Math. Struct. in Comp. Science (2014), vol. 24, iss. 4, e240406, 48 pages. © Cambridge University Press 2014
doi:10.1017/S0960129512000357

$\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation

HARTMUT EHRIG[†], ULRIKE GOLAS[‡], ANNEGRET HABEL[§], LEEN LAMBERS[¶] and FERNANDO OREJAS[‖]

[†] *Technische Universität Berlin, Berlin, Germany*
Email: ehrig@cs.tu-berlin.de
[‡] *Konrad-Zuse-Zentrum für Informationstechnik Berlin, Berlin, Germany*
Email: golas@zib.de
[§] *Universität Oldenburg, Oldenburg, Germany*
Email: habel@informatik.uni-oldenburg.de
[¶] *Hasso-Plattner-Institut, Universität Potsdam, Potsdam, Germany*
Email: leen.lambers@hpi.uni-potsdam.de
[‖] *Universitat Politècnica de Catalunya, Barcelona, Spain*
Email: orejas@lsi.upc.edu.

Received September 2011; revised January 2012

Nested application conditions generalise the well-known negative application conditions and are important for several application domains. In this paper, we present Local Church–Rosser, Parallelism, Concurrency and Amalgamation Theorems for rules with nested application conditions in the framework of $\mathcal{M}$-adhesive categories, where $\mathcal{M}$-adhesive categories are slightly more general than weak adhesive high-level replacement categories. Most of the proofs are based on the corresponding statements for rules without application conditions and two shift lemmas stating that nested application conditions can be shifted over morphisms and rules.

1. Introduction

Standard graph transformation systems have been studied extensively and applied in several areas of computer science (Rozenberg 1997; Ehrig *et al.* 1999a; Ehrig *et al.* 1999b). To cope with the different varieties of graphical structures, they were, first, generalised to high-level replacement (HLR) systems (Ehrig *et al.* 1991) and then, based on the notion of adhesive categories (Lack and Sobocinski 2005), to weak adhesive HLR systems (Ehrig *et al.* 2006a; Ehrig *et al.* 2006b) and more recently to $\mathcal{M}$-adhesive systems[‡] (Ehrig

[¶] The work of Leen Lambers was funded by the Deutsche Forschungsgemeinschaft in the course of the project 'Correct Model Transformations' – for further details, see http://www.hpi.uni-potsdam.de/giese/projekte/kormoran.html?L=1.

[‡] An *$\mathcal{M}$-adhesive system* consists of an $\mathcal{M}$-adhesive category and a set of rules over the category.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

	Category	Class $\mathcal{M}$
graph	graphs	injective
high-level	hypergraphs	injective
	algebraic specifications	injective & strict
weak adh. HLR	typed attributed graphs	injective with isomorphism on the data part
	place/transition nets	monomorphisms
$\mathcal{M}$-adhesive	list sets	monomorphisms

Fig. 1. $\mathcal{M}$-adhesive categories

	Applications with ACs
none	library system (Ehrig and Kreowski 1980)
	telephone system (Reibeiro 1996)
	client-server system (Corradini *et al.* 1997)
negative	library system (Ehrig *et al.* 2010b)
	elevator control (Habel *et al.* 1996; Lambers 2010)
	railroad control (Pennemann 2009)
nested	access control, mobile communication, carplatoon
	manoeuvre protocol (Pennemann 2009)
	model transformation (Golas 2011)

Fig. 2. Application domains where nested application conditions are used

et al. 2010a). There is a proper hierarchy of categories: graph ⇒ high-level ⇒ weak adh(esive) HLR ⇒ $\mathcal{M}$-adhesive – categories that show that the implications are proper are given in Ehrig *et al.* (2006b) and Ehrig *et al.* (2010a). Some examples of $\mathcal{M}$-adhesive categories are given in Figure 1.

Originally, application conditions (ACs), as defined in Ehrig and Habel (1986), were very simple. They were restricted to specifying that a certain graph should not include the match of the rule. For this reason, they were called Negative Application Conditions (Habel *et al.* 1996). This kind of condition is useful in many cases, but is too restrictive in some other cases. As a consequence, they were generalised to nested application conditions in Habel and Pennemann (2009). Nested application conditions can be shown to be expressively equivalent to first-order graph formulas (Courcelle 1997), where one part of the proof is similar to the translation between first-order logic and predicates on edge-labelled graphs with single edges in Rensink (2004). There is a proper hierarchy of types of application conditions: no ACs ⇒ negative ACs ⇒ nested ACs. Some examples of application domains where nested application conditions are used are given in Figure 2. This means that even if nested application conditions do not add any difficulty (undecidability), these results show that, in principle, the expressive power of nested application conditions is no smaller than the expressive power of conditions in term rewriting. However, given their different nature, they are difficult to compare.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Theorem	Informal description	Application areas
Local Church–Rosser	*Parallel independent* transformations applied to the same object can be applied in arbitrary order and still lead to the same result.	database systems (Ehrig and Kreowski 1980) process control (Mahr and Wilharm 1982) algebraic specifications (Parisi-Presicce 1989) distributed information systems (Heckel *et al.* 2002)
Parallelism	*Parallel independent* transformations can be parallelised to a single transformation using the *parallel rule*. *Sequentially independent* transformations can be reduced to a single transformation using the *parallel rule*.	database systems (Ehrig and Kreowski 1980) algebraic specifications (Parisi-Presicce 1989)
Concurrency	Allows *sequentially dependent transformations*. *Related* transformations can be reduced to a single transformation using the *concurrency rule*.	logic programming (Corradini *et al.* 1991)
Amalgamation	Allows *parallel dependent* transformations: *Amalgamable* transformations can be amalgamated to a single transformation using the *amalgamation rule*.	distributed systems (Castellani and Montanari 1983; Boehm *et al.* 1987; Degano and Montanari 1987; Taentzer *et al.* 1999; Golas 2011) model transformation (Biermann *et al.* 2010)

Fig. 3. Informal descriptions of the results with some application areas

The literature on (graph) transformation systems contains a number of results known as Local Church–Rosser, Parallelism, Concurrency and Amalgamation Theorems. Informal descriptions of these results, together with some application areas, are given in Figure 3.

The Local Church–Rosser, Parallelism and Concurrency Theorems were first presented for graph transformation systems on rules without application conditions in Rosen (1975), Kreowski (1977a), Ehrig (1979), Ehrig and Rosen (1980) and Ehrig *et al.* (1986) and were later generalised to high-level replacement systems (Ehrig *et al.* 1991) and rules with negative application conditions (Lambers 2010). The Amalgamation Theorem was presented for graph transformation systems on rules without application conditions (Boehm *et al.* 1987; Corradini *et al.* 1997).

The aim of the current paper is to show that results in the literature based on rules without (Ehrig *et al.* 2006b) or with (Lambers 2010) negative application conditions

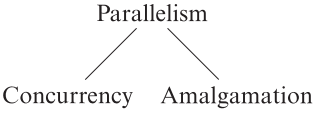
https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

can be generalised to nested application conditions (Habel and Pennemann 2009) in the framework of $\mathcal{M}$-adhesive transformation systems. In order to increase the expressive power of graph transformation systems, for several applications it is important to consider not only negative application conditions but also nested ones. The presentation of the main results in the categorical framework of $\mathcal{M}$-adhesive categories is also highly relevant, since in this way the results are not only valid for classical graph transformation systems, but also for transformation systems based on typed and attributed graphs, hypergraphs, and different kinds of low- and high-level Petri nets (Ehrig *et al.* 2006b).

We will state the Local Church–Rosser, Parallelism, Concurrency and Amalgamation Theorems for $\mathcal{M}$-adhesive systems on rules with nested application conditions. The proofs of the Local Church–Rosser, Parallelism, and Concurrency Theorems are based on the corresponding theorems for $\mathcal{M}$-adhesive systems on rules without application conditions given in Ehrig *et al.* (2006b) together with two shift lemmas for nested application conditions (ACs), which extend those given in Habel and Pennemann (2009), and state that application conditions can be shifted over morphisms and rules.

Theorem + shift lemmas for ACs $\Rightarrow$ Theorem for rules with ACs

The Amalgamation Theorem for $\mathcal{M}$-adhesive systems on rules with nested application conditions can be considered to be a special case of a recent construction, called multi-amalgamation, studied in Golas *et al.* (2014). The Concurrency and Amalgamation Theorems may be viewed as two different generalisations of the Parallelism Theorem: in the first case, sequential independence is dropped and, in the second case, parallel independence.



1.1. *Organisation of the paper*

In Section 2, we review the definition of an $\mathcal{M}$-adhesive category. In Section 3, we introduce rules with nested application conditions. In Section 4, we state and prove the Local Church–Rosser, Parallelism and Concurrency Theorems. In Section 5, we define amalgamated rules and state the Amalgamation Theorem. In Section 6, we describe some related work. Finally, in Section 7, we give an overview of the results of $\mathcal{M}$-adhesive transformation systems with nested application conditions. The concepts are illustrated by examples in the category of directed, labelled graphs with the class of all injective graph morphisms. To motivate the rules with nested conditions, and to help give an understanding of the main concepts, we will present a running example describing a mutual exclusion algorithm closely following Dijkstra's work.

The paper is a long version of the paper Ehrig *et al.* (2010b), and contains a new section on amalgamation as well as a new illustrative example.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

2. Graphs and high-level structures

In this section, we will recall the basic notions of directed, labelled graphs (Ehrig 1979; Corradini *et al.* 1997), and generalise them to high-level structures (Ehrig *et al.* 1991). The motivation behind our considering high-level structures is to avoid repeating similar investigations for similar structures such as Petri nets and hypergraphs. We assume familiarity with the basic notions of graph transformation systems and the basic concepts of category theory – standard references are Ehrig (1979), Arbib and Manes (1975) and Adamek *et al.* (1990).

Directed, labelled graphs and graph morphisms are defined as follows.

Definition 2.1 (graphs and graph morphisms). Let $\mathrm{L} = (\mathrm{L_V}, \mathrm{L_E})$ be a fixed, finite label alphabet. A *graph* over L is a system

$$G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_G, \mathrm{m}_G)$$

consisting of: two finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*; *source* and *target functions* $\mathrm{s}_G, \mathrm{t}_G : \mathrm{E}_G \to \mathrm{V}_G$; and two *labelling functions* $\mathrm{l}_G : \mathrm{V}_G \to \mathrm{L_V}$ and $\mathrm{m}_G : \mathrm{E}_G \to \mathrm{L_E}$. A graph with an empty set of nodes is *empty* and denoted by $\varnothing$. A *graph morphism* $g : G \to H$ consists of two functions $g_\mathrm{V} : \mathrm{V}_G \to \mathrm{V}_H$ and $g_\mathrm{E} : \mathrm{E}_G \to \mathrm{E}_H$ that preserve sources, targets, and labels, that is,

$$\begin{aligned} \mathrm{s}_H \circ g_\mathrm{E} &= g_\mathrm{V} \circ \mathrm{s}_G \\ \mathrm{t}_H \circ g_\mathrm{E} &= g_\mathrm{V} \circ \mathrm{t}_G \\ \mathrm{l}_H \circ g_\mathrm{V} &= \mathrm{l}_G \\ \mathrm{m}_H \circ g_\mathrm{E} &= \mathrm{m}_G. \end{aligned}$$

A morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and it is an *isomorphism* if it is both injective and surjective. In the latter case, G and H are *isomorphic*, which is denoted by $G \cong H$. The *composition* $h \circ g$ of g with a morphism $h : H \to M$ consists of the composed functions $h_\mathrm{V} \circ g_\mathrm{V}$ and $h_\mathrm{E} \circ g_\mathrm{E}$. The category having graphs as objects and graph morphisms as arrows is called Graphs.

Example 2.2. To illustrate our definitions and results in the following sections, we will use an example describing a mutual exclusion algorithm closely following Dijkstra's work (Dijkstra 1965). We begin by introducing the labels and underlying system models. In our system, we have an arbitrary number of processes `P` and resources `R`. To each resource, a turn variable `T` may be connected assigning this resource to a process. Each process may be `idle` or `active` and has a flag with possible values $0, 1, 2$, initially set to 0, which is graphically described by no flag at all at this process. Moreover, a label `crit` marks a process that has entered its critical section and is currently using the resource. Thus, the label alphabet used for our example is

$$\mathrm{L} = (\mathrm{L_V}, \mathrm{L_E})$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

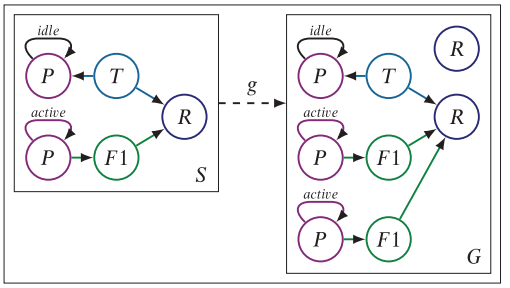


Fig. 4. (Colour online) Example graph and graph morphism.

with

$$\mathrm{L_V} = \{\mathtt{P}, \mathtt{T}, \mathtt{R}, \mathtt{F1}, \mathtt{F2}\}$$
$$\mathrm{L_E} = \{\mathtt{active}, \mathtt{idle}, \mathtt{crit}, \lambda\}.$$

On the left-hand side of Figure 4, we model a system S containing a resource and two processes, one of them being `active` and one of them `idle`, where the active process is connected to the resources via an `F1`-flag and the other process is conneted to them via the turn variable. There is an injective graph morphism $g : S \to G$ extending S by another active process with a flag to the resource and an additional resource that has no turn variable and is thus disabled.

In drawings of graphs, nodes are represented by circles and edges by arrows pointing from the source to the target node. Arbitrary graph morphisms are drawn by the usual arrows '$\to$'; the use of '$\hookrightarrow$' indicates an injective graph morphism, but we will only use it if we need to point it out explicitly. The actual mapping of the elements will be shown by positions, or by indices where necessary.

While the original double-pushout approach was defined on directed, labelled graphs (Ehrig *et al.* 1973; Ehrig 1979), it was later lifted to a categorical setting using a distinguished morphism class $\mathcal{M}$, with various instantiations. In particular, adhesive and weak adhesive HLR categories are a suitable concept providing many of the required properties. The literature contains various versions of adhesive (Lack and Sobocinski 2004), quasiadhesive (Lack and Sobocinski 2005), weak adhesive HLR (Ehrig *et al.* 2006b), partial map adhesive (Heindel 2010) and $\mathcal{M}$-adhesive (Ehrig *et al.* 2010a) categories. In adhesive categories, the class $\mathcal{M}$ of morphisms is all monomorphisms, while in quasiadhesive the class of all regular monomorphisms is considered. With slightly different requirements concerning the existence of pushouts and pullbacks along $\mathcal{M}$-morphisms and requirements of $\mathcal{M}$-morphisms with respect to the van Kampen property, we get what are basically special weak adhesive HLR categories. In contrast, partial map adhesive categories are based on hereditary pushouts, which are pushouts that have to be preserved by the inclusion functor from the category $\mathcal{C}$ into the category of partial maps over $\mathcal{C}$. As shown in Ehrig *et al.* (2010a), partial map adhesive categories are also

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

$\mathcal{M}$-adhesive categories. Since all the main properties are valid in $\mathcal{M}$-adhesive categories, we have chosen to work with these in the current paper.

Definition 2.3 ($\mathcal{M}$-adhesive category). A category $\mathcal{C}$ with a morphism class $\mathcal{M}$ is an *$\mathcal{M}$-adhesive category* if the following properties hold:

(1) $\mathcal{M}$ is a class of monomorphisms containing all isomorphisms, closed under composition and decomposition, that is, for morphisms f and g, we have:

— f being an isomorphism implies $f \in \mathcal{M}$;

— $f, g \in \mathcal{M}$ implies $g \circ f \in \mathcal{M}$; and

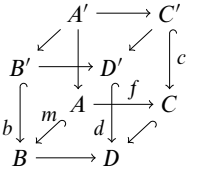
— $g \circ f \in \mathcal{M}$, $g \in \mathcal{M}$ implies $f \in \mathcal{M}$.

(2) $\mathcal{C}$ has pushouts and pullbacks along $\mathcal{M}$-morphisms, that is, pushouts and pullbacks where at least one of the given morphisms is in $\mathcal{M}$, and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks, that is, given a pushout (1)

$$\begin{array}{ccc} A & \longrightarrow & C \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle n} \\ B & \longrightarrow & D \end{array}$$

$m \in \mathcal{M}$ implies $n \in \mathcal{M}$, and given a pullback (1), $n \in \mathcal{M}$ implies $m \in \mathcal{M}$.

(3) Pushouts in $\mathcal{C}$ along $\mathcal{M}$-morphisms are vertical weak van Kampen squares, ($\mathcal{M}$-VK squares for short), that is, for any commutative cube in $\mathcal{C}$



if we have the pushout with $m \in \mathcal{M}$ in the bottom, $b, c, d \in \mathcal{M}$ and the back faces are pullbacks, then the top is a pushout if and only if the front faces are pullbacks.

Remark 2.4. In contrast to a vertical weak van Kampen square, a horizontal one requires that $f \in \mathcal{M}$ rather than $b, c, d \in \mathcal{M}$. Both properties combined represent the weak van Kampen property as used in weak adhesive HLR categories (Ehrig *et al.* 2006b). Adhesive categories (Lack and Sobocinski 2005), which are a special case of $\mathcal{M}$-adhesive categories, are special cases of the weak adhesive HLR categories in Ehrig *et al.* (2006b), where, in addition, the class $\mathcal{M}$ is the class of all monomorphisms.

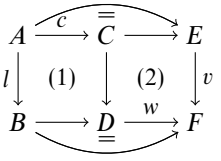
Fact 2.5 (Ehrig *et al.* 2006b). The category $\langle \mathrm{Graphs}, \mathcal{M} \rangle$ with the class $\mathcal{M}$ of all injective graph morphisms is $\mathcal{M}$-adhesive. Moreover, several variants of graphs like typed and typed attributed graphs with a corresponding class $\mathcal{M}$ of injective morphisms form $\mathcal{M}$-adhesive categories. The category $\langle \mathrm{PTNets}, \mathcal{M} \rangle$ of place/transition nets with the class $\mathcal{M}$ of all injective net morphisms and the category $\langle \mathrm{Spec}, \mathcal{M}_{strict} \rangle$ of algebraic specifications with the class $\mathcal{M}_{strict}$ of all strict injective specification morphisms are $\mathcal{M}$-adhesive, but not adhesive.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

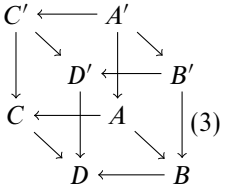
$\mathcal{M}$-adhesive categories have a number of nice properties, known as HLR-properties (Ehrig *et al.* 1991).

Lemma 2.6 (HLR-properties). For an $\mathcal{M}$-adhesive category $\langle \mathcal{C}, \mathcal{M} \rangle$, the following properties hold:

(1) Pushouts along $\mathcal{M}$-morphisms are pullbacks.
(2) We have $\mathcal{M}$ pushout–pullback decomposition. If (1) + (2)



is a pushout, (2) is a pullback, $w \in \mathcal{M}$ and ($l \in \mathcal{M}$ or $c \in \mathcal{M}$), then (1) and (2) are both pushouts and pullbacks.
(3) We have cube pushout-pullback decomposition: that is, given the commutative cube



where all morphisms in the top and the bottom are in $\mathcal{M}$, the top is a pullback and the front faces are pushouts, then the bottom is a pullback if and only if the back faces of the cube are pushouts.
(4) We have uniqueness of pushout complements: that is, given morphisms $A \hookrightarrow C$ in $\mathcal{M}$ and $C \to D$, then there is, up to isomorphism, at most one B with $A \to B$ and $B \to D$ such that diagram (1) above is a pushout.

Proof. The proofs can be found in Lack and Sobocinski (2005) and Ehrig *et al.* (2006b). Ehrig *et al.* (2006c) gives the proofs of the HLR-properties for weak adhesive HLR-categories, but they are also valid for $\mathcal{M}$-adhesive categories because horizontal weak VK-squares are not used anywhere in the proof. □

In order to prove the main results for $\mathcal{M}$-adhesive systems, we need to impose some additional HLR-requirements in the form of unique $\mathcal{E}'$-$\mathcal{M}$ pair factorisation, binary coproducts and initial pushouts over $\mathcal{M}$-morphisms. $\mathcal{E}'$-$\mathcal{M}$ pair factorisation is needed for the proof of all the main results, but they can also be obtained using classical $\mathcal{E}$-$\mathcal{M}$-factorisation and binary coproducts. The latter are also necessary and sufficient for defining parallel rules in the Parallelism Theorem. Initial pushouts are needed for the proof of the Amalgamation Theorem. However, we cannot exclude the possibility that weaker versions of some of these HLR-requirements may be sufficient to show our main results, or suitable variants of them.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

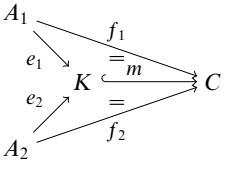
Definition 2.7. Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be an $\mathcal{M}$-adhesive category and $\mathcal{E}'$ be a class of morphism pairs with the same codomain. $\langle \mathcal{C}, \mathcal{M} \rangle$ has a *unique $\mathcal{E}'$-$\mathcal{M}$ pair factorisation* if, for each pair of morphisms

$$f_1 : A_1 \to C$$
$$f_2 : A_2 \to C,$$

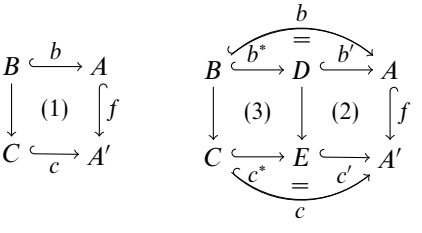
there exist a unique (up to isomorphism) object K and unique (up to isomorphism) morphisms

$$e_1 : A_1 \to K$$
$$e_2 : A_2 \to K$$
$$m : K \hookrightarrow C$$

with $(e_1, e_2) \in \mathcal{E}'$ and $m \in \mathcal{M}$ such that $m \circ e_1 = f_1$ and $m \circ e_2 = f_2$.



$\langle \mathcal{C}, \mathcal{M} \rangle$ has *initial pushouts over $\mathcal{M}$-morphisms* if, for every $\mathcal{M}$-morphism $f : A \hookrightarrow A'$, there exists an initial pushout over f. Consider the diagrams



An $\mathcal{M}$-morphism $b : B \hookrightarrow A$ is a *boundary* over f if there is a pushout complement of f and b such that (1) is an initial pushout over f. *Initiality* of (1) over f means that, for every pushout (2) with $b' \in \mathcal{M}$, there exist unique morphisms $b^*, c^* \in \mathcal{M}$ such that $b' \circ b^* = b$, $c' \circ c^* = c$ and (3) is a pushout. B is called the *boundary* object and C the *context* with respect to f.

Fact 2.8 (Ehrig *et al.* 2006b). The category $\langle \text{Graphs}, \mathcal{M} \rangle$ has a unique $\mathcal{E}'$-$\mathcal{M}$ pair factorisation (where $\mathcal{E}'$ is the class of pairs of jointly surjective graph morphisms), binary coproducts and initial pushouts over $\mathcal{M}$-morphisms. Moreover, all the examples in Fact 2.5 satisfy these requirements.

3. Rules with application conditions

In this section, we use the framework of $\mathcal{M}$-adhesive categories, introduce rules with application conditions for high-level structures such as graphs, Petri nets, (hyper)graphs

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

and algebraic specifications, and show how application conditions can be shifted over morphisms and rules.

Assumption 3.1. We assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is an $\mathcal{M}$-adhesive category with $\mathcal{E}'$-$\mathcal{M}$ pair factorisation (used in the first shift lemma – Lemma 3.11), binary coproducts (used in Definition 4.8) and initial pushouts over $\mathcal{M}$-morphisms (used in Theorem 5.3).

Remark 3.2. The category $\langle \text{Graphs}, \mathcal{M} \rangle$ satisfies Assumption 3.1. For simplicity, we may write: *graph* instead of *object*; *graph morphism* instead of *morphism*; and category $\langle \text{Graphs}, \mathcal{M} \rangle$ instead of $\mathcal{M}$-*adhesive* category $\langle \mathcal{C}, \mathcal{M} \rangle$:

object	—	graph
morphism	—	graph morphism
$\langle \mathcal{C}, \mathcal{M} \rangle$	—	$\langle \text{Graphs}, \mathcal{M} \rangle$

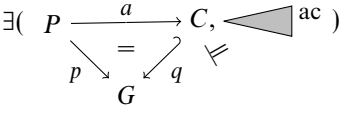
Application conditions, more concretely, nested application conditions, may be represented as a tree of morphisms equipped with logical symbols such as quantifiers and connectives.

Definition 3.3 (application conditions). An *application condition*, also called *nested application condition*, is defined inductively as follows:

(1) For every object P, true is an application condition over P.
(2) For every morphism $a: P \to C$ and every application condition ac over C, $\exists(a, \text{ac})$ is an application condition over P.
(3) For application conditions ac, ac_i over P with $i \in I$ (for all index sets I), $\neg\text{ac}$ and $\wedge_{i \in I} \text{ac}_i$ are application conditions over P.

Satisfiability of application conditions is defined inductively as follows:

(1) Every morphism satisfies true.
(2) A morphism $p: P \to G$ satisfies $\exists(a, \text{ac})$ over P with $a: P \to C$ if there exists a morphism $q: C \to G$ in $\mathcal{M}$ such that $q \circ a = p$ and q satisfies ac:



(3) A morphism $p: P \to G$ satisfies $\neg\text{ac}$ over P if p does not satisfy ac, and p satisfies $\wedge_{i \in I} \text{ac}_i$ over P if p satisfies each ac_i $(i \in I)$.

We write $p \models \text{ac}$ to denote the fact that the morphism p satisfies ac.

Two application conditions ac and ac′ over P are *equivalent*, denoted by $\text{ac} \equiv \text{ac}'$, if for all morphisms $p: P \to G$, $p \models \text{ac}$ if and only if $p \models \text{ac}'$.

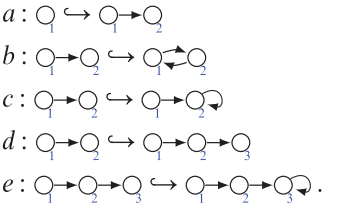
Notation 3.4. We write $\exists a$ as an abbreviation for $\exists(a, \text{true})$ and $\forall(a, \text{ac})$ as an abbreviation for $\neg\exists(a, \neg\text{ac})$.

Remark 3.5. The concept of application conditions was introduced in Ehrig and Habel (1986). Positive and negative application conditions, which were introduced in Habel

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

et al. (1996), correspond to nested application conditions of the form $\exists a$ and $\neg\exists a$, respectively. Negative application conditions are investigated intensively in, for example, Lambers (2010). Nested application conditions were introduced and intensively studied in Habel and Pennemann (2009) and Pennemann (2009), and are generalisations of the corresponding notions in Heckel and Wagner (1995), Koch *et al.* (2005) and Ehrig *et al.* (2006a).

Example 3.6. The following expressions are application conditions based on injective graph morphisms:



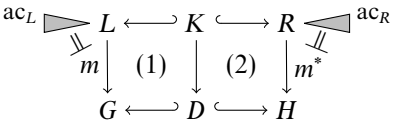
$\exists a$	There exists a proper outgoing edge from the image of 1.
$\neg\exists a$	There does not exist a proper outgoing edge from the image of 1.
$\exists(a, \neg\exists b)$	There exists a proper outgoing edge from the image of 1 without an edge in converse direction.
$\forall(a, \exists c)$	For every proper outgoing edge from the image of 1, the target has a loop.
$\exists(a, \forall(d, \exists e))$	There exists a proper edge outgoing from the image of 1 such that, for all edges outgoing from the target, the target has a loop.

The first application condition is positive, the second is negative and the rest are properly nested.

Rules are specified by a pair of $\mathcal{M}$-morphisms. In order to restrict the applicability of rules, they are equipped with a left and a right application condition. Such a rule is applicable with respect to a 'match' morphism from the left-hand side of the rule to an object if and only if the underlying plain rule is applicable, the match morphism satisfies the left application condition and the comatch morphism satisfies the right application condition.

Definition 3.7 (rules and transformations). A *plain rule* $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ consists of two $\mathcal{M}$-morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$. A *rule* $\varrho = \langle p, \mathrm{ac}_L, \mathrm{ac}_R \rangle$ consists of a plain rule p and two application conditions ac_L and ac_R over L and R, respectively. L and R are called the *left-* and *right-hand side* of p, respectively; ac_L and ac_R are called the *left* and *right application condition* of ϱ, respectively.

A *direct transformation* consists of two pushouts (1) and (2) such that $m \models \mathrm{ac}_L$ and $m^* \models \mathrm{ac}_R$:



https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

We write $G \Rightarrow_{\varrho,m,m^*} H$ and say that $m: L \to G$ is the *match* of ϱ in G and $m^*: R \to H$ is the *comatch* of ϱ in H. We also write $G \Rightarrow_{\varrho,m} H$, $G \Rightarrow_{\varrho} H$ or $G \Rightarrow H$ to express the fact that there is an m^*, m, m^*, or ϱ, m, m^*, respectively, such that $G \Rightarrow_{\varrho,m,m^*} H$. A transformation $G \overset{*}{\Rightarrow} H$ means $G \cong H$ or a sequence of direct transformations

$$G = G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n = H.$$

Fact 3.8. In $\langle \text{Graphs}, \mathcal{M} \rangle$, the application of a rule $\varrho = \langle p, \text{ac}_L, \text{ac}_R \rangle$ to a graph G amounts to the following steps:

(1) Find a match $m: L \to G$ satisfying ac_L and the *gluing condition*:
   — *Dangling condition:*
   No edge in $G - m(L)$ is incident to a node in $m(L) - m(K)$.
   — *Identification condition:*
   For all distinct items $x, y \in L$, we have $m(x) = m(y)$ only if $x, y \in K$. (This condition is understood to hold separately for nodes and edges.)

(2) Remove $m(L - K)$ from G, yielding a graph D, and add $R - K$, yielding a graph H.

(3) Check whether the comatch $m^*: R \to H$ satisfies ac_R.

Example 3.9. We will now introduce the rules for the mutual exclusion algorithm. The main aim is to ensure that at all times, at most one process is using each resource. Another variant of this algorithm implemented by graph transformation can be found in Ehrig *et al.* (2006b), where the lack of application conditions induces a much more complex model, including more types or labels together with additional rules for handling a single resource. Using application conditions, we can simplify the models and do not need additional edges representing the next executable step of the system, while also extending the context to an arbitrary number of resources.

Initially, each process is idle and for each resource the turn variable is connected to an arbitrary process, meaning that this process has the turn to use that resource. If a process P_1 wants to use some resource R, it becomes active and points the flag `F1` to R. If, in addition, it has the turn for R, it may proceed to use it, which is described by an `F2`-flag to the resource and a crit loop at the process. Otherwise, if the turn for R belongs to another process P', P must wait until P' is not flagging R. At this point, the process may get the turn for R and start using it. When P has finished using R, the flag and the crit are removed, and the process is again idle. As an extension of this normal behaviour, a resource may be disabled, denoted by eliminating its turn variable, if there is no flag present for it. Moreover, a resource may be enabled again if all other resources have at least two requests waiting.

The rules `setFlag`, `setTurn`, `enter` and `exit` in Figure 5 describe the standard behaviour of the system. With `setFlag`, a process becomes active and sets its `F1`-flag to a resource. Note that this rule has a positive application condition requiring that the resource has a turn variable noting it as enabled. If a process has set an `F1`-flag to a resource and the turn variable of this resource points to another process, which has no flag to this resource, the turn variable can be assigned to the first process using `setTurn`.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

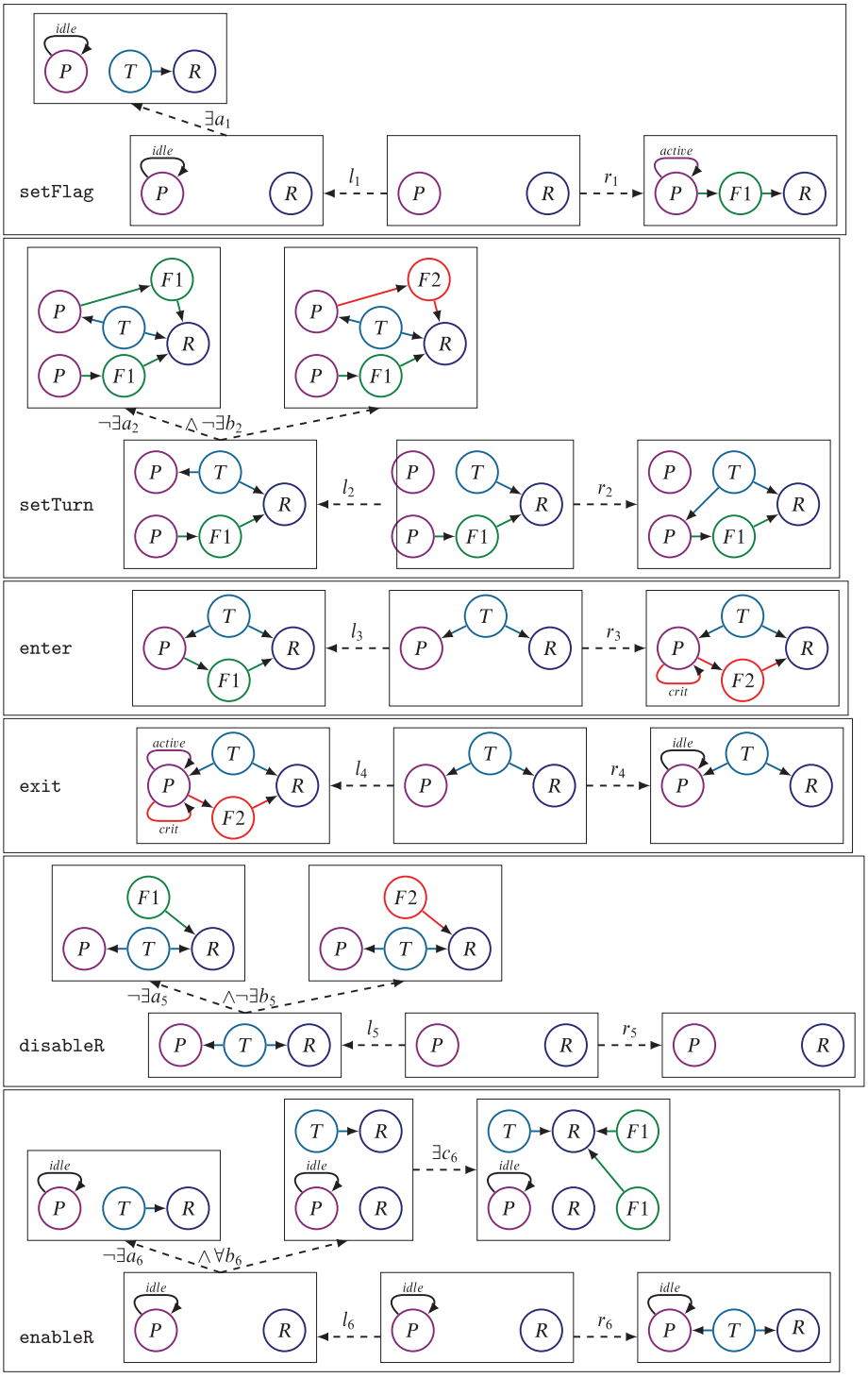


Fig. 5. (Colour online) The rules for the mutual exclusion algorithm

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Here, the application condition forbids the possibility that the process that has the turn of the resource is already flagging that resource. With the rule `enter`, if a process has the turn of a resource R and it points to R with an `F1`-flag, then the flag is replaced by an `F2`-flag and a loop `crit` is added to the process. When the process is finished, the rule `exit` is executed, which deletes the loop and the flag, and the process becomes idle again. Moreover, with the rules `disableR` and `enableR`, a resource can be disabled or enabled if the corresponding application conditions are fulfilled. The application condition true is not included in the figures, while the application conditions $\mathrm{Q}(a, \mathrm{ac})$ with $\mathrm{Q} \in \{\exists, \neg\exists, \forall\}$ are represented by the morphism $a: P \to C$, marked by Qa, combined with a drawing of ac. Conjunctions of application conditions are represented by $\wedge$ between 'the outgoing morphisms'.

Note that we could easily have a rule `setFlag` without any application condition. In particular, it is enough to include in the left-hand side of the rule the turn variable pointing to the resource R. However, the application condition $\forall(b_6, \exists c_6)$ shown on the right of the rule `enableR` cannot be removed, even if it too is a positive application condition. In particular, this condition is nested twice because we need to specify that every other enabled resource has two waiting processes.

Consider the rule `setTurn` with the match m_1 shown on the left-hand side of Figure 6. Note that m_1 matches the two processes of the rule `setTurn` to the upper two processes in G such that m_1 satisfies the gluing condition, as well as the application condition $\neg\exists a_2 \wedge \neg\exists b_2$ and leads to the direct transformation

$$G \Rightarrow_{\mathtt{setTurn},m_1} H_1$$

redirecting the turn variable from the idle process to one of the active ones, as shown in Figure 6. The graph H_1 is obtained from G by removing $m_1(L_1 - K_1)$ and adding $R_1 - K_1$. For the graph H_1, there is no direct transformation

$$H_1 \Rightarrow_{\mathtt{setTurn},m'} H_2$$

because any match $m': L_1 \to H_1$ does not satisfy the application condition $\neg\exists a_2$. Note that the rules are completely symmetric, which means that a rule can be reversed to give its inverse rule.

Fact 3.10 (inverse rule). For every rule

$$\varrho = \langle p, \mathrm{ac}_L, \mathrm{ac}_R \rangle$$

with

$$p = \langle L \hookleftarrow K \hookrightarrow R \rangle,$$

the rule

$$\varrho^{-1} = \langle p^{-1}, \mathrm{ac}_R, \mathrm{ac}_L \rangle$$

with

$$p^{-1} = \langle R \hookleftarrow K \hookrightarrow L \rangle$$

is the *inverse rule* of ϱ. For every direct transformation

$$G \Rightarrow_{\varrho,m,m^*} H,$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

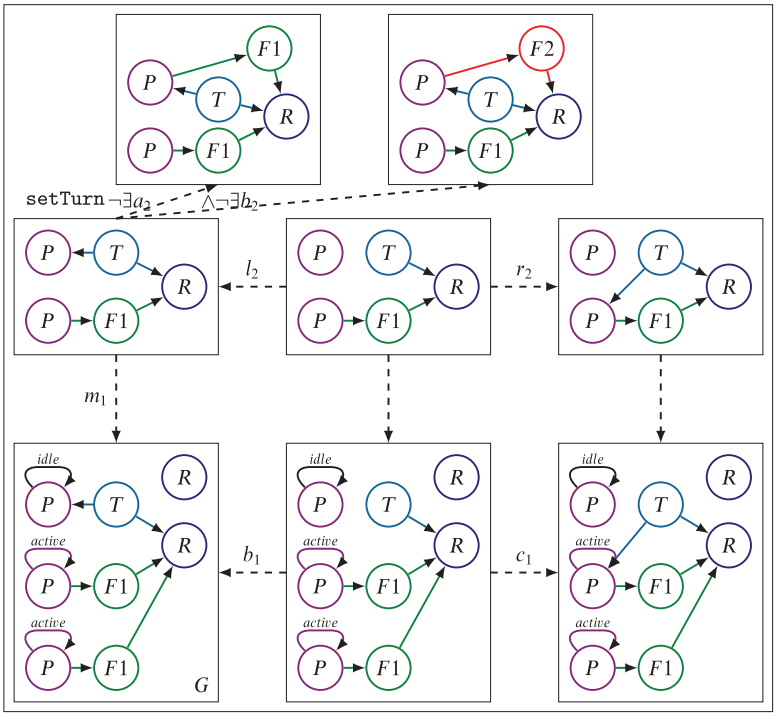


Fig. 6. (Colour online) Rule application

there is a direct transformation

$$H \Rightarrow_{\varrho^{-1},m^*,m} G$$

using the inverse rule.

We will now present two important technical results, which will be key to proving the main results of this paper. The first shows that application conditions can be shifted over morphisms. While the construction in Habel and Pennemann (2009) allows a shift over a monomorphism and uses pushouts along $\mathcal{M}$-morphisms, the construction in the current paper allows a shift over an arbitrary morphism and uses $\mathcal{E}'$-$\mathcal{M}$ pair factorisations.

Lemma 3.11 (shift of application conditions over morphisms). There is a Shift construction such that, for each application condition ac over P and for each morphism $b: P \to P'$, Shift transforms ac via b into an application condition $\mathrm{Shift}(b, \mathrm{ac})$ over P' such that, for each morphism $n: P' \to H$, we have $n \circ b \models \mathrm{ac} \iff n \models \mathrm{Shift}(b, \mathrm{ac})$.

$$\mathrm{ac} \triangleright P \xrightarrow{b} P' \triangleleft \mathrm{Shift}(b, \mathrm{ac}), \quad n \circ b \searrow H \swarrow n, \quad = $$

Construction 3.12. The Shift construction is defined inductively as follows:

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

— Case true:

$$\text{Shift}(b, \text{true}) = \text{true}.$$

— Case $\exists(a, \text{ac})$:
If

$$\mathcal{F} = \{(a', b') \in \mathcal{E}' \mid b' \in \mathcal{M} \text{ and (1) below commutes}\} \neq \varnothing$$

$$\begin{array}{ccc} P & \xrightarrow{b} & P' \\ {\scriptstyle a}\downarrow & (1) & \downarrow{\scriptstyle a'} \\ C & \xhookrightarrow{b'} & C' \\ \triangle & & \\ \text{ac} & & \end{array}$$

then

$$\text{Shift}(b, \exists(a, \text{ac})) = \bigvee_{(a', b') \in \mathcal{F}} \exists(a', \text{Shift}(b', \text{ac}))$$

otherwise

$$\text{Shift}(b, \exists(a, \text{ac})) = \text{false}.$$

— Case $\neg\text{ac}$:

$$\text{Shift}(b, \neg\text{ac}) = \neg\text{Shift}(b, \text{ac}).$$

— Case $\wedge_{i \in I} \text{ac}_i$:

$$\text{Shift}(b, \wedge_{i \in I} \text{ac}_i) = \wedge_{i \in I} \text{Shift}(b, \text{ac}_i).$$

Proof. The statement is proved by structural induction:

— *Base case*:
The equivalence holds trivially for the application condition true.

— *Inductive step*:
For an application condition of the form $\exists(a, \text{ac})$, we have to show

$$n \circ b \models \exists(a, \text{ac}) \iff n \models \text{Shift}(b, \exists(a, \text{ac})).$$

– *Only if direction*:
Let

$$n \circ b \models \exists(a, \text{ac}).$$

By the definition of satisfiability, there is some $q \in \mathcal{M}$ with $q \circ a = n \circ b$ and $q \models \text{ac}$.
By $\mathcal{E}'$-$\mathcal{M}$ pair factorisation, there exist an object C' and morphisms

$$a' : P' \to C'$$
$$b' : C \to C'$$
$$m : C' \hookrightarrow H$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

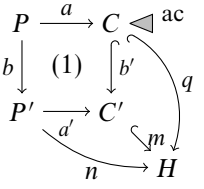
with $(a', b') \in \mathcal{E}'$ and $m \in \mathcal{M}$ such that $m \circ a' = n$ and $m \circ b' = q$. So

$$\begin{aligned} m \circ a' \circ b &= n \circ b \\ &= q \circ a \\ &= m \circ b' \circ a \end{aligned}$$

and, since $m \in \mathcal{M}$, we have

$$a' \circ b = b' \circ a,$$

that is, (1) in the following diagram commutes:



Since $\mathcal{M}$ is closed under decomposition, $q, m \in \mathcal{M}$ implies $b' \in \mathcal{M}$. Thus, $(a', b') \in \mathcal{F}$. By the induction hypothesis,

$$q = m \circ b' \models \text{ac} \Leftrightarrow m \models \text{Shift}(b', \text{ac}).$$

So

$$n = m \circ a' \models \exists(a', \text{Shift}(b', \text{ac}))$$

and, by the definition of Shift,

$$n \models \exists(b, \text{Shift}(a, \text{ac})).$$

– *If direction*:
Let

$$n \models \text{Shift}(b, \exists(a, \text{ac})).$$

Then there is some $(a', b') \in \mathcal{F}$ such that $b' \in \mathcal{M}$, $a' \circ b = b' \circ a$ and $n \models \exists(a', \text{Shift}(b', \text{ac}))$.
By the definition of satisfiability, there is some $m \in \mathcal{M}$ such that $m \circ a' = n$ and $m \models \text{Shift}(b', \text{ac})$.
By the induction hypothesis,

$$m \models \text{Shift}(b', \text{ac}) \Leftrightarrow m \circ b' \models \text{ac}.$$

So there is some $q = m \circ b' \in \mathcal{M}$ such that $q \models \text{ac}$, that is,

$$n \circ b = q \circ a \models \exists(a, \text{ac}),$$

which completes the inductive proof. □

Example 3.13. To illustrate the construction of shifting an application condition over morphisms, consider the application condition $\forall(b_6, \exists c_6)$ of the rule `enableR`, which is an

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

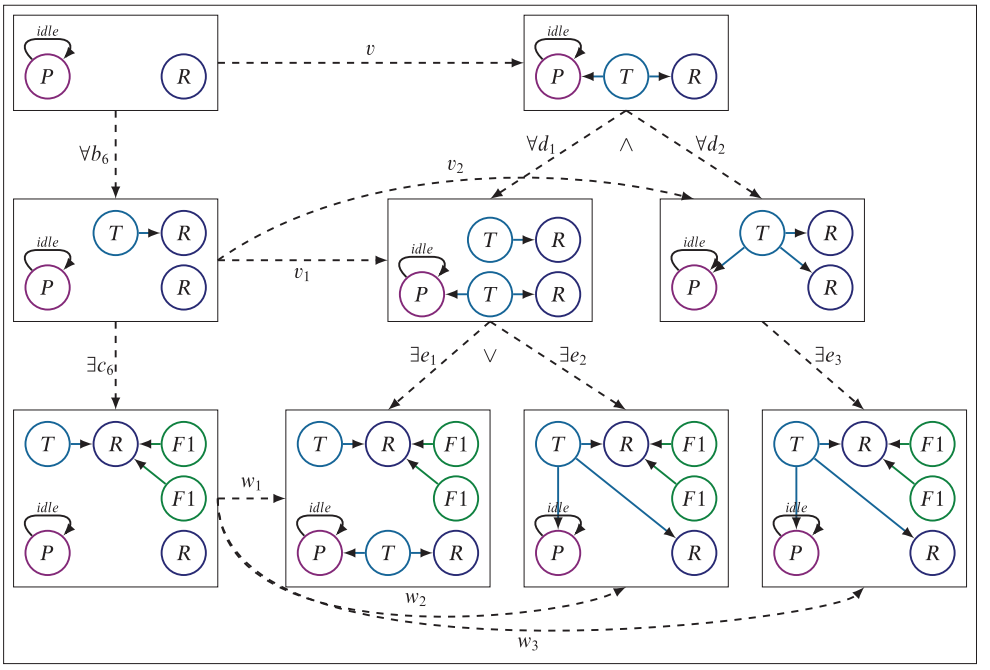


Fig. 7. (Colour online) Shift of the application condition $\forall(b_6, \exists c_6)$ over a morphism.

application condition over the left-hand side of this rule. We want to shift this condition over the morphism v shown at the top of Figure 7.

The first step of the construction is shown in the upper part of Figure 7 – the result is the intermediate application condition

$$\text{Shift}(v, \forall(b_6, \exists c_6)) = \forall(d, \text{Shift}(v_1, \exists c_6)) \wedge \forall(d_2, \text{Shift}(v_2, \exists c_6)).$$

Since v_i has to be injective and the resulting object has to be an overlapping of the codomains of v and b_6 such that the diagram commutes, and these are the only two solutions possible.

In the second step, the second part of the application condition has to be shifted over the two new morphisms v_1 and v_2. The result is shown in the lower part of Figure 7, that is, the application condition

$$\text{Shift}(v, \forall(b_6, \exists c_6)) = \forall(d_1, \exists e_1 \vee \exists e_2) \wedge \forall(d_2, \exists e_3).$$

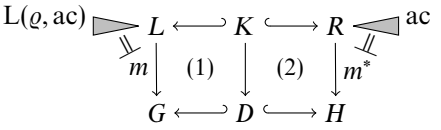
The other key result for proving the main results of the current paper is that application conditions can be shifted along rules.

Lemma 3.14 (shift of application conditions over rules (Habel and Pennemann 2009)). There is a construction L such that, for each rule ϱ and each application condition ac over R, we have L transforms ac through ϱ into $\text{L}(\varrho, \text{ac})$ over L such that for each direct transformation

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

$$G \Rightarrow_{\varrho,m,m^*} H,$$

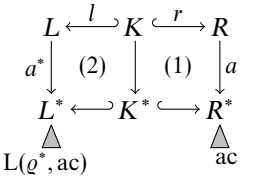
we have $m \models \mathrm{L}(\varrho, \mathrm{ac}) \iff m^* \models \mathrm{ac}$.



Construction 3.15. The construction L is defined inductively as follows:

— Case true:
$\mathrm{L}(\varrho, \mathrm{true}) = \mathrm{true}$.

— Case $\exists(a, \mathrm{ac})$:
Consider



If $\langle r, a \rangle$ has a pushout complement (1) and

$$\varrho^* = \langle L^* \hookleftarrow K^* \hookrightarrow R^* \rangle$$

is the derived rule by constructing the pushout (2), then

$$\mathrm{L}(\varrho, \exists(a, \mathrm{ac})) = \exists(a^*, \mathrm{L}(\varrho^*, \mathrm{ac}))$$

otherwise

$$\mathrm{L}(\varrho, \exists(a, \mathrm{ac})) = \mathrm{false}.$$

— Case $\neg \mathrm{ac}$:

$$\mathrm{L}(\varrho, \neg \mathrm{ac}) = \neg \mathrm{L}(\varrho, \mathrm{ac}).$$

— Case $\wedge_{i \in I} \mathrm{ac}_i$:

$$\mathrm{L}(\varrho, \wedge_{i \in I} \mathrm{ac}_i) = \wedge_{i \in I} \mathrm{L}(\varrho, \mathrm{ac}_i).$$

Remark 3.16. The construction L uses rules to transform right application conditions into left application conditions. The construction R with

$$\mathrm{R}(\varrho, \mathrm{ac}) = \mathrm{L}(\varrho^{-1}, \mathrm{ac})$$

transforms left application conditions ac using the rule ϱ into right application conditions.

Example 3.17. Suppose we want to translate the application condition $\forall(b_6, \exists c_6)$ of the rule `enableR` to the right-hand side. Basically, this means applying the rule to the first graph of the application condition, leading to a span, which is applied as a rule to the second graph. The result is shown in Figure 8, that is, the translated application condition is $\forall(b_6^*, \exists c_6^*)$.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

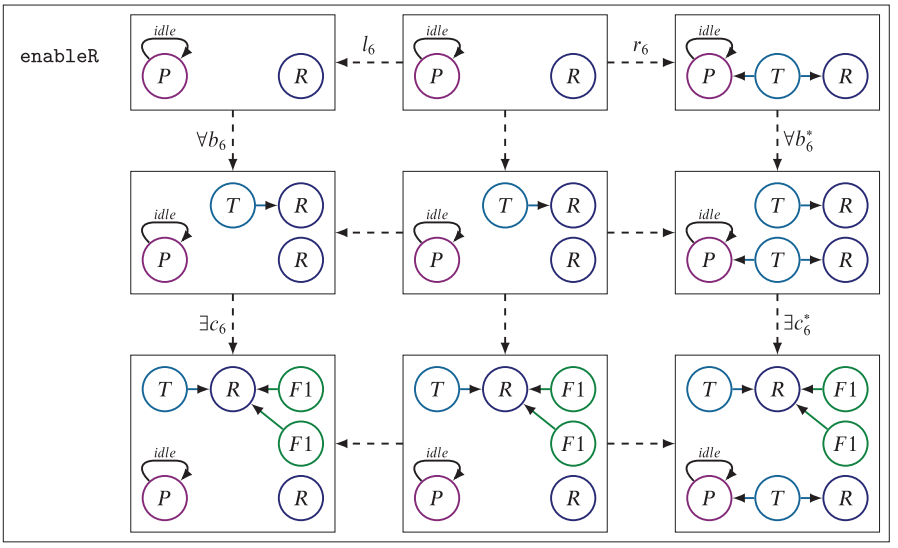


Fig. 8. (Colour online) Shift of the application condition from left to right.

As a consequence of the second shift lemma (Lemma 3.14), every rule can be transformed into an equivalent rule where the right application condition is always true. A rule of the form $\langle p, \mathrm{ac}_L, \mathrm{true}\rangle$ is said to be a rule *with left application condition* and is abbreviated by $\langle p, \mathrm{ac}_L\rangle$. This may be considered an improvement with respect to efficiency since in order to check a right application condition, we must first apply the rule, and then backtrack if the condition is not satisfied. However, left application conditions can be checked immediately after a match has been found.

Corollary 3.18 (rules with left application condition). There is a construction Left such that, for every rule ϱ, the rules ϱ and $\mathrm{Left}(\varrho)$ are equivalent, where $\mathrm{Left}(\varrho)$ is a rule with only left application condition.

Proof. For a rule $\varrho = \langle p, \mathrm{ac}_L, \mathrm{ac}_R\rangle$, let

$$\mathrm{Left}(\varrho) = \langle p, \mathrm{ac}_L \wedge \mathrm{L}(\varrho, \mathrm{ac}_R)\rangle.$$

Then, by Definition 3.7 and the second shift lemma (Lemma 3.14), ϱ and $\mathrm{Left}(\varrho)$ are equivalent:

$$\begin{aligned} G \Rightarrow_{\varrho,m,m^*} H &\Leftrightarrow G \Rightarrow_{p,m,m^*} H \wedge m \models \mathrm{ac}_L \text{ and } m^* \models \mathrm{ac}_R \\ &\Leftrightarrow G \Rightarrow_{p,m,m^*} H \wedge m \models \mathrm{ac}_L \text{ and } m \models \mathrm{L}(\varrho, \mathrm{ac}_R) \\ &\Leftrightarrow G \Rightarrow_{p,m,m^*} H \wedge m \models \mathrm{ac}_L \wedge \mathrm{L}(\varrho, \mathrm{ac}_R) \\ &\Leftrightarrow G \Rightarrow_{\mathrm{Left}(\varrho),m,m^*} H. \end{aligned}$$

□

4. The Local Church–Rosser, Parallelism and Concurrency Theorems

In this section, we present Local Church–Rosser, Parallelism and Concurrency Theorems for rules with application conditions as generalisations of the well-known theorems for

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

rules without application conditions (Ehrig *et al.* 2006b) and with negative application conditions (Lambers 2010). The proofs of the statements are based on the corresponding statements for rules without application conditions and the shift lemmas (Lemmas 3.11 and 3.14), which say that application conditions can be shifted over morphisms and rules.

The structure of the proofs is as follows. We first switch from transformations with application conditions to the corresponding transformations without application conditions, then use the results for transformations without application conditions, and then, finally, lift the results without application conditions to application conditions.

$$\begin{array}{ccc} \text{transformations with ACs} & \Longrightarrow & \text{result with ACs} \\ \downarrow & & \uparrow \\ \text{transformations without ACs} & \Longrightarrow & \text{result without ACs} \end{array}$$

Remark 4.1. For every direct transformation $G \Rightarrow_{\varrho,m} H$ using a rule $\varrho = \langle p, \mathrm{ac}_L, \mathrm{ac}_R \rangle$, there is a direct transformation $G \Rightarrow_{p,m} H$ using the underlying plain rule p, which we call the underlying direct transformation without application conditions.

By Corollary 3.18, we may assume that the rules are rules with left application condition.

Assumption 4.2. In the following, for $i = 1, 2$, we let

$$\varrho_i = \langle p_i, \mathrm{ac}_{L_i} \rangle$$

be a rule with left application condition and

$$p_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$$

be the underlying plain rule.

First, we consider direct transformations

$$H_1 \Leftarrow_{\varrho_1} G \Rightarrow_{\varrho_2} H_2$$

and look for conditions under which there are direct transformations

$$H_1 \Rightarrow_{\varrho_2} M \Leftarrow_{\varrho_1} H_2.$$

In particular, the first obvious condition is that the underlying plain transformations are parallel independent. However, this is not enough, we must also require that the matches of ϱ_2 and ϱ_1 in H_1 and H_2, respectively, satisfy the application conditions of the corresponding rule. Similarly, we consider transformations

$$G \Rightarrow_{\varrho_1} H_1 \Rightarrow_{\varrho_2} M$$

and look for conditions under which there are transformations

$$G \Rightarrow_{\varrho_2} H_2 \Rightarrow_{\varrho_1} M.$$

In this case, in addition to the sequential independence of the underlying plain rules, we require that the match of ϱ_2 in G satisfies its application condition and that the comatch of ϱ_1 to M satisfies the application condition $\mathrm{R}(\varrho_1, \mathrm{ac}_{L_1})$.

We can now formulate the notions of parallel and sequential independence and present the Local Church–Rosser Theorem.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

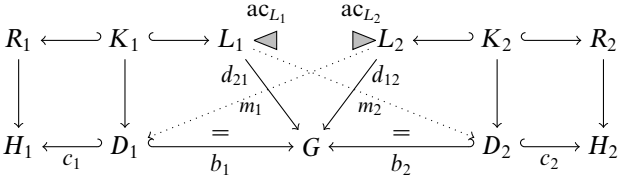
Definition 4.3 (parallel and sequential independence). A pair of direct transformations

$$H_1 \Leftarrow_{\varrho_1,m_1} G \Rightarrow_{\varrho_2,m_2} H_2$$

is *parallel independent* if there are morphisms $d_{ij} : L_i \to D_j$ such that $m_i = b_j \circ d_{ij}$ and

$$m_i' = c_j \circ d_{ij} \models \mathrm{ac}_{L_i}$$

with $i, j \in \{1, 2\}$ and $i \neq j$. Thus



A pair of direct transformations

$$G \Rightarrow_{\varrho_1,m_1} H_1 \Rightarrow_{\varrho_2,m_2} M$$

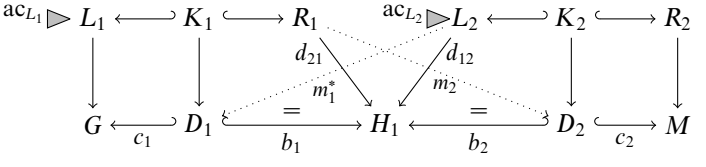
is *sequentially independent* if there are morphisms

$$d_{12} : R_1 \to D_2$$
$$d_{21} : L_2 \to D_1$$

such that $m_1^* = b_2 \circ d_{12}$, $m_2 = b_1 \circ d_{21}$ and

$$m_2' = c_1 \circ d_{21} \models \mathrm{ac}_{L_2}$$
$$m_1' = c_2 \circ d_{12} \models \mathrm{R}(\varrho_1, \mathrm{ac}_{L_1}).$$

Thus



A pair of direct transformations that is not parallel (sequentially) independent is said to be *parallel (sequentially) dependent*.

Example 4.4. The pair

$$H_1 \Leftarrow_{\texttt{setTurn},m_1} G \Rightarrow_{\texttt{enableR},m_2} H_2$$

of direct transformations in Figure 9 is parallel independent. The left rule application is the one we considered in Figure 6. Obviously, m_2 matches the idle process to the uppermost process in G. The morphisms d_{12} and d_{21} exist such that $b_1 \circ d_{21} = m_2$, $b_2 \circ d_{12} = m_1$ and

$$m_1' = c_2 \circ d_{12} \models \neg\exists a_2 \wedge \neg\exists b_2$$
$$m_2' = c_1 \circ d_{21} \models \neg\exists a_6 \wedge \forall(b_6, \exists c_6).$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

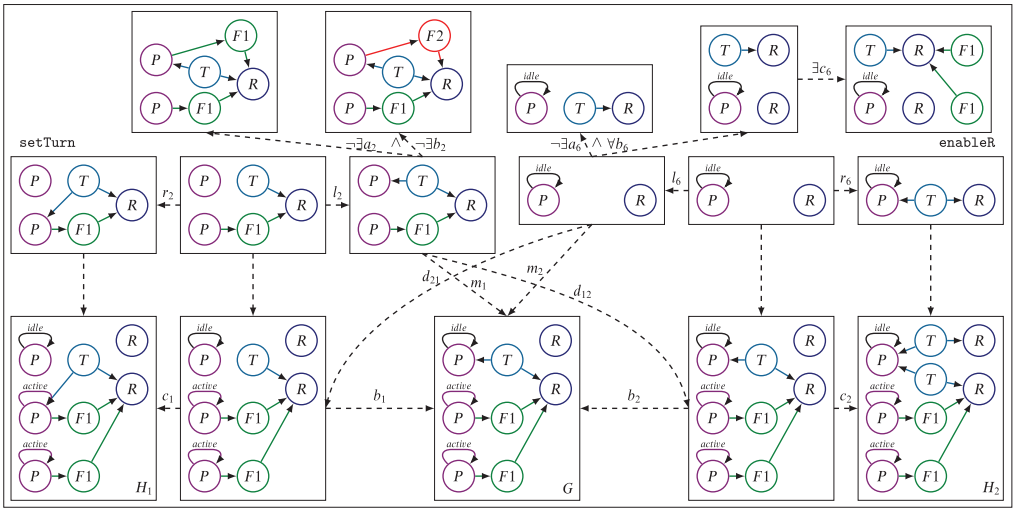


Fig. 9. (Colour online) Parallel independent transformations.

The sequence

$$H_1 \Rightarrow_{\texttt{setFlag},m_0} G \Rightarrow_{\texttt{enableR},m_2} H_2$$

of direct transformations in Figure 10 is sequentially dependent. Note that m_0 matches the process of the rule `setFlag` to the lowermost process in H_1, while the second transformation is the one already considered in Figure 9. The morphisms d_{12} and d_{21} exist such that $c_1 \circ d_{21} = m_2$, $c_2 \circ d_{12} = m_1^*$ and

$$b_2 \circ d_{12} \models \mathrm{R}(\texttt{setFlag}, \exists a_1),$$

but

$$b_1 \circ d_{21} \not\models \neg\exists a_6 \wedge \forall(b_6, \exists c_6).$$

The transformations are sequentially dependent because the rule `setFlag` adds a second flag, which is needed to fulfill the application condition $\forall(b_6, \exists c_6)$ of the rule `enableR`. Note that the transformations without application conditions would be sequentially independent.

By Definition 4.3, we immediately get the following fact.

Fact 4.5. Direct transformations are parallel (sequentially) independent if and only if the underlying direct transformations without application conditions are parallel (sequentially) independent and the 'induced' matches satisfy the corresponding application conditions.

By Definition 4.3, parallel and sequential independence are closely related.

Fact 4.6. Two direct transformations

$$H_1 \Leftarrow_{\varrho_1,m_1} G \Rightarrow_{\varrho_2,m_2} H_2$$

are parallel independent if and only if the two direct transformations

$$H_1 \Rightarrow_{\varrho_1^{-1},m_1^*} G \Rightarrow_{\varrho_2,m_2} H_2$$

are sequentially independent, where m_1^* is the comatch of ϱ_1 in H_1.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

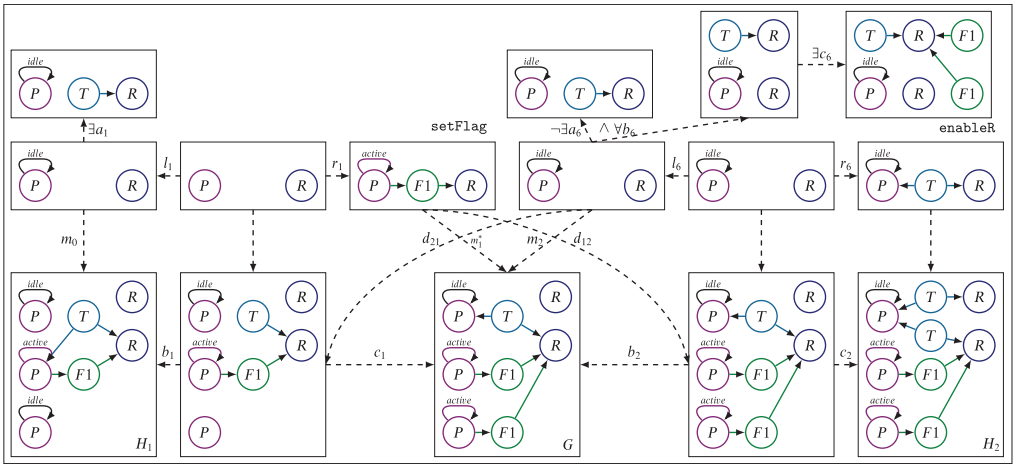


Fig. 10. (Colour online) Sequentially dependent transformations.

We can now present the Local Church–Rosser Theorem.

Theorem 4.7 (Local Church–Rosser Theorem). Given two parallel independent direct transformations

$$H_1 \Leftarrow_{\varrho_1,m_1} G \Rightarrow_{\varrho_2,m_2} H_2,$$

there is an object M and there are direct transformations

$$H_1 \Rightarrow_{\varrho_2,m_2'} M \Leftarrow_{\varrho_1,m_1'} H_2$$

such that the two transformations

$$G \Rightarrow_{\varrho_i,m_i} H_1 \Rightarrow_{\varrho_j,m_j'} M$$
$$G \Rightarrow_{\varrho_2,m_2} H_2 \Rightarrow_{\varrho_1,m_1'} M$$

are sequentially independent.

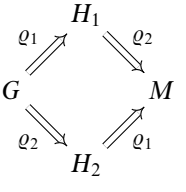
Given two sequentially independent direct transformations

$$G \Rightarrow_{\varrho_1,m_1} H_1 \Rightarrow_{\varrho_2,m_2} M,$$

there is an object H_2 and a transformation

$$G \Rightarrow_{\varrho_2,m_2'} H_2 \Rightarrow_{\varrho_1,m_1'} M$$

such that $H_1 \Leftarrow_{\varrho_1,m_1} G \Rightarrow_{\varrho_2,m_2'} H_2$ are parallel independent.



https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Proof. Let

$$H_1 \Leftarrow_{\varrho_1,m_1} G \Rightarrow_{\varrho_2,m_2} H_2$$

be parallel independent. Then the underlying direct transformations

$$H_1 \Leftarrow_{p_1,m_1} G \Rightarrow_{p_2,m_2} H_2$$

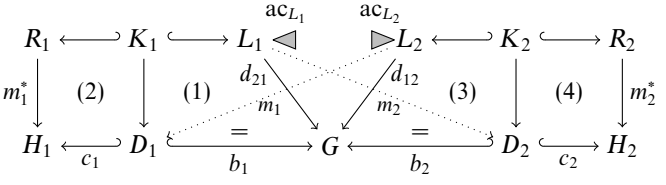
without application conditions are parallel independent. By the Local Church–Rosser Theorem without application conditions (Ehrig *et al.* 2006b), there is an object M and direct transformations

$$H_1 \Rightarrow_{p_2,m_2'} M \Leftarrow_{p_1,m_1'} H_2$$

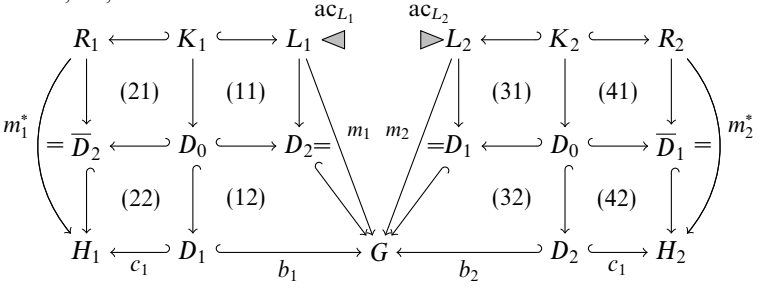
such that the transformations

$$G \Rightarrow_{p_1,m_1} H_1 \Rightarrow_{p_2,m_2'} M$$
$$G \Rightarrow_{p_2,m_2} H_2 \Rightarrow_{p_1,m_1'} M$$

are sequentially independent. By parallel independence, there are morphisms $d_{ij} : L_i \to D_j$ such that $m_i = b_j \circ d_{ij}$ with $(i, j \in \{1,2\}$ and $i \neq j)$.



The morphisms are used for the decomposition of the pushouts (i) into pushouts (i1) and (i2) for $i = 1, \ldots, 4$:



The pushouts can be rearranged as in the figures below. Since the composition of pushouts yields pushouts, we obtain direct transformations

$$H_1 \Rightarrow_{p_2,m_2'} M \Leftarrow_{p_1,m_1'} H_2$$

such that, for $i \in \{1,2\}$ and $i \neq j$, we have

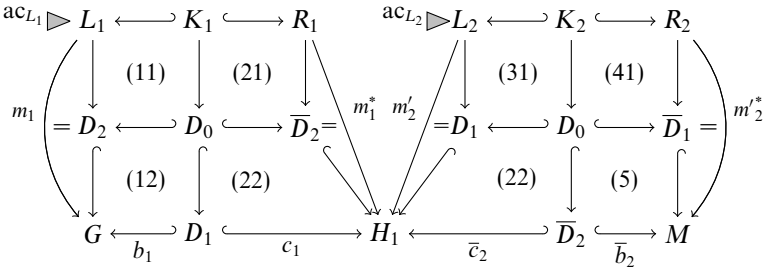
$$G \Rightarrow_{p_i,m_i} H_i \Rightarrow_{p_j,m_j'} M$$

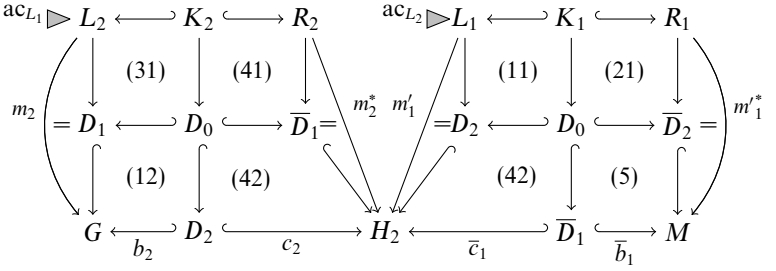
are sequentially independent: there are morphisms

$$\overline{d}_{ij} : R_i \to \overline{D}_j$$
$$\overline{d}_{ji} : L_j \to \overline{D}_i$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

such that $\overline{c}_j \circ \overline{d}_{ij} = m_i^*$ and $c_i \circ \overline{d}_{ji} = m'_j$.





By assumption, $m_i, m'_i \models \text{ac}_{L_i}$. By the second shift lemma (Lemma 3.14),

$$m_i \models \text{ac}_{L_i} \Leftrightarrow m_i^* \models \text{R}(\varrho_i, \text{ac}_i).$$

Thus, there is a transformation

$$G \Rightarrow_{\varrho_i, m_i} H_i \Rightarrow_{\varrho_j, m'_j} M$$

that is sequentially independent.

The second statement can be proved using the first statement and Fact 4.6. □

We will now consider parallel rules and parallel transformations. The parallel rule $\varrho_1 + \varrho_2$ of the rules ϱ_1 and ϱ_2 can be defined with help of the binary coproducts of the components of the rules, because, by the General Assumption (Assumption 3.1), $\langle \mathcal{C}, \mathcal{M} \rangle$ has binary coproducts.

Definition 4.8 (parallel rule and transformation). The *parallel rule* of ϱ_1 and ϱ_2 is the rule

$$\varrho_1 + \varrho_2 = \langle p, \text{ac}_L \rangle$$

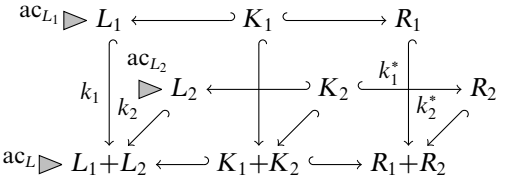
where

$$p = \langle L_1 + L_2 \hookleftarrow K_1 + K_2 \hookrightarrow R_1 + R_2 \rangle$$

is the parallel rule of p_1 and p_2 and

$$\text{ac}_L = \wedge_{i=1}^2 \text{Shift}(k_i, \text{ac}_{L_i}) \wedge \text{L}(p_1 + p_2, \text{Shift}(k_i^*, \text{R}(\varrho_i, \text{ac}_{L_i}))).$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press



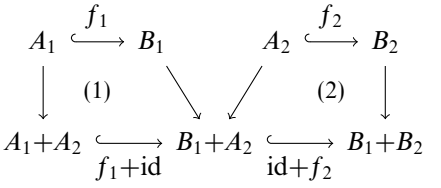
A direct transformation using a parallel rule is called a *parallel* direct transformation, or a parallel transformation for short.

Fact 4.9. The morphisms $K_1+K_2 \hookrightarrow L_1+L_2$ and $K_1+K_2 \hookrightarrow R_1+R_2$ are in $\mathcal{M}$.

Proof. Binary coproducts are compatible with $\mathcal{M}$, that is, $f_1, f_2 \in \mathcal{M}$ implies $f_1+f_2 \in \mathcal{M}$. In fact, pushout (1) in the diagram below with $f_1 \in \mathcal{M}$ implies $(f_1+\mathrm{id}) \in \mathcal{M}$ and pushout (2) with $f_2 \in \mathcal{M}$ implies $(\mathrm{id}+f_2) \in \mathcal{M}$, but now

$$(f_1+f_2) = (\mathrm{id}+f_2) \circ (f_1+\mathrm{id}) \in \mathcal{M}$$

by closure under composition.



This completes the proof. □

Example 4.10. The parallel rule `setTurn+enableR` is shown in the upper row of Figure 11[†]. The application

$$G \Rightarrow_{\texttt{setTurn+enableR},m_1+m_2} H'$$

of this parallel rule is shown in Figure 11 – it combines the effects of both rules to G leading to the graph H', where both the turn points to an active process and the previously disabled resource is now activated.

Two rules ϱ and ϱ' are isomorphic, denoted by $\varrho \cong \varrho'$, if there are isomorphisms $\mathrm{iso}_L, \mathrm{iso}_K, \mathrm{iso}_R$ between the components such that the resulting diagrams commute and the application conditions are isomorphic with respect to iso_L. As an immediate consequence of the definition, we have the following fact.

Fact 4.11. For all rules ϱ_1 and ϱ_2, we have

$$\varrho_1 + \varrho_2 \cong \varrho_2 + \varrho_1.$$

[†] The figure does not show the application conditions because there are so many of them. Basically, they say for various overlappings of processes or resources that there is no `F1`- or `F2`-flag between the process and the resource from the rule `setTurn`, no turn on the resource of `enableR`, and all active resources have at least two `F1`-flags pointing to them.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

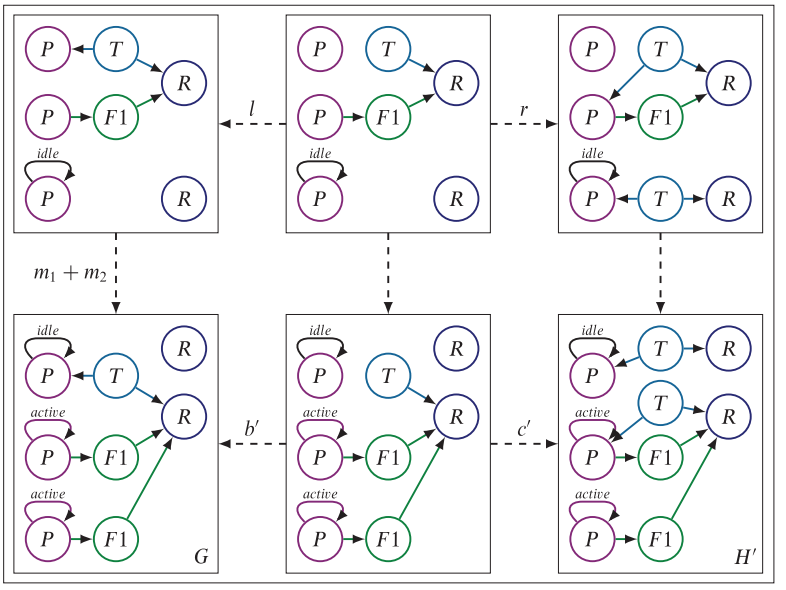


Fig. 11. (Colour online) Parallel rule and transformation.

The connection between sequentially independent direct transformations and parallel direct transformations using the parallel rule (Definition 4.8) is expressed by the Parallelism Theorem.

Theorem 4.12 (Parallelism Theorem). Given two sequentially independent direct transformations

$$G \Rightarrow_{\varrho_1,m_1} H_1 \Rightarrow_{\varrho_2,m_2'} M,$$

there is a parallel transformation

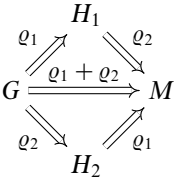
$$G \Rightarrow_{\varrho_1+\varrho_2,m} M.$$

Given a parallel transformation

$$G \Rightarrow_{\varrho_1+\varrho_2,m} M,$$

there are sequentially independent direct transformations

$$G \Rightarrow_{\varrho_1,m_1} H_1 \Rightarrow_{\varrho_2,m_2'} M$$
$$G \Rightarrow_{\varrho_2,m_2} H_i \Rightarrow_{\varrho_1,m_1'} M.$$



https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Proof. Let

$$G \Rightarrow_{\varrho_1,m_1} H_1 \Rightarrow_{\varrho_2,m_2'} M$$

be sequentially independent. Then the underlying transformation without application conditions is sequentially independent and, by the Parallelism Theorem without application conditions (Ehrig *et al.* 2006b), there is a parallel transformation

$$G \Rightarrow_{p_1+p_2,m} M$$

with $m_1 = m \circ k_1$ and $m'^*_2 = m^* \circ k^*_2$.

By assumption,

$$m_1 \models \mathrm{ac}_{L_1}$$
$$m_2' \models \mathrm{ac}_{L_2}.$$

By the shift lemmas (Lemmas 3.11 and 3.14) and Definition 4.8,

$$\begin{aligned} m_1 \models \mathrm{ac}_{L_1} \wedge m_2' \models \mathrm{ac}_{L_2} \qquad (*)\\ \Leftrightarrow m \models \mathrm{Shift}(k_1, \mathrm{ac}_{L_1}) \wedge m'^*_2 \models \mathrm{R}(\varrho_2, \mathrm{ac}_{L_2})\\ \Leftrightarrow m \models \mathrm{Shift}(k_1, \mathrm{ac}_{L_1}) \wedge m^* \models \mathrm{Shift}(k^*_2, \mathrm{R}(\varrho_2, \mathrm{ac}_{L_2}))\\ \Leftrightarrow m \models \mathrm{Shift}(k_1, \mathrm{ac}_{L_1}) \wedge \mathrm{L}(p_1{+}p_2, \mathrm{Shift}(k^*_2, \mathrm{R}(\varrho_2, \mathrm{ac}_{L_2}))) = \mathrm{ac}_L. \end{aligned}$$

Thus, $m \models \mathrm{ac}_L$, that is, the parallel transformation satisfies the application condition.

For the opposite direction, let

$$G \Rightarrow_{\varrho_1+\varrho_2,m} M$$

be a parallel transformation. Then there is an underlying parallel transformation without application conditions, and, by the Parallelism Theorem without application conditions (Ehrig *et al.* 2006b), there is a sequentially independent direct transformation

$$G \Rightarrow_{p_1,m_1} H_1 \Rightarrow_{p_2,m_2'} M$$

with $m_1 = m \circ k_1$ and $m'^*_2 = m^* \circ k^*_2$.

By assumption,

$$m \models \mathrm{ac}_L,$$

and by $(*)$,

$$m_1 \models \mathrm{ac}_{L_1}$$
$$m_2' \models \mathrm{ac}_{L_2},$$

that is, the sequentially independent direct transformations satisfy the application conditions. By

$$\varrho_1 + \varrho_2 \cong \varrho_2 + \varrho_1,$$

there is also a sequentially independent direct transformation

$$G \Rightarrow_{p_2,m_2} H_2 \Rightarrow_{p_1,m_1'} M$$

with $m_2 \models \mathrm{ac}_{L_2}$ and $m_1' \models \mathrm{ac}_{L_1}$. □

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Finally, we consider transformations of the form

$$G \Rightarrow_{\varrho_1} H \Rightarrow_{\varrho_2} M,$$

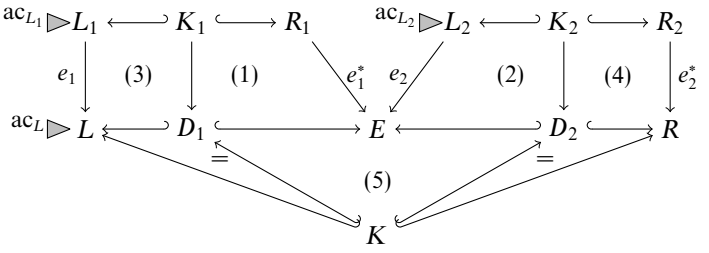
but without the assumption of sequential independence. This leads to the notions of an E-dependency relation, an E-concurrent rule for ϱ_1 and ϱ_2, E-concurrent transformations and E-related transformations. The connection between E-related and E-concurrent transformations is established in the Concurrency Theorem.

The construction of an E-concurrent rule is based on an E-dependency relation, which guarantees the existence of some pushout complements. It is defined with the help of pushouts and pullbacks along $\mathcal{M}$-morphisms. The application condition of the E-concurrent rule guarantees that whenever the E-concurrent rule is applicable, the rule ϱ_1 and, afterwards, the rule ϱ_2 is applicable.

Definition 4.13 (E-concurrent rule). Let $\mathcal{E}'$ be a class of morphism pairs with the same codomain. Given two rules ϱ_1 and ϱ_2, an object E with morphisms

$$e_1^* : R_1 \to E$$
$$e_2 : L_2 \to E$$

is an *E-dependency relation* for ϱ_1 and ϱ_2 if $(e_1^*, e_2) \in \mathcal{E}'$ and the pushout complements (1) and (2) over $K_1 \hookrightarrow R_1 \to E$ and $K_2 \hookrightarrow L_2 \to E$ in the diagram



exist. Given such an E-dependency relation for ϱ_1 and ϱ_2, the *E-concurrent rule* of ϱ_1 and ϱ_2 is the rule

$$\varrho_1 *_E \varrho_2 = \langle p, \mathrm{ac}_L \rangle$$

where

$$p = \langle L \hookleftarrow K \hookrightarrow R \rangle$$

with pushouts (3) and (4) and pullback (5),

$$\varrho_1^* = \langle L \hookleftarrow D_1 \hookrightarrow E \rangle$$

is the rule derived by ϱ_1 and k_1, and

$$\mathrm{ac}_L = \mathrm{Shift}(e_1, \mathrm{ac}_{L_1}) \wedge \mathrm{L}(\varrho_1^*, \mathrm{Shift}(e_2, \mathrm{ac}_{L_2})).$$

Example 4.14. Figure 12 shows the E-concurrent rule construction leading to the E-related sequence

$$G' \Rightarrow_{\mathtt{setFlag},m_0} G \Rightarrow_{\mathtt{enableR},m_2} H_2$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

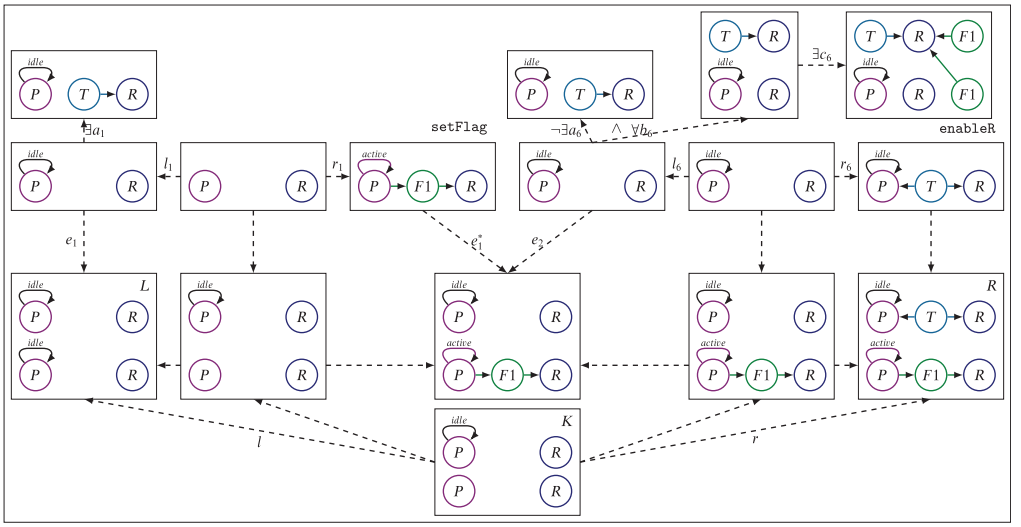


Fig. 12. (Colour online) E-concurrent rule construction.

of direct transformations already considered in Figure 10. Note that e_1 matches the process of `setFlag` to the lowermost process and e_2 matches the process of `enableR` to the uppermost process. Note also that

$$\mathrm{ac}_L = \mathrm{Shift}(e_1, \mathrm{ac}_1) \wedge \mathrm{L}(\varrho_6^*, \mathrm{Shift}(e_2, \mathrm{ac}_6))$$

is not shown explicitly because it becomes too large. The rule says that the lowermost resource should be connected to a token ($\mathrm{Shift}(e_1, \exists a_1)$), that the uppermost resource should not already be connected to a token ($\mathrm{L}(\varrho_6^*, \mathrm{Shift}(e_2, \neg\exists a_6))$), that the lowermost resource should already be connected to a `F1`-flag and that other enabled resources should already be connected to at least two `F1`-flags ($\mathrm{L}(\varrho_6^*, \mathrm{Shift}(e_2, \forall(b_6, \exists c_6)))$).

For rules without application conditions, the parallel rule is a special case of the E-concurrent rule with $E = R_1 + L_2$ (Ehrig *et al.* 2006b), but, in general, this is not the case for rules with application conditions: while the application conditions for the parallel rule must guarantee the applicability of the rules in each order, the application condition for the E-concurrent rule must guarantee the applicability of the rules in the given order. Nevertheless, the parallel rule of two rules can be constructed from two concurrent rules of the rules, one for each order: for $i, j \in \{1, 2\}$ with $i \neq j$, let $\mathrm{ac}_{L_{ij}}$ be the application condition of the E_{ij}-concurrent rule of ϱ_i and ϱ_j with $E_{ij} = R_i + L_j$. The rule $p_1 + p_2$ with application condition $\mathrm{ac}_{L_{12}} \wedge \mathrm{ac}_{L_{21}}$ is then called the *symmetric* concurrent rule of ϱ_1 and ϱ_2 and is denoted by $\varrho_1 * \varrho_2$.

Lemma 4.15 (parallel and symmetric concurrent rules). For rules ϱ_1 and ϱ_2, the parallel rule and the symmetric concurrent rule are equivalent:

$$\varrho_1 + \varrho_2 \equiv \varrho_1 * \varrho_2.$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Proof. For plain rules p_1 and p_2, the parallel rule p_1+p_2 and the concurrent rules $p_i *_{R_i+L_j} p_j$ are equivalent (Ehrig *et al.* 2006b). By Definitions 4.8 and 4.13,

$$\begin{aligned} m \models \mathrm{ac}_L &\Leftrightarrow m \models \wedge_{i=1}^{2} \mathrm{Shift}(k_i, \mathrm{ac}_{L_i}) \wedge \mathrm{L}(\varrho_i^*, \mathrm{Shift}(k'_j, \mathrm{ac}_{L_j})) \\ &\Leftrightarrow m \models \mathrm{ac}_{L_{12}} \wedge \mathrm{ac}_{L_{21}}, \end{aligned}$$

that is, the parallel rule and the symmetric concurrent rule are equivalent. □

We will now consider E-concurrent transformations via E-concurrent rules and E-related transformations via pairs of rules.

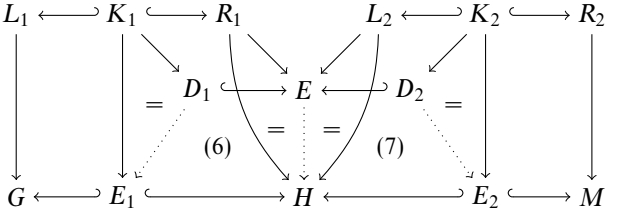
Definition 4.16 (E-concurrent and E-related transformations). A direct transformation via an E-concurrent rule is called an *E-concurrent* direct transformation, or an E-concurrent transformation for short. A transformation

$$G \Rightarrow_{\varrho_1} H \Rightarrow_{\varrho_2} M$$

is *E-related* if there are morphisms

$$\begin{aligned} E &\to H \\ D_1 &\to E_1 \\ D_2 &\to E_2 \end{aligned}$$

such that the triangles in the following diagram commute and (6) and (7) are pushouts:



We will now present a Concurrency Theorem for rules with application conditions.

Theorem 4.17 (Concurrency Theorem). Let E be a dependency relation for ϱ_1 and ϱ_2. For every E-related transformation

$$G \Rightarrow_{\varrho_1, m_1} H \Rightarrow_{\varrho_2, m_2} M,$$

there is an E-concurrent transformation

$$G \Rightarrow_{\varrho_1 *_E \varrho_2, m} M.$$

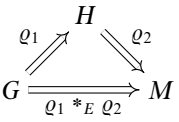
Conversely, for every E-concurrent transformation

$$G \Rightarrow_{\varrho_1 *_E \varrho_2, m} M,$$

there is an E-related transformation

$$G \Rightarrow_{\varrho_1, m_1} H \Rightarrow_{\varrho_2, m_2} M.$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press



Proof. Let

$$G \Rightarrow_{o_1,m_1} H \Rightarrow_{o_2,m_2} M$$

be E-related. Then the underlying transformation without application conditions is E-related and, by the Concurrency Theorem without application conditions (Ehrig *et al.* 2006b), there is an E-concurrent transformation

$$G \Rightarrow_{p_1 * p_2, m} M.$$

By assumption,

$$m_1 \models \mathrm{ac}_{L_1}$$
$$m_2 \models \mathrm{ac}_{L_2}.$$

By the shift lemmas (lemmas 3.11 and 3.14) and Definition 4.13, we have

$$\begin{aligned} m_1 \models \mathrm{ac}_{L_1} \text{ and } m_2 \models \mathrm{ac}_{L_2} \qquad & (*) \\ \Leftrightarrow\ & m \models \mathrm{Shift}(k_1, \mathrm{ac}_{L_1}) \text{ and } m' \models \mathrm{Shift}(k_2, \mathrm{ac}_{L_2}) \\ \Leftrightarrow\ & m \models \mathrm{Shift}(k_1, \mathrm{ac}_{L_1}) \text{ and } m \models \mathrm{L}(p_1^*, \mathrm{Shift}(k_2, \mathrm{ac}_{L_2})) \\ \Leftrightarrow\ & m \models \mathrm{Shift}(k_1, \mathrm{ac}_{L_1}) \wedge \mathrm{L}(p_1^*, \mathrm{Shift}(k_2, \mathrm{ac}_{L_2})) = \mathrm{ac}_L. \end{aligned}$$

Thus, $m \models \mathrm{ac}_L$, that is, the E-concurrent transformation satisfies the application condition.

Let $G \Rightarrow_{o,m} M$ be an E-concurrent transformation. So the underlying direct transformation without application conditions is E-concurrent and, by the Concurrency Theorem without application conditions (Ehrig *et al.* 2006b), there is an E-related transformation

$$G \Rightarrow_{p_1,m_1} H \Rightarrow_{p_2,m_2} M.$$

By assumption, $m \models \mathrm{ac}_L$. By statement (*),

$$m_1 \models \mathrm{ac}_{L_1}$$
$$m_2 \models \mathrm{ac}_{L_2},$$

that is, the E-related transformation satisfies the application conditions. □

In order to apply the Concurrency Theorem to a transformation, we need to construct an E-related transformation corresponding to Ehrig *et al.* (2006b, Fact 5.29). To do this, we use an $\mathcal{M}$-adhesive category with $\mathcal{E}'$-$\mathcal{M}$ pair factorisation.

Fact 4.18 (construction of E-related transformations). For every transformation

$$G \Rightarrow_{o_1,m_1} H \Rightarrow_{o_2,m_2} M$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

there is an E-dependency relation E such that

$$G \Rightarrow_{\varrho_1,m_1} H \Rightarrow_{\varrho_2,m_2} M$$

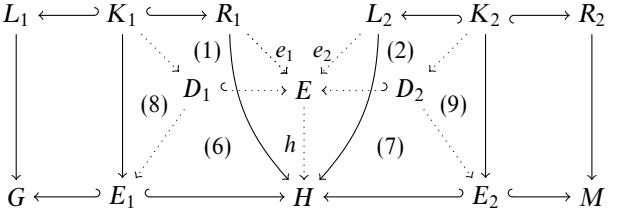
is E-related.

Proof. Given a transformation

$$G \Rightarrow_{\varrho_1,m_1,m_1^*} H \Rightarrow_{\varrho_2,m_2} M,$$

let $(e_1, e_2) \in \mathcal{E}'$, $h \in \mathcal{M}$ be an $\mathcal{E}'$-$\mathcal{M}$ pair factorisation of m_1^* and m_2 with $h \circ e_1 = m_1^*$ and $h \circ e_2 = m_2$.

We now construct (6) in the diagram below as a pullback of $E_1 \hookrightarrow H \hookleftarrow E$. By the universal pullpack property, there is a morphism $K_1 \to D_1$ such that (1) and (8) commute. Since $h \in \mathcal{M}$, (6) is a pullback, and (1)+(6) is a pushout, the $\mathcal{M}$-pushout–pullback decomposition property then implies that diagram (1) is a pushout, and, analogously, (2) is also a pushout.



Thus, E with $(e_1, e_2) \in \mathcal{E}'$ is an E-dependency relation and

$$G \Rightarrow_{\varrho_1,m_1} H \Rightarrow_{\varrho_2,m_2} M$$

is E-related. □

5. Amalgamation

In this section, we present an Amalgamation Theorem for rules with application conditions generalising the well-known theorem for rules without application conditions (Boehm *et al.* 1987; Corradini *et al.* 1997). The Amalgamation Theorem handles two direct transformations, which may be parallel dependent. Roughly speaking, for a ϱ-amalgamable pair of direct transformations $H_1 \Leftarrow_{\varrho_1} G \Rightarrow_{\varrho_2} H_2$, there is a direct transformation $G \Rightarrow M$ via the ϱ-amalgamated rule ϱ', and *vice versa*. The effect of the ϱ-amalgamated rule ϱ' may be described by the application of ϱ_i and the remainder of ϱ' with respect to ϱ_i $(i = 1, 2)$. The Multi-Amalgamation Theorem in Golas *et al.* (2014) and Golas (2011) generalises the Amalgamation Theorem to the case of $n \geqslant 2$ amalgamable direct transformations.

The amalgamation of rules is based on the notions of a subrule and its remainder. In the following, we let $\varrho = \langle p, \mathrm{ac}_L \rangle$ be a rule with

$$p = \langle L \hookleftarrow K \hookrightarrow R \rangle.$$

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Definition 5.1 (subrule and remainder). A rule ϱ is a *subrule* of a rule ϱ_1 if there are embedding $\mathcal{M}$-morphisms

$$L \hookrightarrow L_1$$
$$K \hookrightarrow K_1$$
$$R \hookrightarrow R_1$$

such that diagrams (1) and (2) in

$$\begin{array}{ccccc} \mathrm{ac}_L \triangleright\, L & \hookleftarrow & K & \hookrightarrow & R \\ {\scriptstyle k_1}\downarrow & (1) & \downarrow & (2) & \downarrow \\ \mathrm{ac}_{L_1} \triangleright\, L_1 & \hookleftarrow & K_1 & \hookrightarrow & R_1 \end{array}$$

are pullbacks, the pushout complement (1′) of $K \hookrightarrow L \hookrightarrow L_1$ in

$$\begin{array}{ccccc} & & L & \hookleftarrow & K & \hookrightarrow & R \\ & & \downarrow & (1') & \downarrow & (2') & \downarrow \\ \varrho_{10}\colon & L_1 & \underset{l_{10}}{\hookleftarrow} & L_{10} & \underset{r_{10}}{\hookrightarrow} & E_1 \\ & & & \triangle & & & \\ & & & \mathrm{ac}_{L_0} & & & \end{array}$$

exists, and the application conditions ac_L and ac_{L_1} are *compatible*, that is, there is some application condition $\mathrm{ac}_{L_{10}}$ over L_{10} such that

$$\mathrm{ac}_{L_1} \equiv_{\varrho_1} \mathrm{Shift}(k_1, \mathrm{ac}_L) \wedge \mathrm{L}(\varrho_{10}, \mathrm{Shift}(r_{10}, \mathrm{ac}_{L_{10}}))$$

where $r_{10}\colon L_{10} \hookrightarrow E_1$ and ϱ_{10} is the rule derived from ϱ and k_1, that is, (1′) and (2′) in the above diagram are pushouts. A rule ϱ_1' is a *remainder* of ϱ_1 with respect to ϱ if $\varrho_1 = \varrho *_{E_1} \varrho_1'$ for some E_1-dependency relation for ϱ and ϱ_1'.

Example 5.2. We want to model an additional behaviour of the system in which two active, waiting processes without a turn variable may decide to activate a disabled resource and one of them gets the turn variable. The first rule appears at the top of Figure 13 and shows the handling of the first process – its flag is redirected and it gets the new turn variable. The second rule is shown at the bottom of the figure, and all it does is redirect the flag of a process to a previously disabled resource. The middle row of the figure shows the subrule, which has to ensure that the newly enabled resource and its turn variable are synchronised. This rule is actually a subrule of ϱ_7 and ϱ_8 because the given squares are pullbacks, in both cases the pushout complements exist and are equal to the left-hand sides of the corresponding rule and, for the application conditions, we have

$$\mathrm{ac}_i \cong \mathrm{Shift}(k_i, \mathrm{ac}_0) \wedge \mathrm{L}(\rho_i^*, \mathrm{Shift}(r_i, \neg\exists b_i))$$

for $i = 7, 8$. The remainder rules ϱ_7' and ϱ_8' are shown in Figure 14. Note that in ϱ_7', the turn variable appears because it has to be connected to the process, but it is not needed in ϱ_8'. In addition, the application condition $\neg\exists b_i$ is translated into an application condition $\neg\exists b_i'$ for both remainder rules with $i = 7, 8$.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

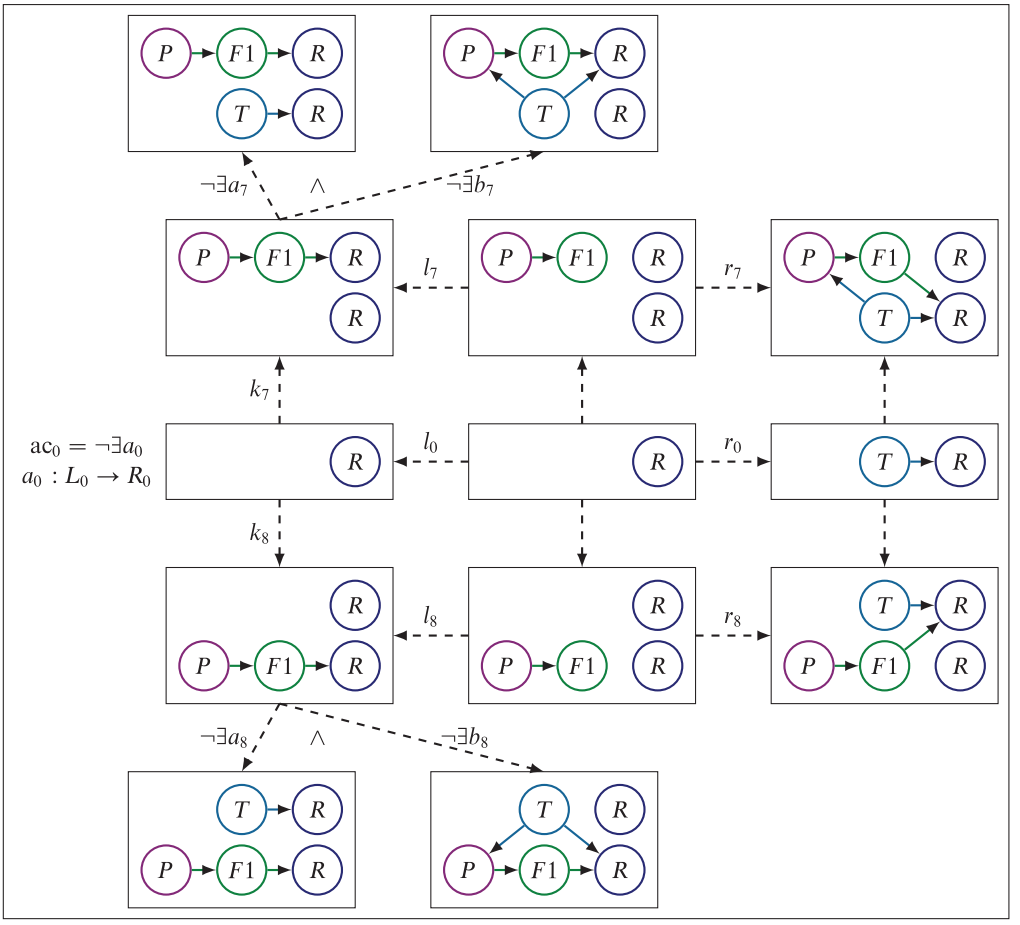


Fig. 13. (Colour online) The subrule ϱ_0 of the rules ϱ_7 and ϱ_8.

Every rule can be decomposed into the subrule and a remainder.

Theorem 5.3 (existence of a remainder Golas *et al.* (2014)). For every rule ϱ_1 with subrule ϱ, there is a remainder ϱ_1' of ϱ_1 with respect to ϱ.

The construction of an amalgamated rule generalises the construction of a parallel rule $\varrho_1 + \varrho_2$ of the rules ϱ_1 and ϱ_2: for a common subrule ϱ of ϱ_1 and ϱ_2, the ϱ-amalgamated rule $\varrho_1 \oplus_\varrho \varrho_2$ of ϱ_1 and ϱ_2 can be defined with the help of pushouts along $\mathcal{M}$-morphisms of the components of the rules. This generalises the construction of amalgamated rules for rules without application conditions (Boehm *et al.* 1987; Corradini *et al.* 1997) and makes use of the shifting of application conditions over morphisms (Lemma 3.11).

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

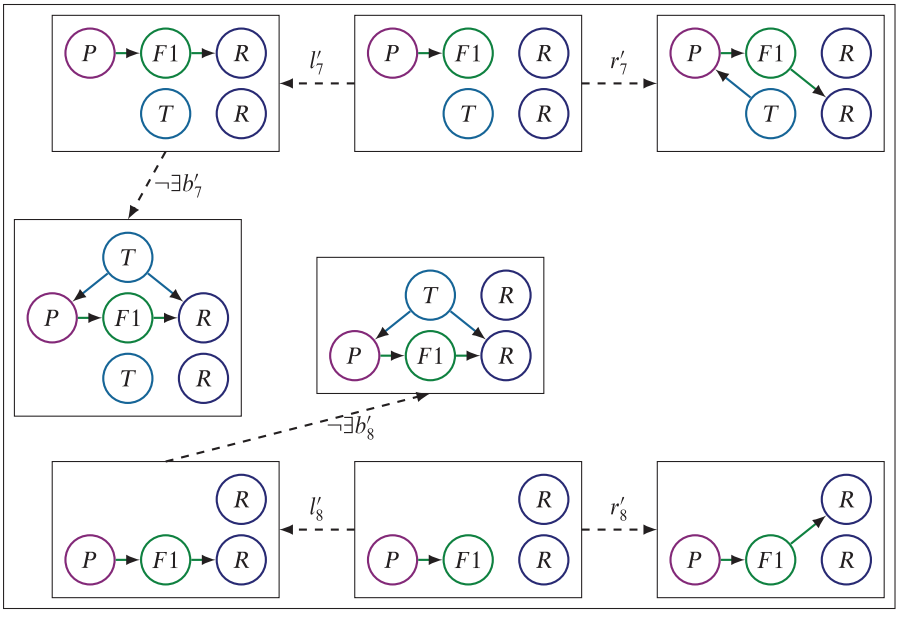
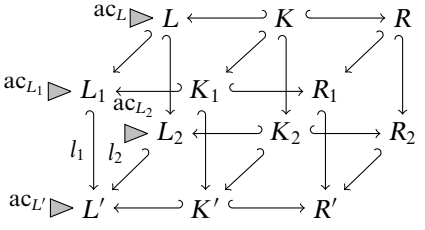


Fig. 14. (Colour online) The remainder rules ϱ'_7 and ϱ'_8.

Definition 5.4 (amalgamated rule). Consider the diagram



Given a common subrule ϱ of rules ϱ_1 and ϱ_2, the *ϱ-amalgamated rule* of ϱ_1 and ϱ_2, denoted by $\varrho_1 \oplus_\varrho \varrho_2$, is the rule $\langle p', \mathrm{ac}_{L'} \rangle$, where L', K' and R' are the pushout objects in the left, middle and right diagram, respectively, $K' \to L'$ and $K' \to R'$ are the uniquely existing morphisms,

$$p' = \langle L' \leftarrow K' \rightarrow R' \rangle,$$

and

$$\mathrm{ac}_{L'} = \mathrm{Shift}(l_1, \mathrm{ac}_{L_1}) \wedge \mathrm{Shift}(l_2, \mathrm{ac}_{L_2}).$$

Note that the morphisms $K' \hookrightarrow L'$ and $K' \hookrightarrow R'$ are in $\mathcal{M}$.

Example 5.5. The amalgamated rule $\varrho = \varrho_7 \oplus_{\varrho_0} \varrho_8$ is shown in the upper rows of Figure 15. It combines the effects of ϱ_7 and ϱ_8, where both rules use the same resource as the new target of the flags and only create one turn variable for this resource. Note that the application condition $\neg\exists d$ prevents the upper and lower processes being matched non-

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

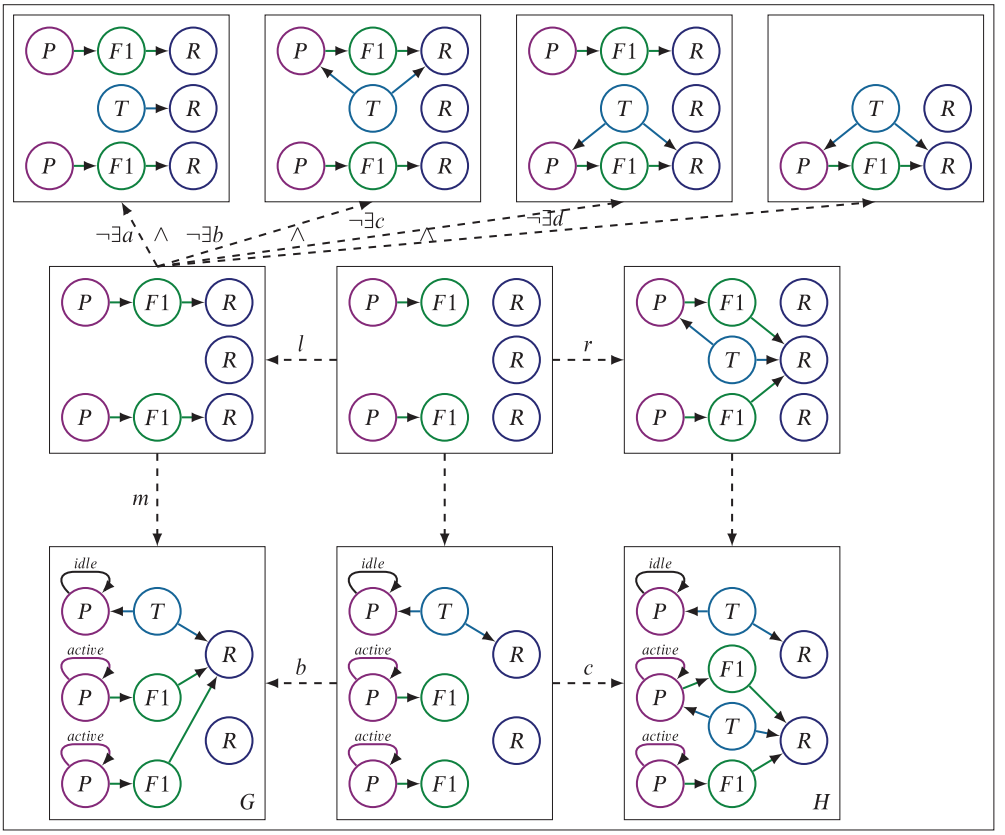


Fig. 15. (Colour online) The amalgamated rule $\varrho = \varrho_7 \oplus_{\varrho_0} \varrho_8$.

injectively and are connected via a turn variable to the resource – other overlappings, which may not occur in valid systems, are not shown explicitly.

By definition, parallel rules are special amalgamated rules.

Fact 5.6 (parallel rules are amalgamated rules). If $\langle \mathcal{C}, \mathcal{M} \rangle$ has an $\mathcal{M}$-initial object I, then $\text{init} = \langle I \hookleftarrow I \hookrightarrow I \rangle$ is a subrule of ϱ_1 and of ϱ_2 and $\varrho_1 + \varrho_2 \cong \varrho_1 \oplus_{\text{init}} \varrho_2$.

The subrule property is inherited by amalgamated rules: if a rule is a common subrule of rules, then these rules are subrules of the amalgamated rule of the rules.

Lemma 5.7 (subrule inheritance Golas *et al.* 2014)). If ϱ is a common subrule of ϱ_1 and ϱ_2, then ϱ_1 and ϱ_2 are subrules of $\varrho_1 \oplus_\varrho \varrho_2$.

The application of an amalgamated rule yields an amalgamated transformation. Amalgamability of direct transformations generalises parallel independence of direct transformations.

Definition 5.8 (amalgamated and amalgamable transformation). A direct transformation via a ϱ-amalgamated rule is called a *ϱ-amalgamated* direct transformation, or a

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

ϱ-*amalgamated* transformation for short. For $i = 1, 2$, the direct transformations

$$G \Rightarrow_{\varrho_i, m_i} H_i$$

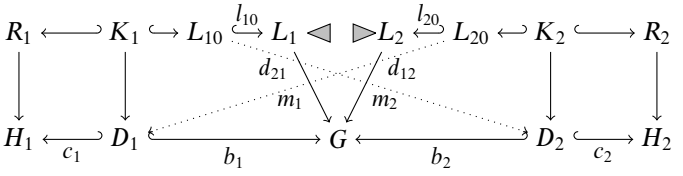
via

$$\varrho_i = \varrho *_{E_i} \varrho_i'$$

are ϱ-*amalgamable* if the matches are *consistent*, that is,

$$m_1 \circ k_1 = m_2 \circ k_2 = m,$$

and, for $i \neq j$, there is a pushout complement L_{i0} of $K \hookrightarrow L \hookrightarrow^{k_i} L_i$ as in Definition 5.1, and there is a morphism $d_{ij} : L_{i0} \to D_j$ such that $b_j \circ d_{ij} = m_i \circ l_{i0}$ and $c_j \circ d_{ij} \models \mathrm{ac}_{L_{i0}}$.



Remark 5.9. The definition of amalgamable direct transformations generalises the definition of parallel independent transformations by requiring the existence of morphisms $L_{i0} \to D_j$ instead of morphisms $L_i \to D_j$.

Example 5.10. Figure 15 shows the amalgamated transformation $G \Rightarrow_{\varrho, m} H$, which applies the amalgamated rule ϱ to the graph G. The two processes with a flag waiting for one resource enable the second, previously disabled resource, and the upper process gets the turn variable.

Fact 5.11 (parallel independence implies init-amalgamability). Parallel independence of direct transformations $G \Rightarrow_{\varrho_i, m_i} H_i$ implies init-amalgamability of $G \Rightarrow_{\varrho_i, m_i} H_i$ where init-amalgamability means ϱ-amalgamability with $\varrho = \mathrm{init} = \langle I \hookleftarrow I \hookrightarrow I \rangle$.

Proof. Let $G \Rightarrow_{\varrho_i, m_i} H_i$ be parallel independent, that is, there are morphisms

$$d_{ij} : L_i \to D_j$$

such that $b_j \circ d_{ij} = m_i$ and $c_j \circ d_{ij} \models \mathrm{ac}_{L_i}$.

For the initial rule $\varrho = \mathrm{init}$, we have

$$\begin{aligned} L_{i0} &= L_i \\ l_{i0} &= \mathrm{id} \\ \mathrm{ac}_{L_{i0}} &= \mathrm{ac}_{L_i} \\ G &\cong G' \\ D_j &= D_j' \\ b_j' &= b_j. \end{aligned}$$

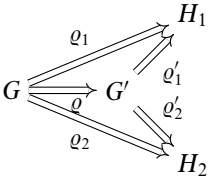
https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Thus, there are morphisms $d_{ij} : L_{i0} \to D'_j$ such that $b'_j \circ d_{ij} = b_j \circ d_{ij} = m_i = m_i \circ l_{i0}$ and $c_j \circ d_{ij} \models \mathrm{ac}_{L_{i0}}$, so

$$G \Rightarrow_{\varrho_i, m_i} H_i$$

is init-amalgamable. □

Lemma 5.12 (amalgamability implies parallel independence (Golas *et al.* 2014)). For a common subrule ϱ of ϱ_1 and ϱ_2, we have that the ϱ-amalgamability of direct transformations $G \Rightarrow_{\varrho_i, m_i} H_i$ via $\varrho_i = \varrho *_{E_i} \varrho'_i$ implies parallel independence of the direct transformations $G' \Rightarrow_{\varrho'_i, m'_i} H_i$ where $G \Rightarrow_{\varrho, m} G'$ and $m = m_i \circ k_i$ for $i = 1, 2$:



We will now present an Amalgamation Theorem for rules with application conditions that generalises the well-known Amalgamation Theorem for rules without application conditions (Boehm *et al.* 1987) and specialises the Multi-Amalgamation Theorem (Golas *et al.* 2014) to the case of the amalgamation of two rules.

Theorem 5.13 (Amalgamation Theorem). Let $\varrho' = \varrho_1 \oplus_\varrho \varrho_2$ and $\varrho' = \varrho_i *_{E'_i} \varrho'_i$ for $i = 1, 2$. Given ϱ-amalgamable direct transformations $G \Rightarrow_{\varrho_i, m_i} H_i$, there is a ϱ-amalgamated transformation $G \Rightarrow_{\varrho', m'} M$ and, for $i = 1, 2$, a direct transformation $H_i \Rightarrow_{\varrho'_i} M$ via ϱ'_i such that

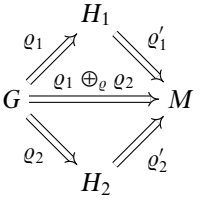
$$G \Rightarrow_{\varrho_i, m_i} H_i \Rightarrow_{\varrho'_i} M$$

is a decomposition of $G \Rightarrow_{\varrho', m'} M$.

Given a ϱ-amalgamated direct transformation $G \Rightarrow_{\varrho', m'} M$, there is, for $i = 1, 2$, a transformation

$$G \Rightarrow_{\varrho_i, m_i} H_i \Rightarrow_{\varrho'_i} M$$

such that the direct transformations $G \Rightarrow_{\varrho_i, m_i} H_i$ are ϱ-amalgamable.



Proof. The theorem follows immediately from the Multi-Amalgamation Theorem in Golas *et al.* (2014) for the case $n = 2$. □

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

6. Related work

In this section, we describe some related work.

6.1. *Regulated string, term and graph rewriting*

In standard graph transformation (Ehrig 1979), as in standard string rewriting (Salomaa 1973) and standard term rewriting (Baader and Nipkow 1998), a rule can always be applied to a graph if a match is found. However, there are many situations where we would only want to apply a rule if certain conditions are met. The approach to restricting the applicability of rules in graph transformation may look, at least superficially, similar to the approaches used in string and term rewriting, but the approach for term rewriting is actually very different.

Regulated string rewriting. In string rewriting, there are several approaches for regulated rewriting (Salomaa 1973; Dassow and Păun 1989), for example, matrix, programmed and random context rewriting. There are various applications of formal language theory where context-free grammars are not enough, thus motivating the introduction of regulated (context-free) grammars. Moreover, there are several other applications of regulated rewriting, for example, relationships with programming languages, regulated rewriting and Petri nets, and modelling of economic processes. Context-sensitive string rewriting (Salomaa 1973), random context rewriting (Dassow and Păun 1989) and string rewriting with local and global context conditions (Csuhaj-Varjú 1993) correspond to context-free graph transformation with positive application conditions.

Conditional term rewriting. In term rewriting, we use conditional rules (Baader and Nipkow 1998), where the conditions have a logical (or operational) nature. Typically, conditions are lists of equations that must be satisfied for the given match, where satisfaction is checked by term rewriting (usually by checking if the terms of each equation can be rewritten into a common term). This means that the process required to see if a rule can be applied to a given term is recursive: to check the applicability of a rule, we have to evaluate its conditions, which means applying other rules. In fact, determining the applicability of conditional rules is undecidable in the general case. Moreover, this recursivity causes various difficulties when trying to extend some results for standard term rewriting to conditional term rewriting.

6.2. *Local and non-local graph conditions*

In graph transformation, we restrict the applicability of rules using application conditions, which essentially have a syntactic nature. In particular, we check the existence (or non-existence) of a given structure that includes the matching. This means that checking application conditions is essentially a matching problem. This is possibly one reason why we are able to extend all the fundamental results of standard graph transformation to this case. Finite nested conditions are expressively equivalent to first-order formulas and

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

local properties (Habel and Pennemann 2009). Non-local properties like 'there exists a path', 'is connected' and 'is cycle-free' are not expressible by finite nested conditions, but can be expressed by finite HR^+ conditions (Habel and Radke 2010), that is, finite nested conditions with variables where the variables are place-holders for graphs and the graphs are generated by a hyperedge replacement (HR) system. (Node-)Counting monadic second-order formulas can be transformed into finite HR^+ conditions, though the reverse direction is not clear.

6.3. *Local Church–Rosser, parallelism and confluence for left-linear rules*

Adhesive categories provide an abstract setting for the double-pushout approach to rewriting, which generalises classical approaches to graph transformation. Fundamental results about parallelism and confluence, including the Local Church–Rosser Theorem, can be proved in adhesive categories, provided we only use linear rules, that is, rules $\langle L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R \rangle$ with l mono and r arbitrary. Baldan *et al.* (2011) identifies a class of categories, including most adhesive categories used in rewriting, where those same results can be proved in the presence of rules that are merely left-linear, that is, rules that can merge different parts of a rewritten object. Such rules naturally emerge, for example, when using graphical encodings for modelling the operational semantics of process calculi.

6.4. *Local Church–Rosser, termination and confluence*

Graph transformation has learnt lessons from term rewriting: the Church–Rosser and Confluence Theorems were originally developed for term rewriting. Checking local confluence for term rewriting is based on the essential technique for analysing critical pairs (Knuth and Bendix 1970) and makes use of powerful techniques available for checking termination. If termination is ensured, the local (and global) confluence of the system is shown by checking for all critical pairs. If the system is not confluent, we may apply the (Knuth–Bendix) completion procedures and try to transform the system into a confluent one by converting all non-confluent critical pairs into rewrite rules (Baader and Nipkow 1998). Checking local confluence for graph transformation (without application conditions) is similar (Plump 2005; Ehrig *et al.* 2006b), though in this case the test only provides a sufficient condition because local confluence for graph transformation systems is undecidable, even for terminating systems (Plump 2005).

6.5. *Weakest preconditions and proof systems*

Nested graph conditions are used in the verification of graph programs: graph programs (Habel and Plump 2001) generalise the notions of programs on linear structures (Dijkstra 1976) to graphs. For graph programs, (extensions of) nested graph conditions are used as preconditions and postconditions. A well-known method for showing the correctness of a program with respect to a precondition and a postcondition (Dijkstra 1976) is to construct a weakest precondition of the program relative to the postcondition and then prove that the precondition implies the weakest precondition. Habel *et al.* (2006)

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

uses the framework of graphs to construct weakest preconditions for graph programs, and Pennemann (2009) uses his algorithm for approximating the satisfiability problem and his resolution-like theorem prover for graph conditions to try to prove that the precondition implies the weakest precondition. A well-known method for verifying the partial correctness of a program with respect to a precondition and a postcondition (Hoare 1969) is to give a proof system and then show its soundness with respect to the operational semantics of the program. Poskitt and Plump (2012) uses the framework of graphs for verifying the partial correctness of a graph program in the graph programming language GP, and then show the soundness with respect to the operational semantics of GP.

6.6. *Weakest preconditions and local confluence*

Bruggink *et al.* (2011) enrich the formalism of reactive systems using the notion of nested application conditions from graph transformation systems to reactive systems, and then shows that some constructions for graph transformation systems (such as computing weakest preconditions and strongest postconditions and showing local confluence by means of critical pair analysis) can be done elegantly in the more general setting.

6.7. *Model transformation*

Negative application conditions, and more generally, nested application conditions, are a key ingredient for many model transformations based on graph transformation. The concept of negative application conditions is often used in Ehrig *et al.* (2009a) to define expressive model transformations and to allow the modeller to specify complex model transformations. The authors of the current paper are currently working on an extension of model transformations based on triple graph grammars to the more general nested applications.

6.8. *OCL constraints*

Nested graph conditions are often used for specifications: for instance, in (UML) model transformations. Restricted OCL constraints (Winkelmann *et al.* 2008; Ehrig *et al.* 2009) can be translated to equivalent local graph constraints such as the existence or non-existence of certain structures (like nodes and edges or subgraphs) in an instance graph (positive constraints have to be checked after the generation of a meta-model instance, but negative graph constraints can be checked during the generation) and, by transformation $\mathcal{A}$ in Habel and Pennemann (2009), graph constraints can be transformed into equivalent application conditions for the corresponding rules. (Note that graph constraints equal application conditions over the empty graph.)

7. Conclusions

In the current paper, we have presented the well-known Local Church–Rosser, Parallelism, Concurrency and Amalgamation Theorems for rules with nested application conditions

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

in the framework of $\mathcal{M}$-adhesive categories. The proofs for transformation systems with nested application conditions are based on the corresponding theorems for transformation systems without application conditions (Ehrig *et al.* 2006b) and two shift lemmas saying that application conditions can be shifted over morphisms and rules. The first shift lemma (Lemma 3.11) requires $\mathcal{E}'$-$\mathcal{M}$ pair factorisation. In addition to this, the Parallelism Theorem also requires binary coproducts and the Amalgamation Theorem also requires initial pushouts over $\mathcal{M}$-morphisms (Golas *et al.* 2014). Summarising, we have

Theorem	Category	Additional requirements
Local Church–Rosser	$\mathcal{M}$-adhesive	$\mathcal{E}'$-$\mathcal{M}$ pair factorisation
Parallelism	$\mathcal{M}$-adhesive	$\mathcal{E}'$-$\mathcal{M}$ pair factorisation and binary coproducts
Concurrency	$\mathcal{M}$-adhesive	$\mathcal{E}'$-$\mathcal{M}$ pair factorisation
Amalgamation	$\mathcal{M}$-adhesive	$\mathcal{E}'$-$\mathcal{M}$ pair factorisation and initial pushouts over $\mathcal{M}$

Golas *et al.* (2014) gives a Multi-Amalgamation Theorem for nested application conditions in the framework of $\mathcal{M}$-adhesive categories. This generalises our Amalgamation Theorem to the case of $n \geqslant 2$ amalgamable direct transformations; Theorem 5.3 (the existence of a remainder) requires $\mathcal{E}'$-$\mathcal{M}$ pair factorisation and initial pushouts over $\mathcal{M}$. Part 2 of the current paper (Ehrig *et al.* 2012) gives the Embedding and Local Confluence Theorems for nested application conditions in the framework of $\mathcal{M}$-adhesive categories, and the results require $\mathcal{E}'$-$\mathcal{M}$ pair factorisation and initial pushouts over $\mathcal{M}$-morphisms. Using the hierarchies of adhesive categories (graph $\Rightarrow$ high-level $\Rightarrow$ weak adhesive HLR $\Rightarrow$ $\mathcal{M}$-adhesive) and application conditions (none $\Rightarrow$ negative $\Rightarrow$ nested), we obtain all results for all these types of categories and application conditions. The following tables provide a summary:

Concurrency	none	negative	nested
graph	√	√	[√]
high-level	√	√	[√]
weak adhesive HLR	√	√	[√]
$\mathcal{M}$-adhesive	[√]	[√]	[√]

Amalgamation	none	negative	nested
graph	√	[√]	[√]
high-level	[√]	[√]	[√]
weak adhesive HLR	[√]	[√]	[√]
$\mathcal{M}$-adhesive	[√]	[√]	[√]

So, the Local Church–Rosser, Parallelism and Concurrency Theorems, which were previously known for weak adhesive HLR transformations systems with negative application conditions (Lambers 2010), marked above by √ in the tables, also hold for proper

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

$\mathcal{M}$-adhesive transformations systems with proper nested application conditions, marked by $\boxed{\surd}$. And the Amalgamation Theorem, which was previously only known for graph transformations systems without application conditions (Boehm *et al.* 1987), also holds for all $\mathcal{M}$-adhesive transformations systems with nested application conditions.

Acknowledgements

We are very grateful to the referees for their careful reading of the draft of this paper and for their stimulating remarks and suggestions, which led to a considerably improved exposition.

References

Adámek, J., Herrlich, H. and Strecker, G. (1990) *Abstract and Concrete Categories*, John Wiley.

Arbib, M. A. and Manes, E. G. (1975) *Arrows, Structures, and Functors*, Academic Press.

Baader, F. and Nipkow, T. (1998) *Term Rewriting and All That*, Cambridge University Press.

Baldan, P., Gadducci, F. and Sobocinski, P. (2011) Adhesivity is not enough: Local Church–Rosser revisited. In: Mathematical Foundations of Computer Science (MFCS 2011). *Springer-Verlag Lecture Notes in Computer Science* **6907** 48–59.

Biermann, E., Ehrig, H., Ermel, C., Golas, U. and Taentzer, G. (2010) Parallel independence of amalgamated graph transformations applied to model transformation. In: Graph Transformations and Model-Driven Engineering. *Springer-Verlag Lecture Notes in Computer Science* **5765** 121–140.

Boehm, P., Fonio, H.-R. and Habel, A. (1987) Amalgamation of graph transformations: A synchronization mechanism. *Journal of Computer and System Sciences* **34** 377–408.

Bruggink, H. J. S., Cauderlier, R., Hülsbusch, M. and König, B. (2011) Conditional reactive systems. In: *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011)* 191–203.

Castellani, I. and Montanari U. (1983) Graph grammars for distributed systems. In: Graph Grammars and Their Application to Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **153** 20–38.

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R. and Löwe, M. (1997) Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach. In: *Handbook of Graph Grammars and Computing by Graph Transformation* **1**, World Scientific 163–245.

Corradini, A., Rossi, F. and Parisi-Presicce F. (1991) Logic programming as hypergraph rewriting. In: Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91). *Springer-Verlag Lecture Notes in Computer Science* **493** 275–295.

Courcelle, B. (1997) The expression of graph properties and graph transformations in monadic second-order logic. In: *Handbook of Graph Grammars and Computing by Graph Transformation* **1**, World Scientific 313–400.

Csuhaj-Varjú, E. (1993) On grammars with local and global context conditions. *International Journal of Computer Mathematics* **47** 17–27.

Dassow, J. and Păun, G. (1989) *Regulated Rewriting in Formal Language Theory*, EATCS Monographs on Theoretical Computer Science **18**, Springer-Verlag.

Degano, P. and Montanari, U. (1987) A model of distributed systems based on graph rewriting. *Journal of the ACM* **34** 411–449.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Dijkstra, E. W. (1965) Solution of a Problem in Concurrent Programming Control. *Communications of the ACM* **8** 569.

Dijkstra, E. W. (1976) *A Discipline of Programming*, Prentice-Hall.

Ehrig, H. (1979) Introduction to the algebraic theory of graph grammars. In: Graph-Grammars and Their Application to Computer Science and Biology. *Springer-Verlag Lecture Notes in Computer Science* **73** 1–69.

Ehrig, H. and Habel, A. (1986) Graph grammars with application conditions. In: Rozenberg, G. and Salomaa, A. (eds.) *The Book of L*, Springer-Verlag 87–100.

Ehrig, H. and Kreowski, H.-J. (1980) Applications of graph grammar theory to consistency, synchronization and scheduling in database systems. *Information Systems* **5** 225–238.

Ehrig, H. and Parisi-Presicce, F. (1992) High-level-replacement systems for equational algebraic specifications. In: Algebraic and Logic Programming – proceedings Third International Conference. *Springer-Verlag Lecture Notes in Computer Science* **632** 3–20.

Ehrig, H. and Rosen, B. (1980) Parallelism and concurrency of graph manipulations. *Theoretical Computer Science* **11** 247–275.

Ehrig, H., Ehrig, K., Habel, A. and Pennemann, K.-H. (2006) Theory of constraints and application conditions: From graphs to high-level structures. *Fundamenta Informaticae* **74** (1) 135–166.

Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G. (2006) *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science, Springer-Verlag.

Ehrig, H., Engels, G., Kreowski, H.-J. and Rozenberg, G. (eds.) (1999) *Handbook of Graph Grammars and Computing by Graph Transformation 2: Applications, Languages and Tools*, World Scientific.

Ehrig, H., Golas, U. and Hermann, F. (2010) Categorical Frameworks for Graph Transformation and HLR Systems based on the DPO Approach. *Bulletin of the EATCS* **112** 111–121.

Ehrig, H., Golas, U., Habel, A., Lambers, L. and Orejas, F. (2012) $\mathcal{M}$-Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence. *Fundamenta Informaticae* **118** 35–63

Ehrig, H., Habel, A. and Lambers, L. (2010) Parallelism and concurrency theorems for rules with nested application conditions. *Electronic Communications of the EASST* **26**.

Ehrig, H., Habel, A. and Rosen, B. K. (1986) Concurrent transformations of relational structures. *Fundamenta Informaticae* **IX** 13–50.

Ehrig, H., Habel, A., Kreowski, H.-J. and Parisi-Presicce, F. (1991) Parallelism and concurrency in high level replacement systems. *Mathematical Structures in Computer Science* **1** 361–404.

Ehrig, H., Habel, A., Padberg, J. and Prange, U. (2006) Adhesive high-level replacement systems: A new categorical framework for graph transformation. *Fundamenta Informaticae* **74** 1–29.

Ehrig, H., Hermann, F. and Sartorius, C. (2009) Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. *Electronic Communications of the EASST* **18**.

Ehrig, H., Kreowski, H.-J., Montanari, U. and Rozenberg, G. (eds.) (1999) *Handbook of Graph Grammars and Computing by Graph Transformation 3: Concurrency, Parallelism, and Distribution*, World Scientific.

Ehrig, K., Küster, J. M. and Taentzer, G. (2009) Generating instance models from meta models. *Software and System Modeling* **8** (4) 479–500.

Ehrig, H., Pfender, M. and Schneider H.-J. (1973) Graph grammars: An algebraic approach. In: *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory* 167–180.

Golas, U. (2011) Analysis and correctness of algebraic graph and model transformation, Vieweg+ Teubner Research.

Golas, U., Habel, A. and Ehrig, H. (2014) Multi-Amalgamation in $\mathcal{M}$-Adhesive Categories. *Mathematical Structures in Computer Science* (this volume).

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Habel, A. and Pennemann, K.-H. (2009) Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* **19** 245–296.

Habel, A. and Plump, D. (2001) Computational completeness of programming languages based on graph transformation. In: Foundations of Software Science and Computation Structures – proceedings FOSSACS 2001. *Springer-Verlag Lecture Notes in Computer Science* **2030** 230–245.

Habel, A. and Radke, H. (2010) Expressiveness of graph conditions with variables. *Electronic Communications of the EASST* **30**.

Habel, A., Heckel, R. and Taentzer, G. (1996) Graph grammars with negative application conditions. *Fundamenta Informaticae* **26** 287–313.

Habel, A., Pennemann, K-H. and Rensink, A. (2006) Weakest preconditions for high-level programs. In: Graph Transformations (ICGT 2006). *Springer-Verlag Lecture Notes in Computer Science* **4178** 445–460.

Heckel, R. and Wagner, A. (1995) Ensuring consistency of conditional graph grammars – a constructive approach. In: Workshop on Graph Rewriting and Computation – proceedings SEGRAGRA'95. *Electronic Notes in Theoretical Computer Science* **2** 95–104.

Heckel, R., Llabrés, M., Ehrig, H. and Orejas, F. (2002) Concurrency and loose semantics of open graph transformation systems. *Mathematical Structures in Computer Science* **12** (4) 349–376.

Heindel, T. (2010) Hereditary Pushouts Reconsidered. In: Graph Transformations (ICGT'10). *Springer-Verlag Lecture Notes in Computer Science* **6372** 250–265.

Hoare, C. A. R. (1969) An axiomatic basis for computer programming. *Communications of the ACM* **12** 576–580, 583.

Knuth, D. E. and Bendix, P. B . (1970) Simple word problems in universal algebras. In: *Computational Problems in Abstract Algebras*, Pergamon Press 263–297.

Koch, M., Mancini, L. V. and Parisi-Presicce, F. (2005) Graph-based specification of access control policies. *Journal of Computer and System Sciences* **71** 1–33.

Kreowski, H.-J. (1977) *Manipulationen von Graphmanipulationen*, Ph.D. thesis, Technical University of Berlin.

Lack, S. and Sobociński P. (2004) Adhesive categories. In: Foundations of Software Science and Computation Structures (FOSSACS'04). *Springer-Verlag Lecture Notes in Computer Science* **2987** 273–288.

Lack, S. and Sobociński P. (2005) Adhesive and quasiadhesive categories. *Theoretical Informatics and Application* **39** (2) 511–546.

Lambers, L. (2010) *Certifying Rule-Based Models using Graph Transformation*, Ph.D. thesis, Technical University of Berlin.

Mahr, B. and Wilharm, A. (1982) Graph grammars as a tool for description in computer processed control: A case study. In: *Graph-Theoretic Concepts in Computer Science*, Hanser Verlag, München 165–176.

Parisi-Presicce, F. (1989) Modular system design applying graph grammar techniques. In: Automata, Languages and Programming – proceedings ICALP89. *Springer-Verlag Lecture Notes in Computer Science* **372** 621–636.

Pennemann, K.-H. (2009) *Development of Correct Graph Transformation Systems*, Ph.D. thesis, Universität Oldenburg.

Plump, D. (2005) Confluence of graph transformation revisited. In: Processes, Terms and Cycles: Steps on the Road to Infinity – Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday. *Springer-Verlag Lecture Notes in Computer Science* **3838** 280–308.

Poskitt, C. M. and Plump, D. (2012) Hoare-style verification of graph programs. *Fundamenta Informaticae* **118** 135–175.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press

Rensink, A. (2004) Representing first-order logic by graphs. In: Graph Transformations – proceedings ICGT'04. *Springer-Verlag Lecture Notes in Computer Science* **3256** 319–335.

Ribeiro, L. (1996) A telephone's system specification using graph grammars. Technical report 96-23, Technical University of Berlin.

Rosen, B. K. (1975) A Church–Rosser theorem for graph grammars (announcement). *SIGACT News* 7 (3) 26–31.

Rozenberg, G. (ed.) (1997) *Handbook of Graph Grammars and Computing by Graph Transformation 1: Foundations*, World Scientific.

Salomaa, A. (1973) *Formal Languages*, Academic Press.

Taentzer, G., Koch, M., Fischer, I. and Volle, V. (1999) Distributed graph transformation with application to visual design of distributed systems. In: Ehrig, H., Kreowski, H.-J., Montanari, U. and Rozenberg, G. (eds.) *Handbook of Graph Grammars and Computing by Graph Transformation 3: Concurrency, Parallelism, and Distribution*, World Scientific 269–340.

Winkelmann, J., Taentzer, G., Ehrig, K. and Küster, J. M. (2008) Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. In: Proceedings GT-VMT 2006. *Electronic Notes in Theoretical Computer Science* **211** 159–170.

https://doi.org/10.1017/S0960129512000357 Published online by Cambridge University Press