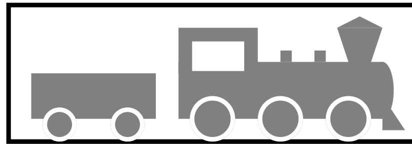
Generalized Constraints and Application Conditions for Graph Transformation Systems

Karl-Heinz Pennemann

Carl v. Ossietzky University of Oldenburg, Germany
k.h.pennemann@informatik.uni-oldenburg.de

September 17, 2004



Abstract

Constraints and application conditions are very important for graph transformation systems in a large variety of application areas. Although different approaches have been presented in the literature, their expressiveness is limited, usually because of restrictions on the maximum depth of nested quantifiers. In this thesis, a generalized definition of constraints is presented, their transformation into right and left application conditions is considered and the correctness of these transformations is formally proved.

Contents

1 Introduction **2**
1.1 History of graph transformation 2
1.2 History of constraints and application conditions 2
1.3 A railroad control system 3
1.4 Topics of this thesis 4
1.5 Structure of this thesis 5

2 Graph transformation systems **6**

3 Constraints and application conditions **8**
3.1 Constraints and application conditions 8
3.2 Equivalences and implications 12
3.3 Normal form for constraints and application conditions 15

4 Transformation of constraints **19**
4.1 Transformation into right application conditions 19
4.2 Example 23
4.3 Complexity of the transformation 25
4.4 Elimination of constraints and application conditions 27

5 Transformation of right application conditions **30**
5.1 Transformation into left application conditions 30
5.2 Example 32
5.3 Complexity of the transformation 33

6 Applications of the transformations **35**

7 Conclusion **39**

A Appendix: Equivalences **45**

1 Introduction

Constraints and application conditions are most important for transformation systems [1, 29] in a large variety of application areas, especially in the area of safety-critical systems e.g. the specification of railroad control systems and access control policies [22, 20, 23, 21]. Application conditions for transformation rules describe classes of morphisms. They restrict the sets of all possible matches of their rules to the sets of morphisms that satisfy them and are in this way used to restrict the applicability of their rules. As a result, application conditions allow not only greater flexibility in formalizing possible transitions, but also increase the expressive power of individual rules, e.g. context-free rules [15, 32].

Constraints, also known as graphical consistency constraints, describe properties of objects and thus define classes of objects. Constraints can be used to specify valid states of a transformation system.

Together, constraints and application conditions can ensure that applications of a rule yield graphs that have certain properties. In this paper it is shown that there exists a transformation of constraints into left application conditions for any given rule, such that the rule with this condition is applicable, if and only if the outcome satisfies the constraint. This allows a separation between the specification of a system in terms of simple transformation rules and constraints, and its implementation in terms of combining the rules with automatically generated application conditions.

1.1 History of graph transformation

The history of graph transformation or graph replacement started in the late 1960s with two papers about so-called "Web grammars" [27] and "Chomsky systems for partial orders" [30]. Three years later, in the early seventies, the so-called "algebraic approach" to graph grammars had been invented by H. Ehrig, M. Pfender and H.J. Schneider [11]. The approach was called "algebraic" because graphs were considered as special kinds of algebras and the gluing of graphs was defined by an "algebraic construction", called *pushout*, in the category of graphs and total morphisms. A direct derivation step was modelled by two of these gluing diagrams, hence the name *double-pushout (DPO) approach*. While the DPO approach is not the only approach to graph transformation it is the most widely used. Informations about other approaches, further details on the DPO approach and pointers to the literature can be found in [29, 7, 10, 5, 4, 16].

1.2 History of constraints and application conditions

The concept of application conditions is older than that of constraints. Application conditions for transformation rules were investigated e.g. in [8, 15, 18, 22, 6], where the last three papers also consider constraints. Initially, the rules of a graph grammar were equipped by a very general notion of application conditions, for the first time in [8]. In a subsequent paper [15], the notion of application conditions was restricted to contextual conditions like the existence or non-existence of nodes and edges or certain subgraphs, such that they could be

represented in a graphical way. In [18], a simple form of "conditional" application conditions were considered. Furthermore the authors introduced graphical consistency constraints, also called graph constraints, which express very basic conditions on graphs, like the existence or uniqueness of certain nodes and edges. In a subsequent paper [22], graph constraints were used to describe policies for access control. In [6], the notions of graph constraints and application conditions were lifted to high-level objects and replacement systems [9]. It was shown that basic constraints could be transformed into simple nested right application conditions and that these right application conditions could be transformed into left application conditions. This thesis continues the work done in [6].

1.3 A railroad control system

In order to help illustrate the following notions and transformations, and to provide a concrete example along the way, we introduce a running example that will be used throughout this paper. This example is a railroad control system (similar to [24]).

Example 1. (railroad control system) The basic items of this system are waypoints, bi-directional tracks and trains. The static part is given by a labelled directed rail net graph: tracks are modelled by undirected (resp. a pair of directed) edges, and trains are modelled by edges. Source and target nodes of a train edge encode the train's position on the track. An example of a circular rail net graph with two trains is given in figure 1.

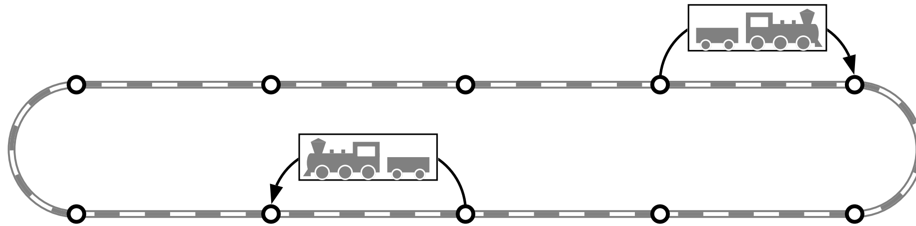


Figure 1: A simple railway model

The dynamic part of the system is specified by a graph transformation system. E.g., the movement of trains is modelled by the rule `Move` as depicted in figure 2.

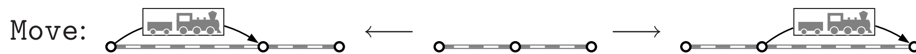


Figure 2: A rule for moving trains

Application of the rule `Move` means to find an occurrence of the left-hand side in the rail net graph and to replace it with the right-hand side of the rule. Effectively this means, if we find a piece of track before a train, we can move the train onto this piece. But the application of `Move` is not safe, yet. Trains

may crash, if a train moves onto a piece of track that is already occupied by another train. Thus we need to formalize, which behaviour we exclude.

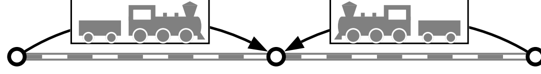
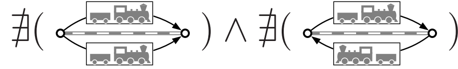


Figure 3: A dangerous situation

For security aspects, rail net constraints for rail net graphs have to be formalized. The control system should ensure that trains do not crash (no two trains occupy the same piece of track) and that trains keep a safety distance of one track (two trains do not occupy neighbouring pieces of track, except they face in opposite directions).

Different trains occupy different pieces of track:



Two adjacent trains face in opposite directions:

But how is this specification implemented? E.g., one could check the satisfaction of the above constraints every time `Move` is applied. However this check may be too late: If one of the constraints is violated, then the security of the system is already compromised. Consequently, we need to find a way to test before `Move` is applied, if the result will satisfy the constraints.

A solution to this problem would be, if we could transform our constraints into equivalent left application conditions for `Move`, such that the rule together with the conditions is applicable, if and only if the resulting rail net graph satisfies the constraints. Such an integration of constraints into rules would have two advantages: The satisfaction of the constraints would be guaranteed by the conditions and the conditions could be checked before any changes to the system are made. Intuitively, we seek application conditions for the rule `Move` to express that every train should move only on a free piece of track and, after movement, two trains should not occupy neighbouring pieces of track (except they have a different direction).

1.4 Topics of this thesis

In this thesis, the existing theory on constraints and application conditions [6] is extended to nested constraints and nested application conditions as proposed in [28]. The transformation of constraints into right application conditions and the transformation of right to left application conditions are generalized to nested constraints and nested application conditions, respectively. Furthermore, the complexity of these transformations are considered with respect to the size of the resulting application conditions.

Two applications of these basic transformations are considered: The conversion of constraints into left application conditions, such that the integration of the resulting application conditions into a given rule guarantees, or alternatively, preserves the satisfaction of the constraints. As the resulting application conditions for a constraint guarantee are rather large, it is sometimes preferable "just" to preserve the satisfaction of a constraint. In general, the corresponding application conditions are smaller, after all redundancies are eliminated.

Additionally, normal form results for nested constraints and application conditions are presented and an elimination theorem is provided that allows to eliminate superfluous subconstraints and subconditions, respectively.

The notations and concepts are illustrated by a simple railroad system. The specification of a railroad system is given in terms of rail net graphs, constraints, simple rules and application conditions. The rules model the dynamic behaviour of the system, like the movement of trains. The application conditions ensure the safety of the system. The integration of general rail net constraints into rail net application conditions is exemplarily studied for the movement of trains.

Parts of this paper will be published in [17].

1.5 Structure of this thesis

This thesis is organized as follows. In section 2, an introduction of graph transformation systems is given. In section 3, nested constraints and application conditions are defined and transformations into normal form are presented. The main results are presented and proved in sections 4 and 5. Applications of the transformations results are presented in section 6. The notations and the concepts are illustrated by a running example, a simple railroad control system. A conclusion including further work is given in section 7.

2 Graph transformation systems

This section recalls the application of graph transformation rules according to the "double-pushout" approach. Details and pointers to the literature can be found in [5, 4, 16]. In the following, the notations of the latter paper are used, as it considers graph transformation with "injective" rules and injective matchings.

A *label alphabet* $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ is a pair of finite sets of *node labels* and *edge labels*. A *graph* over $\mathcal{C}$ is a tuple $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ consisting of two finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*, two *source* and *target functions* $s_G, t_G \colon E_G \to V_G$, and two *labelling functions* $l_G \colon V_G \to \mathcal{C}_V$ and $m_G \colon E_G \to \mathcal{C}_E$.

$\mathcal{C}_V = \{\text{Waypoint}\}$, $\mathcal{C}_E = \{\text{Track}, \text{Train}\}$,
$G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ with
$V_G = \{v_1, v_2\}$ and $E_G = \{e_1, e_2, e_3\}$

i	1	2	3
$s_G(e_i)$	v_1	v_2	v_1
$t_G(e_i)$	v_2	v_1	v_2
$l_G(v_i)$	Waypoint	Waypoint	
$m_G(e_i)$	Track	Track	Train

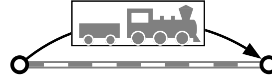


Figure 4: A concrete graph and its graphical representation (abstract graph)

A *graph morphism* $m \colon G \to H$ between two graphs G and H consists of two total functions $m_V \colon V_G \to V_H$ and $m_E \colon E_G \to E_H$ that preserve sources, targets, and labels, that is, $s_H \circ m_E = m_V \circ s_G$, $t_H \circ m_E = m_V \circ t_G$, $l_H \circ m_V = l_G$, and $m_H \circ m_E = m_G$.

A morphism m is *injective* [*surjective*] if both m_V and m_E are injective [surjective], and an *isomorphism* if m is both injective and surjective. In the latter case G and H are *isomorphic*, denoted by $G \cong H$. A morphism m is an *inclusion* if $m_V(v) = v$ and $m_E(e) = e$ for all $v \in V_G$ and $e \in E_G$. In this case, G is a *subgraph* of H. Convention: In the following, M denotes the class of all injective graph morphisms and m in M denotes m is injective.

For a morphism $m \colon G \to H$, G is the *domain* and H the *codomain* of m. The *range* of m is the *image* of G in H, i.e. the subgraph $K \subseteq H$, which consists of all nodes $m_V(v)$ and edges $m_E(e)$, for which there are $v \in V_G$ and $e \in E_G$ in the domain of m. The range of m is not to be confused with the codomain of m.

A *rule* $p = \langle L \leftarrow K \to R \rangle$ consists of two graph morphisms with a common domain K. Assume throughout that both $K \to L$ and $K \to R$ are inclusions. The *application* of p to a graph G amounts to the following steps:

1. Find an injective graph morphism $m \colon L \to G$ satisfying the *dangling condition*: No edge in $G - m(L)$ is incident to a node in $m(L - K)$.

2. Remove $m(L - K)$ from G, yielding a graph D, a graph morphism $K \to D$ which is the restriction of m, and the inclusion $D \to G$.

3. Add $R - K$ to D, yielding a graph H and graph morphisms $D \to H$ and $m^* \colon R \to H$.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ m\big\downarrow & (1) & \big\downarrow & (2) & \big\downarrow m^* \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Figure 5: A transformation step in form of a double pushout

This construction yields the pushout diagrams (1) and (2) in figure 5. The transformation of G into H is denoted by $G \Rightarrow_{p,m,m^*} H$ and one says that $m\colon L \to G$ is the *match* and $m^*\colon R \to H$ is the *comatch* of p in H. $G \Rightarrow_p H$ expresses that there is a graph morphisms m and m^* such that $G \Rightarrow_{p,m,m^*} H$.

Definition 1. (graph transformation system) A *graph transformation system* is a tuple $GTS = \langle \mathcal{C}, \mathcal{R} \rangle$ consisting of a label alphabet $\mathcal{C}$ and a finite set $\mathcal{R}$ of rules. $G \Rightarrow_{\mathcal{R}} H$ denotes that there exists a rule $p \in \mathcal{R}$ such that $G \Rightarrow_p H$.

Consider the following graph transformation system, which constitutes the dynamic part of our railroad control system.

Example 2. (railroad control system) Let $\langle \mathcal{C}, \mathcal{R} \rangle$ be a graph transformation system, where $\mathcal{C}$ consists of suitable labels for waypoints, tracks and trains, and $\mathcal{R}$ is a set of rules as depicted in figure 6. The system models the extension of a net of railroad tracks and the adding and movement of trains thereon.

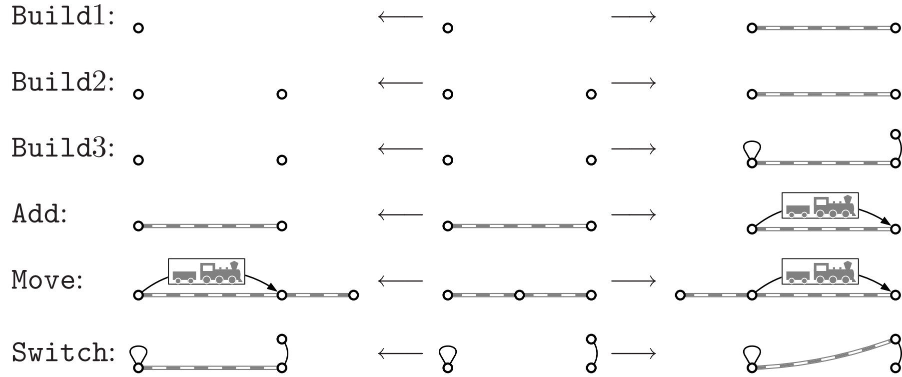


Figure 6: A set of rules for a railroad control system

While already functional, the control system is not safe, given the current definition. E.g., trains may crash, if a train moves onto a piece of track that is already occupied by another train, or trains may derail, if if a switch is thrown, while a train is on it. Thus we need to formalize, which behaviour we exclude. One possible way is to describe the set of unwanted system states and to ensure that these states or graphs are never reached by a transition.

3 Constraints and application conditions

In the following, graph constraints and application conditions are considered. Constraints are a means of defining a set of graphs. The basic idea is to describe graphs by subgraphs and to relate these subgraphs by morphisms.

3.1 Constraints and application conditions

Basic graphical consistency constraints, also called graph constraints, were first investigated in [18]. In a subsequent paper [22], graph constraints were used to describe policies for access control. In [6], the notion of graph constraints and application conditions were lifted to high-level objects and replacement systems. In the following, the notion of constraints is extended to nested constraints.

Definition 2. (constraint) *Constraints* over a graph P are defined inductively as follows: For an arbitrary morphism $x\colon P \to C$, $\exists x$ is a (*basic*) constraint over P. For an arbitrary morphism $x\colon P \to C$ and a constraint c over C, $\forall(x, c)$ and $\exists(x, c)$ are (*conditional*) constraints over P. For constraints c, c_i $(i \in I)$ [over P], true, false, $\neg c$, $\wedge_{i\in I} c_i$ and $\vee_{i\in I} c_i$ are (*Boolean*) constraints [over P].

$$\exists(P \xrightarrow{x} C) \qquad\qquad \forall(O \xrightarrow{w} P,\ \exists(P \xrightarrow{x} C))$$

Figure 7: The basic constraint $\exists x$ and the conditional constraint $\forall(w,\, \exists x)$

A morphism $p\colon P \to G$ *satisfies* a basic constraint $\exists x$ if there exists a morphism $q\colon C \to G$ in M with $q \circ x = p$. A morphism $p\colon P \to G$ *satisfies* a conditional constraint $\forall(x, c)$ $[\exists(x, c)]$ if for all [some] morphisms $q\colon C \to G$ in M with $q \circ x = p$, q satisfies c. Every morphism satisfies true, no morphism satisfies false. A morphism p *satisfies* a Boolean constraint $\neg c$ if p does not satisfy c; p *satisfies* $\wedge_{i\in I} c_i$ $[\vee_{i\in I} c_i]$ if p satisfies all [some] c_i with $i \in I$. We write $p \models c$ to denote that p satisfies c.

A graph G *satisfies* a constraint c of the form $\exists x$, $\exists(x, d)$ $[\forall(x, d)]$ if all [some] morphisms $p\colon P \to G$ in M satisfy c. Every graph satisfies true, no graph satisfies false. A graph G *satisfies* $\neg c$ if G does not satisfy c and $\wedge_{i\in I} c_i$ $[\vee_{i\in I} c_i]$ if it satisfies all [some] c_i with $i \in I$. We write $G \models c$ to denote that G satisfies c. Two constraints c and c' are *equivalent*, denoted by $c \equiv c'$, if, for all graphs G, $G \models c$ if and only if $G \models c'$.

$$\begin{array}{ccc} P & \xrightarrow{x} & C \\ & p \searrow \ = \ \swarrow q & \\ & G & \end{array}$$

Figure 8: Satisfiability of the basic constraint $\exists x$

Constraints of the form $\exists x$ and $\neg\exists x$ with empty morphism $x\colon \emptyset \to C$ are denoted by $\exists C$ and $\nexists C$, respectively. Constraints of the form $\neg\exists x$ are abbreviated by $\nexists x$. Constraints of the form $\exists x$, $\exists(x, c)$ are said to be *existential*; constraints of the form $\forall(x, c)$ are *universal*.

Remark. Constraints correlate morphisms, e.g. in the case of a basic constraint $\exists x$ two morphisms, depicted in figure 8 as vertical morphism p and q. The quantifier for the first morphism is omitted, since it is always complementary to the quantifier of the constraint. E.g., consider the basic constraint $\exists x$ with morphism $x\colon P \to C$. A graph G satisfies a constraint $\exists x$, if *all* injective morphisms $p\colon C \to G$ satisfy $\exists x$. The given quantifier always refers to the second morphism, in this case q. E.g., an injective morphism $p\colon P \to G$ satisfies the constraint $\exists x$, if there *exists* an injective morphism $q\colon C \to G$.

Example 3. (basic graph constraints) The following constraints can be expressed by basic graph constraints.

Every node has a loop:	$\exists(\bigcirc \to \bigcirc\!\!\circlearrowleft)$
There exists a node without loop:	$\neg\exists(\bigcirc \to \bigcirc\!\!\circlearrowleft)$
There exists a node with a loop:	$\exists(\emptyset \to \bigcirc\!\!\circlearrowleft)$
The graph is loop-free:	$\neg\exists(\emptyset \to \bigcirc\!\!\circlearrowleft)$
There exists at most one node:	$\neg\exists(\emptyset \to \bigcirc\ \ \bigcirc)$

Remark. The definition of constraints allows infinite conjunctions and disjunctions of constraints. This increase the expressiveness of constraints, but one cannot decide in general, if an infinite constraint is satisfied or not. Thus they have very little practical relevance. For example, consider the constraint

The graph is acyclic: $\bigwedge_{k=1}^{\infty} \neg\exists(\emptyset \to \mathrm{C}_k)$

where C_k denotes a cycle of length k.

In [6], positive and negative atomic constraints are considered. A morphism $p\colon P \to G$ satisfies the positive atomic constraint $\mathrm{PC}(x)$ with morphism $x\colon P \to C$ if there does exist a morphism $q\colon C \to G$ in M with $q \circ x = p$. A morphism $p\colon P \to G$ satisfies the negative atomic constraint $\mathrm{NC}(x)$ with morphism $x\colon P \to C$ if there does not exist a morphism $q\colon C \to G$ in M with $q \circ x = p$. Negative atomic constraints are equivalent to positive ones (with negation).

[6]	this paper / [17]
$\mathrm{PC}(x)$	$\exists x$
$\mathrm{NC}(x)$	$\neg\exists C$ if x is in M
with $x\colon P \to C$	true if x is not in M

Table 1: Comparison of notations for constraints

Fact 1. For morphisms x in M, we have $\mathrm{NC}(x) \equiv \neg\exists C$; otherwise, $\mathrm{NC}(x) \equiv$ true.

Proof. For the proof, see fact 7 in the appendix. □

Remark. The definition of constraints generalizes the ones in [18, 22, 6], because arbitrary nested constraints are allowed. E.g., it is possible to express constraints like "For all nodes, there exists an outgoing edge such that, for all edges outgoing from the target, the target has a loop.", which was not possible before, because conditional constraints were not considered.

$$\exists(\underset{1}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\rightarrow\!\underset{2}{\bigcirc},\ \forall\,(\underset{1}{\bigcirc}\!\rightarrow\!\underset{2}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\rightarrow\!\underset{2}{\bigcirc}\!\rightarrow\!\underset{3}{\bigcirc}\!\circlearrowleft))$$

Furthermore, counting of elements is possible.

Example 4. (counting) The following properties of graphs can be expressed as graph constraints:

All nodes have exactly n outgoing edges (given $n \in \mathbb{N}$):

$$\exists(\bigcirc \rightarrow \bigcirc, c_{out=n})$$

There exists a node with an even number of incoming edges:

$$\forall(\bigcirc \rightarrow \bigcirc, \vee_{n\in\mathbb{N}_0}\, c_{in=2n})$$

There exists a node, with the same number of outgoing and incoming edges:

$$\forall(\bigcirc \rightarrow \bigcirc, \vee_{n\in\mathbb{N}_0}\, c_{out=n} \wedge c_{in=n})$$

where, given $n \in \mathbb{N}$, $c_{out=n}$ and $c_{in=n}$ are subconstraints that are satisfied, if and only if the number of outgoing and incoming edges equals n, respectively.

For a given $n \in \mathbb{N}$, the constraints $c_{out=n}$ and $c_{in=n}$ are defined with the help of a constraint $c(x)$ which is used to count edges. For a morphism $x\colon P \rightarrow C$, let $c(x) = \vee_{e\in\mathcal{E}}\, \exists(e \circ x)$, where the set $\mathcal{E}$ consists of all epimorphisms that do not identify edges. Then define $c_{out=n} = c(\bigcirc \rightarrow S_n) \wedge \neg c(\bigcirc \rightarrow S_{n+1})$ and $c_{in=n} = c(\bigcirc \rightarrow T_n) \wedge \neg c(\bigcirc \rightarrow T_{n+1})$ where S_n $[T_n]$ denotes the star with outgoing [incoming] edges as depicted in figure 9. E.g., $c(\bigcirc \rightarrow S_n)$ is satisfied, if and only if there exists at least n outgoing edges, and $\neg c(\bigcirc \rightarrow S_{n+1})$ is satisfied, if and only if there exists at most n edges.

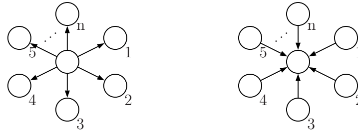


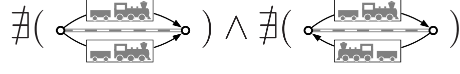
Figure 9: The graphs S_n and T_n

Example 5. (rail net constraints) Consider the railroad system in example 2. For security aspects, some rail net constraints for rail net graphs are formalized. E.g., the control system should ensure that trains do not derail (every train is on a track), that trains do not crash (no two trains occupy the same piece of track) and that trains keep a safety distance of one track (two trains do not occupy neighbouring pieces of track, except they face in opposite directions).

(c_1) Every train occupies one piece of track:

$$\exists(\qquad \rightarrow \qquad)$$

(c_2) Different trains occupy different pieces of track:



(c_3) Two adjacent trains face in opposite directions:

In the following, nested application conditions for rules are considered. Application conditions for rules were first introduced in [8]. In a subsequent paper [15], a special kind of application conditions were considered which can be represented in a graphical way. In the graph case, contextual conditions like the existence or non-existence of certain nodes and edges or certain subgraphs in the given graph can be expressed. In [18, 6], a simple form of nested application conditions are considered.

Definition 3. (application condition for a rule) An *application condition* $a = (a_L, a_R)$ for a rule $p = \langle L \leftarrow K \rightarrow R \rangle$ consists of a constraint a_L over L and a constraint a_R over R, called *left and right application condition*, respectively. A *rule with an application condition* is a tuple $p' = \langle p, a \rangle$, where p is a rule and a is an application condition for p. A *direct derivation* $G \Rightarrow_{p',m,m^*} H$ is possible, if and only if $G \Rightarrow_{p,m,m^*} H$, $m \models a_L$ and $m^* \models a_R$.

Application of a rule $p' = \langle p, a \rangle$ means to find an occurrence of the left-hand side in the input graph G, to check the left application condition a_L, to replace the occurrence of the left-hand side by the right-hand side of the rule and finally to check the right application condition a_R. If a_R is not satisfied, the application of p has to be reversed, i.e. $H \Rightarrow_{p,m^*,m} G$.

Remark. Application conditions are defined as constraints. But their use is different: A constraint defines a set of graphs, an application condition defines a set of morphisms or in this context, matches. In this sense, an application condition restricts the applicability of its rule.

Example 6. (simple application conditions) The following properties of morphisms with domain $\underset{1}{\circ}\ \underset{2}{\circ}$ can be expressed as application conditions:

There is no edge from node $m(1)$ to node $m(2)$: $\neg\exists(\underset{1}{\circ}\ \underset{2}{\circ} \rightarrow \underset{1}{\circ}\!\!\rightarrow\!\!\underset{2}{\circ})$

There is no path connecting node $m(1)$ and $m(2)$: $\bigwedge_{k=1}^{\infty} \neg\exists(\underset{1}{\circ}\ \underset{2}{\circ} \rightarrow \mathrm{P}_k)$
(P_k denotes a path of length k from $\underset{1}{\circ}$ to $\underset{2}{\circ}$)

If there is an edge from $m(1)$ to $m(2)$, then also from $m(2)$ to $m(1)$:
$\forall(\underset{1}{\circ}\ \underset{2}{\circ} \rightarrow \underset{1}{\circ}\!\!\rightarrow\!\!\underset{2}{\circ},\ \exists(\underset{1}{\circ}\!\!\rightarrow\!\!\underset{2}{\circ} \rightarrow \underset{1}{\circ}\!\!\rightleftarrows\!\!\underset{2}{\circ}))$

Remark. In [6], simple nested application conditions of the form $\forall(x, \vee_{i\in I}\exists x_i)$ and $\forall(x, \wedge_{i\in I}\neg\exists(x_i))$ with morphisms x_i are considered. The definition of application conditions generalizes the ones in [18, 6], because arbitrary nested application conditions are allowed.

Example 7. (application conditions) Consider the railway control system in example 2 and the rail net constraints of example 5. Some of the constraints are already "preserved" by some of the rules of the transformation system: `Build1`, 2, 3 preserve all constraints, i.e. for $i, j \in \{1, 2, 3\}$ and for every derivation step $G \Rightarrow_{\texttt{Build}j,g} H$ with injective match g, we have: $G \models c_i$ implies $H \models c_i$. Constraint c_1 is preserved by `Move`, too. Unfortunately, this is not the case for c_2 and c_3. To ensure that `Move` can only be applied if the resulting graph satisfies the constraints c_2 and c_3, these constraints are transformed first into right application conditions and then into left application conditions for `Move`. Intuitively, we seek application conditions for the rule `Move` to express that every train should move only on a free piece of track and, after movement, two trains should not occupy neighbouring pieces of track (except they have a different direction).

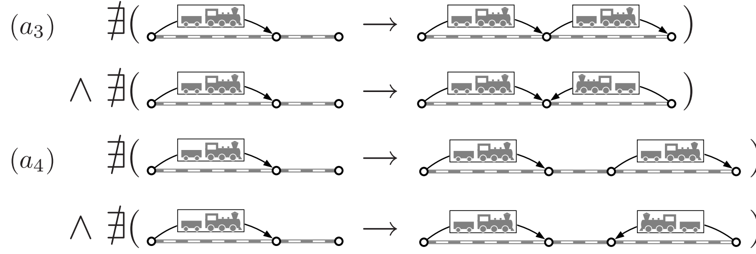


Figure 10: Application conditions for the movement of trains

The rule `Add`, like `Move`, preserves constraint c_1, but not c_2 and c_3. On the other hand, `Switch` preserves c_3 and (under the assumption that c_1 was satisfied) also c_2, but not c_1. Thus, we have to find appropriate application conditions for these rules (for the result, see section 6, example 16).

rule	c_1	c_2	c_3
`Build1`	true	true	true
`Build2`	true	true	true
`Build3`	true	true[1]	true
`Add`	true	a_1	a_2
`Move`	true	a_3	a_4
`Switch`	a_5	true[1]	true

Figure 11: Needed application conditions to preserve the constraints

3.2 Equivalences and implications

Constraints are defined inductively. In the following, we speak of a *subconstraint* c, if c is nested into a conditional constraint of the form $\mathrm{Q}(x, d)$ with $\mathrm{Q} \in \{\forall, \exists\}$,

[1] for $G \Rightarrow H$ holds, $G \models c_1 \wedge c_2$ implies $G \models c_2$.

i.e. $c = d$ or c occurs in d. Be aware that constraints are satisfied by graphs, subconstraints like application conditions are satisfied by morphisms.

E.g., consider the constraint $c = (\forall(x, \exists y) \wedge \exists x)$. A graph G that satisfies c must also satisfy $\forall(x, \exists y)$ and $\exists x$. But G does not have to satisfy the subconstraint $\exists y$. Only certain morphisms with codomain G have to.

As every constraint may be seen as a subconstraint of a larger constraint, it is important to distinguish between these two interpretations. Therefore, let us consider the difference and connection between equivalence and implication of constraints on the one hand and "m-equivalence" and "m-implication" of subconstraints and application conditions on the other, where m stands for morphism.

Definition 4. (implication, m-implication and m-equivalence) A constraint c *implies* a constraint c', denoted by $c \Rightarrow c'$, if, for all graphs G holds $G \models c$ implies $G \models c'$. A constraint c *m-implies* c', denoted by $c \dot{\Rightarrow} c'$, if, for all morphisms $p\colon P \to G$ in M holds $p \models c$ implies $p \models c'$. Two constraints c and c' are said to be *m-equivalent*, denoted by $c \dot{\equiv} c'$, if, for all morphisms $p\colon P \to G$ in M holds $p \models c$ if and only if $p \models c'$.

symbol	meaning
$\equiv$	equivalence for constraints
$\dot{\equiv}$	equivalence for subconstraints and application conditions
$\Rightarrow$	implication for constraints
$\dot{\Rightarrow}$	implication for subconstraints and application conditions

Table 2: List of symbols

Example 8. Consider the following equivalence for constraints, which clearly does not hold for subconstraints or application conditions:

$$\exists(\underset{1}{\circ}\;\underset{2}{\circ} \to \underset{1}{\circ}\!\rightarrow\!\underset{2}{\circ}) \equiv / \not\dot{\equiv}\; \exists(\underset{1}{\circ}\;\underset{2}{\circ} \to \underset{1}{\circ}\!\leftarrow\!\underset{2}{\circ})$$

For two existential or universal subconstraints, it suffices to show the m-equivalence [m-implication] to conclude the equivalence [implication] as constraints. The other direction does not hold.

Fact 2. For two existential [universal] constraints c_1, c_2 over P holds: $c_1 \dot{\Rightarrow} c_2$ implies $c_1 \Rightarrow c_2$ and $c_1 \dot{\equiv} c_2$ implies $c_1 \equiv c_2$.

Proof. Let G be an arbitrary graph. We have $c_1 \dot{\Rightarrow} c_2$ iff for all $m\colon P \to G$ in M holds $m \models c_1$ implies $m \models c_2$. One can show: $G \models c_1$ iff for all [there exists] $p\colon P \to G$ in M holds [such that] $p \models c_1$ implies for all [there exists] $p\colon P \to G$ in M holds [such that] $p \models c_2$ iff $G \models c_2$. □

Constraints can be replaced by equivalent constraints. Subconstraints of a constraint can be replaced by equivalent subconstraints yielding an equivalent constraint.

Theorem 1. (substitution theorem) Let d, d' two constraints. Let c be a constraint with at least one occurrence of d and let c' be a constraint yielded from c by replacing some occurrence of d with d'. We distinguish three cases:

(1) $c \doteq c'$, if $d \doteq d'$.

(2) $c \equiv c'$, if $d \equiv d'$ and d is not a subconstraint.

(3) $c \equiv c'$, if $d \doteq d'$ and d is a subconstraint.

Proof.

(1) Proof by induction over the structure of constraints (see [31])

Basis: Let $c = d$ and $c' = d'$. Then $c = d \doteq d' = c'$.

Step: Case $c = \mathrm{Q}(x, b)$. By induction hypothesis $b \doteq b'$, where b' is yielded from b by replacing some occurrence of d with d'. Then we have $p \models \exists(x, b)$ $[\forall(x, b)]$ iff there exists [for all] $q\colon C \to G$ in M with $q \circ x = p$ such that [holds] $q \models b$. Because $b \doteq b'$, $q \models b'$ and $p \models \mathrm{Q}(x, b')$.

Case $c = \neg b$ and $c' = \neg b'$. By induction hypothesis $b \doteq b'$, where b' is yielded from b by replacing some occurrence of d with d'. From the semantic of $\neg$ follows $c = \neg b \doteq \neg b' = c'$

Case $c = b_1 \vee b_2$. Then d occurs either in b_1 or b_2. By induction hypothesis, $b_1 \doteq b_1'$ $[b_2 \doteq b_2']$, where b_1' $[b_2']$ is yielded from b_1 $[b_2]$ by replacing some occurrence of d with d'. From the semantic of $\vee$ follows $b_1 \vee b_2 \doteq b_1' \vee b_2$ $[b_1 \vee b_2 \doteq b_1 \vee b_2']$

Case $c = b_1 \wedge b_2$. Analogue to case $c = b_1 \vee b_2$.

(2) By induction over the structure of Boolean constraints (analogue to (1))
Basis: Let $c = d$ and $c' = d'$. Then $c = d \equiv d' = c'$.

Step: *Case* $c = \neg b$ and $c' = \neg b'$. By induction hypothesis $b \equiv b'$, where b' is yielded from b by replacing some occurrence of d with d'. From the semantic of $\neg$ follows $c = \neg b \equiv \neg b' = c'$

Case $c = b_1 \vee b_2$. Then d occurs either in b_1 or b_2. By induction hypothesis, $b_1 \equiv b_1'$ $[b_2 \equiv b_2']$, where b_1' $[b_2']$ is yielded from b_1 $[b_2]$ by replacing some occurrence of d with d'. From the semantic of $\vee$ follows $b_1 \vee b_2 \equiv b_1' \vee b_2$ $[b_1 \vee b_2 \equiv b_1 \vee b_2']$

Case $c = b_1 \wedge b_2$. Analogue to case $c = b_1 \vee b_2$.

(3) Let d be a subconstraint of c, i.e. d occurs in a conditional constraint $\mathrm{Q}(x, s)$ with $\mathrm{Q} \in \{\forall, \exists\}$. Without loss of generality assume $\mathrm{Q}(x, s)$ is not a subconstraint, i.e. there does not exist another conditional constraint $\mathrm{Q}(y, t)$ in c, such that $\mathrm{Q}(x, s)$ occurs in t. We have already shown in case (1): $\mathrm{Q}(x, s) \doteq \mathrm{Q}(x, s')$. It remains to show $\mathrm{Q}(x, s) \equiv \mathrm{Q}(x, s')$ and finally $c \equiv c'$. The former follows from fact 2, the latter is shown in case (2).

□

3.3 Normal form for constraints and application conditions

In the following, we define a normal form for constraints and application conditions. A constraint is in normal form, if it is true or false or no non-injective morphisms are used for its definition, every Boolean subconstraint is in disjunctive [conjunctive] normal form, every negation symbol is innermost and if the logical constants true and false do not occur.

Definition 5. (normal form) A constraint c is in *normal form*, if the following conditions are satisfied:

(p_1) every morphism of c is in M
(p_2) every negation symbol is innermost, i.e. every negated subconstraint is basic
(p_3) every Boolean subconstraint is in disjunctive [conjunctive] normal form
(p_4) there is no subconstraint true or false in c, unless c is true or false

Theorem 2. (normal form) For every constraint, there is an equivalent constraint in normal form.

Construction. For each criterion, there is a transformation that transforms a given constraint into an equivalent one, such that it has the desired property p_i, while preserving the satisfaction of the above criteria p_j with $j \leq i$.

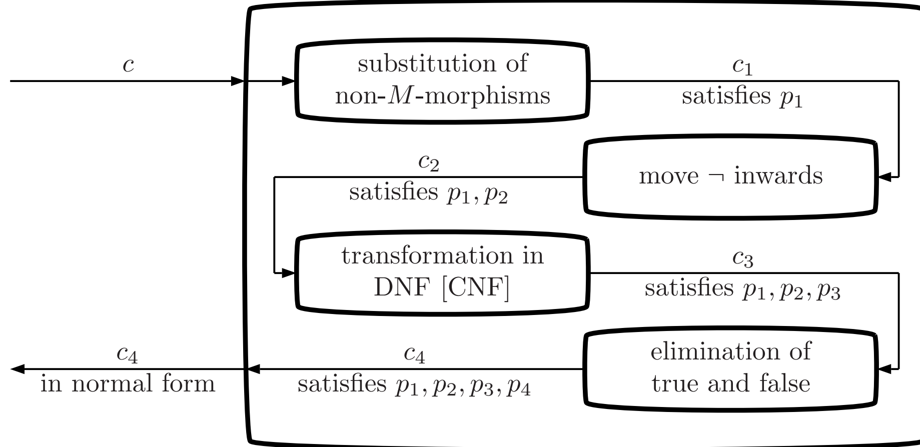


Figure 12: Transformation of a constraint c into normal form

The idea of the transformations is to use equivalences as rules for transforming a constraint into an equivalent one. An equivalence, e.g. $\neg\neg c \equiv c$, must be read strictly from left to right and gives rise to a rule $\neg\neg c \rightsquigarrow c$. An application of a rule means to replace an occurrence of the left-hand side in a constraint d with the right hand side leading to an equivalent constraint d'. A rule $c \equiv d$ may only be applied for constraints, a rule $c \doteq d$ may only be applied for subconstraints and application conditions.

Constraints with non-M-morphisms do not give more expressive power. For every constraint with arbitrary morphisms, there is an equivalent constraint with morphisms in M. A constraint c is said to be an *M-constraint*, if all morphisms of c are in M.

Lemma 3. (substitution of non-M-morphisms) For every constraint, there is an equivalent M-constraint.

Construction. Given a constraint c, apply the following rules (read the equivalences from left to right), as long as possible:

1. $\exists x \dot{\equiv}$ false
2. $\exists x \equiv \neg\exists P$
3. $\exists(x, d) \dot{\equiv}$ false
4. $\exists(x, d) \equiv \neg\exists P$
5. $\forall(x, d) \dot{\equiv}$ true
6. $\forall(x, d) \equiv \exists P$

where $x\colon P \to C$ not in M.

Proof. The proof of the equivalences 1-6 is given in the appendix, fact 5 and fact 7. From the substitution theorem 1 follows that the result of the transformation is equivalent to the input. The transformation terminates after a finite number of steps, as in each step, the number of non-M-morphisms is reduced at least by one. Since the rule-set is complete, i.e. for every possible occurring of a morphism, there is a fitting rule, the result is a M-constraint. □

In the following, we show, how negation symbols can be moved into subconstraints till they are innermost.

Lemma 4. (move ¬ inwards) For every constraint, there is an equivalent constraint with innermost negation symbols.

Construction. Given a constraint c, apply the following rules (read the equivalences from left to right), as long as possible:

1. $\neg\neg c \dot{\equiv}/\equiv c$
2. $\neg$true $\dot{\equiv}/\equiv$ false
3. $\neg$false $\dot{\equiv}/\equiv$ true
4. $\neg(c \vee d) \dot{\equiv}/\equiv \neg c \wedge \neg d$
5. $\neg(c \wedge d) \dot{\equiv}/\equiv \neg c \vee \neg d$
6. $\neg\forall(x, c) \dot{\equiv}/\equiv \exists(x, \neg c)$
7. $\neg\exists(x, c) \dot{\equiv}/\equiv \forall(x, \neg c)$

Proof. The rules 1-5 are logical tautologies. The equivalence of the rules 6-7 is shown in fact 5 and fact 7 in the appendix. From the substitution theorem 1 follows that the result of the transformation is equivalent to the input. The transformation terminates after finite many steps, as in each step, negation symbols are consumed or a ¬ symbol is moved deeper into the constraint.

Since the rule-set is complete, i.e. for every possible occurring of negation symbol, there is a fitting rule, the result is a constraint with innermost negation symbols. □

If every negation symbol is in front of basic atomic constraints, every Boolean subconstraint can be brought into normal form.

Lemma 5. (transformation in disjunctive [conjunctive] normal form) For every constraint with innermost negation symbols, there is an equivalent constraint, such that every Boolean subexpression is in disjunctive [conjunctive] normal form.

Construction. Given a constraint c, apply the following rules (read the equivalences from left to right), as long as possible:

1. $(b \vee d) \wedge c \mathrel{\dot{\equiv}/\equiv} (b \wedge c) \vee (d \wedge c) \qquad [(b \wedge d) \vee c \mathrel{\dot{\equiv}/\equiv} (b \vee c) \wedge (d \vee c)]$

Proof. The rule is a logical tautology. From the substitution theorem 1 follows that the result of the transformation is equivalent to the input. The transformation terminates after finite many steps. It remains to show that the result is in disjunctive [conjunctive] normal form. This can be shown by structural induction (see [14]). Consider a Boolean constraint a. If a is either conditional or basic, it is per definition in disjunctive [conjunctive] normal form. If $a = \neg b$, b is basic and per definition in disjunctive [conjunctive] normal form. If $a = b \vee c$ $[b \wedge c]$, by induction one can rewrite b and c into disjunctive [conjunctive] normalform b^D and c^D $[b^C$ and $c^C]$. Then $b^D \vee c^D$ $[b^C \wedge c^C]$ is already in disjunctive [conjunctive] normal form. If $a = b \wedge c$ $[b \vee c]$, by induction one can rewrite b and c into disjunctive [conjunctive] normalform b^D and c^D $[b^C$ and $c^C]$. $b^D \wedge c^D$ $[b^C \vee c^C]$ can be rewritten using the rule 1: Assume that $b^D = b_1^D \wedge \ldots \wedge b_m^D$ and $c^D = c_1^D \wedge \ldots \wedge c_n^D$. Then

$$\begin{array}{rl} & (b_1^D \vee \ldots \vee b_m^D) \wedge (c_1^D \vee \ldots \vee c_n^D) \\ \dot{\equiv}/\equiv & (b_1^D \wedge (c_1^D \vee \ldots \vee c_n^D)) \vee \ldots \vee (b_m^D \wedge (c_1^D \vee \ldots \vee c_n^D)) \\ \dot{\equiv}/\equiv & ((b_1^D \wedge c_1^D) \vee \ldots \vee (b_1^D \wedge c_n^D)) \vee \ldots \vee ((b_m^D \wedge c_1^D) \vee \ldots \vee (b_m^D \wedge c_n^D)) \end{array}$$

As each b_i^D and c_i^D are conjunctions of propositions or their negations, the entire formula is in disjunctive normal form. Analogously, if $a = b^C \vee c^C$. □

Remark. In general, it is not possible to bring an arbitrary constraint into a form, such that every logical symbol is outside, or alternatively innermost. Conjunctive and disjunctive symbols cannot jump over an arbitrary conditional constraint, i.e. $\vee_{i \in I} \mathrm{Q}(x, c_i) \not\equiv \mathrm{Q}(x, \vee_{i \in I} c_i)$ for $\mathrm{Q} \in \{\forall, \exists\}$.

More precisely, only the following equivalences/implications hold:

$$\begin{array}{lclclcl} \exists(x, \vee_{i \in I} c_i) & \dot{\equiv} & \vee_{i \in I} \exists(x, c_i) & \qquad & \exists(x, \vee_{i \in I} c_i) & \Leftarrow & \vee_{i \in I} \exists(x, c_i) \\ \forall(x, \wedge_{i \in I} c_i) & \dot{\equiv} & \wedge_{i \in I} \forall(x, c_i) & & \forall(x, \wedge_{i \in I} c_i) & \Rightarrow & \wedge_{i \in I} \forall(x, c_i) \\ \exists(x, \wedge_{i \in I} c_i) & \dot{\Rightarrow} & \wedge_{i \in I} \exists(x, c_i) & & \exists(x, \wedge_{i \in I} c_i) & \Rightarrow & \wedge_{i \in I} \exists(x, c_i) \\ \forall(x, \vee_{i \in I} c_i) & \dot{\Leftarrow} & \vee_{i \in I} \forall(x, c_i) & & \forall(x, \vee_{i \in I} c_i) & \Leftarrow & \vee_{i \in I} \forall(x, c_i) \end{array}$$

For a proof, see fact 9 in the appendix.

In the following it is shown that true and false can be eliminated from a constraint, if it is not always true or false.

Lemma 6. (elimination of true **and** false**)** For every constraint different from true or false, there is an equivalent constraint without true or false.

Construction. Given a constraint c, apply the following rules (read the equivalences from left to right), as long as possible:

1. $\exists(x, \text{false}) \doteq \text{false}$
2. $\exists(x, \text{false}) \equiv \neg\exists P$ — provided that $x\colon P \to C$
3. $\forall(x, \text{true}) \doteq \text{true}$
4. $\forall(x, \text{true}) \equiv \exists P$ — provided that $x\colon P \to C$
5. $\exists(x, \text{true}) \doteq / \equiv \exists x$
6. $\forall(x, \text{false}) \doteq / \equiv \neg\exists x$
7. $\exists(\text{id}_P) \doteq / \equiv \text{true}$
8. $\neg\text{true} \doteq / \equiv \text{false}$
9. $\neg\text{false} \doteq / \equiv \text{true}$
10. $\vee_{i\in I}\, c_i \doteq / \equiv \text{true}$ — if $c_j = \text{true}$ for some $j \in I$
11. $\vee_{i\in I}\, c_i \doteq / \equiv \vee_{i\in I\setminus\{j\}}\, c_i$ — if $c_j = \text{false}$ for some $j \in I$
12. $\wedge_{i\in I}\, c_i \doteq / \equiv \wedge_{i\in I\setminus\{j\}}\, c_i$ — if $c_j = \text{true}$ for some $j \in I$
13. $\wedge_{i\in I}\, c_i \doteq / \equiv \text{false}$ — if $c_j = \text{false}$ for some $j \in I$
14. $\vee_{i\in\emptyset}\, c_i \doteq / \equiv \text{false}$
15. $\wedge_{i\in\emptyset}\, c_i \doteq / \equiv \text{true}$

Proof. The equivalence of the rules 1-7 is shown in fact 5 and fact 7 in the appendix. The rules 8-15 are logical tautologies. From the substitution theorem 1 follows that the result of the transformation is equivalent to the input. The transformation terminates after a finite number of steps, as in each step the "size" of the constraint is reduced, i.e. where

$$size(c) = \begin{cases} 8 + size(d) & \text{if } c = \mathrm{Q}(x, d) \\ 4 & \text{if } c = \exists x \\ 2 + \Sigma_i size(d_i) & \text{if } c = \vee_{i\in I}\, d_i \text{ or } c = \wedge_{i\in I}\, d_i \\ 2 + size(d) & \text{if } c = \neg d \\ 1 & \text{if } c = \text{true or } c = \text{false} \end{cases}$$

It remains to show that the result has the desired form. This proof is straightforward and can be done by structural induction. □

Proof of lemma 2 (normal form) Consider the transformation of a constraint into normal form, as depicted in figure 12. For every transformation holds: the result is equivalent to the input, thus the whole transformation is correct. It remains to show that the result satisfies the desired properties. This is the case, as none of the transformations does introduce subconstraints with non-M-morphisms, transformations 3 and 4 introduce negation symbols only in front of basic constraints, and the elimination of true and false preserves the disjunctive [conjunctive] normal form of Boolean constraints. □

4 Transformation of constraints

Constraints can be used to specify valid states of a transformation system. But how is this specification implemented? E.g., one could check the satisfaction of a constraint every time a rule is applied. However for safety-critical systems this check may be too late: If a constraint is not satisfied, then the unwanted changes have already been made to the system. Consequently, we need to find a way, to test before a rule is applied, if the result will satisfy a constraint. One step towards such a look into the future is to transform the constraint into equivalent right application conditions for the rules of a transformation system. Instead to test the satisfaction of the constraint, it suffices to respect the condition of the rule which shall be applied. Indeed, a right application condition is also tested after the application of its rule, but we will see in section 5 that right application conditions can be transformed into left application conditions, which can be checked before their rules are applied.

4.1 Transformation into right application conditions

Nested constraints can be transformed into right application conditions for any given rule, such that the condition of a rule is satisfied, if and only if the outcome will satisfy the constraint. In the following, it is shown how.

Theorem 7. (transformation of constraints into right application conditions) Let $\langle L \leftarrow K \rightarrow R \rangle$ be a given rule and c be a given constraint. Then there is a right application condition $T(c)$ such that, for all comatches $m^*\colon R \rightarrow H$ in $\mathcal{M}$,

$$m^* \models T(c) \Leftrightarrow H \models c.$$

The construction is an extended version of the one for basic constraints in [6]. The idea behind this transformation is to enrich the constraint with the right-hand side of the rule. This is possible, since given a comatch, we already know that corresponding elements exists in the resulting graph. In the following, we elaborate on the underlying idea of this transformation, before we present the exact construction.

Idea of the construction. Let $\delta = \langle L \leftarrow K \rightarrow R \rangle$ be a given rule and c be a given constraint. Assume for the sake of simplicity that $c = \exists(\emptyset \rightarrow C) \equiv \exists C$. The constraint c is satisfied by a graph H, if and only if we find an injective morphism $C \rightarrow H$. Our goal is to replace c with an equivalent right application condition for δ, i.e. a condition on morphisms with domain R.

Assume, the rule δ has just been applied, i.e. $G \Rightarrow_{\delta,m,m^*} H$ and $m^*\colon R \rightarrow H$ is the comatch. What do we have to check with respect to m^* to decide, if H satisfies c, but if we do not want to test c directly? What additionally elements have to exist in H aside from the image of R, which existence we must verify? How do we have to "extend" the morphism m^* to cover these elements? Intuitively, we search a graph U together with morphisms $R \rightarrow U$ and $C \rightarrow U$. If there exists an injective morphism $U \rightarrow H$, we can conclude that there exists a morphism $C \rightarrow H = C \rightarrow U \rightarrow H$.

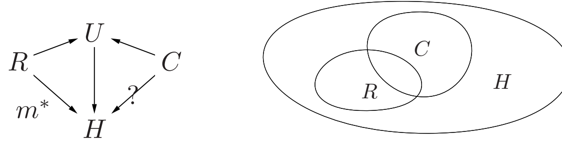


Figure 13: A condition $\exists(R \to U)$ for m^*

If H satisfies c, then there exists at least one morphism $C \to H$. Because $U \to H$ is required to be injective, its domain depends on the overlapping of R and C's images in H. E.g., U could be $R{+}C$ for the case that the image of C does not intersect with the image of R in H. Nevertheless the union of the images in H itself is always an adequate candidate for the domain U, as there exist apparently surjective morphisms $R \to U$ and $C \to U$. Since we have to find a condition for all possible morphisms with domain R, we have to consider all possible overlappings of R and C, which corresponds to the search for all surjective morphisms $e\colon R{+}C \to U$, such that R and C apart are mapped injectively. For each morphism e, we define $R \to U = R \to R{+}C \to U$. We end up with a disjunction of conditions of the form $\exists(R \to U)$, from which only one must apply for a given comatch $R \to H$.

Consider the following example: Let $\delta = \langle$○⟲ $\leftarrow$ ○ $\to$ ○$\rangle$ be a rule that deletes a loop of some node and let $c = \exists($○⟲$)$ a constraint that says the graph is not loop-free. For the construction of an equivalent right application condition, we have to consider all morphisms $e\colon$○ + ○⟲ $\to U$ and yield a disjunction of two conditions,

$\exists($○ $\to$ ○⟲$) \vee \exists($○ $\to$ ○ ○⟲$)$

which says either the node of the right-hand side has still a loop left or some other node has a loop.

Construction. For an existential [universal] constraint c over P and a graph R, $T(c)$ is constructed with the help of a right application condition $T_p(c')$. Let $p\colon \emptyset \to R$ and $y\colon \emptyset \to P$. Define

$$T(c) = T_p(\forall(y, c)) \qquad [T(c) = T_p(\exists(y, c))]$$

T is compatible with Boolean operations, i.e. $T(\text{true}) = \text{true}$, $T(\text{false}) = \text{false}$, $T(\neg d) = \neg T(d)$, $T(\wedge_{i \in I} c_i) = \wedge_{i \in I} T(c_i)$, and $T(\vee_{i \in I} c_i) = \vee_{i \in I} T(c_i)$.

Given a constraint c over P and a morphism $p\colon P \to S$ in M, construct $T_p(c)$ as follows: For a basic constraint $c = \exists x$, construct the pushout (1) in figure 14 of p and x leading to $t\colon S \to T$ and $q\colon C \to T$ and all surjective morphisms $e\colon T \to U$ such that $u = e \circ t$ and $r = e \circ q$ are in M. Let $\mathcal{E}$ denote the set of all these surjective morphisms and, for $e \in \mathcal{E}$, $u = e \circ t$.

$$T_p(c) = \vee_{e \in \mathcal{E}} \exists u$$

For a conditional constraint $c = \forall(x, d)$ [$\exists(x, d)$] and a morphism $p\colon P \to S$, construct $T_p(\exists x) = \vee_{e \in \mathcal{E}} \exists u$ over S according to p. The choice of a surjective morphism $e \in \mathcal{E}$ determines morphisms $u = e \circ t$ and $r = e \circ q$. For the constraint d and the morphism r, construct $T_r(d)$ of d according to r.

$$T_p(c) = \wedge_{e \in \mathcal{E}} \forall(u, T_r(d)) \quad [\vee_{e \in \mathcal{E}} \exists(u, T_r(d))]$$

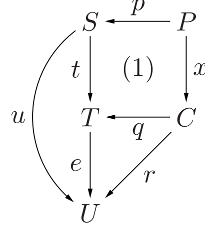


Figure 14: Construction of $T_p(c)$

T_p is compatible with Boolean operations.

Remark. Consider the definition of $T(c)$. The first step is to transform the constraint c into a state, where it can be interpreted ambiguously as a constraint or alternatively as a right application condition for rules with right-hand side $\emptyset$. Afterwards, this condition is transformed with the help of T_p into a right application condition for rules with right-hand side R.

Assume c is a basic or conditional constraint with morphism $x: P \to C$ and $p: P \to S$. $T_p(c)$ enriches the morphism x with the right-hand side R according to p. All elements outside the range of p originate from R, and the morphism p specifies, how (much) R is glued with P. If applied for the first time, P is $\emptyset$ and S=R. Consequently all overlappings of C and S are considered. Otherwise p represents the knowledge, how R was integrated by an earlier iteration of T_p. Then C and S are glued to T according to p and x and all contractions of T are considered, such that C and S apart are mapped injectively. These contractions are necessary, as they allow elements that originate from C and are in this sense new, to be identified with elements from outside the range of p, i.e. elements of R that were in this branch not yet glued together with elements of the constraint. Altogether we get all possible extensions of x with R.

While this transformation is as simple as it gets, it is not optimal in the sense that every resulting subcondition is really necessary.

Proof. By structural induction, we show: (*) For arbitrary constraints c over P and morphisms $p: P \to S$ holds: For arbitrary morphisms $p'': S \to H$ in M,

$$p'' \models T_p(c) \text{ if and only if } p' = p'' \circ p \models c.$$

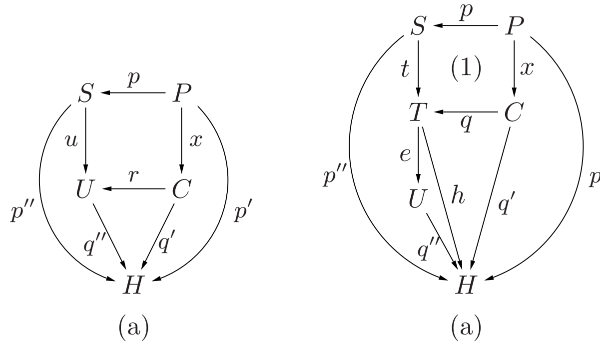


Figure 15: Correspondence of $T_p(c)$ and c

The following statements will be used: (1) If $q'': U \to H$ is a morphism in M with $q'' \circ u = p''$, then there exists a morphism $q' = q'' \circ r: C \to H$ in M with $q' \circ x = p'$. (2) If $q': C \to H$ is a morphism in M with $q' \circ x = p'$, then there exist morphisms $e \in \mathcal{E}$ and $q'': U \to H$ in M with $q'' \circ e \circ q = q'$ and $q'' \circ e \circ t = p''$ (see figure 15). The second statement may be seen as follows: The universal property of pushouts implies the existence of a unique morphism $h: T \to H$ with $h \circ t = p''$ and $h \circ q = q'$. Now let $h = q'' \circ e$ be a epi-mono factorization of h with surjective morphism e and injective morphism q'' in M. Then $q'' \circ e \circ t = h \circ t = p''$ in M implies $e \circ t$ in M and $q'' \circ e \circ q = h \circ q = q'$ in M implies $e \circ q$ in M (M closed under decompositions). Thus, there exists a morphism $q'': U \to H$ in M with $q'' \circ r = q'$ and $q'' \circ e \circ t = p''$ for some $e \in \mathcal{E}$.

Now (*) is proven for arbitrary constraints. For basic constraints, (*) follows from the definitions and (1) and (2): $p'' \models T_p(\exists x)$ iff for some surjective morphisms $e \in \mathcal{E}$, $p'' \models \exists u$ iff for some surjective morphisms $e \in \mathcal{E}$, there exists a morphism $q'': U \to H$ in M with $q'' \circ u = p''$ if$^{(2)}$ and only if$^{(1)}$ there exists a morphism $q': C \to H$ in M with $q' \circ x = p'$ iff $p' \models \exists x$. For conditional constraints, (*) follows from the definitions, (1) and (2), and the inductive hypothesis: $p'' \models T_p(\forall(x, d))$ iff for all surjective morphisms $e \in \mathcal{E}$, $p'' \models \forall(u, T_r(d))$ iff for all surjective morphisms $e \in \mathcal{E}$ and all morphisms $q'': U \to H$ in M with $q'' \circ u = p''$, $q'' \models T_r(d)$ if$^{(1)(*)}$ and only if$^{(2)(*)}$ for all morphisms $q': C \to H$ in M with $q' \circ x = p'$, $q' \models d$ iff $p' \models \forall(x, d)$. $p'' \models T_p(\exists(x, d))$ iff for some surjective morphisms $e \in \mathcal{E}$, $p'' \models \exists(u, T_r(d))$ iff for some surjective morphisms $e \in \mathcal{E}$, there exists a morphism $q'': U \to H$ in M with $q'' \circ u = p''$such that $q'' \models T_r(d)$ if$^{(2)(*)}$ and only if$^{(1)(*)}$ there exists a morphism $q': C \to H$ in M with $q' \circ x = p'$ such that $q' \models d$ iff $p' \models \exists(x, d)$. For Boolean constraints, (*) follows directly from the definitions and the inductive hypothesis. Consequently, (*) holds for all constraints.

It remains to prove the main statement: For all morphisms $m^*: R \to H$ in M, $m^* \models T(c) \Leftrightarrow H \models c$. This is done by structural induction. Let $p: \emptyset \to R$ and $y: \emptyset \to P$. For existential [universal] constraints c over P, one can show (see appendix, fact 7): $c \equiv \forall(y, c)$ [$c \equiv \exists(y, c)$]. Thus it suffices to show: $m^* \models T_p(\forall(y, c))$ iff $H \models \forall(y, c)$ [and analogously $m^* \models T_p(\exists(y, c))$ iff $H \models \exists(y, c)$].

This follows from (*) and the existence and uniqueness of $p' = m^* \circ p$ in M: $m^* \models T_p(\forall(y, c))$ iff $m^* \circ p \models \forall(y, c)$ iff there exists $p': \emptyset \to H$ in M such that $p' \models \forall(y, c)$. Analogue $m^* \models T_p(\exists(y, c))$ iff $m^* \circ p \models \exists(y, c)$ iff for all $p': \emptyset \to H$ in M holds $p' \models \forall(y, c)$.

For Boolean constraints, the statement follows directly from the definitions and the inductive hypothesis. This completes the proof. □

Compared to [17], the above transformation is simpler. The reason for this is that unlike [17], this paper considers graph transformation with injective matchings.

Remark. In [17], $T(c)$ is defined as follows: Let A denote the set of all triples $a = \langle S, s, p \rangle$ with arbitrary $s: R \to S$ and $p: P \to S$ in M such that the pair $\langle s, p \rangle$ is jointly epimorph. $T(c) = \vee_{a \in A} \exists(s, T_p(c)) \quad [\wedge_{a \in A} \forall(s, T_p(c))]$

For the double-pushout approach with matches in M (see [16]), the construction of $T(c)$ can be simplified: Since M is closed under decompositions, $p'' \circ s = m^*$ and $p'': S \to H$ in M implies s in M. Therefore, it suffices to consider the subset

$A' \subseteq A$ of all triples $a = \langle S, s, p\rangle$ with both $s\colon R \to S$ and $p\colon P \to S$ in M such that the pair $\langle s, p\rangle$ is jointly epimorph. These are exact the morphisms s, p, one gets if the pushout T of $\emptyset \to P$ and $\emptyset \to R$ with $t\colon R \to T$ and $q\colon P \to T$ is constructed and and all surjective morphisms e are considered such that both $s = e \circ t$ and $p = e \circ q$ are in M.

Up to now, we have seen, how a constraint can be transformed into an equivalent right application condition with the property that if its rule is to be applied, the condition is satisfied, if and only if the constraint is satisfied afterwards. It is also possible to transform a constraint into a left application condition, such that if the rule shall be applied, the condition is true, if only if the input satisfies the constraint. This transformation is not to be confused with the guarantee or preservation of a constraint, as defined in section 6.

Remark. Given a rule p and a constraint c, there is a left application condition $T_L(c)$ such that, for all matches $m\colon L \to G$ in M, $m \models T_L(c) \Leftrightarrow G \models c$: Let $T_L(c)$ be the right application condition of the inverse rule $p^{-1} = \langle R \leftarrow K \to L\rangle$ and c. Then $T_L(c)$ is a left application condition of p with the wanted property.

4.2 Example

Example 9. (transformation of constraints into right application conditions) Consider the rule `Move` and the constraint $\text{NoTwo} = \neg\exists(x) \equiv \nexists\text{Two}$ where $x\colon \emptyset \to \text{Two}$ and Two denotes the graph with a track edge and two parallel train edges (see the first subconstraint of c_2 in example 5), saying that no two trains are allowed to occupy the same piece of track in the same direction.

$$\text{NoTwo} = \neg\exists(\emptyset \longrightarrow \qquad) \equiv \nexists(\qquad)$$

In the following, a right application condition for `Move` is constructed, which can alternatively be checked to ensure that the result of the application of `Move` satisfies the constraint NoTwo. The idea of the transformation is, to “enrich” a constraint with the right-hand side of a rule. In this case, one intuitively seeks every possible injective embedding of the forbidden context Two into the right-hand side of `Move`.

For the transformation of constraint NoTwo, apply T_p two times and yield the following right application condition for `Move`:

$$\begin{aligned} T(\text{NoTwo}) &= T(\neg\exists x) = \neg T(\exists x) \\ &= \neg T_p(\forall(\mathrm{id}_\emptyset, \exists x)) = \neg \wedge_{a\in A} \forall(s, T_r(\exists x)) \\ &= \neg \wedge_{a\in A} \forall(s, \vee_{e\in\mathcal{E}_a} \exists u) = \neg\forall(id_R, \vee_{e\in\mathcal{E}} \exists u). \end{aligned}$$

The pushout (1) of $p\colon \emptyset \to R$ and $\mathrm{id}_\emptyset\colon \emptyset \to \emptyset$ is the disjoint union $T = R + \emptyset = R$ with the injections id_R and $q = p$. Let A denote the set of all surjective morphisms $a\colon T \to S$ with injective $s = a \circ \mathrm{id}_R$ and $r = a \circ q$. In this case, there is only one a and $A = \{\mathrm{id}_R\}$.

For $r = a \circ q = \mathrm{id}_R \circ q = q = p$, the pushout (2) of $r\colon \emptyset \to S$ and $x\colon \emptyset \to \text{Two}$ is the disjoint union $T = S + \text{Two}$ with the injections t and q. Let $\mathcal{E}$ denote the

set of all surjective morphisms $e: T \to U$ with injective $u = e \circ t$ and $r = e \circ q$. By the equivalence $\forall(\mathrm{id}, c) \doteq c$, we have

$$T(\mathrm{NoTwo}) \doteq \neg \vee_{e \in \mathcal{E}} \exists u \doteq \wedge_{e \in \mathcal{E}} \neg \exists u.$$

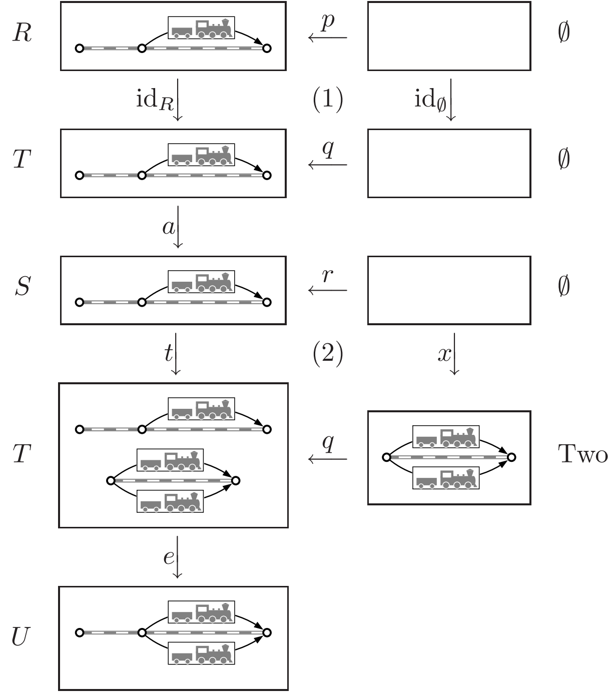


Figure 16: Transformation of NoTwo into a right application condition of `Move`

The result is a conjunction of application conditions from which one is depicted in figure 16. The depicted one says that the moved train is not on a piece of track which is occupied by another train moving in the same direction.

The complete list of morphisms $(u)_{e \in \mathcal{E}}$ for $a = id_R$ consists of all morphisms $u = (e \circ t): R \to U$ with surjective morphisms $e: T \to U$ as sketched by their codomains U in figure 17. Be aware that there are U which represent more than one morphism u respectively e.

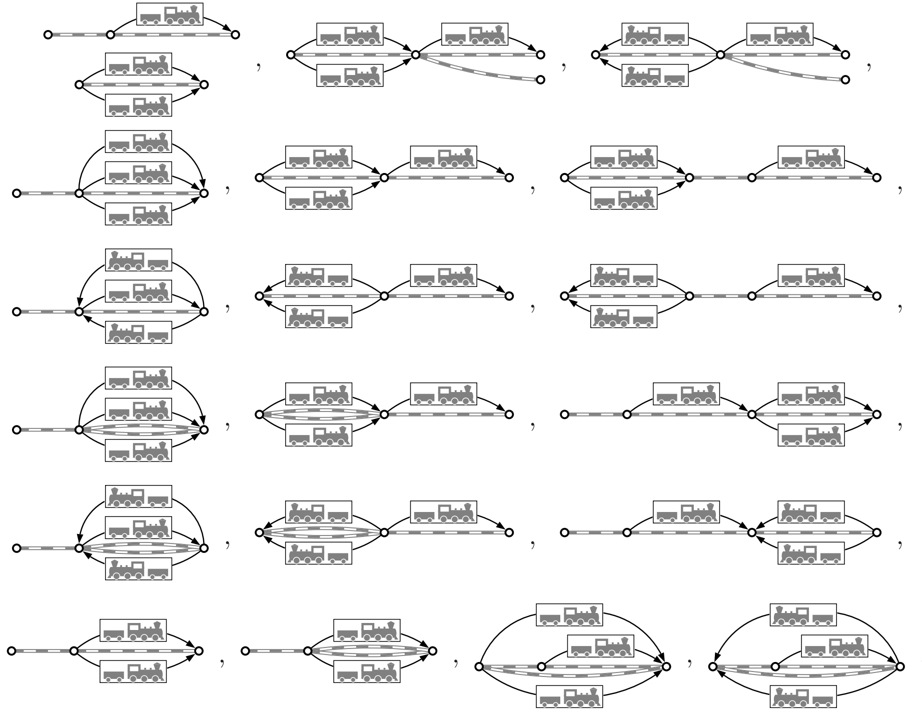


Figure 17: The codomains U of all morphisms $(u)_{e\in\mathcal{E}}$ for $a = id_R$

4.3 Complexity of the transformation

The transformation of constraints into application conditions relies on the transformation T_p. To consider the complexity of the first, one has to investigate the complexity of the latter. In the following, a metric for the size of constraints and application conditions is proposed, and the complexity of T_p is examined for basic constraints with respect to the size of the input and the result.

A simple measure for the size of a constraint (application condition) would be the number of morphisms. The complexity of a constraint would correspond to the number of conditional and basic constraints it consists of. However, this measure is too rough to make any meaningful statement about the complexity of T_p: The size of the application condition $T_p(c)$ depends on the kind of morphisms used in the constraint c, as well as on the morphism p. Consequently, a measure for the size of morphisms is needed and the size of a constraint has to reflect this.

Definition 6. (size of graphs and morphisms) For a morphism $x\colon P \to C$, define

$$size(x) = size(C) - size(x(P))$$

where $x(P)$ denotes the image of P in G and, for a graph G

$$size(G) = |V_G| + |E_G|$$

Example 10. (size of morphisms) Consider the following morphisms:

$$size(\bigcirc \rightarrow \bigcirc\!\!\rightarrow\!\!\bigcirc) = 2 \qquad \text{and} \qquad size(\underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc} \rightarrow \underset{1=2}{\bigcirc}\!\!\circlearrowleft) = 0.$$

Definition 7. (size of constraints and application conditions) For constraints [application conditions] c, the size is defined inductively as follows:

$$size(c) = \begin{cases} size(x) & \text{if } c = \exists x \\ size(x) + size(d) & \text{if } c = \mathrm{Q}(x,d) \text{ with } \mathrm{Q} \in \{\forall, \exists\} \\ \Sigma_i size(d_i) & \text{if } c = \vee_{i\in I}\, d_i \text{ or } c = \wedge_{i\in I}\, d_i \\ size(d) & \text{if } c = \neg d \end{cases}$$

In the case of conjunctions and disjunctions, we use sum of the sizes of all constraints d_i for the definition of *size*. This is motivated by the fact that in the worst case all constraints d_i have to be checked.

Example 11. (size of constraints) Consider the size of the following constraint $c = \neg\exists(\underset{1}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc},\ \forall(\underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc}\!\!\rightarrow\!\!\underset{3}{\bigcirc}\!\!\circlearrowleft))$. The size of c is calculated as follows:

$$\begin{aligned} size(c) &= size(\neg\exists(\underset{1}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc},\ \forall(\underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc}\!\!\rightarrow\!\!\underset{3}{\bigcirc}\!\!\circlearrowleft))) \\ &= size(\exists(\underset{1}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc},\ \forall(\underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc}\!\!\rightarrow\!\!\underset{3}{\bigcirc}\!\!\circlearrowleft))) \\ &= size(\underset{1}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc}) + size(\forall(\underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc}\!\!\rightarrow\!\!\underset{3}{\bigcirc}\!\!\circlearrowleft)) \\ &= 2 + size(\underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc} \rightarrow \underset{1}{\bigcirc}\!\!\rightarrow\!\!\underset{2}{\bigcirc}\!\!\rightarrow\!\!\underset{3}{\bigcirc}\!\!\circlearrowleft) \\ &= 2 + 3 \end{aligned}$$

Lemma 8. In the worst case, the size of application condition $T_p(c)$ grows at least exponentially with respect to the size of the input constraint c.

Proof. In the following, a more concrete statement is proved: In the worst case, $n! \leq size(T_p(c)) \leq m!^2$, where $n = \min(size(p), size(\exists x))$ and $m = \max(size(p), size(\exists x))$. Consider the worst case. Given a morphism $p\colon \emptyset \rightarrow R$ and a constraint $c = \exists(\emptyset \rightarrow C)$, C and R are discrete and all node have the same label. $T_p(c)) = \vee_{e\in\mathcal{E}}\, e \circ t$, where $t\colon R \rightarrow R{+}C$ and all surjective morphisms e have to be considered, such that R and C are mapped injectively. The size of $T_p(c)$ depends on $m = size(p)$ and $n = size(x)$ as follows:

$$f(n,m) = \sum_{k=0}^{\min(m,n)} \binom{m}{k} \cdot \binom{n}{k} \cdot k! \cdot (m{+}n{-}k)$$

We have to consider all gluings of R and C, which can be divided into $\min(m,n)$ categories, where $\min(m,n)$ represents the maximum number of nodes that are to be identified. For every k with $0 \leq k \leq \min(m,n)$, we have consider all possibilities, how to choose k nodes out of R, k nodes of out C and all $k!$ possible permutations, how to glue them together, multiplied with the size of the resulting morphism.

Let $m \geq n$ or alternatively, exchange m and n. The term $f(m,n)$ can be downward measured as follows:

$$f(m,n) \;=\; \sum_{k=0}^{\min(m,n)} \binom{m}{k} \cdot \binom{n}{k} \cdot k! \cdot (m{+}n{-}k)$$

$$
\begin{aligned}
&\geq \sum_{k=n}^{n} \binom{n}{k} \cdot \binom{n}{k} \cdot k! \cdot (m+n-k) \\
&\geq n!
\end{aligned}
$$

and upward measured as follows:

$$
\begin{aligned}
f(m,n) &= \sum_{k=0}^{\min(m,n)} \binom{m}{k} \cdot \binom{n}{k} \cdot k! \cdot (m+n-k) \\
&= \sum_{k=0}^{\min(m,n)} \frac{m! \cdot n! \cdot k! \cdot (m+n-k)}{k! \cdot (m-k)! \cdot k! \cdot (n-k)!} \\
&= \sum_{k=0}^{\min(m,n)} \frac{m! \cdot n! \cdot (m+n-k)}{k! \cdot (m-k)! \cdot (n-k)!} \\
&= m! \cdot n! \cdot \sum_{k=0}^{m} \frac{(m+n-k)}{k! \cdot (m-k)! \cdot (n-k)!} \\
&\leq m!^2 \cdot \sum_{k=0}^{m} \frac{(2m-k)}{k! \cdot (m-k)!^2} \\
&\leq m!^2
\end{aligned}
$$

where the last step holds for $m \geq 5$. □

Remark. The notations O and Ω are not used, as they present an upper bound of the worst case and a lower bound of the best case, but we were interested in a lower bound of the worst case.

4.4 Elimination of constraints and application conditions

In the following section, equivalence-preserving transformations are presented that allow to eliminate superfluous subconstraints and subconditions, respectively. A constraint should be reduced as much as possible before it is transformed into a right application condition, because the size of the condition grows exponential with respect to the size of the constraint, as shown in section 4.3.

Furthermore, an application condition has to be checked every time its rule shall be applied. Thus it is important to optimize application conditions, too. While the transformation of constraints into right application conditions is correct, it is not optimal in the sense that every subcondition is always necessary, as example 12 will show.

Constraints without logical symbols, i.e. without Boolean subconstraints, should have alternating quantifiers. For every such constraint with consecutive quantifiers Q, there is an equivalent constraint with single quantifier Q. In this way, equal consecutive quantifiers can be eliminated and constraints shortened.

Fact 3. $\mathrm{Q}(x, \mathrm{Q}(y, c)) \equiv / \dot{\equiv}\ \mathrm{Q}(y \circ x, c))$ for $\mathrm{Q} \in \{\forall, \exists\}$.

Proof. The equivalence is shown in fact 5 and fact 7 in the appendix. □

In general, constraints are a collection of constraints combined by conjunction and disjunction. Several of these constraints may be similar. In the following, some equivalence-preserving rules for modifying and condensing constraints are presented.

Fact 4. For constraints and application conditions the following implications hold:

$$\begin{array}{ll}
\exists x \Rightarrow/\dot{\Rightarrow} \exists x' & \text{if } \exists i \text{ in } M \text{ such that } i \circ x' = x. \\
\exists(x,c) \Rightarrow/\dot{\Rightarrow} \exists(x',c') & \text{if } c \dot{\Rightarrow} c' \text{ and } \exists i \text{ in } M \text{ such that } i \circ x' = x. \\
\forall(x,c) \Rightarrow/\dot{\Rightarrow} \forall(x',c') & \text{if } c \dot{\Rightarrow} c' \text{ and } \exists i \text{ in } M \text{ such that } i \circ x = x'.
\end{array}$$

Analogue to constraints and subconstraints, it is necessary to distinguish between inner and outer logical symbols: A logical symbol in a constraint is said to be *inner* if it occurs in a subconstraint.

$$\forall(x,\ \underline{\neg\exists y \vee \exists z}) \wedge \exists x$$

Figure 18: Subconstraints and inner logical symbols vs. constraints and outer logical symbols

Lemma 9. (elimination of constraints) Every constraint can be transformed into an equivalent constraint according to the following rules: (1) Replace subconstraints by equivalent ones and (2) condense outer [inner] conjunctions and disjunctions: (a) Eliminate c_l from $\wedge_{i\in I} c_i$ provided $c_k \Rightarrow c_l$ [$c_k \dot{\Rightarrow} c_l$] for some $k \neq l$ and (b) eliminate c_k from $\vee_{i\in I} c_i$ provided $c_k \Rightarrow c_l$ [$c_k \dot{\Rightarrow} c_l$] for some $k \neq l$.

Proof.

(1) See substitution theorem 1.

(2) To show:

$$\begin{array}{ll}
\wedge_{i\in I} c_i \doteqdot \wedge_{i\in I\setminus\{j\}} c_i & \text{if } c_k \dot{\Rightarrow} c_j \text{ for some } j \neq k \text{ and } j,k \in I \\
\wedge_{i\in I} c_i \equiv \wedge_{i\in I\setminus\{j\}} c_i & \text{if } c_k \Rightarrow c_j \text{ for some } j \neq k \text{ and } j,k \in I \\
\vee_{i\in I} c_i \doteqdot \vee_{i\in I\setminus\{j\}} c_i & \text{if } c_j \dot{\Rightarrow} c_k \text{ for some } j \neq k \text{ and } j,k \in I \\
\vee_{i\in I} c_i \equiv \vee_{i\in I\setminus\{j\}} c_i & \text{if } c_j \Rightarrow c_k \text{ for some } j \neq k \text{ and } j,k \in I
\end{array}$$

These equivalences are shown in fact 10 in the appendix. The rest follows from the substitution theorem 1.

□

Application conditions can be treated like subconstraints: Both are satisfied by morphisms. In this sense, we have only inner logical symbols. Thus the elimination procedure of constraints can be applied to application conditions as well, as long as the m-implication and m-equivalence ($\dot{\Rightarrow}$ and $\doteqdot$) between the conditions is shown.

Example 12. (elimination of application conditions)
Consider the right application condition $T(\text{NoTwo}) = \wedge_{e\in\mathcal{E}}\neg\exists u$ of example 9 and the complete list of morphisms $(u)_{e\in\mathcal{E}}$ for $a = \text{id}_R$ as depicted in figure 17. According to the elimination lemma 9, an application condition $\exists u$ from a conjunction can be eliminated, if $\exists u' \Rightarrow \exists u$ is shown for another condition $\exists u'$, i.e. if there exists an injective morphisms i, such that $i \circ u = u'$. Conversely, $\neg\exists u$ can be eliminated, if there is another condition $\neg\exists u'$ and an injective morphisms $U' \rightarrow^i U$, such that $i \circ u' = u$.

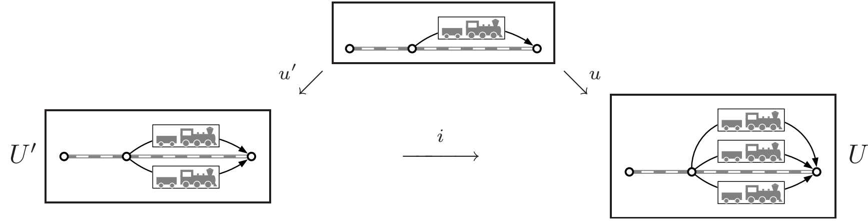


Figure 19: Elimination of $\neg\exists u$

Thus, it suffices to consider the following morphisms as sketched by their codomains U in figure 20.

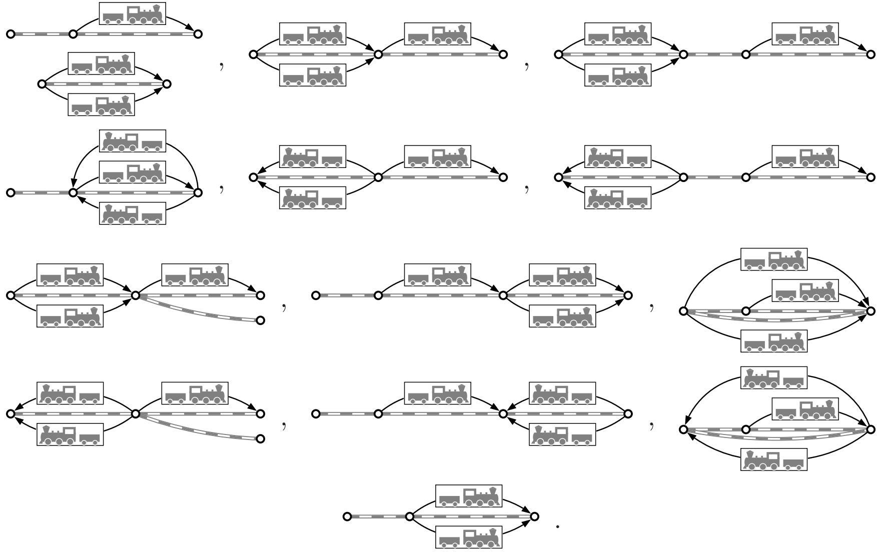


Figure 20: The codomains U of the morphisms $(u)_{e\in\mathcal{E}}$ left after the elimination

5 Transformation of right application conditions

Up to now, we have seen, how a constraint can be transformed into an equivalent right application condition. Every right application basically states that a rule should not have been applied, if the condition is not satisfied afterwards. It is possible to transform a right application condition into a left one, which can be checked before any changes to a system are made.

5.1 Transformation into left application conditions

Given an arbitrary right application condition for a rule, it can be transformed into an equivalent left application condition. In the following, it is shown how.

Theorem 10. (transformation from right to left application conditions) Given a rule $p = \langle L \leftarrow K \rightarrow R \rangle$ and a right application condition a. Then there is a left application condition $T_p(a)$ such that, for all direct derivations $G \Rightarrow_{p,m,m^*} H$ with m, m^* in M holds:

$$m \models T_p(a) \Leftrightarrow m^* \models a.$$

The construction of $T_p(a)$ is an extended version of the corresponding construction for basic application conditions in [6]. The idea is to use the rule itself to transform the condition: The rule is inversely applied, such that what was once the right-hand side in the condition, becomes the left-hand side.

Construction. Given a rule $p = \langle L \leftarrow K \rightarrow R \rangle$ and an arbitrary right application condition a, construct the left application condition $T_p(a)$ of a according p as follows: For a basic right application condition $\exists x$ with morphism $x\colon R \rightarrow X$, define $y\colon L \rightarrow Y$ by two pushouts (1) and (2) in figure 21 if the pair $\langle r, x \rangle$ has a pushout complement. Let $T_p(\exists x) = \exists y$ if $\langle r, x \rangle$ has a pushout complement and false otherwise. For a conditional right application condition $\mathrm{Q}(x, a)$ with $\mathrm{Q} \in \{\forall, \exists\}$, construct the left application condition $T_p(\exists x) = \exists y$ of $\exists x$ according to p, if the pair $\langle r, x \rangle$ has a pushout complement, and the left application condition $T_{p^*}(a) = b$ of a according to the "derived" rule $p^* = \langle Y \leftarrow Z \rightarrow X \rangle$. Let $T_p(\mathrm{Q}(x, a)) = \mathrm{Q}(y, b)$ if $\langle r, x \rangle$ has a pushout complement and false otherwise. T_p is compatible with Boolean operations.

$$\begin{array}{ccccc} L & \xleftarrow{\;l\;} & K & \xrightarrow{\;r\;} & R \\ {\scriptstyle y}\downarrow & (2) & \downarrow & (1) & \downarrow{\scriptstyle x} \\ Y & \xleftarrow[\;l^*\;]{} & Z & \xrightarrow[\;r^*\;]{} & X \end{array}$$

Figure 21: Transformation of application conditions

Proof. Let $G \Longrightarrow_{p,m,m^*} H$ be any direct derivation. To show is $m \models T_p(a) \Leftrightarrow m^* \models a$ for every application condition a. The proof is done by induction on the structure of application conditions. For basic right application conditions,

the statement follows immediately from the statement in [6]. For conditional right application conditions of the form $\mathrm{Q}(x, a)$ with $\mathrm{Q} \in \{\forall, \exists\}$, two cases may occur:

Case 1. The pair $\langle r, x\rangle$ has no pushout complement. Then $T_p(\exists x) = \text{false}$ and $m \not\models T_p(a)$. To show is $m^* \not\models a$. This is true, because there is no $q\colon X \to H$ with $q \in M$ and $q \circ x = m^*$. Otherwise, since the pair $\langle r, m^*\rangle$ has a pushout complement, the pair $\langle r, x\rangle$ would have a pushout complement in contradiction to case 1.

Case 2. The pair $\langle r, x\rangle$ has a pushout complement. Then the left application condition is of the form $T_p(a) = \mathrm{Q}(y, b)$. It remains to show that $m \models \mathrm{Q}(y, b) \Leftrightarrow m^* \models \mathrm{Q}(x, a)$. This is done by structural induction.
The following statements will be used: $[\alpha]$ Given a morphism $q'\colon Y \to G$ in M with $q' \circ y = m$, construct pushouts (1), (2), (5), (6) as above, where this time, first construct (6) as pullback leading in the right-hand side to a morphism $q\colon X \to H$ in M with $q \circ x = m^*$. $[\beta]$ Given a morphism $q\colon X \to H$ in M with $q \circ x = m^*$. From the double pushout for $G \Rightarrow_{p,m,m^*} H$ and $q \circ x = m^*$. Obtain the following decomposition in pushouts (1), (2), (5), (6): First (5) is constructed as pullback of q and d_1 leading to pushouts (1) and (5), with same square (1) as in the construction because of uniqueness of pushout complements for M-morphisms. Then (2) is constructed as pushout and we have $q'\colon Y \to G$ with $q' \circ y = m$ and pushout (6) induced by the pushouts (2) and (2)+(6). Since q is in M, z and q' are in M. $[\gamma]$ Given application conditions as above and a "derived" rule $p^* = \langle Y \leftarrow Z \to X\rangle$ with morphisms $q'\colon Y \to G$ and $q\colon X \to H$ we apply the inductive hypothesis to conclude $q' \models b$ iff $q \models a$.

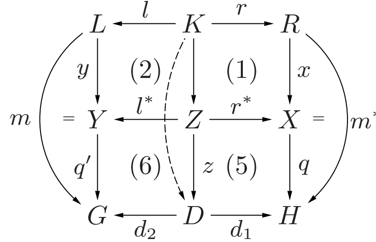


Figure 22: Decomposition of pushouts

For an universal application condition we have: $m \models \forall(y, b)$ iff for all morphisms $q'\colon Y \to G$ in M with $q' \circ y = m$ holds $q' \models b$ if$^{[\alpha],[\gamma]}$ and only if$^{[\beta],[\gamma]}$ for all morphisms $q\colon X \to H$ in M with $q \circ x = m^*$ holds $q \models a$ iff $m^* \models \forall(x, a)$. For an existential application condition we have: $m \models \exists(y, b)$ iff for some morphisms $q'\colon Y \to G$ in M with $q' \circ y = m$ holds $q' \models b$ if$^{[\beta],[\gamma]}$ and only if$^{[\alpha],[\gamma]}$ for some morphisms $q\colon X \to H$ in M with $q \circ x = m^*$ holds $q \models a$ iff $m^* \models \forall(x, a)$. For Boolean right application conditions, the statement follows directly from the definition and the inductive hypothesis. □

Up to now, we have seen, how a right application condition can be transformed into an equivalent left application condition. This is useful, as all left application

conditions can be checked, before any changes to a system are made. However, it is also possible to transform a left application condition into a right application condition. The idea is to use the inverse rule with the above transformation.

Remark. Given a rule p and a left application condition a_L. Then there is a right application condition a_R such that, for all direct derivations $G \Rightarrow_{p,m,m^*} H$ with m, m^* in M, $m^* \models a_R \Leftrightarrow m \models a_L$: Consider the inverse rule p^{-1} of p with right application condition a_L. Let $a_R = T_{p^{-1}}(a_L)$ be the left application condition of p^{-1}. Then a_R is a right application condition of p with the wanted property.

5.2 Example

Example 13. (transformation of right into left application conditions) Up to now, we have constructed a right application condition for `Move`, which basically states that the rule should not have been applied, if the condition is not satisfied afterwards. It is possible to transform this right application condition into a left one, which can be checked, before any changes to the system are made.

Consider the rule `Move` in example 2 and the right application condition

$$T(\text{NoTwo}) = \wedge_{e \in \mathcal{E}} \neg \exists u$$

of example 12. Then the transformation of the right application condition $T(\text{NoTwo})$ according to `Move` yields the left application condition

$$T_{\texttt{Move}}(T(\text{NoTwo})) = \wedge_{e \in \mathcal{E}} \neg \exists v.$$

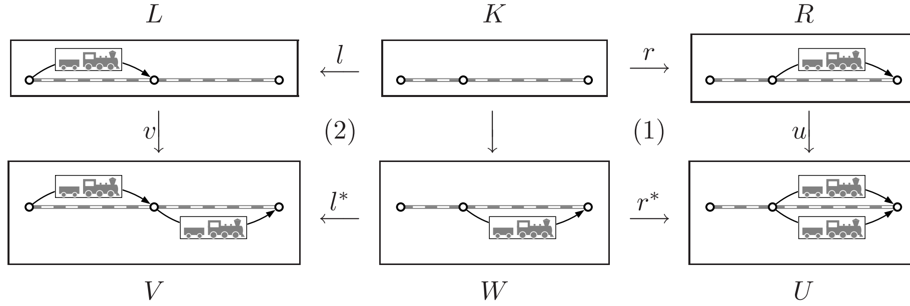


Figure 23: Transformation of $T(\text{NoTwo})$ into a left application condition of `Move`

Given a morphism $u: R \to U$, one has to check whether the pair $\langle r, u \rangle$ has a pushout complement, and if so, to apply the inverse rule of `Move` according to $u: R \to U$ yielding a comatch $v: L \to V$. The result of the transformation of a subcondition of $T(\text{NoTwo})$ is presented in figure 23. The left application condition is obtained from the transformation of the right subconditions. It says more or less that "the next piece of track is not allowed to be occupied by a train moving in the same direction".

The left application condition $\wedge_{e \in \mathcal{E}} \neg \exists v$ consists of a number of application conditions where the codomains of the morphisms $v: L \to V$ look as depicted in figure 24:

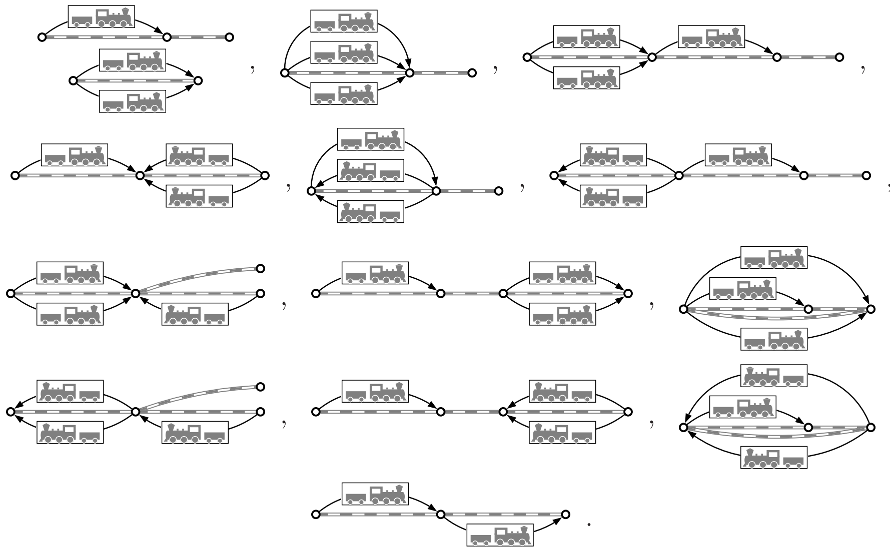


Figure 24: The codomains V of the morphisms $(v)_{e\in\mathcal{E}}$

Example 14. (nonexistence of pushout complement) There are configurations, for which there does not exist a pushout complement, and the inverse application of the rule is not possible. This involves rules, which add nodes. In these cases, the corresponding left application condition is true, which practically means the situation described by the right application condition can never occur.

Consider the rule `Build1` : $\langle L \leftarrow K \rightarrow R\rangle$ as depicted in figure 25, which adds a waypoint and a piece of track to the rail net, and the right application condition $a_r = \neg\exists(R \rightarrow X)$. It exists no triple Z, z, r^* such that (1) is a pushout. This

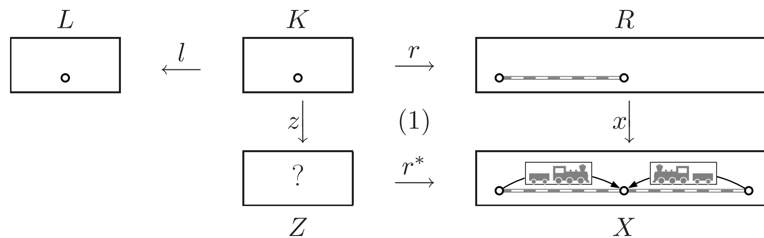


Figure 25: Transformation of a_r into a left application condition of `Build1`

situation cannot occur, because there cannot be a train on a track that has just been built, unless the train was constructed explicitly. For the same reason, there is no other track adjacent to a waypoint that has just been created.

5.3 Complexity of the transformation

In the following, the complexity of the transformation from right to left application conditions with respect to the size of the input and the result is considered.

Lemma 11. Let $p = \langle L \leftarrow K \rightarrow R \rangle$ a rule and n be the size of the right application condition a. Then the size of the left application condition $T_p(a)$ is in $O(n)$.

Proof. To show is that $\mathit{size}(T_p(a)) \in O(n)$ if $n = \mathit{size}(a)$. Consider the worst case, i.e. assume that the pushout complement always exists. For every subcondition in a, there is corresponding subcondition in $T_p(a)$, i.e. the number of morphisms stays the same. Let $R = K$. For a basic application condition $\exists x$, the size stays the same, i.e. $\mathit{size}(R \rightarrow C) = \mathit{size}(L \rightarrow C')$, because the difference between $|L|$ and $|R|$ is equal to the difference between $|C|$ and $|C'|$. The same holds for conditional constraints. □

6 Applications of the transformations

In the following, two important relations between constraints and application conditions are considered. The first one is the “guarantee” of a constraint by an application condition resp. rule, and the second one is the “preservation” of the satisfaction of a constraint.

Definition 8. (guarantee and preservation of constraints) A rule p *guarantees* a constraint c, if for every derivation step $G \Rightarrow_p H$ holds: $H \models c$. A rule p *preserves* a constraint c, if for every derivation step $G \Rightarrow_p H$ holds: If $G \models c$, then $H \models c$. A graph transformation system *guarantees* [*preserves*] a constraint c, if every rule guarantees [preserves] c. Sometimes, we say that a rule or graph transformation system is *c-guaranteeing* [*c-preserving*] and we mean that it *guarantees* [*preserves*] the constraint c.

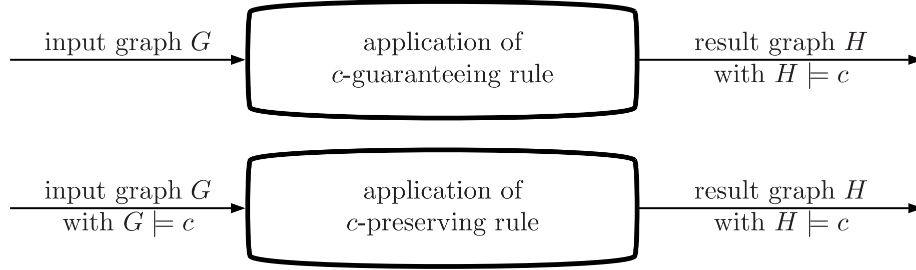


Figure 26: Guarantee and preservation of constraints

Preservation of constraints is weaker than guarantee, since the result graph does not have to satisfy the constraint, if input graph did not.

If a rule p does not guarantee or preserve a constraint c, we need to find an application condition $a(c)$, such that the rule $\langle p, a(c)\rangle$ with application condition $a(c)$ has the wanted property. An immediate consequence of theorem 7 and 10 is that for every constraint and every rule, there exists a left application condition that guarantees the constraint.

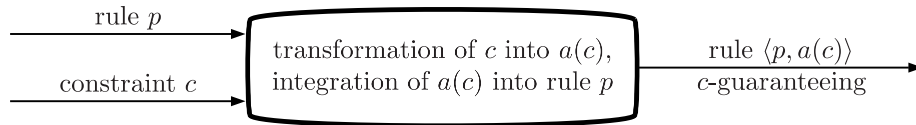


Figure 27: Integration of a constraint into a constraint-guaranteeing rule

Corollary. (constraint-guaranteeing integration) Given a rule p and a constraint c. There is a left application conditions $a(c)$ such that, for all direct derivations $G \Rightarrow_{p,m,m^*} H$ with m, m^* in M holds,

$$m \models a(c) \Leftrightarrow H \models c.$$

Proof. Let $T_R(c)$ be the right application condition of c, and $T_p(T_R(c))$ the left application condition of $T_R(c)$. Let $a(c) = T_p(T_R(c))$. Then the left application condition has the wanted property. □

With such an application condition, it is now possible to check the satisfaction of a constraint, before any changes to a system are made. In this sense, constraints can be integrated into each rule of a transformation system, ensuring that the system never makes a transition to a state that violates these constraints.

While guarantee of constraints is useful, e.g. for safety critical systems, it is a very strict requirement. Furthermore, the resulting conditions of the above transformations are rather large. Often, it suffices, if a rule preserves a constraint. The motivation behind this relation is to reduce the size of the condition that has to be checked dynamically, i.e. each time the rule is applied and to check instead the properties of the first input graph initially.

For every constraint and every rule, there exists a left application condition that preserves the constraint.

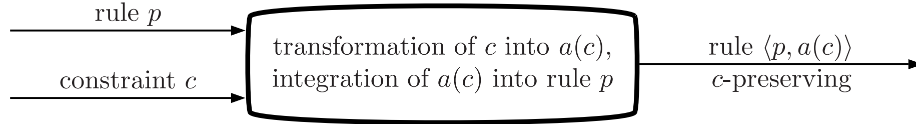


Figure 28: Integration of a constraint into a constraint-preserving rules

Corollary. (constraint-preserving integration) Given a rule p and a constraint c. Then there are left application conditions $a(c)$ and $a'(c)$ such that, for all direct derivations $G \Rightarrow_{p,m,m^*} H$ with m, m^* in M,

$$\text{(I)} \quad m \models a(c) \Leftrightarrow G \models c \text{ implies } H \models c$$

$$\text{(II)} \quad m \models a'(c) \Leftrightarrow G \models c \text{ if and only if } H \models c.$$

Proof. Let $T_L(c)$ $[T_R(c)]$ be the left [right] application condition of c and $T_p(T_R(c))$ the left application condition of $T_R(c)$. Let $a(c) = T_L(c) \Rightarrow T_p(T_R(c))$ and $a'(c) = T_L(c) \Leftrightarrow T_p(T_R(c))$. Then we have $G \models c$ implies $H \models c$ iff $\neg(G \models c) \vee H \models c$ iff $\neg(m \models T_L(c)) \vee m^* \models T_R(c)$ iff $\neg(m \models T_L(c)) \vee m \models T_p(T_R(c))$ iff $m \models \neg T_L(c) \vee T_p(T_R(c))$ iff $m \models a(c)$. Analogously, one obtains the second statement. □

It seems that $T_L(c) \Rightarrow T_p(T_R(c))$ is an even more complex application condition. But the premise does not have to be checked, if the rules applied before, preserve the satisfaction of c, especially, if c was integrated into each rule of the transformation system in the above manner. On the contrary, several subconditions of the conclusion can now be eliminated, as example 15 shows.

Example 15. (constraint-preserving application conditions) Up to now, it was demonstrated how a constraint can be transformed into a left application conditions for a rule such that it is satisfied if and only if the result graph will satisfy

the constraint. While this is a useful relation for e.g. safety critical systems, it is a very strict one. Often, one intuitively seeks a weaker relation, i.e. an application condition of the form: the input graph satisfies the constraint implies the result graph will satisfy the constraint, too.

Consider the rule `Move` in example 2 and the left application condition $T(\text{NoTwo}) = \wedge_{e \in \mathcal{E}} \neg \exists v$ of example 13. Let $m\colon L \to G$ and assume that G satisfies NoTwo. A subcondition of $T(\text{NoTwo})$ can be eliminated, if it contradicts this assumption, i.e. if there exists an injective morphism $i\colon \text{Two} \to V$. It remains a single negative application condition as depicted in figure 29.

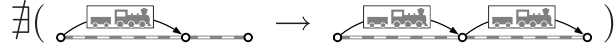


Figure 29: NoTwo-preserving left application condition for `Move`

Example 16. (application conditions, part II) Consider figure 11 of example 7. Constraint-preserving subconditions $a_1, \ldots, a_5$ are depicted below in figure 30. For a secure railway control system, replace rule `Add` with $\langle \texttt{Add}, a_1 \wedge a_2 \rangle$, `Move` with $\langle \texttt{Move}, a_3 \wedge a_4 \rangle$ and `Switch` with $\langle \texttt{Switch}, a_5 \rangle$. The system will always

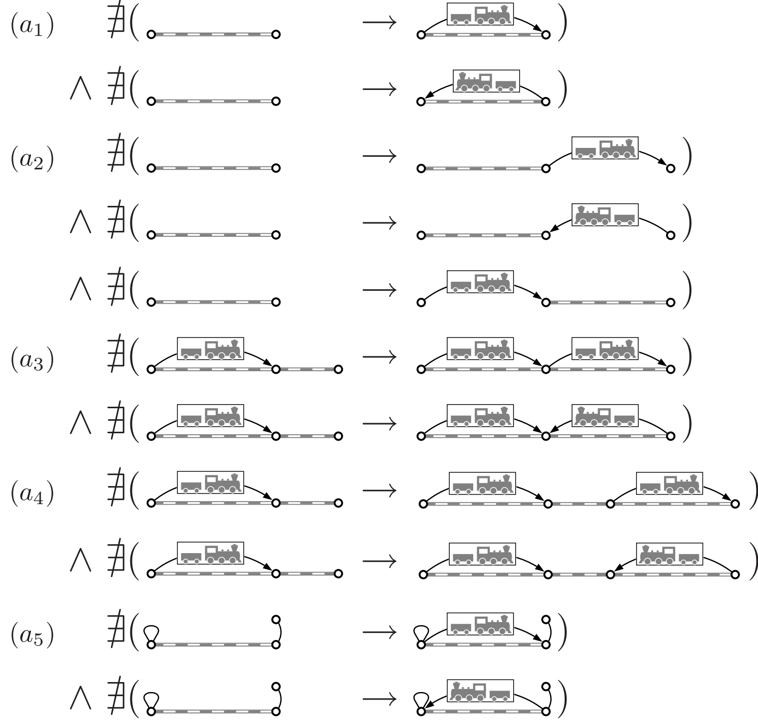


Figure 30: Application conditions for a secure railroad control system

be in a secure state, if the system starts with an input graph that satisfies all constraints $c_1 - c_3$, e.g. an initial waypoint.

Also of interest would be the transformation of application conditions back into constraints. Such a transformation could be used to infer knowledge about rules, which would be useful for verification purposes: E.g., true could be transformed subsequently into a left and right application condition for a rule and then back into a constraint. Unfortunately, there is no transformation from application conditions into constraints in general: Application conditions can express local properties of the subgraph surrounding the range of a match, constraints cannot.

Definition 9. An application condition a is called *non-trivial*, if there is a graph G such that there exist two morphisms m_1, m_2 with codomain G, such that $m_1 \models a$ and $m_2 \not\models a$.

Example 17. (non-trivial application condition) Let $a = \neg\exists(\bigcirc \to \bigcirc\!\!\circlearrowleft)$. Let $G = \bigcirc \;\; \bigcirc\!\!\circlearrowleft$. There are two morphisms $m_1, m_2 \colon \bigcirc \to G$, one which satisfies a, while the other does not.

Lemma 12. For non-trivial application conditions a, there does not exist a constraint $c(a)$ such that, for all matches $m\colon L \to G$ in M, $m \models a$ iff $H \models c(a)$.

Proof. Assume a is a non-trivial application condition, i.e. m_1, $m_2\colon L \to G$ are two morphisms such that $m_1 \models a$ and $m_2 \not\models a$. Assume there exists a constraint $c(a)$ such that, for all matches $m\colon L \to G$ in M, $m \models a \Leftrightarrow G \models c(a)$. Then $m_1 \models a$ and $G \models c(a)$ and $m_2 \not\models a$ and $G \not\models c(a)$. Thus, $G \models c(a)$ and $G \not\models c(a)$. Contradiction. □

7 Conclusion

Constraints and application conditions were considered by a number of authors, e.g. [8, 15, 18, 22, 6]. First only basic constraints and application conditions with Boolean operations were investigated, later on also simple conditional application conditions.

To classify the different notions of constraints and application conditions as provided by their authors, we will speak of n-nested constraints and conditions, which can be defined straightforward. true, false and every basic constraint are 0-nested constraints. Every conditional constraint $\forall(x, c)$ and $\exists(x, c)$ is $n+1$-nested, if c is an n-nested constraint. Every Boolean constraint $\wedge_{i \in I} c_i$ and $\vee_{i \in I} c_i$ is $n+1$-nested, if for some $i \in I$, c_i is an n-nested constraint and there does not exists an m-nested constraint c_j with $j \in I$, such that $m > n$.

	basic	conditional	Boolean
Heckel-Wagner [18]	yes	no	no
Koch-Parisi-Presicce [22]	yes	no	no
Ehrig-Ehrig-Habel-Pennemann [6]	yes	no	(yes)[†]
this paper / Habel-Pennemann [17]	yes	n-nested	yes

Table 3: Classification of constraints

	basic	conditional	Boolean
Ehrig-Habel [8]	yes	no	no
Habel-Heckel-Taentzer [15]	yes	no	(yes)[†]
Heckel-Wagner [18]	yes	1-nested	no
Koch-Parisi-Presicce [22]	yes	no	no
Ehrig-Ehrig-Habel-Pennemann [6]	yes	1-nested	yes
this paper / Habel-Pennemann [17]	yes	n-nested	yes

Table 4: Classification of application conditions

This thesis continues the work done in [6]. It generalizes the notions of constraints and application conditions to nested ones. The presented notions allows n-nested constraints and application conditions with arbitrary $n \geq 0$. Additionally, transformations of constraints and application conditions into normal form are considered and an elimination theorem is provided that allows to eliminate superfluous subconstraints and subconditions, respectively.

It is shown that nested constraints can be transformed into equivalent right application conditions, and that nested right application conditions can be transformed into equivalent left application conditions. The complexity of these two transformations is investigated with respect to the size of the resulting application conditions. For the first transformation, a slightly simpler construction is presented than in [17], and a shorter proof of its correctness is given.

[†]In [6], only Boolean expressions on constraints are considered. E.g., the basic constraint "all nodes have an outgoing or incoming edge" is not expressible. In [15], only conjunctions of positive and negative basic application conditions are considered.

Two applications of these basic transformations are considered: The conversion of constraints into left application conditions, such that the integration of the resulting application conditions into a given rule guarantees, or alternatively, preserves the satisfaction of the constraints. As the resulting application conditions for a constraint guarantee are rather large, it is sometimes preferable "just" to preserve the satisfaction of a constraint. In general, the corresponding application conditions are smaller, after all redundancies are eliminated.

The notations and concepts are illustrated by a simple railroad system. The specification of a railroad system is given in terms of rail net graphs, constraints, simple rules and application conditions. The rules model the dynamic behaviour of the system, like the movement of trains, and the application conditions ensure the safety of the system. The integration of general rail net constraints into rail net application conditions is exemplarily studied for the movement of trains.

All results presented in this paper hold for adhesiv HLR categories with binary coproducts and epi-M-factorizations, that is, for every morphism there is an epi-mono-factorization with monomorphism in M [17]. Further topics are:

- The extension of the underlying first-order logic by adding counting quantifiers as proposed in [19, 26],
- an application to typed attributed graph transformation [12], already under investigation in [25],
- an application to graph-based specification of access control policies [22, 20],
- an implementation of the transformation of nested constraints and application conditions (e.g. in the AGG tool [13] or GraJ [2]).
- a comparison of constraints with other formalisms to describe graphs, e.g. typegraphs [3, 12].

References

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[2] Giorgio Busatto. GraJ: A system for executing graph programs in Java. Berichte aus dem Fachbereich Informatik, Universität Oldenburg.

[3] Andrea Corradini, Hartmut Ehrig, Michael Löwe, Ugo Montanari, and Julia Padberg. The category of typed graph grammars and its adjunctions with categories of derivations. In *Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 56–74. Springer-Verlag, 1996.

[4] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, pages 163–245. World Scientific, 1997.

[5] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, 1979.

[6] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Constraints and application conditions: From graphs to high-level structures. In *Graph Transformation (ICGT'04)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[7] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[8] Hartmut Ehrig and Annegret Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer-Verlag, Berlin, 1986.

[9] Hartmut Ehrig, Annegret Habel, Julia Padberg, and Ulrike Prange. Adhesive high-level replacement categories and systems. In *Graph Transformation (ICGT'04)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[10] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

[11] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph grammars: An algebraic approach. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180, Iowa City, 1973.

[12] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory of typed attributed graph transformation. In *Graph Transformation (ICGT'04)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[13] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG approach: Language and environment. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 551–603. World Scientific, 1999.

[14] Dov Gabbay. *Elementary Logics: A Procedural Perspective*. Prentice Hall Europe, 1998.

[15] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.

[16] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[17] Annegret Habel and Karl-Heinz Pennemann. Constraints and Application Conditions for High-Level Structures. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabi Taentzer, editors, *tHE BOOK on Formal Methods in Software and System Modeling*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[18] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars — a constructive approach. In *SEGRAGRA 95*, volume 2 of *Electronic Notes in Theoretical Computer Science*, pages 95–104, 1995.

[19] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1-3):86–104, 1986.

[20] Manuel Koch, Luigi Vincenzo Mancini, and Francesco Parisi-Presicce. A graph-based formalism for RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):332–365, 2002.

[21] Manuel Koch, Luigi Vincenzo Mancini, and Francesco Parisi-Presicce. Administrative scope in the graph-based framework. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 97–104. ACM Press, 2004.

[22] Manuel Koch and Francesco Parisi-Presicce. Describing policies with graph constraints and rules. In *Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 223–238. Springer-Verlag, 2002.

[23] Manuel Koch and Francesco Parisi-Presicce. Visual specification of policies and their verification. In *Proceedings of Fundamental Approaches to Software Engineering (FASE)*, pages 278–293. LNCS 2621, 2003.

[24] Bernd Mahr and Anne Wilharm. Graph grammars as a tool for description in computer processed control: A case study. In *Graph-Theoretic Concepts in Computer Science*, pages 165–176. Hanser Verlag, München, 1982.

[25] Roland Meyer. Graph transformation systems with time: Realtime constraints. Berichte aus dem Fachbereich Informatik, to appear, Universität Oldenburg, 2004.

[26] Mohamed Mosbah and Rodrigue Ossamy. A programming language for local computations in graphs: Logical basis. Technical report, University of Bordeaux, 2003.

[27] John L. Pfaltz and Azriel Rosenfeld. Web grammars. In *Int. Joint Confenence on Artificial Intelligence*, pages 609–619, 1969.

[28] Arend Rensink. Representing first-order logic by graphs. In *Graph Transformation (ICGT'04)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[29] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[30] Hans Jürgen Schneider. Chomsky-Systeme für partielle Ordnungen. Arbeitsbericht 3,3, Institut für Mathematische Maschinen und Datenverarbeitung, Erlangen, 1970.

[31] Uwe Schöning. *Logic for Computer Scientists*, volume 8. Birkhäuser, 1989.

[32] Anika Wagner. On the expressive power of algebraic graph grammars with application conditions. In *Proc. TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1995.

Index

application condition, 11

codomain, 6
complexity, 25, 33
constraint, 8
  basic, 8
  Boolean, 8
  conditional, 8
  n-nested, 39

domain, 6

elimination of
  application conditions, 28
  constraints, 27
equivalence, 8, 12

graph, 6
graph transformation rule, 6
graph transformation system, 6
guarantee of constraints, 35

implication, 12

m-equivalence, 12
m-implication, 12
morphism, 6
  in M, 6
  injective, 6
  surjective, 6

n-nested, 39
normal form, 15

preserving of constraints, 35

range, 6
rule, 6
rule application, 6

size of
  constraints, 25
  graphs, 25
  morphisms, 25
subconstraint, 12
substitution theorem, 14

transformation of constraints into right application conditions, 19
transformation right into left application conditions, 30

A Appendix: Equivalences

In this appendix, we present all equivalences used in the normal form and elimination results in section 3.

Fact 5. (Equivalences) Let $x\colon P \to C$. The following subconstraints and application conditions are equivalent:

(1)	$\neg\exists(x, \neg c) \doteqdot \forall(x, c)$	(6)	$Q(x, Q(y, c)) \doteqdot Q(y \circ x, c)$	$Q \in \{\forall, \exists\}$
(2)	$\exists(x, \text{true}) \doteqdot \exists(x)$	(7)	$Q(id_P, c) \doteqdot c$	$Q \in \{\forall, \exists\}$
(3)	$\forall(x, \text{false}) \doteqdot \neg\exists(x)$	(8)	$\exists(\text{id}_P) \doteqdot \text{true}$	
(4)	$\forall(x, \text{true}) \doteqdot \text{true}$	(9)	$\forall(x, c) \doteqdot \text{true}$	if $x \notin M$
(5)	$\exists(x, \text{false}) \doteqdot \text{false}$	(10)	$\exists(x, c) \doteqdot \exists(x) \doteqdot \text{false}$	if $x \notin M$

where c is in case (7) a constraint over P.

Proof. The proof follows immediately from the definition of satisfiability

(1) $m \models \neg\exists(x, \neg c)$
$\iff$ $\neg\exists q.q : C \to G \text{ in } M \wedge q \circ x = m \wedge q \models \neg c$
$\iff$ $\forall q.\neg(q : C \to G \text{ in } M \wedge q \circ x = m) \vee q \models c$
$\iff$ $\forall q.q : C \to G \text{ in } M \wedge q \circ x = m \Rightarrow q \models c$
$\iff$ $m \models \forall(x, c)$.

(2) $m \models \exists(x, \text{true})$
$\iff$ $\exists q.q\colon C \to G \text{ in } M \wedge q \circ x = m \wedge q \models \text{true}$
$\iff$ $\exists q.q\colon C \to G \text{ in } M \wedge q \circ x = m$
$\iff$ $m \models \exists x$.

(3) $\forall(x, \text{false}) \doteqdot^{(1)} \neg\exists(x, \text{true}) \doteqdot^{(2)} \neg\exists x$ where $\doteqdot^{(n)}$ denotes the use of equivalence (n).

(4) $m \models \forall(x, \text{true})$
$\iff$ $\forall q.q\colon C \to G \text{ in } M \wedge q \circ x = m \Rightarrow q \models \text{true}$
$\iff$ $m \models \text{true}$.

(5) $\exists(x, \text{false}) \doteqdot^{(1)} \neg\forall(x, \text{true}) \doteqdot^{(4)} \neg\text{true}$

(6) $m \models \exists(x, \exists(y, c))$
$\iff$ $\exists q.q\colon C \to G \text{ in } M \wedge q \circ x = m \wedge \exists q'.q'\colon C' \to G \text{ in } M \wedge q' \circ y = q \wedge q' \models c$
$\iff$ $\exists q'.q'\colon C' \to G \text{ in } M \wedge q' \circ y \circ x = m \wedge q' \models c$
$\iff$ $m \models \exists(y \circ x, c)$.

Define $c = \neg d$. $\forall(x, \forall(y, c)) \doteqdot \neg\neg\forall(x, \forall(y, \neg d)) \doteqdot^{(1)} \neg\exists(x, \neg\forall(y, \neg d)) \doteqdot^{(1)} \neg\exists(x, \exists(y, d)) \doteqdot^{(6)} \neg\exists(y \circ x, d) \doteqdot^{(1)} \forall(y \circ x, \neg d) \doteqdot \forall(y \circ x, c)$.

(7) $m \models c$
$\iff$ $\exists m'\colon P \to G \text{ in } M \wedge m' \circ \text{id}_P = m \wedge m' \models c$
$\iff$ $m \models \exists(id_P, c)$

Define $c = \neg d$. $\forall(id_P, c) \doteqdot \neg\neg\forall(id_P, \neg d) \doteqdot^{(1)} \neg\exists(id_P, d) \doteqdot^{(7)} \neg d \doteqdot^{(1)} c$.

(8) $m \models \exists(\text{id}_P)$
$\iff$ $\exists q.q\colon P \to G \text{ in } M \wedge q = q \circ \text{id}_P = m$
$\iff$ $m \models \text{true}$.

(9) Let $x \notin M$. Assume $m \models \forall(x,c)$. Then $\forall q.q\colon C \to G$ in M with $q \circ x = m$ holds $q \models c$. Assume there exists such a q. Then m, q in M implies x in M. Contradiction.

(10) Let $x \notin M$. Assume $m \models \exists(x)$ [$\exists(x,c)$]. Then $\exists q.q\colon C \to G$ in M with $q \circ x = m$ [such that $q \models c$]. But m, q in M implies x in M. Contradiction. □

In the following a number equivalences for constraints are presented. For the proof, we use fact 2 of section 3 which states that it suffices to show the m-equivalence [m-implication] for two existential or universal subconstraints, to conclude the equivalence [implication] as constraints. The other direction does not hold.

Fact 6. For two existential [universal] constraints c_1, c_2 over P holds: $c_1 \dot{\Rightarrow} c_2$ implies $c_1 \Rightarrow c_2$ and $c_1 \dot{\equiv} c_2$ implies $c_1 \equiv c_2$.

Fact 7. (Equivalences of constraints) The following constraints are equivalent:

(1)	$\neg\exists(x,\neg c) \equiv \forall(x,c)$	(7)	$Q(x,Q(y,c)) \equiv Q(y \circ x, c)$	$Q \in \{\forall,\exists\}$
(2)	$\exists(x,\text{true}) \equiv \exists(x)$	(8)	$\exists(o,c) \equiv c$	if $c = \forall(x,d)$
(3)	$\forall(x,\text{false}) \equiv \neg\exists(x)$	(9)	$\forall(o,c) \equiv c$	if $c = \exists x$ or $\exists(x,d)$
(4)	$\forall(x,\text{true}) \equiv \exists P$	(10)	$\forall(x,c) \equiv \exists P$	if $x \notin M$
(5)	$\exists(x,\text{false}) \equiv \neg\exists P$	(11)	$\exists(x,c) \equiv \exists(x) \equiv \neg\exists P$	if $x \notin M$
(6)	$\exists(\text{id}_P) \equiv \text{true}$	(12)	$\text{NC}(x) \equiv \text{true}$	if $x \notin M$
		(13)	$\text{NC}(x) \equiv \text{NC}(\emptyset \to C)$	if $x \in M$
		(14)	$\text{NC}(\emptyset \to C) \equiv \neg\exists(\emptyset \to C)$	but $\text{NC}(x) \not\equiv \neg\exists(x)$

Proof. The proof immediately follows from the definition of constraints.

(1) $G \models \neg\exists(x,\neg c)$
$\iff \neg\forall p.p : P \to G \text{ in } M \Rightarrow p \models \exists(x,\neg c)$
$\iff \neg\forall p.\neg(p : P \to G \text{ in } M) \vee p \models \exists(x,\neg c)$
$\iff \exists p.p : P \to G \text{ in } M) \wedge p \not\models \exists(x,\neg c)$
$\iff \exists p.p : P \to G \text{ in } M \wedge p \models \forall(x,c)$
$\iff G \models \forall(x,c)$.

(2) $\exists(x,\text{true}) \equiv^{fact} \exists(x)$.

(3) $\forall(x,\text{false}) \equiv^{(1)} \neg\exists(x,\text{true}) \equiv^{(2)} \neg\exists x$

(4) $G \models \forall(x,\text{true})$
$\iff \exists p.p\colon P \to G \text{ in } M \wedge p \models \forall(x,\text{true})$
$\iff \exists p.p\colon P \to G \text{ in } M \wedge p \models \text{true}$
$\iff \exists p.p\colon P \to G \text{ in } M$
$\iff G \models \exists P \equiv \exists(\emptyset \to P)$.

(5) $\exists(x,\text{false}) \equiv^{(1)} \neg\forall(x,\text{true}) \equiv^{(4)} \neg\exists P \equiv \neg\exists(\emptyset \to P)$

(6) $G \models \exists(\text{id}_P)$
$\iff \forall p\colon P \to G \text{ in } M \text{ holds } p \models \exists(\text{id}_P)$
$\iff \forall p\colon P \to G \text{ in } M \text{ holds } p \models \text{true}$
$\iff G \models \text{true}$.

(7) $Q(x, Q(y, c)) \equiv^{fact} Q(y \circ x, c)$.

(8) $G \models \exists(o, c)$
$\iff$ for all $p: \emptyset \to G$ in M there exists $q: P \to G$ in M and $q \circ o = p$ and $q \models c$
$\iff$ for all $p: \emptyset \to G$ in M there exists $q: P \to G$ in M and $q \models c$
$\iff$ there exists a $q: P \to G$ in M such that $q \models c$
$\iff$ $G \models c$ and c is universal.

(9) $G \models \forall(o, c)$
$\iff$ there exists $p: \emptyset \to G$ in M and for all $q: P \to G$ in M with $q \circ o = p$ holds $q \models c$
$\iff$ there exists $p: \emptyset \to G$ in M and for all $q: P \to G$ in M holds $q \models c$
$\iff$ for all $q: P \to G$ in M holds $q \models c$
$\iff$ $G \models c$ and c is existential.

(10) Let $x \notin M$. $\forall(x, c) \equiv^{fact} \forall(x, \text{true}) \equiv^{(4)}$ true.

(11) Let $x \notin M$. $\exists(x, c) \equiv^{fact} \exists(x) \equiv^{fact} \exists(x, \text{false}) \equiv^{(5)}$ false.

(12) Let $x \notin M$. We can show: $\text{NC}(x) \equiv$ true (see EEHP04)

(13) Let $x \in M$. $G \models \text{NC}(x)$
$\iff$ $\forall p : P \to G$ in M $\nexists q : C \to G$ in M with $q \circ x = p$
$\iff$ $\nexists q : C \to G$ in M (if $\exists q$, define $p := q \circ x$ in M and contradict)
$\iff$ $\forall p : \emptyset \to G$ in M $\nexists q : C \to G$ in M with $q \circ x = p$
$\iff$ $G \models \text{NC}(\emptyset \to C)$

(14) $G \models \text{NC}(\emptyset \to C)$
$\iff$ $\forall p: \emptyset \to G$ in M $\nexists q: C \to G$ in M with $q \circ x = p$
$\iff$ $\neg(\exists q: C \to G \text{ in } M)$ (because $q \circ x = p$ is always true)
$\iff$ $\neg(\forall p: \emptyset \to G \text{ in } M\ \exists q: C \to G \text{ in } M \text{ with } q \circ x = p)$
$\iff$ $G \models \neg\exists(\emptyset \to P)$.

□

In section 3, we presented some implications which show that it is not possible to bring an arbitrary constraint into a form, such that every logical symbol is outside, or alternatively innermost. For the proof, we will use some of the following logical equivalences and implications:

Fact 8. (logical equivalences and implications) The following logical equivalences and implications hold for variables x and formulas F_1, F_2:

(1) $\exists x.F_1 \vee F_2 \iff \exists x.F_1 \vee \exists x.F_2$
(2) $\exists x.F_1 \wedge F_2 \implies \exists x.F_1 \wedge \exists x.F_2$
(3) $\forall x.F_1 \wedge F_2 \iff \forall x.F_1 \wedge \forall x.F_2$
(4) $\forall x.F_1 \vee F_2 \impliedby \forall x.F_1 \vee \forall x.F_2$
(5) $\exists x.F_1 \Rightarrow F_2 \implies \exists x.F_1 \Rightarrow \exists x.F_2 \iff \neg\exists x.F_1 \vee \exists x.F_2$
(6) $\forall x.F_1 \Rightarrow F_2 \implies \forall x.F_1 \Rightarrow \forall x.F_2 \iff \neg\forall x.F_1 \vee \forall x.F_2$
(7) $\forall x.F_1 \Rightarrow F_2 \implies \exists x.F_1 \Rightarrow F_2$

Proof. See [31]. □

Conjunctive and disjunctive symbols cannot jump over an arbitrary conditional constraint, i.e. $\vee_{i\in I}\,\mathrm{Q}(x, c_i) \not\equiv \mathrm{Q}(x, \vee_{i\in I}\,c_i)$. More precisely, only the following equivalences and implications hold.

Fact 9. (implications on constraints and applications) For application conditions resp. constraints, we have:

$\exists(x, \vee_{i\in I}\,c_i) \;\dot{\equiv}\; \vee_{i\in I}\,\exists(x, c_i)$	$\exists(x, \vee_{i\in I}\,c_i) \Leftarrow \vee_{i\in I}\,\exists(x, c_i)$
$\forall(x, \wedge_{i\in I}\,c_i) \;\dot{\equiv}\; \wedge_{i\in I}\,\forall(x, c_i)$	$\forall(x, \wedge_{i\in I}\,c_i) \Rightarrow \wedge_{i\in I}\,\forall(x, c_i)$
$\exists(x, \wedge_{i\in I}\,c_i) \;\dot{\Rightarrow}\; \wedge_{i\in I}\,\exists(x, c_i)$	$\exists(x, \wedge_{i\in I}\,c_i) \Rightarrow \wedge_{i\in I}\,\exists(x, c_i)$
$\forall(x, \vee_{i\in I}\,c_i) \;\dot{\Leftarrow}\; \vee_{i\in I}\,\forall(x, c_i)$	$\forall(x, \vee_{i\in I}\,c_i) \Leftarrow \vee_{i\in I}\,\forall(x, c_i)$

Proof. The proof follows immediately from the definitions of equivalence and implication. Let $x\colon P \to C$ and $m\colon P \to G$ in M. $m \models \exists(x, \vee_{i\in I}\,c_i)$ iff there exists $q\colon C \to G$ in M such that $q \circ x = m$ and $q \models \vee_{i\in I}\,c_i$ iff$^{(1)}$ there exists an $i \in I$ such that there exists $q\colon C \to G$ in M such that $q \circ x = m$ and $q \models c_i$ iff $\vee_{i\in I}\, m \models \exists(x, c_i)$ iff $m \models \vee_{i\in I}\,\exists(x, c_i)$. ($\forall(x, \wedge_{i\in I}\,c_i) \;\dot{\equiv}\; \wedge_{i\in I}\,\forall(x, c_i)$ analogue).

$m \models \exists(x, \wedge_{i\in I}\,c_i)$ iff there exists $q\colon C \to G$ in M such that $q \circ x = m$ and $q \models \wedge_{i\in I}\,c_i$ implies$^{(2)}$ for all $i \in I$ there exists $q\colon C \to G$ in M such that $q \circ x = m$ and $q \models c_i$ iff $\wedge_{i\in I}\, m \models \exists(x, c_i)$ iff $m \models \wedge_{i\in I}\,\exists(x, c_i)$. ($\forall(x, \vee_{i\in I}\,c_i) \;\dot{\Leftarrow}\; \vee_{i\in I}\,\forall(x, c_i)$ analogue).

For constraints, we show: $G \models \exists(x, \vee_{i\in I}\,c_i)$ iff for all $p\colon P \to G$ in M holds $p \models \exists(x, \vee_{i\in I}\,c_i)$ iff$^{\dot{\equiv}}$ for all $p\colon P \to G$ in M holds $p \models \vee_{i\in I}\,\exists(x, c_i)$ if$^{(4)}$ there exists an $i \in I$ such that for all $p\colon P \to G$ in M holds $p \models \exists(x, c_i)$ iff $\vee_{i\in I}\, G \models \exists(x, c_i)$ iff $G \models \vee_{i\in I}\,\exists(x, c_i)$. ($\forall(x, \wedge_{i\in I}\,c_i) \Rightarrow \wedge_{i\in I}\,\forall(x, c_i)$ analogue).

$G \models \exists(x, \wedge_{i\in I}\,c_i)$ iff for all $p\colon P \to G$ in M holds $p \models \exists(x, \wedge_{i\in I}\,c_i)$ implies$^{\dot{\Rightarrow}}$ for all $p\colon P \to G$ in M holds $p \models \wedge_{i\in I}\,\exists(x, c_i)$ iff$^{(3)}$ for all $i \in I$ and for all $p\colon P \to G$ in M holds $p \models \exists(x, c_i)$ iff $\wedge_{i\in I}\, G \models \exists(x, c_i)$ iff $G \models \wedge_{i\in I}\,\exists(x, c_i)$. ($\forall(x, \vee_{i\in I}\,c_i) \Leftarrow \vee_{i\in I}\,\forall(x, c_i)$ analogue). □

The following equivalences are needed in lemma 9 in section 4.4.

Fact 10. The following equivalences hold for constraints and application conditions.

(1) $\wedge_{i\in I}\,c_i \;\dot{\equiv}\; \wedge_{i\in I\setminus\{j\}}\,c_i$	if $c_k \;\dot{\Rightarrow}\; c_j$ for some $j \neq k$ and $j, k \in I$
(2) $\wedge_{i\in I}\,c_i \equiv \wedge_{i\in I\setminus\{j\}}\,c_i$	if $c_k \Rightarrow c_j$ for some $j \neq k$ and $j, k \in I$
(3) $\vee_{i\in I}\,c_i \;\dot{\equiv}\; \vee_{i\in I\setminus\{j\}}\,c_i$	if $c_j \;\dot{\Rightarrow}\; c_k$ for some $j \neq k$ and $j, k \in I$
(4) $\vee_{i\in I}\,c_i \equiv \vee_{i\in I\setminus\{j\}}\,c_i$	if $c_j \Rightarrow c_k$ for some $j \neq k$ and $j, k \in I$

Proof.

(1) $\dot{\Rightarrow}$: We have $m \models \wedge_{i\in I}\,c_i$ iff $m \models c_i$ for every $i \in I$ implies $m \models c_i$ for every $i \in I \setminus \{j\}$ iff $m \models \wedge_{i\in I\setminus\{j\}}\,c_i$.
$\dot{\Leftarrow}$: We have $m \models \wedge_{i\in I\setminus\{j\}}\,c_i$ iff $m \models c_i$ for every $i \in I \setminus \{j\}$. As $m \models c_k$ and $c_k \;\dot{\Rightarrow}\; c_j$ we conclude $m \models c_j$. Thus $m \models c_i$ for every $i \in I$ iff $m \models \wedge_{i\in I}\,c_i$.

(2) $\Rightarrow$: We have $G \models \wedge_{i\in I}\,c_i$ iff $G \models c_i$ for every $i \in I$ implies $G \models c_i$ for every $i \in I \setminus \{j\}$ iff $G \models \wedge_{i\in I\setminus\{j\}}\,c_i$.

$\Leftarrow$: We have $G \models \bigwedge_{i \in I \setminus \{j\}} c_i$ iff $G \models c_i$ for every $i \in I \setminus \{j\}$. As $G \models c_k$ and $c_k \Rightarrow c_j$ we conclude $G \models c_j$. Thus $G \models c_i$ for every $i \in I$ iff $G \models \bigwedge_{i \in I} c_i$.

(3) $\dot{\Rightarrow}$: We have $m \models \bigvee_{i \in I} c_i$ iff $m \models c_i$ for some $i \in I$. Assume $m \models c_j$. Then $c_j \dot{\Rightarrow} c_k$ implies $m \models c_k$ for some $k \in I$ with $k \neq j$. Thus $m \models c_i$ for some $i \in I \setminus \{j\}$ iff $m \models \bigvee_{i \in I \setminus \{j\}} c_i$.
$\dot{\Leftarrow}$: We have $m \models \bigvee_{i \in I \setminus \{j\}} c_i$ iff $m \models c_i$ for some $i \in I \setminus \{j\}$ iff $m \models c_i$ for some $i \in I$ iff $m \models \bigvee_{i \in I} c_i$.

(4) $\Rightarrow$: We have $m \models \bigvee_{i \in I} c_i$ iff $m \models c_i$ for some $i \in I$. Assume $m \models c_j$. Then $c_j \Rightarrow c_k$ implies $m \models c_k$ for some $k \in I$ with $k \neq j$. Thus $m \models c_i$ for some $i \in I \setminus \{j\}$ iff $m \models \bigvee_{i \in I \setminus \{j\}} c_i$.
$\Leftarrow$: We have $m \models \bigvee_{i \in I \setminus \{j\}} c_i$ iff $m \models c_i$ for some $i \in I \setminus \{j\}$ iff $m \models c_i$ for some $i \in I$ iff $m \models \bigvee_{i \in I} c_i$.

$\square$