Computer Science, vol. 73, pp. 1–69. Springer (1979)
11. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae 74(1), 135–166 (2006)
12. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2005)
13. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Volumne II: Applications, Languages and Tools. World Scientific (1999)
14. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement categories and systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT. Lecture Notes in Computer Science, vol. 3256, pp. 144–160. Springer (2004)
15. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 2. Module Specifications and Constraints. Springer (1990)
16. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) Graph Transformations, Second International Conference, ICGT 2004, Proceedings. Lecture Notes in Computer Science, vol. 3256, pp. 161–177. Springer (2004)
17. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Proc. of the 3th International Conference on the Unified Modeling Language: Modeling Languages and Applications (UML'00). Lecture Notes in Computer Science, vol. 1939, pp. 323–337. Springer (2000)
18. 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, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 3393, pp. 38–63. Springer (2005)
19. Engels, G., Heckel, R., Taentzer, G., Ehrig, H.: A combined reference model- and view-based approach to system specification. International Journal of Software Engineering and Knowledge Engineering 7(4), 457–477 (1997)
20. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language. In: Engels, G., Rozenberg, G. (eds.) Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT'98). Lecture Notes in Computer Science, vol. 1764, pp. 296–309. Springer (2000)
21. Große-Rhode, M., Parisi-Presicce, F., Simeoni, M.: Spatial and temporal refinement of typed graph transformation systems. In: Brim, L., Gruska, J., Zlatuska, J. (eds.) Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Proceedings. Lecture Notes in Computer Science, vol. 1450, pp. 553–561. Springer (1998)
22. Große-Rhode, M., Parisi-Presicce, F., Simeoni, M.: Formal software specification with refinements and modules of typed graph transformation systems. Journal of Computer and System Sciences 64(2), 171–218 (2002)
23. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19(2), 245–296 (2009)
24. Heckel, R., Cherchago, A.: Structural and behavioural compatibility of graphical service specifications. Journal of Logic and Algebraic Programming 70(1), 15–33 (2007)
25. Heckel, R., Corradini, A., Ehrig, H., Löwe, M.: Horizontal and vertical structuring of typed graph transformation systems. Mathematical Structures in Computer Science 6(6), 613–648 (1996)

26. Heckel, R., Engels, G., Ehrig, H., Taentzer, G.: Classification and comparison of modularity concepts for graph transformation systems. In: Ehrig et al. [13], chap. 17, pp. 669–690
27. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: A case study in transformation modularity. Software and Systems Modelling 9(3), 375–402 (Jun 2010)
28. Katz, S.: Aspect categories and classes of temporal properties. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD I, LNCS, vol. 3880, pp. 106–134. Springer (2006)
29. Klein, J., Kienzle, J.: Reusable aspect models. In: Aspect-Oriented Modeling Workshop at MODELS 2007 (2007)
30. Klein, J., Hélouët, L., Jézéquel, J.M.: Semantic-based weaving of scenarios. In: Proc. 5th Int'l Conf. Aspect-Oriented Software Development (AOSD'06). ACM (2006)
31. Kreowski, H., Kuske, S.: Graph transformation units and modules. In: Ehrig et al. [13], chap. 15, pp. 607–6380
32. Lack, S., Sobocinski, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FoSSaCS. Lecture Notes in Computer Science, vol. 2987, pp. 273–288. Springer (2004)
33. de Lara, J., Vangheluwe, H.: Automating the transformation-based analysis of visual languages. Formal Aspects of Computing 22(3-4), 297–326 (2010)
34. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented metalanguages. In: Proc. of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'05). Lecture Notes in Computer Science, vol. 3713, pp. 264–278. Springer (2005)
35. Parisi-Presicce, F.: Transformations of graph grammars. In: Cuny et al. [6], pp. 428–442
36. Rivera, J.E., Durán, F., Vallecillo, A.: Formal specification and analysis of domain specific models using Maude. Simulation 85(11-12), 778–792 (Nov 2009)
37. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In: Proc. of the 1st Intl. Conf. on Software Language Engineering (SLE'08). Lecture Notes in Computer Science, vol. 5452, pp. 54–73 (2008)
38. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Proceedings. pp. 51–55. IEEE (2009)
39. Rivera, J.E., Durán, F., Vallecillo, A.: On the behavioral semantics of real-time domain specific visual languages. In: Ölveczky, P.C. (ed.) Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6381, pp. 174–190. Springer (2010)
40. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Volume I: Foundations. World Scientific (1997)
41. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2), 25–31 (Feb 2006)
42. Schürr, A., Winter, A., Zündorf, A.: The PROGRES-approach: Language and environment. In: Ehrig et al. [13], chap. 13, pp. 487–550
43. Taentzer, G., Schürr, A.: DIEGO, another step towards a module concept for graph transformation systems. Electronic Notes on Theoretical Computer Science 2, 277–285 (1995)
44. 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.) Proc. 5th European Conf. on Model Driven Architecture – Foundations and Applications (ECMDA-FA'09). Lecture Notes in Computer Science, vol. 5562, pp. 18–33. Springer (2009)
45. Troya, J., Rivera, J.E., Vallecillo, A.: Simulating domain specific visual models by observation. In: Proc. 2010 Spring Simulation Multiconference (SpringSim '10). pp. 128:1–128:8. ACM (2010)

46. Troya, J., Vallecillo, A., Durán, F., Zschaler, S.: Model-driven performance analysis of rule-based domain specific visual models. Information and Software Technology 55(1), 88–110 (2013)
47. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A unified approach for composing UML aspect models based on graph transformation. In: Katz, S., Ossher, H. (eds.) Transactions on Aspect-Oriented Development (TAOSD VI), Special Issue on Aspects and MDE, LNCS, vol. 5560, pp. 191–237. Springer (Oct 2009)
48. Zschaler, S.: Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. Software and Systems Modelling (SoSyM) 9, 161–201 (Apr 2009)