GT-VC 2006

Graph Transformation for Verification and Concurrency

31 August 2006, Bonn, Germany
Satellite workshop of CONCUR 2006

Arend Rensink, Reiko Heckel, Barbara König

Table of Contents

Matching of Bigraphs
Lars Birkedal, Troels Christoffer Damgaard, Arne John Glenstrup, Robin Milner 1

Adhesive DPO Parallelism for Monic Matches
(Work-in-progress paper)
Filippo Bonchi, Tobias Heindel 19

A Graph Abstract Machine Describing Event Structure Composition
(Work-in-progress paper)
Claudia Faggian, Mauro Piccolo 34

Formal Verification of Object-Oriented Graph Grammars Specifications
Ana Paula Lüdtke Ferreira, Luciana Foss, Leila Ribeiro 49

Modeling and Verification of Reliable Messaging by Graph
Transformation Systems
László Gönczy, Máté Kovács, Dániel Varró 64

Stochastic Graph Transformation Systems with General Distributions
Piotr Kosiuczenko, Georgios Lajios 80

Verification of Random Graph Transformation Systems
(Work-in-progress paper)
Vitali Kozioura 95

Termination Criteria for DPO Transformations with Injective Matches
Tihamér Levendovszky, Ulrike Prange, Hartmut Ehrig 102

Matching of Bigraphs

Lars Birkedal [1] Troels Christoffer Damgaard [1]
Arne John Glenstrup [1]

IT University of Copenhagen, Denmark

Robin Milner [2]

University of Cambridge, UK

Abstract

We analyze the matching problem for bigraphs. In particular, we present a sound and complete inductive characterization of matching of binding bigraphs. Our results pave the way for a provably correct matching algorithm, as needed for an implementation of bigraphical reactive systems.

1 Introduction

Over the last decade, a theory of bigraphical reactive systems has been developed [9,12,14]. Bigraphical reactive systems (BRSs) provide a graphical model of computation in which both locality and connectivity are prominent. In essence, a *bigraph* consists of a *place graph*; a forest, whose nodes represent a variety of computational objects, and a *link graph*, which is a hyper graph connecting ports of the nodes. Bigraphs can be reconfigured by means of *reaction rules*. Loosely speaking, a *bigraphical reactive system* consists of set of bigraphs and a set of reaction rules, which can be used to reconfigure the set of bigraphs. BRSs have been developed with principally two aims in mind: (1) to be able to model directly important aspects of ubiquitous systems by focusing on mobile connectivity (the link graph) and mobile locality (the place graph), and (2) to provide a unification of existing theories by developing a general theory, in which many existing calculi for concurrency and mobility may be represented, with a uniform behavioural theory. The latter is achieved by representing the dynamics of bigraphs by an abstract definition of reaction rules from which a labelled transition system may be derived in such a way

[1] Email: {birkedal,tcd,panic}@itu.dk
[2] Email: Robin.Milner@cl.cam.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

that an associated bisimulation relation is a congruence relation. The unification has recovered existing behavioural theories for the π-calculus [9], the ambient calculus [8], and has contributed to that for Petri nets [11]. Thus the evaluation of the second aim has so far been encouraging. In [3], Birkedal et al. initiate an evaluation of the first aim, in particular it is shown how to give bigraphical models of context-aware systems.

As suggested and argued in [9,1,3] it would be very useful to have an implementation of the dynamics of bigraphical reactive systems to allow experimentation and simulation. In the Bigraphical Programming Languages research project at the IT University, we are working towards such an implementation. The core problem of implementing the dynamics of bigraphical reactive systems is the *matching problem*, that is, to determine for a given bigraph and reaction rule whether and how the reaction rule can be applied to rewrite the bigraph. The topic of the present paper is to analyze the matching problem.

In Figure 1 we show several bigraphs. Consider the bigraph named a. It is intended to model two buildings, one belonging to a corporation and one belonging to a consultancy group. Inside the buildings are laptops with data nested inside folders. The nesting structure depicts the place graph. Links are used to name the buildings and, moreover, to model which folders can be shared between the corporation and the consultancy group and inside the corporation. Thus the laptop shown in the middle is intended to belong to a consultant working for the corporation — the consultant has folders with data belonging to the consultancy group (the link shown to the left) and folders with data belonging to the corporation (the link shown to the right). The fact that folders belonging to the corporation should not leave the corporation is expressed by linking those folders to a so-called binding port on the corporation building, indicated by the circle.

The abstract semantic definition of matching, as defined in the theory of bigraphs [9], is roughly as follows (omitting many details): Given a reaction rule with redex R and reactum R' (with R and R' both bigraphs), and a bigraph a (the agent to be rewritten), if $a = C \circ (R \otimes \mathsf{id}_Z) \circ d$, then it can be rewritten to $C \circ (R' \otimes \mathsf{id}_Z) \circ d$. Here $\circ$ denotes composition of bigraphs and Z is the set of names of d. In other words, if the reaction rule *matches* a, in the sense that a can be decomposed into a context C, redex R and a parameter d, then a can be rewritten.

Consider again the example in Figure 1. There is a reaction rule expressed by the redex R and the reactum R'; the intention of the reaction rule is to allow copying of data between connected folders in the same nesting hierarchy (note the link in R between the two folders and the so-called local name y). The agent a can be written as a composition of C, R and d — formally, $a = C \circ (R \otimes \mathsf{id}_z) \circ d$. Composition works by (1) plugging the roots of R and d into the holes (aka sites) of C respectively R; (2) fusing together the connections between folder and z (in d) and z and folder (in C), removing the

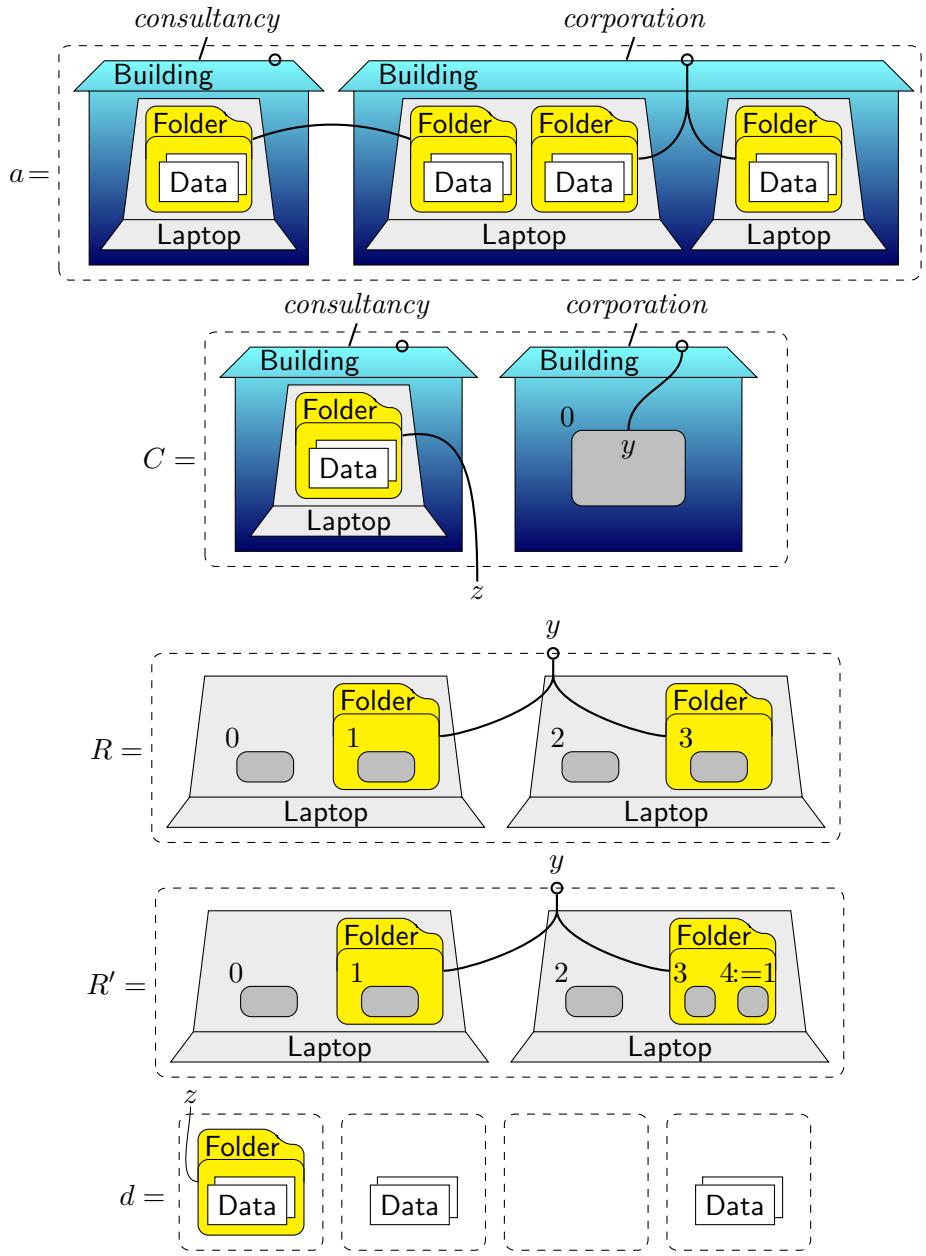


Fig. 1. Example of a ground agent $a = C \circ (\mathsf{id}_z \otimes R) \circ d$. Reaction rule $R \to R'$ copies data between connected folders.

name z in the process; (3) fusing together the connection between the local name y and the two folders in R and the name y and the bound port in C, removing the name y in the process. Note the use of id_z in the composition $a = C \circ (R \otimes \mathsf{id}_z) \circ d$; it allows a name z from the parameter d to be "passed through" the redex and be attached to something in the context C. The reactum R' contains a copy of the site numbered 1 in R, expressing that data is copied between the shared folders. The sites numbered 0 and 2 in R allow the reaction rule to apply also when the laptops contain other folders than the two that are connected. Thus a can be rewritten using the reaction rule to

another agent a' like a but with two data items in the rightmost laptop (the agent a' is not shown in Figure 1).

In the present paper we provide an *inductive characterization* of when $a = C \circ (R \otimes \mathsf{id}_Z) \circ d$ holds, by induction on a and R (the input to a matching algorithm). It is a precise characterization in the sense that it is both sound and complete with respect to the abstract definition. This provides a detailed analysis of the matching problem, and paves the way for developing and *proving correct* an actual matching algorithm (which, given a and R, must find C, d, and Z such that $a = C \circ (R \otimes \mathsf{id}_Z) \circ d$ holds). We further include a discussion of how one may derive matching algorithms from our inductive characterization. We will report on our work on an actual implementation of matching in a subsequent paper.

Our inductive characterization is based on normal form theorems for bigraphs [12,5], which express how general bigraphs may be decomposed into a composition of simpler graphs. The normal form theorems and also the inductive characterization we present here is based on so-called *discrete* decompositions of bigraphs. Discrete bigraphs are bigraphs with a simple form of linkage. To a large extent, this allows us to analyze matching of a general bigraph by considering its link graph and place graph separately.

Of course, the matching problem is closely related to the NP-complete graph embedding problem. Thus we analyze the embedding problem for a restricted class of graphs, and our inductive characterization makes good use of the algebraic presentation of such graphs [12,5]. One hopes that matching implementations will be efficient in practice since redices typically are small. Furthermore, sorting bigraphs [4] could be a source of early search elimination.

The remainder of this paper is organized as follows. In Section 2 give an informal description of binding bigraphs. The main contribution of this paper is in Section 3, where we present our inductive characterization of matching. Section 4 discusses how the inductive characterization may ensure a correct and efficient algorithm for matching. In the final sections we discuss related work and conclude.

For lack of space, most proofs [2] have been omitted from this extended abstract.

2 Binding Bigraphs

Here we present bigraphs informally; for a formal definition, see [9,5].

2.1 Concrete Bigraphs

A concrete binding bigraph G consists of a *place graph* $G^{\mathbf{P}}$ and a *link graph* $G^{\mathbf{L}}$. The place graph is an ordered list of trees indicating *location*, with roots $r_0, \ldots, r_n$, nodes $v_0, \ldots, v_k$, and a number of special leaves $s_0, \ldots, s_m$ called *sites*, while the link graph is a general graph over the node set $v_0, \ldots, v_k$ ex-

tended with *inner names* $x_0, \ldots, x_l$, and equipped with hyper edges, indicating *connectivity*.

We usually illustrate the place graph by nesting nodes, as shown in the upper part of Figure 2 (ignore for now the interfaces denoted by " $: \cdot \to \cdot$"). A *link* is a hyper edge of the link graph, either an *internal edge* e or a *name*

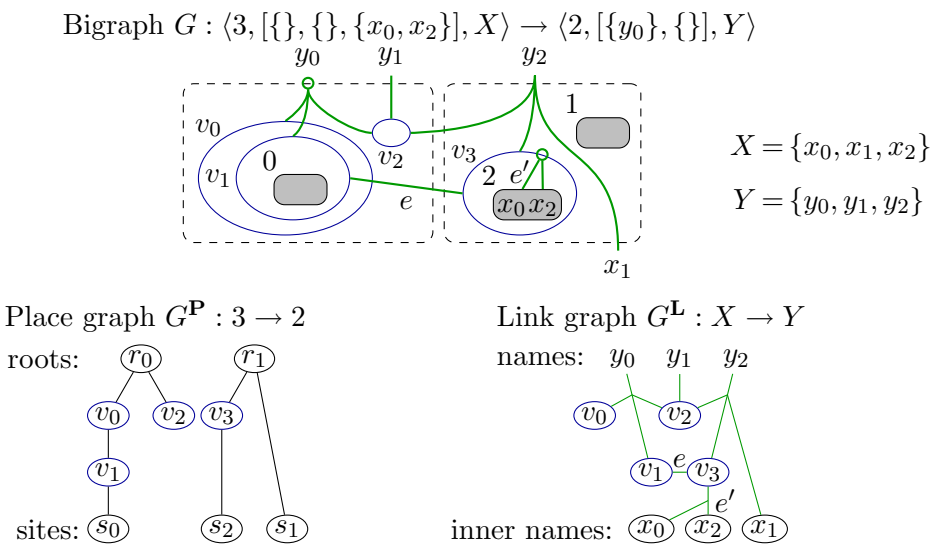


Fig. 2. Example bigraph illustrated by nesting and as place and link graph.

y. Names and inner names can be *global* or *local*, the latter being located at a specific root or site, respectively. In Figure 2, y_0 is located at r_0, indicated by a small ring, and x_0 and x_2 are located at s_2, indicated by writing them within the site. Global names like y_1 and y_2 are drawn anywhere at the top, while global inner names like x_1 are drawn anywhere at the bottom. A link, including internal edges like e' in the figure, can be located with one *binder* (the ring), in which case it is a *bound link,* otherwise it is *free*. However, a bound link must satisfy the *scope rule*, a simple structural requirement that all points of the link lie within its location (in the place graph), except for the binder itself. This prevents y_2 and e in the example from being bound.

2.2 Controls

Every node v has a *control* K which determines a binding and free arity, indicated by $v : K$. In the example of Figure 2, we could have $v_i : K_i, i = 0, 1, 2, 3$, where $K_0 : 0 \to 1$, $K_1 : 0 \to 2$, $K_2 : 0 \to 3$, $K_3 : 1 \to 2$. The arities determine the number of bound and free *ports* of the node, to which bound and free links, respectively, are connected. Ports and inner names are collectively referred to as *points*.

2.3 Abstract Bigraphs

While concrete bigraphs with named nodes and internal edges are the basis of bigraph theory [9], our prime interest is in *abstract bigraphs,* equivalence

classes of concrete bigraphs that differ only in the names of nodes and internal edges. Abstract bigraphs are illustrated with their node controls, as shown in Figure 1 with Building, Laptop etc. In what follows, "bigraph" will thus mean "abstract bigraph."

2.4 Interfaces

Every bigraph G has two *interfaces* I and J, written $G : I \to J$, where I is the *inner face* and J the *outer face*. An interface is a triple $\langle m, \vec{X}, X \rangle$, where m is the *width* (the number of sites or roots), X the entire set of local and global names, and $\vec{X}$ indicates the locations of each local name, cf. Figure 2. We let $\epsilon = \langle 0, [\,], \{\} \rangle$; when $m = 1$ the interface is *prime,* and if all $x \in X$ are located by $\vec{X}$, the interface is *local.*

A bigraph $G : I \to J$ is called *ground,* or an *agent*, if $I = \epsilon$, *prime* if I is local and J prime, and a *wiring* if $m = n = 0$, where m and n are the widths of I and J, respectively. For $I = \langle m, \vec{X}, X \rangle$, bigraph $\mathsf{id}_I : I \to I$ consists of m roots, each root r_i containing just one site s_i, and a link graph linking each inner name $x \in X$ to name x.

2.5 Discrete and Regular Bigraphs

We say that a bigraph is *discrete* iff every free link is a name and has exactly one point. The virtue of discrete bigraphs is that any connectivity by internal edges must be bound, and node ports can be accessed individually by the names of the outer face. In Figure 1, only R, R' and d are discrete, because the free internal edges of a and C have two points. Further, a bigraph is *name-discrete* iff it is discrete and every bound link is either an edge, or (if it is an outer name) has exactly one point. Note that name-discrete implies discrete.

A bigraph is *regular* if, for all roots $r_{i'}$ and $r_{j'}$, and all sites s_i and s_j where s_i is a descendant of $r_{i'}$ and s_j of $r_{j'}$, if $i \leq j$ then $i' \leq j'$. The bigraphs in the figures are all regular, the permutation in Table 1 is not. The virtue of regular bigraphs is that certain permutations can be avoided when composing them from basic bigraphs.

2.6 Tensor Product, Parallel Product, and Composition

For bigraphs G_1 and G_2 that share no names or inner names, we can make the *tensor product* $G_1 \otimes G_2$ by juxtaposing their place graphs, constructing the union of their link graphs, and increasing the indexes of sites in G_2 by the number of sites of G_1. For instance, bigraph d of Figure 1 is a tensor product of four primes. We write $\bigotimes_i^n G_i$ for the iterated tensor $G_0 \otimes \cdots \otimes G_{n-1}$.

The *parallel product* $G_1 \,\|\, G_2$ is like the tensor product, except global names can be shared: if y is shared, all points of y in G_1 and G_2 become the points of y in $G_1 \,\|\, G_2$.

We can *compose* bigraphs $G_2 : I \to I'$ and $G_1 : I' \to J$, yielding bigraph $G_1 \circ G_2 : I \to J$, by "plugging in" the roots of G_2 into the sites of G_1, eliminating both, and connecting names of G_2 with inner names of G_1—as in Figure 1, where $a = C \circ (\mathsf{id}_z \otimes R) \circ d$. In the following, we will omit the '$\circ$', and simply write $G_1 G_2$ for composition, letting it bind tighter than tensor product.

2.7 Active, Passive and Atomic Controls

In addition to arity, each control is assigned a *kind*, either atomic, active or passive, and describe nodes according to their control kinds. We require that atomic nodes contain no nodes except sites; any site being a descendant of a passive node is *passive*, otherwise it is *active*. If all sites of a bigraph G are active, G is *active*.

For Figure 1 we could have $\mathsf{Data} : \mathsf{atomic}(0 \to 0)$, $\mathsf{Folder} : \mathsf{passive}(0 \to 1)$, $\mathsf{Laptop} : \mathsf{active}(0 \to 0)$, $\mathsf{Building} : \mathsf{active}(1 \to 1)$.

2.8 Bigraphical Reactive Systems

Bigraphs in themselves model two essential parts of context: locality and connectivity. To model also *dynamics,* we introduce *bigraphical reactive systems* (BRS) as a collection of *rules.* Each rule $R \to^{\varrho} R'$ consists of a regular *redex* $R : I \to J$, a regular *reactum* $R' : I' \to J$, and an *instantiation* ϱ, mapping each site of R' to a site of R. Interfaces $I = \langle m, \vec{X}, X \rangle$ and $I' = \langle m', \vec{X}', X' \rangle$ must be local, and are related by $X'_i = X_{\varrho(i)}$. We illustrate ϱ by a '$i := j$', as shown in Figure 1, whenever $\varrho(i) = j \neq i$. Given an instantiation ϱ and a discrete bigraph $d = d_0 \otimes \cdots \otimes d_k$ with prime d_i's, we let $\varrho(d) = d_{\varrho(0)} \otimes \cdots \otimes d_{\varrho(k)}$, i.e., by copying, discarding and reordering parts of d.

Given an agent a, a *match* of redex R is a decomposition $a = C(\mathsf{id}_Z \otimes R)d$, with active context C, discrete parameter d, and some set of names Z. Dynamics is achieved by transforming a into a new agent $a' = C(\mathsf{id}_Z \otimes R')d'$, where $d' = \varrho(d)$—an example is shown in Figure 1. This definition of a match is as in [9], except that we here also require R to be regular. This restriction to regular redexes R (and to discrete parameters d) does not limit the set of possible reactions. We restrict attention to regular R's because it simplifies the inductive characterization of matching by allowing us to omit trivial permutations.

2.9 Notation, Basic Bigraphs, and Abstraction

In the sequel, we will use the following notation: $\uplus$ denotes union of sets required to be disjoint; we write $\{\vec{Y}\}$ for $Y_0 \uplus \cdots \uplus Y_{n-1}$ when $\vec{Y} = Y_0, \ldots Y_{n-1}$, and similarly $\{\vec{y}\}$ for $\{y_0, \ldots, y_{n-1}\}$. For interfaces, we write X to mean $\langle 0, [], X \rangle$, $\langle X \rangle$ to mean $\langle 1, [\{\}], X \rangle$ and (X) to mean $\langle 1, [X], X \rangle$.

Any bigraph can be constructed by applying composition, tensor product

and abstraction to a set of basic bigraphs, shown in Table 1 [5]. Given a

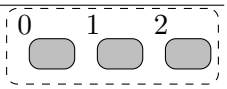
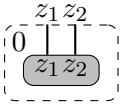
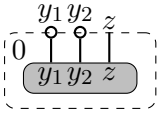
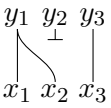
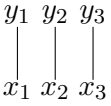
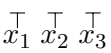
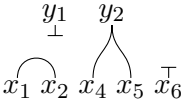
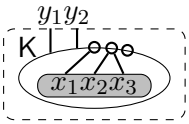
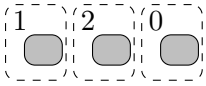
	Notation	*Example*	
Merge	$merge_n : n \to 1$	$merge_3 =$	
Concretion	$\ulcorner X \urcorner : (X) \to \langle X \rangle$	$\ulcorner \{z_1, z_2\} \urcorner =$	
Abstraction	$(Y)P$ $: I \to \langle 1, [Y], Z \uplus Y \rangle$	$(\{y_2\})(\{y_1\})\ulcorner \{y_1, y_2, z\} \urcorner =$	
Substitution σ	$\vec{y}/\vec{X} : X \to Y$	$[y_1, y_2, y_3]/[\{x_1, x_2\}, \{\}, \{x_3\}] =$	
Renaming α, β	$\vec{y}/\vec{x} : X \to Y$	$[y_1, y_2, y_3]/[x_1, x_2, x_3] =$	
Closure	$/X : X \to \{\}$	$/\{x_1, x_2, x_3\} =$	
Wiring ω	$(/Z \otimes \alpha)\sigma$ $: X \to Y$	$(/\{z_2, z_4\} \otimes [y_1, y_2]/[z_1, z_3])$ $[z_1, z_2, z_3, z_4]/$ $[\{\}, \{x_1, x_2\}, \{x_4, x_5\}, \{x_6\}] =$	
Ion	$K_{\vec{y}(\vec{X})}$ $: (\{\vec{X}\}) \to \langle \{\vec{y}\} \rangle$	$\mathsf{K}_{[y_1,y_2]([\{x_1\},\{x_2,x_3\},\{\}])} =$	
Permutation π	$\{i \mapsto j, \ldots\}$ $: m \to m$	$\{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 1\} =$	

Table 1
Basic bigraphs, the abstraction operation, and variables ranging over bigraphs.

prime P, the abstraction operation localises a subset of its outer names. Note that the scope rule is necessarily respected since the inner face of a prime P is required to be local, so all points of P are located within its root. The abstraction operator is denoted by $(\cdot)\cdot$ that reaches as far right as possible.

For a renaming $\alpha : X \to Y$, we write $\ulcorner \alpha \urcorner$ to mean $(\mathsf{id}_1 \otimes \alpha)\ulcorner X \urcorner$, and when $\sigma : U \to Y$, we let $\widehat{\sigma} = (Y)(\sigma \otimes \mathsf{id}_1)\ulcorner U \urcorner$.

As an example, the bigraph of Figure 2 can be written

$$G = (\omega \otimes ((\{y_0\})y_0/Y\ulcorner Y \urcorner) \otimes \ulcorner \{\} \urcorner)\,(((Y)P_1) \otimes P_2 \otimes y_2/x_1)\,, \text{ where}$$

$$\omega = (/e \otimes \mathsf{id}_{\{y_1,y_2\}})[y_1, y_2, e]/[\{y_1\}, \{y_2, y_2', y_2''\}, \{e, e'\}], \quad Y = \{y_0, y_0', y_0''\}$$

$$P_1 = (\mathsf{id}_{\{y_0,y_1,y_2',e\}} \otimes merge_2)\left((\mathsf{id}_{\{y_0,e\}} \otimes K_{0[y_0']})K_{1[y_0,e]} \otimes K_{2[y_0'',y_1,y_2']}\, merge_0\right)$$

$$P_2 = (\mathsf{id}_{\{e',y_2''\}} \otimes merge_2)(K_{3[e',y_2'']([\{x_0,x_2\}])} \otimes \ulcorner \{\} \urcorner),$$

and for Figure 1 we have $a = (\mathsf{id}_{\{consultancy,corporation\}} \otimes /z)\,(p_1 \parallel p_2)$, where

$$p_1 = (\mathsf{id}_z \otimes \mathsf{Building}_{[consultancy]([\{\}])}\mathsf{Laptop})\mathsf{Folder}_{[z]}\mathsf{Data}\; merge_0$$

$$p_2 = (\mathsf{id}_z \otimes \mathsf{Building}_{[corporation]([\{y_1,y_2\}])})(\{y_1,y_2\})(\mathsf{id}_{\{z,y_1,y_2\}} \otimes merge_2)\,(p_2' \otimes p_2'')$$

$$p_2' = (\mathsf{id}_{\{z,y_1\}} \otimes \mathsf{Laptop}\; merge_2)(\mathsf{Folder}_{[z]}\mathsf{Data} merge_0 \otimes \mathsf{Folder}_{[y_1]}\mathsf{Data}\; merge_0)$$

$$p_2'' = (\mathsf{id}_{y_2} \otimes \mathsf{Laptop})\mathsf{Folder}_{[y_2]}\mathsf{Data}\; merge_0$$

Finally, a *molecule* is a prime with just one outermost node.

3 Inductive Characterization of Matching

In this section we present our inductive characterization of matching. To ease the presentation we shall disregard the requirement that the context in a match must be active (it is straightforward to extend the following presentation to include the active requirement).

3.1 Preliminaries

In this subsection we introduce useful notation and establish some propositions about how one may decompose bigraphs. To simplify notation we shall simply write id for identity bigraphs, without a subscript showing the interface, when it is clear from the context what interface is intended.

The following propositions express how bigraphs may be decomposed into simpler constitutent components. The proofs follow easily from the normal form theorem in [5]. Note that ω, α, σ and π range over wirings, renamings, substitutions and permutations, cf. Table 1.

Proposition 3.1 *Any bigraph G can be decomposed into a composition of the following form*

$$G = (\omega \otimes \mathsf{id})(D \otimes \mathsf{id}_Y),$$

where D is discrete and with local innerface. Any other decomposition of G on this form takes the form $G = (\omega' \otimes \mathsf{id})(D' \otimes \mathsf{id}_Y)$, where $\omega' = \omega(\alpha \otimes \mathsf{id}_Y)$ and $D' = (\alpha^{-1} \otimes \mathsf{id})D$, for suitable α.

Proposition 3.2 *Any discrete bigraph D of width n with local innerface can be decomposed such that*

$$D = \Big(\bigotimes_i^n (\widehat{\sigma_i} \otimes \mathsf{id})P_i\Big)\pi,$$

where the P_i's are name-discrete and prime. Any other decomposition on this form of D takes the form $\big(\bigotimes_i^n (\widehat{\sigma_i'} \otimes \mathsf{id})P_i'\big)\pi'$, where, for some $\widehat{\alpha_i}$, ρ_i, for all i, $P_i' = (\widehat{\alpha_i}^{-1} \otimes \mathsf{id})P_i\rho_i$ $(\bigotimes_i^n \rho_i)\pi' = \pi$, and $\widehat{\sigma_i'} = \widehat{\sigma_i}\widehat{\alpha_i}$.

For primes and molecules, the normal form can be found in *loc. cit.*

One can decompose binding ions $K_{\vec{y}(\vec{X})}$ into $K_{\vec{y}(\vec{u})} \bigotimes_i^n (u_i)/(X_i)$. Such decompositions will be useful because of the following proposition, which is a corollary of Theorem 1, item 1, in [5] (specialized to free discrete ions).

Proposition 3.3 *Any free discrete molecule* $M : I \to (\{\vec{y}\} \uplus Z)$ *can be decomposed as*

$$M = (K_{\vec{y}(\vec{u})} \otimes \mathsf{id}_Z)P,$$

where P *is a discrete prime. Any other decomposition of* M *on this form, has the form* $(K_{\vec{y}(\vec{x})} \otimes \mathsf{id}_Z)P'$*, where there exists a* unique $\widehat{\alpha}$*, given by* $u_i \mapsto x_i$*, such that* $K_{\vec{y}(\vec{u})}\widehat{\alpha} = K_{\vec{y}(\vec{x})}$ *and* $P = (\widehat{\alpha} \otimes \mathsf{id}_Z)P'$*.*

3.2 Matching Sentences

We now define matching sentences and rules for deriving valid matching sentences.

Definition 3.4 A *matching sentence* is a 7-place relation among wirings and bigraphs, written $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, R \hookrightarrow C, d$, satisfying that $\omega_{\mathbf{a}}$, $\omega_{\mathbf{R}}$, $\omega_{\mathbf{C}}$ are wirings, and a, R, C, d are discrete bigraphs, R and C have local inner faces, and R is regular.

Definition 3.5 A matching sentence $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, R \hookrightarrow C, d$, where $\omega_{\mathbf{R}} : U \to Y$, C has global outer names V, and d has global outer names Z, is *valid*, denoted $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vDash a, R \hookrightarrow C, d$, iff

$$(\mathsf{id} \otimes \omega_{\mathbf{a}})a = (\mathsf{id} \otimes \omega_{\mathbf{C}})(C \otimes \mathsf{id}_Y \otimes \mathsf{id}_Z)(\omega_{\mathbf{R}} \otimes \mathsf{id})(R \otimes \mathsf{id}_Z)d.$$

Note that for a valid sentence $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, R \hookrightarrow C, d$, if we let $a' = (\mathsf{id} \otimes \omega_{\mathbf{a}})a$, $C' = (\mathsf{id} \otimes \omega_{\mathbf{C}})(C \otimes \mathsf{id}_Y \otimes \mathsf{id}_Z)$, and $R' = (\omega_{\mathbf{R}} \otimes \mathsf{id})R$, then $a' = C'(R' \otimes \mathsf{id}_Z)d$. Conversely, if, for general a', C', R', d we have a match $a' = C'(R' \otimes \mathsf{id}_Z)d$, then by Proposition 3.1, we can decompose a', C', and R' and obtain a corresponding valid sentence. Thus valid sentences precisely capture the abstract definition of matching.

3.3 Rules for Matching

In Figure 3 we present a set of rules for inferring matching sentences. In the premises of the rules PERM and ION, and in the conclusion of rules MERGE, ION, and SWITCH we require the id's to have width 0 (hence be link graph identities). This determines them entirely from the context.

We now explain the rules.

The PERM rule simply pushes a permutation on the inside of the context through the redex, permuting the discrete primes, and producing a pushed-through permutation $\overline{\pi}$, depending on π and the innerface of the redex, as stated in the Push-Through Lemma [5].

$$\textsc{perm}\ \frac{\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, \bigotimes_i^m P_{\pi(i)} \hookrightarrow C, (\overline{\pi} \otimes \mathsf{id})d}{\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, \bigotimes_i^m P_i \hookrightarrow C\pi, d}$$

$$\textsc{par}\ \frac{\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \parallel \omega \vdash a, R \hookrightarrow C, d \qquad \omega_{\mathbf{b}}, \omega_{\mathbf{S}}, \omega_{\mathbf{D}} \parallel \omega \vdash b, S \hookrightarrow D, e}{\omega_{\mathbf{a}} \parallel \omega_{\mathbf{b}}, \omega_{\mathbf{R}} \parallel \omega_{\mathbf{S}}, \omega_{\mathbf{C}} \parallel \omega_{\mathbf{D}} \parallel \omega \vdash a \otimes b, R \otimes S \hookrightarrow C \otimes D, d \otimes e}$$

$$\textsc{lsub}\ \frac{\sigma_{\mathbf{a}} \otimes \omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \sigma_{\mathbf{C}} \otimes \omega_{\mathbf{C}} \vdash p, R \hookrightarrow P, d \qquad \sigma_{\mathbf{a}} : Z \to W \qquad \sigma_{\mathbf{C}} : U \to W}{\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash (\widehat{\sigma_{\mathbf{a}}} \otimes \mathsf{id})(Z)p, R \hookrightarrow (\widehat{\sigma_{\mathbf{C}}} \otimes \mathsf{id})(U)P, d}$$

$$\textsc{merge}\ \frac{\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, R \hookrightarrow C, d}{\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash (\mathit{merge} \otimes \mathsf{id})a, R \hookrightarrow (\mathit{merge} \otimes \mathsf{id})C, d}$$

$$\textsc{ion}\ \frac{\begin{array}{c}\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash (\bigotimes_i^n (v_i)/(X_i) \otimes \mathsf{id})p, R \hookrightarrow (\bigotimes_i^n (v_i)/(Z_i) \otimes \mathsf{id})P, d \\ \alpha = \vec{y}/\vec{u} \quad \sigma_{\mathbf{y}} : \{\vec{y}\} \to \end{array}}{\sigma_{\mathbf{y}} \parallel \omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \sigma_{\mathbf{y}}\alpha \parallel \omega_{\mathbf{C}} \vdash (K_{\vec{y}(\vec{X})} \otimes \mathsf{id})p, R \hookrightarrow (K_{\vec{u}(\vec{Z})} \otimes \mathsf{id})P, d}$$

$$\textsc{switch}\ \frac{\omega_{\mathbf{a}}, \mathsf{id}_\epsilon, \omega_{\mathbf{C}}(\sigma_{\mathbf{R}} \otimes \omega_{\mathbf{R}} \otimes \mathsf{id}) \vdash p, \mathsf{id} \hookrightarrow P, d \qquad \sigma_{\mathbf{R}} : W \to U}{\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash p, (\widehat{\sigma_{\mathbf{R}}} \otimes \mathsf{id})(W)P \hookrightarrow \ulcorner U \urcorner, d}$$

$$\textsc{prime-axiom}\ \frac{}{\omega, \mathsf{id}_\epsilon, \omega(\alpha^{-1} \otimes \mathsf{id}) \vdash p, \mathsf{id} \hookrightarrow \ulcorner \alpha \urcorner, p}$$

$$\textsc{wiring-axiom}\ \frac{}{y, X_{\mathbf{R}}/\emptyset, y/(X_{\mathbf{R}} \uplus X_{\mathbf{d}}) \vdash \mathsf{id}_\epsilon, \mathsf{id}_\epsilon \hookrightarrow \mathsf{id}_\epsilon, X_{\mathbf{d}}/\emptyset}$$

$$\textsc{close}\ \frac{\begin{array}{c}\{\vec{y_i}\} = Y \qquad \{\vec{Z_i}\} = Z \qquad \sigma^{\mathbf{Z}}_{\mathbf{R}} : U \to Z \\ (\bigotimes_i^m y_i/X_i) \otimes (\bigotimes_i^l z_i/X_{i+m}) \otimes \sigma_{\mathbf{a}}, \\ (\bigotimes_i^m y_i/Y_i) \otimes \mathsf{id}_U \otimes \sigma^{\mathbf{C}}_{\mathbf{R}}, (\bigotimes_i^l z_i/Z_i)\sigma^{\mathbf{Z}}_{\mathbf{R}} \otimes \sigma_{\mathbf{C}} \otimes \mathsf{id}_Y \vdash a, R \hookrightarrow C, d \end{array}}{\begin{array}{l}(\bigotimes_i^m /y_i\, y_i/X_i) \otimes (\bigotimes_i^l /z_i\, z_i/X_{i+m}) \otimes \sigma_{\mathbf{a}}, \\ (\bigotimes_i^m /y_i\, y_i/Y_i) \otimes \sigma^{\mathbf{Z}}_{\mathbf{R}} \otimes \sigma^{\mathbf{C}}_{\mathbf{R}}, (\bigotimes_i^l /z_i\, z_i/Z_i) \otimes \sigma_{\mathbf{C}} \vdash a, R \hookrightarrow C, d \end{array}}$$

Fig. 3. Rules for matching binding bigraphs

The PAR rule explains how to match a product, given two valid matches, which *share* some context wiring ω if the two parts of the redex share a (necessarily global) name, cf. Figure 4.

The LSUB rule allows us to match any discrete prime (c.f. Proposition 3.2) by matching an underlying *free* (name)discrete prime with the wiring of agent and context extended with the underlying global substitutions $\sigma_{\mathbf{a}}$ and $\sigma_{\mathbf{C}}$. In

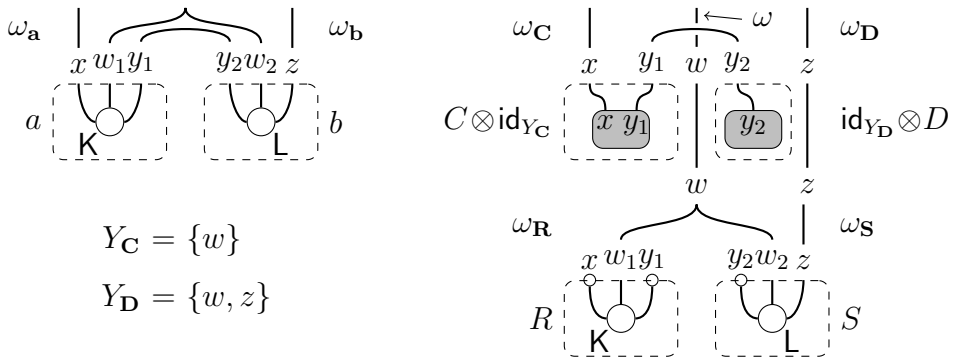


Fig. 4. Matching a product using the PAR rule

other words, this rule expresses that we can match a bigraph with *local* names by matching the corresponding free bigraph (forgetting that the names are local) and then remember to make the correct names local again.

The MERGE rule simply states that to match bigraphs with an outer merge and a global id, we must be able to match the underlying bigraphs.

The ION rule works intuitively by splitting up a binding ion into a free, discrete ion and an underlying local substitution. For any given match of discrete primes, we can compose with ions $K_{\vec{y}(\vec{X})}$ or $K_{\vec{n}(\vec{Z})}$, if we extend the wirings of agents and contexts with isomorphic wiring on the outer names $\vec{y}$ and $\vec{n}$; stated in the rule by requiring that we extend with $\sigma_{\mathbf{y}}$ and $\sigma_{\mathbf{y}}\alpha$ (where $\alpha = \vec{y}/\vec{n}$). For example, if we seek to match the agent $a = (\mathsf{id} \otimes K_{\vec{y}(\vec{X})})p$ yielding a context $C = (\mathsf{id} \otimes K_{\vec{u}(\vec{Z})})P$, then it suffices to consider matching of $a' = (\vec{v})/(\vec{X})p$ yielding a context $C' = (\vec{v})/(\vec{Z})$, as illustrated in Figure 5.

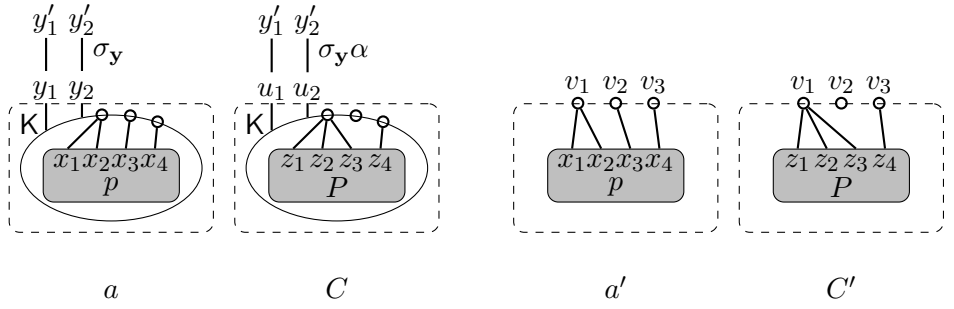


Fig. 5. Matching ion agent a yielding context C by matching a' yielding context C'

Given an agent and considering an inference tree operationally bottom up, the rules specify how to decompose the agent while *constructing* the corresponding context (cf. e.g. the ION rule). At the point where the root of the redex is matched, the SWITCH rule is applied, switching the redex into context position, so that further decomposition of the agent *checks* that the redex matches. Thus, when inferring a match, every rule except SWITCH can be used in two modes: one where the agent and redex are given, resulting in a context and parameter; and one where the agent and context are given, resulting in a parameter.

The PRIME-AXIOM and WIRING-AXIOM axioms are our base cases and are intuitively clear (the latter is used to match bigraphs of zero width).

The CLOSE rule allows us to infer a match for *open* bigraphs and "close" this match by replacing names in wirings with edges, taking care to split multi-closures appropriately. For example, the agent a in Figure 6 is matched by matching agent a' and then closing the names y, z_1 and z_2. So internal agent

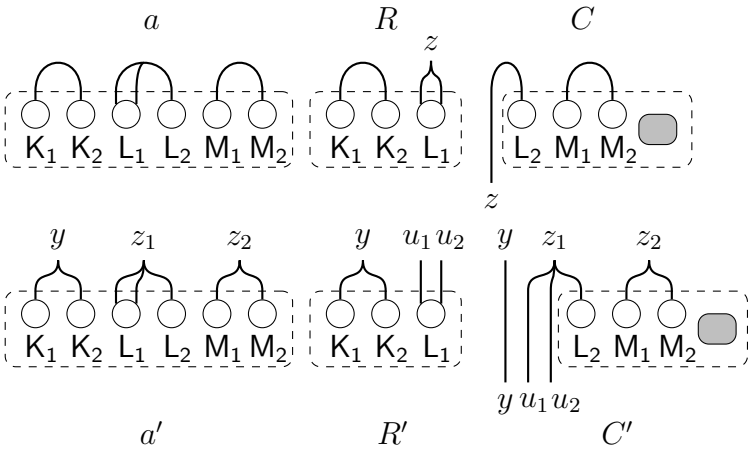


Fig. 6. Matching closed links within and between redex and context

edges matched by internal redex edges are named y_i, and edges matched by internal context edges are named z_i.

Theorem 3.6 *The rules for matching in Figure 3 are sound, i.e., any matching sentence that can be derived is valid.*

Proof. Straightforward, but tedious, standard algebraic manipulations. □

The completeness theorem will be proved by induction on the size of valid sentences, which is defined as follows.

Definition 3.7 The size of a matching sentence $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, R \hookrightarrow C, d$ is the number of ions in a.

The following lemmas express how a valid sentence may be derived by applications of inference rules to valid sentences of lesser or equal size. The proofs proceed by first decomposing the components of the given valid sentence, then defining the components of the valid sentence(s) claimed to exist and, finally, verifying that (1) the sentences claimed to exist really are valid and (2) that the given sentence can indeed be derived as claimed. The decompositions are obtained via Propositions 3.1, 3.2, and 3.3, and the verifications proceed using lemmas found in [5] (in particular, the "push-through-lemma," which expresses how we can push a permutation "through" a product of primes, permuting the order in which they appear in the product, and producing a permutation that reorders the sites in the primes to preserve the inner face).

Lemma 3.8 *Every valid sentence* $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vDash a, R \hookrightarrow C, d$ *is provable using the* CLOSE *and the* PERM *rule on a valid sentence, of equal size, of the form* $\sigma'_{\mathbf{a}}, \sigma'_{\mathbf{R}}, \sigma'_{\mathbf{C}} \vDash a, S \hookrightarrow \bigotimes_i^n P_i, e$.

Lemma 3.9 *Every valid sentence* $\sigma_{\mathbf{a}}, \sigma_{\mathbf{R}}, \sigma_{\mathbf{C}} \vDash a, R \hookrightarrow Q \otimes \bigotimes_i^n P_i, d$, *with* P *and* P_i *prime and discrete, is provable using the* PAR *rule on valid sentences, of lesser or equal size, of the form* $\sigma^P_{\mathbf{a}}, \sigma^P_{\mathbf{R}}, \sigma^P_{\mathbf{C}} \parallel \sigma^S_{\mathbf{C}} \vDash p, S \hookrightarrow P, e$ *and* $\sigma^C_{\mathbf{a}}, \sigma^C_{\mathbf{R}}, \sigma^C_{\mathbf{C}} \parallel \sigma^S_{\mathbf{C}} \vDash a', R' \hookrightarrow \bigotimes_i^n P_i, e'$.

Lemma 3.10 *Every valid sentence* $\sigma_{\mathbf{a}}, \sigma_{\mathbf{R}}, \sigma_{\mathbf{C}} \vDash a, R \hookrightarrow \mathsf{id}_\epsilon, d$ *is provable using the* PAR *and* WIRING-AXIOM.

Lemma 3.11 *Every valid sentence* $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vDash p, R \hookrightarrow P, d$, *with* p *and* P *prime and discrete, is provable using the* LSUB *rule on a valid sentence, of lesser or equal size, of the form* $\omega'_a, \omega'_R, \omega'_C \vDash p', R \hookrightarrow P', d$, *where* p' *and* P' *are discrete free primes.*

Lemma 3.12 *Every valid sentence* $\sigma_{\mathbf{a}}, \sigma_{\mathbf{R}}, \sigma_{\mathbf{C}} \vDash p, R \hookrightarrow P, d$, *with* p *and* P *discrete and free primes, is provable using* MERGE, PAR *(iterated), and* SWITCH *rules on valid sentences, each of lesser or equal size, and each on one of two forms:*

- $\sigma'_a, \sigma'_R, \sigma'_C \vDash p^N, \mathsf{id} \hookrightarrow P^N, e$, *where* p^n *and* P^N *are free discrete primes,*
- $\sigma'_a, \sigma'_R, \sigma'_C \vDash m, S \hookrightarrow M, e$, *where* m *and* M *are free discrete molecules.*

Lemma 3.13 *Every valid sentence* $\sigma_{\mathbf{a}}, \sigma_{\mathbf{R}}, \sigma_{\mathbf{C}} \vDash m, R \hookrightarrow M, d$, *with* m *and* M *free discrete molecules, is provable using the* ION *rule on a valid sentence* $\sigma'_a, \sigma'_R, \sigma'_C \vDash p, R \hookrightarrow P, d$, *of lesser size, where* p *and* P *are discrete primes.*

Lemma 3.14 *Every valid sentence* $\sigma_{\mathbf{a}}, \sigma_{\mathbf{R}}, \sigma_{\mathbf{C}} \vDash p, \mathsf{id} \hookrightarrow P, e$, *with* p *and* P *free discrete primes, is provable using the* MERGE *and* PAR *(iterated) rules on valid sentences of equal or lesser size, which are either instances of rule* PRIME-AXIOM *or of the form* $\sigma'_a, \sigma'_r, \sigma'_M \vDash m, R \hookrightarrow M, d$.

Theorem 3.15 *The rules for matching in Figure 3 are complete, i.e., any valid matching sentence can be derived from the rules.*

Proof. By induction on the size of a sentence. By the lemmas above, we have that all valid sentences with size n can be derived from valid sentences of the form $\sigma_{\mathbf{a}}, \sigma_{\mathbf{R}}, \sigma_{\mathbf{C}} \vDash m, R \hookrightarrow M, d$, with m and M free discrete molecules, of size less than or equal to n. By Lemma 3.13, these can be derived from sentences of size less than n. □

4 Towards Algorithms for Matching

The completeness theorem tells us that we can find all valid matching sentences by applications of the rules for matching. Thus the rules for matching define an algorithm for matching, for instance easily expressed in Prolog, which simply operates by searching for inference trees using the rules.

Although we can (e.g. in prolog) base a matching algorithm directly upon the matching rules, we do not claim that an efficient matching algorithm has to be so based. We have introduced matching rules for a dual purpose: first, to characterise matching structurally and inductively in order to understand it (in particular to understand the relation to representations based on normal forms and to understand where exactly choices between different matches can be made during matching); second, to provide a point from which to begin the search for truly efficient matching algorithms, and to verify them. This rigorous approach to matching is justified, in our view, because matching will be the workhorse of any implementation of bigraph dynamics.

In practice, one is, of course, interested in minimizing unnecessary blind search, and thus, e.g., only search for inference trees of a certain form. Indeed, one can show that it suffices to consider so-called *normal inference trees*, which put restrictions on the order in which the inference rules are applied (such as, e.g., always concluding with the CLOSE rule). We shall not include a formal definition of normal inference trees here, but rather discuss some of the possibilities for defining normal inference trees. We first remark that to retain completeness, any definition of normal inference must, of course, ensure no loss of provability. Looking at the formulations of the lemmas leading up to the completeness theorem, we see that there are indeed several possibilities for the definition of normal inference tree. For example, from Lemma 3.8 we see that we are free to conclude each inference tree with CLOSE and then PERM or vice versa. Further, in several rules we are allowed to propagate closed links, even though CLOSE intuitively makes that unnecessary. We have chosen to leave this freedom in the rule system and instead comment on how we could *extend* the set of rules to allow even more freedom in chosing our definition of normal inference tree. This is important when thinking about implementations, as each definition of normal inference tree corresponds to a different algorithmic approach to matching.

One may say that the current set of rules naturally give rise to normal inferences that are a mix between matching the link graph "lazily"or "eagerly". Instead of the CLOSE rule, one could have amended the PAR and ION rules (those with $\|$ in the conclusion) such that they would also handle matching of closures. This would have allowed true "by need" link-matching. Conversely, one could have amended the CLOSE to also compare substitutions, allowing us to consider matching of discrete bigraphs up to renaming isos on their outerfaces. If we amended the LSUB and SWITCH rules to work accordingly, this would actually preclude the need for the wirings $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}}$ in matching sentences. It seems, though, that the tedious complexity added into these rules would mean that we would gain little in removing complexity from the rules as a whole. Anyhow, these changes would allow us to define a variant of normal inferences, which would be "strict" in the link graph, in that we would immediately be able to reject possible matches based on the link graph (instead of the place graph).

Another possibility would be to add a rule GLOB, allowing us to match all wiring stemming from a *single* prime as global wiring. This idea seems to indicate that matching in *local* bigraphs [13] (where there is no global linkage but instead multilocated names) could be handled similarly, by recasting the rules to work on local links and just locating names at all roots where they occur.

4.1 Representations of Graphs

An implementation of matching must, of course, represent bigraphs in some way. One possibility is to represent bigraphs directly by place and link graphs, and then implement the normal form lemmas, which express how bigraphs may be decomposed into simpler bigraphs; then matching can proceed by induction on the decomposed graph. In general, however, the "decomposition functions" return *sets* of possible decompositions, because normal forms are only unique up to certain permutations. (For example, $merge(M_1 \otimes M_2) = merge(M_2 \otimes M_1)$.) A matching implementation needs to explore all the possible decompositions. This can be made explicit formally, by phrasing the inductive characterization of matching not on bigraphs but on bigraphical *expressions* (syntax), as defined in [12,5]. Doing so forces us to add an inference rule, which allows one to replace any expression in a matching sentence $\omega_{\mathbf{a}}, \omega_{\mathbf{R}}, \omega_{\mathbf{C}} \vdash a, R \hookrightarrow C, d$, say a, by another, say a', that is provably equal via the axioms for equality in [5]. Doing so clearly yields a complete set of rules on bigraphical expressions. When defining normal inference trees for these, one seeks, of course, to restrict the application of the equality axioms. The definition of normal inference trees will then *formally explicate* all the possibilities that a matching algorithm needs to explore. We have worked out a definition of normal inference tree for matching of place graph *expressions* and proved it complete. Based on that experience, we believe it should not be too hard to work out a suitable definition of normal inference tree binding bigraph expressions and prove it complete.

5 Conclusion and Related Work

We have presented a sound and complete inductive characterization of matching for binding bigraphs. We are currently working toward an implementation of matching based upon the characterization.

Bigraphical reactive systems are related to graph transformations systems; see [6] for a recent comprehensive overview of graph transformation systems. In particular, bigraph matching is strongly related to the general graph pattern matching (GPM) problem, so general GPM algorithms might be applicable [17,7,10,20]. Due to the special structure of bigraphs, general GPM algorithms are expected to be inefficient, although some GPM tools [19] use heuristic search strategies that might be able to discover and exploit bigraph

structure. A special aspect of bigraphs is that we may match a set of subtrees with a single node (site) in the redex, and match multiple redex roots in different places within the agent. Fu [7] handles such wildcard nodes and multiple patterns, but directly applying his algorithm is not straightforward, as he attacks the problem of tree isomorphism of rooted graphs unfolded to finite unbounded depths. The subtree isomorphism problem [15,18,16] is simpler than GPM, but applying it directly to the place graphs of bigraphs would not exploit the constraints imposed by the link graphs. Rather, efficient implementations of bigraph matching should be derived from the initial implementation by experimenting with different normal inference tree definitions, and combining it with subtree isomorphism algorithms. The inductive characterization provided here will make it easier to prove the actual algorithm correct.

6 Acknowledgments

This work was funded in part by the Danish Research Agency (grant no.: 2059-03-0031) and the IT University of Copenhagen (the LaCoMoCo project).

References

[1] Birkedal, L., *Bigraphical Programming Languages—a LaCoMoCo research project*, in: *Second UK UbiNet Workshop, Cambridge*, 2004, position Paper.

[2] Birkedal, L., T. C. Damgaard, A. J. Glenstrup and R. Milner, *Matching of bigraphs — proofs of soundness and completeness*, available on request.

[3] Birkedal, L., S. Debois, E. Elsborg, T. Hildebrandt and H. Niss, *Bigraphical models of context-aware systems*, in: L. Aceto and A. Ingólfsdóttir, editors, *FOSSACS '06: Proceedings of 9th International Conference on Foundations of Software Science and Computation Structures*, LNCS **3921** (2006), pp. 187–201.

[4] Birkedal, L., S. Debois and T. Hildebrandt, *Sortings for reactive systems*, Technical Report 84, IT University of Copenhagen (2006), iSBN 87-7949-124-3.

[5] Damgaard, T. C. and L. Birkedal, *Axiomatizing binding bigraphs (revised)*, Technical Report TR-2005-71, IT University of Copenhagen (2005).

[6] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation," Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2006.

[7] Fu, J. J., *Directed graph pattern matching and topological embedding*, Journal of Algorithms **22** (1997), pp. 372–391.

[8] Jensen, O. H., "Mobile Processes in Bigraphs," Ph.D. thesis, Univ. of Cambridge (2005), forthcoming.

[9] Jensen, O. H. and R. Milner, *Bigraphs and mobile processes (revised)*, Technical Report 580, University of Cambridge (2004), iSSN 1476-2986.

[10] Larrosa, J. and G. Valiente, *Constraint satisfaction algorithms for graph pattern matching*, Mathematical Structures in Computer Science **12** (2002), pp. 403–422.

[11] Leifer, J. J. and R. Milner, *Transition systems, link graphs and Petri nets*, Technical Report 598, University of Cambridge (2004).

[12] Milner, R., *Axioms for bigraphical structure*, Technical Report 581, University of Cambridge (2004).

[13] Milner, R., *Bigraphs whose names have multiple locality*, Technical Report UCAM-CL-TR-603, University of Cambridge, Computer Laboratory (2004).
URL http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-603.pdf

[14] Milner, R., *Pure bigraphs*, Technical Report 614, University of Cambridge (2005).

[15] Selkow, S. M., *The tree-to-tree editing problem*, Information Processing Letters **6** (1977), pp. 184–186.

[16] Shamir, R. and D. Tsur, *Faster subtree isomorphism*, Journal of Algorithms **33** (1999), pp. 267–280.

[17] Ullman, J. D., *An algorithm for subgraph isomorphism*, Journal of the ACM **23** (1976), pp. 31–42.

[18] Valiente, G., "Algorithms on Trees and Graphs," Springer, Berlin, 2002.

[19] Varró, G., D. Varró and K. Friedl, *Adaptive graph pattern matching for model transformations using model-sensitive search plans*, in: G. Karsai and G. Taentzer, editors, *GraMot 2005, International Workshop on Graph and Model Transformations*, Electronic Notes in Theoretical Computer Science, 2005, pp. 191–205.
URL http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/gramot05_vvf.pdf

[20] Zündorf, A., *Graph pattern matching in PROGRES*, in: J. Cuny, H. Ehrig, G. Engels and G. Rozenberg, editors, *Proceedings of the 5th International Workshop on Graph-Grammars and their Application to Computer Science*, LNCS **1073** (1996), pp. 454–468.

Adhesive DPO Parallelism for monic matches

Filippo Bonchi, Tobias Heindel

Dipartimento di Informatica, Università di Pisa, Italy

Abstract

This paper presents indispensable technical results of a general theory that will allow to systematically derive from a given reduction system a behavioral congruence that respects concurrency. The theory is developed in the setting of adhesive categories and is based on the work by Ehrig and König on borrowed contexts; the latter are an instance of relative pushouts, which have been proposed by Leifer and Milner.

In order to lift the concurrency theory of DPO rewriting to borrowed contexts we will study the special case of DPO rewriting with monic matches in adhesive categories: more specifically we provide a generalized Butterfly Lemma together with a Local Church Rosser and Parallelism theorem.

1 Introduction and Motivation

Process calculi are a well established tool to describe interactive systems. The progression of a process, if it is interpreted as a *closed system*, is described by a *reduction system* (RS); moreover each process is a state of a *labeled transition system* (LTS), which describes how the process may interact with its environment: in this case the process is thought of as an *open system*. Also the the double pushout approach (DPO) can be used to model closed and open systems: a reduction step corresponds to a DPO *rewrite* while interaction with the environment is described as a transition that is labeled by a *borrowed context*, which is a part of the environment. One of the advantages of the DPO approach is that one can distinguish between concurrent and necessarily interleaved events of a closed system. Now the main motivation of this paper to lift this advantage to the setting of open systems, i.e. to provide LTSs with labels that describe *concurrent interaction* with the environment.

One of the first approaches to derive a LTS from a given RS, was presented in [12]. The transitions of the generated LTS are labeled by the "minimal"

[1] This work is partially supported by the SegraVis research training network.

contexts that allow a reduction (as a consequence all the internal actions of a system correspond to transitions which have the "empty" environment as label). For example in CCS, which has the reduction rule $\bar{x}.P \mid x.Q \to P \mid Q$, the process $\bar{a}.0$ cannot perform any reduction by itself but can only be reduced in a context of the form $[-] \mid a.P$: hence the derived LTS contains a transition $\bar{a}.0 \xrightarrow{[-]|a.P} P$. The main property of the derived LTS is that its associated bisimulation relation is a congruence, i.e. it relates two processes that exhibit the same behavior w.r.t. to every environment. However to check bisimilarity one does *not* need to check *all* contexts but it is enough to consider the "minimal" ones, which are given as the labels.

Leifer and Milner's work [12] has been extended to an enriched category context by Sassone and Sobocinski in [13], while Ehrig and König developed a similar framework for DPO rewriting (on **Graphs**) in [6], called *borrowed context rewriting* (DPOBC). Finally [14] introduces an encompassing theory (following the bi-categorical approach of DPO rewriting of [7]). The results of this last most general work apply to every adhesive category. This means that given a system specification by an adhesive rewriting system [11] one can generate a LTS with an associated bisimulation congruence.

Whereas RSs and LTSs are (families of) relations between *states* of a system, the concurrency theory for DPO rewriting is concerned with relations between the *transitions*, i.e. the rewrites (see e.g. [9,1]). For example two consecutive applications of the rule $\circ\circ \leftarrow \circ\circ \rightarrow \backsim$ may result in the graph ∞. The two rewrites are sequential independent, i.e. one can swap them without any further complications; moreover one can even apply them "at the same time", that is *concurrently*: the concurrent application corresponds to a single application of the *parallel rule* $\circ\circ \leftarrow \circ\circ \rightarrow \infty$. In contrast, consider a coffee vending machine: it can sell a coffee and then a latte macchiato or do this in the reversed order but not at the same time (unless you operate a buggy machine which produces a puddle of cappuccino as the result of the concurrent execution). The latter example explains the difference between the two CCS processes $\bar{c} \mid \bar{m}$ and $\bar{c}.\bar{m} + \bar{m}.\bar{c}$, which nevertheless are equivalent according to the standard bisimulation of CCS. Also the generated LTSs discussed before do *not* take into account these finer differences in behavior.

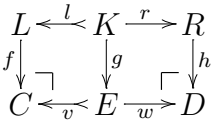
This paper is aimed towards the generation of bisimulation congruences that do respect concurrency. Here we report about the first steps of research in this direction. The main idea is to saturate a given set of productions with *all parallel productions* and then apply the borrowed context method to generate a bisimulation congruence that respects concurrency. More specifically, given an initial set of rules P, we will construct a saturation $\bar{P}$ that will be used to synthesize a LTS using the results of [14]; the set $\bar{P}$ contains for every (finite) subset $P' \subseteq P$ and every way in which the members of P' might be applied concurrently the corresponding parallel production.

One central issue is the apropriate notion of parallel rule. Parallel rules are usually defined as coproducts in DPO; but this construction cannot be used in DPOBC since there, matching morphisms are required to be monic. The required notion of parallel rule is given in [9], which studies DPO rewriting with monic matches (DPO$^{a/i}$), for the case of **Graphs**. However this work cannot be directly adapted to the adhesive setting since the proofs of its results depend on coproducts.

2 Local Church Rosser and Parallelism for DPO$^{a/i}$

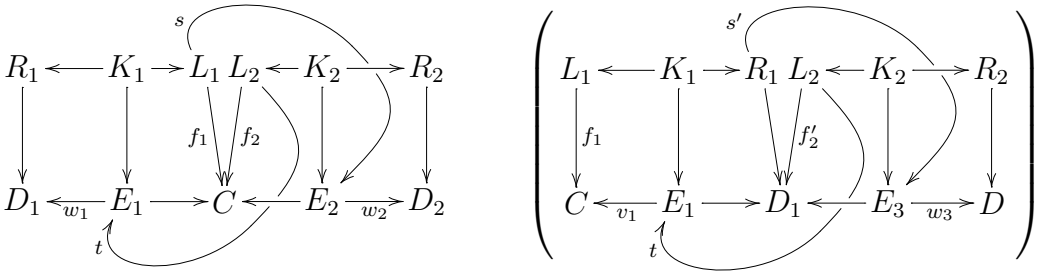
We first recall the essential definitions of DPO rewriting in adhesive categories as presented in [11], to which we refer the reader for more details. For the remainder of this section we fix an adhesive category $\mathbb{C}$, to which all mentioned objects and arrows belong.

Definition 2.1 (Productions and rewriting)
A production p *is a span of arrows* $p = L \xleftarrow{l} K \xrightarrow{r} R$ *with monic* l*. Given an arrow* $f\colon L \to C$ *we say that* p rewrites C to D at match f*, and we write* $C \overset{\langle f,p\rangle}{\Longrightarrow} D$ *if there exists a diagram containing two pushouts as shown on the right.*



In the theory of borrowed contexts in adhesive categories, one only encounters the special case where the matching morphism f is monic, and hence from now on we will assume all matches to be monic. This fragment of DPO rewriting in the category of **Graphs** has been studied in [9] by the name DPO$^{a/i}$. Their results involve the strong versions of sequential and parallel independence.

Definition 2.2 ((Strong) parallel and sequential independence) *Let be given productions* $p_i = L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i$ *for* $i \in \{1,2\}$ *and let there be given the rewrites* $D_1 \overset{\langle f_1,p_1\rangle}{\Longleftarrow} C \overset{\langle f_2,p_2\rangle}{\Longrightarrow} D_2$ *(*$C \overset{\langle f_1,p_1\rangle}{\Longrightarrow} D_1 \overset{\langle f_2',p_2\rangle}{\Longrightarrow} D$*). They are* parallel (sequential) independent*, if there exist morphisms* s *and* t *(*s' *and* t'*) such that they commute in the composed diagram of the rewrites below.*



They are strongly *parallel (sequential) independent if* $w_1 \circ t$ *and* $w_2 \circ s$ *(*$v_1 \circ t'$ *and* $w_3 \circ s'$*) are monic.*

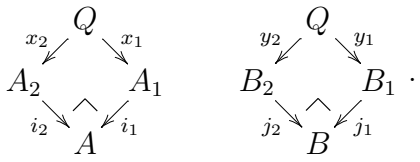
In [9] the Parallelism theorem for DPO$^{a/i}$ for the case of **Graphs** has been proven. However the proof cannot be lifted directly to adhesive categories

since it depends on the existence of coproducts. Moreover the Parallelism theorem for adhesive categories with coproducts presented in [11], does not transfer to DPO$^{a/i}$.

Technical contribution

The main idea is to replace coproducts, which are just pushouts from the empty graph in **Graphs**, by pushouts. This will allow us to make the DPO$^{a/i}$ theory of [9] available for adhesive categories. How coproducts can be replaced by pushouts will be explained in terms of the next definition.

Definition 2.3 *Let the following squares be pushouts:*



Then we will denote A by $A_1 +_Q A_2$ and B by $B_1 +_Q B_2$.

If $y_1 = f_1 \circ x_1$ and $y_2 = f_2 \circ x_2$ holds for two morphisms f_1 and f_2, and $f\colon A \to B$ is the unique morphism which satisfies $f \circ i_1 = j_1 \circ f_1$ and $f \circ i_2 = j_2 \circ f_2$: then f will be denoted by $f_1 +_Q f_2$.[a]

[a] If we have space we can add a diagram.

For the initial object 0 the expression $A_1 +_0 A_2$ is equivalent to $A_1 + A_2$, and similarly for $f_1 +_0 f_2$ and $f_1 + f_2$. This "generalized coproduct" is used to describe the parallel composition of two rules that rewrite an object in a parallel independent way: a combined rule is constructed that allows to apply the two rules "at the same time", i.e. concurrently. More specifically the two rules need to be glued together at the intersection of their read-only parts.

Definition 2.4 (Parallel productions) *Let $p_1 = L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1$ and $p_2 = L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2$ be productions, and let $K_1 \xleftarrow{x_1} Q \xrightarrow{x_2} K_2$ be a span of morphisms. If the pushouts for all the pairs $(l_1 \circ x_1, l_2 \circ x_2), (x_1, x_2)$ and $(r_1 \circ x_1, r_2 \circ x_2)$ exist, then the* parallel composition of p_1 and p_2 over Q *is*

$$p_1 +_Q p_2 = L_1 +_Q L_2 \xleftarrow{l_1 +_Q l_2} K_1 +_Q K_2 \xrightarrow{r_1 +_Q r_2} R_1 +_Q R_2.$$

The production $p_1 +_Q p_2$ is called proper *if all the morphisms of the three involved pushout diagrams are monic.*[2]

Now we are ready to formulate the main theorem, which might be of interest whenever one uses DPO$^{a/i}$ rewriting in adhesive categories. The proof relies on an adapted version of the Butterfly Lemma of [10] for "generalized" coproducts (see Appendix).

[2] This construction is equivalent to the one given in Definition 9.5 of [9].

Theorem 2.5 (Parallelism and Local Church Rosser in DPO$^{a/i}$)
Let $p_1 = L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1$ and $p_2 = L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2$ be productions,[3] *and let $L_1 \overset{f_1}{\rightarrowtail} C$ and $L_2 \overset{f_2}{\rightarrowtail} C$ be morphisms. Then the following are equivalent.*

(i) There are strongly parallel independent rewrites $D_1 \overset{\langle f_1,p_1\rangle}{\Longleftarrow} C \overset{\langle f_2,p_2\rangle}{\Longrightarrow} D_2$.

(ii) There are strongly sequential independent rewrites $C \overset{\langle f_1,p_1\rangle}{\Longrightarrow} D_1 \overset{\langle f_2',p_2\rangle}{\Longrightarrow} D$.

(iii) There are strongly sequential independent rewrites $C \overset{\langle f_2,p_2\rangle}{\Longrightarrow} D_2 \overset{\langle f_1',p_1\rangle}{\Longrightarrow} D$.

(iv) There is a rewrite $C \overset{\langle f_1+_Q f_2,p_1+_Q p_2\rangle}{\Longrightarrow} D$ with a proper parallel production $p_1+_Q p_2$ where Q is constructed as the pullback $Q \overset{K_1\to C}{\underset{K_2}{\nearrow\nearrow}}$, i.e. $Q = K_1 \cap K_2$.

3 Conclusion and work in progress

Motivated by extending the existing concurrency theory of DPO rewriting to the interactive setting of DPO with borrowed contexts (DPOBC), we have defined the required kind of parallel productions and proved the Local Church Rosser and Parallelism theorem for DPO$^{a/i}$ in adhesive categories. Besides filling this gap in the literature, these theorems might prove useful for future research concerned with DPO$^{a/i}$ rewriting in adhesive categories. This is not unlikely since the DPO$^{a/i}$ approach is more intuitive and more expressive than DPO as shown in [9]. In fact, DPOBC is not the only application where the requirement of monic matches arises naturally: consider e.g. the work on processes of adhesive rewriting systems [2] and encondig of nominal calculi [8].

We will use the presented results for the generation of a concurrency respecting bisimulation congruence from a given set of rules. More specifically the construction of parallel rules will be used to generate a closure of all given productions as follows: given a set of productions P we construct the closure $\bar{P}$ via the two rules

$$\frac{p \in P}{p \in \bar{P}} \qquad \frac{p, p' \in \bar{P} \;\&\; K_p \overset{i}{\leftarrowtail} Q \overset{j}{\rightarrowtail} K_{p'}}{p +_Q p' \in \bar{P}}$$

where K_p denotes the interface of a rule p, i.e. given a rule $p = X \leftarrow Y \rightarrow Z$ we write K_p for Y.

Usually in borrowed context rewriting and in the more general setting of the theory of reactive systems, the LTS is derived using the set of rules P, while here we propose to use $\bar{P}$. Reconsider the CCS example from the introduction where we hinted at the difference between the two processes $\bar{c} \,|\, \bar{m}$ and $\bar{c}.\bar{m} + \bar{m}.\bar{c}$. This now can be made formal, since the LTS generated from $\bar{P}$ using the borrowed context technique of [14] allows the former to communicate with the environment concurrently at the channels m and c (this corresponds

[3] These are not required to be linear, as is assumed in [11].

to the transition $\bar{c} \mid \bar{m} \xrightarrow{[-]|c.P|m.Q} P \mid Q$) while the latter cannot (in signs $\bar{c}.\bar{m} + \bar{m}.\bar{c} \xrightarrow{[-]|c.P|m.Q}\!\!\!\!\!\!\not\quad\; P \mid Q$).

There are several other proposals of bisimulations that respect concurrency [4, 3, 5] however they are based on the notion of causality. Our proposal conceptually differs from these since it does not allow the environment to observe causality but just the possible ways in which a system could interact with the environment concurrently. In other words, we consider systems as black boxes, while intuitively the existing equivalences seem to open the black box by observing causal dependencies. Reconsidering our CCS example, our proposed bisimilarity distinguishes $\bar{c} \mid \bar{m}$ and $\bar{c}.\bar{m} + \bar{m}.\bar{c}$ because an external observer can parallely communicate with the former but not with the latter, while the bisimilarities of the cited works distinguish the processes because the former can perform its transitions independently and the latter cannot. The subtle interplay between causality and concurrency especially in the context of borrowed context rewriting is the main interest of ongoing research.

Acknowledgements
We would like to thank Fabio Gadducci, Ugo Montanari and the anonymous referees for their comments on the paper.

References

[1] Baldan, P., A. Corradini, H. Ehrig, M. Löwe, U. Montanari and F. Rossi, *Concurrent semantics of algebraic graph transformations*, in: H. Ehrig, H.-J. Kreowski, U. Montanari and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation* (1999), pp. 107–188.

[2] Baldan, P., A. Corradini, T. Heindel, B. König and P. Sobocinski, *Processes for adhesive rewriting systems.*, in: *FoSSaCS*, 2006, pp. 202–216.

[3] Baldan, P., A. Corradini and U. Montanari, *History preserving bisimulation for contextual nets.*, in: D. Bert, C. Choppy and P. D. Mosses, editors, *WADT*, Lecture Notes in Computer Science **1827** (1999), pp. 291–310.

[4] Bednarczyk, M. A., *Hereditary history preserving bisimulations or what is the power of the future perfect in program logics*, Technical report, Polish Academy of Sciences, Gdansk (1991).

[5] Best, E., R. R. Devillers, A. Kiehn and L. Pomello, *Concurrent bisimulations in petri nets.*, Acta Inf. **28** (1991), pp. 231–264.

[6] Ehrig, H. and B. König, *Deriving bisimulation congruences in the DPO approach to graph rewriting*, in: *Proc. of FOSSACS '04* (2004), pp. 151–166, LNCS 2987.

[7] Gadducci, F., R. Heckel and M. Llabrés, *A bi-categorical axiomatisation of concurrent graph rewriting.*, Electronic Notes Theoretical Computer Science **29** (1999).

[8] Gadducci, F. and U. Montanari, *Observing reductions in nominal calculi via a graphical encoding of processes*, in: *Processes, Terms and Cycles*, 2005, pp. 106–126.

[9] Habel, A., J. Müller and D. Plump, *Double-pushout graph transformation revisited.*, Mathematical Structures in Computer Science **11** (2001), pp. 637–688.

[10] Kreowski, H.-J., "Manipulationen von Graphmanipulationen," Ph.D. thesis, Technische Universität Berlin (1977).

[11] Lack, S. and P. Sobociński, *Adhesive and quasiadhesive categories*, Theoretical Informatics and Applications **39** (2005), pp. 511–546.
URL
http://www.maths.usyd.edu.au:8000/u/stevel/papers/vkjournal.html

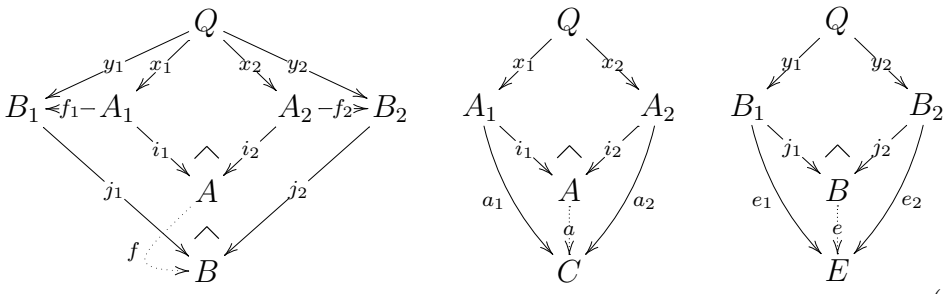
[12] Leifer, J. J. and R. Milner, *Deriving bisimulation congruences for reactive systems.*, in: *CONCUR*, 2000, pp. 243–258.

[13] Sassone, V. and P. Sobociński, *Deriving bisimulation congruences using 2-categories.*, Nord. J. Comput. **10** (2003), pp. 163–.

[14] Sassone, V. and P. Sobociński, *Reactive systems over cospans*, in: *Proc. of LICS '05* (2005), pp. 311–320.

A The extended Butterfly Lemma

Lemma A.1 (General butterfly lemma)



(A.1)

Let the above be commuting diagrams where all interior squares and the boundary of the left one are pushouts, and $f\colon A \to B, a\colon A \to C$, *and* $e\colon B \to E$ *are the unique mediating morphisms, such that*

$$j_1 \circ f_1 = f \circ i_1 \qquad j_2 \circ f_2 = f \circ i_2 \tag{A.2}$$

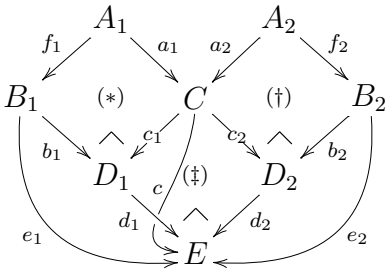
$$a_1 = a \circ i_1 \qquad a_2 = a \circ i_2 \tag{A.3}$$

$$e_1 = e \circ j_1 \qquad e_2 = e \circ j_2 \tag{A.4}$$

Finally let $\mathbb{C}$ *have pushouts of the two diagrams* $B_1 \xleftarrow{f_1} A_1 \xrightarrow{a_1} C$ *and* $B_2 \xleftarrow{f_2} A_2 \xrightarrow{a_2} C$.

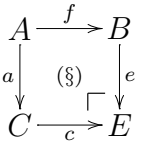
Then for any morphism $c\colon C \to E$ *the following are equivalent.*

(i) There exists a commuting diagram



where the squares $(*), (\dagger)$ *and* $(\ddagger)$ *are pushouts.*

(ii) The diagram

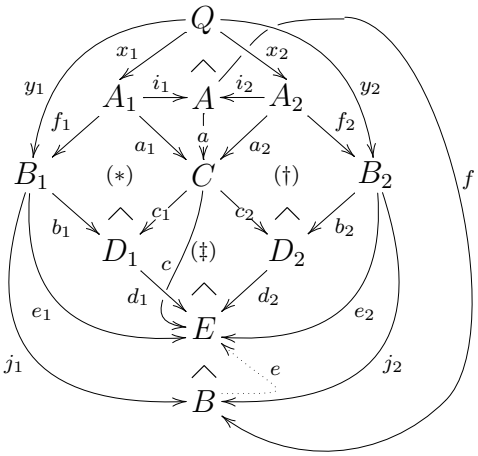


is a pushout.

Proof.

(i) ⇒ (ii)

First assemble the given diagrams into one.



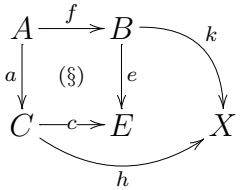
(A.5)

Next we need to check that $e \circ f = c \circ a$; for this we will use that i_1 and i_2 are jointly epic, i.e. we will show that both $e \circ f \circ i_1 = c \circ a \circ i_1$ and $e \circ f \circ i_2 = c \circ a \circ i_2$ hold.

$$
\begin{aligned}
e \circ f \circ i_1 &= e \circ j_1 \circ f_1 && \text{see Item (i)} \\
&= e_1 \circ f_1 && \text{Equation (A.4)} \\
&= d_1 \circ b_1 \circ f_1 && \text{see Diagram (A.5)} \\
&= d_1 \circ c_1 \circ a_1 && \text{Square } (*) \\
&= c \circ a \circ i_1 && \text{Region } (\ddagger) \text{ and Equation (A.3)}
\end{aligned}
$$

By symmetry we get also $e \circ f \circ i_2 = c \circ a \circ i_2$ and hence we have shown that the square (§) of Item *(ii)* commutes; it remains to show that it satisfies the universal property of pushouts.

Hence assume there is a commuting diagram as



(A.6)

Now we have

$$\begin{aligned} h \circ a_1 &= h \circ a \circ i_1 && \text{Equation (A.3)} \\ &= k \circ f \circ i_1 && \text{Diagram (A.6)} \\ &= k \circ j_1 \circ f_1 && \text{see Diagram (A.1)} \end{aligned}$$

and similarly we derive $h \circ a_2 = k \circ j_2 \circ f_2$. Because the squares $(*)$ and $(\dagger)$ are pushouts there are uniquely determined morphisms $z_1 \colon D_1 \to X$ and $z_2 \colon D_2 \to X$ which satisfy

$$z_1 \circ c_1 = h \qquad z_2 \circ c_2 = h \tag{A.7}$$

$$z_1 \circ b_1 = k \circ j_1 \qquad z_2 \circ b_2 = k \circ j_2. \tag{A.8}$$

Using Equation (A.7) and the fact that square $(\ddagger)$ is a pushout we derive that there is exactly one morphism $z \colon E \to X$ such that

$$z \circ d_1 = z_1 \qquad \text{and} \qquad z \circ d_2 = z_2 \tag{A.9}$$

hold. This z is a candidate for the mediating morphism we are are looking for (see Diagram (A.6)).

In fact we derive

$$\begin{aligned} z \circ e \circ j_1 &= z \circ e_1 && \text{Equation (A.4)} \\ &= z \circ d_1 \circ b_1 && \text{see Diagram (A.1)} \\ &= z_1 \circ b_1 && \text{Equation (A.9)} \\ &= k \circ j_1 && \text{Equation (A.8)} \end{aligned}$$

and by symmetry we may conclude that also $z \circ e \circ j_2 = k \circ j_2$. However j_1 and j_2 are jointly epic, which yields $z \circ e = k$. Moreover

$$\begin{aligned} z \circ c &= z \circ d_2 \circ c_2 && \text{Square } (\ddagger) \\ &= z_2 \circ c_2 && \text{Equation (A.9)} \\ &= h && \text{Equation (A.7)} \end{aligned}$$

i.e. we have the equalities

$$z \circ e = k \qquad \text{and} \qquad z \circ c = h. \tag{A.10}$$

It remains to show that z is the unique mediating morphism, i.e. that every

other morphism $\zeta\colon E \to X$ satisfying

$$\zeta \circ e = k \qquad \text{and} \qquad \zeta \circ c = h \tag{A.11}$$

is equal to z. So assume that some morphism ζ satisfying Equation (A.11) is given. We put

$$\zeta_1 := \zeta \circ d_1 \qquad \text{and} \qquad \zeta_2 := \zeta \circ d_2. \tag{A.12}$$

Now we derive

$$\begin{aligned}
\zeta_1 \circ b_1 &= \zeta \circ d_1 \circ b_1 && \text{Equation (A.12)} \\
&= \zeta \circ e_1 && \text{by assumption of Item (i)} \\
&= \zeta \circ e \circ j_1 && \text{Equation (A.4)} \\
&= k \circ j_1 && \text{Equation (A.11)} \\
&= z \circ e \circ j_1 && \text{Equation (A.10)} \\
&= z \circ e_1 && \text{by assumption of Item (i)} \\
&= z \circ d_1 \circ b_1 && \text{Equation (A.12)} \\
&= z_1 \circ b_1 && \text{Equation (A.9)}
\end{aligned}$$

and in the same way arrive at $\zeta_2 \circ b_2 = z_2 \circ b_2$, i.e. we have shown that

$$\zeta_1 \circ b_1 = z_1 \circ b_1 \qquad \text{and} \qquad \zeta_2 \circ b_2 = z_2 \circ b_2. \tag{A.13}$$

Further

$$\begin{aligned}
\zeta_1 \circ c_1 &= \zeta \circ d_1 \circ c_1 && \text{Equation (A.12)} \\
&= \zeta \circ c && \text{Square (‡)} \\
&= h && \text{Equation (A.11)} \\
&= z \circ c && \text{Equation (A.10)} \\
&= z \circ d_1 \circ c_1 && \text{Square (‡)} \\
&= z_1 \circ c_1 && \text{Equation (A.9)}
\end{aligned}$$

and similarly $\zeta_2 \circ c_2 = z_2 \circ c_2$ and thus we also have shown

$$\zeta_1 \circ c_1 = z_1 \circ c_1 \qquad \text{and} \qquad \zeta_2 \circ c_2 = z_2 \circ c_2. \tag{A.14}$$

Using Equation (A.13) and Equation (A.14) we may conclude that $\zeta_1 = z_1$ and $\zeta_2 = z_2$ since the pairs (b_1, c_1) and (b_2, c_2) are jointly epic because the squares

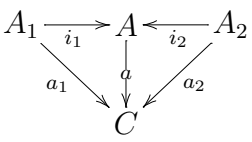
$(*)$ and $(\dagger)$ are pushouts. However using Equation (A.12) and Equation (A.9) we can also derive

$$\begin{aligned} \zeta \circ d_1 &= \zeta_1 \\ &= z_1 \\ &= z \circ d_1 \end{aligned} \qquad \begin{aligned} \zeta \circ d_2 &= \zeta_2 \\ &= z_2 \\ &= z \circ d_2, \end{aligned}$$

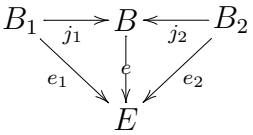
from which $z = \zeta$ follows since d_1 and d_2 are jointly epic.

(ii) $\Rightarrow$ **(i)**

By assumption we have the following commuting diagrams.

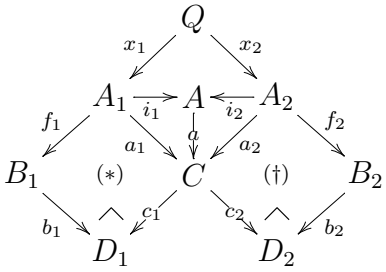


(A.15a)



(A.15b)

Further we construct the pushouts for the pairs (f_1, a_1) and (f_2, a_2), and assemble them into the following diagram



where the upper triangle commutes by assumption. Now we derive

$$\begin{aligned} c \circ a_1 &= c \circ a \circ i_1 && \text{Diagram (A.15a)} \\ &= e \circ f \circ i_1 && \text{Square (§)} \\ &= e \circ j_1 \circ f_1 && \text{Diagram (A.1)} \\ &= e_1 \circ f_1 && \text{Diagram (A.15b)} \end{aligned}$$

and similarly we we derive $c \circ a_2 = e_2 \circ f_2$. Hence there are unique morphisms $d_1 \colon D_1 \to E$ and $d_2 \colon D_2 \to E$ such that the following hold.

$$d_1 \circ b_1 = e_1 \quad \text{and} \quad d_1 \circ c_1 = c \tag{A.16a}$$

$$d_2 \circ b_2 = e_2 \quad \text{and} \quad d_2 \circ c_2 = c \tag{A.16b}$$

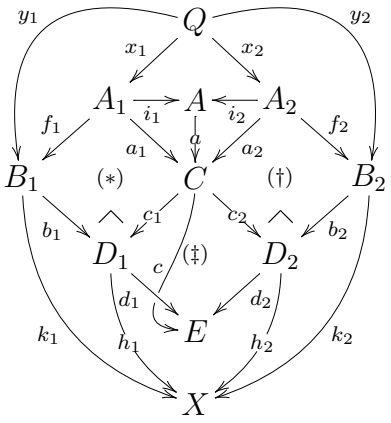
It remains to show that the square $C \xrightarrow{c_1} D_1 \underset{d_1}{\overset{c_2}{\rightrightarrows}} D_2 \xrightarrow{d_2} E$ is a pushout.

For this let there be two morphisms $h_1 \colon D_1 \to X$ and $h_2 \colon D_2 \to X$ such that

$$h_1 \circ c_1 = h_2 \circ c_2. \tag{A.17}$$

Hence after defining $k_1 := h_1 \circ b_1$ and $k_2 := h_2 \circ b_2$,

and because Diagram (A.1) commutes we arrive at the following commuting diagram.



(A.18)

Using its commutativity we derive

$$\begin{aligned} k_1 \circ y_1 &= h_1 \circ b_1 \circ f_1 \circ x_1 \\ &= h_1 \circ c_1 \circ a_1 \circ x_1 && \text{Equation (A.17)} \\ &= h_2 \circ c_2 \circ a_2 \circ x_2 \\ &= h_2 \circ b_2 \circ f_2 \circ x_2 \\ &= k_2 \circ y_2 \end{aligned}$$

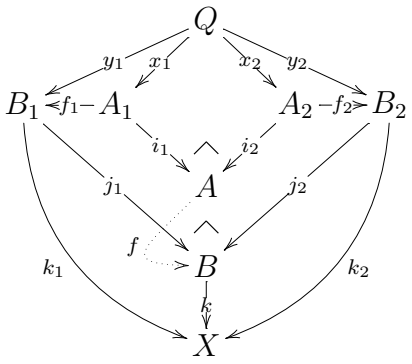
and therefore there exists a unique morphism $k \colon B \to X$ such that

$$k_1 = k \circ j_1 \qquad \text{and} \qquad k_2 = k \circ j_2. \tag{A.19}$$

Moreover $k_1 \circ y_1 = k_2 \circ y_2$ implies $k_1 \circ f_1 \circ x_1 = k_2 \circ f_2 \circ x_2$ by "expansion" of y_1 and y_2, which provides us with a uniquely determined morphism $u \colon A \to X$ such that

$$k_1 \circ f_1 = u \circ i_1 \qquad \text{and} \qquad k_2 \circ f_2 = u \circ i_2. \tag{A.20}$$

Now looking at



we see that $k_1 \circ f_1 = k \circ f \circ i_1$ and $k_2 \circ f_2 = k \circ f \circ i_2$ and hence

$$k \circ f = u \tag{A.21}$$

follows from the characterization of u in (A.20).

However we can also derive the following:

$$\begin{aligned} k_1 \circ f_1 &= h_1 \circ b_1 \circ f_1 \\ &= h_1 \circ c_1 \circ a_1 \\ &= h_1 \circ c_1 \circ a \circ i_1 \end{aligned} \qquad \begin{aligned} k_2 \circ f_2 &= h_2 \circ b_2 \circ f_2 \\ &= h_2 \circ c_2 \circ a_2 \\ &= h_1 \circ c_1 \circ a \circ i_2 \end{aligned}$$

whence $h_1 \circ c_1 \circ a = u \overset{(A.21)}{=} k \circ f$ where we used uniqueness of the mediating morphism u to derive the first equality (see Equation (A.20)). Since $h_1 \circ c_1 = h_2 \circ c_2$ we also get $h_2 \circ c_2 \circ a = k \circ f$.

Now since Square (§) is a pushout we know that there is a unique morphism $z \colon E \to X$ such that

$$z \circ c = h \quad \text{and} \quad z \circ e = k \qquad \text{where } h = h_1 \circ c_1 = h_2 \circ c_2. \tag{A.22}$$

This is z is the candidate for the mediating morphism we are looking for.

To show that it is the unique one we will use that b_1 and c_1 are jointly epic to derive that $z \circ d_1 = h_1$.

$$\begin{aligned} z \circ d_1 \circ c_1 &\overset{(\S)}{=} z \circ c \\ &\overset{(A.22)}{=} h_1 \circ c_1 \end{aligned} \qquad \begin{aligned} z \circ d_1 \circ b_1 &\overset{(A.16a)}{=} z \circ e_1 \\ &\overset{(A.15b)}{=} z \circ e \circ j_1 \\ &\overset{(A.19)}{=} k_1 \\ &\overset{(A.18)}{=} h_1 \circ b_1 \end{aligned}$$

This shows that $z \circ d_1 = h_1$ and mutatis mutandis $z \circ d_2 = h_2$, and z is a mediating morphism from $E \to X$: it remains to show that it is the only one.

Let $\zeta \colon E \to X$ be a morphism such that $\zeta \circ d_1 = h_1$ and $\zeta \circ d_2 = h_2$ hold; we have to show that $\zeta = z$. We derive

$$\begin{aligned} \zeta \circ c &= \zeta \circ d_1 \circ c_1 && \text{Square } (\ddagger) \\ &= h_1 \circ c_1 && \text{by assumption} \\ &= z \circ c && \text{Equation (A.22)} \end{aligned}$$

If also $\zeta \circ e = k$ then $z = \zeta$ holds because e and c are jointly epic; thus it remains to show that $\zeta \circ e = k$.

Since j_1 and j_2 are jointly epic it is enough to show that $\zeta \circ e \circ j_1 = k \circ j_1$ and $\zeta \circ e \circ j_2 = k \circ j_2$. However we can derive (see Diagram (A.18)).

$$\begin{aligned} k_1 \circ y_1 &= h_1 \circ b_1 \circ f_1 \circ x_1 \\ &= \zeta \circ d_1 \circ c_1 \circ a_1 \circ x_1 && \text{by assumption} \\ &= \zeta \circ c \circ a \circ i_1 \circ x_1 \\ &= \zeta \circ e \circ f \circ i_1 \circ x_1 && (\S) \\ &= \zeta \circ e \circ j_1 \circ f_1 \circ x_1 && \text{Diagram (A.1)} \\ &= \zeta \circ e \circ j_1 \circ y_1 \end{aligned}$$

and mutatis mutandis also $k_2 \circ y_2 = \zeta \circ e \circ j_2 \circ y_2$ This yields that $\zeta \circ e$ is the unique arrow such that $\zeta \circ e \circ j_1 = k_1$ and $\zeta \circ e \circ j_2 = k_2$. Expanding the definition of k_1 and k_2 we arrive at $\zeta \circ e \circ j_1 = k \circ j_1$ and $\zeta \circ e \circ j_2 = k \circ j_2$ and the proof is finished.

A graph abstract machine describing event structure composition

Claudia Faggian and Mauro Piccolo
Dip. Matematica Pura e Applicata, Universitá di Padova – PPS, Paris7-CNRS

Abstract

Event structures, Game Semantics strategies and Linear Logic proof-nets arise in different domains (concurrency, semantics, proof-theory) but can all be described by means of directed acyclic graphs (dag's). They are all equipped with a specific notion of composition, interaction or normalization.

We report on-going work, aiming to investigate the common dynamics which seems to underly these different structures.

In this paper we focus on confusion free event structures on one side, and linear strategies [Gir01, FM05] on the other side. We introduce an abstract machine which is based on (and generalizes) strategies interaction; it processes labelled dag's, and provides a common presentation of the composition at work in these different settings.

1 Introduction

Event structures [NPW81], Game Semantics strategies and Linear Logic proof-nets [Gir87] arise in different domains (concurrency, semantics, proof-theory) but can all be described by means of directed acyclic graphs (dag's). They are all equipped with a specific notion of composition, interaction or normalization. In this paper we report ongoing work whose first aim is to investigate the common dynamics which appears to underly all these different structures, and eventually to transfer technologies between these settings.

As a first step in this direction, here we present an abstract machine which processes labelled dag's. The machine is based on the dynamics at work when composing Game Semantics strategies. When applied to linear strategies (in the form introduced in [Gir01] or [FM05]) the machine implements strategies composition. When applied to event structures, the result is the same as the *paralle composition* of event structures defined by Varacca and Yoshida in [VY06].

Event structures. *Event structures* are a *causal model* of concurrency (also called true concurrency models), i.e. causality, concurrency and conflict are directly represented, as opposite to *interleaving models*, which describe the system by means of all possible scheduling of concurrent actions.

An event structure models parallel computation by means of

- occurrence of events;
- a partial order expressing causal dependency.

Non-determinism is modelled by means of:

- a *conflict* relation, which expresses how the occurrence of certain events rules out the occurrence of others.

Two events are *concurrent* if they are neither causally related, nor in conflict. Events which are in conflict live in different possible evolutions of the system.

In this paper we will consider two classes of event structures:

confusion free event structure (where conflict, and hence non-determinism, is well behaving)

conflict free event structures (where there is no conflict, and hence no non-determinism).

Confusion free event structure, are an important class of event structures because the choices which a process can do are "local" and not influenced by independent actions. In this sense, confusion freeness generalizes *confluence* to systems that allow nondeterminism.

A point which is centra to our approach is that a confusion free event structure $\mathcal{E}$ can be seen as a superposition of conflict-free event structures (which we will call the *slices* of $\mathcal{E}$): each slice represents a possible and *independent* evolution of the system.

Because of this, if $\mathcal{E}$ is confusion free, the study of several properties can be reduced to the study of such properties in conflict free event structures.

Game Semantics A distinction between causal and interleaving models is appearing also in Game Semantics. In this setting, a *strategy* describes in an abstract way the operational behaviour of a term. In the standard approach, a strategy is described by sequences of actions (*plays*), which represent the traces of the computation. However, there is an active line of research in Game Semantics aiming at relaxing sequentiality, either with the purpose to have "partial order" models of programming languages or to capture concurrency [AM99, Mel04, HS02, MW05, SPP05, FM05, CF05, Lai05, GM04]. The underlying idea is to not completely specify the order in which the actions should be performed, while still being able to express constraints. Certain tasks may have to be performed before other tasks; other actions can be performed in parallel, or scheduled in any order. A strategy is here a *directed acyclic graph*.

Interaction and composition. Games and strategies provide denotational models for programming languages and logical systems; games correspond to types (formulas), and strategies to programs (proofs).

The central notion is that of composition, which models program composition (normalization of proofs).

Confusion free event structures and linear strategies We are interested in relating strategies and event structures. Abramsky, Mellies, Schalk have already used event structures in Game Semantics as arenas (i.e. types, or objects). However, our aim is to see *event structures as strategies* (i.e. as morphisms).

We focus on the class of linear strategies, i.e. strategies which correspond to the multiplicative-additive structure of Linear Logic, Linear strategies (as defined in [Gir01] and then [FM05]) can be described as partial orders with a conflict relation, i.e. as a sort of event structures, which satisfy a number of conditions. In particular, they are confusion free. Many of the properties which make the composition work appear to depend only on confusion freeness .

Our aim is therefore to see *confusion free* event structures as a generalization of strategies, and the composition of such event structures as strategies composition.

An idea which underlies the work on types by Honda and Yoshida is that typed processes should be seen as a sort of Hyland-Ong strategies; this is implicit in particular in [VY06], on which our work builds.

In [VY06], Varacca and Yoshida provide a typing system which guarantees that the composition of confusion free event structures is confusion free. The typing is inspired by Linear logic and Hyland-Ong strategies, and allows them to give an event structure semantics for (a variant of) Sangiorgi's πI-calculus.

In the paper we define composition *"operationally"*, in such a way that when restricted to strategies the machine implements strategies composition. Applied to confusion free event structures, the result is the same as the composition obtained in a more standard way. In particular, we prove the equivalence with the composition in [VY06].

We believe that the machine provides a simple and direct implementation of event structures composition.

2 Background

2.1 A sketch of strategies composition

In Game Semantics, the execution of a program is modelled as interaction between two players; let us call them P (Proponent) and O (Opponent). The role of a strategy is to tell the player how to respond to a counter-player move. The dialogue between the two players will produce an interaction (a play).

Figure 1 presents a simplified example of two (sequential) strategies. A specific move is played by (belongs to) only one of the players, so there are P-moves and O-moves. The active (positive) move of P are those that P plays, while its passive (negative) moves are those played by O, and to which P has to respond. In the picture, for each player strategy we distinguish the actives (positive) moves, i.e. those which belong to that player, with circles.

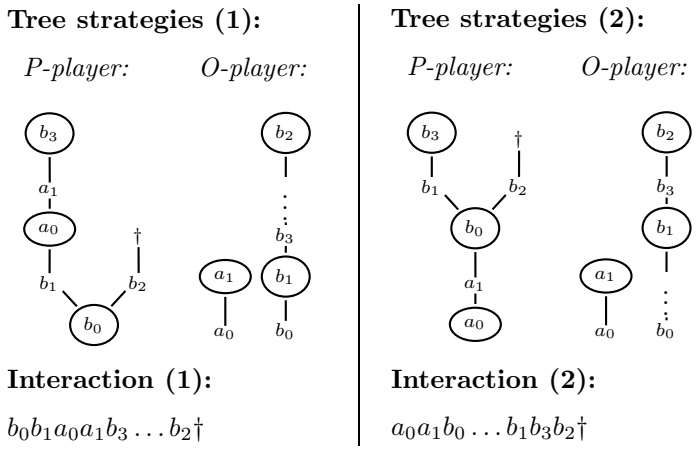


Figure 1: Tree strategies

Let us look at the strategies (1). According to the P-player strategy, it will start with b_0, then respond with a_0 to Opponent move b_1, and with † (termination) to Opponent move b_2. Let us make it interact with the O-player strategy. The interaction goes as follows: O answer to b_0 is b_1, P answer to b_1 is a_0, O answer to a_0 is a_1, and so on.

The algorithm to calculate the interaction is simple. (i.) Start from P-player initial move, (ii.) Check counter-player answer to that move, that is, go to the corresponding opposite action, and take the following move. (iii.) Repeat step (ii.) until terminating on †.

Figure 2 illustrates the same ideas for more parallel strategies.

The strategies are here graphs. The way to make them interact is similar to the previous one, but (1.) there are several threads running in parallel, (2.) on certain moves we need to synchronize.

2.2 Event structures

Event structures were introduced by Nielsen, Plotkin, and Winskel [NPW81, Win87, WN95], as a theory combining Petri nets and domain theory.

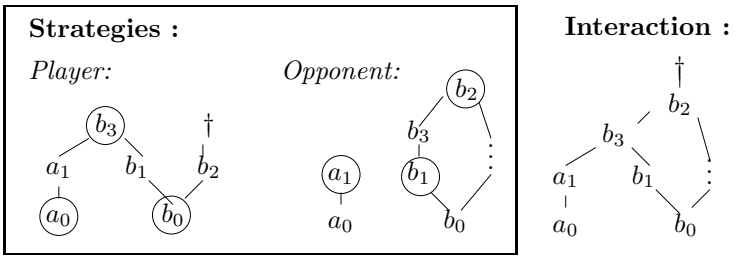


Figure 2: Graph strategies

Let $(\mathcal{E}, \leq)$ be a partially ordered set. Elements of $\mathcal{E}$ are called *events*; we assume that $\mathcal{E}$ is at most countable. The order relation is called *causality relation*.

The *downward closure* of a subset $S \subseteq \mathcal{E}$ is defined by $\lceil S \rceil = \{e' : e' \leq e, e \in S\}$. For a singleton, we write $\lceil e \rceil$.

An **event structure**[1] is a triple $(\mathcal{E}, \leq, \smile)$ such that

- $(\mathcal{E}, \leq)$ is a partial order, as above;
- For every $e \in \mathcal{E}$, $\lceil e \rceil$ is finite.
- $\smile$ is an irreflexive and symmetric relation, called *conflict relation*, which satisfies the following:

$$\text{for every } e_1, e_2, e_3 \in \mathcal{E}, \text{ if } e_1 \leq e_2 \text{ and } e_1 \smile e_3 \text{ then } e_2 \smile e_3.$$

If $e_1 \leq e_2$ we say that the conflict $e_2 \smile e_3$ is *inherited* from the conflict $e_1 \smile e_3$. If a conflict is not inherited, we say that it is *immediate*, written $\smile_\mu$

Causal order and conflict are mutually exclusive. If two events are not causally related nor in conflict, they are said to be *concurrent*.

With a slight abuse of notations, we identify an event structure $(E, \leq, \smile)$ and its set $\mathcal{E}$ of events.

A *labelled event structure* is an event structure $\mathcal{E}$ together with a labelling function $\lambda : \mathcal{E} \to L$, where L is a set of labels.

Conflict free A set $S \subseteq E$ is *conflict free* if it does not contain any two elements in conflict; in particular, an event structure $\mathcal{E}$ is conflict free if its conflict relation is empty.

Hence, a conflict free event structure is simply a partial order.

Observe that $\lceil e \rceil$ is conflict free.

Parents and enabling set. Let us introduce two notations that will be useful.

Given $e \in \mathcal{E}$

- *Parents*(e) denotes the set of immediate predecessors of e in $\leq$ (its preconditions);
- $[e) = \lceil e \rceil \setminus \{e\}$ is the enabling set of e.

2.3 Confusion free event structures

Confusion free event structures are a class of event structures where every choice is localized to *cells*, where a cell is a set of events that are pairwise in immediate conflict, and have the same enabling set.

[1] In this paper we say event structures always meaning *prime event structures*.

Definition 2.1 (Cell) *A* cell *c is a maximal set of events such that $e, e' \in c$ implies $e \smile_m e'$ and $[e) = [e')$.*

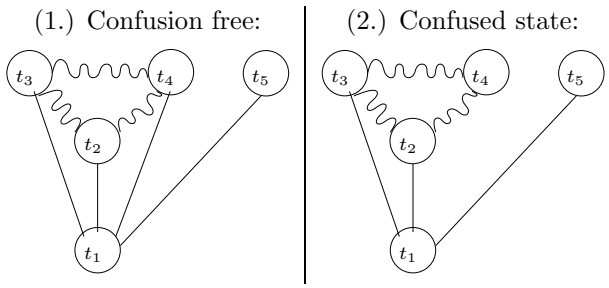
Definition 2.2 (Confusion free) *$\mathcal{E}$ is confusion free if the following holds:*

(a.) for all distinct $e, e', e'' \in \mathcal{E}$, $e \smile_\mu e'$ and $e' \smile_\mu e''$ implies $e \smile_\mu e''$

(b.) for all $e, e' \in \mathcal{E}, e \smile_\mu e'$ implies $[e) = [e')$

2.3.1 Example.

Below, we give an example of event structure which is confusion free, and an example of an event structure which is not. Waved lines denote conflict.



The intuition behind. Let $T = \{t_1, t_2, t_3, t_4, t_5\}$ be a set of tasks on which an order (propedeuticity) and a conflict relation are defined. We have to schedule them.

In the confusion free case (1.), the scheduler must start with t_1. Then the situation is the following:

- after t_1, the scheduler can choose t_5 or either one of t_2, t_3, t_4.
- if the scheduler choose t_1 and then one of t_2, t_3, t_4, then it can still schedule t_5.
- if the scheduler choose t_1 and then t_5, then it can still choose to schedule either of t_2, t_3, t_4.

The case (2.), instead, describe a state which is confused: changing the scheduling of the tasks, some choices which were available may be no longer available.

If we look at the picture, we see that both t_1 and t_4 have no precondition, and hence can be scheduled first. After t_1, we can schedule either of t_2, t_3, t_4 if we start with t_1. However, if we schedule t_4 first, and then t_1, after t_1 the choices t_2, t_3 are no longer available.

3 Typed event structures

In this section we introduce a notion of labelled event structure, where the labelling guarantees that the composition of confusion free event structures is a confusion free event structure.

Our labelling can be seen as a minimalist variant of the typing in [VY06], without the whole setting of linear types and morphisms; this because here we are only interested in the preservation of confusion freeness via composition.

Our labelling is indeed a straightforward generalization of the technique developed in [Gir01] to deal with additive strategies.

A key features of the labelling is that *a name identifies a cell* (rather than a single event).

3.1 Names and actions.

We assume a countable set of names N, ranged over by $\alpha, \beta, \ldots$. We are going to label a confusion free event structure with actions on these names. Let S be an index set. We define the alphabet $\mathcal{N}$ as follows:

$$\mathcal{N} = \sum_{i \in S} N_i = \{(\alpha, i) : \alpha \in N and\ i\ \in S\}$$

We say that $a = (\alpha, i)$ uses the name α, and also write $name(a) = \alpha$.

A (polarized) action is given by an element $a \in \mathcal{N}$ and a polarity, which can be positive (a^+), negative (a^-), or neutral ($a^\pm$).

Remark 3.1 *Actions of opposite polarity (a^+, a^-) denote matching dual actions, such as c and $\overline{c}$ in CCS, or Player/Opponent moves in Game Semantics.*

We think of $a^\pm$ as a pair of matching actions a^+, a^- which have synchronized. A more traditional and suggestive denotation for $a^\pm$ would be τ_a.

We use the variable ϵ to vary over polarities: $\epsilon \in \{+, -, \pm\}$. When clear from the context, or not relevant, we omit the explicit indication of the polarity.

The polarities $+$ and $-$ are said opposite. If a is a positive or negative action, $\overline{a}$ will denote its opposite action.

3.2 Interfaces.

We are going to type event structures with an interface. The interface provides in particular the set of names on which the event structure communicate, and their polarity.

An **interface** (A, π_A) is given by a finite set of names A, and a polarity (positive, negative, neutral) for each name. The polarization partitions A into three disjoint sets: positive, negative and neutral names.

Remark 3.2 *The positive names can be thought of as sending, the negative name as receiving, and the neutral names as private.*

The interface (A, π) generates the set of actions $\mathcal{A} = \sum_{i \in S} A_i = \{(\alpha, i) : \alpha \in A\}$. The polarization of the names extends to the actions with that name.

3.3 Typed event structures.

An event structure of interface A is a tuple $(\mathcal{E}, A, \lambda, \pi)$ where

- $\mathcal{E}$ is an event structure;
- $A = (A, \pi_A)$ is an interface;
- $\lambda : \mathcal{E} \to \{(\alpha, i) : \alpha \in A, i \in S\}$ is a labelling function;
- $\pi : \mathcal{E} \to \{+, -, \pm\}$ is the polarization induced on the actions by π_A.

If $\lambda(e) = (\alpha, i)$, we say that the event e uses the name α, and write $name(e) = \alpha$.

Remark 3.3 *If $\lambda(e) = a$, with $a = (\alpha, i)$, and $\pi_A(\alpha) = \epsilon$, then e is labelled by the action a^ϵ.*

We type an event structure of interface A only when it is confusion free; we ask that:

- all the events in the same cell use the same name (and hence also have the same polarity).
- two events which use the same name (and the same polarity) are in conflict.

Definition 3.4 *An event structure $\mathcal{E}$ of interface A has type A, written $\mathcal{E} : A$ if it satisfies the following, for all distinct $e_1, e_2 \in \mathcal{E}$.*

1. *if* $e_1 \smile_\mu e_2$ *and* $\lambda(e_1) = (\alpha, i)$*, then* $\lambda(e_2) = (\alpha, j)$*, with* $i \neq j$*.*

2. *if* $name(e_1) = name(e_2)$*then* $e_1 \smile e_2$*.*

3. $e_1 \smile_\mu e_2 \Rightarrow Parents(e_1) = Parents(e_2)$

Remark 3.5 *$\mathcal{E}$ can receive a type if and only if it is confusion free.*
Properties 1. and 2. imply 2.2.i; property 3. is equivalent to 2.2.ii.

3.4 Properties of the labelling

Given a labelled event structure $\mathcal{E}$, and a set of events $S \subseteq \mathcal{E}$, we use the notation $\lambda S = \{\lambda(s) | s \in S\}$.

3.4.1 Set of labels identify events

Each event $e \in \mathcal{E}$ is uniquely identified by the set of labels $\lambda\lceil e \rceil = \{\lambda e', e' \leq e\}$.

Proposition 3.6 *Given* $e_1 \neq e_2 \in \mathcal{E}$*, we have that* $\lambda\lceil e' \rceil \neq \lambda\lceil e' \rceil$

3.4.2 Conflicts

The conflict relation in typed event structures can be inferred from the labels:

Proposition 3.7 *Let* $\mathcal{E} : A$ *a typed event structure and let* $e_1, e_2 \in \mathcal{E}$*. Then the following holds:*

$$(**) \quad e_1 \smile e_2 \iff \exists e_1' \leq e_1, e_2' \leq e_2 : name(e_1') = name(e_2')$$

Since in a typed event structure the labels carry all the information on the conflict relation, from now on, *we deal with the conflict implicitly*: two distinct events e_1 and e_2 are in conflict iff (**) holds.
This allows us to only focus on the partial order.

It is easy to see that

Proposition 3.8 *Given* $e_1 \neq e_2 \in \mathcal{E} : A$*, we have that* $e_1 \smile_\mu e_2$ *iff*

- e_1, e_2 *use the same name*
- $[e_1) = [e_2)$

3.4.3 Typed event structures as dag's

As seen in the previous section, given a typed event structure, we can deal with the conflict implicitly; we are left to deal only with the partial order $\mathcal{E}, \leq$.
In the following, it will be convenient to identify the partial order on $\mathcal{E} : A$ with the associated dag. This in particular allow us to describe composition in terms of graph rewriting.

A *directed acyclic graph* (dag) G is an oriented graph without (oriented) cycles. We will write $c \leftarrow b$ if there is an edge from b to c. It is standard to represent a strict partial order as a dag, where we have a (non transitive) edge $a \leftarrow b$ whenever there is no c such that $a < c$ and $c < b$. Conversely (the transitive closure of) a dag is a strict partial order on the nodes.
In the following, we will identify the partial order on $\mathcal{E} : A$ with the associated dag. We take as canonical representative of $\mathcal{E}$ its *skeleton* (the minimal graph whose transitive closure is the same as that of $\mathcal{E}$).

Remark 3.9 *Observe that, by construction, the skeleton is always defined, even if $\mathcal{E}$ can have a countable number of events (in particular, a cell can have a countable number of events). In fact, for each event $e \in \mathcal{E}$, $\lceil e \rceil$ does not contain any conflict, and it is finite.*

4 Composition

We define composition *"operationally"*, in such a way that when restricted to strategies this procedure produces strategies composition.

Composition between event structures relies on two notions: synchronization and enabling (reachability). Intuitively, to compose, we synchronize (match) events which are labelled by the same action, and opposite polarity. The synchronization is possible only between events which have been *enabled*. We enable (reach) an action only if all the actions before it have been enabled (reached). We better illustrate this in Section 4.2.

4.1 Compatible interfaces

We compose two event structures when their interfaces are able to communicate.

Definition 4.1 (Compatible interfaces) *Let (A, π_A) and (C, π_C) be two interfaces. The interfaces A and C are* compatible *if*

for all $b \in A \cap C$, the polarity of b in A is opposite to the polarity of b in C.

If the interfaces are compatible, we define their composition $A \odot C = (A \cup C, \pi)$ where

$$\pi(\alpha) \quad = \quad \begin{cases} \pi_A(\alpha) & \text{if } \alpha \in A \setminus C \\ \pi_C(\alpha) & \text{if } \alpha \in C \setminus A \\ \pm & \text{otherwise} \end{cases}$$

Definition 4.2 (Private and public names) *Given two compatible interfaces A, C, we say that the name α is* **private** *if $\alpha \in A \cap C$,* **public** *otherwise.*

Example. If $A = \{a^-, b^+\}$ and $C = \{b^-, c^+\}$, then $A \odot C = \{a^-, c^+, b^\pm\}$. The name b is private, while the names a, c are public.

Composition is only defined on event structures which have compatible interfaces.

4.2 Conflict free composition

Let us first analyze composition in the case of conflict free event structures, i.e. when no two events are in conflict. This case is very simple and clear, but contains all the dynamics of the general case.

The key property of this case is the following

If $\mathcal{E} : A$ is conflict free, *no two events use the same name.*

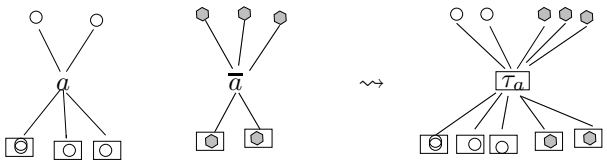
Through this section, let us assume that $\mathcal{E}_1 : A$ and $\mathcal{E}_2 : C$ are conflict free and have compatible interfaces. Their composition $\mathcal{E}_1 \| \mathcal{E}_2$ is a conflict free event structure of interface $A \odot C$.

Let us describe the composition by means of a wave of tokens travelling up on $\mathcal{E}_1 \uplus \mathcal{E}_2$. When a private action is reached, to continue, it is necessary to synchronize it with an action of opposite polarity. Observe that, by construction, there is at most one such action.

Remark 4.3 *In $\mathcal{E}_1 \uplus \mathcal{E}_2$ there is only one occurrence of each polarized action. For this reason, in this section, we can identify each event with the polarized action which labels it.*

1. If a is public, and its parents have been enabled, then a is enabled. We illustrate this in the picture below, where the squares mark the enabled nodes.

2. If a is private, a^+, a^- are both present, and their parents have been enabled, then a is enabled, and the graph is transformed as follows:



end: The actions which have not been enabled are deleted (garbage collection).

The process described above generates a new conflict free event structure $(E, \leq)$, where E is a set of events labelled by the actions which have been enabled; the actions have the polarity induced by the new interface $A \odot C$.

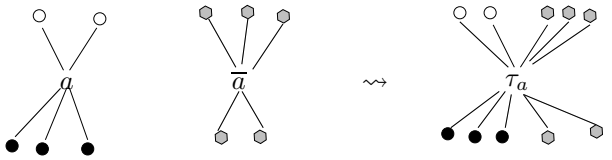
It is straightforward to give a direct recursive definition of $\mathcal{E}_1 \| \mathcal{E}_2$. We do not do this in this Section, but in Section 4.5 we use a definition of this kind to describe composition in the general case, and to establish the equivalence of our procedure with more standard definitions.

4.3 Local rewriting rules

The process described in Section 4.2 can be expressed by means of a set of local graph rewriting rules on $\mathcal{E}$, which we describe in Figure4.3.

Private a:

1. There are $a, \overline{a}$, parallel



2. Otherwise:

Figure 3: Graph Rewriting Rules

It is straightforward to show these rules are confluent. By using this fact, one can prove associativity for the composition.

Proposition 4.4 (Associativity) *Let $\mathcal{E}_1 : A, \mathcal{E}_2 : C, \mathcal{E}_3 : D$ be conflict free event structures. If the interfaces allow the composition, we have that*

$$(\mathcal{E}_1 \| \mathcal{E}_2) \| \mathcal{E}_3 = \mathcal{E}_1 \| (\mathcal{E}_2 \| \mathcal{E}_3)$$

4.4 Reducing composition to conflict free composition

A confusion free event structures $\mathcal{E}$ can be seen as a superposition of conflict-free event structures (which we call the slices of $\mathcal{E}$). The study of confusion free event structures can be reduced to the study of conflict free event structures. In particular, composition of confusion free event structure can be reduced to the composition of its slices.

4.4.1 Slices

A slice $\mathcal{S}$ of $\mathcal{E}$ is a downward closed, conflict free subset of $\mathcal{E}$, with the order induced by $\mathcal{E}$.

To choose a (maximal) slice of $\mathcal{E}$ corresponds to the selection of a single element in each cell of $\mathcal{E}$.

4.4.2 Studying composition by slices

A key feature of the composition is that it takes place *independently* inside each single slices (Proposition 4.10) .

Several interesting properties of the composition of two event structures (such as confluence, or deadlocks) can be analyzed as properties of their slices (see 4.9).

Actually, following an approach which is well studied for proof nets and linear strategies, the process of composition itself could be reduced purely to conflict free composition:

- decompose $\mathcal{E}$ into its slices
- compose all slices pairwise
- *superpose* the composed slices

Proposition 3.6 allows us to perform the superposition, by using the same technique developed in [Gir01, FM05].

We do not give details here; however, after providing a direct description of composition in the general case, we show that the study of the composition can be reduced to the study of conflict free composition (Proposition 4.10).

4.5 Global composition

In this section, we provide a direct description of composition of typed event structures, in the general case.

Let $\mathcal{E}_1 : A$ and $\mathcal{E}_2 : C$ be typed event structures with compatible interfaces. $\mathcal{E}_1 \| \mathcal{E}_2$ is an event structure of interface $A \odot C$, obtained as follows.

Case 1. Let $e \in \mathcal{E}_i$ such that $\lambda(e) = a$ and $name(a)$ is public.

If $S \subseteq \mathcal{E}$ satisfies the following conditions:

parent condition: $\lambda S = \lambda[e)$.

conflict condition: the set S is conflict free

add to $\mathcal{E}$ an event v such that

label: $\lambda(v) = a$

edges: for all $v_i \in \lceil S \rceil$ we have $v_i \leftarrow v$

Case 2. Let $e_1 \in \mathcal{E}_1$ and $e_2 \in \mathcal{E}_2$ such that $\lambda(e_1) = \lambda(e_2) = b$ and $name(b)$ is private.

If $S = S_1 \cup S_2$ where $S_1, S_2 \subseteq \mathcal{E}$ satisfy the following conditions

parent condition: $\lambda S_1 = \lambda[e_1)$, $\lambda S_2 = \lambda[e_2)$.

conflict condition: the set S is conflict free

add to $\mathcal{E}$ an event v such that

label: $\lambda(v) = b$ (this should be thought as τ_b, since $\pi(b) = \pm$)

edges: for all $v_i \in \lceil S \rceil$ we have $v_i \leftarrow v$

Remark 4.5 *The **parent condition** checks that the enabling set of $e \in \mathcal{E}_i$ has been considered, and relies on Proposition 3.6.*

Remark 4.6 *The **conflict condition** says that in $\lceil S \rceil$ there are no two events using the same names (we are using Proposition 3.7.)*

The conflict condition, essentially guarantees that we are working slice by slice, i.e. independently in each slice (see Proposition 4.10)

The machine generates $\mathcal{E} = \mathcal{E}_1 \| \mathcal{E}_2$ step by step; each time we add to $\mathcal{E}$ an event v which refers to [comes from] an event (or a pair of matching events) x in $\mathcal{E}_1 \uplus \mathcal{E}_2$. We add v to $\mathcal{E}$ only if:

- the enabling set of x has already an "image" in $\mathcal{E}$;
- this image is conflict free.

To understand the conflict condition, remember that events in conflict are events which are mutually exclusive. If we need a set of precondition to occur together, they must live in a conflict free event structure $\mathcal{S} \subseteq \mathcal{E}$.

In fact, we can analyze, and even calculate, composition, only by means of conflict free event structures, as we see in Section

4.5.1 Example of composition

Consider the following event structures

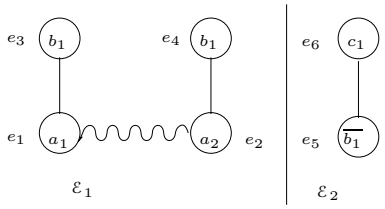
- $\mathcal{E}_1 = \{e_1, e_2, e_3, e_4\}$ with $e_1 < e_3, e_2 < e_4$ and $e_1 \smile_\mu e_2$.
- $\mathcal{E}_2 = \{e_5, e_6\}$ with $e_5 < e_6$

and the interfaces $A = \{\alpha^-, \beta^-\}$ and $C = \{\beta^+, \gamma^-\}$.
We abbreviate (α, i) into a_i, and similarly use b_j, c_k for actions on β, γ.

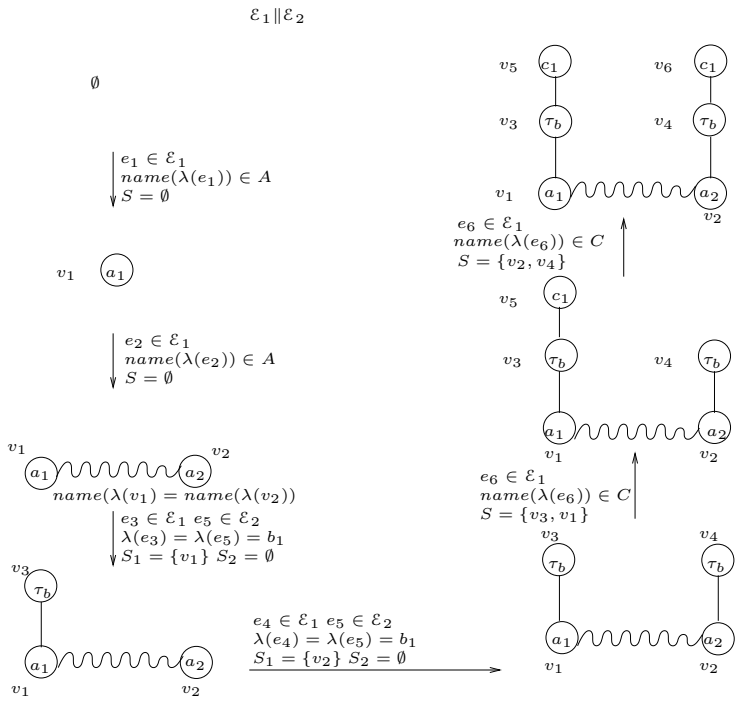
Let us consider

- $\mathcal{E}_1 : A$ with $\lambda(e_1) = a_1, \lambda(e_2) = a_2, \lambda(e_3) = \lambda(e_4) = b_1$
- $\mathcal{E}_2 : C$ with $\lambda(e_5) = b_1, \lambda(e_6) = c_1$.

Here is a graphical representation of the two event structures:



And here we run the machine:



4.5.2 Composition is well defined and associative

Composition of typed event structures produces a typed event structure. Moreover, composition is associative.

Theorem 4.7 $\mathcal{E}_1 \| \mathcal{E}_2 : A \odot C$

Proof. W.r.t. definition 3.4, conditions 1. and 2. (the properties of the labelling) hold by construction. We have to verify 3., i.e. that $u \smile_\mu v$ implies $[v) = [v)$. Let S_u, S_v the two subset of the labelled parent condition. $[u) = [v)$ if and only if $S_u = S_v$. By labelling we have that $name(\lambda(u)) = name(\lambda(v))$ public or private. We develop the public case: the other is analogous. Without loss of generality we can assume $name(\lambda(u)) \in A$. Hence there exists $e, d \in \mathcal{E}_1$ such that $\lambda S_u = \lambda[e)$ and $\lambda S_v = \lambda[e)$. We have that $e \smile_\mu d$: this holds (by 1. and 2.) and this conflict cannot be inherited because otherwise also $u \smile v$ should be inherited. Hence we have $\lambda S_u = \lambda S_v$ by confusion freeness of $\mathcal{E}_1$ and as immediate consequence of Proposition 3.6, we have $S_u = S_v$, as required.

Remark 4.8 *As a consequence, composition of confusion free event structures is confusion free.*

Proposition 4.9 (Associativity) *Given $\mathcal{E}_1 : A, \mathcal{E}_2 : C, \mathcal{E}_3 : D$, if the interfaces allow the composition, we have that*

$$(\mathcal{E}_1 \| \mathcal{E}_2) \| \mathcal{E}_3 = \mathcal{E}_1 \| (\mathcal{E}_2 \| \mathcal{E}_3) =$$

Proof. The result follows from Proposition 4.4 and Proposition 4.10. □

4.5.3 Working by slices

Proposition 4.10 (Slices) *Let $\mathcal{E} = \mathcal{E}_1 : A \| \mathcal{E}_2 : C$. We have the following.*

- *If $\mathcal{S} \subseteq \mathcal{E}$ is a slice of $\mathcal{E}$, then there exist two slices $\mathcal{S}_1 \subseteq \mathcal{E}_1$ and $\mathcal{S}_2 \subseteq \mathcal{E}_2$ such that $\mathcal{S} = \mathcal{S}_1 : A \| \mathcal{S}_2 : C$.*
- *If $\mathcal{S}_1 \subseteq \mathcal{E}_1$ and $\mathcal{S}_2 \subseteq \mathcal{E}_2$ are slices, then $\mathcal{S} = \mathcal{S}_1 \| \mathcal{S}_2$ is a slice of $\mathcal{E}$.*

By reducing composition to composition of conflict free event structures, we can easily prove associativity.

5 Discussion

5.1 Relating with standard event structure composition

In this section we want to argument that the abstract machine we have defined produces the same result as a "standard" approach to event structure composition. To do this, we choose a specific synchronization algebra, which is that defined in [VY06].

The typing defined by Varacca and Yoshida guarantees that the composition preserves confusion freeness, and allows the interpretation of a linear fragment of the π calculus.

The labelling induced by their typing is easily seen as a case of the labelling we define here, hence in particular we can apply our machine.

We have that

Proposition 5.1 *If $\mathcal{E}_1, \mathcal{E}_2$ are confusion free event structures typed in the sense of [VY06], they are also typed in the sense defined here, and their composition $\mathcal{E}_1 \| \mathcal{E}_2$ as defined here is isomorph to the parallel composition as defined in [VY06].*

The details are given in [Pic06]

Let as briefly resume what a "standard approach" looks like.

5.1.1 Parallel composition of event structures

A more standard definition for the parallel composition of event structures is that used [VY06], based on the following ingredients:

1. fix through a synchronization algebra the events which will synchronize and those which will not. Two events synchronize if they have dual labels (for example one event has label a and the other $\overline{a}$);
2. build the categorical product of the event structures
3. discard some events, and everything above them:
   (a) discard all the events of the product which are generated from pair which are not able to synchronize because they do not have matching labels
   (b) discard all the events of the product which are generated from a single private event: these are events which are private but not "consumed".

5.2 Linear strategies with parallel composition are a sub-class of typed event structures

Linear strategies as introduced in [Gir01] and extended to dag's in [FM05] are labelled dag's. The labels are taken in

$$\sum_{i \in \mathcal{P}_{fin}(\mathbb{N})} N_i = \{(\alpha, i) : \alpha \in N and\ i\ \in \mathcal{P}_{fin}(\mathbb{N})\}$$

where N are the strings of natural numbers.

The labelling satisfies a certain discipline, which in particular satisfies all the constraints in Definition 3.4.

As for composition, the machine introduced here extends the LAM machine defined in [Fag02, FM05] to implement the composition of linear strategies. The new machine has the same behaviour of the LAM when restricted to strategies. This in particular means that there is a morphism from strategies to typed event structures, which preserves the paralle composition.

More precisely, strategies composition decomposes into parallel composition plus hiding, where parallel composition is the operation we have described here, and the hiding concerns the τ actions.

5.3 More comments and future work

The dynamics The machine we have presented makes it immediate what is going on when composing two labelled event structures $\mathcal{E}_1, \mathcal{E}_2$: we merge together the structure (events, order and conflicts) of $\mathcal{E}_1, \mathcal{E}_2$ to create a new event structure $\mathcal{E}$. The dynamics appears the same as that which takes place when composing strategies, λ-terms or Linear Logic proof-nets.

Bridging Game Semantics and event structures. In future work, we plan to use event structure as a guide to generalize the definition of strategies. We hope to build on the work on event structures to extend the approach of Game Semantics, in order to deal with non determinism, concurrency and process calculi.

Event structures and proof-nets This work meet also another line of research is bringing together graph strategies and proof-nets, which are a graph representation of proofs introduced by Linear Logic [Gir87] and which are powerful tool for the analysis of normalization. In particular, they have been a fertile tool in the study of functional programming, in particular for optimization. Observe that proof-net normalization is performed via local rewriting rules.

We see event structure as a form of multiplicative-additive proof-nets, and hope to be able to apply some of the technology developed for proof-nets. For example, a key notion in proof-nets is that of correctness criterion, which states that there are no cyclic path, for a certain definition of path which is sensitive to the polarity of the nodes. The correctness criterion has a crucial role in guaranteeing that the normalization (composition) works, and in fact it guarantees that there are no deadlock. We intend to investigate if a similar notion could be used on event structure, for an opportune typing, to guarantee that there are no deadlocks.

Acknowledgments

This work was motivated from discussion with Nabuko Yoshida and Daniele Varacca.

We are in debt with Daniele Varacca for many explanations, comments, and suggestions. We are grateful to Martin Hyland, Emmanuel Beffara, and Pierre-Louis Curien for interesting discussions.

We also wish to thank the referees for many usefull remarks and suggestions.

References

[AM99] S. Abramsky and P.-A. Mellies. Concurrent games and full completeness. In *Proceedings 15th Annual Symposium on Logic in Computer Science*, 1999.

[CF05] P.-L. Curien and C. Faggian. L-nets, strategies and proof-nets. In *CSL 05 (Computer Science Logic)*, LNCS. Springer, 2005.

[Fag02] C. Faggian. Travelling on designs: ludics dynamics. In *CSL'02 (Computer Science Logic)*, volume 2471 of *LNCS*. Springer Verlag, 2002.

[FM05] C. Faggian and F. Maurel. Ludics nets, a game model of concurrent interaction. In *Proc. of LICS'05 (Logic in Computer Science)*. IEEE Computer Society Press, 2005.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir01] Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11:301–506, 2001.

[GM04] Dan R. Ghica and Andrzej S. Murawski. Angelic semantics of fine-grained concurrency. In *FOSSACS*, 2004.

[HS02] M. Hyland and A. Schalk. Games on graphs and sequentially realizable functionals. In *LICS 02*, pages 257–264. IEEE, 2002.

[Lai05] J. Laird. A game semantics of the asynchronous pi-calculus. In *Concur 05*, volume 3653 of *LNCS*, 2005.

[Mel04] P.-A. Mellies. Asynchronous games 2 : The true concurrency of innocence. In *CONCUR 04*, volume 3170 of *LNCS*. Springer Verlag, 2004.

[MW05] G. McCusker and M. Wall. Categorical and game semantics for scir. In *Galop 2005*, pages 157–178, 2005.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Event structures and domains 1. *Theoretical Computer Science*, 13:85–108, 1981.

[Pic06] M. Piccolo. Event structures and strategies. Master's thesis, Dip. Matematica Pura e Applicata, Universitá di Padova, 2006.

[SPP05] A. Schalk and J.J. Palacios-Perez. Concrete data structures as games. In *CTCS 04*, volume 122 of *Electr. Notes Theor. Comput. Sci.*, 2005.

[VY06] D. Varacca and N. Yoshida. Typed event structures and the pi-calculus. In *MFPS*, 2006.

[Win87] G. Winskel. Event structures. *Advances in Petri Nets 1986, Part II*, volume 140 of LNCS:561–576, 1987.

[WN95] G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science*, volume 4, chapter Models for concurrency. Clarendon Press, 1995.

Formal Verification of Object-Oriented Graph Grammars Specifications

Ana Paula Lüdtke Ferreira [1]

Universidade do Vale do Rio dos Sinos
São Leopoldo, Brazil

Luciana Foss, Leila Ribeiro [2]

Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

Concurrent object-oriented systems are ubiquitous due to the importance of networks and the current demands for modular, reusable, and easy to develop software. However, checking the correctness of such systems is a hard task, mainly due to concurrency and inheritance aspects. In this paper we present an approach to the verification of concurrent object-oriented systems. We use graph grammars equipped with object oriented features (including inheritance and polymorphism) as the specification formalism, and define a translation from such specifications to Promela, the input language of the SPIN model checker.

Key words: Graph grammars, object orientation, model checking.

1 Introduction

Software development techniques have evolved over the years to deal with current developing demands. The paradigms on which those techniques are based (especially objects, events and concurrency) make the modeling and coding processes easier. However, testing and validation of such systems became more complex, mainly due to the non-deterministic behavior of multiple processes competing for the same resources. Object-oriented systems features like inheritance, polymorphism and dynamic binding of method calls also make static analysis of limited use in the validation process. Thus, correctness assurance of concurrent object-oriented systems is a difficult task.

[1] Email:anap@unisinos.br
[2] Email:{lfoss,leila}@inf.ufrgs.br

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The first step to enable correctness proofs of a system is to provide a formal specification of it. The semantic model can be analyzed to check whether the desired properties hold. The choice on which specification language to use depends on the application characteristics, but also on the development paradigm chosen. We suggest that, if an object-oriented development process is followed, an adequate formal specification formalism should offer compatible constructs. Object-oriented graph grammars were first presented in [8] as an extension of the algebraic single-pushout approach [13] to encompass object-oriented features such as inheritance, polymorphism, and dynamic binding. The main contribution of this paper is to present a verification method for specifications written as object-oriented graph grammars. This method is based on a translation of such specification to Promela programs. Promela is the input language of the SPIN model checker [10]. Particularly, features like inheritance, polymorphism, and dynamic binding will also be faithfully encoded in Promela.

Our approach for object-oriented verification is a straightforward, translation-based one, and differs from approaches relying on analysis of the specification languages *per se*, such as [2], [12], [1], [15], [16]. We follow a line of work presented in [5] for graph grammars without object-oriented features. However, besides building the translations for inheritance, polymorphism and dynamic binding features, we also do the translation based on a well defined observational semantics [7] which interprets object-oriented graph grammars computations from the object-oriented paradigm view. This article is structure as follows: Sec. 2 presents the main components of object-oriented graphs and grammars. Sec. 3 presents the guidelines followed for the definition of a formal translation from object-oriented graph grammars specifications into Promela programs, followed by an example shown in Sec. 4. The running example is a classic problem in the theory of concurrency, known as the *Dining Philosophers* problem. Final remarks are presented in Sec. 5.

2 Object-oriented graph grammars

Object-oriented systems consist of instances of previously defined classes having an internal structure defined by attributes and communicating among themselves through message passing. An object-oriented system state consists of objects, together with a set of messages yet to be consumed. Messages are the triggers of method executions, and their implementation may be redefined within derived classes. Classes and messages are modeled together in a *class-model graph*. Formally, class identifiers are graph nodes, attributes are modeled as hyperarcs (that is, each class may be connected to many others via an attribute hyperarc), and messages are also modeled as hyperarcs (in which the target is the destination of the message, and sources are its parameters). The inheritance hierarchy is defined by imposing a *strict relation* among the graph nodes. A strict relation is an irreflexive, acyclic, functional relation,

with the additional property that there is no infinite chain of elements connected through it (the reflexive and transitive closure of a strict relation is a partial order [7]). Message hyperarcs also possess an order structure, which reflects the possibility of a derived object to override inherited methods of its superclasses. A set carrying a reflexive and transitive closure of a strict relation is called a strict ordered set.

Definition 2.1 [Class-model graph] A class-model graph is a tuple $\langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ where $V_{\sqsubseteq} = \langle V, \sqsubseteq_V^*\rangle$ is a strict ordered set of vertices, $E_{\sqsubseteq} = \langle E, \sqsubseteq_E^*\rangle$ is a strict ordered set of (hyper)edges, $L = \{\text{attr}, \text{msg}\}$ is an unordered set of two edge labels, $src, tar : E \to V^*$ are monotonic order-preserving functions, called respectively *source* and *target* functions, $lab : E \to L$ is the edge *labeling* function, such that the following constraints hold:

- *Structural constraints:* for all $e \in E$, the following holds:
  (i) if $lab(e) = \text{attr}$ then $src(e) \in V$ and $tar(e) \in V^*$, and
  (ii) if $lab(e) = \text{msg}$ then $src(e) \in V^*$ and $tar(e) \in V$.
- *Order relations constraints:* for all $e \in E$, the following holds:
  (i) if $(e, e') \in \sqsubseteq_E$ then $lab(e) = lab(e') = \text{msg}$,
  (ii) if $(e, e') \in \sqsubseteq_E$ then $src(e) = src(e')$,
  (iii) if $(e, e') \in \sqsubseteq_E$ then $(tar(e), tar(e')) \in \sqsubseteq_V^+$, and
  (iv) if $(e', e) \in \sqsubseteq_E$ and $(e'', e) \in \sqsubseteq_E$, with $e' \neq e''$, then $(tar(e'), tar(e'')) \notin \sqsubseteq_V^*$ and $(tar(e''), tar(e')) \notin \sqsubseteq_V^*$.

Sets $\{e \in E \mid lab(e) = \text{attr}\}$ and $\{e \in E \mid lab(e) = \text{msg}\}$ are denoted by $E|_{\text{attr}}$ and $E|_{\text{msg}}$, respectively.

Structural constraints assure that hyperarcs modeling attributes and messages have the correct source and targets. Inheritance and overriding hierarchies are made explicit by imposing that graph nodes (i.e., classes) and message edges (i.e., methods) are strict ordered sets. Only single inheritance is allowed, since $\sqsubseteq_V$ is required to be a function. The relation between message arcs, $\sqsubseteq_E$, establishes which methods are overridden within the derived object, by mapping them. The restrictions applied to $\sqsubseteq_E$ ensure that methods are redefined consistently, i.e., only message arcs can be mapped (i), their parameters are the same (ii), the method being redefined is located somewhere (strictly) above in the class-model graph (under $\sqsubseteq_V^+$) (iii), and only the closest message with respect to relations $\sqsubseteq_V$ and $\sqsubseteq_E$ can be redefined (iv).

Example 2.2 The class-model graph in Figure 1 depicts an object-oriented system structure for the Dining Philosophers problem. Graph nodes represent classes: *Philosopher*, which is derived into two different types: *Left-HandedPhilosopher* and *Right-HandedPhilosopher* (the inheritance relation is pictured as a dotted arrow); *Fork*, to represent the shared resources the philosophers are competing upon; *Table*, to model both the place where the philosophers sit and from where forks can be picked up; and *ForkHolder*, which can be either a *Philosopher* or a *Table*. The attributes are the information

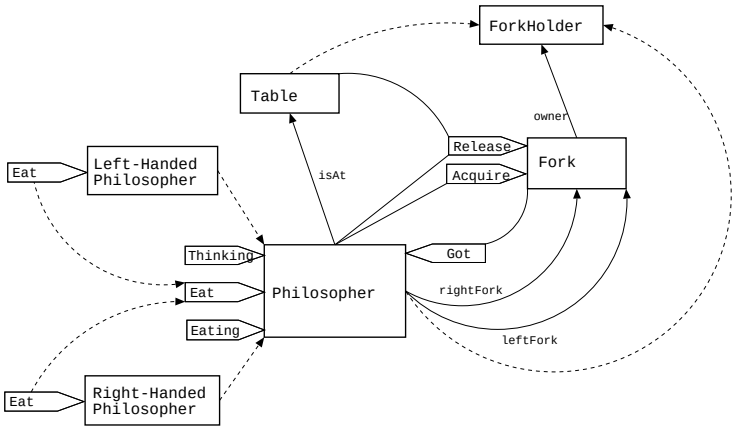


Figure 1. Class-model graph for the Dining Philosophers problem.

the elements must possess to compute correctly: a *Philosopher* sits at a *Table*, has a left and a right *Fork* to get in order to eat; a *Fork* has an owner, which is a *ForkHolder*. Messages stand for the actions performed by the actors in the program. A *Fork* can be acquired by a *Philosopher*, and released by a *Philosopher* to a *Table*. A *Philosopher* can be Thinking, Eating, or receive a message Eat, which sends him to the process of acquiring his forks, and a message Got, to notify that a *Fork* has been acquired. Left-handed and right-handed philosophers override message Eat, which is indicated by the lines connecting both hyperarcs.

Class-model graphs can be used as typing structures for states of object-oriented systems. Before defining such states, that will be object-oriented graphs, we will define how to map a graph into a class-model graph, and then impose restrictions to make this mapping compatible with inheritance. Based on the inheritance and overriding relations, we define auxiliary functions that, given a class identifier (node), return the sets of attributes (inherited or not) of this class, and the sets of messages (method triggers) that this class may receive.

Definition 2.3 [$\mathcal{C}$-typed graph] A $\mathcal{C}$-typed graph $G^{\mathcal{C}}$ is a tuple $\langle G, t, \mathcal{C}\rangle$, where $\mathcal{C} = \langle V_{\mathcal{C}_\sqsubseteq}, E_{\mathcal{C}_\sqsubseteq}, L, src_{\mathcal{C}}, tar_{\mathcal{C}}, lab_{\mathcal{C}}\rangle$ is a class-model graph, $G = \langle V_G, E_G, src_G, tar_G\rangle$ is a hypergraph, and t is a pair of total functions $\langle t_V : V_G \to V_{\mathcal{C}}, t_E : E_G \to E_{\mathcal{C}}\rangle$ such that $(t_V^* \circ src_G) \sqsubseteq_{V_{\mathcal{C}}^*} (src_{\mathcal{C}} \circ t_E)$, and $(t_V^* \circ tar_G) \sqsubseteq_{V_{\mathcal{C}}^*} (tar_{\mathcal{C}} \circ t_E)$. Moreover, we define:

- the *attribute set function* $attr_G : V_G \to 2^{E_G}$ returns for each vertex $v \in V_G$ the set of attribute edges with source v;
- the *message set function* $msg_G : V_G \to 2^{E_G}$ returns for each vertex $v \in V_G$ the set of message edges with target v.
- the *extended attribute set function*, $attr_{\mathcal{C}}^* : V \to 2^E$, where $attr_{\mathcal{C}}^*(v) = \{e \in E \mid lab(e) = \text{attr} \wedge src(e) \in \uparrow v\}$, and $\uparrow v$ is the set of all superclasses of v.
- the *extended message set function*, $msg_{\mathcal{C}}^* : V \to 2^E$, where $msg_{\mathcal{C}}^*(v) = \{e \in$

$E|_{\text{msg}} \mid tar(e) \in \uparrow v \wedge \neg\exists e' \in E|_{\text{msg}} : tar(e') \in \uparrow v \wedge e' \sqsubseteq_E e\}$.

$\mathcal{C}$-typed graphs reflect the inheritance of attributes and methods from the object-oriented paradigm. Notice that they are ordinary hypergraphs typed over a class-model graph. However, the typing morphism is more flexible than the traditional one [3]: a $\mathcal{C}$-typed graph edge e can be incident to any $\mathcal{C}$-typed graph node v as long as its typing edge $t_E(e)$ (in $\mathcal{C}$) is incident to a node type v' (also in $\mathcal{C}$), such that $t_V(v)$ and v' are connected by the underlying order relation (i.e., $t_V(v) \sqsubseteq^*_{V_\mathcal{C}} v'$). This definition reflects the fact that an object can use any attribute belonging to one of its primitive classes, since it was inherited when the class was specialized.

The *extended attribute set function* returns the set of all attribute arcs whose source is v or any other vertex to which v connected via the inheritance relation $\sqsubseteq^*_V$. The *extended message set function* returns all messages an object of a specific type may receive. Notice that message redefinition within objects, expressed by the overriding relation $\sqsubseteq^*_E$ on the class-model graph, is taken into account, since only the redefined methods can be seen within the scope of a specialized class.

For a $\mathcal{C}$-typed graph $\langle G, t, \mathcal{C}\rangle$, let the total function $t^*_E : 2^{E_G} \to 2^{E_\mathcal{C}}$ be the extension of the typing function to edge (or node) sets. Notation $t^*_E|_{\text{msg}}$ and $t^*_E|_{\text{attr}}$ will be used to denote the application of t^*_E to sets containing exclusively message and attribute (respectively) hyperarcs. Now we can present a definition of the kind of graph which represents an object-oriented system.

Definition 2.4 [Object-oriented graph] Let $\mathcal{C}$ be a class-model graph. A $\mathcal{C}$-typed graph $\langle G, t, \mathcal{C}\rangle$ is an *object-oriented graph* if and only if all squares in the diagram below (in **Set**) commute. If, for each $v \in V_G$, the function $t^*_E|_{\text{attr}}(attr_G(v))$ is injective, $G^\mathcal{C}$ is said a *strict* object-oriented graph. If $t^*_E|_{\text{attr}}(attr_G(v))$ is also surjective, $G^\mathcal{C}$ is called a *complete* object-oriented graph.

$$\begin{array}{ccccc} 2^{E_G} & \xleftarrow{msg_G} & V_G & \xrightarrow{attr_G} & 2^{E_G} \\ \downarrow{\scriptstyle t^*_E|_{\text{msg}}} & & \downarrow{\scriptstyle t_V} & & \downarrow{\scriptstyle t^*_E|_{\text{attr}}} \\ 2^{E_\mathcal{C}} & \xleftarrow{msg^*_\mathcal{C}} & V_\mathcal{C} & \xrightarrow{attr^*_\mathcal{C}} & 2^{E_\mathcal{C}} \end{array}$$

The left square on the diagram of Def. 2.4 ensures that a message edge can only target an object if it is typed over one of the edges returned by the extended message set function applied to the object type. It means that the only messages allowed are the least ones in the redefinition chain to which the typing message belongs. This is compatible with the notion of *dynamic binding*, since the method actually called by any object is determined by the actual object present at a certain computation state. Injectivity of all $t^*_E|_{\text{attr}}(attr_G(v))$, $v \in V_G$, expresses that all attribute arcs are typed differently (i.e., an object has no exceeding attribute). Surjectivity means that *all* attributes defined on all levels along the class-model graph (via the inheritance relation on nodes) are present. The definition of a complete object-oriented graph is coherent

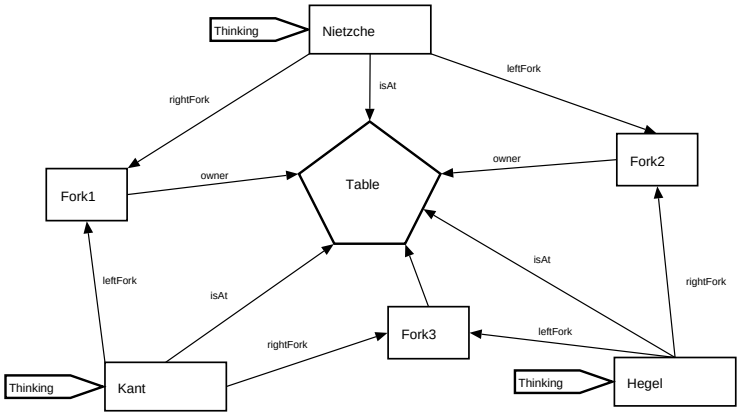


Figure 2. The initial graph for the Dining Philosophers problem.

with the notion of inheritance within the object-oriented framework, since an object inherits all attributes, and exactly those, from its primitive classes.

Example 2.5 Figure 2 shows a complete object-oriented graph, typed over the class-model graph portrayed in Figure 1. Let the three elements called Kant, Hegel and Nietzsche be *Right-HandedPhilosopher*, and the other elements be typed as their names indicate. According to the typing class-model graph, a *Right-HandedPhilosopher* has no attribute at all, and also does not receive a message typed as Thinking. However, since its parent class *Philosopher* has those arcs connected to it, they can be connected to any derived object, thus allowing inheritance of elements. To see that, consider the attribute *isAt* of the right-handed philosopher Kant. The edge it is mapped to by the typing morphism has as source an element of class *Philosopher*, and so $(t^*_V \circ src_G)(\text{isAt}) = \textit{Right-HandedPhilosopher} \sqsubseteq_{V^*_{\mathcal{C}}} \textit{Philosopher} = (src_{\mathcal{C}} \circ t_E)(\text{isAt})$, and the morphism is allowed.

Relationships between $\mathcal{C}$-typed graphs can be described by morphisms.

Definition 2.6 [$\mathcal{C}$-typed graph morphism] Let $G^{\mathcal{C}}_1 = \langle G_1, t_1, \mathcal{C}\rangle$ and $G^{\mathcal{C}}_2 = \langle G_2, t_2, \mathcal{C}\rangle$ be two $\mathcal{C}$-typed graphs typed over the same class-model graph $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$. A *$\mathcal{C}$-typed graph morphism* $h : G^{\mathcal{C}}_1 \to G^{\mathcal{C}}_2$ between $G^{\mathcal{C}}_1$ and $G^{\mathcal{C}}_2$, is a pair of partial functions $h = \langle h_V : V_{G_1} \to V_{G_2}, h_E : E_{G_1} \to E_{G_2}\rangle$ such that the diagram below (in category **SetP**) commutes, for all elements $v \in dom(h_V)$, $(t_{2V} \circ h_V)(v) \sqsubseteq_{V_{\mathcal{C}}} t_{1V}(v)$, and for all elements $e \in dom(h_E)$, $(t_{2E} \circ h_E)(e) \sqsubseteq_{E_{\mathcal{C}}} t_{1E}(e)$. If $(t_{2E} \circ h_E)(e) = t_{1E}(e)$ for all elements $e \in dom(h_E)$, the morphism is said to be *strict*.

$$\begin{array}{ccccc} E_{G_1} & \xleftarrow{h_E?} & dom(h_E) & \xrightarrow{h_E!} & E_{G_2} \\ {\scriptstyle src_{G_1},tar_{G_1}}\downarrow & & & & \downarrow{\scriptstyle src_{G_2},tar_{G_2}} \\ V^*_{G_1} & & \xrightarrow{\quad h^*_V \quad} & & V^*_{G_2} \end{array}$$

A graph morphism is a mapping which preserves hyperarcs sources and

targets. A typed graph morphism also preserves (node and edge) types. Ordinary typed graph morphisms [3], however, cannot describe correctly morphisms on object-oriented systems because the existing inheritance relation among objects causes that actions available for objects of a certain kind are valid to *all* objects derived from it. So, an object can be viewed as not being uniquely typed, but having a type *set* (namely, the set of all types it is connected via the inheritance relation). Defining a graph morphism compatible with the underlying order relations assures that polymorphism can be applied consistently.

The behavior of object-oriented systems (implementation of methods) will be modeled by rules, in which the left- and right-hand sides are object-oriented graphs. Besides structural restrictions (imposed by the fact that rules as $\mathcal{C}$-typed graph morphisms), some others are necessary to assure compatibility with the concepts of the object paradigm. Particularly, a rule left-hand side contains exactly one element of type message, and this particular message must be deleted by the rule application, i.e., each rule represents an object reaction to a message which is consumed in the process. This demand poses no unreasonable restriction, since systems may have many rules specifying reactions to the same type of message (non-determinism) and many rules can be applied in parallel if their triggers are present at an actual state and the referred rules are not in conflict [6]. At most one object having attributes will be allowed on the left-hand side of a rule, along with the requirement that this same object must be the target of the above cited message. This restriction implements the principle of *information hiding*, which states that the internal configuration (implementation) of an object can only be visible, and therefore accessed, by itself. The rule morphism must be invertible, to assure that an object does not have its type changed along the computation. Finally, there must be a bijection between the edges on both sides, and so an object does not gain or loose attributes as the computation evolves.

Object-oriented graph grammars are composed by a class-model graph, an initial state (a complete object-oriented graph) and a set of object-oriented rules.

Example 2.7 An object-oriented graph grammar for the Dining Philosophers problem is presented in Figures 3, 4, and 5. All object-oriented rules left- and right-hand sides are object-oriented graphs typed over the class-model graph portrayed in Figure 1. However, in order to make the presentation clearer, all nodes and edges are named after their types, making the typing morphism explicit.

The semantics of an object-oriented graph grammar is based on rule applications. Matches and direct derivations are defined in the same way as the single-pushout approach: a match is a total $\mathcal{C}$-typed graph morphism, and a direct derivation is the pushout of the match and rule arrows in the category of object-oriented graphs and their morphisms [9]. Instead of using the usual

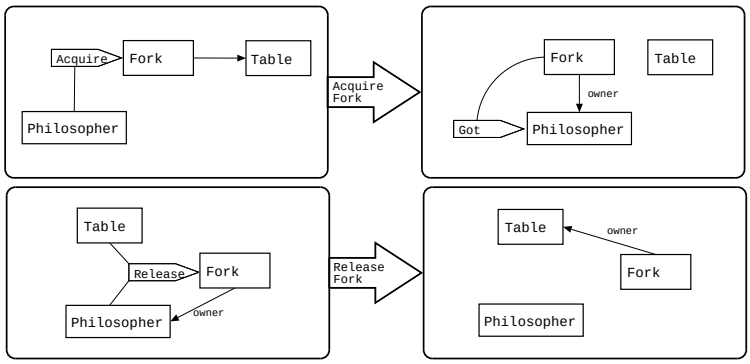


Figure 3. *Fork* rules for the Dining Philosophers problem.

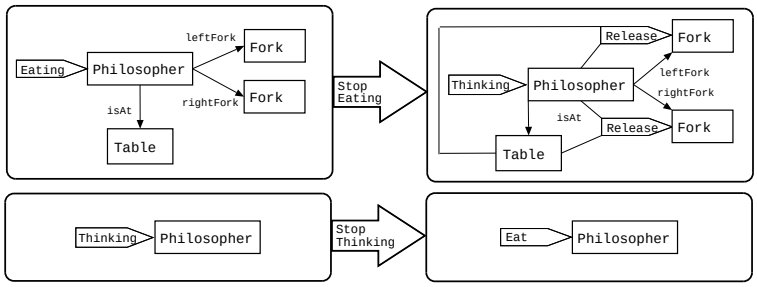


Figure 4. *Philosopher* rules for the Dining Philosophers problem.

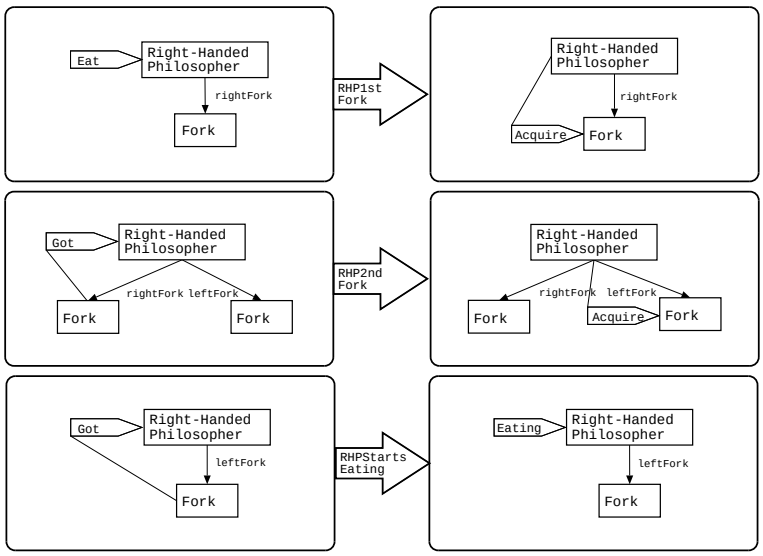


Figure 5. *Right-HandedPhilosopher* rules for the Dining Philosophers problem.

transition system induced by the application of rules starting at the initial graph of the system (states are graphs and transitions are graph morphisms), we defined an abstract semantics based on *observations*. This semantics holds information about events happening in a system (message exchange among objects), and forgets about system structure. Therefore, although we are not able to express properties based on object states, we are still allowed to investigate properties of objects based on how they respond to the rules applied to

them. The abstract semantics is given by a labeled transition system where its states are the graphs generated by rule applications in the grammar, and the transition between two states is labeled with the name of the rule applied together with the object identity the rule was applied to.

Definition 2.8 [Object-oriented graph grammar transition semantics] Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C} \rangle$ be an object-oriented graph grammar. The transition semantics of $\mathcal{G}$ is given by the labeled transition system $\mathcal{T}^{\mathcal{G}} = \langle S, s_0, L, \rightarrow \rangle$, where $S = \{G^{\mathcal{C}} \mid I^{\mathcal{C}} \Rightarrow^* G^{\mathcal{C}}\}$ is the set of states, $s_0 = I^{\mathcal{C}}$ is the initial state, $L = \{\langle p, o \rangle \in P^{\mathcal{C}} \times V_G \mid G \in S^{\mathcal{T}} \wedge p \in \Pi^E_{t_G(o)}\}$, is the set of labels, where $\Pi^E_{t_G(o)}$ is the grammar set of productions that can be applied to an object of type $t_G(o)$, $\rightarrow$ is the transition relation, and object-oriented graphs $G^{\mathcal{C}}$ and $H^{\mathcal{C}}$ are related under $\rightarrow$ if there is an object-oriented graph production $r : L^{\mathcal{C}} \rightarrow R^{\mathcal{C}} \in P^{\mathcal{C}}$, an object-oriented match $m : L^{\mathcal{C}} \rightarrow G^{\mathcal{C}}$ such that $G^{\mathcal{C}} \overset{r,m}{\Rightarrow} H^{\mathcal{C}}$.

3 Translation

The input language of SPIN [10] is Promela (PROtocol/PROcess MEta LAnguage) which is a specification language to model state transition systems. The complete translation algorithm is rather long and will be presented here informally. Objects are modeled as Promela processes, and message exchange between objects through asynchronous communication channels. To overcome the FIFO policy of buffered channels in Promela, the same solution from [5] is used: a local buffer is used to "shuffle" received messages and so maintain the non determinist rule application semantics. The inheritance relation appears as a global array visible to any program element. Subclass polymorphism is coded through an inspection in this array, to assure that rule matches only occur if the matched elements are correctly related. Dynamic binding is implemented as a message dispatch procedure within each object process definition. Differently from classic object-oriented programming languages implementations [14], where a virtual table determines which method should be called in execution time, our approach uses a little computational reflection [17], in the sense that each object (process) is aware of its own type, and that information is made available to other entities when they have access to the object (as an attribute, or as a message parameter). So an object can decide, at run time, the adequate message to send based on the actual type of the message receiver.

Each initial graph node is transformed into a process, having as parameters all the targets of its attributes (using an arbitrary total order imposed on each object attributes). Each initial message is put into the proper object channel, together with its parameters (the sources of each message arc in the initial graph). Therefore, targets of attribute edges become processes parameters, and message parameters become processes local variables. A process code (the object behaviour) consists of an infinite loop that continuously tests

(non deterministically) if either a new message has arrived at the object main channel — in which case the message is retrieved and placed in some empty slot of the local message buffer — or if there is a message in the local buffer waiting to be consumed. In the latter case, the message is atomically retrieved from the buffer and the production it refers to is applied. In case neither the object channel nor the local message buffer contain any messages to be consumed, the process will jump to the beginning of the main loop, and stay blocked until a new message arrives.

The matching procedure tests if (i) all attributes are typed correctly, and (ii) all attribute values are correct. For instance, consider the first rule in Figure 3. This match will only be possible if the holder of the attribute vertex (of type *Fork*) is an object typed as *Table*. If it is a *Philosopher*, the match will not occur, because those two elements are not related by inheritance (although they both derive from a *ForkHolder*). Type testing is performed by inspection on the aforementioned global inheritance array. Now, consider the second rule from that same figure. The match is possible only if the *Philosopher* passed as parameter of message Release is the same one holding the fork. Therefore, an equality test is carried on between objects which are sources or targets of distinct arcs.

The choice on which production to apply is performed by a conditional test for all rules to which a match (for the received message) exist. Since a conditional test in Promela has a non deterministic result if more than one conditional is true, the choice of which production to apply is also non deterministic, as required by the grammar semantics.

Rule application can be described as: (i) object attributes are modified according to the rule morphism; (ii) a global variable `event_RuleName` is set with the applied production name; (iii) the set of variables `event_`x, for all classes to which the type of the production attribute vertex is related by inheritance, are set to the object identity; (iv) finally, all messages appearing in the right-hand side of the applied production are created, and it is particularly relevant, since it is this procedure which performs dynamic binding. If no rule is applied (because no match were possible for any production implementing the received message), then the message is put back in the local buffer, and marked as inspected. An already inspected message will not be retrieved for application until a new message arrives. Since only an object can change its own state, a match for this message could only happen after another production is actually applied. This procedure also helps to decrease the program state space for the verification process.

The right message to send is based on the type of the actual object which is receiving the message. Since the lower set of any node (respecting the inheritance hierarchy) is finite and does not change along the program execution, a conditional structure takes care of this. For instance, consider the second rule in Figure 4. A message Eat is sent to a *Philosopher*. This message, however, is redefined by all *Philosopher* subclasses, so one must know the element type to

send the correct message. The code generated is illustrated by the following pseudo-code:

```
if (receiving object message channel is not full)
  if
    . receiving object type is a Philosopher ->
          send message Eat for the Philosopher
    . receiving object type is a Left-HandedPhilosopher ->
          send message Eat for the Left-HandedPhilosopher
    . receiving object type is a Right-HandedPhilosopher ->
          send message Eat for the Right-HandedPhilosopher
```

The whole rule application procedure is performed atomically. Therefore, from the time a message is taken out of the local buffer to the time a rule application is completed — by either applying the rule or by putting the message back to the local buffer, if no match exist for that rule — no other process can interleave with that execution, because of the `atomic` keyword. The atomicity of the rule application process is necessary, to mimic the way rules are applied in the graph grammar, where the whole matching and application procedure is performed in a single step. Furthermore, if interleaving was allowed, errors could appear: if a process is stopped between finding a match for a production and the application of that production, meanwhile the state graph could be altered in a way that turns the rule application impossible; therefore a match/application procedure is considered a critical region of any object behaviour.

4 Verification

Property verification in SPIN can be done using a multiplicity of methods, among which there is LTL [11] property verification. Meaningful events to verification of the Dining Philosophers problem can be stated, for instance, as "philosopher X starts to eat", or "fork Y is grabbed by a philosopher". We will use the already presented object-oriented graph grammar for the Dining Philosophers problem as the running example. For reasons of space, we will verify only the liveness property stated as "anytime a philosopher decides to eat, he eventually does so". We will show that this property is false in the provided model.

SPIN performs model-based verification, which means that properties can only be defined over states, and not over transitions. The translation we propose defines a set of global variables to allow verification over events: (i) one global variable for each class belonging to the class-model graph over which the grammar is typed, to identify the last object of that type that had a production applied to it (if a message is received by an object, and consumed by some rule application, then the object identity is assigned to the respective variable), and (ii) one global variable to identify which *rule* was applied, and it is updated every time such action occurs. Notice that rule

application is not equivalent to message consumption. Although each rule application corresponds exactly to a response to a received message, there can be multiple (different) rules implementing actions for the same type of message. This variable is necessary if one is interested in verify possible orders in which rules can be applied.

The XSpin tool allows that propositions can be defined in a C-like way, using the preprocessor macro `#define`. Those properties can be defined in terms of the actual objects belonging to the system initial graph. Since we are only interested in the behaviour of the philosophers, a proposition to identify each of them is defined as in *#define isKant (event_Philosopher == Kant)*. Propositions for events of interest can be defined using the global variable for rule identification, as in *#define aPhilWantsToEat (event_RuleName == rule_Philosopher_StopThinking)* or in *#define aPhilStartsToEat (event_RuleName == rule_Philosopher_StartsEating)*. In order to discover if a known event occurs with a specific object, propositions such as *#define philKantWantsToEat (isKant && aPhilWantsToEat)* and *#define philKantStartsToEat (isKant && aPhilStartsToEat)* can be defined.

Using the propositions defined above, LTL properties about the system behaviour can be written. Property "anytime a philosopher decides to eat, he eventually does so" can be stated, for philosopher Kant as *[] (philKantWantsToEat − > <> philKantStartsToEat)* where symbols `<>` and `[]` stand for the usual linear temporal logic quantifiers ◊ (eventually) and □ (always). For all philosophers, it can be stated as *[] ((philKantWantsToEat − > <> philKantStartsToEat) && (philHegelWantsToEat − > <> philHegelStartsToEat) && (philNietzscheWantsToEat − > <> philNietzscheStartsToEat))*.

This last property is not true within the model provided. Figure 6 shows a graphical counterexample (taken from the model checker output, and generated by the system developed in [4]) for them. The counterexample shows three philosophers (Nietzsche, Hegel, and Kant) and their respective forks. The processes are indicated by the horizontal lines, and the arrows indicate the messages arriving and departing from each process. Notice that a deadlock situation is set: each philosopher have grabbed one fork, and a message was sent to the other fork in an attempt to acquire it. However, since each fork now has a philosopher owning it, rule `AcquireFork` cannot ever be applied again, and all philosophers will wait forever.

5 Conclusions

Object-oriented graph grammars provide a graph-based specification framework for object-oriented systems, where special partial orders represent the inheritance and overriding hierarchies, making polymorphism and dynamic binding built-in features of the formalism.

We have presented a (sketch) translation from object-oriented graph grammars specifications into Promela programs. All object-oriented features are

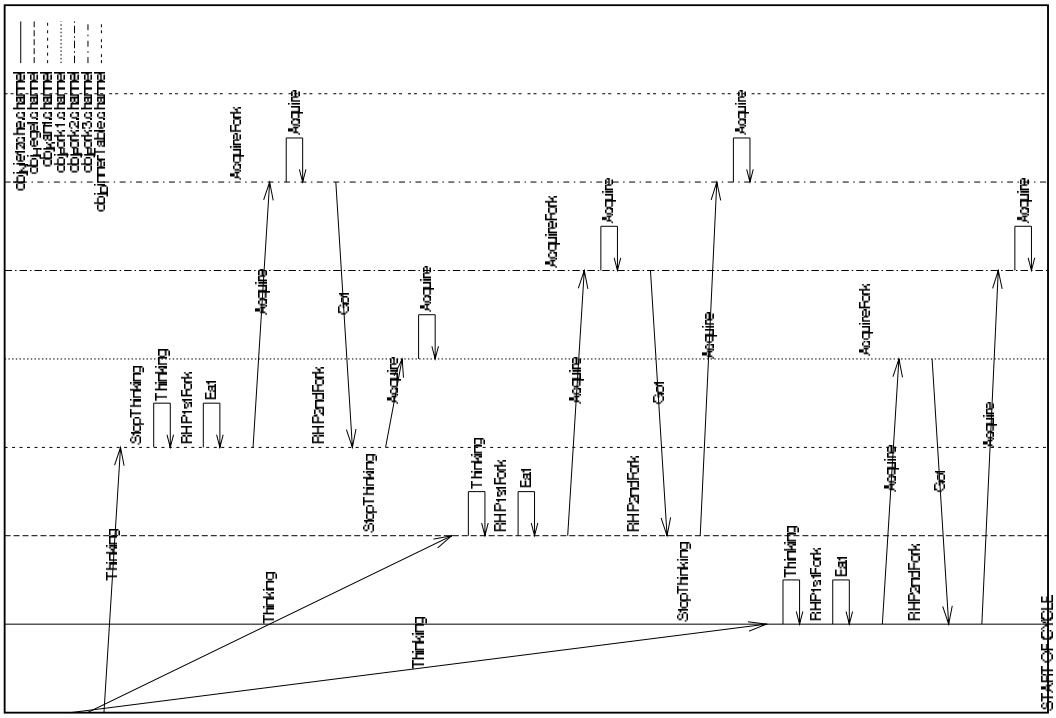


Figure 6. Counterexample of the absence of deadlock property

translated into Promela: inheritance appears as a global array; polymorphism is implemented in the matching procedure through an inspection on this array; dynamic binding is implemented through the message dispatching mechanism, which checks the message receiver type to determine the correct message to send; information hiding and encapsulation appear naturally on the translation, since a single process implements each system object. The translation of graph rules applications establish the existence of matches before the rule can be applied, and the choice of which message to consume and which production to apply is non deterministic, as required by the defined grammar semantics.

We are not currently dealing with object creation and deletion, but it is a straightforward extension to this translation, which is currently being automatized (using and extension of the XML-based languages GXL and GTXL [18]). An effort can be done to customize the translation to the application characteristics, in order to reduce the produced state space.

The translation proposed is arguably semantically sound, in the sense that no graph system behaviour is removed or introduced by the translation. Even if there are states in the Promela program that do not correspond to any graph belonging to the grammar language, those states can always be translated if they are not part of a rule application procedure. If they are, they cannot interleave with any other process execution, since rule application is performed atomically, and hence it suffices to leave the atomic block for the Promela state can be translated to a graph state. A formal proof of this translation soundness is being prepared for publication. Finally, we have used a modeling for the

Dining Philosophers problem to illustrate how verification can be performed, and how errors can be found using our approach.

References

[1] Baldan, P., A. Corradini and B. König, *Verifying finite-state graph grammars: An unfolding-based approach*, in: *CONCUR*, 2004, pp. 83–98.

[2] Burkart, O. and Y.-M. Quemener, *Model-checking of infinite graphs defined by graph grammars*, Technical Report 995, IRISA — Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (1996).

[3] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamentae Informatica **26** (1996), pp. 241–265.

[4] dos Santos, O. M., "Verificação Formal de Sistemas Distribuídos Modelados na Gramática de Grafos Baseada em Objetos," Masters thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre (2004), 89p.

[5] Dotti, F. L., L. Foss, L. Ribeiro and O. M. Santos, *Verification of object-based distributed systems*, in: E. Najm, U. Nestmann and P. Stevens, editors, *Proceedings of the 6th IFIP TC6/WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, Lecture Notes in Computer Science **2884** (2003), pp. 261–275.

[6] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, *Algebraic approaches to graph transformation. Part II: single-pushout approach and comparison with double pushout approach*, , **1 (Foundations)**, World Scientific, Singapore, 1996 pp. 247–312.

[7] Ferreira, A. P. L., "Object-oriented graph grammars," PhD thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil (2005), 156p.

[8] Ferreira, A. P. L. and L. Ribeiro, *Towards object-oriented graphs and grammars*, in: E. Najm, U. Nestmann and P. Stevens, editors, *Proceedings of the 6th IFIP TC6/WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, Lecture Notes in Computer Science **2884** (2003), pp. 16–31.

[9] Ferreira, A. P. L. and L. Ribeiro, *Derivations in object-oriented graph grammars*, in: H. Ehrig, G. Engels, F. Parisi-Presicce and G. Rozenberg, editors, *Proceedings of the 2nd International Conference on Graph Transformations (ICGT 2004)*, Lecture Notes in Computer Science **3256** (2004), pp. 416–430.

[10] Holzmann, G. J., *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), pp. 1–17.

[11] Huth, M. R. A. and M. D. Ryan, "Logic in Computer Science: Modelling and reasoning about systems," Cambridge University Press, Cambridge, 2000.

[12] Koch, M., "Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems," PhD Thesis, Technische Universität Berlin, Berlin (1999).

[13] Löwe, M., "Extended Algebraic Graph Transformation," Ph.D. thesis, Technischen Universität Berlin, Berlin (1991).

[14] Pratt, T. W. and M. V. Zelkowitz, "Programming languages : design and implementation," Prentice-Hall, Upper Saddle River, 1996, 3 edition, 654p.

[15] Rensink, A., *Towards model checking graph grammars*, in: M. Leuschel, S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)* (2003), pp. 150–160.

[16] Rensink, A., *The GROOVE simulator: A tool for state space generation*, in: J. Pfalz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science **3062** (2004), pp. 479–485.

[17] Smith, B. C., "Reflection and Semantics in a Procedural Language," PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA (1982), mIT-LCS-TR-272.

[18] Winter, A., B. Kullbach and V. Riediger, *An overview of the GXL graph exchange language*, in: S. Diehl, editor, *International Seminar on Software Visualization*, Lecture Notes in Computer Science **2269** (2001), pp. 324–336.

Modeling and Verification of Reliable Messaging by Graph Transformation Systems [4]

László Gönczy [1], Máté Kovács [2] and Dániel Varró [3]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary

Abstract

Due to the increasing need of highly dependable services in Service-Oriented Architectures (SOA), service-level agreements include more and more frequently such non-functional aspects as security, safety, availability, reliability, etc. Whenever a service can no longer be provided with the required QoS, the service requester needs to switch dynamically to a new service having adequate service parameters after exchanging a sequence of messages. In the current paper, we first extend the core SOA metamodel with parameters required for reliable messaging in services. Then we model reconfigurations for reliable message delivery by graph transformation rules. Finally, we carry out a formal verification of the proposed rule set by combining analysis tools for graph transformation and labeled transition systems.

Keywords: Service Oriented Architecture, Graph Transformation, Reliable Messaging

1 Introduction

Service-Oriented Architectures (SOA) provide a flexible and dynamic platform for implementing business-critical services. The main business-level driver of the SOA paradigm is componentization, which raises the level of abstraction from objects to services in the design process of distributed applications. The main architectural-level driver of the SOA paradigm is to provide a common middleware framework for dynamic discovery, interaction and reconfiguration of service components independently of the actual business environment.

[1] Email: gonczy@mit.bme.hu
[2] Email: km432@hszk.bme.hu
[3] Email: varro@mit.bme.hu
[4] This work was partially supported by the SENSORIA European project (IST-3-016004).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Recently, the identification of non-functional parameters of services have been addressed by various XML-based standards related to web services (such as WS-Reliable Messaging, WS-Reliable Messaging Policies, etc.). *Reliable messaging between services* — where the delivery of a message can be guaranteed by the underlying platform by appropriate reconfiguration mechanisms — plays an important role in many of these standards, because of the growing need for asynchronous yet reliable Web service invocations. Despite the wide range of standards addressing the specification of these reliability service properties, currently only very experimental solutions exist in the industry (such as RAMP-Toolkit [18] by IBM or RM4GS [22] by a consortium led by Fujitsu-Siemens, Hitachi and NEC) that actually implement these reconfigurations in order to maintain the required level of reliability.

In the current paper, we conceptually follow [2] where a semi-formal platform-independent and a SOA-specific metamodel (ontology) was developed to capture service architectures on various levels of abstraction in a model-driven service development process. Furthermore, reconfigurations for service publishing, querying and binding were captured by graph transformation rules [6], which provides a visual yet formal, rule and pattern-based specification formalism widely used in various application areas. This combination of metamodeling and graph transformation rules fits well to a model-based development process for service middleware.

This paper extends the core metamodel defined in [2] (and overviewed in Sec.2) by a new package for reliable messaging (Sec. 3.2). Moreover, we provide new high-level reconfiguration primitives for reliable message delivery in the form of graph transformation rules (Sec. 4.2) by integrating dependability techniques [16]. Finally, we carry out a formal verification of the proposed rule set by combining various analysis tools: the state space of the graph grammar will be first explored by GROOVE [19] while the generated graph transition system is transformed into a fromat accepetd by the Labeled Transition System Analyzer (LTSA) tool where the automated formal verification of certain safety properties is carried out. Our aim is to provide a generic way to capture the dynamic fault-tolerant behavior of a SOA. In the current paper we used the reliable messaging as a case study for this.

Note that we first modeled reconfiguration rules for reliable messaging by graph transformation rules in an ad hoc way in [10]. The current paper extends that approach by formally verifying the rules by integrating analysis tools (Sec 5.3). In fact, we managed to find conceptual flaws in this initial rule set during verification, and thus the current paper already presents the corrected version of the rules (in Sec. 4.2).

2 Core SOA Metamodel

The main architectural concepts of the domain of service-oriented architectures are captured by a corresponding metamodel. An extract of the meta-

model of "core" SOA functionality is shown in Fig. 1. It is based on the metamodel presented in [2], with minor simplifications and modifications to keep the current paper better focused.

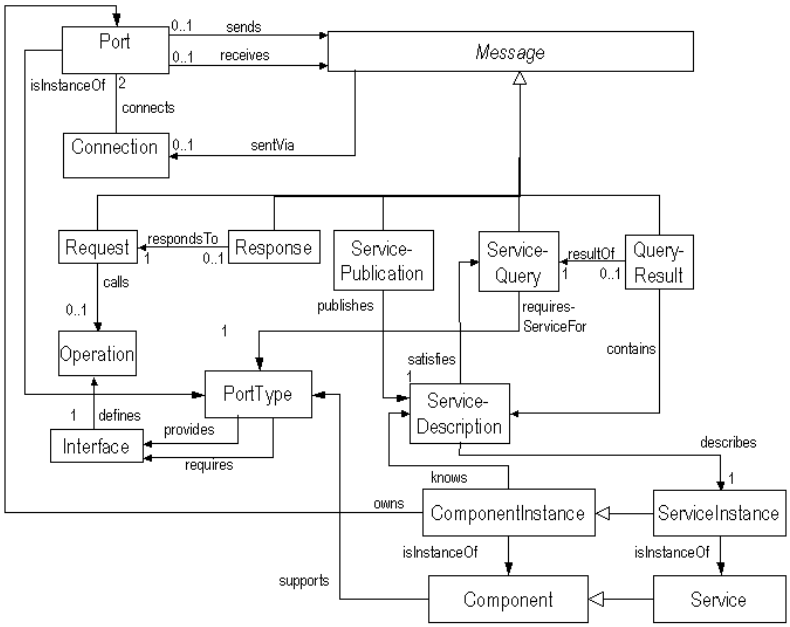


Fig. 1. Core metamodel of SOA

The core model to service-oriented architectures consists of the following main elements:

- A *component* is a basic "module" in the system which provides a service.
- A *service* is a set of functionalities with well-defined ports and interfaces. Note that in the paper, we merge the notions of service (and component) types and service instances into a single service (component) concept for the sake of simplicity.
- A *port* is the communication "endpoint" (with a set of abstract operations and messages) where a service can be accessed.
- A *connection* denotes a bidirectional channel between two ports at run-time.
- An *operation* is an "atomic" action with input and output messages. There can be multiple operations defined on the same port. .
- A *message* is a set of parameters with pre-defined subtypes such as *request*, *response*, *service publication*, *service query* and *query results*. For the current paper we treat these messages on an abstract level regardless of their actual subtypes. However, we will derive additional subtypes in Sec. 3.2 required for reliable messaging.
- A *service description* is a descriptor file containing all necessary information about the runtime cooperation with the service, such as description of port, operations, messages, etc.

3 Extensions for Reliable Messaging in Web Services

In this section, after a brief overview on capturing non-functional requirements in existing web service technologies, we extend the core SOA metamodel by non-functional attributes required for reliable messaging in order to provide a model-based solution.

3.1 Non-functional Requirements in Existing Web Service Technologies

While there are several initiatives to define the so-called "non-functional" properties of services, such as Web Services Modeling Ontology [25], W3C Web Services Architecture [24], DublinCore Metadata for ServiceDiscovery [5], the terminology is still ambiguous.

To illustrate the modeling of non-functional properties by a practical and simple example, hereby we present a model-based reconfiguration for reliable messaging to tolerate communication faults. As the consumers of the Web services are not aware of the details of underlying network protocol, the semantics of the message delivery has to be specified at the application level as requirements for reliable messaging. This needs a platform-independent representation of message attributes, which is reflected by a number of emerging standards [26,27]. Some reference implementations for popular application servers like IBM WebSphere or Apache Tomcat are available.

These industrial standards and initiatives usually suppose that the service provider signs a contract with each client about the Quality of Service, measured in terms such as average response time, minimal throughput, type of message delivery, etc. These contracts are typically identical for classes of similar clients (roles), for instance, Golden User, Business Partner, Individual Customer, etc. The runtime service instances send their messages according to these contracts, while additional information, including such non-functional aspects, is hidden from the application layer. As a consequence, it is not necessary to modify the original service clients on the consumers' side.

Additional information is handled by components aware of reliability attributes, called *"Reliable Message Endpoints"*. In technological terms, the header of SOAP envelopes is extended with some attributes by a "Reliable Message Endpoint" on the provider's side, which are then removed from the messages by another "Reliable Message Endpoint" at the client side. Since the concrete format of these attributes in message headers is out of scope, here we model an abstract envelope concept. In the future, we plan to map such concepts into existing technologies by model transformation techniques.

3.2 Metamodel Extensions for Reliable Messaging in Services

Now we extend the core SOA metamodel of [2] to capture properties of reliable messaging between services. After enriching the domain metamodel, our long term goal is to define a corresponding UML profile to provide extensions to the

UML language tailored to a specific application domain by introducing domain concepts, attributes and relations in the form of stereotypes and tagged values. However, the current paper only focuses on metamodel-level extensions for reliable messaging in the SOA metamodel.

We first derive a subclass from SOA element in the reliable SOA metamodel, and then create an association from the child class (e.g. RelMsgEnvelope) to the parent class (e.g. Message) in addition. As a result, unreliable messaging can be carried out by the original SOA reconfiguration rules defined in [2]. Furthermore, the original messages are kept but wrapped into an envelope by introducing a new association. As a consequence, only very minor extensions are required to the rules of [2] to transport these envelopes between services to properly memorize the sender and the receiver of a message.

The extensions of the SOA metamodel for reliable messaging is presented in Fig. 2:

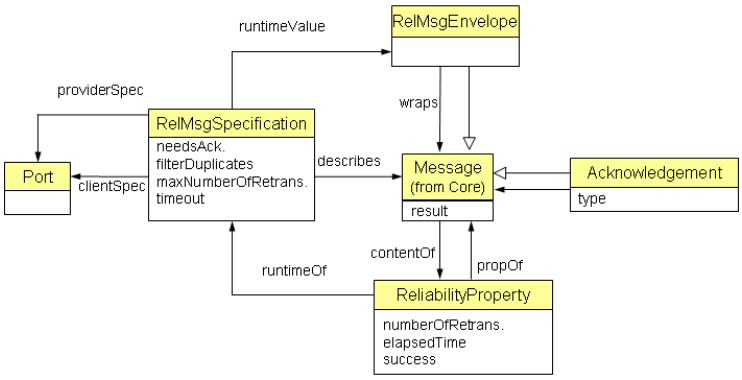


Fig. 2. Metamodel of Reliability Extensions

- RelMsgSpecification (shortly, RelSpec) is a class for specifying the requirements for reliable messaging between SOA services (see association describes, clientSpec, providerSpec).
  - Attribute needsAck is a boolean value to express if an acknowledgement should be sent to a message. If an acknowledgement arrives to the sender for a message, then it is guaranteed that the message is received at least once.
  - Attribute filterDuplicates is a boolean value to express that a message should be accepted and processed by the receiver at most once.
  - Attribute timeout is a timer constraint which specifies how much the sender waits for the acknowledgement of a message before retransmission.
  - Attribute maxNumberOfRetrans is an integer which puts an upper limit on how many times a message can be retransmitted by the sender due to the lack of acknowledgement from the receiver.
- RelMsgEnvelope (shortly, Envelope) is a subclass of core SOA Message which serves as an envelope for wrapping up the real message to be sent (wraps).
- ReliabilityProperty (shortly, RelProp) contains the runtime properties of a

message:
· Attribute numberOfRetrans is a serial number for the envelope which is increased by one each time the same message is retransmitted.
· Attribute timeElapsed denotes the time elapsed since the (last) transmission of a message.

The content of the message is also attached to the properties (contentOf) since the retransmission of the message has to be transparent for the application.

- Acknowledgement (shortly, Ack) is a subclass of core SOA Message which denotes an acknowledgement sent in response to a message.

As this extension is closely related to existing standards, we plan to map such high-level models into implementations of these standards following a model-driven approach: runtime values of XML descriptors will be derived from the attributes of our model.

3.3 Semantics for Message Delivery

In traditional distributed systems, communication middleware have to guarantee the desired semantics of message delivery. The most common semantics are the following:

- *At-Least-Once* is one of the weakest, requiring that every message has to arrive to the receiver at least once. This does not exclude the possibility of sending a message multiple times.
- *At-Most-Once* is ensuring that a message won't be sent more than once, which means the elimination of duplicates.
- *Exactly-Once* is the "subset" of the previous ones both messaeg delivery and filtering of duplicates are guranteed.

There are of course other semantics, hereby we will use *At-Least-Once* as a running example since this is the easier to present. However, our methodology naturally works for the other delivery semantics as well.

4 Reconfiguration for Reliable SOA Messaging by Graph Transformation

We now propose to describe the reconfiguration mechanisms of reliable SOA messaging by graph transformation rules (conceptually following [2]).

4.1 Overview of Graph Transformation

A main benefit of using graph transformations as a formal specification paradigm for capturing reconfiguration rules is that they are visual, intuitive, therefore they can be understood by service engineers as well. The interested reader

may find a detailed theoretical discussion of graph transformation in [6], here we present just a brief overview on it.

Furthermore, graph transformation allows dynamic metamodeling [12] in a certain domain. The high-level (ontological) concepts are visualized as UML class diagrams while graph patterns are considered to be UML object diagrams to express that concrete models are instances (objects) of the metamodel (classes) combining the advantage of precise modeling and visual design.

A graph transformation rule consists of a *Left Hand Side (LHS)*, a *Right Hand Side (RHS)* and optionally a *Negative Application Condition (NAC)*. The *LHS* is a graph pattern consisting of the *mandatory* elements which prescribes a precondition for the application of the rule. The *RHS* is a graph pattern containing all elements which should be present after the application of the rule. Elements in the $RHS \cap LHS$ are left unchanged by the execution of the transformation, elements in $LHS \setminus RHS$ are deleted while elements in $RHS \setminus LHS$ are newly created by the rule. The fulfillment of the negative condition prevents the rule from being executed on the particular matching. Hereby we follow the Single Pushout Approach (SPO) approach [6] with negative application conditions [11].

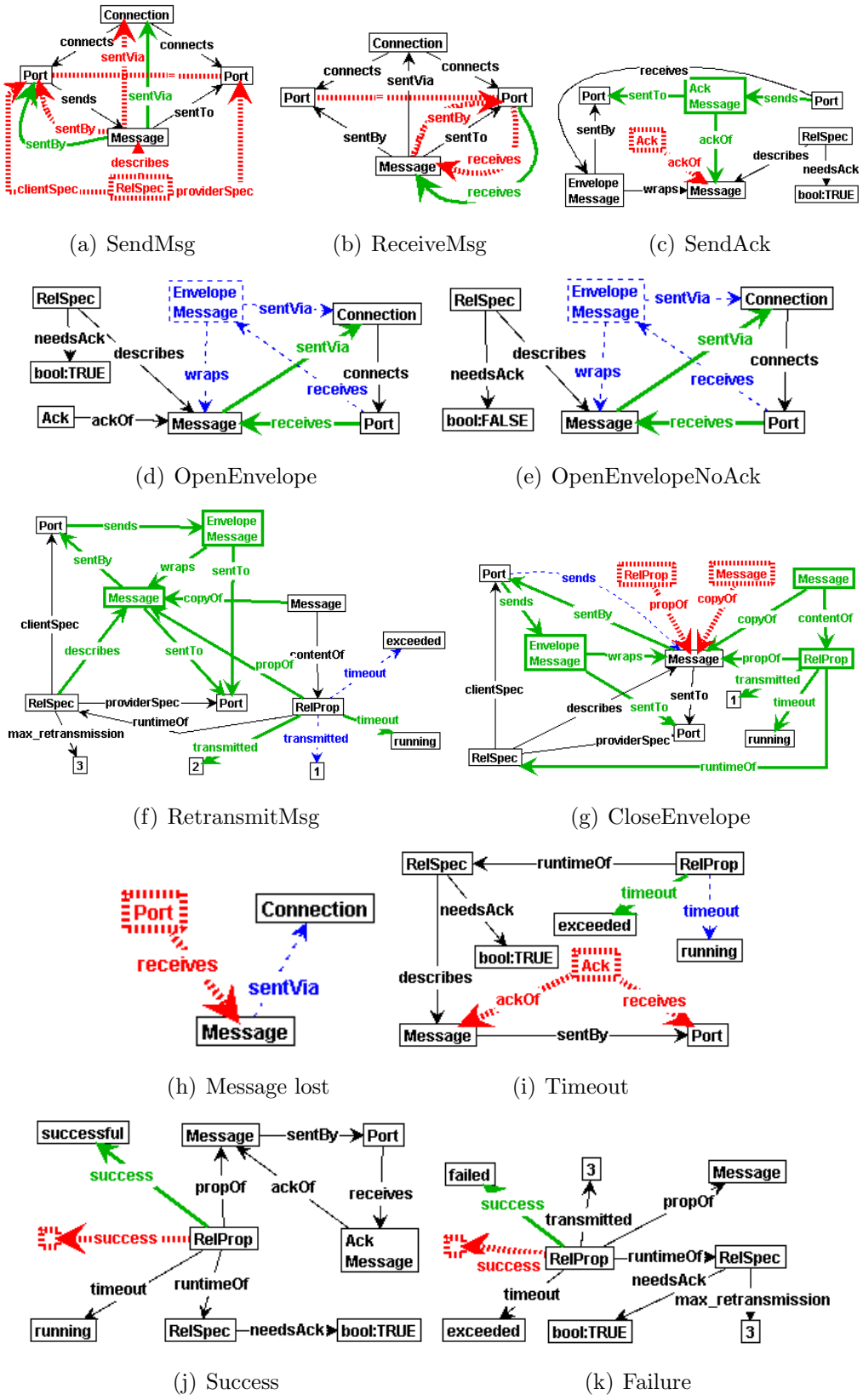
A *graph grammar (GG)* consists of a start graph and a set of graph transformation rules. A *graph transition system (GTS)* represents the state space generated by a graph grammar. The different states of the GG (i.e. the derived instance graphs) appear as nodes while edges denote state transition caused by the application of a graph transformation rule. An edge going from state *s1* to state *s2* with label *r,o* represents that from the graph instance *s1* one can get graph instance *s2* by the application of transformation rule *r* at match *o*.

In this paper, we use a compact visualization of graph transformation rules (first introduced in the Fujaba framework [8] and used in Groove [19]), when the entire rule is merged into a single pattern. Newly created elements are denoted by solid thick (green) lines (tagged as {*new*} in the editor) while deleted elements are depicted by dashed blue lines (tagged as {*deleted*}). Elements in the intersection of the *LHS* and the *RHS* are visualized normally (in black), and elements of NAC appear in thick dotted (red) lines. A negative condition is used in the current paper to prevent the rule from creating infinite number of new elements on the same matching (e.g. in the case of messaging, the same message is received only once).

4.2 Reconfiguration Rules

The reliable messaging with at least once message delivery can be assured by the reconfiguration rules captured by graph transformation in Fig. 3 (using the Groove notation).

First, the normal messages have to be packed into and wrapped from en-



(a) SendMsg (b) ReceiveMsg (c) SendAck

(d) OpenEnvelope (e) OpenEnvelopeNoAck

(f) RetransmitMsg (g) CloseEnvelope

(h) Message lost (i) Timeout

(j) Success (k) Failure

Fig. 3. Transformation rules in GROOVE for reliable messaging

velopes (as in the case of present reliable messaging technologies). Thus, the messages are wrapped up in the sender side instead of being transmitted (rule closeEnvelope in Fig. 3(g)) and envelopes are opened before receiving their content at the receiver side (rules openEnvelope and openEnvelopeNoAck in Fig. 3(d) and Fig. 3(e) where the two separate rules depend on whether a message needs an acknowledgement). As the most general type is used for messages, these rules will match for instances of every subclass of message class with a reliability specification. Thus, reliable messaging is also provided for asynchronous service invocations, discovery queries, etc with a typed, attributed graph transformation engine.

Basic delivery modes include *AtLeastOnce*, *AtMostOnce* and *ExactlyOnce*, determined by the parameters needsAck and filterDuplicates, respectively. Hereby we consider 'primitives' as basic operation, supported by the runtime Web service platform (such as RAMP) to ensure the desired delivery semantics.

At the sender side, there are basically two message sending modes, depending on the value of the needsAck parameter of the RelSpec object describing the requirements for messaging. If this parameter is true, reliable message sending required for a particular message, which corresponds to the *AtLeastOnce* messaging semantics. In this case, the sender will wait for an acknowledgement and consider the transmission of a message successful only if the acknowledgement arrives within the timeout interval. The rule of the successful message transmission (more precisely, the arrival of an acknowledgement in time) is shown in Fig. 3(j).

On the other hand, if the acknowledgement does not arrive in time (rule *Timeout*, Fig. 3(i)), then the next action (i.e. the next rule to be applied) depends on the number of retransmitted messages. If the actual retransmission number of a particular message is smaller than the allowed, then a new instance of the Envelope class is created and sent with the same content and a higher retransmission number (rule *RetransmitMsg*, Fig. 3(f)). If the same message content cannot be sent again (precondition of rule *TransmissionFailure*, Fig. 3(k)), then the transmission of the message is considered to be failed. Note that if no acknowledgement is needed, then no additional rules are applied at message sending, only the core *SendMsg* rule matches the instance graph.

On the receiver side, the messages are acknowledged if needed (see rule *SendAck* in Fig. 3(c)), otherwise the core *ReceiveMsg* rule is applied (Fig. 3(b)).

Additional rules have been introduced to inject faults into the system according to a fault model. In our fault model, we assume that the message may be lost during submission (Fig. 3(h)), or it eventually arrives but a timeout has already occured (Fig. 3(i)). Acknowledgements can also be lost.

These fault injection rules of Fig. 3 are an extension of [10]. Furthermore, since we used a richer graph model in our previous work, all the rules had to be translated into Groove manually (see Sec. 5.2). Finally, during verification, we also found conceptual flaws in the original rule set. For instance, there we

erroneously allowed a message to be received by the sender party itself. These changes are already included in Fig. 3.

5 Verification of Reliable Messaging Rules

5.1 Verification Tool Chain

The transformation rules were implemented in the Groove [19] tool, which supports the generation of the state space (i.e. a Graph Transition System - GTS) derived by a graph grammar. Using the Groove simulator, one can manually inspect the state space from a given start graph for verification purposes. While this is convenient for early tests of the GT specification, this is not very convincing in case of large state spaces. Unfortunately, the current public version of Groove (March, 2006) that we used in our experiments did not yet support the verification of CTL-like properties (reported recently in [20]).

For this reason, we decided to carry out the verification of the reconfiguration rules for reliable messaging by post-processing the generated GTS in the Labeled Transition System Analyzer (LTSA, [15]) tool. This tool supports the safety, deadlock and liveness analysis of Labeled Transition Systems. A requirement to be verified is defined by a normal *(requirement) process*, which explicitly captures correct and incorrect execution paths (wrt. a subset of actions) or a *property* from which the corresponding process is generated automatically by the tool. For verification runs, the requirement process and the system process are composed concurrently. The result of verification is either successful or a counterexample is provided in the form of a transition sequence which leads to the violation of the requirement.

In order to project GTSs into the input format of LTSAs, a translator was implemented which takes the GXL input of the GTS generated by Groove and creates LTSA processes accordingly.

Furthermore, since the LTSA analyzer always checks for the existence of deadlocks (even if a deadlock means correct termination of the system), we had to guarantee that the GTS is cyclic by introducing additional "restart" graph transformation rules. These rules are applicable to any configuration and delete all information regarding to the state of the system. Alternatively, this also could be done by implementing an extension in the translator from GTS to LTSA to create loops on the final states.

5.2 Groove-specific Adaptations of Transformation Rules

In order to encode the transformation rules into Groove (see Fig.3), we had to model concepts such as inheritance, types and instantiation in Groove which supports only labeled edges between nodes. Therefore, the types of the nodes were modeled as self-edges, and we used multiple edge labels in case of inheritance. For instance, the object in the top of Fig. 3(c) has a type of

Acknowledgement (shortly Ack) which is a specialization of Message. Concrete attribute values (such as the counter of the transmitted messages) were implemented as nodes, linked to their container nodes.

As the current version of Groove had some minor bugs which prevented the correct handling of primitive datatypes such as integers, the required constant values were inserted as individual nodes (e.g. node with the self-edged "exceeded" represented the string "exceeded", etc.). For the same reasons, we had to properly duplicate the rules which used comparison operations on integer values. For instance, sending a message for the second and third was implemented as two different rules.

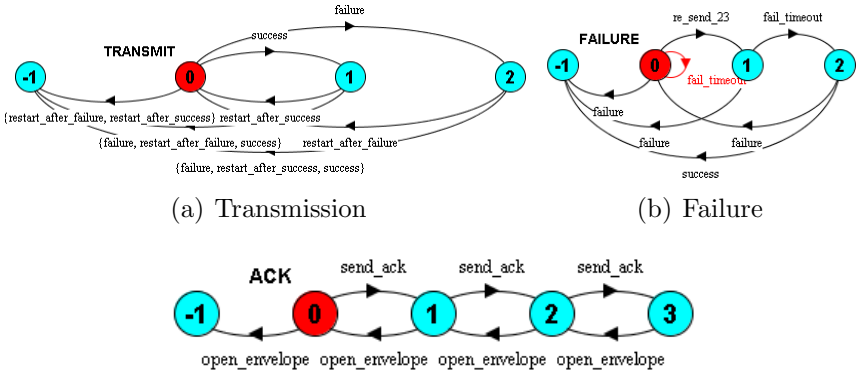
For the verification of these rules, the graph grammar had to be extended to ensure that *it will have an infinite lifecycle*. We ensured the start state could be reached from any subsequent state by systematic modifications of the rules. Firstly, two new rules were created to restore the start state of the system after the (either successful or a failed) transmission of a message. Secondly, three auxiliary transformations were implemented to delete the unnecessary elements such as messages, envelopes and acknowledgements to keep the state-space finite. Third, all other rules were extended by a NAC containing the success attribute of the MessageProperty class to prevent these rules from being concurrently executed with the initialization sequence.

5.3 Verification of Properties

We identified the following (non-exclusive) list of important requirements for reliable messaging:

- The transmission of a message is either successful or failed (but the submission has a definite result).
- The transmission is considered to be failed exactly when the timeout of the acknowledgement for the last transmittable message instance is exceeded.
- Incoming messages are read only after being acknowledged (if acknowledgement is required).
- Multiple messages of the same port (to the same or different ports) are managed correctly, i.e., their runtime properties are handled serapately.
- Sending and receiving normal (unreliable) messages can still be carried out.

When formalizing these requirements in LTSA, we ran into two main problems. On the one hand, the GTS generated by Groove only contains the applied rule as labels but no information is provided on the occurrence. For this reason, we can capture only those requirements where the identity of messages are irrelevant. In several cases, only a weakened form of the requirement was actually verified due to this problem. In our opinion, providing also information for the matching is an interesting direction for future improvements in Groove.



(a) Transmission (b) Failure

(c) Acknowledgement

Fig. 4. Automata of the properties (-1 represents the error state)

On the other hand, LTSA offers a limited way for checking liveness properties, therefore, our attention was mainly focused on to verify safety properties of reliable messaging.

The main benefits of using these two tools together was the easy generation of the available states and an automated check of properties. The previous requirements were interpreted and formalized in the form of LTSAprocesses / properties (see Fig. 4) as follows.

- The transmission of a message is either successful or failed: Exactly one of the transformation rules Success, Failure is applied in each path to the restart state(Fig. 4(a)).
- The transmission is considered failed exactly in the case when the timeout of the acknowledgement of the last transmittable message instance is exceeded (Fig. 4(b)).
- Incoming messages are read only after being acknowledged, i.e. the application of the SendAck rules precedes that of the OpenEnvelope rule. Note that if no acknowledgement is needed, then the property is not violated as OpenEnvelopeNoAck rule is applied instead (Fig. 4(c)).

By carefully selecting initial models to capture small but typical configurations, we were able to verify that our reconfiguration rules fulfill these requirements. We believe that these models are minimal but representative configurations which could be extended if this didn't raise a conflict with the limitation of the state space generation. On the other hand, some of our preliminary expectations turned out to be false during the verification (for instance, that a message lost would always cause timeout).

The main lessons we learned from this verification case study for reconfiguration rules used in reliable messaging are the following:

- *High-level vs. low-level graph models and rules:* We were able to translate (by hand) rich graph transformation rules and models (as used in [10] with inheritance, types, attributes, etc.) into lower level verification mod-

els (used in Groove) with relatively simple modeling tricks. Problems have mainly arisen in case of integer attributes where rules have to be copied to handle all different matches of the integer attributes. Future support for core datatypes in Groove would further reduce the complexity of this last task.

- *Testing in Groove:* Minor conceptual flaws have been identified in the rules of [10] by manually inspecting the generated GTS for smaller examples. However, such manual inspection for deciding the correctness of a property was infeasible for large state spaces.
- *Verification in LTSA:* Automated post-processing in LTSA is a feasible solution for verifying meaningful safety properties with obvious limitations due to the lack of identity information in the GTS generated by Groove.

6 Related work

Related work in this field usually concentrate either on describing the non-functional attributes of services, or modeling dynamic aspects of Service Oriented Architectures by graph transformation.

Our work conceptually follows the approach of [2] for specifying services in SOA. The authors of [3] describe the application of graph transformations in the runtime matching of behavioral Web service specifications. In [13], the conformance testing of Web services is based on graph transformations, focusing on the automated test case generation. However, none of these works discusses the aspects of reliable messaging. Our aim was to utilize the benefits of this approach by extending the metamodel and the transformation rules.

Graph transformation is used as a specification technique for dynamic architectural reconfigurations in [7] using the algebraic framework CommUnity. Hirsch uses graph transformations over hypergraphs in [14] to to specify runtime interactions among components, reconfigurations, and mobility in a given architectural style. However, the problem of reliable messaging in SOA is not addressed in either case.

LTSA [15] has already been applied successfully for the formal analysis of business processes given in the form of BPEL specifications in [9], but reliable messages are not considered in these papers.

In the future, we plan to investigate the use of other verification tools for graph grammars. A primary candidate is Augur [21], which uses unfolding techniques to derive a finite approximation of possible traces in a GTS.

The specification and analysis of fault behaviors have been carried out in [4] using graph grammars. While this approach is not directly related to SOA, it may serve as a starting point for incorporating additional dependability aspects for our research.

Reliable messaging were verified in other papers, (for instance, [1]), our technique differs mostly in the level of modeling, which is closer to that of the

usual SOA description, and therefore, is more appropriate to apply for the verification of fault tolerant mechanisms in SOA.

Finally, in the industrial field, there are existing and emerging specifications and technologies like [26,27]. However, their use is still ad-hoc and no model-based design-time support is available for reliable messaging.

7 Conclusion

In this paper we first proposed an extension to the core SOA metamodel of [2] and a technique to capture the reconfiguration mechanisms to enhance the development of more robust SOA middleware. Reconfiguration rules for reliable messaging in SOA have been captured by graph transformation rules.

Then the correctness of these rules were verified by first generating the state space of the graph grammar by Groove (in the form of graph transition systems), then transforming it into labeled transition system in order to carry out formal verification of correctness requirements using the LTSA analyzer tool.

As the next step in the future, we plan to focus on bridging the gap between our abstract reconfiguration rules and existing implementation technologies for reliable web services. The formally verified set of reconfiguration rules will definitely serve as a sound starting point for this activity. Our long term goal is to automatically derive implementations of reliable messaging on various existing platforms based directly upon provenly correct dynamic reconfiguration mechanisms.

References

[1] P. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans. *Modeling and verifying a bounded retransmission protocol.* In Proc. of COST-247 Int. Workshop on Applied Formal Methods in System Design, 1996, Slovenia.

[2] L. Baresi, R. Heckel, S. Thöne and D. Varró, *Style-Based Modeling and Refinement of Service-Oriented Architectures.* To appear in Journal of Software and Systems Modelling, 2006.

[3] A. Cherchago and R. Heckel. *Specification Matching of Web Services Using Conditional Graph Transformation Rules.* In Proc. of Int. Conference on Graph Transformations, 2004, LNCS Vol.3256., Springer, pp. 304-318.

[4] F. L. Dotti, L. Ribeiro, and O. M. dos Santos. *Specification and analysis of fault behaviours using graph grammars.* In Applications of Graph Transformations with Industrial Relevance, Second Int. Workshop, AGTIVE 2003, USA, vol. 3062 of LNCS, pp. 120–133. Springer, 2003.

[5] DublinCore Metadata Initiative, 2006. http://dublincore.org/

[6] H. Ehrig, R. Heckel, M. Korff, M. Lowe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic approaches to graph transformation, Part II: Single pushout approach and comparison with double pushout approach.* In Handbook of Graph Grammars and Computing by Graph Transformation, volume I: Foundations, pp. 247–312. World Scientific, 1997.

[7] M. Wermelinger and J. L. Fiadeiro. *A graph transformation approach to software architecture reconfiguration.* Science of Comp. Progr., 44(2):133–155, 2002.

[8] T. Fischer, J. Niere, L. Torunski and A. Zündorf, "Story Diagrams: A new Graph Transformation Language based on UML and Java", Proc. Theory and Application to Graph Transformations (TAGT'98), vol. 1764 of LNCS, 2000, Springer.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. *Model-based verification of web service compositions.* In 18th IEEE Int. Conf. on Automated Software Engineering (ASE 2003), Canada, pp. 152–163. IEEE Computer Society, 2003.

[10] L. Gönczy, D. Varró: *Modeling of Reliable Messaging in Service Oriented Architectures*, In Proc. Int. Workshop on Web Service Modeling and Testing (WS-MATE 2006). To appear.

[11] A. Habel, R. Heckel, and G. Taentzer. *Graph grammars with negative application conditions.* Fundamenta Informaticae, 26(3-4):287313, 1996.

[12] J.H. Hausmann, Heckel, R., Lohmann, M.: *Model-based Discovery of Web Services*, In Proc. of the IEEE Int. Conference on Web Services (ICWS), June 6-9, 2004, USA,

[13] R. Heckel, and L. Mariani. *Automated Conformance Testing of Web Services.* In Proc. of 8th Int. Conference on Fundamental Approaches to Sofware Engineering (FASE 2005), vol. 3442 of LNCS, Springer, pp. 34-48.

[14] D. Hirsch. *Graph transformation models for software architecture styles.* PhD thesis, Departamento de Computacion, Universidad de Buenos Aires, 2003.

[15] Labelled Transition System Analyser (Version 2.2) http://www-dse.doc.ic.ac.uk/concurrency/ltsa-v2/index.html

[16] J. Laprie, B. Randell, C. Landwehr, *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Transactions on Dependable and Secure Computing, (Vol.1, No.1.) (2004) pp. 11-33

[17] Object Management Group. Model Driven Architecture. http://www.omg.org/mda

[18] Reliable Asynchronous Message Profile (RAMP) Toolkit, IBM alphaworks. http://www.alphaworks.ibm.com/tech/ramptk

[19] A. Rensink, *The GROOVE simulator: A tool for state space generation.* In Proc. of Application of Graph Transformations with Industrial Relevance (AGTIVE'03), 2003, LNCS Vol.3062, Springer, pp. 479-485.

[20] Kastenberg H., and Rensink A. *Model Checking Dynamic States in GROOVE.* In Proc. of the 13th Int. Workshop on Software Model Checking (SPIN'06), Volume 3925 of LNCS, Springer-Verlag, 2006, to appear.

[21] B. König and V. Kozioura. *Augur - a tool for the analysis of graph transformation systems.* Bulletin of the EATCS, vol. 87:pp. 126–137, 2005.

[22] RM4GS Reference Guide, Version 1.0, FUJITSU LIMITED, Hitachi, Ltd. and NEC Corporation, 2004.
http://xml.coverpages.org/rm4gs20041125-reference.pdf

[23] Software Engineering for Service-Oriented Overlay Computers (SENSORIA) European project IST-3-016004. http://sensoria.fast.de

[24] Web Services Architecture, 2004. http://www.w3.org/TR/ws-arch/

[25] Web Service Modeling Ontology (WSMO), W3C Member Submission, 2005. http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/

[26] Web Services Reliable Messaging Protocol (WS-ReliableMessaging) BEA Systems, IBM, Microsoft Corporation, Inc, and TIBCO Software Inc., 2002.

[27] Web Services Reliable Messaging TC WS-Reliability 1.1 Committee Draft 1.086, OASIS Open Consortium, 2004.

Simulation of Generalised Semi-Markov Processes based on Graph Transformation Systems

Piotr Kosiuczenko, Georgios Lajios [1]

Department of Computer Science
University of Leicester, UK
{pk82, gl51}@mcs.le.ac.uk

Abstract

Stochastic Graph Transformation combines graphical modelling of various software artefacts with stochastic analysis techniques. Existing approaches are restricted to processes with exponential time distribution. Such processes are sufficient for modelling a significant class of stochastic systems, however there are interesting systems which cannot be specified appropriately in such a framework. In several cases one needs to consider non-exponential time distributions. This paper proposes a stochastic model based on graph transformation with general probability distributions. This model is well suited to represent concurrency and performance aspects of architecture reconfiguration. It is also possible to apply Monte Carlo simulation techniques in order to analyse behaviour of complex stochastic systems. The new model is implemented and used to simulate simple networks.

Key words: Stochastic modelling, graph transformation, simulation

1 Introduction

The specification of distributed systems, telecommunication systems, multimedia applications or computer networks must take into account not only functional properties but also real-time and performance aspects. To analyse such properties, stochastic methods are required.

Stochastic models like Generalised Semi-Markov Processes (cf. e.g. [13]) have a long history of application, but they do not provide primitives for mod-

[1] Corresponding author. This research was partially funded by European Community's Human Potential Programme under contract HPRN-CT-2002-00275, [SegraVis].

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

elling of concurrency aspects. They also lack mechanisms for compositional specification. Thus models of larger systems tend to be very complex.

There exist several formalisms for the analysis of performance and concurrency aspects. We will discuss briefly the most prominent of them. One of the first models used for this purpose were Stochastic Petri nets [8]. Generalized Stochastic Petri Nets found a wide acceptance (cf. [1]). Those nets are defined as usual Petri nets, with the addition of random assignment of a firing delay to each transition. There is a race condition between all enabled transitions. In the case of exponential probability distributions, this model corresponds to Continuous Time Markov Chains (CTMC) [13]. Stochastic Petri nets proved to be very expressive. They are well suited for specification of concurrency aspects, but the resulting model is rather low level.

Process algebras provide compositional facilities for modelling of concurrent systems. Stochastic process algebras are a natural extension of process algebras (cf. e.g. [12] and part two of [2]). They are used for performance modelling. As in the case of Stochastic Petri nets, a non-negative real number is randomly associated to an action. That number determines the delay of the corresponding action. While basic approaches rely on continuous-time Markov chains, there are also extensions to general distributions, based on Generalised Semi-Markov Processes and Stochastic Automata [2].

Nevertheless, architectural aspects of distributed systems, computer networks and mobile applications can be hardly specified with those formalisms. Especially in the case of high degree of architectural reconfiguration a high level formalism is needed with facilities for modelling architectural artefacts. This gave rise to the notion of Stochastic Graph Transformation, which combines the benefits of using graph transformation for system modelling with the power of stochastic analysis [11] (see also [18]). Those approaches enrich graph grammars by associating an exponential time distribution to each transformation rule. The distribution models the random delay of rule application. This model is a special case of a CTMC. There exist powerful model checking tools such as PRISM [14] which can be used for analysing properties of CTMCs. However, there are several stochastic phenomena, which cannot be modelled using exponential distribution. For example, file sizes and document transmission times over HTTP/IP and timeouts in communication protocols cannot be appropriately modelled with exponential distributions. Further, one would often like to include results of measurement into the modelling. There are standard techniques for extracting normal distributions from a random sample. Sometimes, only the minimum and maximum value of a quantity are known, thus modelling could be done by assuming a uniform distribution. Those cases can only be modelled using a wider class of distributions and a more general model.

In this paper we propose *Generalised Stochastic Graph Transformation Systems* to model and analyse architectural evolution with non-exponential time distributions. States of concurrent systems are modelled by graphs.

Transitions of those systems are modelled by graph transformations. To model delays, we associate arbitrary continuous probability distributions with graph transformation rules. Graph transformation executions have delays, which adhere to those distributions. The model works as follows: a system state is modelled by a graph and the delay of rule application is measured by a separate timer. Different rules may be applicable, but only the rule with the smallest delay can be executed. If a rule is executed, then a new state is reached, timers corresponding to enabled rules are decreased, timers corresponding to disabled rules are removed, and new timers are set for rules which become enabled. Let us observe that graph transformation rules can be in conflict, and an application of one rule may disable application of another one.

This model is well suited for modelling of concurrency aspects, architectural aspects as well as stochastic aspects. It uses timed events to model the concurrent execution of events, thereby giving a direct representation of the intuitive idea that each event has its own timer and will be applied when its time expired, independent of other events. Our approach can be seen in the line of research combining high-level modelling techniques with stochastic analysis.

Stochastic modelling is only useful if there are analysis techniques to investigate the properties of the systems which are modelled. The more general a modelling approach is, the more difficult is its analysis. Stochastic model checkers like PRISM [14] are very powerful tools for Markov Chains, but fail when more general stochastic processes are involved. Anyway, their power could not be fully exploited for stochastic graph transformation, because the complete state space of the model has to be generated first, before model checking tools can be used. This procedure emerged as a serious bottleneck, because the isomorphism checking involved is very complex. Even systems consisting of a small number of nodes and edges can lead to an enormous amount of states, a phenomenon known as *state space explosion*. When switching to arbitrary distributions, model checking itself becomes more complex [16].

We propose using Monte Carlo simulation techniques for testing stochastic graph transformation systems with arbitrary distributions. Monte Carlo Methods are stochastic simulation methods based on pseudo-random numbers. These methods allow us to make predictions about system's behaviour. The simulated system is traversed on randomly chosen paths; those paths simulate real-time behaviour. After a sufficient number of such paths is traversed, knowledge on the probabilistic behaviour the system is gathered, with a certain confidence interval which can be narrowed by further runs. Simulation is thus a very well scalable technique. First experiments with simulation have been promising, and we are currently developing a tool for the analysis of stochastic graph transformation systems based on these techniques.

The paper is organized as follows: The next section explains why it is necessary to use general distributions. Section 3 presents the underlying notion of Generalised Semi-Markov Processes. The new model called (Generalized)

Stochastic Graph Transformation System and the corresponding stochastic process are defined in Section 4. Section 5 explains how Monte Carlo simulation techniques can be used in the case of stochastic graph transformation systems and presents some experimental results. Section 6 concludes this paper.

2 Integrating Graph Transformation and general distributions

Combining graph transformation systems with stochastic models which have non-exponentially distributed application delays is not as trivial as one might expect. Graph transformation is intrinsically concurrent; in general, more than one rule can be applied to a graph. Non-deterministic choice between the alternatives leads to the problem of sequential and parallel independence [19]. Stochastic graph transformation aims at making this choice stochastic. Rule applications have stochastic delays, and there is a *race* between all applicable rule matches which the fastest contestant wins. Formally, this means that the stochastic application delay of the rules is computed for each match by drawing a random sample according to a stochastic specification, and the rule match with the smallest delay is applied first. For instance, think of a graph grammar modelling a network where rules *connect* and *disconnect* can be applied to a certain state. For each rule, a probability distribution specifies the behaviour of these actions. Say, *connect* is exponentially distributed with expected value 1 sec, and disconnect is normally distributed with mean 25 sec and standard deviation 25 sec (truncated at 0). Then, in most cases *connect* will win the race, leaving only probability 1% for *disconnect*. When all actions are exponentially distributed, evaluating the race condition is easy because of the memoryless property. This means that the time which had already elapsed for the "loser" action need not be taken into account for the next race, as the probability that an exponentially distributed random variable is greater than $t + s$ conditional on being greater than t is the same as being greater than s. Therefore, in a continuous-time Markov chain, which is the structure obtained in the pure exponential case, a transition can be performed and one can forget about everything which happened before.

In the case of general distributions, this simplification is not possible. In the example above, if rule *disconnect* has lost a couple of races, say adding up to 20 sec, then this waiting time has to be considered, making an application more probable – about 7% in one step, yielding more than 50% for ten consecutive steps. It is this semantics which a modeller presumably intends when assigning probabilistic waiting times to rules.

Proper assignment of waiting times in the context of graph transformation means that matches have to be traced through transformation sequences. The waiting time of a rule application has to be considered for every match of the rule as long as the match is present. So if a transformation step changes

elements of the graph which are not in conflict with the match, i.e. if the productions are independent, the old match is still present in the new graph, and the value of the timer measuring waiting time has to be decreased. We will address this issue by using unfolding grammars, which allows unique identification of elements [3].

One solution avoiding waiting times is to approximate general distributions by introducing virtual states and combining exponential distributed transitions in such a way that the desired distribution results. This is known as Cox's method of phase-type distributions [6]. A major drawback of this approach is that it expands the state space. Also, many interesting distributions can only be approximated with a very high number of virtual states. The resulting models are not intuitive, as there is no direct correspondence between the states of the stochastic model and the states of the system. We therefore propose a stochastic model which considers waiting times – Generalised Semi-Markov Processes.

3 Generalised Semi-Markov Processes

Generalising the notion of CTMCs to arbitrary distributions, there are two options: Semi-Markov Chains [16] or Generalised Semi-Markov Processes. As discussed above, semantic models of interleaving processes need to consider waiting times, which are not supported by Semi-Markov Chains. We therefore vote for the second alternative and adopt the following definition from [2].

Definition 3.1 A generalised semi-Markov scheme (GSMS) is a structure $(Z, E, active, next, F)$ where

- Z is the set of states;
- E is a set of events;
- $active : Z \to \mathcal{P}(E)$ assigns a finite set of active events to each state;
- $next : Z \times E \to Z$ is a partial function that assigns the next state according to the current state and the event that is triggered. We assume that $next(z, e)$ is always defined for $z \in Z$ and $e \in active(z)$;
- $F : E \to (\mathbb{R} \to [0, 1])$ assigns to each event a continuous distribution function such that $F(e)(0) = 0$; we write F_e instead of $F(e)$.

As initial condition a state $z_0 \in Z$ is appointed. A generalised semi-Markov process (GSMP) is the stochastic process defined by a GSMS.

The behaviour of a GSMP can be described as follows. In each state z, all active events $e \in active(z)$ are assigned a real number $\rho(e)$, the remaining time to execute the event. The next step is determined by the active event e^* with smallest number $\rho(e)$. One can think of race between the competing events which is won by the fastest event. Note that the probability that two events have the same time is 0 due to the fact that the distribution function is

continuous. Once the event is chosen, the next state is given deterministically by $next(z, e^*)$. The set $New(z, e^*)$ of newly activated events is defined as

$$New(z, e^*) = active(next(z, e^*)) \setminus \Big(active(z) \setminus \{e^*\}\Big),$$

i.e. the events which became active in the new state and have not been active before. The set $Old(z, e^*)$ is given by all active events that have been active in the old state (without e^*):

$$Old(z, e^*) = active(z) \cap \Big(active(next(z, e^*)) \setminus \{e^*\}\Big).$$

Now, the value of ρ is determined randomly for all newly activated events e according to their distribution F_e, and the value of all old events is decreased by $\rho(e^*)$. Thus the updated function ρ' is given by

$$\rho'(e) = \begin{cases} Random(F_e), & \text{if} \quad e \in New(z, e^*) \\ \rho(e) - \rho(e^*), & \text{if} \quad e \in Old(z, e^*) \end{cases}$$

where $Random(F_e)$ denotes a random number determined by drawing a sample according to distribution F_e.

The operational semantics of this model is defined by mapping a GSMP to a Stochastic Automaton [2]. Despite their more complex semantics, GSMPs are a direct generalisation of continuous-time Markov chains (CTMC). In fact, a GSMP in which all events are associated an exponential distribution is a CTMC. Because of the memoryless property of the exponential distribution, it is not necessary to consider how long an event has already been active, as the conditional probability $\mathbf{P}(X > s + t \mid X > t)$ equals $\mathbf{P}(X > s)$. GSMPs provide an excellent basis for modelling state-based systems with arbitrary distributions.

4 (Generalised) Stochastic Graph Transformation Systems

In this paper we use the single-pushout approach for graph transformation [19]. To make the new model more general, we define graphs in a categoric way instead of set theoretic one. Typed graph transformation is a technique of key relevance in the modelling of visual languages and in model transformation. The type graph can be defined in several ways and used for various purposes; in particular it can be defined as a colimit obtained in a graph unfolding process [3]. Types can be understood in particular as object IDs.

Definition 4.1 A directed graph is a quadruple $G = \langle G_V, G_E, src_G, tar_G \rangle$ with a set of vertices G_V, a set of edges G_E, and functions $src_G : G_E \to G_V$ and $tar_G : G_E \to G_V$ associating to each edge its source and target vertex.

A graph morphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ preserving source and target, i.e., such that $f_V \circ src_G = src_H \circ f_E$ and $f_V \circ tar_G = tar_H \circ f_E$. A partial graph morphism $f : G \to H$ is a graph morphism which is defined on a subgraph $dom(f)$ of G. A typed graph t over a (fixed) type graph TG is a graph morphism $t : G \to TG$ which assigns types to nodes and edges [7]. A morphism of typed graphs is a graph morphism compatible with the typing.

A rule $p : L \xrightarrow{r} R$ consists of a rule name p and an injective partial graph morphism r. A match for $r : L \to R$ into some graph G is a total injective morphism $m : L \to G$. Given a rule p and a match m for p in a graph G, the SPO-transformation from G with p at m is the pushout of r and m in the category of graphs and partial graph morphisms.

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m^*} \\ G & \xrightarrow{r^*} & H \end{array}$$

A graph transformation system $\langle TG, P, \pi, G_0 \rangle$ consists of a type graph TG, a set P of rule names and a function π mapping each rule name to a TG-typed rule $\pi(p) : L_p \to R_p$. We make use of negative application conditions (NACs) [9], shown as crossed out nodes and edges in the figures. Images of crossed out elements must not be present in the instance graph, otherwise the rule is not applicable. Every crossing line represents one NAC, and all have to be satisfied in order to apply a rule. Formally, a NAC is given by an injective morphism $n : L \to N$ which maps the left hand side of a rule to a pattern N, and a rule is applicable iff its match $m : L \to G$ does not factor through n, i.e. there is no injective (total) morphism $f : N \to G$ such that $f \circ n = m$.

A generalised[2] stochastic graph transformation system associates with each rule name a distribution function governing the delay of its application.

Definition 4.2 [stochastic GTS] A (generalised) stochastic graph transformation system $\mathcal{SG} = \langle TG, P, \pi, G_0, F \rangle$ consists of a graph transformation system $\mathcal{G} = \langle TG, P, \pi, G_0 \rangle$ and a function $F : P \to (\mathbb{R} \to [0, 1])$ associating with every rule name a continuous distribution function F_p with $F(e)(0) = 0$.

Example 4.3 Figure 1 shows the rules of a graph transformation system modelling a peer-to-peer network. New peers enter the network (rule *new*), establish a connection (rule *connect*), and eventually disappear (rule *kill*). The empty graph serves as start graph. We assume that there is no dangling edge condition preventing connected peers from being killed.

In order to obtain a stochastic graph transformation system, we have to associate continuous time distributions with the rules. We assume that the arrival rate of new peers is exponentially distributed with rate λ. This is

[2] We call this kind of system *generalised* as it uses general distributions in contrast to the approach in [11]. In the rest of the paper, we will omit this word.

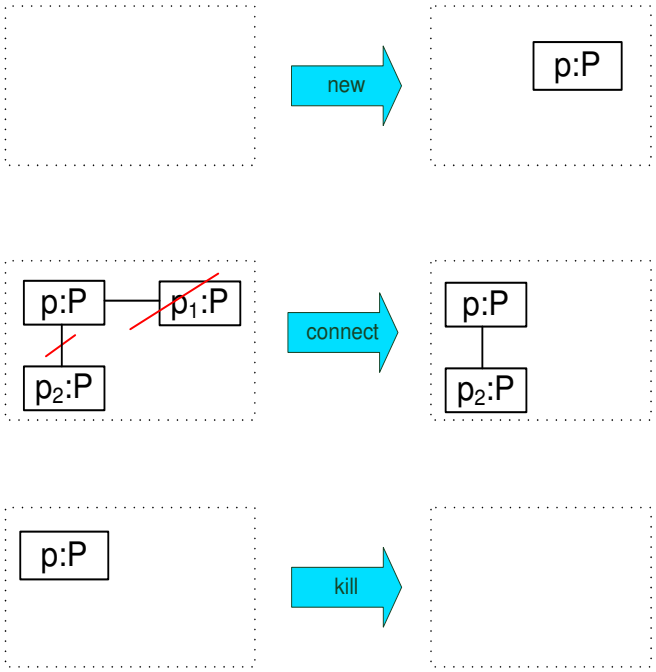


Figure 1. Peer-to-peer network: Basic rules

a standard assumption in queuing theory. Let the application delay of rule *connect* be governed by a normal distribution with mean μ and standard deviation σ. So we assume that most peers tend to be connected approximately the same time, which is chosen to be the mean of the normal distribution. Finally, the lifetime of a peer (which corresponds to the application delay of rule *kill*) is assumed to have Erlang distribution with shape k and rate κ. The Erlang distribution specialises to the exponential distribution for $k = 1$, but allows a greater flexibility for defining probability densities with peaks later than time 0. It is therefore often applied for waiting times.

rule name	distribution	parameters
new	$1 - e^{-\lambda x}$	$\lambda = 0.5$
connect	$\frac{1}{\int_0^\infty \exp(-(t-\mu)^2/(2\sigma^2))dt} \int_0^x \exp\left(-\frac{(t-\mu)^2}{2\sigma^2}\right)dt$	$\mu = 1, \sigma = 1$
kill	$\frac{1}{(k-1)!}\gamma(k, \kappa x)$	$k = 2, \kappa = 0.5$

The intuitive idea of the semantics of a Stochastic Graph Transformation System can be explained as follows. In the start state, all matches for all rules are determined and stored as pairs $< rule, match >$. We call such a pair a *rule match*. For each rule match the application time is set to a random number corresponding to the distribution the rule obeys. Then, the rule match

with the smallest time is chosen, applied, and the remaining rule matches are checked. If their match is no more applicable, they are removed. If it is still applicable to the new graph (we will soon define this thoroughly), its time is reduced by the time that already elapsed. All new rule matches are computed and assigned a random application time. Then again, the fastest of them is chosen for application. So the waiting time of the events which lost the last race is considered.

We formally define the semantics of a (Generalised) Stochastic Graph Transformation System by mapping it to GSMP. The rough idea is to define the set of states as all reachable graphs of the graph transformation system. An active event in a state is a rule match. Thus, the newly activated events in a sequence of states are those rule matches which did not exist in the previous state. We therefore have to compare matches to different reachable graphs, which can be done by introducing a global name space. For this purpose, the concept of *unfolding grammars* [3] comes handy, as it allows to derive from an arbitrary graph grammar a safe (i.e. injectively typed) grammar providing a compact representation of the transition system. The unfolding type graph TG' can be seen as a global name space, and the rules of the unfolding represent rule matches of the original grammar (typed over the global name space). By using the unfolding, we can directly compare matches into different graphs, just by calculating their intersection in TG'.

Let $\mathcal{SG} = \langle \mathcal{G}, F \rangle$ be a Generalised Stochastic Graph Transformation System, where $\mathcal{G} = \langle TG, P, \pi, G_0 \rangle$ is a typed graph grammar, and let $\mathcal{UG} = \langle TG', P', \pi', G'_0 \rangle$ be the unfolding grammar associated with $\mathcal{G}$ (ignoring F). The construction of the unfolding is explained in detail in [3]. Roughly, the type graph TG' is a colimit of the whole transition system which serves as a global name space, and a rule name $p' \in P'$ represents a rule match $\langle p, m \rangle$ of the underlying grammar $\mathcal{G}$, where m is an embedding into TG'.

The unfolding is mapped over the original grammar by the so-called folding morphism $\chi = \langle \chi_T, \chi_P \rangle : \mathcal{UG} \to \mathcal{G}$. The first component $\chi_T : TG' \to TG$ is a graph morphism mapping each graph item in the type graph of the unfolding to the corresponding item in the type graph of the original grammar $\mathcal{G}$. The second component $\chi_P : P' \to P$ maps any production occurrence $\langle p, m \rangle$ in the unfolding to the corresponding production p of $\mathcal{G}$.

We are now ready to define the GSMP associated with a Stochastic Graph Transformation System. The state space Z consists of all graphs reachable from the start graph by applying the rules from the unfolding grammar. The set E of events is defined as $E := P'$. Rule matches which coincide on the global name space are thus identified. The set *active* of active events in a state $G \in Z$ consists of all rules applicable to G. Given such a match, the result of function $next(G, p')$ is defined to be the unique graph resulting from applying rule p' to G in the unfolding grammar. We assume a concrete deterministic definition of the pushout.

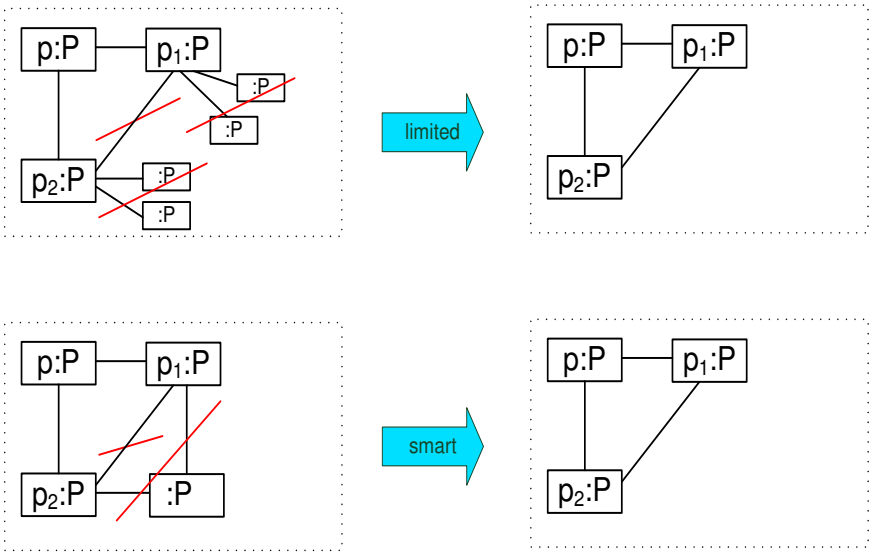


Figure 2. Peer-to-peer network: Alternative shortcut rules

The definition of function F is extended from rule names to events:

$$F_{p'} = F_p \quad \text{for } p = \chi_P(p').$$

Putting the parts together, we obtain a generalised semi-Markov scheme $(Z, E, active, next, F)$. The GSMP associated with this scheme defines the semantics of $\mathcal{SG}$.

5 Analysis

Once a Generalised Semi-Markov Process is obtained, one can analyse its properties in order to get knowledge on the behaviour of the system which was modelled. Important aspects include the behaviour on the long run – the *steady state* – as well as transient analysis, which gives the probability that a transition is performed in a certain period of time.

The numerical analysis of stochastic graph transformation systems can be done in different ways. The approach proposed in [11] consists in generating the state space of the system, transforming the result to a stochastic model, and using existing model checking tools to analyse its properties. This procedure suffers from the drawback that state space generation is very complex due to the fact that isomorphic graphs have to be determined. When leaving the realm of exponential distributions, model checking tools become less efficient, and arbitrary distributions are usually not covered. So we use Monte Carlo simulation as an alternative approach [5]. The intuitive idea is to tra-

verse a number of randomly chosen paths through the system, and thereby sample information on its behaviour. More precisely, pseudo-random numbers are generated according to a given distribution. Depending on the outcome of this pseudo-random experiment, the successor state of the currently occupied state is chosen. Repeating this procedure results in a path through the system, and generating a sufficient number of such paths, one can make predictions on the system's properties.

The *Event Scheduling Scheme* proposed in [5, Sect. 10.2] provides a simulation algorithm for GSMPs. First, an initialisation procedure has to be performed: The *state* of the system is set to the initial state, the *simulation time* is set to 0. Random numbers t are determined for all events e active in the initial state, and the *scheduled event list* is initialised with them, with all entries sorted in increasing order according to their scheduled times.

After that, the following steps are repeated until some termination condition is reached:

Step 1 Remove the first entry (e, t) from the scheduled event list.

Step 2 Update the simulation time by advancing it to the new event time t.

Step 3 Update the state according to the state transition function $z' = next(z, e)$.

Step 4 Delete from the scheduled event list all entries corresponding to inactive events in z', i.e. delete all (e_k, t_k) such that $e_k \notin active(z')$.

Step 5 Add to the scheduled event list any active event which is not already scheduled (possibly including the triggering event removed in Step 1). The scheduled event time is computed by randomly generating a number according to the event's distribution function and adding it to the current simulation time.

Step 6 Reorder the scheduled event list such that all entries are sorted in increasing order of their scheduled times.

We prepared an experimental implementation of the event scheduling scheme in Java, using AGG [21] and the stochastic simulation library SSJ [15]. AGG is a rule based tool for graph transformation providing a visual user interface and a Java API. Models are represented by attributed graphs which are typed by type graphs. AGG is based on the single-pushout approach, but also allows to simulate the double-pushout approach by checking the identification- and dangling-edge-conditions. The implementation was used to compare two different strategies for introducing shortcuts in peer-to-peer networks [10]. Figure 5 shows the rules *limited* and *smart*. The first one adds shortcut connections whenever there is no direct connection, with a limited total number of three connections for each peer. The latter rule adds shortcuts whenever there is neither a direct connection, nor a connection via one other peer. Thus, the shortcut is only added if there is no other peer to replace peer p in case of failure [17].

Visual Modelling of stochastic graph transformation systems was done with the AGG graphical user interface. The additional information on the distribution associated with each rule was provided in a text based property file. Integration of this information into existing tools is an interesting issue as it would be more convenient for the user.

The main objective of stochastic simulation is to estimate quantities related to the modelled system by analysing the simulation results. Depending on whether we are interested in the *long run* behaviour of the system or in transient analysis, different simulation techniques have to be applied. We will shortly discuss both cases.

An interesting long run property of our network example is the proportion of unconnected peers. It is possible that a system converges to the steady-state when time progresses and can then be characterised by a discrete distribution over the state space. This is not always the case, e.g. when the number of nodes of a stochastic graph transformation system is not limited. However, we are in general not interested in the steady-state per se, but in the value of some function, such as for example the proportion of unconnected peers. These quantities may converge as $t \to \infty$, even if no steady-state is reached. A simulation strategy for the long-run behaviour can be described as follows: Let T be the length of the simulation run and let the quantity of interest be $\phi(T)$. Extend the simulation to $2T$ and determine $\phi(2T)$. If $|\phi(2T) - \phi(T)| < \varepsilon$ for some predefined ε, terminate. Otherwise, extend the simulation time T until the condition holds. Of course, this does not guarantee that $|\phi(t_1) - \phi(t_2)| < \varepsilon$ for all $t_1, t_2 > T$, because the system may fluctuate, but it is a reasonable assumption in many practical cases [5].

With transient analysis, the simulation strategy is different. Consider for instance the probability that a peer which uses the network for 1 hour suffers disconnection in this period of time. Here, it is not reasonable to extend the simulation time for more than 1 hour. One simulation run will only give one sample path, and estimates can be obtained by repeating the simulation with the same initial conditions, following the method of *independent replications* [5]. Statistical analysis of the simulation results involves computation of confidence intervals, which is automised in the SSJ framework [15]. The number of simulation runs depends on the desired confidence level. Other transient properties can even involve the simulation time itself. For instance, the average time until an error occurs might be of interest. In this case, a simulation run is performed until an error state is reached, after that the next run is started. The number of runs depends again on the desired confidence interval.

We deployed our experimental implementation to analyse the system from Example 4.3, to which either rule *limited* or rule *smart* was added as shortcut strategy, and compared the results. Both rules were assumed to obey an exponential distribution with parameter 1. As start graph, the empty graph was used. Undirected edges were modelled in AGG by bidirectional one. The property under investigation was the proportion of unconnected peers in the

long run. The simulation ran over 100,000 transitions, taking approximately 8 hours on a laptop computer with Pentium M 1.40 GHz processor and 500 MB RAM. Obviously, this time could be reduced significantly on high-capacity workstations. The result was that rule *smart* leads to 99.2% of connected nodes in the long run, while with *limited*, only 91.4% of the peers are connected.

Time measurement was done in milliseconds using the system time. As expected, almost the whole computing time was used for calculating matches. The time needed for one match differed widely depending on the rule and the size of the instance graph. While rule *new* does not require any significant computation time ($<$ 1ms), rule smart was the most time consuming rule, with a range between 100 and 400 ms for graphs of around ten nodes and between 20 and 40 edges. More detailed performance measurement is planned for the future.

The simulation algorithm involves computing all matches of rules to the current graph, and compare them to old matches, in order to compute the probabilities correctly. AGG provides an efficient graph pattern matching algorithm for calculating a single match on the basis of a constraint satisfaction problem. For every match of the current graph, two conditions need to be checked: First, we have to check whether it is a new match. This can be done easily by comparing the unique object identities of the graph elements involved. Second, the negative application conditions have to be checked again, because they may be violated by parts of the graph which had not been present before the last production. The performance of this procedure proved to be tolerable for experimental purposes. However, in order to perform more realistic simulations, performance of the implementation needs to be improved. A possible solution is provided by the database approach presented in [22]. This technique keeps track of all possible matchings of graph transformation rules in database tables, and updates these tables incrementally to exploit the fact that rules typically perform only local modifications to models. Database management systems provide efficient algorithms for computing and updating views. We plan to take advantage of them for developing an efficient tool.

6 Conclusion

In this paper, we propose a new model of Stochastic Graph Transformation with general distributions. This model allows us to study a wider class of systems than the models based on exponential distribution. However analysis of those models is more complex than the restricted one. There are no powerful model checking techniques for Generalised Semi-Markov Processes. This can be partially remedied by Monte Carlo simulation techniques.

In the future we are going to implement a tool for stochastic graph transformation with general distributions. We are going to investigate to what extend data base management systems can be used for this purpose. Our goal

is also to study more realistic examples. We will further investigate applicability of already existing simulation techniques to stochastic graph rewriting. On the other hand, we will develop a logic for specification of stochastic properties in SGT and investigate the possibilities of model checking. In many systems, there is a mix between stochastic and deterministic behaviour. We will therefore relate our model to Stochastic Automata.

Acknowledgements. Discussions with Reiko Heckel improved the ideas presented in this paper. Karsten Ehrig and Olga Runge helped us with AGG. Many thanks to all of them.

References

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, 1995.

[2] P. R. D'Argenio, J.-P. Katoen, *A theory of stochastic systems: Part I and II*, Information and Computation, Vol. 203, number 1, pages 1–38 and 39–74, 2005.

[3] P. Baldan, A. Corradini, U. Montanari, *Unfolding and Event Structure Semantics for Graph Grammars.* FoSSaCS 1999: 73-89

[4] E. Brinksma, H. Hermanns, *Process Algebra and Markov Chains*, In: Lectures on Formal Methods and Performance Analysis, editors: Ed Brinksma and H. Hermanns and J.P. Katoen, Springer LNCS 2090, pages: 183 – 231, 2001

[5] Chr. G. Cassandras, St. Lafortune, *Introduction to discrete event systems* Kluwer Academic Publishers, Boston 1999

[6] D.R. Cox, A use of complex probabilities in the theory of stochastic processes. *Proc. Camb. Phil. Soc., 51, 1955, pp. 313-319*, 51:313–319, 1955.

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

[8] P. J. Haas, G. S. Shedler, *Regenerative stochastic Petri nets*, Performance Evaluation, Volume 6, Issue 3 , September 1986, Pages 189-204

[9] Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions, *Fundamenta Informaticae*, **26**(3,4), 1996, 287 – 313.

[10] R. Heckel, *Stochastic analysis of graph transformation systems: A case study in P2P networks*, In H. Dan Van and M. Wirsing, editors, Proc. Intl. Colloquium on Theoretical Aspects of Computing (ICTAC'05), Hanoi, Vietnam, volume 3722 of LNCS. Springer-Verlag, October 2005

[11] R. Heckel, G. Lajios, S. Menge, *Stochastic Graph Transformation Systems.* In: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Hrsg.): Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28–October 1, 2004. Proceedings. Lecture Notes in Computer Science 3256. Springer, Oktober 2004. pp. 210-225

[12] J. Hillston, M. Ribaudo, *Stochastic process algebras: a new approach to performance modeling.* In: K. Bagchi, J. Walrand, G. Zobrist (Eds.), Modeling and Simulation of Advanced Computer Systems, Gordon Breach, 1998.

[13] V.G. Kulkarni, *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, 1995.

[14] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[15] P. L'Ecuyer, L. Meliani, J. Vaucher, *SSJ: A Framework for Stochastic Simulation in Java*, Proceedings of the 2002 Winter Simulation Conference, IEEE Press, Dec. 2002, 234-242.

[16] G. G. Infante López, H. Hermanns and J.-P. Katoen, *Beyond Memoryless Distributions: Model Checking Semi-Markov Chains* PAPM-PROBMIV 2001, Springer LNCS 2165, pp. 5770, 2001.

[17] L. Mariani, *Fault-tolerant routing for p2p systems with unstructured topology.* In Proc. International Symposium on Applications and the Internet (SAINT 2005), Trento (Italy), 2005. IEEE Computer Society.

[18] O. M. Mendizabal, F. L. Dotti, L. Ribeiro, *Stochastic Object-Based Graph Grammars* Brazilian Symposium on Formal Methods SBMF-2005, Porto Alegre

[19] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: foundations*, volume 1. World Scientific, River Edge, NJ, USA, 1997.

[20] R. Schassberger, *Insensitivity of steady-state distributions of generalized semi-Markov processes*, Annals of Probability, 5(1):87-99, 1977.

[21] G. Taentzer, *AGG: A Graph Transformation Environment for Modeling and Validation of Software*, Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), Pfaltz, J. and Nagl, M., Charlottesville/Virgina, USA, 2003, http://tfs.cs.tu-berlin.de/agg.

[22] G. Varró, D. Varró: *Graph Transformation with Incremental Updates.* In Proc. GT-VMT 2004, Graph Transformation and Visual Modelling Techniques, Barcelona, Spain, March 2004.

Verification of Random Graph Transformation Systems ⋆

Vitali Kozioura

Universität Duisburg-Essen, Germany
vitali.kozioura@uni-due.de

Abstract

In this paper we describe some statistical results obtained by the verification of random graph transformation systems (GTSs). As a verification technique we use over-approximation of GTSs by Petri nets. The idea of the paper is to see how many of the generated systems can be successfully verified using this technique.

Key words:

1 Introduction

In the last few years a technique for analysing of graph transformation systems (GTSs) [10] based on approximations has been developed [2]. GTSs are approximated by Petri graphs which are Petri nets with additional hypergraph structure. Petri nets can then be analyzed with standard verification techniques. A software tool (Augur) supporting this verification approach has been developed [4] and a number of successful case studies have been reported (see for example the case study on red-black trees [1]).

Still, verification remains undecidable in general (because of the Turing-completeness of GTSs). The interesting question is how many GTSs can be verified in practice using the over-approximation of GTSs by Petri nets and standard techniques for analysing Petri nets. This work is a first attempt to give an answer to this question. We generate some random GTSs and verify them with help of Augur in order to obtain statistical results. We consider some classes of GTSs identified by a number of parameters.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

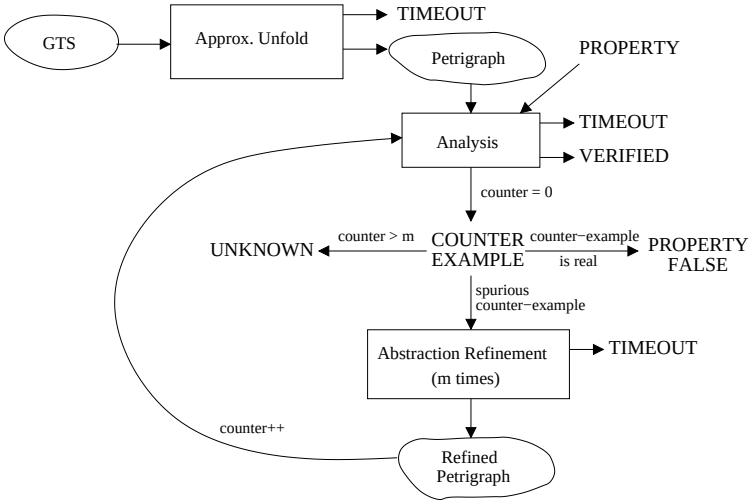


Fig. 1. Verification technique

2 Verification of Graph Transformation Systems

Fig. 1 depicts the verification technique which is used in this paper. We have a GTS and a (reachability) property [1] we want to verify as an input of the system and first of all we construct a Petri graph which is an over-approximation of a GTS having both hypergraph and Petri net structures [2]. In the analysis block we first calculate the Petri net marking corresponding to the property to verify (which is usually obtained from a regular expression on the hypergraph structure of the Petri graph [8]). The marking is then analysed with the help of a coverability algorithm [7]. If the marking is not coverable, then we terminate with "VERIFIED". This means that the corresponding subgraph (described by the regular expression and a corresponding marking) cannot be reached during the reduction of the GTS. Otherwise we have two possibilities. The obtained trace to the coverable marking (counter-example) can be real or spurious (i.e., reproducible only in the over-approximation and not in the original GTS). In the case of a real counter-example we terminate with "PROPERTY FALSE". Otherwise we start a counter-example guided abstraction refinement procedure [3,6] and obtain a refined Petri graph. The refinement procedure can be iterated a predefined number of times. If we still do not have a verification result, then we terminate with "UNKNOWN". For each operation a timeout is set. If the timeout is reached, the verification process stops with "TIMEOUT". We say that the verification problem for GTS is solved if the property is verified or we have found a (non-spurious) counter-example.

⋆ Research supported by DFG project SANDS.

[1] Checking reachability property for GTSs is already a rather complex problem, and we restrict our experiments to it, instead of checking general temporal properties of GTSs.

3 Random Graph Transformation Systems

In this paper we generate GTS with hyperedges having arity (number of connected nodes) one or two. Edges can be labeled (we consider two labels for each arity). We do not allow two edges having the same labels to be on the left-hand side of the rule. We also do not delete any nodes. Therefore we describe below only the nodes being added to the right-hand side of the rule.

The following parameters describe the class of generated GTS:

(i) Minimal/Maximal number of nodes in the left-hand side of the rule.
(ii) Minimal/Maximal number of additional nodes in the right-hand side of the rule (see the explanation above).
(iii) Minimal/Maximal number of edges in the left-hand side of the rule.
(iv) Minimal/Maximal number of edges in the right-hand side of the rule.
(v) Minimal/Maximal number of nodes in the initial graph.
(vi) Minimal/Maximal number of edges in the initial graph.
(vii) Minimal/Maximal number of rules.

In this paper we consider the following classes of random systems defined by their parameters.

(i) (1, 2; 0, 1; 1, 2; 1, 2; 2, 5; 2, 5; 3, 5)
(ii) (1, 2; 0, 2; 1, 3; 1, 3; 2, 5; 3, 7; 3, 7)
(iii) (2, 3; 1, 5; 3, 7; 3, 7; 3, 10; 3, 10; 5, 10)

Each class of GTSs is strictly included in the next one. In each class we generate 100 GTSs. The numbers are relative small because we tried to keep the sizes of generated GTSs manageable in order to obtain enough statistical material.

In each GTS we insert additionally the special rule "Error", where the left-hand side is random and the right-hand side consists of an edge labelled "Error". The property we want to verify is "the Error rule cannot be applied in the generated GTS". If the rule "Error" can be applied, then the verification algorithm (Fig 1) should give the answer "FALSE" and generates a counter-example. If the rule "Error" cannot be applied, then we should obtain the answer "VERIFIED". Fig. 2 represents an example of a generated GTS from the first class.

4 Statistical Results

The experiments have been done on 2*Xeon 2.4 GHz, 2GB RAM. We fix 3 iterations for the abstraction refinement procedure and 30 minutes as timeout value. In Table 1 average values obtained during the verification of generated systems are represented, namely the number of nodes, edges and transitions in the constructed over-approximations and the verification times (including the timeouts). The verification time is measured in seconds and represents

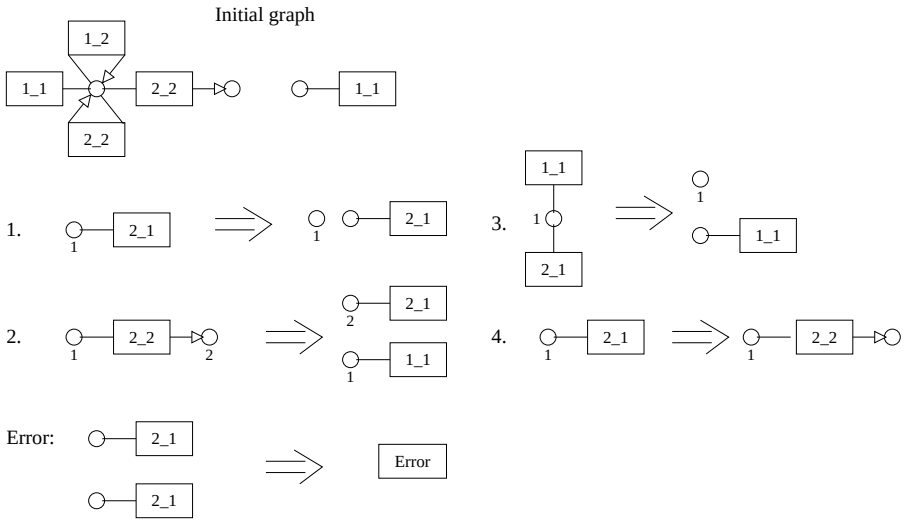


Fig. 2. Example of a generated GTS (first class of systems)

the time of the whole verification procedure.

system class	nodes	edges	transitions	verification time
1	4.21	7.67	4.07	0.01
2	7.47	14.5	10.55	59.87
3	10.01	22.28	25.78	351.53

Table 1
Average values of the verified systems.

Diagrams in Fig. 3 ((a),(b) and (c), ignore (d) for the moment) describe the distribution of the verification results for the three classes of random systems described above.

An interesting value is also the total number of refinement steps during the verification of one class of GTSs. This value grows rather quickly: 0 steps for the first class of systems, 18 steps for the second class and 83 steps for the third class. But note that the number of refinement steps for each GTS is restricted by 3.

As we can see in Fig. 3 we have successfully solved the verification problem for all 100 GTSs in the first class of systems whereas in the third class the number of problems we could not solve is one third of the number of solved systems. This pictures gives us an idea of possibilities and constraints of the verification approach based on the over-approximation of GTSs with Petri nets. To achieve better verification results we can increase the number of refinement steps and/or the timeout interval. If we start the verification procedure for the same systems from the third class with maximally five refinement steps and with two hours timeout, then we can additionally solve the

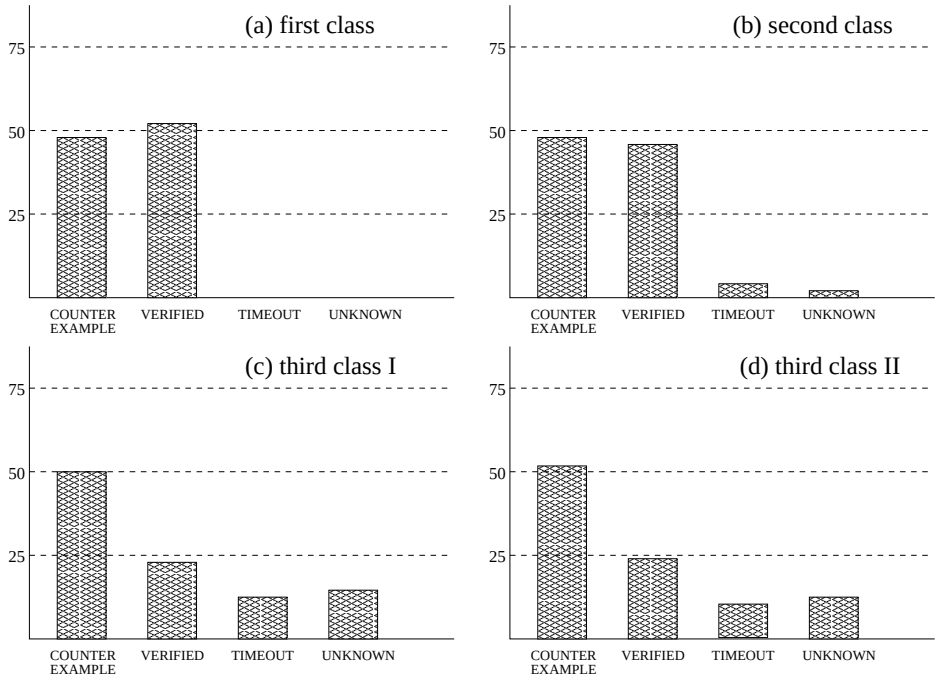


Fig. 3. Statistic of verification results

verification problem for *five* more GTSs, Fig. 3(d). The average verification results in this case are represented in Table 2. The total number of abstraction refinement steps is 109.

system class	nodes	edges	transitions	verification time
3	11.87	26.57	33.06	1273.74

Table 2
Average values for the third class with five refinement steps and two hours timeout

5 Conclusion

In this paper we considered statistical results of the verification of random GTSs by approximating them with Petri nets. The verification technique is implemented in Augur 1 and this tool has been used as a basis for our experiments. The purpose of the paper is to show how many of the random GTSs can be verified with this technique. Obviously the systems appearing in real case studies differ from random systems by having a more regular structure, but this papers gives us some (approximative) notion about the possibilities and difficulties of this approach.

The statistical results can be seen as rather positive and hence the verification approach of approximating GTSs by Petri nets can be seen as a promising approach for the verification of GTSs. Of course it will also be

necessary to compare these results with related results stemming from other methods. However we are currently not aware of any such results for random systems which have been published.

Some experimental results on the verification of GTSs have been reported in [9]. Note that we are here working in a different setting since we consider potentially infinite state GTSs, whereas [9] considers finite state GTSs.

As future work we mention here experiments on random GTSs with higher degrees of hyperedges, checking the effect of individual parameters on the results, experiments with generic systems (random GTSs generated according to some regular template), experiments with attributed GTSs and experiments with a new version of the tool Augur 2 [5], which is currently under development.

References

[1] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICAH '05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).

[2] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.

[3] E. Clarke, S. Grumberg, S. Jha, and H. Lu, Y. und Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, pages 154–169. Springer, 2000. LNCS 1855.

[4] Barbara König and Vitali Kozioura. Augur—a tool for the analysis of graph transformation systems. *EATCS Bulletin*, 87:125–137, November 2005. Appeared in The Formal Specification Column.

[5] Barbara König and Vitali Kozioura. Augur 2—a new version of a tool for the analysis of graph transformation systems. In *Proc. of GT-VMT '06 (Workshop on Graph Transformation and Visual Modeling Techniques)*, pages 195–204, 2006. ENTCS.

[6] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. of TACAS '06*, pages 197–211. Springer, 2006. LNCS.

[7] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[8] Nicolas Relange. Verifikation dynamischer Systeme: Reguläre Ausdrücke zur Spezifikation verbotener Pfade. Master's thesis, Universität Stuttgart, September 2004. No. 2192.

[9] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. ICGT 2004: Second International Conference on Graph Transformation*, volume 3256 of *LNCS*, pages 226–241, Rome, Italy, 2004. Springer.

[10] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.

Termination Criteria for DPO Transformations with Injective Matches

Tihamér Levendovszky [1]

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Hungary

Ulrike Prange [2] Hartmut Ehrig [2]

Department of Software Engineering and Theoretical Computer Science
Technical University of Berlin
Germany

Abstract

Reasoning about graph and model transformation systems is an important means to underpin model-driven software engineering, such as Model-Driven Architecture (MDA) and Model Integrated Computing (MIC). Termination criteria for graph and model transformation systems have become a focused area recently. This paper provides termination criteria for graph and model transformation systems with injective matches and finite input structure. It proposes a treatment for infinite sequences of rule applications, and takes attribute conditions, negative application conditions, and type constraints into account. The results are illustrated on case studies excerpted from real-world transformations, which show the termination properties of the frequently used "transitive closure" and "leaf collector" transformation idioms. An intuitive comparison with other approaches is also given.

Key words: Termination Criteria, Graph Transformation, Model Transformation, DPO Approach

1 Introduction

Statements about termination of graph and model transformation systems have been proven recently, and a few transformation tools already support checking termination criteria [12]. This issue has mainly arisen for the following reason. When graph transformation is used for model transformation, the

[1] Email: tihamer@aut.bme.hu
[2] Email: {uprange,ehrig}@cs.tu-berlin.de

objective is to create an output mode either from the ground up or modifying existing models. If an output model must be achieved, a transformation must provide it within a finite number of steps. Therefore, examining the termination properties of the transformation can help to find an error in the model transformation. Taking into account that one of the most important applications of graph transformation is model transformation, well-developed termination criteria can be useful support for this application area.

When transforming a model, one or more input graph, a set of rules and constraints are available along with a control structure. The nontermination can be caused by the (i) input graph or (ii) the executed sequence of the rules. In the first case several examples can be constructed that illustrate nontermination. Assume a transformation rule takes an attribute of a node, and decrements it each time when the rule is fired. The rule has a constraint that it cannot be applied for zero attribute value. When infinity is allowed as the initial value of the attribute, this rule can be applied forever. A more obvious example is an input graph with infinite size. However, in practical model transformation applications, the input model is stored on a computer or on a distributed computer system. Therefore, assuming finite input graphs does not restrict the practical scope of the results.

The nontermination caused by a sequence of finite rules is more interesting for model transformation and its tool support. In this case the transformation either becomes stagnant or starts consuming the available system resources. An example for the stagnation case would be a transformation consisting of two rules executed in a loop after each other. The first rule creates an element, the second one deletes it. The transformation does not consume all the available system resources, but never stops. In our experience, the most prevalent reason is that the designer must have forgotten a constraint from the rules, and it is really useful to warn him of this fact. When a transformation needs a growing amount of system resources, the underlying reason can be twofold. (i) This transformation needs a stronger execution environment, or (ii) the transformation is nonterminating in nature, thus, there is no execution environment strong enough to perform this transformation. For instance, if a rule creates a node and can be executed exhaustively, it never stops creating nodes. Termination analysis can be a basis to prove that a stronger computational environment is needed, or the transformation suffers from an unintended side effect.

We use the formal framework of Adhesive High-Level Replacement (AHLR) Systems [6] applied to typed attributed graphs. We assume finite input structures and rules. Since this problem is algorithmically undecidable, we prove termination properties which can be used to examine the termination properties of the individual transformations analytically.

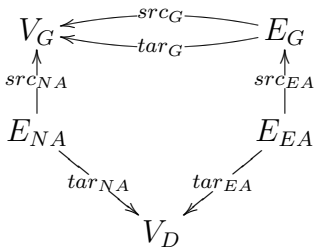
The main line of thought in this paper is as follows. The sequential rule applications are substituted with the composition of the rules. If one can show for the infinite rule sequences that the left-hand side of their composition tends

to infinity, then the rule sequence terminates, since only finitely many elements are available in the start graph. This does not necessarily hold if one element in the rule can be matched to multiple elements in the host graph. Therefore, injective matches are assumed.

2 Backgrounds

Since we use the formalism and the results of the AHLR approach, we summarize the necessary definitions and results, based on [6]. In these definitions, we always mean typed, attributed graphs by mentioning graphs, which are defined as follows.

Definition 2.1 An E-graph $EG = (V_G, V_D, E_G, E_{NA}, E_{EA}, (src_j, tar_j)_{j\in\{G,NA,EA\}})$ consists of graph and data nodes V_G and V_D, and graph, node attribute and edge attribute edges E_G, $E_{NA}$and E_{EA}, respectively. The domains and codomains of the source and target functions src_j and tar_j for the corresponding edges E_j are depicted below.



Given a signature $DSIG = (S, OP)$ with attribute value sorts $S_D \subseteq S$, an attributed graph $AG = (EG, D)$ is an E-graph EG together with a $DSIG$-algebra D such that $V_D = \dot{\bigcup}_{s\in S_D} D_s$.

Given an attributed graph TG as type graph, a (typed attributed) graph $G = (AG, t)$ is an attributed graph AG together with a typing morphism $t : AG \to TG$.

Typed attributed graphs and the corresponding morphisms form the category $\mathbf{AGraphs_{ATG}}$.

We define a function to measure the size of a graph G.

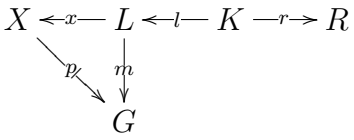
Definition 2.2 Given a graph $G = ((V_G, V_D, E_G, E_{NA}, E_{EA}, (src_j, tar_j)_{j\in\{G,NA,EA\}}), D)$, the size of G is denoted by $|G|$ and calculated as follows: $|G| = |V_G| + |E_G| + |E_{NA}| + |E_{EA}|$. G is finite if $|G| < \infty$.

We do not count the data nodes, since there may be infinitely many of them, but those relevant for the actual graph are linked by the attribute edges, which we do count. Moreover, the data part cannot be changed by applying a production.

Definition 2.3 A production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of finite graphs L, K and R, called left hand side, gluing graph and right hand side respectively, and two injective graph morphisms l and r that preserve the data part.

For practical purposes, it is important to restrict the applicability of a production by application conditions. In particular, we use negative application conditions, which forbid the existence of a certain subgraph.

Definition 2.4 A negative application condition of a production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is of the form $NAC(x)$, where $x : L \to X$ is an injective graph morphism. A graph morphism $m : L \to G$ satisfies $NAC(x)$ if there does not exist an injective graph morphism $p : X \to G$ with $p \circ x = m$.



Two graph productions (rules) are presented in Figure 1. The upper rule is applied first, as long as it can be matched against the input graph. A negative application condition ensures that at most one dashed arrow can be created between two vertices. The rule below "short-circuits" a dashed path with a length of two edges as long as possible. The resulted construct is referred to as transitive closure.

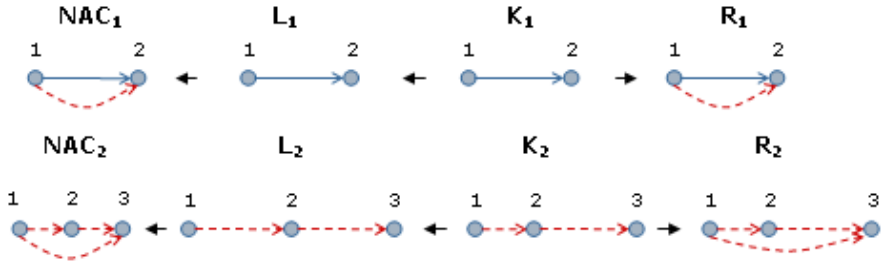
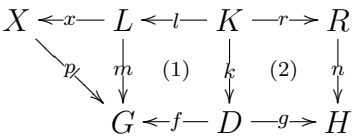


Fig. 1. Two productions computing the transitive closure

Definition 2.5 Given a graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a graph G with a graph morphism $m : L \to G$, called match. If m satisfies all negative application conditions of p, a direct graph transformation $G \xRightarrow{p,m} H$ from G to a graph H is given by the following double pushout (DPO) diagram, where (1) and (2) are pushouts.



A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of direct graph transformations is called a

graph transformation and is denoted as $G_0 \overset{*}{\Rightarrow} G_n$. For $n = 0$ we have the identical graph transformation $G_0 \overset{id}{\Rightarrow} G_0$.

We say p is applicable to G via m, if m satisfies the NACs of p, pushouts (1) and (2) exist, and the resulting graph H satisfies additional constraints given by the system. In this paper we assume injective matches m and comatches n.

Definition 2.6 A graph transformation system $GTS = (P)$ consists of a set of graph productions P with or without negative application conditions. For a graph transformation system, there may be given a set of finite input graphs.

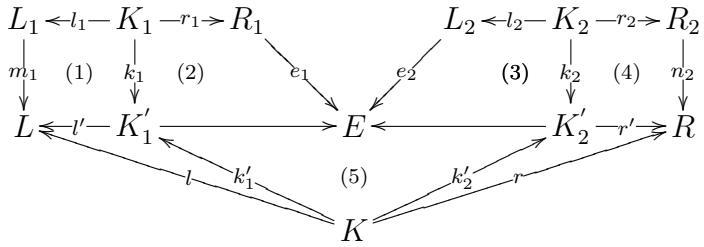
Remark 2.7 In Definition 2.6, we do not take arbitrary constraints into account. However, the results in this paper can treat all sorts of constraints, including those that are not formally defined, since their satisfaction is contained in the applicability of a production, therefore, they have been integrated into the definition dealing with the applicability of a production, and thus, it appears in Definition 3.1.

Primarily, we need a definition for the termination of a graph transformation system. We extend the definition used in [10]. In [5], the treatment of a layering control structure is added to this definition. We extend the definition in such a way that an arbitrary control structure can be handled.

Definition 2.8 A graph transformation system $GTS = (P)$ terminates if there is no infinite sequence of direct graph transformations $G_0 \Rightarrow G_1 \Rightarrow ...$ applying rules from P starting from any input graph G_0, with respect to the control structure of the given graph transformation system.

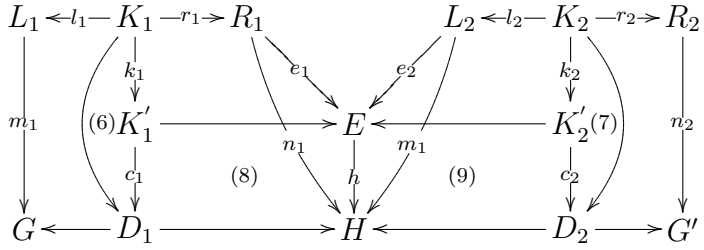
Up to now, the following definition of E-concurrent productions and the Concurrency Theorem have not been extended to productions with some kind of application conditions. Therefore we consider only plain productions in the following definition and theorem, as given in [6]. The results contributed in Section 3 are also valid when the rules contain negative application conditions.

Definition 2.9 Given two productions $p_1 = (L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1)$ and $p_2 = (L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2)$, an E-dependency relation (E, e_1, e_2) is given by a graph E and injective morphisms $e_1 : R_1 \rightarrow E$, $e_2 : L_2 \rightarrow E$, which are jointly surjective. The E-concurrent production $p_1 *_E p_2$ is a production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ computed based on the following diagram, where double squares (1)(2) and (3)(4) form double pushouts, and (5) is a pullback. Note that the injectivity of e_1 and e_2 implies that of k_1, m_1, k_2, and n_2.



This definition can be applied recursively, using an E-concurrent production for p_1.

A transformation $G \overset{p_1,m_1}{\Longrightarrow} H \overset{p_2,m_2}{\Longrightarrow} G'$ is called E-related to $p_1 *_E p_2$ if there exist morphisms $h : E \to H$, $c_1 : K_1^{'} \to D_1$ and $c_2 : K_2^{'} \to D_2$ such that $h \circ e_1 = n_1$, $h \circ e_2 = m_2$, (6) and (7) commute and (8) and (9) are pushouts.

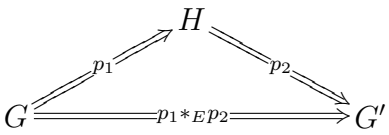


Theorem 2.10 (Concurrency Theorem) *Let (E, e_1, e_2) be an E-dependency relation for the productions p_1 and p_2 leading to the E-concurrent production $p_1 *_E p_2$.*

(i) *Synthesis: Given an E-related transformation sequence $G \Rightarrow H \Rightarrow G'$ via p_1 and p_2, then there is a synthesis construction leading to a direct transformation $G \Rightarrow G'$ via $p_1 *_E p_2$.*

(ii) *Analysis: Given a direct transformation $G \Rightarrow G'$ via $p_1 *_E p_2$, then there is an analysis construction leading to an E-related transformation $G \Rightarrow H \Rightarrow G'$ via p_1 and p_2.*

(iii) *Bijective correspondence: The synthesis and analysis constructions are inverse to each other up to isomorphism.*



3 A General Criterion for Injective Matches

In this section, we provide a general approach for termination within the scope of the DPO approach. These results also apply when the rules contain negative application conditions and other constraints.

Definition 3.1 An E-concurrent production p^* is an E-based composition if there is at least one input graph G_0 with an E-related transformation

$G_0 \overset{p^*}{\Longrightarrow} H$.

This definition is required, because for the DPO approach, the definition of E-concurrent productions and the Concurrency Theorem have not been extended to handle negative application conditions and other constraints. Moreover, this definition guarantees, among others, that the constraints enforced by p_1 do not contradict to the constraints necessary for the application of p_2. Typical examples for constraints are attribute constraints, negative application conditions, type conformance for metamodels, constraints from the control flow branches, but other constructs are also possible.

In most of the practical cases it is simple to find such an input graph for an E-concurrent production, when there is no contradiction between the rules. Then, the left hand side of this rule is already an input graph, or it can be extended with regard to possible constraints.

An example for composing the bottom rule in Figure 1 with itself via a chosen E_1 is depicted in Figure 2.

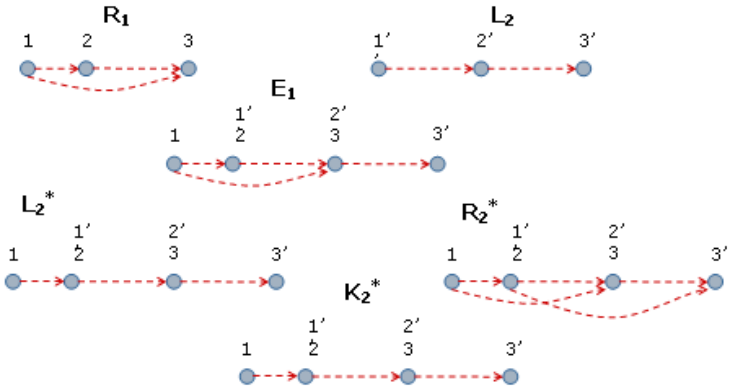


Fig. 2. An E-based composition of Rule 2 with itself

Definition 3.2 Consider a possibly infinite sequence of graph productions p_i, $(i = 1, 2, ...)$ and a sequence of E-dependency relations $((E_i, e_i^*, e_{i+1}))$ leading to a sequence of their E-based compositions $(p_i^* = (L_i^* \leftarrow K_i^* \rightarrow R_i^*))$ with $p_1^* = p_1$ and $p_n^* = (p_1 *_{E_1} p_2) *_{E_2} ... *_{E_n} p_n$.

A cumulative LHS series of this sequence is the graph series L_n^* consisting of the left-hand side graphs of p_n^*. Moreover, a cumulative size series of a production sequence is the nonnegative integer series $|L_n^*|$.

It is possible that there are several cumulative LHS series of a given production sequence, since, in general, two rules can be composed in different ways, choosing different E-dependency relations. For instance, if we want to compute p_3^* for our example, then we take the cumulative rule p_2^*, and compose it with the bottom rule from Figure 1. There are several possibilities to choose E_2: (i) we can short-circuit the path $1 - 2 - 3'$, (ii) $1 - 3 - 3'$, or (iii) R_2^* and L do not fully overlap. It is easy to see that in the first two cases L_3^* is isomorphic to L_2^*, but in the third case L_2^* must be extended to obtain L_3^*.

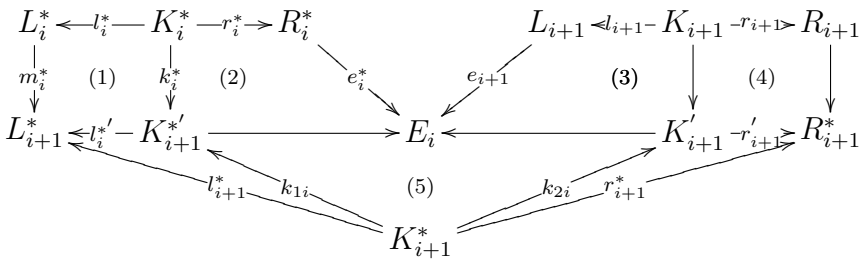
However, there is no case, when L_3^* is smaller than L_2^*. If we consider injective matches only, this is true for the DPO approach in general.

Lemma 3.3 *The sequence $|L_i^*|$ (Def. 3.2) is monotonic nondecreasing. If $E_i \cong R_i^*$, L_i^* remains unchanged, thus, $|L_i^*| = \left|L_{i+1}^*\right|$. Otherwise, $L_i^* \ncong L_{i+1}^*$ and $|L_i^*| < \left|L_{i+1}^*\right|$, but L_{i+1}^* always contains an isomorphic subgraph of L_i^*.*

Proof. Since there is an injective morphism $m_i^* : L_i^* \to L_{i+1}^*$, we have $|L_i^*| \leq \left|L_{i+1}^*\right|$, and L_{i+1}^* contains an isomorphic subgraph of L_i^*.

Pushouts along isomorphisms are pullbacks, and pushouts and pullbacks are closed under isomorphism. Therefore, if $E_i \cong R_i^*$, we have isomorphisms k_i^* and m_i^*, which means that $L_i^* \cong L_{i+1}^*$ and $|L_i^*| = \left|L_{i+1}^*\right|$.

If $E_i \ncong R_i^*$, there are items $x \in E_i \backslash e_i^*(R_i^*)$, which have preimage in $K_{i+1}^{*'}$ but not in K_i^*, because (2) is a pushout. For (1) being a pushout, these items have to be added to L_i^* to obtain L_{i+1}^*, therefore $L_i^* \ncong L_{i+1}^*$, and $|L_i^*| < \left|L_{i+1}^*\right|$. □



Theorem 3.4 *A GTS $= (P)$ (Def. 2.6)terminates if for all infinite cumulative LHS sequences (L_i^*) of the graph productions created from the members of P, it holds that*

$$\lim_{i \to \infty} |L_i^*| = \infty.$$

Note that we assume finite input graphs and injective matches.

Proof. We rely on the fact that if the constraints are satisfied, the E-based compositions are E-concurrent productions as well. Therefore, we can apply Theorem 2.10 for the topological part of the transformation, when we can assume that the constraints hold, which means the existence of the E-based composition.

In $\mathbf{AGraphs_{ATG}}$, Theorem 2.10 holds. Suppose there is an infinite transformation $G_0 \overset{p_1}{\Rightarrow} G_1 \Rightarrow \cdots$. Then there is a sequence of E-concurrent productions p_i^* leading to the transformations $G_0 \overset{p_i^*}{\Rightarrow} G_i$ (Theorem 2.10). All these productions are also E-based compositions with cumulative LHS series L_i^*. Since $\lim_{i \to \infty} |L_i^*| = \infty$, there exists an $N \in \mathbb{N}$ with $|G_0| < |L_N^*|$. But this means that there is no injective match $m_N^* : L_N^* \to G_0$, i.e. p_N^* is not applicable to G_0. □

The opposite direction of Theorem 3.4 does not hold in general, but for a finite number of input graphs. In this case, no infinite sequences of E-based compositions can be constructed.

Theorem 3.5 *If a GTS = (P) (Def. 2.6) terminates and we have only a finite number of input graphs up to isomorphism, then there are no infinite cumulative LHS sequences* (L_i^*) *of graph productions created from the members of P.*

Proof. Assume that *GTS* terminates and there is an infinite sequence (p_i^*) of E-based compositions.

For each p_i^* there exists an input graph G_i with an E-related transformation $G_i \overset{p_i^*, m_i}{\Longrightarrow} H_i$. Since there are only finite many input graphs, at least one of them has to appear infinitely many often. This means we have an input graph G with $\forall N \in \mathbb{N} \exists j > N : G \overset{p_j^*}{\Longrightarrow} H_j$. From Theorem 2.10 it follows that all p_i^* are applicable to G leading to an infinite transformation sequence. □

From Theorem 3.4 the next statement follows:

Lemma 3.6 *If* $L_i^* \ncong L_{i+1}^*, \forall i$ *for every cumulative LHS series (Def. 3.2), then the GTS terminates. If each graph appears only finitely many times in all cumulative LHS series, the GTS still terminates.*

Proof. Considering the first statement of the lemma, if two subsequent graphs in the cumulative LHS sequence are not isomorphic, they must grow in size because of Lemma 3.3. According to Theorem 3.4, this means that the GTS terminates.

Taking a cumulative LHS series at any position i, it grows in size within finite number of steps if there are only finite number of graphs in the series that are isomorphic to L_i^*. The series must grow because of Lemma 3.3. Then we have $\lim_{i \to \infty} |L_i^*| = \infty$, and the GTS terminates because of Theorem 3.4.□

4 Case Studies

To show the practical relevance of the presented termination criteria, two case studies are provided. We take two transformation idioms from [1], and analyze their termination properties. Obviously, there are other proofs for these case studies, but we would like to illustrate how the technique contributed in this paper works for practical software model transformations, where the tool supports strict control flow constructs.

4.1 Transitive Closure

Using Theorem 3.4, we show that the transitive closure terminates. This is a frequently used transformation pattern. In case of variation of the 'class

model to relational database management system (RDBMS) model' transformation [12] (also referred to as object-relational mapping), the traversal of the inheritance hierarchy and the association chains are performed using the transitive closure pattern.

Lemma 4.1 *The injective application of the transitive closure rule (the bottom rule in Figure 1) terminates for all finite input graphs.*

Proof. There are two cases. (i) When constructing E_k, it is not isomorphic to R_k^*. This means that in this case L_k^* must be extended to obtain L_{k+1}^* by Lemma 3.3. Therefore, in these steps, the cumulative LHS series grows. (ii) The other case needs more attention, since the cumulative LHS series does not grow in every step this time. We show that at a given stage of the transformation, this is possible finite times only. Suppose $E_k \cong R_k^*$ when constructing p_{k+1}^* from p_k^* and the original rule p. This leads only to a valid E-based composition if there is no dashed edge between $e_{k+1}(1)$ and $e_{k+1}(3)$ in E_k. In R_{k+1}^* no new nodes are added, but an additional edge (compared with R_k^*). Thus, after finite many steps we can only construct E-concurrency relations not isomorphic to the right hand side. This stems from the fact that the negative application condition forbids creating dashed edges between the nodes where there is one already. This means that an LHS can appear only a finite number of times in the cumulative LHS sequence, therefore, according to Lemma 3.6 the GTS terminates. □

4.2 Leaf Collector

The *LeafCollector* pattern is used to find the leaf elements in a tree structure. This idiom has been distilled from the transformation flattening a hierarchical data flow diagrams to a flat data flow representation [1]. In fact, *LeafCollector* does not modify the input graph, but finds a place where the next rule can be applied. Therefore, *LeafCollector* is a useful idiom of many software model transformations, and it is worth examining its termination properties.

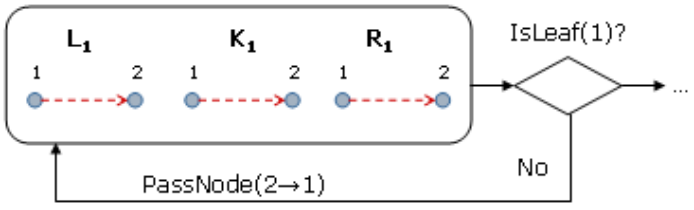


Fig. 3. The Leaf Collector Transformation Idiom

A possible formulation of the pattern is depicted in Figure 3. This idiom is particularly interesting, because it strongly builds on a sophisticated control structure of the transformation tool. The diamond in the figure can be implemented in several ways. In GReAT [1], it is implemented as test rule,

whereas it is a branch condition in VIATRA [13] and VMTS [11]. The other required feature is parameter passing. This means that host graph nodes and edges matched in one of the previous rules can be passed to a subsequent rule. The matching algorithm considers these elements already bound. This can accelerate the matching process, and facilitates the separation of complex rules. If there are no passed parameters, the matching algorithm starts to match with unbound elements. In our example, the rule is bound to any of the suitable places in the input graph on the first execution. On the subsequent runs, the graph node matched to the rule node 2 is passed to the rule node 1. Therefore, the matching algorithm finds a node adjacent to the one passed as a parameter. The output of this idiom is *node*1 when it is a leaf. Therefore, *node*1 is passed further along the branch where the ellipses are depicted. The parameter passing mechanism is implemented with different syntax in the aforementioned tools, thus, we focus on the notion only without formalizing it. From the mathematical point of view, this construct is modeled as a restriction on the possible E-based compositions.

Since the idiom is obviously not concerned with self-loops, injective matches are assumed. Then we compute the E-based composition of the rule with itself. In this case it is rather simple, because the parameter passing reduces the number of the possible E-dependency relations to one.

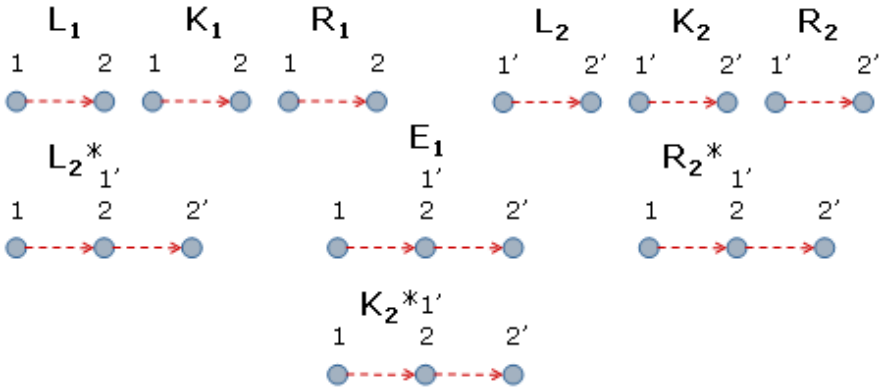


Fig. 4. E-based Composition for Leaf Collector - Acyclic Case

Lemma 4.2 *The transformation Leaf Collector (depicted in Figure 3) terminates if and only if the input graph does not contain a directed cycle.*

Proof. Firstly, we compute the E-based composition of the rule with itself.

Because of the parameter passing, the E_i is created as follows: R_i^* contains only one node n_i that is a target of an incoming edge and it is not a source of any outgoing edge. Then the node $1^{(i)}$ in L_{i+1} is mapped to n_i in E_i, the others are mapped to different vertices and edges.

$1^{(i)}$ in L_{i+1} can either be mapped to an E_i element that is not mapped to any R_i^* element, or otherwise. Based on that there are two possible categories of E-dependency relations. The first option is depicted in Figure 4 for p_2^*.

Since there are only control conditions, it is the same as the E-concurrent production, where the E_1 is determined by the parameter passing. The control structure limits the number of the composed productions only. Obviously, L_i^*, K_i*, and R_i^* are the same, because the rule does not change the input graph: it searches for a specific element.

Pushouts along isomorphisms are pullbacks, and pushouts and pullbacks are closed under isomorphism. Therefore, $E_i \cong L_{i+1}^*$. Thus, L_i^* is a directed path consisting of i edges. This means that in this case the transformation terminates according to Theorem 3.4.

When $1^{(i)}$ in L_{i+1} is mapped to an E_i element that is mapped to any R_i^* element, it automatically creates a directed cycle. An example for this structure is depicted in Figure 5. In this case it is possible that we have a nonincreasing cumulative size series. According to Theorem 3.5, it is possible that this structure does not terminate. □

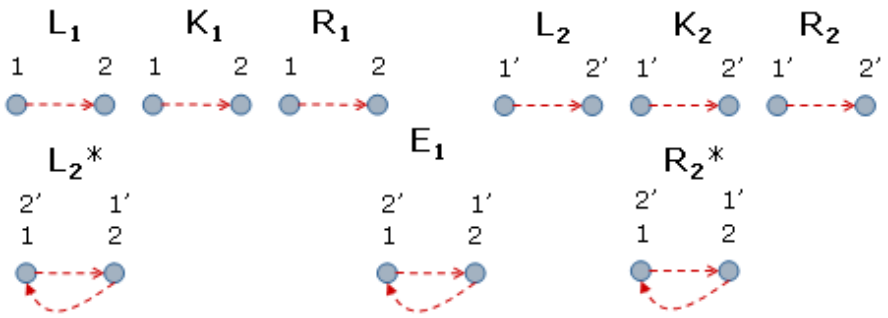


Fig. 5. E-based Composition for Leaf Collector - Cyclic Case

According to Lemma 4.2, if the input graph does not contain directed cycles, the transformation terminates, otherwise it is possible that the transformation does not terminate. In practice, this condition can be guaranteed in model transformation systems. (i) Most of the modeler tools offer a containment hierarchy, and along this hierarchy it is ensured by the tool that there are no directed cycles. (ii) Directed cycles in inheritance hierarchy causes semantical problems, it may also be forbidden by the tool.

If there are no such constraints in the model, the Leaf Collector should be extended with additional construct in order to avoid nontermination. A possible solution is to add an *isProcessed* attribute to the nodes, which is *false* by default, and set by the rule if it is matched. Another solution is to introduce helper edges between the processed nodes, and introduce NACs to forbid the match at the same place again.

This case study illustrates that with the proposed termination analysis method, we could obtain the structure that causes the nontermination. Therefore, this technique is suitable for constructive analysis besides the decision issues.

5 Related Work

In [2], termination criteria have been developed for graph rewriting applied to program transformation. The criteria aim at this specific problem domain. The approach assumes that there can be no parallel edges with the same labels between two nodes. This leads to a termination criteria for specific (edge-accumulative) rules if the label and node sets are finite. Moreover, subtractive rules are investigated, which are conceptually similar to deletion layers examined in [5]. These results assume more restricted types of rules, compared to those analyzed in this paper.

In [3], a theory has been developed for the DPO approach. It provides abstract termination criteria by a measure function F. The paper also shows concrete termination criteria such as the number of nodes, the number of edges. Based on this assumption, it proves termination criteria for other control structures. However, these criteria are violated in the second case study with respect to the concrete criteria of edge and node numbers. However, no explicit relationship has been established between the proposed definition of a termination criterion and the notion that the transformation stops within a finite number of steps.

In [5], results have been developed for layered grammars. These results formalize and extend the contributions provided in [7] [4]. The provided criterion ensures that the creation of all objects of a type should precede the deletion of the object of this type. Therefore, a layer deleting an object of a given type cannot create such an object, nor the subsequent rules. This means that the productions in a deletion layers terminate for the reasons detailed above if the types are taken into consideration.

A nondeletion layer cannot contain rules that delete a node. It is ensured by a negative application condition that a rule cannot be applied twice at the same match. Furthermore, if a rule creates an object of a given type, it is not allowed to match any object of that type in that or any subsequent layers. Since Layer 0 uses the finite input graph, and there cannot be a match at the same place, and the rules in Layer 0 cannot create elements of a type whose instances they match, the rules can be executed only a finite number of times. The next layer terminates for similar reasons: it can only use elements of a type whose instances have already been created. Since Layer 0 has terminated, Layer 1 is passed a finite graph, thus, the situation is similar to that in case of Layer 0.

In our context, this means that only a finite fully overlapping ($E_i \cong R_i^*$) sequences are possible, since the the NAC forbids the E-based composition at a given position more than once. Otherwise $|L_i^*|$ must increase. Unfortunately, there are situations, where these criteria do not hold. In our first case study, the second rule matches and creates an element of the same type.

The methods discussed as related work are not restricted to injective matches as opposed to our approach.

6 Conclusions

A novel contribution of this paper is to provide termination criteria for general productions allowing recursion within the scope of DPO and typed attributed graph transformation, assuming injective matches. This can be a theoretical basis to prove that certain control flows of rules are terminating, where the other - algorithmically underpinned - criteria cannot be applied. In general, however, it is hard to find all the possible sequences of graph productions, and prove that the corresponding series $|L_i^*|$ exceeds all limits. This is expected, since the termination of a GTS is undecidable [10]. However, the stricter and the more deterministic the ordering of the rules is, the higher is the chance that we can deal with the sequences. For example, in the tool Visual Modeling and Transformation System (VMTS) [11], the control structures are as strict as possible, and nondeterminism is avoided if possible. Moreover, parameter passing between the rules (external causalities) decrease the number of the possible E_k graphs, since the nodes and edges connected by a morphism from R_k^* to L_{k+1} must be mapped to the same nodes and edges in E_k. We have also contributed two case studies, which solve the termination issue of two frequently used transformation idioms called "transitive closure" and "leaf collector".

Another contribution is that in the composition of the productions in Definition 3.1, attributes are also considered, and the proposed method is open to other constraint specification approaches. Furthermore, it regards control structures and parameter passing.

Future work includes the extension of these results to noninjective matches. Furthermore, constraint checking to decide whether a composition rule exists is not simple in the general case, when not only the attributes set by the transformation steps are considered. Also, we would like to analyze more idioms and frequent building blocks. A library of building blocks with proven termination properties may help the tools to overcome the algorithmic undecidability. Since where the algorithms fail, the structural investigation can offer a solution.

7 Acknowledgments

The activities described in this paper were supported, in part, by the SegraVis Training Network and by the National Office for Research and Technology (Hungary).

References

[1] Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G: Reusable Idioms and Patterns in Graph Transformation Languages 2004. Proc.

2nd International Workshop on Graph Based Tools (GraBaTs 2004). Satellite workshop of ICGT 2004, Rome, Italy, 2004.

[2] Assmann, U., Graph rewrite systems for program optimization, ACM TOPLAS 22, 2000, pp. 583-637

[3] Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: "Termination of High-Level Replacement Units with Application to Model Transformation", VLFM 2004, Electronic Notes of Theoretical Comp.Sci. (ENTCS) vol.127, no.4 (2005), Elsevier, pp. 71-86.

[4] Bottoni, P., Taentzer, G., Schuerr, A. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In Proc. Visual Languages 2000 IEEE Computer Society. pp.: 59-60.

[5] Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay,Sz.: "Termination Criteria for Model Transformation", FASE 2005, LNCS, pp. 49-63.

[6] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: "Fundamentals of Algebraic Graph Transformation", EATCS Monographs in Theoretical Computer Science, Springer, 2006

[7] de Lara, J., Taentzer, G. 2004. Automated Model Transformation and its Validation with AToM3 and AGG. In DIAGRAMS2004 (Cambridge, UK). Lecture Notes in Artificial Intelligence 2980, pp.: 182198. Springer.

[8] Lengyel, L., Levendovszky, T., Charaf, H.: "Eliminating Crosscutting Constraints from Visual Model Transformation Rules", ACM/IEEE 7th International Workshop on Aspect-Oriented Modeling, Montego Bay, Jamaica, October 2, 2005.

[9] Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: "A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS", Electronic Notes in Theoretical Computer Science, International Workshop on Graph-Based Tools (GraBaTs) Rome, 2004.

[10] Plump, D.: "Termination of graph rewriting is undecidable", Fundamenta Informaticae, 33(2):201209, 1998

[11] VMTS Web Site, `http://avalon.aut.bme.hu/~tihamer/research/vmts`

[12] Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., Varró-Gyapay, Sz.: Model Transformation by Graph Transformation: A Comparative Study,ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, 2005

[13] Varró, D.: Automated Model Transformations for the Analysis of IT Systems. PhD thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems (2004)