On the Observable Behavior of Graph Transformation Systems

Tamim Ahmed Khan[1], Rodrigo Machado[2], and Reiko Heckel[1]

[1] Department of Computer Sciences, Leicester University, UK
{tak12, reiko}@mcs.le.ac.uk
[2] Univ. Federal do Rio Grande do Sul, Porto Alegre, Brazil
rma@inf.ufrgs.br

Abstract. While modelling or testing component-based or service-oriented applications we often complement the external perspective, describing the system in terms of its interactions, by the internal one specifying its implementation. To formalise this view, a graph transformation system is seen as an encapsulated component: The implementation, described by type graph and rules, is equipped with an interface (a signature of rules with parameters) defining possible observations. We use this model to study the conditions under which observations of interactions denote faithfully the internal concurrent processes in the system.
Software systems evolve to accommodate new requirements as well as to improve the quality of the existing features. Based on the characterisation of observable behaviour we establish a notion of morphism between systems that preserves observable behaviour without relating the internal structure (type graphs and rules) of the two components.

1 Introduction

A software system may evolve either as a consequence to maintenance or in order to cater for the addition, modification or deletion of functionality. Maintenance activities consume a major portion of the total cost and start as soon as a software is delivered to the production environment.

A system can be specified abstractly as a graph transformation system consisting of a type graph modelling the data structures of the application and a set of rules representing its operations or methods. The concurrent behaviour of such a system is described by equivalence classes of derivations, identifying derivations that are permutations of each other obtained by swapping independent steps. To construct such a model we have to be aware of the internal states of the system and the data dependencies between its operations. For graph transformation systems, the evolution of the corresponding concurrent behaviour has been studied extensively (starting in [1–3]) for properties such as preservation, refinement, and reflection of computations.

However, this view is not appropriate for the modelling and testing of service-oriented or component-based systems which encapsulate and hide their internal state. Instead, their behaviour is better expressed through sequences of messages or method invocations, i.e., an observational rather than concurrent semantics. In this paper we develop such a semantics for graph transformation systems based on the intuition that

a system is a component that interacts with the outside world through an abstract interface made of operation signatures only. But even in this abstract view we would like to be able to express the concurrent nature of the implementation, e.g., in order to regard two test cases as equivalent if they invoke equivalent internal derivations. Therefore, we use information about dependencies and conflicts extracted from the implementation in order to define an equivalence on these sequences matching that of the internal concurrent semantics.

Our choice of working at the level of observable behaviour is motivated by the prevalence of situations (e.g., in testing) where the system's behaviour is visible only via traces of messages exchanged. Then, in the case of evolution, we are interested in checking if this observable behaviour is preserved. In order to represent this type of evolution, we define a notion of morphism between typed attributed graph transformation systems such that labels are preserved and study the properties of such morphisms by analysing its impact on dependencies. It turns out that morphisms which preserve or reflect dependencies between labels enjoy certain preservation and / or reflection properties of traces. This allows us to add or replace freely types and rules as long as their dependencies and conflicts are preserved.

The preservation of observable behaviour is a basic requirement of refactorings [4]. With respect to regression testing, where we rerun test cases generated for the old system on the new version, an understanding of preservation or reflection of observable traces is required to decide which test cases need to be rerun, which ones are redundant or obsolete. The paper is organised as follows. After background on typed attributed graph transformation, in Section 3 we introduce observations and traces. Section 4 deals with behaviour-preserving evolution, before we review related work and conclude.

2 Typed Attributed Graph Transformation

This section provides the basic notions on typed attributed graph transformation, following the algebraic approach [5]. A graph is a tuple (V, E, src, tgt) where V is a set of nodes (or vertices), E is a set of edges and $src, tgt : E \to V$ associate, respectively, a source and target node for each edge in E. Given graphs G_1 and G_2, a graph morphism is a pair (f_V, f_E) of total functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that source and targets of edges are preserved.

An E-graph is a graph equipped with an additional set V_D of data nodes (or values) and special sets of edges E_{EA} (edge attributes) and E_{NA} (node attributes) connecting, respectively, edges in E and nodes in V to values in V_D. An attributed graph is a tuple (EG, D) where EG is an E-graph and D is an algebra with signature $\Sigma = (S, OP)$ such that $\biguplus_{s \in S} D_s = V_D$. Intuitively, an attributed graph is an E-graph where V_D is the set of all data values available for attribution. A morphisms $f : (EG, D) \to (EG', D')$ of attributed graphs is a pair of an E-graph morphism $f_{EG} : EG \to EG'$ and a compatible algebra homomorphism $f_D : D \to D'$. Fixing as type graph an attributed graph ATG over a final Σ-algebra, we define the category $\mathbf{AGraph_{ATG}}$ of ATG-typed attributed graphs [5]. Objects are pairs (G, t) of attributed graphs G with typing homomorphisms $t : G \to ATG$ and morphisms $f : G \to H$ are attributed graph morphisms compatible with the typing.

Let us denote by $X = (X_s)_{s \in S}$ a family of countable sets of variables, indexed by sorts $s \in S$, and write $x : s \in X$ for $x \in X_s$. An ATG-typed graph transformation rule (or production) over X is a span $L \xleftarrow{l} K \xrightarrow{r} R$ where l, r are monomorphisms, the algebra component of L, K, R is $T_\Sigma(X)$, the term algebra of Σ with variables in X, which the rule morphisms preserve, i.e., $l_D = r_D = id_{T_\Sigma(X)}$. That means, names of variables are preserved across the rule. The class of all rules over ATG with variables in X is denoted $Rules(ATG, X)$. A typed attributed graph transformation system (TAGTS) is a tuple (ATG, P, π) where ATG is an attributed type graph, P is a set of rule names and $\pi : P \to Rules(ATG, X)$ maps rule names to ATG-typed graph transformation rules.

In addition to describing conditions and computations on data we use attributes to identify elements in the graph. In order to formalise the idea of a key constraint, we require that every graph appropriately attributed has at most one homomorphic embedding into every reachable graph.

Definition 1 (key attributes). *An ATG-typed graph (G, t) is* functionally attributed *if for every node or edge $x \in V_G \cup E_G$ and attribute declaration $d \in (E_{NA})_{ATG} \cup (E_{EA})_{ATG}$ with $src^{ATG}(d) = t(x)$, there exists exactly one attribute $a \in (E_{NA})_G \cup (E_{EA})_G$ such that $src^G(a) = x$ and $t(a) = d$. A graph H* has keys *if it is functionally attributed, and for every functionally attributed graph G and morphisms $f, g : G \to H$, $f_A = g_A$ implies $f = g$. A transformation $G \overset{p,\, m}{\Longrightarrow} H$ has keys if both G, H do. A TAGTS* preserves keys *if for all $G \overset{p,\, m}{\Longrightarrow} H$, if G has keys, so does H. We assume in what follows that graphs occurring in transformations have keys and that they are preserved by TAGTS.*

If we consider a graph G containing just one node and its attributes, the condition states that the node is uniquely identified within H by its attributes. In database terms, the nodes attributes jointly form a key.

Example 1. (hotel service) We consider a service for managing hotel guests. A registered guest can book a room subject to availability. There are no booking charges and the bill starts to accumulate once the room is occupied. Since credit card details are already with the hotel management, the bill is automatically deducted when the guest announces their intention to check out. The guest can check out successfully only when the bill is paid. The type graph and rules modelling this service are shown in Fig. 1 using AGG [6] notation.

A functionally attributed graph is one that has one value per node or edge for each of the attributes declared for the corresponding type. If node types have enough attributes to identify their instances, the node mapping f_V of a graph morphism f is uniquely determined by its algebra part f_A. In graphs without parallel edges this extends to the mapping of edges. In our example, rooms and bills have unique numbers, and we can assume that guests are identified by a combination of name and credit card number. Thus every graph providing enough values for these attributes would have a unique embedding into any other graph.

The operational semantics of rules is defined by the double-pushout construction. Given an ATG-typed graph G and graph production $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ together with a match (i.e. a ATG-typed graph morphism) $m : L \to G$, a *direct derivation* $G \overset{p,m}{\Longrightarrow} H$ exists if

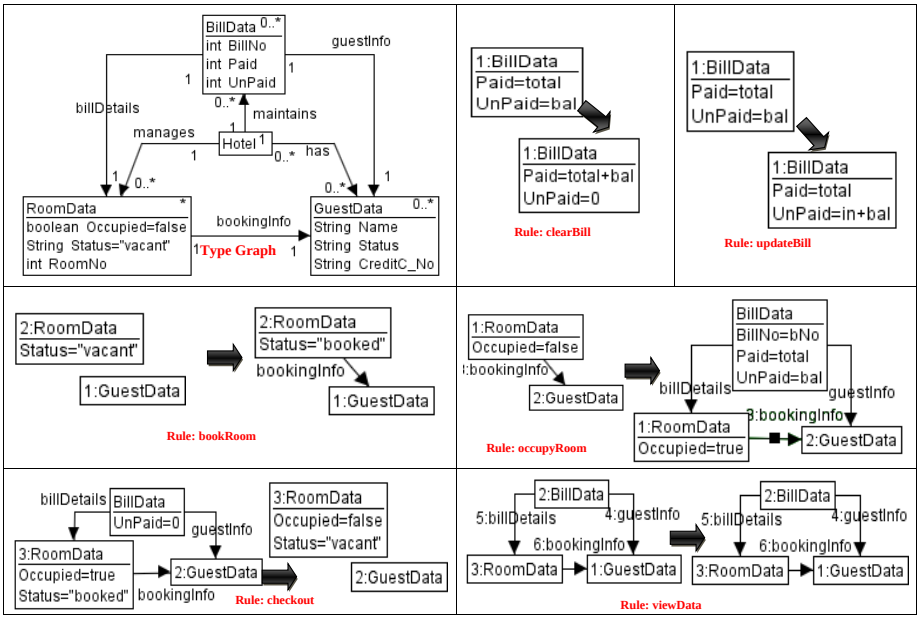


Fig. 1: Type graph and rules for the hotel service

and only if the diagram below can be constructed, where both squares are pushouts in $\mathbf{AGraph}_{\mathbf{ATG}}$ such that G, C, H share the same algebra D and the algebra components l^*_D, r^*_D of morphisms l^*, r^* are identities on D. This ensures that data elements are preserved across derivation sequences, which is relevant for their use as actual parameters. We also write $G \overset{p,d}{\Longrightarrow} H$ for $d = (m = d_L, D_K, m^* = d_R)$ if we want to refer to the entire DPO diagram. A derivation is a sequence $G_0 \overset{p_1,\, m_1}{\Longrightarrow} G_1 \overset{p_2,\, m_2}{\Longrightarrow} \ldots \overset{p_n,\, m_n}{\Longrightarrow} G_n$ of direct derivations. The class of all derivations for a given $TAGTS$ $\mathcal{G}$ is denoted $Der(\mathcal{G})$.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m=d_L}\downarrow & (1) & \downarrow{\scriptstyle d_K} & (2) & \downarrow{\scriptstyle m^*=d_R} \\ G & \xleftarrow[l^*]{} & C & \xrightarrow[r^*]{} & H \end{array}$$

The standard model of concurrency for graph transformation systems is based on equivalence classes of derivations with respect the so called shift-equivalence. The notion abstracts from the order in which independent steps are applied within a derivation, identifying all derivations that represent serialisations of the same concurrent process. Two direct derivations $G \overset{p_1,\, m_1}{\Longrightarrow} H$ and $H \overset{p_2,\, m_2}{\Longrightarrow} I$ are sequentially independent if they can be swapped. That means, the first cannot create anything used or deleted by the second, and the second cannot delete anything used by the first.

Definition 2 (independence and shift-equivalence). *Derivation* $\rho = G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} G_2$ *in the lower diagram of Fig. 3 is sequentially independent iff there exist morphisms*

$i : R_1 \to D_2$ and $j : L_2 \to D_1$ such that $r'_1 \circ j = m_2$ and $l'_2 \circ i = m'_1$. Using the local Church-Rosser theorem ([5], theorem 3.20) it is possible to construct a derivation $\rho = G_0 \overset{p_2,m'_2}{\Longrightarrow} G'_1 \overset{p_1,m'_1}{\Longrightarrow} G_2$. We use $\rho \sim_{sh} \rho'$ to denote this relation.

Shift-equivalence $\equiv_{sh} \subseteq Der(\mathcal{G}) \times Der(\mathcal{G})$ over derivations of $\mathcal{G}$ is defined as the transitive and "context" closure of $\sim_{sh}$, i.e., the least equivalence relation containing $\sim_{sh}$ and such that if $\rho \equiv_{sh} \rho'$ then $\rho_1\rho\rho_2 \equiv_{sh} \rho_1\rho'\rho_2$, with sequential composition as juxtaposition. The quotient set $Der(\mathcal{G})/\equiv_{sh}$ defines the set of concurrent derivations of the system.

Example 2 (shift equivalence). Referring to the TAGTS in Fig. 1, consider the sequences in Fig. 2(a) and Fig. 2(b). They are shift equivalent since applications of *viewData* and *clearBill* are independent and can be swapped. The graphs shown represent sample states of a hotel with only one room and one registered guest.

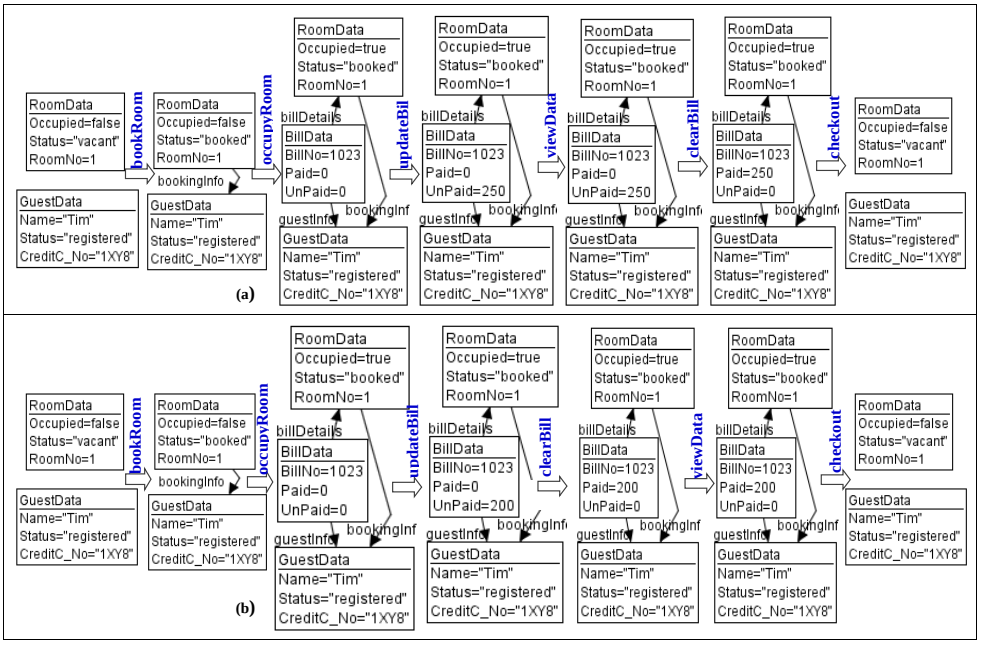


Fig. 2: Two shift equivalent derivations

3 Observational Semantics

Based on rule signatures equipping rule names with parameter declarations, in this section we introduce a notion of observation on transformation steps. Sequences of such observations will be the basis for the definition of equivalence classes (traces) as observational semantics of a graph transformation system.

Definition 3 (TAGTS with rule signatures). *A typed attributed graph transformation system with rule signatures is a tuple $\mathcal{G} = (ATG,\ P,\ X,\ \pi)$ where*

- *ATG is an attributed type graph,*
- *P is a countable set of rule names,*
- *X is an S-indexed family $(X_s)_{s\in S}$ of sets of variables,*
- *$\pi : P \longrightarrow Rules(ATG, X) \times X^*$ assigns each rule name a span sp and a list of formal parameter declarations $(x_1 : s_1, \ldots, x_n : s_n)$ were $x_i \in X_{s_i}$ for $1 \leq i \leq n$.*

For $\pi(p) = \langle sp, (x_1 : s_1, \ldots, x_n : s_n)\rangle$ we also write $p(x_1 : s_1, x_2 : s_2, \ldots, x_n : s_n) \in \mathcal{G}$. The set of all these rule signatures $Sig(\mathcal{G})$ is called the signature of $\mathcal{G}$.

As sp is attributed over $T_\Sigma(X)$, rule parameters $x_i \in X_{s_i}$ are taken from the set of variables used in attribute expressions. Hence, actual parameters will not refer to nodes or edges, but only to attribute values in the graph. Since the algebra part of attributed graphs is preserved, actual parameters have a global name space across transformations.

Example 3 (rule signatures). For the system in Fig. 1, the rule signatures shown below are based on data sorts $S = \{\text{int, boolean, string}\}$.

- bookRoom(room_no:int, guest_name:string)
- occupyRoom(room_no:int, guest_name:string, bill_no:int)
- clearBill(bill_no:int)
- checkout(room_no:int, guest_name:string, bill_no:int)
- updateBill(bill_no:int)
- viewData(room_no:int)

The signatures' purpose is to provide labels of transformations as observations. Below we define the label alphabet, then the observations associated with direct derivations.

Definition 4 (labels). *Given a rule $p : L \leftarrow K \rightarrow R$ with signature $p(x_1 : s_1, \ldots, x_n : s_n)$ and a Σ-algebra D, we denote by $p(D)$ the set of all rule labels $p(a_1, \ldots, a_n)$ where $a_i \in D_{s_i}$. The label alphabet $L_{Sig(\mathcal{G}),D}$ for a system $\mathcal{G}$ is defined as the union over all rule labels $\bigcup_{p\in P} p(D)$. If D is understood from the context, we just write $L_{Sig(\mathcal{G})}$.*

The (usually infinite) alphabet of labels $L_{Sig(\mathcal{G})}$ consists of all possible instances of rule signatures, replacing their formal parameters by values from the algebra D. Labels in $L_{Sig(\mathcal{G})}$ may be interpreted as observations of direct derivations, where the instantiation is given by the algebra component of the matches. Let $L^*_{Sig(\mathcal{G})}$ denote the Kleene closure over the label alphabet, providing the set of all finite sequences of labels.

Definition 5 (observations from derivations). *Let $G \overset{p,\,m}{\Longrightarrow} H$ be a direct derivation of a TAGTS $\mathcal{G}$ with algebra component D. The observation function $\delta : Der(\mathcal{G}) \rightarrow L^*_{Sig(\mathcal{G})}$ is defined on direct derivations by $\delta(G \overset{p,\,m}{\Longrightarrow} H) = \langle p(a_1, a_2, \ldots, a_n)\rangle$ if p's signature is $p(x_1 : s_1, \ldots, x_n : s_n)$ with $a_i = m(x_i)$. Here $\langle p(a_1, a_2, \ldots, a_n)\rangle$ denotes the one-element sequence. The observation function freely extends to derivations of arbitrary lengths, yielding sequences of labels.*

This definition describes the observational semantics of TAGTS via sequences of labels produced by its derivations. We have to ensure that labels carry enough information to determine matches and co-matches of transformations up to additional context. This is expressed as a property of the observation map δ.

Definition 6 (faithfulness). *For a DPO diagram $d = (d_L, d_K, d_R)$, its* attribute-preserving image factorisation $i \circ e$ *is given by triples of morphisms $e = (e_L : L \to G', e_K : K \to D', e_R : R \to H')$ and $i = (i_G : G' \to G, i_D : D' \to D, i_H : H' \to H)$ such that, G' is the smallest subgraph of G with functional attributes containing $d_L(L)$, with $i_G : G' \to G$ the corresponding inclusion and e_L the composition of the canonical epimorphism from L to $d_L(L)$ with the inclusion of $d_L(L)$ into G'. Morphisms e_K, i_D and e_R, i_H are defined analogously.*

The observation function δ is faithful if for all direct derivations $\rho_1 = G_1 \overset{p,d_1}{\Longrightarrow} H_1$ and $\rho_2 = G_2 \overset{p,d_2}{\Longrightarrow} H_2$, $\delta(\rho_1) = \delta(\rho_2)$ implies that the image factorisations of d_1 and d_2 are isomorphic, that is, there are isomorphisms $k_G : G'_1 \to G'_2$, $k_D : D'_1 \to D'_2$, $k_H : H'_1 \to H'_2$ such that all resulting diagrams commute. In the following we will assume observation function δ to be faithful.

Lemma 1 (attribute-preserving image factorisation). *The triples $e = (e_L : L \to G', e_K : K \to D', e_R : R \to H')$ and $i = (i_G : G' \to G, i_D : D' \to D, i_H : H' \to H)$ form double pushout diagrams.*

Proof. Follows by pushout decomposition properties and preservation of epis by pushouts (both true in all categories) and preservation of monos by pushouts (true in graph-like, adhesive categories such as ours).

The faithfulness condition is satisfied, for example, if each rule signature lists all the elements of its left-and right-hand side as parameters, thus specifying completely the embedding of the rule into graphs G and H. In most practical cases, however, parameters will only need to identify some anchor elements, which will then determine the other elements in the match and co-match. For example, a Bill will always point to a unique client, so by identifying the bill we implicitly know the client as well.

To derive an observational *concurrent* semantics, the following definition lifts weak dependency and conflict relations to the level of labels. The relations are essentially those of asymmetric event structures [7]. The asymmetry arises from the interplay of deletion and preservation, which is specific to rewriting approaches with explicit read access to resources, such as graph transformation or contextual Petri nets.

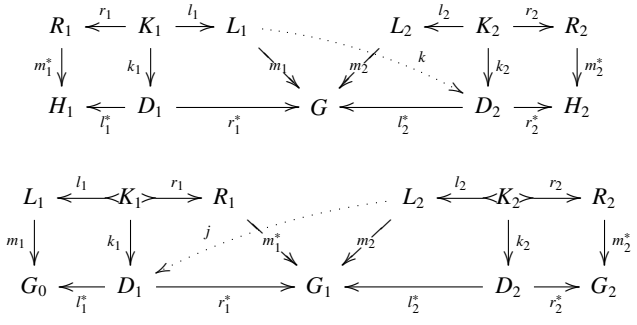


Fig. 3: Asymmetric conflicts and dependencies

Definition 7 (asymmetric dependencies and conflicts). *Two labels l_1 and l_2* are in (asymmetric) conflict*, written $l_1 \nearrow l_2$, iff there exist transformations $\rho_1 = (G \overset{p_1,m_1}{\Longrightarrow} H_1)$ and $\rho_2 = (G \overset{p_2,m_2}{\Longrightarrow} H_2)$ such that $l_i = \delta(\rho_i)$ and ρ_2* disables *ρ_1, i.e., in the upper diagram in Fig. 3 there exist no $k : L_1 \to D_2$ such that $m_1 = l_2^* \circ k$. Two labels l_1 and l_2* are in (asymmetric) dependency*, $l_1 \prec l_2$, iff there exist transformations $\rho_1 = (G_0 \overset{p_1,m_1}{\Longrightarrow} G_1)$ and $\rho_2 = (G_1 \overset{p_2,m_2}{\Longrightarrow} G_2)$ such that $l_i = \delta(\rho_i)$ and ρ_2* requires *ρ_1, i.e., in the lower diagram in Fig. 3 there exist no $j : L_2 \to D_1$ such that $m_2 = r_1^* \circ j$. Labels l_1 and l_2 are independent, $l_1 \mid l_2$, iff they are unrelated by $\nearrow$ and $\prec$.*

Due to faithfulness, dependencies and conflicts of transformations are captured precisely by the corresponding relations on labels. That means, we can replace a statement such as *$l_1 \nearrow l_2$, iff there exist transformations $\rho_1, \rho_2 \ldots$* (as in Def. 7) by one of the form *$l_1 \nearrow l_2$, iff for all transformations $\rho_1, \rho_2 \ldots$* as below.

Lemma 2. *(invariance of dependency) For suitable transformations ρ_1 and ρ_2, $l_1 \prec l_2$ ($l_1 \nearrow l_2$) iff for all ρ_1 and ρ_2 with $\delta(\rho_i) = l_i$, ρ_2 requires ρ_1 (ρ_2 disables ρ_1).*

Proof. Recall from Def. 1 that all our graphs have keys preserved by transformations. We show for all ρ_1 and ρ_2, σ_1 and σ_2 that, if $\delta(\rho_i) = l_i = \delta(\sigma_i)$ for $i = 1, 2$, then ρ_2 requires ρ_1 if and only if σ_2 requires σ_1. By definition, $l_1 \prec l_2$ iff one such dependent pair exists. The proof for ρ_2 disables ρ_1 is analogous. Consider transformations $\rho_1\rho_2 = G_0 \overset{p_1,d_1}{\Longrightarrow} G_1 \overset{p_2,d_2}{\Longrightarrow} G_2$ and $\sigma_1\sigma_2 = H_0 \overset{p_1,e_1}{\Longrightarrow} H_1 \overset{p_2,e_2}{\Longrightarrow} H_2$ in Fig. 4, where DPO diagrams d_1, d_2 and e_1, e_2 are decomposed by attribute-preserving image factorisation. By Lemma 1 these yield transformations $\rho_1' = G_0' \overset{p_1,d_1'}{\Longrightarrow} G_1', \rho_2^* = G_1^* \overset{p_2,d_2^*}{\Longrightarrow} G_2^*$ and $\sigma_1' = H_0' \overset{p_1,e_1'}{\Longrightarrow} H_1', \sigma_2^* = H_1^* \overset{p_2,e_2^*}{\Longrightarrow} H_2^*$. By faithfulness there exist isomorphisms i_X for $X \in \{G_0', D_1', G_1', G_1^*, D_2^*, G_2^*\}$ such that all resulting diagrams commute. These isomorphisms are unique because, e.g., $G_0' \to G_0$ and $H_0' \to H_0$ are inclusion and thus identities on the algebra part, which means that in order to commute with these inclusions $i_{G_0'} : G_0' \to H_0'$ has to be an identity on the data algebra, too. Since all graphs have functional attributes, the graphical part of the morphism is determined by the algebra part, i.e., there is only one such morphism between G_0' and H_0'.

Forming $G_1' \leftarrow G_1^0 \to G_1^*$ as pullback of $G_1' \to G_1 \leftarrow G_1^*$ and then $G_1' \to \bar{G}_1 \leftarrow G_1^*$ as pushout of $G_1' \leftarrow G_1^0 \to G_1^*$, we obtain $\bar{G}_1 \to G_1$ by the pushout property of $\bar{G}_1$. Similarly, $\bar{H}_1$ is constructed on the right-hand side with induced morphism $\bar{H}_1 \to H_1$. By the pushout property of $\bar{G}_1$, if $G_1^0 \to G_1' \overset{i_{G_1'}}{\longrightarrow} H_1' \to \bar{H}_1 = G_1^0 \to G_1^* \overset{i_{G_1^*}}{\longrightarrow} H_1^* \to \bar{H}_1$, it follows that there is a unique $i_{\bar{G}_1} : \bar{G}_1 \to \bar{H}_1$ commuting the resulting diagrams. The equation holds because all graphs involved use the same data algebra and all morphisms preserve it, so there is only one morphism from G_1^0 to $\bar{H}_1$ commuting the diagram. From $i_{G_1'}, i_{G_1^*}$ isos it follows that $i_{\bar{G}_1}$ is an isomorphism, too.

We proceed to decompose the DPOs in the four corners of the diagrams, obtaining, e.g., $\bar{D}_1$ and $\bar{G}_0$ in the top left by decomposition of the DPO between G_1', D_1', G_0' and G_1, D_1, G_0 along $G_1' \to \bar{G}_1 \to G_1$. This results in isomorphic transformations $\bar{G}_0 \overset{p_1,\bar{d}_1}{\Longrightarrow} \bar{G}_1 \overset{p_2,\bar{d}_2}{\Longrightarrow} \bar{G}_2$ and $\bar{H}_0 \overset{p_1,\bar{e}_1}{\Longrightarrow} \bar{H}_1 \overset{p_2,\bar{e}_2}{\Longrightarrow} \bar{H}_2$.

Now, ρ_2 requires ρ_1 iff σ_2 requires σ_1 because given $j : L_2 \to D_1$ such that $L_2 \xrightarrow{j} D_1 \to G_1 = L_2 \to G_1$, we construct $k : L_2 \to E_1$ and vice versa using the fact that, as a pushout with injective morphisms, $\bar{D}_1, D_1, \bar{G}_1, G_1$ is also a pullback. Using the universal property, j factorises as $L_2 \xrightarrow{j'} \bar{D}_1 \xrightarrow{\bar{j}}$. We define k as $L_2 \xrightarrow{j'} \bar{D}_1 \xrightarrow{i_{\bar{D}_1}} \bar{E}_1 \to E_1$ and commutativity $L_2 \xrightarrow{k} E_1 \to H_1 = L_2 \to H_1$ follows by diagram chasing.

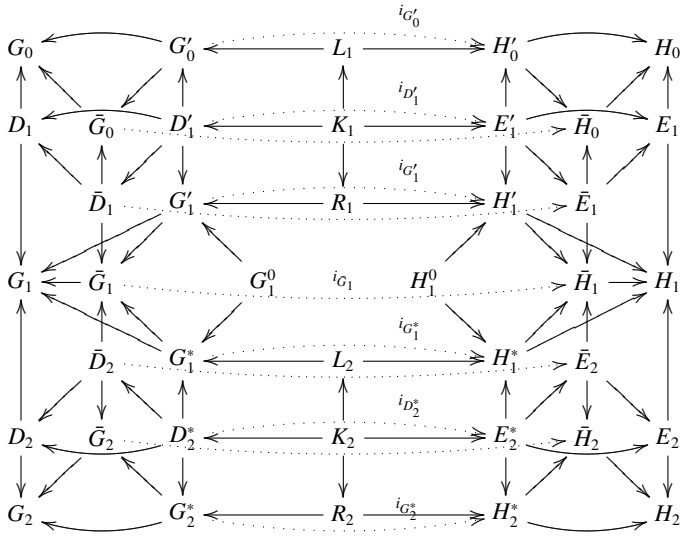


Fig. 4: Invariance of dependencies between transformations with identical labels

Asymmetric conflict $l_1 \nearrow l_2$ ensures that, if l_1, l_2 occur in the same sequence, then l_1 must precede l_2. Asymmetric dependency means that if l_2 is in a sequence and $l_1 \prec l_2$, then l_1 occurs in the sequence before l_2. These considerations can be used to define the legal traces over an alphabet equipped with these two relations.

Definition 8 (traces of a TAGTS). *We define $L^\circ_{Sig(\mathcal{G})} \subseteq L^*_{Sig(\mathcal{G})}$ as the set of all sequences $s \in L^*_{Sig(\mathcal{G})}$ respecting dependencies and conflicts, i.e., for $s = l_1 \dots l_n$ and $i,\ j,\ k \in \{1...n\}$*

- *if $i \neq j$ and $l_i \nearrow l_j$ then $i < j$*
- *if there exists $l \in L_{Sig(\mathcal{G})}$ with $l \prec l_i$ then $l = l_k$ for some $k < i$.*

The relation $\triangleq \subseteq L^\circ_{Sig(\mathcal{G})} \times L^\circ_{Sig(\mathcal{G})}$ is the least equivalence relation such that $l_i \mid l_j$ implies $s_x\, l_i\, l_j\, s_y \triangleq s_x\, l_j\, l_i\, s_y$ for all $l_i,\ l_j \in L_{Sig(\mathcal{G})}$ and $s_x, s_y \in L^\circ_{Sig(\mathcal{G})}$. The set $Traces(\mathcal{G})$ is the quotient of $L^\circ_{Sig(\mathcal{G})}$ under this equivalence.

Example 4 (traces of $\mathcal{G}$). Using the rule signatures in Example 3 we obtain labels by instantiating formal parameters by possible data values. As the set of all labels is usually infinite due to infinite data types, in this example we limit ourselves to a small subset sufficient to label the transformations in Fig. 2. For example, *clearBill*(*bill_no* : *int*) is instantiated by *clearBill*(1023), replacing the variable *bill_no* : *int* by the value 1023.

Let us analyse more closely the weak conflicts and dependencies in the derivations of Fig. 2. We have represented conflicts and dependencies between these labels in Table 1. For example we find a dependency

$$bookRoom(1,\ "Tim") \prec occupyRoom(1,\ "Tim",\ 1023).$$

Notice the relation between the parameters for room number and client name, which determines the overlap of the transformations denoted by these labels. Similarly, there exists a conflict

$$updateBill(1023) \nearrow checkout(1, "Tim", 1023),$$

i.e., *updateBill* changes the unpaid amount in the *BillData* object while *checkout* deletes the object. If we only consider sequences that respect

First/Second (↓) / (→)	*bookRoom* (1,"*Tim*")	*occupyRoom* (1,"*Tim*", 1023)	*clearBill* (1023)	*checkout* (1,"*Tim*", 1023)	*updateBill* (1023)	*viewData* (1)
bookRoom(1,"*Tim*")	$\nearrow$	$\prec$		$\prec$		$\prec$
occupyRoom(1,"*Tim*", 1023)		$\nearrow$	$\prec$	$\prec$	$\prec$	$\prec$
clearBill(1023)				$\prec$	|	|
checkout(1,"*Tim*", 1023)	$\prec$		$\nearrow$	$\nearrow$	$\nearrow$	$\nearrow$
updateBill(1023)			|	$\nearrow \prec$	|	|
viewData(1)			|	$\prec$	|	|

Table 1: Conflicts $\nearrow$ and dependencies $\prec$ between labels

conflicts and dependencies, we arrive at the filtered set $L^{\circ}_{Sig(\mathcal{G})} \subseteq L^{*}_{Sig(\mathcal{G})}$. We partition this set into traces by building equivalence classes. Consider sample sequence $s = bookRoom(1,"Tim"); occupyRoom(1,"Tim", 1023); updateBill(1023); viewData(1); clearBill(1023); checkout(1,"Tim", 1023)$. Labels *viewData*(1) and *clear Bill*(1023) are unrelated, so they can occur in any order. Therefore, the sequence obtained from the above by swapping these two steps is $\triangleq$-equivalent to s and therefore falls in the same trace.

As shown in the example we are able to lift information about dependencies and conflicts to the level of labels and use this to derive concurrent traces as an observational semantics for TAGTS with rule signatures. As a consequence of faithfulness, the observation function δ preserves information about dependencies between steps. It is therefore possible to relate equivalence classes in $Der(\mathcal{G})/\equiv_{sh}$ to traces in $Traces(\mathcal{G})$.

Proposition 1 (faithfulness). *For any two derivations sequences $s_1,\ s_2 \in \mathcal{D}er(\mathcal{G})$, $s_1 \equiv_{sh} s_2$ iff $\delta(s_1) \triangleq \delta(s_2)$. That means, δ extends to $\delta : \mathcal{D}er(\mathcal{G})/_{\equiv_{sh}} \to Traces(\mathcal{G})$, establishing a one-to-one correspondence given by $\delta([s]_{\equiv_{sh}}) = [\delta(s)]_{\triangleq}$.*

Proof. We show that derivation steps $\rho_1\rho_2$ are sequentially independent iff $\delta(\rho_1)|\delta(\rho_2)$. First, steps $\rho_1\rho_2$ are sequentially independent iff(1) ρ_2 does not require ρ_1 and the transformation ρ_2', obtained by anticipating ρ_2, does not disables ρ_1. By Def. 7 and Lemma 2 this is equivalent to saying that $\delta(\rho_1) \not\prec \delta(\rho_2)$ and $\delta(\rho_1) \not\nearrow \delta(\rho_2)$.

To see equivalence (1), consider that sequential (in)dependence is expressed in terms of existence or otherwise of morphisms $i : L_1 \to D_2$ such that $m_1 = r_2^* \circ i$ $(\exists i)$ and $j : L_2 \to D_1$ such that $m_2 = r_1^* \circ j$ $(\exists j)$. If $\exists i \wedge \exists j$, we have independence. In the case of a sequential dependency, there are three cases. If $\exists i \wedge \nexists j$ or $\nexists i \wedge \nexists j$, by Def. 7

$l_1 \prec l_2$. Now assume $\nexists i \wedge \exists j$, i.e., the steps are in deliver-delete dependency, where the 1st produces or uses an item which is deleted by the 2nd ($\nexists i$), but the 2nd does not use anything created by the 1st ($\exists j$). Put together, this means we are faced with a use-delete dependency. Defining match $m'_2 = l_1^* \circ j : L_2 \to G$ leads to a transformation $G \overset{p_2,m'_2}{\Longrightarrow} H'_2$ where the use-delete dependency on $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$ translates into a use-delete dependency between $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m'_2}{\Longrightarrow} H'_2$. Since both $H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$ and $G \overset{p_2,m'_2}{\Longrightarrow} H'_2$ generate the same label l_2, this implies that $l_1 \nearrow l_2$. The proof for $\prec$ is analogous.

4 Preserving Observable Behaviour

The definition of abstract conflicts and dependencies not only allows us to filter sequences and partition them into traces, but can also be useful in the analysis of observable behaviour under evolution of the system. Evolution scenarios such as internal refactorings or addition of new features should preserve not only the interface, but the observable behaviour in the sense that the observed sequences of the old system can be mapped to those of the new one. Moreover, this mapping should extend to traces, i.e., respect the equivalence relation on sequences. We start by defining a mapping between the signatures of system.

Definition 9 (label-preserving morphism). *Let $\mathcal{G} = (ATG, P, X, \pi)$ and $\mathcal{G}' = (ATG', P', X, \pi')$ be TAGTS over the same data signature $\Sigma = (S, OP)$. A label-preserving morphism from $f : \mathcal{G} \to \mathcal{G}'$ is a mapping of rule names $f : P \to P'$ preserving rule signatures, i.e., for all $p \in P$, $\pi(p) = \langle sp, (x_1 : s_1, \ldots, x_n : s_n)\rangle$ implies $\pi'(f(p)) = \langle sp', (x_1 : s_1, \ldots, x_n : s_n)\rangle$.*

Notice that we do not impose any condition on the relation of type graphs or rule spans. Not surprisingly, therefore this mapping will not preserve or reflect behaviour, but it will allow us to translate labels.

Proposition 2 (translating labels). *Given a label-preserving morphism $f : \mathcal{G} \to \mathcal{G}'$ and an algebra homomorphism $f_D : D \longrightarrow D'$, a mapping of labels $f_L : L_{Sig(\mathcal{G}),D} \to L'_{Sig(\mathcal{G}),D'}$ is given by $p(a_1, \ldots, a_n) \mapsto f_P(p)(f_D(a_1), \ldots, f_D(a_n))$.*

Proof. By Def. 9, $f_P(p)(f_D(a_1), \ldots, f_D(a_n))$ is a label in $L'_{Sig(\mathcal{G}),D'}$ because f_D preserves sorts.

In order to study the impact of evolution on the observable behaviour, we relate the traces of the new and the old system based on the free extension f_L^* of f_L to sequences.

Definition 10 (induced relation between traces). *We define $\leftrightsquigarrow \subseteq Traces(\mathcal{G}) \times Traces(\mathcal{G}')$ by $t \leftrightsquigarrow t'$ if and only if there exist sequences of observations $s \in L^\circ_{Sig(\mathcal{G})}$ and $s' \in L^\circ_{Sig(\mathcal{G}')}$ such that $t = [s]_{\doteq}$, $t' = [s']_{\doteq'}$ and $f_L^*(s) = s'$.*

The relation $\leftrightsquigarrow$ records whatever traces are preserved, but it could be partial or empty, merge or split traces. How can we guarantee preservation of behaviour at the interface level? Existence and equivalence of sequences are based on dependencies and conflicts

between labels. If we introduce new conflicts or dependencies, we will potentially make illegal existing sequences or differentiate between sequences that previously have been equivalent. In order to preserve observable behaviour in the sense of making $\leftrightsquigarrow$ a total function, dependencies and conflicts have to be reflected. We first define the various preservation and reflection properties that we will require and then state this observation formally.

Definition 11 (reflection/preservation properties). *Given a label-preserving morphism $f : \mathcal{G} \to \mathcal{G}'$ and its induced mapping of labels $f_L : L_{Sig(\mathcal{G})} \to L'_{Sig(\mathcal{G}')}$, f*

- *preserves conflicts iff for all $l_1, l_2 \in L_{Sig(\mathcal{G})}$, $l_1 \nearrow l_2$ implies $f_L(l_1) \nearrow f_L(l_2)$*
- *reflects conflicts iff for all $l_1, l_2 \in L_{Sig(\mathcal{G})}$, $f_L(l_1) \nearrow f_L(l_2)$ implies $l_1 \nearrow l_2$.*
- *preserves dependencies iff for all $l_1, l_2 \in L_{Sig(\mathcal{G})}$, $l_1 \prec l_2$ implies $f_L(l_1) \prec f_L(l_2)$*
- *reflects dependencies iff for all $l_1, l_2 \in L_{Sig(\mathcal{G})}$, $f_L(l_1) \prec f_L(l_2)$ implies $l_1 \prec l_2$.*

Theorem 1 (preservation of observable traces). *If f_L reflects dependencies and conflicts, then relation $\leftrightsquigarrow$ is a total function $\leftrightsquigarrow$: $Traces(\mathcal{G}) \to Traces(\mathcal{G}')$. If, moreover, f_L preserves dependencies and $f : P \to P'$ is injective, then function $\leftrightsquigarrow$ is injective.*

Proof. We have to prove that for traces $t_1, t_2 \in Traces(\mathcal{G})$ and $t'_1, t'_2 \in Traces(\mathcal{G}')$

- $\leftrightsquigarrow$ is a function, i.e., $t_1 \leftrightsquigarrow t'_1$ and $t_1 \leftrightsquigarrow t'_2$ implies $t'_1 = t'_2$
- $\leftrightsquigarrow$ is total, i.e., for all t_1 there exists t'_1 s.t. $t_1 \leftrightsquigarrow t'_1$
- $\leftrightsquigarrow$ is injective, i.e., $t_1 \leftrightsquigarrow t'_1$ and $t_2 \leftrightsquigarrow t'_1$ implies $t_1 = t_2$

Let's consider the first item above.

1. If f_L reflects dependencies and conflicts then, for all sequences $s \in L^\circ_{Sig(\mathcal{G})}$ and labels l_1, l_2 in s, $l_1 | l_2$ implies $f_L(l_1) | f_L(l_2)$, i.e., f_L preserves independence.
2. If f_L reflects independence, $s_1 \doteq s_2$ implies that $f^*_L(s_1) \doteq f^*_L(s_2)$, i.e., f^*_L preserves $\doteq$. To see that this is true, assume two strings $s_1, s_2 \in L^\circ_{Sig(\mathcal{G})}$. If they are $\doteq$-equivalent, one is a permutation of the other by exchanging independent labels only. Since f_L preserves independence of labels, the chain of permutations leading from s_1 to s_2 can be simulated to obtain a corresponding chain linking $f^*_L(s_1)$ and $f^*_L(s_2)$, so that $f^*_L(s_1) \doteq f^*_L(s_2)$.
3. From the definition of $\leftrightsquigarrow$ it follows that, if $t_1 \leftrightsquigarrow t_2$ and $t_1 \leftrightsquigarrow t_3$, then there exists $s_1, s_2 \in t_1$ such that $f^*_L(s_1) = s'_1 \in t_2$ and $f^*_L(s_2) = s'_2 \in t_3$. By definition of $\doteq$, $s_1 \doteq s_2$. By 2 above, $s'_1 \doteq s'_2$ and, therefore, $t_2 = t_3$.

To see that relation $\leftrightsquigarrow$ is total we have to make sure that f^*_L is a total function, that is, for each sequence $s \in L^\circ_{Sig(\mathcal{G})}$ there exists $s' \in L^\circ_{Sig(\mathcal{G}')}$ such that $f^*_L(s) = s'$. It is clear that f^*_L is total as a function $L^*_{Sig(\mathcal{G})} \to L^*_{Sig(\mathcal{G}')}$. For a sequence $s \in L^*_{Sig(\mathcal{G})}$ to be in $L^\circ_{Sig(\mathcal{G})}$ it has to respect the dependencies and conflicts of $\mathcal{G}$. Since any dependencies and conflicts in $\mathcal{G}'$ are reflected in $\mathcal{G}$, $s \in L^\circ_{Sig(\mathcal{G})}$ implies $f^*_L(s) \in L^\circ_{Sig(\mathcal{G}')}$. For injectivity we have

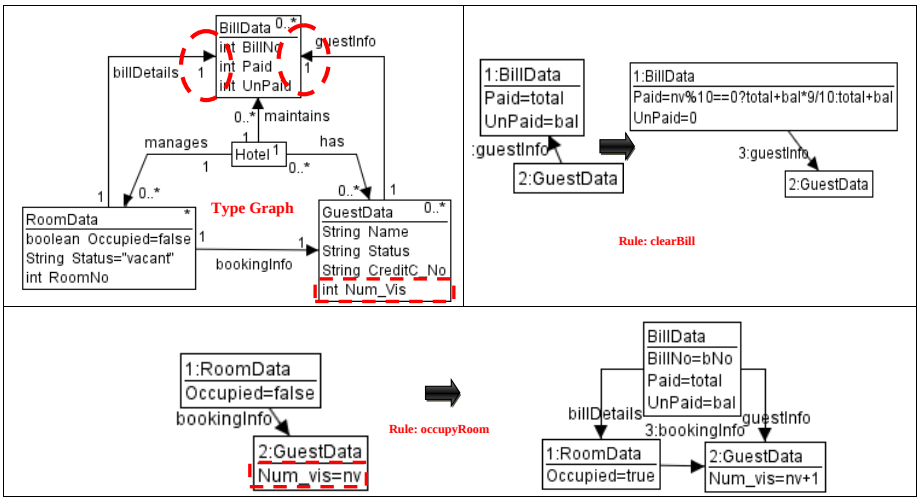
1. If f_L preserves dependencies then $f^*_L(s_1) \doteq' f^*_L(s_2)$ implies $s_1 \doteq s_2$, i.e., f^*_L reflects $\doteq'$. To see this, we show that $s_1 \neq s_2$ implies $f^*_L(s_1) \neq f^*_L(s_2)$.
   (a) If s_1 and s_2 have different lengths, $f^*_L(s_1)$ and $f^*_L(s_2)$ will because f^*_L preserves the lengths of sequences.

(b) If s_1 and s_2 contain different labels, they are still different under f_L and so f_L^* because f is injective on rule names. *****

(c) If s_1 and s_2 have the same lengths and contain the same labels, $s_1 \neq s_2$ only if s_2 is a permutation of s_1 reversing two dependent symbols. Since f_L preserves dependencies, the same holds for $f_L^*(s_1)$ and $f_L^*(s_2)$.

2. If f_L^* reflects $\doteq$ then $\leftrightsquigarrow$ is injective, because if $t_1 \leftrightsquigarrow t_3$ and $t_2 \leftrightsquigarrow t_3$, there exists $s_1 \in t_1$ s.t. $f_L^*(s_1) = s_1' \in t_3$ and $s_2 \in t_2$ s.t. $f_L^*(s_2) = s_2' \in t_3$. By definition of $\doteq$, $s_1' \doteq s_2'$. By 2 above, $s_1 \doteq s_2$ and, therefore, $t_1 = t_2$.

Example 5 (preservation of observable behaviour). Considering again the example discussed in Fig. 1, we present two changes to the model. The first is to reverse the direction of the associations highlighted by the small circles on the type graph shown in the figure below. The second is to record the number of visits of each customer to the hotel, to introduce a promotional 10% discount on all payments of every 10^{th} visit. This is achieved by an update to the *occupyRoom* rule to count visits, and a change to *clearBill* to distinguish the cases where the bill is paid with or without discount. Note the use of a conditional expression for calculating the value of the *paid* attribute.



Obviously, there is no total morphism between the new type graph and the original one (in neither direction) because of the reversal of edges, so this evolution scenario would not fall into any of the established notions of morphism between graph transformation systems designed to preserve or reflect derivations. None of the existing dependencies/conflicts are dropped, nor are new ones added. Hence all dependencies/conflicts are reflected, in which case Theorem 1 implies that traces are preserved.

Let us conclude by discussing the relation with regression testing, i.e., the execution of a set of test cases on each new version of the system to avoid a deterioration of existing functionality. To save time and effort it is advisable to reuse existing test cases and concentrate tests on features which have changed [8]. Given a test case, it can be

classified as required, reusable or obsolete [9]. In our example above, the model-level evolution preserves traces. Therefore, all test cases are reusable (none are obsolete), and the tests should produce the same observable behaviour, provided individual rules are implemented correctly.

5 Related Work

We discuss a couple of approaches to behaviour-preserving evolution as well as a related line of work characterising derivations through analysis of dependencies. As mentioned in the Introduction, semantics-preserving morphisms between graph transformation systems have been studied for a number of years, starting with [1, 2]. The common idea is to relate type graphs and rules of the system by means of morphisms such that both instance graphs and transformations can be translated from one system to the other. Approaches vary for the direction and level of generality of these relations (see [10] for an analysis of possible combinations), but they all constrain the relation between (what we consider) the implementations of the systems. Instead, we do not consider directly the relation between type graphs and rules, but only between dependencies and conflicts over labels of the two versions of the system.

Evolution at the level of observable behaviour has been studied in [11] based on rewriting with borrowed context [12]. In this approach, labels are derived as extensions of given graphs required to apply a rule. Thus they have a different structure than our observations, which are based on rule names and formal parameters, but this may be a matter of representation. A more detailed analysis would be in order to see if we can model communication with a component in such a way that borrowed context provides an observational semantics fit for testing. More significantly, [11] addresses the refactoring of graphs representing diagrams whose operational semantics is specified by a fixed set of rules. That means, it is only this graph that is being evolved, not the rules themselves. Technically, unlike [13], our approach is based on trace theory [14] rather than labelled transition systems and bisimulation. We have analysed conflicts and dependencies to provide a mechanism by which we can filter out invalid traces at the level of observations. Work in [15] has used conflicts and dependencies to define conditions for the (non-)existence of actual derivations. Our approach is different for its focus on observable behaviour, but shares some of the intuitions and techniques.

6 Conclusion

We have taken a view of typed attributed graph transformation systems as components encapsulating their type graphs and rules and communicating with the outside world only through interfaces defined by rule signatures. Based on such a model, we have investigated a concurrent observational semantics based on traces defined as equivalence classes of sequences of observations. Traces can be extracted from the standard concurrent semantics of shift-equivalence classes, but they can also be characterised by conflict and dependency relations defined at the level of observations. This model is used to study behaviour-preserving evolution of systems in terms of the preservation and reflection of dependencies and conflicts. It turns out that adding dependencies and

conflicts results in a refinement, reducing the set of possible sequences of observations as well as differentiating their equivalence. We are using these results to develop and substantiate formally a method for regression testing, where traces represent equivalence classes of test cases that have to be generated and filtered, and that can be carried over to a new version of the system. We also intend to study an extension to the case of rules with application conditions.

References

1. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: 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, Springer-Verlag (1996) 56–74
2. Ribeiro, L.: Parallel Composition and Unfolding Semantics of Graph Grammars. PhD thesis, TU Berlin (1996)
3. Heckel, R., Corradini, A., Ehrig, H., Löwe, M.: Horizontal and vertical structuring of typed graph transformation systems. Mathematical Structures in Computer Science **6**(6) (1996) 613–648
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code (Addison-Wesley Object Technology Series). Addison-Wesley Professional (July 1999)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer (2006)
6. AGG: AGG - Attributed Graph Grammar System Environment. `http://tfs.cs.tu-berlin.de/agg` (2007)
7. Baldan, P., Corradini, A., Montanari, U.: Contextual petri nets, asymmetric event structures, and processes. Information and Computation **171**(1) (2001) 1 – 49
8. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Transactions on Software Engineering **22** (1996)
9. Leung, H., White, L.: Insights into regression testing [software testing]. In: Software Maintenance, 1989., Proceedings., Conference on. (Oct 1989) 60–69
10. Engels, G., Heckel, R., Cherchago, A.: Flexible interconnection of graph transformation modules - a systematic approach. In Kreowski, H.J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G., eds.: Formal Methods in Software and System Modeling. LNCS, Springer-Verlag (2005) 38–63
11. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In: ICGT '08: Proceedings of the 4th international conference on Graph Transformations, Berlin, Heidelberg, Springer-Verlag (2008) 242–256
12. Baldan, P., Ehrig, H., König, B.: Composition and decomposition of DPO transformations with borrowed context. In: Proc. ICGT 2006. Volume 4178 of LNCS. (2006) 153–167
13. Rangel, G., König, B., Ehrig, H.: Bisimulation verification for the DPO approach with borrowed contexts. In: Proc. GT-VMT 2007. Volume 6 of ECEASST. (2007)
14. Diekert, V., Rozenberg, G., eds.: The Book of Traces. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1995)
15. Lambers, L., Ehrig, H., Taentzer, G.: Sufficient criteria for applicability and non-applicability of rule sequences. In: Proc. GT-VMT 2008. Volume 10 of ECEASST. (2008)