Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 236 (2009) 5–19

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Compositionality of Model Transformations

Dénes Bisztray[a,1] Reiko Heckel[a,2] Hartmut Ehrig[b,3]

[a] *Department of Computer Science, University of Leicester, United Kingdom*

[b] *Institut für Softwaretechnik und Theoretische Informatik, Technische Universität Berlin, Germany*

Abstract

Model transformations can be used not only for code or platform-specific model generation, but also for denotational semantics definition, e.g. using process algebras as semantics for visual modeling languages. Denotational semantics of programming languages are by definition compositional. In order to enjoy a similar property in the case of model transformations, every component of the source model should be distinguishable in the target model and the mapping compatible with syntactic and semantic composition. Since typed graphs are a natural representation of visual models, model transformations are often described by typed graph transformations. This paper proposes a formal definition of compositionality for mappings from typed graphs to semantic domains. To verify compositionality, syntactic criterion has been established for the implementation of the mappings by graph transformations with negative application conditions. An example compositional transformation is presented that maps architectural models described in UML component diagrams to CSP.

Keywords: compositionality, graph transformation, category theory, typed graphs, model driven engineering, denotational semantics

1 Introduction

As a consequence of the widespread use of visual languages, new applications for model transformations arise. Model transformation techniques are not only used for code or platform-specific model generation, but also for software refactoring, definition of semantics, formal verification and even architecture migration purposes [11]. Most of these applications are assumed to preserve the structure of the participant models, i.e. they should be compositional.

Compositionality is a property of model transformations that may be interpreted in two different ways according to [21]. *Sequential compositionality* is similar to function composition, i.e. given two transformations $f : A \to B$ and $g : B \to C$, it is possible to compose them for a transformation $g \circ f : A \to C$. While

[1] Email: dab24@mcs.le.ac.uk

[2] Email: reiko@mcs.le.ac.uk

[3] Email: ehrig@cs.tu-berlin.de

doi:10.1016/j.entcs.2009.03.011

sequential composition is obviously an essential property of any transformation, we are interested in *spatial composition.*

Spatial composition is similar to the compositionality property of denotational semantics. As for simple mathematical expressions, we assume that the meaning of expression $2 + 5$ is determined by the meaning of 2, 5 and the semantics of the $+$ operator, i.e. $[[2 + 5]] = [[2]] \bigoplus [[5]]$.

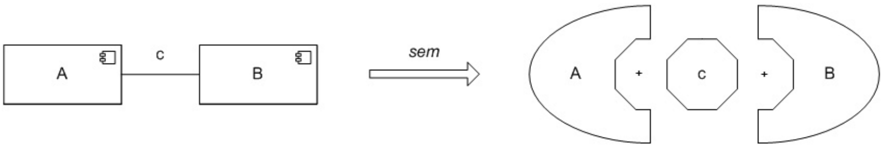

Figure 1. Compositional Semantic Mapping

In terms of model transformations, compositionality is presented in Figure 1. A system consisting of components A and B with a connector c is mapped to a semantic domain through transformation *sem*. The result is a set of semantic expressions $sem(A)$, $sem(B)$ and $sem(c)$ such that their composition represents the semantics of the whole system.

Compositionality is an important property for denotational semantics and thus for model transformations that establish a mapping between existing modelling artifacts and a semantic domain. Without compositionality, the modular specification and verification of model transformations would be impossible.

A typical semantic verification scenario is depicted in Figure 2. A modelling language (ML) is mapped to a semantic domain (SD) and programming language (PL) code is generated. To verify semantic consistency, a semantics of the programming language has to be defined typically through a mapping $PL \to SD$. The generated source code is semantically correct if the triangle commutes. Although the different model instances are numerous, they are composed from the basic elements (BE) of the modelling language. In case of compositional transformations the mapping can be described in terms of the basic building blocks, enabling the modular verification of various semantic properties.

Models and denotational semantics can be represented as instances of metamodels. A mathematical model is provided by type and instance graphs. Model transformations can be described by graph transformations. In this paper we present a notion of compositionality for any total functions defined by graph transformations between sets of graphs (representing models). Conditions are also provided and proved which guarantee compositionality for simple graph transformations and graph transformations with negative application conditions.

$$\begin{tikzcd}
BE \arrow[rrr, dotted, bend left] \arrow[drr, dotted] & \subseteq & ML \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & PL \arrow[dl] \\
 & & SD &
\end{tikzcd}$$

Figure 2. Semantic Verification

The paper is organised as follows. In Section 2 related approaches are summarised. Compositionality is formally introduced in Section 3. In Section 4 a condition for compositionality is given for simple graph transformations, generalised to graph transformations with negative application conditions in Section 5. In Section 6 a case study of compositional model transformations is introduced. Section 7 concludes the paper.

2 Related Work

Several contributions have been published on the verification of graph transformations.

To ensure uniqueness of results the concepts of confluence and critical pair analysis were presented in [12]. A working implementation can be found in AGG [1] for checking critical pairs [5]. Criteria for termination were introduced in [6,17]. An implementation for termination checking is also provided in AGG [4]. These definitions are applied to graph transformations with basic control conditions in [14] through the introduction of transformation units. Compositionality in bidirectional transformations discussed in [21].

In terms of bidirectional transformations that may be used as semantic relation between the model and semantic domain, there are two notable approaches: Triple Graph Grammars (TGGs) [20] and QVT [18]. TGGs have reliable tool support [10].

3 Formalising Compositionality

In this section compositionality is introduced formally. As the results proposed in this paper are generic with respect to the semantic domain, we provide a general, axiomatic definition.

Definition 3.1 (semantic domain) A *semantic domain* is a triple $(D, \sqsubseteq, \mathcal{C})$ where D is a set, $\sqsubseteq$ is a partial order on D, $\mathcal{C}$ is a set of total functions $C[\,] : D \to D$, called contexts, such that $d \sqsubseteq e \implies C[d] \sqsubseteq C[e]$ ($\sqsubseteq$ is closed under contexts).

The equivalence relation $\equiv$ is the symmetric closure of $\sqsubseteq$.

Both source and target models and their semantics are represented as typed graphs. For clarity, we present the definition of typed graphs that we use.

Definition 3.2 (typed graph and typed graph morphism [7]) Given type graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$, V_{TG} and E_{TG} are called vertex and edge label alphabet respectively. Then, a tuple $(G, type)$ of a graph G together with a graph morphism $type : G \to TG$ is called a *typed graph.*

Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a typed graph morphism $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \circ f = type_1$.

$$\begin{tikzcd}
G_1 \arrow[r, "f"] \arrow[d, "type_1"'] \arrow[dr, phantom, "="] & G_2 \arrow[dl, "type_2"] \\
TG &
\end{tikzcd}$$

The concept of context is central to compositionality. In set-theoretic terms a context C of graph D in graph G is given by $C = G \setminus D$. Context C is not necessarily a graph due to the possible dangling edges. To have a correct separation of context and included graph, we need a special set of contexts $\mathcal{S}_\mathcal{C}$ and set of graphs $\mathcal{S}_\mathcal{G}$.

Definition 3.3 (Compositionality) A semantic mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ with set of contexts $\mathcal{S}_\mathcal{C}$ and set of graphs $\mathcal{S}_\mathcal{G}$ is *compositional* if, for any injective $m_0 : G_0 \to H_0$ and pushout (1) with $G_0, G'_0 \in \mathcal{S}_\mathcal{G}$, there is a context $E \in \mathcal{S}_\mathcal{C}$ with $sem(H_0) \equiv E[sem(G_0)]$ and $sem(H'_0) \equiv E[sem(G'_0)]$

$$\begin{tikzcd}
G_0 \arrow[r] \arrow[d, "m_0"'] \arrow[dr, phantom, "(1)"] & G'_0 \arrow[d, "m'_0"] \\
H_0 \arrow[r] & H'_0
\end{tikzcd}$$

Intuitively the concept of compositionality is depicted in Figure 3. The semantic expression generated from G contains the one derived from L, through the inclusion morphism m.

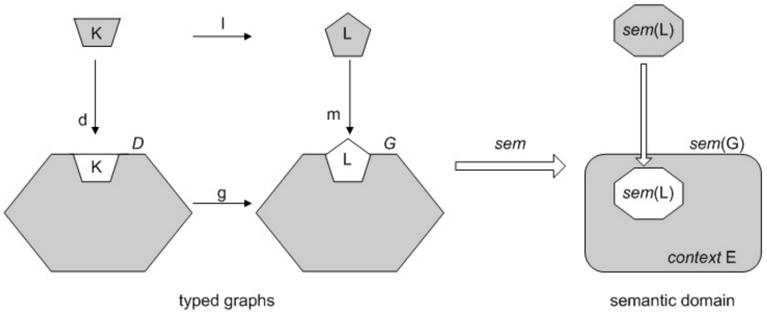

Figure 3. Intuitive approach to compositionality

Theorem 3.4 *Assume a compositional mapping sem* $: Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$. *Then, for all transformations* $G \overset{p,m}{\Longrightarrow} H$ *via rule* $p : L \to R$ *with injective match* m, *it holds that* $sem(L) \sqsubseteq sem(R)$ *implies* $sem(G) \sqsubseteq sem(H)$.

Proof Pushout (1) implies that $sem(G) \equiv E[sem(L)]$ and $sem(H) \equiv E[sem(R)]$.

$$\begin{tikzcd}
L \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & R \arrow[d] \\
G \arrow[r] & H
\end{tikzcd}$$

Now, $E[sem(L)] \sqsubseteq E[sem(R)]$ since $sem(L) \sqsubseteq sem(R)$ and $\sqsubseteq$ is closed under context. Hence $sem(G) \equiv E[sem(L)] \sqsubseteq E[sem(R)] \equiv sem(H)$ □

The statements in Theorem 3.4 also hold for the relation $\equiv$, obtained as the symmetric closure of $\sqsubseteq$.

4 Compositionality of Basic Graph Transformations

After giving an abstract definition in Section 3, in this section we prove a condition for compositionality of semantic mappings specified by graph transformations without negative application conditions. We assume that the semantic mapping *sem* is defined by a typed graph transformation system $GTS = (TG, P)$ consisting of a type graph TG and a set of typed graph productions P.

Definition 4.1 (nontermination) A graph G is *non-terminating* with respect to a typed graph transformation system $GTS = (TG, P)$ if $\exists p \in P$ that is applicable to G. The notation for a non-terminating graph is $G \gg$.

Definition 4.2 The result of the semantic mapping *sem* on a source graph G_0 is $sem(G_0) = G_n$ if and only if there is a transformation $G_0 \overset{p_1}{\Rightarrow} G_1 ... G_{n-1} \overset{p_n}{\Rightarrow} G_n$ with rules $p_1, ..., p_n \in P$ which is terminating.

It is important to note that only a *locally confluent* and *terminating* [7] transformation produces a unique result for a source graph. Thus, *local confluence* is also required for *sem*.

Definition 4.3 (separable semantics) A semantic mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ is *separable* with respect to a set of contexts $\mathcal{S}_{\mathcal{C}}$ and set of graphs $\mathcal{S}_{\mathcal{G}}$ if for all pushouts (1) with $G \in \mathcal{S}_{\mathcal{G}}$ and $C \in \mathcal{S}_{\mathcal{C}}$ it holds that if $H \gg$ then either $G \gg$ or $C \gg$.

$$\begin{tikzcd}
B \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & G \arrow[d] \\
C \arrow[r] & H
\end{tikzcd}$$

We recall the definition of the initial pushout [7], as it is used extensively throughout the paper.

Definition 4.4 (initial pushout) Given a morphism $f : A \to A'$, an injective morphism $b : B \to A$ is called the *boundary* over f if there is a pushout complement of f and b such that (1) is a pushout initial over f. Initiality of (1) over f means, that for every pushout (2) with injective b' there exists unique morphism $b^* : B \to D$ and $c^* : C \to E$ with injective b^* and c^* such that $b' \circ b^* = b$, $c' \circ c^* = c$ and (3) is a pushout. B is then called the boundary object and C the context with respect to f.

$$\begin{tikzcd}
B \arrow[r, "b"] \arrow[d] \arrow[dr, phantom, "(1)"] & A \arrow[d, "f"] & B \arrow[r, "b^*"] \arrow[rr, bend left, "b"] \arrow[d] \arrow[dr, phantom, "(3)"] & D \arrow[r, "b'"] \arrow[d] \arrow[dr, phantom, "(2)"] & A \arrow[d, "f"] \\
C \arrow[r, "c"] & A' & C \arrow[r, "c^*"] \arrow[rr, bend right, "c"] & E \arrow[r, "c'"] & A
\end{tikzcd}$$

For the transformation $sem : G_0 \overset{*}{\Rightarrow} G_n$ we create a boundary graph B and a context graph C through an initial pushout. The boundary graph is the smallest subgraph of G_0 which contains the identification points and dangling points of m_0.

The next definition is *IPO compatibility* of semantic mappings. While *compositionality* was defined through an unknown context E, *IPO compatiblity* defines it through the semantics of the context graph.

Definition 4.5 (IPO Compatibility) A semantic mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ with sets $\mathcal{S}_\mathcal{C}$ and $\mathcal{S}_\mathcal{G}$ is *initial pushout compatible* (IPO compatible) if for any injective $m_0 : G_0 \to H_0$ and initial pushout (2) over m_0 we have $sem(H_0) \equiv sem(C)[sem(G_0)]$ and $C \in \mathcal{S}_\mathcal{C}, G_0 \in \mathcal{S}_\mathcal{G}$.

$$\begin{tikzcd}
B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & G_0 \arrow[d, "m_0"] \\
C \arrow[r] & H_0
\end{tikzcd}$$

Lemma 4.6 *If a semantic mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ with sets $\mathcal{S}_\mathcal{G}$ and $\mathcal{S}_\mathcal{C}$ is IPO compatible, then it is also compositional.*

Proof Given pushout (1) with injective morphism $m_0 : G_0 \to H_0$ and initial pushout (2) over m_0. The closure property of initial pushouts [7] implies that pushout (2) + (1) is also initial over m'_0.

$$\begin{tikzcd}
B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & G_0 \arrow[r] \arrow[d, "m_0"] \arrow[dr, phantom, "(1)"] & G'_0 \arrow[d, "m'_0"] \\
C \arrow[r] & H_0 \arrow[r] & H'_0
\end{tikzcd}$$

Since sem is IPO-compatible with (2), $sem(H_0) \equiv sem(C)[sem(G_0)]$ and $G_0 \in \mathcal{S}_\mathcal{G}$. As (2) + (1) is also an initial pushout, sem is compatible with it, and thus $sem(H'_0) \equiv sem(C)[sem(G'_0)]$ and $G'_0 \in \mathcal{S}_\mathcal{G}$ as well. Hence sem is compositional with $E = sem(C) \in \mathcal{S}_\mathcal{C}$. □

The definition of *initial-preserving* graph transformations is inspired by the world of Triple Graph Grammars [20]. We assume an implicit source model left untouched by the transformation, while the transformation constructs the target model.

Definition 4.7 (initial-preserving) A (typed) graph transformation $t : G_0 \overset{*}{\Longrightarrow} G_n$ is *initial-preserving* if it is non-deleting with respect to its initial graph G_0.

A (typed) graph transformation system $GTS = (TG, P)$ is *initial-preserving* if all transformation sequences are *initial-preserving.*

Although *initial-preservation* seems similar to *nondeletion*, but elements of the target model may be deleted or modified through the transformation process.

Theorem 4.8 ***(Basic Compositionality Theorem)*** *A semantic mapping* $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ *with sets* $\mathcal{S}_{\mathcal{G}}$ *and* $\mathcal{S}_{\mathcal{C}}$ *is compositional if it is* initial-preserving *and* separable.

Proof

The main argument is based on the *Embedding Theorem* [7]. For the transformation $sem : G_0 \overset{*}{\Rightarrow} G_n$ we create a boundary graph B and context graph C. The boundary graph is the smallest subgraph of G_0 which contains the identification points and dangling points of m_0. Pushout (2) is the initial pushout of m_0.

$$\begin{tikzcd}
B \arrow[r, "b"] \arrow[d] \arrow[rr, bend left, "b"] \arrow[dr, phantom, "(2)"] & G_0 \arrow[d, "m_0"] \arrow[rr, Rightarrow, bend right, "sem_a"'] & G_0 \arrow[l, "id"'] \arrow[r, "d_n"] & G_n \\
C_0 \arrow[r] & H_0 & &
\end{tikzcd}
\qquad
\begin{tikzcd}
B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & G_0 \arrow[r, Rightarrow, "sem_a"] \arrow[d, "m_0"] \arrow[dr, phantom, "(3)"] & G_n \arrow[d] \\
C_0 \arrow[r, "c_0"] \arrow[d, Rightarrow, "sem_b"'] \arrow[dr, phantom, "(4)"] & H_0 \arrow[r, Rightarrow, "sem_a"] \arrow[d, Rightarrow, "sem_b"] & H_n \\
C_m \arrow[r] & H_m &
\end{tikzcd}$$

Since t is *initial-preserving* the consistency diagram [7] above can be used with initial pushout (2). G_0 replaces D as it is preserved throughout the transformation. Hence m_0 is consistent with respect to sem and there is an *extension diagram* over sem and m_0 [7]. Transformations sem_a only denotes particular rule application order of sem.

This essentially means that (3) is a pushout and H_n is the pushout object of sem and m_0, thus can be determined without applying the transformation sem on H_0.

While $sem(G_0) = G_n$ and thus $G_0 \overset{*}{\Rightarrow} G_n$ is terminating, H_n is possibly not terminating. The parts of H_0 not present in G_0 were not transformed to the semantic domain by the rules in sem_a, but the reasoning above holds for C_0 as well. The extension diagram over C_0 is pushout (4) and $sem(C_0) = C_m$. The termination of H_m is also unknown.

According to the Concurrency Theorem [7] the concurrent production can be created for both $H_0 \overset{*}{\Rightarrow} H_m$ and $H_0 \overset{*}{\Rightarrow} H_n$. Since the transformation is *initial-preserving*, the resultant morphisms h_n and h_m are inclusions (or identities) and the extension diagrams (5) and (6) exist. Since pushouts are unique, (5) = (6) and thus $H_A = H_B = H$.

$$\begin{tikzcd}
H_0 \arrow[r, Rightarrow, "sem_a"] \arrow[d, "h_m"'] \arrow[dr, phantom, "(5)"] & H_n \arrow[d, "h_a"] \\
H_m \arrow[r, Rightarrow, "sem_a"] & H_A
\end{tikzcd}
\quad
\begin{tikzcd}
H_0 \arrow[r, "h_n"] \arrow[d, Rightarrow, "sem_b"'] \arrow[dr, phantom, "(6)"] & H_n \arrow[d, Rightarrow, "sem_b"] \\
H_m \arrow[r, "h_b"] & H_B
\end{tikzcd}$$

This leads to the diagram below. Since $G_n \in \mathcal{G}_{\mathcal{S}}$ and $C_m \in \mathcal{G}_{\mathcal{C}}$, the semantics is *separable* and they are terminating (i.e. no semantic rule applicable), H must be also terminating. If H is terminating, that means $sem(H_0) = H$.

$$\begin{tikzcd}
B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & G_0 \arrow[r, Rightarrow, "sem_a"] \arrow[d, "m_0"] \arrow[dr, phantom, "(3)"] & G_n \arrow[d] \\
C_0 \arrow[r, "c_0"] \arrow[d, Rightarrow, "sem_b"] \arrow[dr, phantom, "(4)"] & H_0 \arrow[r, Rightarrow, "sem_a"] \arrow[d, Rightarrow, "sem_b"] \arrow[dr, phantom, "(5)"] & H_n \arrow[d, Rightarrow, "sem_b"] \\
C_m \arrow[r] & H_m \arrow[r, Rightarrow, "sem_a"] & H
\end{tikzcd}$$

According to the composition property of pushouts [7], $(2) + (3)$ and $(4) + (5)$ are pushouts and thus the big $(2)+(3)+(4)+(5)$ square is a pushout as well. Since H is a pushout object, $H \cong G_n +_B C_m = C_m[G_n]$ which means that $sem(H_0) = sem(C_0)[sem(G_0)]$. Hence *sem* is *IPO compatible* and according to Lemma 4.6 it is also compositional. □

5 Graph Transformations with NACs

In Section 4 the compositionality of graph transformations with non-deleting rules was proved. However in order to control the transformation, negative application conditions (NACs) need to be used as well.

The definition of *separable* semantics (Definition 4.3) carries over to this section with the presence of negative application conditions allowed.

Definition 5.1 (negative application condition [7]) A simple negative application condition is of the form $NAC(x)$, where $x : L \to X$ is a morphism. A morphism $m : L \to G$ satisfies $NAC(x)$ if there does not exist a morphism $p : X \to G$ in M' with $p \circ x = m$:

$$\begin{tikzcd}
L \arrow[r, "x"] \arrow[d, "m"] & X \arrow[dl, dotted, "p"] \\
G &
\end{tikzcd}$$

Before the establishment of Theorem 5.6, the necessary definitions are presented.

Definition 5.2 (gluing and created points) Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$.

- The gluing points GP are those nodes and edges in L that are not deleted by p, i.e. $GP = l_V(V_K) \cup l_E(E_K) = l(K)$.
- The created points CP are those nodes and edges in R that are created by p, i.e. $CP = r_V(V_K)\backslash V_K \cup r_E(E_K)\backslash E_K$.

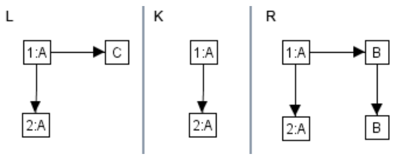

Figure 4. Production Rule with Created Points

The concept of created points is demonstrated in Figure 4. Since the C node is deleted, the only gluing points are the two A nodes. They are not deleted by the rule in Fig. 4. The created points are the B nodes on the right hand side of the rule.

It is possible that the B nodes are always - if present - gluing points in every production rule of the graph transformation system. This means that the node type B is such a special type that its instances are never deleted. This observation leads to the definition of *constant types* that are already present in the start graph are not deleted throughout the transformation.

Definition 5.3 (constant types) Given a typed graph transformation system $GTS = (TG, P)$. Constant types $CT \subseteq TG = (V_{CT}, E_{CT})$ are those nodes and edges in the type graph TG, whose instances are not deleted or modified by any production $p \in P$. i.e. $CT = \{v \in V_{TG} \mid \forall p \in P : v = type_V(w) \wedge w \in GP_{p_i}\} \cup \{e \in E_{TG} \mid \forall p \in P : e = type_E(f) \wedge f \in GP_{p_i}\}$.

In an instance graph, constant points are those nodes and edges that are of a constant type.

The definition of *constructive transformations* are inspired by Triple Graph Grammars. While the NACs contain only non-constant elements, the intial graph consists exclusively constant points. This way the NACs concentrate on the target elements of the transformation.

Definition 5.4 (constructive transformation) A graph transformation $t : G \overset{*}{\Rightarrow} G_n$ with NACs through embedding $m_0 : G_0 \to H_0$ is *constructive* if

(i) G_0 and H_0 contains only constant points, i.e. $type(G_0), type(H_0) \subseteq CT$.

(ii) all NACs consist of non-constant points, i.e. $\forall p \in P$ with each $NAC(n), n : L \to N$ of p we have $\exists x \in N \setminus n(L)$ with $type(x) \notin CT$

Corollary 5.5 *A* constructive *graph transformation $t : G \overset{*}{\Rightarrow} H$ with NACs is also* initial-preserving *because G consists of constant points that are not deleted through the transformation.*

Theorem 5.6 ***(compositionality theorem with nacs)*** *Given a semantic mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ with sets $\mathcal{S}_{\mathcal{G}}$ and S_C described by a graph transformation system $GTS = (TG, P)$ with constant types $CT \subseteq TG$. Then, it is compositional if it is separable and constructive.*

Proof The proof is based on the *basic compositionality theorem.* In order to apply the *Embedding Theorem* in the proof of Theorem 4.8, it is suffice to show that the extension diagrams over m_0 and c_0 exist in the presence of NACs.

As the equivalent left NACs can be constructed from the right NACs, the NACs throughout this proof are assumed to be left NACs, if not explicitly stated on the contrary.

The extension diagram exists in case of NACs, if the transformation not only boundary-consistent [15], but also NAC-consistent [16]. According to the synthesis construction of *Concurrency Theorem* a concurrent rule p_c with a concurrent match

g_c exists [7]. The concurrent rule p_c is basically the merge of all rules of a specific rule application order in *sem* : $G_0 \overset{*}{\Rightarrow} G_n$ such that the target graph G_n is produced by the application of p_c on the source graph G_0.

In graph transformations containing NACs, a concurrent NAC_{p_c} exists for the concurrent rule p_c. To achieve NAC-consistency, we have to show, that $k_0 \circ g_c \models NAC_{p_c}$ with NAC_{p_c} being the concurrent NAC, g_c the concurrent match induced by t and $k_0 : G_0 \to H_0$ the inclusion morphism [16].

Since $type(H_0) \subseteq CT$, this follows from Lemma 5.7, because the existence of a morphism $q : N_c \to H_0$ that satisfies an arbitrary NAC would imply for $x \in N_C \setminus n(L_C)$ with $type(x) \notin CT$ also $type(q(x)) = type(x) \notin CT$ which is contradiction to $type(H_0) \subseteq CT$. □

Lemma 5.7 *All NACs of p_c are non-constant provided that $\forall p \in P$ have non-constant NACs.*

Proof The proof is by mathematical induction over the length of *sem* : $G_0 \overset{*}{\Rightarrow} G_n$.

Basis. $n = 1$. We have $p_c = p_0$ which has the property of the assumption.

Induction Step. Consider $t_n : G_0 \overset{n}{\Rightarrow} G_n \Rightarrow G_{n+1}$ via the rules $p_0, p_1, ..., p_n$. We can assume by induction that $p'_c = N_i \xleftarrow{n_i} L'_c \leftarrow K'_c \rightarrow R'_c$ (the concurrent rule for $p_0, p_1, ..., p_{n-1}$) and $p_n : N_j \xleftarrow{n_j} L_n \leftarrow K_n \rightarrow R_n$ have non-constant NACs. We have to show that all NACs for p_c are non-constant.

$$\begin{tikzcd}
N_i \arrow[d, dotted, "e_i"'] & & & & & N_j \arrow[dl, dotted, "z_1"'] & & \\
N'_i \arrow[r, phantom, "(1)"] & L'_c \arrow[ul, "n_i"'] \arrow[d, dotted, "m_c"] \arrow[ddd, bend right, "g'_c"'] & K'_c \arrow[l] \arrow[r] \arrow[d, dotted] & R'_c \arrow[dr, dotted, "e'_c"'] \arrow[dddr, bend right, "h'_c"'] & N'_j \arrow[r, phantom, "(3)"] & L_n \arrow[u, "n_j"'] \arrow[dl, dotted, "e_n"] \arrow[dddl, bend left, "g_n"'] & K_n \arrow[l] \arrow[r] \arrow[d, dotted] & R_n \arrow[d, dotted] \arrow[ddd, bend left, "g_{n+1}"] \\
 & L_c \arrow[ul, dotted, "n'_i"] \arrow[dd, dotted, "g_c"] & C_c \arrow[l, dotted, "l"'] \arrow[rr, dotted] \arrow[dd, dotted] & & E \arrow[u, dotted, "n'_j"] \arrow[dd, dotted, "h"] & & C_n \arrow[ll, dotted] \arrow[r, dotted, "r"] \arrow[dd, dotted] & R_c \arrow[dd, dotted, "h_c"'] \\
 & & & K_c \arrow[ul, dotted, "k_c"] \arrow[urrr, dotted, "k_n"'] \arrow[u, phantom, "(2)"] & & & & \\
 & G_0 & D_n \arrow[l] \arrow[rr] & & G_n & & D \arrow[ll] \arrow[r] & G_{n+1}
\end{tikzcd}$$

According to the synthesis construction of *Concurrency Theorem with NACs* the concurrent rule p_c with NACs induced by $G_0 \overset{n+1}{\Longrightarrow} G_{n+1}$ is $p_c = L_c \xleftarrow{l \circ k_c} K_c \xrightarrow{r \circ k_n} R_c$ (with match $g_c : L_c \to G_0$, comatch $h_c : R_c \to G_{n+1}$). The concurrent NAC_{p_c} consists two parts.

Case 1

$n'_i : L_C \to N'_i$ defined by $n_i : L'_C \to N_i$ from p'_c.

By assumption ii of *constructiveness* we have $\exists x_i \in N_i \setminus n_i(L'_C)$ with $type(x_i) \notin CT$. Let $x'_i = e_i(x_i)$ such that $type(x'_i) = type(x_i) \notin CT$. Moreover $x'_i \in N'_i \setminus n'_i(L_C)$

because otherwise pushout and pullback (1) implies that $\exists y_i \in L'_C$ with $n_i(y_i) = x_i$ and hence $x_i \in n_i(L'_C)$ which is a contradiction. Thus n'_i is non-constant.

Case 2

$n''_j : L_C \to N''_j$ defined by $n_j : L_n \to N_j$ with $n'_j : E \to N_j$ through pushouts (3) – (5).

$$\begin{tikzcd}
N''_j & Z \arrow[l] \arrow[r] & N'_j \\
L_c \arrow[u, "n''_j"] \arrow[ur, phantom, "(5)"] & C_c \arrow[l] \arrow[r] \arrow[u] \arrow[ur, phantom, "(4)"] & E \arrow[u, "n'_j"']
\end{tikzcd}$$

If the pushout complement C_c of (4) does not exists, the induced NAAC is always true.

By assumption on p_n we have $x_j \in N_j \setminus n_j(L_n)$ with $type(x_j) \notin CT$. Because (3) is a pushout and pullback, $\exists x'_j = z_1(x_j) \in N_j \setminus n'_j(E)$ and $type(x'_j) = type(x_j) \notin CT$. Also $\exists y_j \in Z \setminus C_c$ with $(Z \to N'_j)(y_j) = x'_j$ using pushout (4) with $type(y_j) = type(x'_j) \notin CT$. And finally $\exists x''_j = (Z \to N''_j)(y_j) \in N''_j \setminus n''_j(L_C)$ because (5) is a pushout and pullback with type $type(x''_j) = type(y_j) \notin CT$. Thus n''_j is non-constant. $\square$

6 Application to the Case Study

Compositionality is an important property of typed graph transformations in the field of semantic verification. In this section we present an application of the presented theoretical concepts using the mapping mentioned in [3].

In order to improve the internal structure, performance or scalability of a software system, behaviour-preserving changes are introduced known as refactorings [9]. As the applications of today tend to be service-oriented, we deal with refactorings of business workflows.

We use UML activity diagrams specifying the workflows executed by service instances [19]. The semantics of the relevant fragment of the UML is expressed in a denotational style, using CSP [13] as semantic domain and defining the mapping from UML diagrams to CSP processes by means of graph transformation rules. The semantic relation of behaviour preservation can be expressed using one of the refinement and equivalence relations on CSP processes, and checked using FDR2 [8].

6.1 Short Introduction to CSP

In this section we briefly introduce the necessary concepts from CSP. A *Process* is the behaviour pattern of a component with an alphabet of events. Processes are defined using recursive equations called *ProcessAssignments*. The *ProcessExpressions* used in Section 6.2 are based on the following syntax.

$$P ::= event \to P \mid P \parallel Q \mid R$$

The *PrefixOperator* $a \rightarrow P$ performs action a and then behaves like P. The *Concurrency* process $P \parallel Q$ behaves as P and Q engaged in a lock-step synchronisation.

Formally, the UML models are instances of metamodels represented by attributed typed graphs. The abstract syntax of CSP can be also represented as a typed graph. The typed graph based metamodel for CSP is defined in [2]. This way, the semantic mapping can be defined as a typed graph transformation.

CSP is a semantic domain in the sense of Definition 3.1. D is the set of CSP expressions and $\sqsubseteq$ can be trace, failure or divergence refinement as they are closed under context [13]. A context is a process expression $E(X)$ with a single occurrence of a distinguished process variable X.

6.2 Implementation

Although the transformation mechanics was inspired by TGGs, the transformation was implemented in Tiger EMF Transformer [22] and thus it is unidirectional.

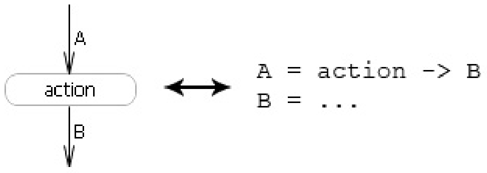

Figure 5. Mapping of Actions and ActivityEdges

To present the definitions in practice, a simple pair of rules are introduced in the following. Figure 5 shows their concrete syntax. The rules in Figure 6 and 7 are responsible for creating the edge processes, as well as weaving the edge processes to the events created from the actions. The attribute values A and B denote variables that holds the same values in both sides of the rule.

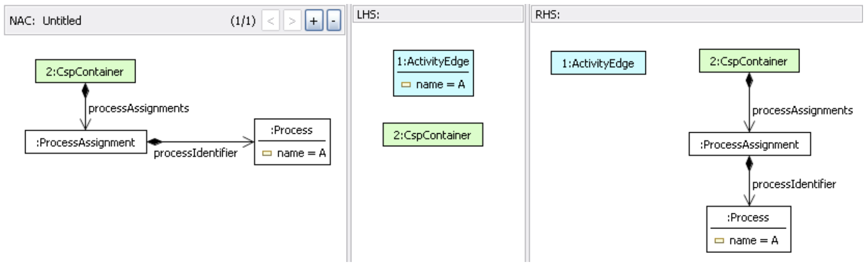

Figure 6. Implementation of Edge Rule

The *Edge* rule in Figure 6 creates the process definition for the corresponding edge. The NAC, defined on CSP expressions, checks the existence of a similar process definition. If none exist, the matched edge has not been transformed yet. Thus it creates the empty $A =$ process definition.

The *Action* rule in Figure 7 creates an event from the corresponding action and inserts it to the definition of the related process. The NAC, consisting of CSP expressions only, works the same way as the one in the *Edge* rule.

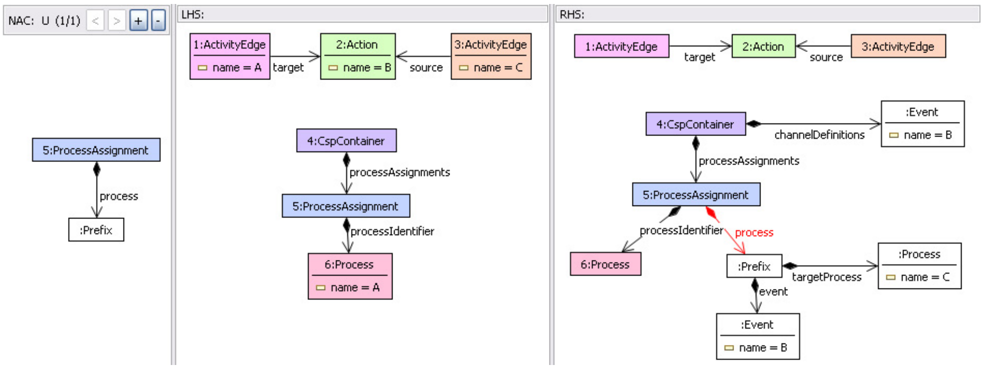

Figure 7. Implementation of Action Rule

6.3 *Proof of Compositionality*

This semantic transformation is compositional, if it is *terminating*, *locally confluent*, *separable*, *integrated* and *constructive*.

Termination and *local confluence* can be proven through a critial pair and termination analysis in AGG [1].

The semantic mapping basically reads the architectural model and creates the corresponding set of CSP expressions. None of the rules deletes any element typed from the CSP metamodel. Thus, all elements of the CSP metamodel are stable types, since all instances of them are stable points in this semantic mapping. Also, the NACs are defined on CSP expressions only. These three observations corresponds to the assumptions of Definition 5.4, thus the transformation is *constructive*.

To prove *separablility* the sets $\mathcal{S}_{\mathcal{C}}$ and $\mathcal{S}_{\mathcal{G}}$ need to be defined. We define on set of graphs $\mathcal{S}_{\mathcal{AD}}$ that represents both. $\mathcal{S}_{\mathcal{AD}}$ is a set of subgraphs of H_0 such that if an *ActivityNode* is present in G, then all incoming and outgoing *ActivityEdges* connected to the node should be present as well. Since every transformation rule in *sem* transforms a single *ActivityNode* into the semantic domain, every node type has a related production rule in *sem*. As *ActivityEdges* are transformed to processes, they form a frame around the *ActivityNodes*, enabling their transformation. Thus, boundary graphs consist of only *ActivityEdges*. If all the incoming and outgoing *ActivityEdges* are included with the relevant node, all corresponding rules are triggered in the subgraph. Hence no new structures are created during the merge process that enables a previously disabled rule. Thus if C_0 and G_0 are of $\mathcal{S}_{\mathcal{AD}}$, then if C_0 and G_0 are terminating, then H as well.

7 Conclusions

In this paper the notion of compositionality has been formalised for mappings between typed graphs and semantic domains. The semantic mapping was represented by typed graph transformations. As a main result, conditions for compositionality has been established for simple graph transformations as well as graph transformations with negative application conditions with the necessary concepts defined. An example compositional transformation was presented that maps UML activity diagrams to CSP.

Future work includes the research on graph transformation with basic control structures as well as advanced control flow.

Acknowledgement

This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

References

[1] AGG - Attributed Graph Grammar System Environment (2007).
URL http://tfs.cs.tu-berlin.de/agg

[2] Bisztray, D. and R. Heckel, *Rule-level verification of business process transformations using csp*, in: *Proc. of 6th International Workshop on Graph Transformations and Visual Modeling Techniques (GTVMT'07)*, 2007.

[3] Bisztray, D., R. Heckel and H. Ehrig, *Verification of architectural refactorings by rule extraction*, in: *Proc. of Fundamental Approaches to Software Engineering (FASE'08)*, Lecture Notes in Computer Science, 2008, to appear.

[4] Bottoni, P., M. Koch, F. Parisi-Presicce and G. Taentzer, *Termination of high-level replacement units with application to model transformation.*, Electr. Notes Theor. Comput. Sci. **127** (2005), pp. 71–86.
URL http://dblp.uni-trier.de/db/journals/entcs/entcs127.html#BottoniKPT05

[5] Bottoni, P., G. Taentzer and A. Schürr, *Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation*, in: *Proceedings of the IEEE International Symposium on Visual Languages (VL'00)*, IEEE Computer Society, 2000, p. 59.

[6] Ehrig, H., K. Ehrig, J. de Lara, G. Taentzer, D. Varró and S. Varró-Gyapay, *Termination criteria for model transformation*, in: M. Cerioli, editor, *Proc. of International Conference on Fundamental Approaches to Software Engineering (FASE'05)*, Lecture Notes in Computer Science **3442** (2005), pp. 49–63.

[7] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation," Monographs in Theoretical Computer Science, Springer-Verlag, 2006.

[8] Formal Systems Europe Ltd, *FDR2 User Manual* (2005).
URL http://www.fsel.com/documentation/fdr2/html/index.html

[9] Fowler, M., K. Beck, J. Brant, W. Opdyke and D. Roberts, "Refactoring: Improving the Design of Existing Code," Addison-Wesley Professional, 1999, 1st edition.

[10] Fujaba Tool Suite (2007).
URL http://wwwcs.uni-paderborn.de/cs/fujaba

[11] Heckel, R., R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos and L. Andrade, "Architectural Transformations: From Legacy to Three-tier and Services," To appear.

[12] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, in: *Proc. of the 1st International Conference on Graph Transformation (ICGT'02)* (2002), pp. 161–176.

[13] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall International Series in Computer Science, Prentice Hall, 1985.

[14] Küster, J. M., *Definition and validation of model transformations*, Software and Systems Modeling **5** (2006), pp. 233–259.

[15] Lambers, L., *Adhesive high-level replacement systems with negative application conditions*, Technical report, Technische Universität Berlin (2007).

[16] Lambers, L., H. Ehrig, F. Orejas and U. Prange, *Adhesive high-level replacement systems with negative application conditions*, in: *Proc. of Applied and Computational Category Theory Workshop* (2007).

[17] Levendovszky, T., U. Prange and H. Ehrig, *Termination criteria for dpo transformations with injective matches*, Electron. Notes Theor. Comput. Sci. **175** (2007), pp. 87–100.

[18] MOF Query/View/Transformation (QVT) Final Adopted Specification (2005).
URL http://www.omg.org/docs/ptc/05-11-01.pdf

[19] OMG, *Unified Modeling Language, version 2.1.1* (2006).
URL http://www.omg.org/technology/documents/formal/uml.htm

[20] Schürr, A., *Specification of graph translators with triple graph grammars*, in: *Proc. of Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG'94)*, Lecture Notes in Computer Science **903** (1994), pp. 151–163.

[21] Stevens, P., *Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions*, in: *Proc. of 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, Lecture Notes in Computer Science **4735** (2007), pp. 1–15.

[22] Team, T. D., "Tiger EMF Transformer," (2007).
URL http://www.tfs.cs.tu-berlin.de/emftrans