High-Level Programs and Program Conditions*

Karl Azab and Annegret Habel

Carl v. Ossietzky Universität Oldenburg, Germany
{azab,habel}@informatik.uni-oldenburg.de

Abstract. High-level conditions are well-suited for expressing structural properties. They can describe the precondition and the postcondition for a high-level program, but they cannot describe the relationship between the input and the output of a program derivation. Therefore, we investigate program conditions, a generalized type of conditions expressing properties on program derivations. Program conditions look like nested rules with application conditions. We present a normal form result, a suitable graphical notation, and conditions under which a satisfying program can be constructed from a program condition. We define a sequential composition on program conditions and show that, for a suitable type of program conditions with a complete dependence relation we have that: Whenever the original programs satisfy the original program conditions, then the composed program satisfies the composed program condition.

1 Introduction

Constraints, also called graph conditions, in the sense of [1,2,3,4] are a visual and intuitive, yet precise formalism, well suited to describe structural properties of system states. With these concepts we can express requirements for a program by a pair $\langle \text{pre}, \text{post} \rangle$ of a pre- and a postcondition. A relationship between the input and the output graph cannot be expressed. In imperative programming languages, the relationship between the input and the output is given by the names of variables: in general, the value of a variable after program execution depends on its value before. In this paper, we introduce so-called program conditions which allows us to express the relationship between the input- and output object for high-level programs. The concept is illustrated in the category of graphs by a simple example of repairs of broken routers in a computer network.

Example 1 (router repair). Consider a simple network consisting of computers () and routers () as nodes. Network links between routers and computers are shown as undirected edges (in reality two directed edges in the opposite direction). Inoperable routers have an incoming edge from a failure () node. Broken routers can be put on a priority list for quicker repair, which is modelled by an edge from a priority () node. Figure 1 shows a graph G which is transformed

* This work is supported by the German Research Foundation (DFG) under grant no. HA 2936/2 (Development of Correct Graph Transformation Systems).

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 211–225, 2008.

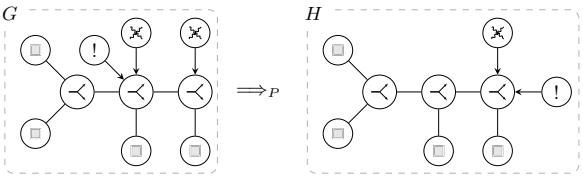


Fig. 1. Failing routers that were on the priority list should be repaired

into H by a graph program P. We will show how our program conditions can express temporal properties over such pairs, e.g. "Every router that were on the priority list has been repaired after the execution of P".

The paper is organized as follows: In Sect. 2, we review the definitions of conditions and rules. In Sect. 3, we introduce programs and program conditions, a generalized type of conditions expressing properties on program derivations. We also prove a normal form result for program conditions saying that, for every program condition, there is an equivalent program condition without pre- and postcondition. For this type of program conditions, a graphical notation is presented. In Sect. 4, for specific program conditions, a satisfying program is constructed. Moreover, for composed programs, a satisfying program condition is composed from the program conditions of the subprograms. The concepts are illustrated by examples in the category of graphs with the class of all injective graph morphisms. A conclusion including further work is given in Sect. 5.

2 Conditions and Rules

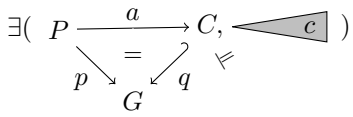
We use the framework of weak adhesive HLR categories [3,5] and introduce conditions and rules for high-level structures like Petri nets, (hyper)graphs, and algebraic specifications.

Assumption 1. We assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category with $\mathcal{M}$-initial object I satisfying the special pullback-decomposition property [6].

E.g. the category $\langle \mathrm{Graphs}, \mathrm{Inj} \rangle$ of graphs with class Inj of all injective morphisms is a weak adhesive HLR category satisfying the assumption.

(High-level) Conditions are defined as in [7,4]. Syntactically, the conditions may be seen a tree of morphisms equipped with certain logical symbols such as quantifiers and connectives.

Definition 1 (conditions). A *condition* over an object P is of the form true or $\exists(a, c)$, where $a\colon P \to C$ is a morphism and c is a condition over C. Moreover, Boolean formulas over conditions over P are conditions over P. $\exists a$ abbreviates $\exists(a, \mathrm{true})$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$. Every morphism and every object *satisfies* true. A morphism $p\colon P \to G$ *satisfies* a condition $\exists(a, c)$ if there exists a morphism q in $\mathcal{M}$ such that $q \circ a = p$ and $q \models c$.



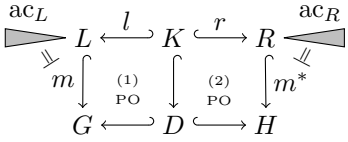
An object G *satisfies* a condition if all morphisms $p\colon P \to G$ in $\mathcal{M}$ satisfy the condition. The satisfaction of conditions over P by objects or morphisms with domain P is extended to Boolean formulas over conditions in the usual way. We write $p \models c$ [$G \models c$] to denote that the morphism p [the object G] satisfies c.

Remark 1. In the context of objects, conditions are also called *constraints* [1] and, in the context of rules, conditions are also called *application conditions*. The definition generalizes those in [1,2,3].

Remark 2. We sometimes use a short notation for conditions: For a morphism $a\colon P \to C$ in a condition, we just depict C, if P can be unambiguously inferred, i.e. for conditions over the $\mathcal{M}$-initial object I. The graph condition $\forall(\emptyset \to \bigcirc, \exists(\bigcirc \hookrightarrow \bigcirc\!\!-\!\!\prec))$ has the meaning "Every computer is connected to a router". In the short form, the condition is written as $\forall(\bigcirc, \exists(\bigcirc\!\!-\!\!\prec))$. A more complex example is the condition $\forall(\bigcirc, \exists(\bigcirc\!\!-\!\!\prec) \wedge \neg\exists(\prec\!\!-\!\!\bigcirc\!\!-\!\!\prec))$ which has the meaning "Every computer is connected to exactly one router".

(High-level) Rules are defined as in [3,4]. They are specified by a span of $\mathcal{M}$-morphisms $\langle L \hookleftarrow K \hookrightarrow R\rangle$ with a left and a right application condition. We consider the classical semantics based on the double-pushout construction [8,9] and restrict on $\mathcal{M}$-matching, i.e. matching morphisms for rules in $\mathcal{M}$.

Definition 2 (rules). A *rule* $\rho = \langle p, \mathrm{ac}_L, \mathrm{ac}_R\rangle$ consists of a plain rule $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$ with $K \hookrightarrow L$ and $K \hookrightarrow R$ in $\mathcal{M}$ and two application conditions ac_L and ac_R over L and R, respectively. L is called the left-hand side, R the right-hand side, and K the interface; ac_L and ac_R are the *left* and *right* application condition of p.



Given a morphism $K \hookrightarrow D$ in $\mathcal{M}$, a *direct derivation* consists of two pushouts (1) and (2) such that $m \models \mathrm{ac}_L$ and $m^* \models \mathrm{ac}_R$. We write $G \Rightarrow_{p,m,m^*} H$ and say that $m\colon L \to G$ is the match, $m^*\colon R \to H$ is the comatch of p in H, and D is the *intermediate object*. We also write $G \Rightarrow_{\rho,m} H$ or $G \Rightarrow_{\rho} H$ to express that there is an m^* and there are m and m^*, respectively, such that $G \Rightarrow_{r,m,m^*} H$.

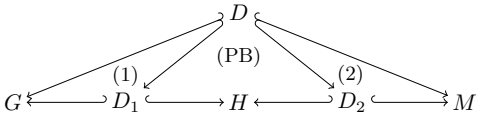
The well-known notions of parallel and sequential independence and the constructions of a parallel, concurrent, and amalgamated rule [10,9] partly have already been extended to rules with negative application conditions. In [11], a Local Church-Rosser, Parallelism, and Concurrency Theorem for rules with

negative application conditions is given. There are generalized Local Church-Rosser, Parallelism, Concurrency and Amalgamation Theorems for rules with application conditions, see the long version of this paper [5].

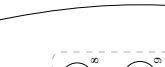
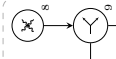
3 Programs and Program Conditions

In this section, we recall the definition of programs and introduce program conditions, a generalized type of conditions expressing properties on program derivations. (High-level) Programs with input-output semantics are defined in [12,7]. The semantics presented here is based on the track morphism of [13].

Definition 3 (programs). Every rule p is a *program*. Every finite set $\mathcal{S}$ of programs is a program. If P and Q are programs, then $(P;Q)$, P^* and $P\downarrow$ are programs. The *semantics* of a program P is a set $[\![P]\!]$ of spans: For a rule p, $[\![p]\!] = \{\langle G \hookleftarrow D \hookrightarrow H\rangle \mid G \Rightarrow_p H \text{ with intermediate object } D\}$. For a finite set $\mathcal{S}$ of programs, $[\![\mathcal{S}]\!] = \cup_{P\in\mathcal{S}}[\![P]\!]$. For programs P and Q, $[\![(P;Q)]\!] = [\![P]\!] \circ [\![Q]\!]$, where the composition of spans is defined by $\langle G \hookleftarrow D_1 \hookrightarrow H\rangle \circ \langle H \hookleftarrow D_2 \hookrightarrow M\rangle = \langle G \hookleftarrow D \hookrightarrow M\rangle$, where D is the pullback object of $D_1 \hookrightarrow H \hookleftarrow D_2$, and (1) and (2) in the diagram below commutes.



The composition is extended to sets of spans in the usual way. For a program P, $[\![P^*]\!]=[\![P]\!]^*$ and $[\![P\downarrow]\!]=\{\langle G \hookleftarrow D \hookrightarrow H\rangle \in [\![P]\!]^* \mid \neg\exists M.\langle H \hookleftarrow E \hookrightarrow M\rangle \in [\![P]\!]\}$.

Example 2. Consider the graph program `Repair`; `Deprioritize`; `Prioritize`, where `Repair` $= \langle$ $\rangle$, `Prioritize` $= \langle$  $\rangle$, and `Deprioritize` $= \langle$  $\rangle$, and the derivation below.

The span $\langle G_1 \hookleftarrow G_3 \hookrightarrow G_4\rangle$ is in the semantics of that program. The trace of the nodes is visualized by indices: The nodes with the indices $1-3$ and $6-10$ are in the interface of the span, the nodes with the indices $4, 5$ are deleted, and the node with index 11 is inserted. The trace of the edges is given implicitly.

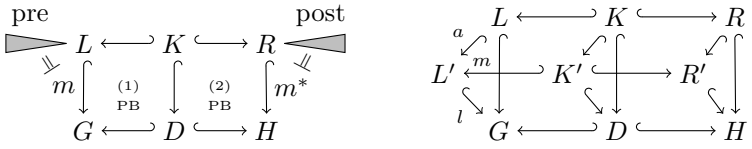
As in [14], we investigate spans and span morphisms. A *span* $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ is a pair of $\mathcal{M}$-morphisms with common domain. A *(span) morphism* from $\mathrm{pc} = \langle L \hookleftarrow K \hookrightarrow R \rangle$ to $\mathrm{pc}' = \langle L' \hookleftarrow K' \hookrightarrow R' \rangle$ is a triple of morphisms $\langle L \to L', K \to K', R \to R' \rangle$ such that (1) and (2) below commute. It is a *(span) $\mathcal{M}$-morphism* if morphisms in the triple are $\mathcal{M}$-morphisms. The class of such span $\mathcal{M}$-morphisms is denoted by $\mathcal{M}^3$.

$$\begin{array}{ccccc} L & \hookleftarrow & K & \hookrightarrow & R \\ {\scriptstyle a}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle a^*} \\ L' & \hookleftarrow & K' & \hookrightarrow & R' \end{array}$$

The *composition* of morphisms is defined by the componentwise composition of the underlying morphisms. The *identity* on span morphisms is defined by the triple of identities on its components. In category theory, spans and $\mathcal{M}$-morphisms form a category.

(High-level) Conditions are well-suited for expressing structural properties. They can be used for describing the precondition and the postcondition for a high-level program, but they cannot be used for describing the relationship between the input and the output of a program. Therefore, we investigate program conditions, a generalized type of conditions expressing properties on program derivations.

Definition 4 (program conditions). A *program condition* (over p) is of the form $\langle p, \mathrm{pre}, \mathrm{post} \rangle$ and called *basic* if $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ is a span of $\mathcal{M}$-morphisms and pre and post are conditions over L over R, respectively, or of the form true or $\exists(\bar{a}, \mathrm{pc})$ and called *nested* if $\bar{a}$ is a span morphism (with domain p) and pc is a program condition (over the codomain of $\bar{a}$). Moreover, Boolean formulas over program conditions yield program conditions. p abbreviates $\langle p, \mathrm{true}, \mathrm{true} \rangle$, $\exists \bar{a}$ abbreviates $\exists(\bar{a}, \mathrm{true})$, and $\forall(\bar{a}, \mathrm{pc})$ abbreviates $\neg\exists(\bar{a}, \neg\mathrm{pc})$. A morphism $\bar{m} = \langle m, d, m^* \rangle$ *satisfies* $\langle p, \mathrm{pre}, \mathrm{post} \rangle$ if (1) and (2) are pullbacks, $m \models \mathrm{pre}$, and $m^* \models \mathrm{post}$. The morphism $\bar{m}$ *satisfies* $\exists(\bar{a}, \mathrm{pc})$ if (1) and (2) are pullbacks and there is an $\mathcal{M}$-morphism $\bar{l}$ with $\bar{m} = \bar{l} \circ \bar{a}$ satisfying pc.



A span $s = \langle G \hookleftarrow D \hookrightarrow H \rangle$ *satisfies* $\langle p, \mathrm{pre}, \mathrm{post} \rangle$ if for all $\mathcal{M}$-morphisms $m\colon L \to G$ satisfying pre, there is an $\mathcal{M}$-morphism $\bar{m} = \langle m, d, m^* \rangle$ satisfying $\langle p, \mathrm{pre}, \mathrm{post} \rangle$. The span s *satisfies* a program condition $\exists a$ $[\exists(a, c)]$ over p if all $\mathcal{M}$-morphisms from p to s satisfy the condition. The satisfaction of program conditions is extended to Boolean formulas over program conditions in the usual way. We write $\bar{m} \models \mathrm{pc}$ $[s \models \mathrm{pc}]$ to denote that the span morphism $\bar{m}$ [the span s] satisfies pc. Two program conditions pc_1 and pc_2 are *equivalent*, denoted by $\mathrm{pc}_1 \equiv \mathrm{pc}_2$, if $[\![\mathrm{pc}_1]\!] = [\![\mathrm{pc}_2]\!]$ where, for a program condition pc, $[\![\mathrm{pc}]\!]$ denotes the

set $\{\bar{m} \mid \bar{m} \models \mathrm{pc}\}$. A program P *satisfies* the program condition pc, denoted by $P \models \mathrm{pc}$, if for all spans $s \in [\![P]\!]$ in the semantics of P, $s \models \mathrm{pc}$.

Remark 3. We sometimes use a short notation for program conditions: For a morphism $\bar{a}\colon p \to p'$ in a program condition, we just depict p', if p can be unambiguously inferred, i.e. for program conditions over the initial span $\langle I \hookleftarrow I \hookrightarrow I \rangle$.

Remark 4. Basic program conditions may be seen as ordinary rules with a left and a right application condition. Nested program conditions may be seen as nested rules in the sense of Rensink [15]. The main difference between double-pullback transitions and basic program condition satisfaction is informally that a span $\langle G \hookleftarrow D \hookrightarrow H \rangle$ satisfies the condition $\langle L \hookleftarrow K \hookrightarrow R \rangle$ if, for every instance of L in G, there corresponding instance of R in H. However, in the case of double-pullback transitions, $\langle G \hookleftarrow D \hookrightarrow H \rangle$ is in the semantics of $\langle L \hookleftarrow K \hookrightarrow R \rangle$ if, for a given instance of L in G, there is a corresponding instance of R in H. In a sense, program conditions are universal, while double-pullback transitions are existential conditions.

Fact 1. Program conditions generalize conditions in the sense of [1,2,3,4]: Every pair $\langle \mathrm{pre}, \mathrm{post} \rangle$ of conditions over the $\mathcal{M}$-initial object I constitutes a basic program condition $\langle i, \mathrm{pre}, \mathrm{post} \rangle$ with $i = \langle I \hookleftarrow I \hookrightarrow I \rangle$.

In the following, we present some examples for program conditions.

Example 3. The program condition with empty span $\langle \emptyset \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle$, precondition $\texttt{simple} = \neg\exists(\bigcirc_1 \rightleftarrows \bigcirc_2) \wedge \neg\exists(\bigcirc_1 \circlearrowright)$, and postcondition $\texttt{complete} = \forall(\bigcirc_1\ \bigcirc_2, \exists(\bigcirc_1 \to \bigcirc_2))$ has the meaning "For simple graphs, the resulting graph has to be complete". (A graph is *simple* if it contains no parallel edges or loops and *complete* if every pair (v_1, v_2) of distinct nodes is connected by an edge from v_1 to v_2.) The program condition $\langle \bigcirc_1 \to \bigcirc_2 \hookleftarrow \bigcirc_1\ \bigcirc_2 \hookrightarrow \bigcirc_1 \leftarrow \bigcirc_2 \rangle$ has the meaning "Replace all edges with edges in opposite direction". Table 1 shows some pre- and postconditions and basic program conditions with their interpretation. The short notation $\bigcirc_1 \not\to \bigcirc_2$ denotes the graph $\bigcirc_1\ \bigcirc_2$ together with the application condition $\neg\exists(\bigcirc_1\ \bigcirc_2 \to \bigcirc_1 \to \bigcirc_2)$, meaning "There does not exist an edge from the image of 1 to the image of 2.

The definition of program conditions combines pre- and postconditions with the concept of nesting. Pre- and post condition are well-known, well-understood, and natural; nesting is complicated and powerful. Every basic program condition can be transformed into an equivalent nested program condition *without pre- and postcondition*, i.e. in a program condition with subcondition of the form $\langle p, \mathrm{true}, \mathrm{true} \rangle$.

Theorem 1. *For every (basic) program condition, there is an equivalent (nested) program condition without pre- and postcondition.*

Proof. We define a transformation Nf from program conditions into an equivalent program conditions without pre- and postcondition. If we have a transformation

Table 1. Conditions and basic program conditions

condition	interpretation as pre/post condition
	All old/new nodes have a loop.
	No old/new node has a loop.
	Previously/afterwards, all pairs of distinct nodes are connected by at most one edge.
	Previously/afterwards, all nodes are loop-free.
program condition	**interpretation**
	All nodes are preserved.
	All nodes are deleted.
	At least one node is deleted.
	At least one node is preserved.
	All proper edges are preserved.
	All proper edges are deleted and no new ones are created.
	Whenever there were no proper edge, an edge is inserted.
	Proper parallel edges are deleted.
	All loops are deleted.

for basic program conditions, then the transformation can be extended to arbitrary program conditions by replacing basic subprogram conditions by equivalent ones without pre- and postcondition. For basic program conditions of the form $\langle p, \text{pre}, \text{post}\rangle$, we may use the fact that, by Definition 4, $\langle p, \text{pre}, \text{post}\rangle \equiv \langle p, \text{pre}, \text{true}\rangle \wedge \langle p, \text{true}, \text{post}\rangle$. For basic program conditions with true precondition, we may use the fact that $\langle p, \text{true}, \text{post}\rangle \equiv \langle p^{-1}, \text{post}, \text{true}\rangle^{-1}$, where, for a basic program condition $\text{pc} = \langle p, \text{pre}, \text{post}\rangle$ with $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$, pc^{-1} denotes the inverse basic program condition $\langle p^{-1}, \text{post}, \text{pre}\rangle$ with $p^{-1} = \langle R \hookleftarrow K \hookrightarrow L\rangle$. Without loss of generality, we can assume that pre and post are $\mathcal{M}$-conditions, i.e. for all subconditions of the form $\exists(a, c)$, the morphism a is in $\mathcal{M}$ (see [4]).

Construction. For basic program conditions of the form $\langle p, \text{pre}, \text{true}\rangle$, the transformation Nf is defined by induction on the structure of the condition pre:

$$\begin{aligned} \text{Nf}(\langle p, \text{true}, \text{true}\rangle) &= \langle p, \text{true}, \text{true}\rangle \\ \text{Nf}(\langle p, \exists(a, c), \text{true}\rangle) &= \bigvee_{\bar{a}\in A} \exists(\bar{a}, \text{Nf}(\langle p', c, \text{true}\rangle)) \end{aligned}$$

where $\bar{a} \in A$ ranges over all $\mathcal{M}$-morphisms $\bar{a} = \langle a, z, a^*\rangle$ such that (11) and (21) are pullbacks and $p' = \langle L' \leftarrow K' \rightarrow R'\rangle$ is the codomain of $\bar{a}$. For Boolean formulas over conditions, the transformation is straightforward: $\text{Nf}(\langle p, \neg c, \text{true}\rangle) = \neg(\text{Nf}(\langle p, c, \text{true}\rangle))$ and $\text{Nf}(\langle p, c_1 \overset{\wedge}{\vee} c_2, \text{true}\rangle) = \text{Nf}(\langle p, c_1, \text{true}\rangle) \overset{\wedge}{\vee} \text{Nf}(\langle p, c_2, \text{true}\rangle)$.

We prove $\text{Nf}(\langle p, \text{pre}, \text{true}\rangle) \equiv \langle p, \text{pre}, \text{true}\rangle$ by induction on the structure of the condition pre. **Basis.** Let pre = true. By definition of Nf, $\text{Nf}(\langle p, \text{true}, \text{true}\rangle) =$

$\langle p, \text{true}, \text{true}\rangle$. **Induction.** Assume the hypothesis is true for conditions c, c_1, and c_2. Let pre $= \exists(a, c)$. For morphisms $\bar{m} = \langle m, d, m^*\rangle$, we have

$$
\begin{array}{lll}
& \bar{m} \models \langle p, \exists(a,c), \text{true}\rangle & \\
\Leftrightarrow & \text{(1), (2) PB's} \wedge m \models \exists(a,c) & \text{(Definition 4)} \\
\Leftrightarrow & \text{(1), (2) PB's} \wedge \exists\, l \in \mathcal{M}.\ m = l \circ a \wedge l \models c & \text{(Def. of } \models) \\
\Leftrightarrow & \bar{m} \models p \wedge \exists \bar{a} \in A.\ \exists\, \bar{l} \in \mathcal{M}^3.\ \bar{m} = \bar{l} \circ \bar{a} \wedge \bar{l} \models \langle p', c, \text{true}\rangle & (*) \\
\Leftrightarrow & \bar{m} \models p \wedge \exists \bar{a} \in A.\ \exists\, \bar{l} \in \mathcal{M}^3.\ \bar{m} = \bar{l} \circ \bar{a} \wedge \bar{l} \models \text{Nf}(\langle p', c, \text{true}\rangle) & \text{(Ind hyp)} \\
\Leftrightarrow & \bar{m} \models \bigvee_{\bar{a} \in A} \exists(\bar{a}, \text{Nf}(\langle p', c, \text{true}\rangle)) & \text{(Def. 4)} \\
\Leftrightarrow & \bar{m} \models \text{Nf}(\langle p, \exists(a,c), \text{true}\rangle) & \text{(Def. of Nf)}
\end{array}
$$

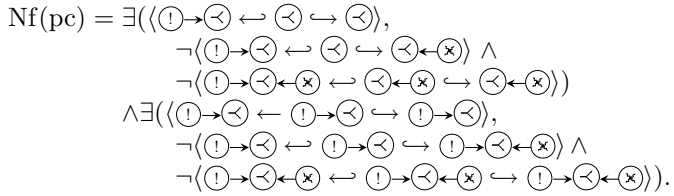
The equivalence (*) may be seen as follows: "⇒": Let (1) and (2) be pullbacks, $\mathrm{l} \in \mathcal{M}$, $m = l \circ a$, and $l \models c$. Then the decomposition of m induces a decomposition of the pullbacks (1) and (2) into pullbacks (11), (12) and (21), (22) as follows: Construct $L' \hookleftarrow K' \hookrightarrow D$ as a pullback of $L' \to G \hookleftarrow D$ (12). Then there is a unique morphism $K \to K'$ such that diagram (11) below commutes and $K \to K' \hookrightarrow D = K \hookrightarrow D$. Since $\mathcal{M}$ is closed under decompositions, the morphism $K \to K'$ is in $\mathcal{M}$. By the pullback-decomposition property, (11) is a pullback. Construct $K' \hookrightarrow R' \hookleftarrow R$ as a pushout of $K' \hookleftarrow K \hookrightarrow R$ (21). Then there is a unique morphism $R' \to H$ such that (22) commutes and $R \hookrightarrow R' \to H = R \hookrightarrow H$. By the special pullback-decomposition property, (21) and (22) are pullbacks. Since $\mathcal{M}$-morphisms are closed under decompositions, the morphism $R' \to H$ is in $\mathcal{M}$. Finally, let $p' = \langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ and $\bar{a}$ and $\bar{l}$ be the triples of morphisms in the construction. Then $\bar{m} = \bar{l} \circ \bar{a}$. Now $l \models c$ implies $\bar{l} \models \langle p', c, \text{true}\rangle$. "⇐": Let $\bar{m} \models p$, $\bar{m} = \bar{l} \circ \bar{a}$, $\bar{l} \models \langle p', c, \text{true}\rangle$, Then the diagrams (1) and (2) are pullbacks, $m = l \circ a$, where m and l are the projections of $\bar{m}$ and $\bar{l}$ to the first component, and $l \models c$.

$$
\begin{array}{ccccc}
L & \hookleftarrow & K & \hookrightarrow & R \\
{\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\
G & \hookleftarrow & D & \hookrightarrow & H
\end{array}
\qquad\qquad
\begin{array}{ccccc}
L & \hookleftarrow & K & \hookrightarrow & R \\
{\scriptstyle a}\downarrow & (11) & \downarrow & (21) & \downarrow \\
L' & \hookleftarrow & K' & \hookrightarrow & R' \\
{\scriptstyle l}\downarrow & (12) & \downarrow & (22) & \downarrow \\
G & \hookleftarrow & D & \hookrightarrow & H
\end{array}
\quad (m : L \to G)
$$

Let pre be a Boolean formula over conditions. By the definition of Nf and the inductive hypothesis, $\text{Nf}(\langle p, \neg c, \text{true}\rangle) = \neg\text{Nf}(\langle p, c, \text{true}\rangle) \equiv \neg\langle p, c, \text{true}\rangle \equiv \langle p, \neg c, \text{true}\rangle$ and $\text{Nf}(\langle p, c_1 \bigwedge_{\bigvee} c_2, \text{true}\rangle) = \text{Nf}(\langle p, c_1, \text{true}\rangle \bigwedge_{\bigvee} \text{Nf}(\langle p, c_2, \text{true}\rangle) \equiv \langle p, c_1, \text{true}\rangle \bigwedge_{\bigvee} \langle p, c_2, \text{true}\rangle \equiv \langle p, c_1 \bigwedge_{\bigvee} c_2, \text{true}\rangle$. This completes the inductive proof.

Fact 2. The converse of Theorem 1 does not hold, e.g. see the program condition in Example 4.

Example 4. The property "Every router that were on the priority list has been repaired" can be expressed by the basic program condition pc with span $\langle \textcircled{\prec} \hookleftarrow \textcircled{\prec} \hookrightarrow \textcircled{\prec}\rangle$, precondition $\exists(\textcircled{!} \to \textcircled{\prec})$, and postcondition $\neg\exists(\textcircled{\prec} \leftarrow \textcircled{\times})$. The program condition can be transformed into a nested program condition Nf(pc), where

$\mathrm{Nf}(\mathrm{pc}) =$ 

As illustrated by Example 4, nested program conditions can have a quite verbose form. Therefore, we introduce a shorter visual notation for program conditions.

Notation (graphical notation for program conditions). In the case of graphs, there is a nice short notation for program conditions, also used for representing rules in [16] and somewhat similar to the X and Y notations of [17]. The basic idea is that a graph span is represented as a single colored graph. The colored graph of the span $\langle L \hookleftarrow K \hookrightarrow R \rangle$ is structurally isomorphic to the pushout of $L \hookleftarrow K \hookrightarrow R$. Additionally, elements in the colored graphs are visually represented with dashed lines (deleted) if they are in $L - K$, solid lines (preserved) if in K, and with bold lines (created) if in $R - K$. Span morphisms are then visualized as morphisms on colored graphs, yielding a visualization of a graph program condition that is quite similar to the (standard) graph condition. A graph span with two deleted, one preserved, and two created elements, together with the program condition for "Every router that were on the priority list has been repaired" is shown in Fig. 2 using the visual notation.

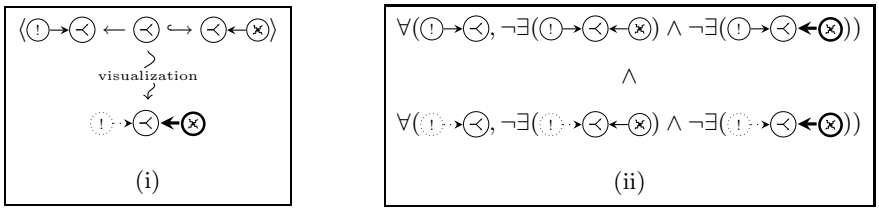


Fig. 2. Visual representation of (i) a relative graph and (ii) a graph program condition

4 Program Construction

We now investigate program construction. For a special kind of program conditions, a program can be constructed such that all spans in its semantics satisfy the program condition (and there exists a span). Based on sequential independence of direct derivations for rules with application conditions [5] and termination of rewrite systems [18], sequential self-independence and termination of basic program conditions are introduced.

Definition 5 (always applicable, self-independent, terminating). A basic program condition $\mathrm{pc} = \langle p, \mathrm{pre}, \mathrm{post} \rangle$ is *always applicable* if, for every $\mathcal{M}$-morphism $m\colon L \hookrightarrow G$ satisfying pre, there exists a direct derivation $G \Rightarrow_{\mathrm{pc},m,m^*}$

H such that m^* satisfies post. A basic program condition pc is *sequentially self-independent* if all derivation sequences $G \Rightarrow_{\mathrm{pc},m_1} H_1 \Rightarrow_{\mathrm{pc},m_2} M$ are sequentially independent and *terminating* if there is no infinite derivation sequence of the form $G_1 \Rightarrow_{\mathrm{pc}} G_2 \Rightarrow_{\mathrm{pc}} G_3 \ldots$.

Example 5. The program condition $\langle \bigcirc \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle$, meaning "Every node shall be deleted", is not always applicable; it is applicable if and only if the dangling condition [8] is satisfied. The program condition $\langle \bigcirc_1 \to \bigcirc_2 \hookleftarrow \bigcirc_1 \; \bigcirc_2 \hookrightarrow \bigcirc_1 \leftarrow \bigcirc_2 \rangle$ is not sequentially self-independent, but the modified program condition $\langle \bigcirc_1 \to \bigcirc_2 \hookleftarrow \bigcirc_1 \; \bigcirc_2 \hookrightarrow \bigcirc_1 \overset{*}{\leftarrow} \bigcirc_2 \rangle$ is. The (empty) program condition $\langle \emptyset \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle$ is always applicable, sequentially self-independent, but not terminating. The program condition $\langle \bigcirc_1 \not\to \bigcirc_2 \hookleftarrow \bigcirc_1 \; \bigcirc_2 \hookrightarrow \bigcirc_1 \to \bigcirc_2 \rangle$ is always applicable, sequentially self-independent, and terminating: the application condition "There does not exist an edge" guarantees termination because, for every finite graph with n nodes, at most 2^{n-1} proper edges can be inserted.

For basic program conditions, which are always applicable, sequentially self-independent, and terminating, a satisfying program can be constructed.

Theorem 2 (program construction). *For always applicable, sequentially self-independent, and terminating basic program conditions* pc*, the program* $\mathrm{pc}\downarrow$ *satisfies* pc*.*

Proof. By the Parallelism Theorem for rules with application conditions [5]. Let $\mathrm{pc} = \langle p, \mathrm{pre}, \mathrm{post} \rangle$ be always applicable, sequentially self-independent, and terminating. Let $s = \langle G \hookleftarrow D \hookrightarrow H \rangle \in [\![\mathrm{pc}\downarrow]\!]$ and $m\colon L \hookrightarrow G$ be an $\mathcal{M}$-morphism such that $m \models \mathrm{pre}$. By the always applicability and termination, there is a derivation $G \Rightarrow_{\mathrm{pc},m} G_1 \Rightarrow_{\mathrm{pc}} \ldots \Rightarrow_{\mathrm{pc}} G_n = H$ such that there is no M with $\langle H \hookleftarrow E \hookrightarrow M \rangle \in [\![\mathrm{pc}]\!]$. By sequential self-independence and the Parallelism Theorem, there is a parallel derivation $G \Rightarrow H$ through the parallel rule $k\mathrm{pc}$ with $kp = \langle kL \hookleftarrow kK \hookrightarrow kR \rangle$ where, for $k \geq 0$, kA denotes the k-fold disjoint union of A. Then the diagrams (k1) and (k2) are pushouts. Since pushouts along $\mathcal{M}$-morphisms are pullbacks, (k1) and (k2) are also pullbacks.

$$\begin{array}{ccccc} kL & \hookleftarrow & kK & \hookrightarrow & kR \\ \downarrow & \text{(k1)} & \downarrow & \text{(k2)} & \downarrow \\ G & \hookleftarrow & D & \hookrightarrow & H \end{array} \qquad \begin{array}{ccccccc} & & L & \hookleftarrow & K & \hookrightarrow & R \\ & & \downarrow & (1) & \downarrow & (2) & \downarrow \\ m & \curvearrowright & kL & \hookleftarrow & kK & \hookrightarrow & kR \\ & & \downarrow & \text{(k1)} & \downarrow & \text{(k2)} & \downarrow \\ & & G & \hookleftarrow & D & \hookrightarrow & H \end{array}$$

By the pullback decomposition property, (1) and (2) are pullbacks and, by the pullback composition property, the diagrams (1)+(k1) and (2)+(k2) are pullbacks. Thus, $m \models \mathrm{pc}$, Consequently $s \models \mathrm{pc}$. Therefore, $\mathrm{pc}\downarrow \models \mathrm{pc}$.

Example 6. In Table 2 we show a number of program conditions and their constructed programs that follow from Theorem 2.

Table 2. Program conditions and satisfying programs

program condition	program
`MakeLoopFree` $= \langle$ $\rangle$	`MakeLoopFree`$\downarrow$
`DeleteEdge` $= \langle$ $\rangle$	`DeleteEdge`$\downarrow$
`Complete` $= \langle$ $\rangle$	`Complete`$\downarrow$
`Converse`* $= \langle$ $\rangle$	`Converse`$^*\downarrow$
`Subdivide`* $= \langle$ $\rangle$	`Subdivide`$^*\downarrow$
`Relabel` $= \langle$ $\rangle$	`Relabel`$\downarrow$

Given satisfying program conditions for programs, a satisfying program condition for a set of programs can be constructed considering the disjunction of the original program conditions.

Fact 3. For programs P_1, P_2 and program conditions $\mathrm{pc}_1, \mathrm{pc}_2$

$$P_i \models \mathrm{pc}_i \ (i = 1, 2) \ \text{ implies } \ \{P_1, P_2\} \models \mathrm{pc}_1 \vee \mathrm{pc}_2.$$

Proof. Let $P_i \models \mathrm{pc}_i$ $(i = 1, 2)$. By the semantics of programs, the assumptions, and the definition of $\models$, $s \in [\![\{P_1, P_2\}]\!] = [\![P_1]\!] \cup [\![P_2]\!] \Leftrightarrow s \in [\![P_1]\!] \vee s \in [\![P_2]\!] \Leftrightarrow s \models \mathrm{pc}_1 \vee s \models \mathrm{pc}_2 \Leftrightarrow s \models \mathrm{pc}_1 \vee \mathrm{pc}_2$. Thus, for all $s \in [\![\{P_1, P_2\}]\!]$, $s \models \mathrm{pc}_1 \vee \mathrm{pc}_2$, i.e. $\{P_1, P_2\} \models \mathrm{pc}_1 \vee \mathrm{pc}_2$.

Given satisfying basic program conditions for programs, the concurrent program condition is a satisfying program condition for the sequence of programs.

Theorem 3 (composition of basic program conditions). *For programs P_1, P_2 and basic program conditions $\mathrm{pc}_1, \mathrm{pc}_2$ with dependence relation d for pc_1 and pc_2,*

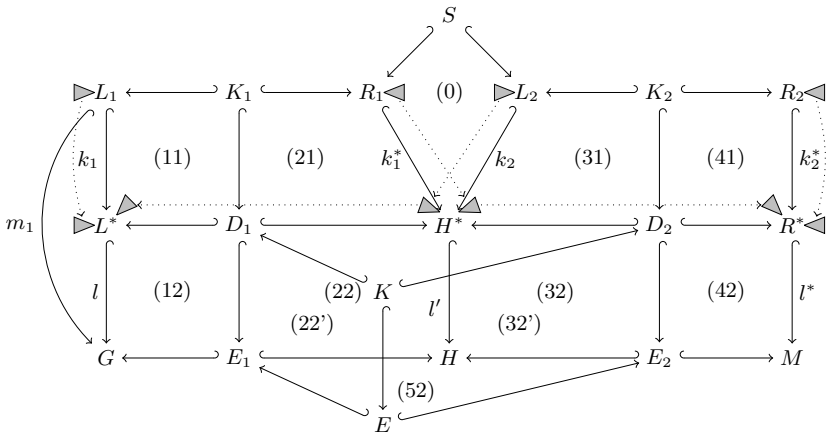
$$P_i \models \mathrm{pc}_i \ (i = 1, 2) \ \textit{ implies } \ P_1; P_2 \models \mathrm{pc}_1 *_d \mathrm{pc}_2,$$

where $\mathrm{pc}_1 *_d \mathrm{pc}_2$ is the d-concurrent program condition of pc_1 and pc_2 [5].

Proof. By the Concurrency Theorem for rules with application conditions [5]. Let $\mathrm{pc}_i = \langle p_i, \mathrm{pre}_i, \mathrm{post}_i \rangle$ with $p_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ $(i = 1, 2)$ be basic program conditions and $\mathrm{pc}_1 *_d \mathrm{pc}_2 = \langle p, \mathrm{pre}, \mathrm{post} \rangle$ with $p = \langle L^* \hookleftarrow K^* \hookrightarrow R^* \rangle$, $\mathrm{pre} = \mathrm{Shift}(k_1, \mathrm{pre}_1) \wedge \mathrm{L}(\mathrm{pc}_1^*, \mathrm{Shift}(k_2, \mathrm{pre}_2))$, and $\mathrm{post} = \mathrm{R}(\mathrm{pc}_2^*, \mathrm{Shift}(k_1^*, \mathrm{post}_1)) \wedge \mathrm{Shift}(k_2^*, \mathrm{post}_1)$ the d-concurrent program condition of pc_1 and pc_2, where Shift and L, and R are the shiftings of application conditions over morphisms and rules, from the right-hand side to the left-hand side, and from left-hand side to right-hand side, respectively [4]. Let $P_i \models \mathrm{pc}_i$ $(i = 1, 2)$. We will show that $P_1; P_2 \models \mathrm{pc}_1 *_d \mathrm{pc}_2$. Let $s = \langle G \hookleftarrow E \hookrightarrow M \rangle \in [\![P_1; P_2]\!]$ be an arbitrary span. By Definition 3, there are spans $\langle G \hookleftarrow E_1 \hookrightarrow H \rangle \in [\![P_1]\!]$ and $\langle H \hookleftarrow E_2 \hookrightarrow M \rangle \in [\![P_2]\!]$ such that $E_1 \hookleftarrow E \hookrightarrow E_2$ is the pullback of $E_1 \hookrightarrow H \hookleftarrow E_2$.

Let $l\colon L^* \hookrightarrow G$ be an $\mathcal{M}$-morphism satisfying pre. By the properties of Shift, the composed $\mathcal{M}$-morphism $m_1 = l \circ k_1$ satisfies pre_1. By assumption, we know that $P_1 \models \mathrm{pc}_1$, i.e., there are $\mathcal{M}$-morphisms $d_1\colon K_1 \to E_1$ and $m_1^*\colon R_1 \to H$ such

that the diagrams (1), (2) are pullbacks and m_1^* satisfies post_1. By the pullback decomposition property, the pullbacks (1) and (2) can be decomposed into pullbacks (11), (12), (21), (22). Now we have an $\mathcal{M}$-morphism $l'\colon H^* \to H$ such that $m_1^* = l' \circ k_1^*$. By the properties of Shift and L, the composed $\mathcal{M}$-morphism $m_2 = l' \circ k_2$ satisfies pre_2. By $P_2 \models \text{pc}_2$, there are $\mathcal{M}$-morphisms $d_2\colon K_2 \to E_2$ and $m_2^*\colon R_2 \to M$ such that the diagrams (3), (4) are pullbacks and m_2^* satisfies post_2. By the pullback decomposition property, there is a decomposition of the pullbacks (3) and (4) into pullbacks (31), (32), (41), (42). Now we have an $\mathcal{M}$-morphism $l^*\colon H^* \to M$ such that $m_2^* = l^* \circ k_2^*$. Since the class of $\mathcal{M}$-morphisms is closed under decomposition, the morphism l^* is in M. Constructing E as the pullback object of $E_1 \hookrightarrow H \hookleftarrow E_2$, be the universal property of pushouts, there is a morphism $K \to E$ such that the diagrams (22') and (32') commute. By the (cube) pullback decomposition property, (22') and (32') are pullbacks. By the composition property of pullbacks, the diagrams (22')+(12) and (32')+(42) are pullbacks. Since $\mathcal{M}$-morphisms are closed under pullbacks, the morphisms $K \to E$ is in $\mathcal{M}$. By the properties of Shift and R, the $\mathcal{M}$-morphism $l^*\colon R^* \to M$ satisfies post. As a consequence, $l \models \text{pc}_1 *_d \text{pc}_2$. For every span $s \in [\![P_1; P_2]\!]$, $s \models \text{pc}_1 *_d \text{pc}_2$. Consequently, $P_1; P_2 \models \text{pc}_1 *_d \text{pc}_2$.



Example 7. The program condition $\texttt{Converse} = \langle \bullet_1 \to \bullet_2 \hookleftarrow \bullet_1\ \bullet_2 \hookrightarrow \bullet_1 \leftarrow \bullet_2 \rangle$ is always applicable, but not sequentially self-independent; the program $\texttt{Converse}\downarrow$ is non-terminating. For avoiding sequentially self-dependence, the program condition could be modified into a sequentially self-independent program condition $\texttt{Converse}' = \langle \bullet_1 \to \bullet_2 \hookleftarrow \bullet_1\ \bullet_2 \hookrightarrow \bullet_1 \overset{*}{\leftarrow} \bullet_2 \rangle$. For correcting the modification, a sequentially self-independent program condition $\texttt{Relabel} = \langle \bullet_1 \overset{*}{\to} \bullet_2 \hookleftarrow \bullet_1\ \bullet_2 \hookrightarrow \bullet_1 \to \bullet_2 \rangle$ may be considered. By Theorem 3, the program $\texttt{Converse}'\downarrow\ ;\texttt{Relabel}\downarrow$ satisfies the program condition $\texttt{Converse}' *_d \texttt{Relabel} = \texttt{Converse}$ (where d is the "complete" dependence relation $\langle \bullet_1 \overset{*}{\to} \bullet_2 \hookleftarrow \bullet_1 \overset{*}{\to} \bullet_2 \hookrightarrow \bullet_1 \overset{*}{\to} \bullet_2 \rangle$).

Parallel self-independence and parallel independence is defined analogously to sequential self-independence and sequential independence of program conditions, see [5] for details. E.g., the program condition `Converse` is parallelly

self-independent; a program `Converse` $\Downarrow$ based on a *parallel* composition of programs would satisfy `Converse`. Generally, it could be important to introduce a parallel composition of programs. and to show how a parallel composition can be expressed by the sequential composition. In this case, one could use the parallel composition as an abbreviation of a more complex sequential operation.

5 Conclusion

In this paper, we introduced the syntax and semantics of high-level program conditions. The syntax of program conditions is well-known: Basic program conditions syntactically may be seen as ordinary rules with left and right application conditions; nested program conditions may be seen as nested rules in the sense of Rensink [15] but with a categorical definition. Their semantic differences were discussed.

- **Normal form.** We showed a normal form result, relating basic program conditions with nested program conditions.
- **Program construction.** For special types of program conditions, we could create satisfying programs.
- **Sequential composition.** For basic program conditions, a composition is defined by constructing the concurrent program condition according to a dependence relation.

Further topics could be the following.

- **Expressiveness of program conditions.** As known by [4], graph conditions are expressively equivalent to first-order formulas on graphs. How powerful are graph program conditions?
- **Concurrency Theorem for nested rules.** It would be important to generalize the Concurrency Theorem to nested rules such that Theorem 3 can be formulated for program conditions instead of basic program conditions.
- **Comparison with first-order process logic.** In [19], the authors extend dynamic logic with the additional trace modalities *throughout* and *at least once*, which refer to all the states a program reaches and allow one to specify and verify invariants and safety constraints that have to be valid throughout the execution of a program.

References

1. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars – a constructive approach. In: SEGRAGRA 1995. ENTCS, vol. 2, pp. 95–104 (1995)
2. Koch, M., Mancini, L.V., Parisi-Presicce, F.: Graph-based specification of access control policies. Journal of Computer and System Sciences 71, 1–33 (2005)
3. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae 74(1), 135–166 (2006)

4. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. In: Mathematical Structures in Computer Science (to appear, 2008)
5. Azab, K., Habel, A.: High-level programs and program conditions (long version). Technical report, University of Oldenburg (2008)
6. Ehrig, H., Kreowski, H.J.: Pushout-properties: An analysis of gluing constructions for graphs. Mathematische Nachrichten 91, 135–149 (1979)
7. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer, Heidelberg (2006)
8. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Ng, E.W., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979)
9. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach. In: Handbook of Graph Grammars and Computing by Graph Transformation, pp. 163–245. World Scientific, Singapore (1997)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory of typed attributed graph transformation based on adhesive HLR-categories. Fundamenta Informaticae 74(1), 31–61 (2006)
11. Lambers, L., Ehrig, H., Prange, U., Orejas, F.: Parallelism and concurrency in adhesive high-level replacement systems with negative application conditions. In: Workshop on Applied and Computational Category Theory (ACCAT 2007). ENTCS. Elsevier, Amsterdam (to appear, 2008)
12. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 230–245. Springer, Heidelberg (2001)
13. Plump, D.: Confluence of graph transformation revisited. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) Processes, Terms and Cycles: Steps on the Road to Infinity. LNCS, vol. 3838, pp. 280–308. Springer, Heidelberg (2005)
14. Heckel, R., Ehrig, H., Wolter, U., Corradini, A.: Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. Applied Categorical Structures 9(1), 83–110 (2001)
15. Rensink, A.: Nested quantification in graph transformation rules. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 1–13. Springer, Heidelberg (2006)
16. Lambers, L.: A new version of GTXL: An exchange format for graph transformation systems. In: Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004). ENTCS, vol. 127, pp. 51–63 (2005)
17. Göttler, H.: Deriving productions from productions with an application to picasso's ceuvre. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 459–484. World Scientific, Singapore (1999)
18. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
19. Beckert, B., Schlager, S.: A sequent calculus for first-order dynamic logic with trace modalities. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 626–641. Springer, Heidelberg (2001)

A Shifting of Conditions

In the following, we review some improtant statements on the shifting of conditions over morphisms and rules investigated in detail in [4].

Fact 4 (shifting of conditions over morphisms [4]). There is a transformation Shift such that, for all conditions c over P, all morphisms $m\colon P \to P'$, and all $n\colon P' \hookrightarrow H$ in $\mathcal{M}$, $n \models \mathrm{Shift}(m, c) \Leftrightarrow n \circ m \models c$.

Right application conditions of a rule can be transformed into equivalent left application conditions.

Fact 5 (shifting of application conditions over rules [4]). There are transformations L and R such that, for every right application condition ac_R and every left application condition ac_L of a rule ρ and every direct derivation $G \Rightarrow_{\rho,m,m^*} H$, $m \models \mathrm{L}(\rho, \mathrm{ac}_R) \Leftrightarrow m^* \models \mathrm{ac}_R$ and $m \models \mathrm{ac}_L \Leftrightarrow m^* \models \mathrm{R}(\rho, \mathrm{ac}_L)$.

B Concurrent Rules

In the following we present the construction of a concurrent rule for rules with application conditions. It generalizes the well-known construction of concurrent rules for rules without application conditions [10] and makes use of shifting of application conditions over morphisms and rules (see Facts 4 and 5).

Definition 6 (d-concurrent rule). Let $\rho_i = \langle p_i, \mathrm{ac}_{Li}, \mathrm{ac}_{Ri}\rangle$ with $p_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i\rangle$ for $i = 1, 2$ be rules. A pair $d = \langle S \hookleftarrow R_1 \hookrightarrow L_2\rangle$ is a *dependence relation* for rules ρ_1 and ρ_2, if (0) is the pushout of $R_2 \hookleftarrow S \hookrightarrow L_2$ and the pushout complements (1) and (2) of $K_1 \hookrightarrow R_1 \hookrightarrow H^*$ and $K_2 \hookrightarrow L_2 \hookrightarrow H^*$ exist. Given a dependence relation d for ρ_1 and ρ_2, the *d-concurrent rule* of ρ_1 and ρ_2 is the rule $\rho_1 *_d \rho_2 = \langle p, \mathrm{ac}_L, \mathrm{ac}_R\rangle$ with $p = \langle L^* \hookleftarrow K^* \hookrightarrow R^*\rangle$, where (3) and (4) are pushouts and (5) is a pullback, $\mathrm{ac}_L = \mathrm{Shift}(k_1, \mathrm{ac}_{L1}) \wedge \mathrm{L}(p_1^*, \mathrm{Shift}(k_2, \mathrm{ac}_{L2}))$, and $\mathrm{ac}_R = \mathrm{R}(p_2^*, \mathrm{Shift}(k_1^*, \mathrm{ac}_{R1})) \wedge \mathrm{Shift}(k_2^*, \mathrm{ac}_{R2})$, where $p_1^* = \langle L^* \hookleftarrow D_1 \hookrightarrow H^*\rangle$ and $p_2^* = \langle H^* \hookleftarrow D_2 \hookrightarrow R^*\rangle$ are the rules derived by p_1 and k_1 and p_2 and k_2, respectively.

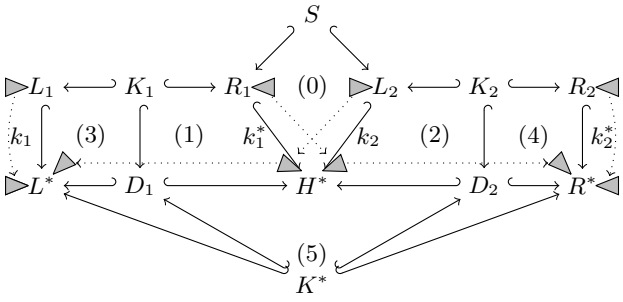


Fig. 3. Construction of a d-concurrent rule $\rho_1 *_d \rho_2$