CARL
VON
OSSIETZKY
universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Proving correctness of graph programs relative to recursively nested conditions

Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften

vorgelegt von

Dipl.-Inf. Nils Erik Flick

Datum der Disputation: 21.10.2016
Gutachter: Prof. Dr. Annegret Habel, Prof. Dr. Barbara König

Abstract

Graph programs provide a formal way to model the behaviour of a wide range of discrete systems. These programs are an extension of graph rewriting with control structures (sequence, nondeterministic choice and iteration).

This thesis presents a theoretically founded formalism for specifying properties of graph programs and a proof-based approach to verifying the partial correctness of a graph program with respect to a precondition and a postcondition.

First, a novel specification language, namely recursively nested conditions (or μ-conditions) is introduced which can express non-local state properties and is proven to be distinct from comparable formalisms.

The verification approach consists of two parts:

- an adaptation of Dijkstra's weakest precondition calculus to graph programs and μ-conditions,
- a proof calculus for μ-conditions, whose core part is a rule schema for inductive refutation.

Comparisons with other relevant formalisms are presented. Several additional parts are elaborated within the same framework, such as a formulation of correctness under adversity and structure-changing Petri nets.

Zusammenfassung

Graphprogramme bieten die Möglichkeit, vielerlei Arten diskreter Systeme formal zu modellieren. Es handelt sich bei diesen Programmen um Graphersetzungssysteme, erweitert um Kontrollstrukturen (Sequenz, nichtdeterministische Auswahl und Iteration).

In dieser Arbeit wird ein theoretisch begründeter Formalismus zum Spezifizieren von Eigenschaften von Graphprogrammen eingeführt, außerdem ein beweisbasiertes Verfahren zum Nachweisen der partiellen Korrektheit von Graphprogrammen bezüglich einer Vor- und einer Nachbedingung.

Zunächst wird eine neue Spezifikationssprache eingeführt, nämlich rekursiv geschachtelte Graphbedingungen (μ-Bedingungen), die geeignet ist, nichtlokale Zustandseigenschaften auszudrücken und die sich nachweislich von vergleichbaren Formalismen unterscheidet.

Der Verifikationsansatz besteht aus zwei Teilen:

- einer Übertragung von Dijkstra's Kalkül der schwächsten Vorbedingungen auf Graphprogramme und μ-Bedingungen
- einem Beweiskalkül für μ-Bedingungen, dessen Kernstück ein Regelschema für induktive Widerlegung ist.

Vergleiche mit mehreren anderen Formalismen werden angestellt. In den weiteren Teilen der Arbeit werden zusätzliche Betrachtungen innerhalb des gleichen Frameworks präsentiert, beispielsweise Korrektheit von Graphprogrammen unter widrigen Umständen und strukturveränderliche Petrinetze.

Acknowledgements

I dedicate this thesis to the memory of Manfred Kudlek, humanist and polymath, who left this world too soon.

I thank all my family for their unwavering, confident support.

I thank the thesis committee: Annegret Habel for sharing wisdom, ideas, resources, and patience during the entire time. Barbara König for reading, discussing and finally grading this work, providing in-depth feedback on all its parts. Ernst-Rüdiger Olderog for running a correctness-centered research training group and making me feel at home in Oldenburg. Astrid Rakow for agreeing to join the committee when my project neared completion and then providing criticism that proved essential during defense. I thank Eike Best and Arend Rensink for their insightful comments.

My sincere gratitude also goes to Ira Wempe for removing all obstacles from my path, and to Andrea Göken and Marion Bramkamp, Jörg Lehners and Patrick Uven for making things work really smoothly. I also would like to thank the numerous unnamed reviewers who provided helpful feedback on my submitted papers.

I thank all members of SCARE, supervisors and fellow doctorands, roughly in order or appearance: Mohamed Abdelaal, Yang Gao, Evgeny Erofeev, Dilshod Rahmatov, Saifullah Khan, Sören Jeserich, Thomas Strathmann, Md Habibur Rahman, Peilin Zhang, Peter Nazier Mosaad, Heinrich Ody, Uli Schlachter, Awais Usman, Maike Schwammberger. I thank Christian Kuka, Christoph Peuser, Dennis Kregel especially for proofreading. I also thank my non-SCARE colleagues Jan-Steffen Becker, Mani Swaminathan, Sven Linker, Martin Hilscher, Tim Strazny, Stephanie Kemper, Manuel Gieseking, as well as Hans-Jörg Kreowski and Berthold Hoffmann at Uni Bremen, for their advice.

Last but not least I thank fellow SCAREian Björn Engelmann, with whom I shared many a train ride poring over tricky problems, for his famously thorough proofreadings, and Hendrik Radke for a shared appreciation of science and intricate wordplay. And of course everyone whose names I may have forgotten to mention.

This thesis was prepared using 100% free and open source software, as it should always remain possible. A huge thank you to the community that made this possible through dedication and hard work, valiantly and as a matter of course staving off the encroaching walls. For without freedom of thought, our dreams have no meaning.

Contents

1. Introduction

"To seek it with thimbles, to seek it with care;
To pursue it with forks and hope;
To threaten its life with a railway-share;
To charm it with smiles and soap!
— Lewis Carroll, *The Hunting of the Snark*:
Fit the Fourth (The Hunting)

Contents

This thesis deals with methods and aspects of formally proving the *correctness* of *graph programs*. It aims at addressing problems of the following abstract form:

$$\mathbf{Asm} \vdash (\mathbf{Sys} \| \mathbf{Env}) \;\; \mathbf{sat} \;\; \mathbf{Spec}$$

Under formalised assumptions **Asm**, a system (model) **Sys** is composed ($\|$) with an environment model **Env** which describes the faults or modifications that are outside the influence of the controller. Given these data, the task is to verify the correctness of (**Sys**$\|$**Env**) with respect to **Spec** under **Asm**. This means that formal deduction is applied in order to derive either a proof that the formal specification **Spec** is indeed satisfied (**sat**), or a counterexample where the system does not behave as specified.

To ensure correctness under *adverse conditions*, one has to contend with difficulties:

1. limited knowledge
2. unpredictable behaviour
3. changing system environment and structure

We concentrate on points 2 (unpredictable behaviour) and 3 (changing structure). Our scope in this thesis is restricted to discrete, nondeterministic (point 2) systems that represent changing structures (point 3). In a major part of the thesis, the abstract notion of system correctness is instantiated (given a precise formal sense) and worked out in the setting of graph programs. Graph programs play the role of the system model **Sys** and **Env**, graph conditions make up the specification **Spec** and a classical notion of program correctness is adapted for **sat**, then extended to adverse conditions.

The formalism of graph programs serves to describe discrete systems whose states may be modelled as entities (nodes) linked by edges. It generalises graph transformation systems which in turn generalise Petri nets. Graph programs allow an abstract treatment of graph-like aspects of ordinary imperative programs, heap operations being expressed as graph transformations, and also the direct modelling of many discrete systems. Graph programs come with a notion of interface, which determines the (variable) part of the state that is currently subject to rewriting. Our goal is to provide a theoretical foundation for specifying properties of such systems subject to adverse conditions as described above, and a proof-based approach to verifying these properties.

To this aim, existing work on the correctness of graph programs, namely a Dijkstra-style verification approach, is extended. Correctness is understood with respect to specifications consisting of pre- and postconditions as graph conditions. These are graphical expressions akin to formulas of graph logics. Part of the thesis is dedicated to the examination of the expressivity of new kinds of graph properties, and the decidability of classes of properties that are interesting from a modelling point of view, i.e. whether a given system model, under a set of assumptions (**Asm**), satisfies (**sat**) a given specification (**Spec**) of correct system behaviour. Non-local state assertions (recursively nested, or μ-conditions) that can express the existence of paths, amongst other interesting properties, are of interest in verifying graph programs that model e.g. operations on common data structures such as search trees. A weakest precondition calculus and a proof calculus are provided for the novel formalism. A more special instantiation of the general problem using structure-changing Petri nets is also presented, but these can be subsumed under the more general graph programs.

To address adverse conditions, it is important to understand how to formalise an integrated view of the interplay of the environment and the system. It would not be reasonable to impose a sequentialisation or alternation on system operation and faults, as this does not do justice to the distributed functioning of large systems or the unpredictability of environment action. Our original intuition is that **Env** and **Sys** can be understood as actors playing a game. Both parts are composed in parallel ($\|$) and interact only via the shared state. To progress toward that goal, we introduce two-player programs and extend the notion of weakest precondition to these. The two-player programs contain a mix of steps belonging either to **Env** or **Sys**. No assumption is made on the scheduling of these steps and the formalism is flexible enough to leave it entirely to the modeller.

Finally, several examples are provided to demonstrate the whole method, from the weakest precondition computations to the application of the proof calculus. The steps of the proofs are discussed in detail in the corresponding appendix.

1.1. Contributions

In this thesis, we first investigate language-theoretic notions of correctness and found that even for **structure-changing Petri nets** (as an intuitive and apparently simple special case of graph programs), some properties one would be interested in are already undecidable while others can be decided, but only for very restricted subclasses.

The chief contributions of this thesis are a theory of **recursively nested conditions**, together with newly developed methods for proving the correctness of programs relative to these conditions. Our work on graph conditions includes a **weakest precondition calculus** that allows graph programs to be proved correct with respect to μ-conditions by means of logical deduction using a sound **proof calculus**, significantly expanding upon existing work in this direction, and basic results on extending the method to correctness proofs for programs under **adverse conditions**.

Specifically, the contributions are as follows:

- Part of the proofs, algorithms and presentation (especially of Proposition 6) of these parts is joint work (electronic journal reference [FE15]): Section 2.3 and Chapter 4.
- The work on recursively nested conditions is original (electronic journal reference [Fli16]): Chapter 3 except Section 3.4 and the whole of Chapter 5 are issued from that publication.
- The material in Section 3.4, Chapter 6 and Chapter 7 is completely new and hitherto unpublished.

Chapter E of the appendix contains a bibliography of papers (co)authored by the author of this thesis during the thesis project, in inverse chronological order. The papers that are part of the work presented here are marked with an asterisk. The rest is mostly work on formal languages:

∗ Nils Erik Flick, *Correctness of Graph Programs Relative to Recursively Nested Conditions* [Fli16]

Nils Erik Flick, *Quotients of Unbounded Parallelism* [Fli15]

∗ Nils Erik Flick and Björn Engelmann, *Analysis of Petri Nets with Context-Free Structure Changes* [FE15]

Björn Engelmann, Ernst-Rüdiger Olderog and Nils Erik Flick, *Closing the Gap – Formally Verifying Dynamically Typed Programs like Statically Typed Ones Using Hoare Logic – Extended Version* [EOF15]

Manfred Kudlek and Nils Erik Flick, *Properties of Languages with Catenation and Shuffle* [KF14]

Manfred Kudlek and Nils Erik Flick, *A Hierarchy of Languages with Catenation and Shuffle* [KF13]

Nils Erik Flick, *Derivation Languages of Graph Grammars* [Fli13]

1.2. Outline

Chapter 2 introduces graph transformation systems and graph programs and Chapter 3 introduces recursively nested conditions. Chapter 4 instantiates the idea of correctness in typical ways, introducing so-called structure-changing Petri nets. Chapter 5 combines graph conditions with the correctness of graph programs. Chapter 6 explains our ideas on the interaction between system and environment. Chapter 7 concludes the main part by providing small case studies. Chapter 8 concludes with an outlook. Each chapter has an appended bibliography section that lists our sources and situates our work in the research landscape. Figure 1.1 shows the logical dependencies between the chapters.

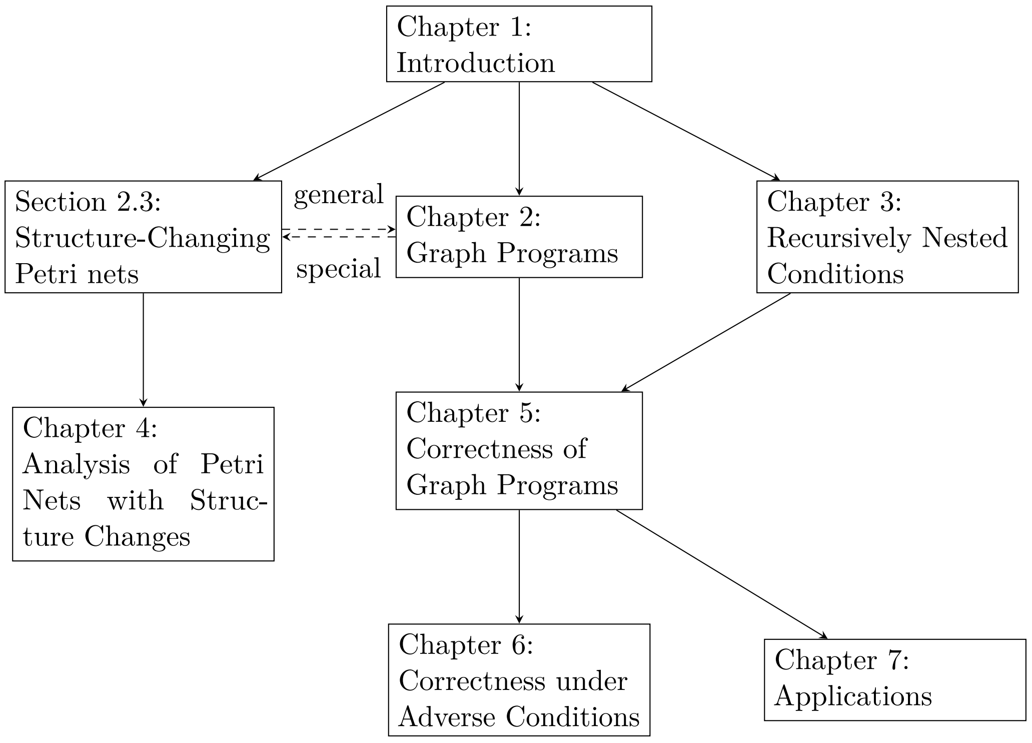

Chapter 1: Introduction
Section 2.3: Structure-Changing Petri nets
general
special
Chapter 2: Graph Programs
Chapter 3: Recursively Nested Conditions
Chapter 4: Analysis of Petri Nets with Structure Changes
Chapter 5: Correctness of Graph Programs
Chapter 6: Correctness under Adverse Conditions
Chapter 7: Applications

Figure 1.1.: Logical dependencies between the chapters.

2. Graph Programs

We may even find out why the duck-billed platypus.
— Terry Pratchett, *The Last Continent*

Contents

In this chapter, we introduce the formalism used to describe the dynamics of system and environment. These *graph programs* are constructed from the elementary rewriting steps which underlie graph transformations in the sense of Ehrig et al. [EEPT06] (double-pushout graph transformation). This chapter is structured as follows: in Section 2.1, we define graphs and morphisms, Section 2.3 relates graph transformations and Petri nets. In Section 2.3, Structure-Changing Petri nets are introduced as a special case. We assume familiarity with the basic notions of category theory. Bibliographic notes for the whole chapter are given in Section 2.4.

2.1. Graphs and Morphisms

A *graph* consists of a set of vertices (or nodes) V, a set of edges E and two functions $s, t : E \to V$ (*source* and *target*). A *labelled graph* over the (node, edge) label alphabets Λ_V, Λ_E also has mappings $l_V : V \to \Lambda_V$, $l_e : E \to \Lambda_E$. In this text, the sets V and E (and any label alphabets) are always assumed to be finite. Label alphabets are assumed to be fixed within each construction. Unlabelled graphs can be considered to be labelled graphs over singleton alphabets, or defined without the labelling functions.

Notation. *The empty graph is denoted by $\emptyset$. An edgeless graph is called* discrete*.*

Graphs are related by morphisms: when $G = (V_G, E_G, s_G, t_G, l_{vG}, l_{eG})$ and $H = (V_H, E_H, s_H, t_H, l_{vH}, l_{eH})$ are graphs labelled over (Λ_V, Λ_E), a morphism $f : G \to H$ is a pair of functions $(f_V : V_G \to V_H, f_E : E_G \to E_H)$ such that the following conditions hold:

- $f_V \circ t_G = t_H \circ f_E$
- $f_V \circ s_G = s_H \circ f_E$

- $(l_V)_H = (l_V)_G \circ f_V$, $(l_E)_H = (l_E)_G \circ f_E$.

A morphism is said to be *injective* (*surjective*, *bijective*) if both of its components are. Nodes and edges are collectively called *items.* The *identity* morphism on a graph G, id_G, is the identity on nodes and on edges. A pair of morphisms $(f : F \to H, g : G \to H)$ is said to be *jointly surjective* when every item in their common target is the image of some item in one of the sources.

Notation. *The domain and codomain of a morphism* $f : G \to H$ *are denoted by* $\mathrm{dom}(f) = G$ *and* $\mathrm{cod}(f) = H$. *Injective morphisms (also known as monomorphisms or monos*[1]*) are distinguished typographically by a hooked arrow* $f : G \hookrightarrow H$ *and* $f : G \subseteq H$ *if both components are set inclusions, while double-ended arrows* $f : G \twoheadrightarrow H$ *denote surjective ones (also known as epimorphisms or epis*[2]*). We use the symbol* $\mathcal{M}$ *to denote the class of all graph monomorphisms and the symbol* $\mathcal{A}$ *for all graph morphisms. If both components of a monomorphism* f *are set inclusions, then we write* $f : G \subseteq H$ *and call* f *an* inclusion. *An* isomorphism *of graphs* $f : G \cong H$ *is a morphism which is both a mono- and an epimorphism. This induces an equivalence relation.* $f\mid_X$ *is the restriction of the function* $f : Y \to Z$ *to* $X \subseteq Y$. *It is the function with domain* X *that coincides with* f *on every element of* X*:* $\forall x \in X, f(x) = f\mid_X(x)$. *Clearly, this determines* $f\mid_X$ *uniquely by extensionality.*

In this thesis, graphical notations are used. Graphs are depicted using little circles (nodes, possibly colored to indicate node labels) joined by arrows (edges, possibly decorated with a symbol to indicate edge labels). The arrows going from the source node to the target node. Morphisms are represented graphically by drawing morphism arrows between pictures of graphs. While the layout itself has no formal meaning, we generally try to place the image of each node or edge in the same relative place, but sometimes this is not possible. If the convention has to be violated or there is any question of ambiguity, then little blue numbers are used to specify the images. Figure 2.1 depicts an injective morphism between two graphs.

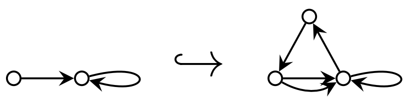


Figure 2.1.: A morphism $G \hookrightarrow H$.

2.2. Graph Transformation and Graph Programs

Next, we introduce graph transformations. We follow the double pushout approach with injective rules and injective matches, present the definitions in a set-theoretical way. No

[1]A category theoretical notion that coincides with the notion of injective morphisms in our case.
[2]Dito, for surjective morphisms.

knowledge of category theory is needed to understand the definitions.[3]

The basic device used in graph transformation is the graph transformation *rule*. A rule relates a *left hand side* graph (the subgraph to be replaced) and a *right hand side* graph:

Definition 1 (**Rule**). *A graph transformation rule, short* rule, *is a tuple $\varrho = (L \hookleftarrow K \hookrightarrow R)$ of graphs, where K is called the* interface *and is contained both in the* left hand side *graph L and in the* right hand side *graph R. A rule ϱ is called* identical *and denoted by id_K if $L = K = R$. A* graph transformation system *is a set of rules, usually finite.*

In the application of a rule, certain nodes and edges will be added, others deleted and the rest preserved. We describe the effect set-theoretically, where $-$ denotes set difference and $+$ denotes a disjoint union. Rules are applied to graphs as described below (for notational convenience, the injective morphisms of the rule can be chosen to be graph inclusions, and so can those of the transformed graphs):

Definition 2 (**Direct Derivation Step**). *Let ϱ be a rule, G a graph and $g : L \hookrightarrow G$ an injective morphism. The* dangling condition *requires that for every node $n \in V_G$ in the range of g_V that is* not *the image of a node in K, every edge $e \in E_G$ with $s_G(e) = n$ or $t_G(e) = n$ is in the range of g_E. If g satisfies the dangling condition, then g is called a* match *of L in G. A* direct derivation step *$G \overset{\varrho,g}{\Rightarrow} H$ from a graph G to a graph H with a rule ϱ via a match g consists of a* double-pushout *diagram:*

$$\begin{tikzcd}
L \arrow[d, hook, "g"'] \arrow[r, phantom, "(1)" description, shift right=20] & K \arrow[l, hook', "\supseteq"'] \arrow[r, hook, "\subseteq"] \arrow[d, hook] & R \arrow[d, hook, "h"] \\
G & D \arrow[l, hook', "\supseteq"] \arrow[r, hook, "\subseteq"'] & H
\end{tikzcd}$$

$$(1) \qquad (2)$$

where h is also injective and

$$(1)\ D = G - g(L - K), \qquad (2)\ H = D + h(R - K).$$

The equation (1) *is an abbreviation for $V_D = V_G - g_V(V_L - V_K)$ and $E_D = E_G - g_E(E_L - E_K)$,* (2) *means $V_H = V_D + h_V(V_R - V_K)$ and $E_H = E_D + h_E(E_R - E_K)$. The labels of the items of D are inherited from G, and so are sources and targets:*

for node and edge labels respectively, $(l_V)_D = (l_V)_G |_{V_G}$ and $(l_E)_D = (l_E)_G |_{E_G}$; sources and targets of edges are determined by $s_D = s_G |_{E_D}$ and $t_D = t_G |_{E_D}$ while the dangling condition guarantees that the codomains of the functions s_D and t_D are indeed subsets of V_D and thus the construction defines a graph.

$(l_V)_H$ is defined as follows: for any node $x \in V_H$ which is the image $h_V(h_V^{-1}(x))$ of some node $h_V^{-1}(x) \in V_R$ (which is unique because of the injectivity of h_V), the label $(l_V)_H(x)$

[3]However, we make heavy use of the categorial framework in many proofs, because it simplifies the proofs a lot. Some known theorems are used but not established. In those cases, a source is given where the proof can be found.

is defined as $(l_V)_R(h_V^{-1}(x))$*. Otherwise,* x *must by definition of* V_H *be in* $V_D \subseteq V_H$ *and then* $(l_V)_H(x) = (l_V)_D(x)$*.* $(l_E)_H$ *is defined analogously for edges.*

$(s_V)_H$ *is defined as follows: for any edge* $x \in E_H$ *which is the image* $h_E(h_E^{-1}(x)$ *of some edge* $h_E^{-1}(x) \in E_R$ *(which is unique because of the injectivity of* h_E*), if* $(s_V)_R(h_E^{-1}(x)) \in V_R - V_K$*, then* $(s_V)_H(x) = h_V((s_V)_R(h_E^{-1}(x)))$*. Otherwise,* $(s_V)_R(h_E^{-1}(x)) \in V_K \subseteq V_D$ *and* $(s_V)_H(x)$ *is defined as* $(s_V)_R(h_E^{-1}(x))$*. If* x *is not the image via* h_E *of an edge in* E_R*, then it is an element of* E_D*. In this case,* $(s_V)_H(x) = (s_V)_D(x)$*. The definition of* $(t_V)_H$ *is exactly analogous.*

We may omit the match in the notation: $G \overset{\varrho}{\Rightarrow} H$*.*

Figure 2.2 shows a direct derivation step. As in the diagram of the definition, the top row represents the rule. The graph in the bottom left hand corner (G) is transformed into the graph in the bottom right hand corner (H) by the rule. There are two possible matches, because there are two edges between the bottom two nodes in the graph G, both yielding the same result.

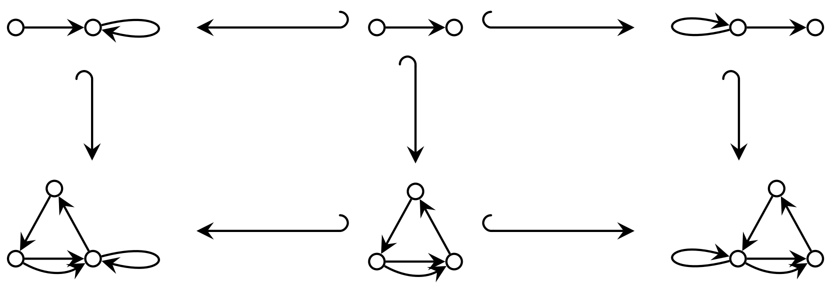


Figure 2.2.: A direct derivation step.

The squares (1), (2) can be shown to be *pushout* squares in the sense of category theory, see [EEPT06]. A pushout of $f : A \to B$ and $g : A \to C$ is a graph D with a pair of morphisms $f' : C \to D$ and $g' : B \to D$ such that $g' \circ f = f' \circ g$, subject to the *universal* condition that *any* pair $f'' : C \to D'$ and $g'' : B \to D'$ with $g'' \circ f = f'' \circ g$, there is a unique morphism $h : D \to D'$ with $h \circ f' = f''$ and $h \circ g' = g''$ (the pair (f', g') is then always jointly surjective). We will also make use of the notion of *pullback*, which is the categorial dual of a pushout (same definition, but with all directions of morphisms reversed) and *pushout complement*, which is a pair of morphisms that completes a pushout square as shown in Figure 2.3. That is, when $f : A \to B$ and $g' : B \to D$ can be completed to a pushout square with morphisms $g : A \to C$ and $f' : C \to D$, then we say that (g, f') is a pushout complement of (f, g').

In the category of graphs and morphisms, pushouts and pullbacks are guaranteed to exist while pushout complements are not. Nevertheless, the following holds:

Fact 1 (**Pushout Complements** [LS04, EEPT06])**.** *The pushout complement of graph morphisms (f, g'), if it exists, is uniquely determined provided that f is a monomorphism.*[4] *It exists provided that the dangling condition holds and, in the case of g' not being a monomorphism, the so-called identification condition must hold as well, which says that all non-injectively mapped items of* $\mathrm{dom}(g')$ *have preimages in* $\mathrm{dom}(f)$*.*

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "\text{(PO)}"] & B \arrow[d] \arrow[ddr, dotted, bend left] & \\
C \arrow[r] \arrow[drr, dotted] & D \arrow[dr, dashed] & \\
 & & D'
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, dotted, "g"'] \arrow[dr, phantom, "\text{(PO)}"] & B \arrow[d, "g'"] \\
C \arrow[r, dotted, "f'"'] & D
\end{tikzcd}$$

Figure 2.3.: Pushouts and pushout complements

A *partial monomorphism* is a pair of monomorphisms with the same domain. By abuse of notation, the partial monomorphism $(\mathrm{id}_{\mathrm{dom}(x)}, x)$ is also denoted x. The partial monomorphism $(x, \mathrm{id}_{\mathrm{dom}(x)})$ is also denoted x^{-1}. The class of all partial monomorphisms is denoted $\mathcal{PM}$. A partial monomorphism $(f : A \hookrightarrow B, g : A \hookrightarrow C)$ can be understood as first selecting part of B to be mapped (regarding f as an inclusion $f : A' \subseteq B$ for some graph $A' \cong A$) and then mapping that subgraph to C via g. Partial monomorphisms $p' = (l_1, r_1)$, $p'' = (l_2, r_2)$ compose as $p'; p'' = (l_1 \circ l'_2, r_2 \circ r'_1)$ using the pullback (r'_1, l'_2) of (r_1, l_2), as in Figure 2.4.

$$\begin{tikzcd}
 & & A_{1,2} \arrow[dl, dotted, "l'_2"'] \arrow[dr, dotted, "r'_1"] & & \\
 & A_1 \arrow[dl, "l_1"'] \arrow[dr, "r_1"] & \text{(PB)} & A_2 \arrow[dl, "l_2"'] \arrow[dr, "r_2"] & \\
B & & C & & E
\end{tikzcd}$$

Figure 2.4.: Composition of partial monomorphisms (l_1, r_1) and (l_2, r_2).

We re-define graph transformations in terms of four elementary steps, namely selection, deletion, addition and unselection. Deletion and addition always apply to a selected subgraph, and selection and unselection allow the selection to be changed. This approach

[4]The proof can be found in Lack and Sobociński [LS04], which introduces the more general setting of adhesive categories. Graphs and graph morphisms are shown in [LS04] to form an adhesive category. A thorough account based on both sets and categories is found in [EEPT06].

to defining graph programs is also called *programs with interface*, because the currently selected subgraph acts as a kind of *interface* in the sequential composition of programs. This intuition will be made precise when defining the semantics. The role of the interface is to ensure that an addition, for instance, can be performed in the same place as the deletion without introducing any special labels to mark that place, as one would be forced to do in plain graph transformation formalisms.

A graph transformation rule is then nothing else but the sequence of a selection; a deletion; an addition; an unselection. We also introduce *skip* as a no-op used in the definition of sequential composition. The definition below allows for somewhat more general combinations of the basic steps, which cannot be expressed as sets of graph transformation rules. Another reason for breaking up rules into more elementary steps is to make constructions and proofs easier to follow.

First, the syntax of graph programs is defined.

Definition 3 (**Graph Programs**). *Let x be a monomorphism.*

Then $\mathrm{Sel}(x)$*,* $\mathrm{Del}(x)$*,* $\mathrm{Add}(x)$*,* $\mathrm{Uns}(x)$ *are graph programs called the* selection*,* deletion*,* addition*,* unselection *of x, respectively.*

If P and Q are graph programs, so are their disjunction $P \cup Q$ *and* sequence $P;Q$*. If P is a graph program, so is its* iteration P^**.*

The semantics of a graph program is a triple of two monomorphisms and one partial monomorphism. The two monomorphisms, called *input interface* and *output interface*, represent the selected subgraphs before and after the execution of the program respectively, while the partial monomorphism records the changes effected by the program. Our programs are a proper subset of the *programs with interfaces* in Pennemann [Pen09], and use the same semantics. Note that some program steps (those leaving the codomain of the interface unchanged) change only the selection without modifying the graph while others modify both the selection and the graph.

Definition 4 (**Semantics of Graph Programs**). *In the following table, x, l, r, y, m_{in} and m_{out} are monomorphisms, with x, l, r and y arbitrarily chosen to define a program step, while the* input interface m_{in} *and the* output interface m_{out} *are universally quantified in the set comprehensions that appear in the definitions below.*

Note that each triple $(m_{in}, m_{out}, (p_l, p_r))$ must have $\mathrm{cod}(p_l) = \mathrm{dom}(m_{in})$ *and* $\mathrm{cod}(p_r) = \mathrm{dom}(m_{out})$ *for the relevant compositions of morphisms to exist; there are no restrictions on the domains and codomains other than those implicit in the postulated compositions and pushout squares.*

Name	*Program P*	*Semantics* $[\![P]\!]$
selection	$\mathrm{Sel}(x)$	$\{(m_{in}, m_{out}, x) \mid m_{out} \circ x = m_{in}\}$
deletion	$\mathrm{Del}(l)$	$\{(m_{in}, m_{out}, l^{-1}) \mid \exists l', (m_{out}, l, m_{in}, l') \text{ pushout}\}$
addition	$\mathrm{Add}(r)$	$\{(m_{in}, m_{out}, r) \mid \exists r', (m_{in}, r, m_{out}, r') \text{ pushout}\}$
unselection	$\mathrm{Uns}(y)$	$\{(m_{in}, m_{out}, y^{-1}) \mid m_{out} = m_{in} \circ y\}$
skip	*skip*	$\{(m, m, id_{\mathrm{dom}(m)}) \mid m \in \mathcal{M}\}$

The semantics of disjunction is a set union $[\![P \cup Q]\!] = [\![P]\!] \cup [\![Q]\!]$ *and the semantics of sequence is* $[\![P;Q]\!] = \{(m, m', p) \mid \exists (m, m'', p') \in [\![P]\!], (m'', m', p'') \in [\![Q]\!], p = p';p''\}$*, where* $(p';p'')$ *is the composition of the partial monomorphisms; the semantics of iteration is* $[\![P^*]\!] = \bigcup_{j\in\mathbb{N}} [\![P^j]\!]$ *where* $P^j = P;P^{j-1}$ *for* $j \geq 1$ *and* $P^0 = skip$*.*

Programs P and Q can be sequentially composed to a compound program but on the semantics side, the composition has a result only in the cases where the output interface of an element of $[\![P]\!]$ matches the input interface of an element of $[\![Q]\!]$.

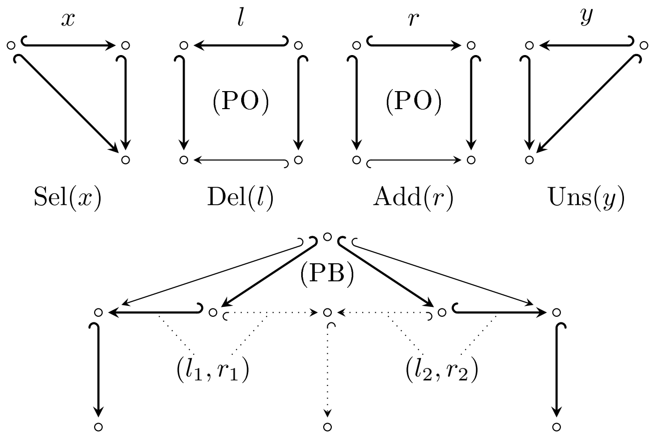
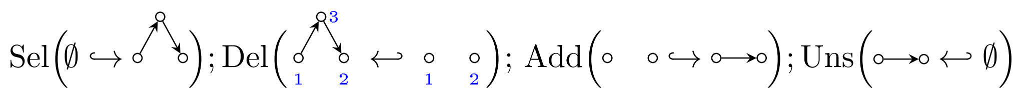


Figure 2.5.: Illustration of the semantics of the basic graph programs and sequential composition.

Remark 1. *The definitions generalise the state transitions in plain graph transformation: a rule* $\varrho = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ *is exactly simulated by the program* $\mathrm{Sel}(\emptyset \hookrightarrow L); \mathrm{Del}(l); \mathrm{Add}(r); \mathrm{Uns}(\emptyset \hookrightarrow R)$*.*

The disjunction $P \cup Q$ of graph programs is a nondeterministic choice between P and Q. Nondeterminism also appears in the semantics of selection steps. Sequential composition inherits the associativity of the composition of partial monomorphisms, which is easily seen from the categorial constructions used.

Example 1 (A graph program simulating a rule)**.**



This graph program selects a subgraph of three nodes and two edges, then removes one of the nodes and both edges, then glues in a new edge between the two preserved nodes, then forgets its selection.

2.3. Structure-Changing Petri nets

The well-known Petri nets [Pet62], especially the unbounded place/transition (P/T) nets (cf. Desel [DR98]), have been regarded as graph transformation systems in their own right. Usually, one understands them as rewriting systems that operate on labelled discrete graphs, where nodes correspond to tokens, node colours correspond to places and transitions are not explicitly represented in the graph (they become graph transformation rules). Another possibility is to model the net structure explicitly, opening up the option of rewriting that structure using supplementary rules.

P/T nets can be converted to graphs by turning places and tokens into nodes. Firing of transitions can then be simulated with graph transformation rules. Transitions can be explicitly represented as nodes, which allows reconfigurations to be modelled as graph transformation rules as well, see Figure 2.6. For formal accounts of the encodings of Petri nets as graph transformation, we refer to the bibliographic notes section at the end of this chapter and of Chapter 4. Petri nets are system models where resource tokens are moved around on an immutable underlying structure. They originally lacked a notion of structure change or reconfiguration, but several structure-changing extensions have been formulated. In this section, we define Petri nets together with transition replacement rules similar to graph replacement rules [EEPT06], as a simple instantiation of what is to be our general framework. Graph transformations are well suited for rewriting the net structure. In this section, we introduce structure-changing Petri nets, which offer dynamic *context-free* rewriting of Petri nets. The intention is to describe a formalism midway between Petri nets and graph transformation.

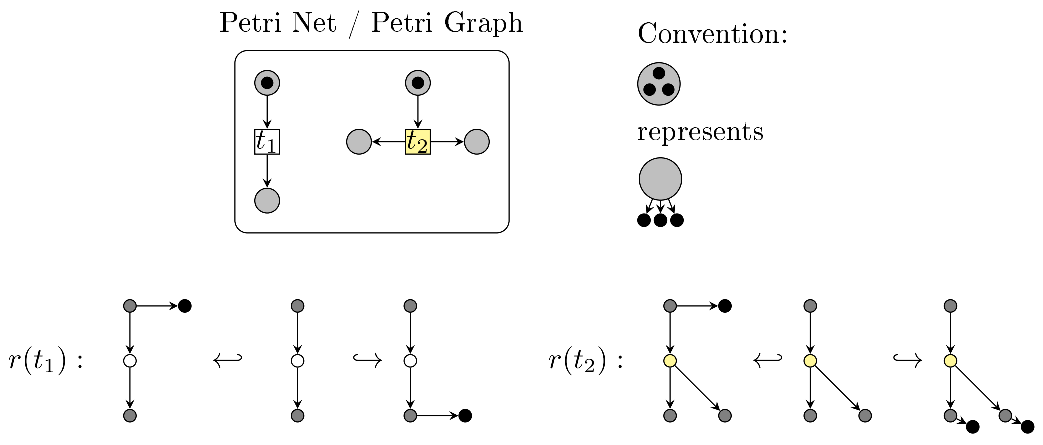


Figure 2.6.: Encoding P/T nets as graph transformations

Petri nets are popular in the context of workflow modelling [vdA97], where labelled transitions correspond to tasks to be executed. An important property of workflow nets is *soundness*, which is decidable (in the absence of extra features such as reset arcs), and intuitively means the workflow can always terminate correctly and there are no useless transitions. Plain workflow nets, however, lack the ability of representing dynamic evolution; this shortcoming has been recognised by the workflow modelling community [vdA01, WRRM08]. Combining some Petri nets with graph transformations promises a solution for modelling dynamic change in workflows.

We extend Petri nets in the sense of [DR98, PW02] to structure-changing Petri nets. In such a system, structure-changing rules interfere with the behaviour of a Petri net. Reconfigurations occur unpredictably, modelling the influence of an uncontrolled environment, such as the dynamic addition of a component, or the unexpected complication or iteration of a task.

Notation. *We use* $-$ *and* $+$ *to denote set difference and disjoint set union, respectively, and* $\mathbb{N}$ *for the set of natural numbers including* 0*. The cardinality of a set* X *is denoted by* $|X|$*, the length of a sequence* w *is also denoted* $|w|$*, and* $|w|_Y$ *denotes the number of occurrences of symbols from* $Y \subseteq X$ *in* $w \in X^*$*. The symbol* ϵ *denotes the empty word.*

Assumption 1. *There are two disjoint alphabets* Σ *and* R *of transition labels and rule names, respectively.*

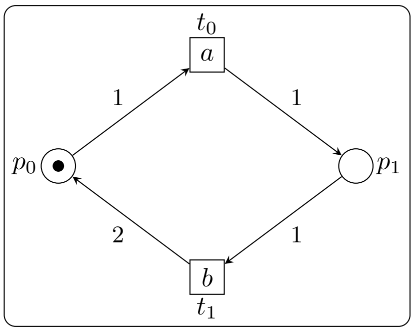
We introduce structure-changing Petri nets with coloured places.

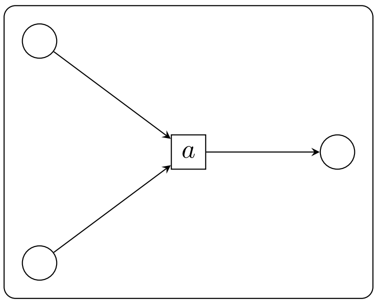
Definition 5 (Petri Nets). *A* Petri net*, briefly called* net*, is a tuple* (P, T, F^-, F^+, l, c) *where* P *is a finite set of* places*,* T *is a finite set of* transitions *with* $P \cap T = \emptyset$*,* $F^-, F^+ : T \times P \to \mathbb{N}$ *are functions assigning* preset *and* postset arc multiplicities*, respectively,* $l : T \to \Sigma$ *is the* labelling *function,* $c : P \to \mathbb{N}$ *is the* colouring *function. A* marked net *is a pair* $\mathcal{N} = (N, M)$*, where* N *is a net and* $M : P \to \mathbb{N}$ *is called the* marking *of* $\mathcal{N}$*.*

Places and transitions are collectively called *items*. A net is said to be *(strongly) connected* if it is (strongly) connected as a graph, regarding arcs as directed edges. A net is *acyclic* if it is acyclic as a graph. *Isomorphism*, denoted $N \cong N'$ (resp. $\mathcal{N} \cong \mathcal{N}'$) of (marked) nets means that there are bijections between the respective place and transition sets that preserve $F^{\pm}$, l and c (and M). The *size* of a net is $|P| + |T|$. A connected net with a single transition is a *1-net*.

Notation. *When* $F^-(t, p) = k$*, one says there is an arc of weight* k *from* p *to* t*. Likewise, when* $F^+(t, p) = k$*, there is an arc of weight* k *from* t *to* p*. We write* ${}^\bullet t$ *for* $\{p \in P \mid F^-(t, p) > 0\}$ *and* $t^\bullet$ *for* $\{p \in P \mid F^+(t, p) > 0\}$*. When* $M(p) = k$*, it means that* p *is marked with* k *tokens. When* $c(p) = k$*, we say that* p *has colour* k*. The components of a net named* N_x *will likewise be named* P_x *and so on. If there is no subscript, they will be referenced as* P *and so on. Likewise, the marked net* (N_x, M_x) *is usually referred to as* $\mathcal{N}_x$*. The pictorial representation of nets as graph-like diagrams is well known, and a translation to graph grammars has been formalised in [Kre81] and [Cor95].*

Example 2 (Graphical representation of a marked net; a 1-net).

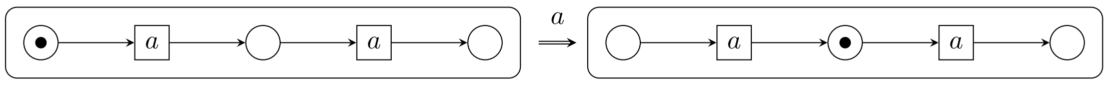




Let N be a net. The transition $t \in T$ is *enabled* by the marking M iff for all places p, $F^-(t,p) \leq M(p)$. The successor marking Mt of M via t is then defined by $\forall p \in P$, $(Mt)(p) = M(p) - F^-(t,p) + F^+(t,p)$, and $(N, M) \overset{t}{\Rightarrow} (N, Mt)$ is called a *firing step*.

For a sequence $u \in T^*$, the marking Mu is defined recursively as $M\epsilon := M$ and $Mau := (Ma)u$. The sequence u is said to be *enabled* in M iff Mu is defined.

Example 3 (**A firing step**).



Throughout this chapter, all replacement rules are of a simple context-free kind that replace a single transition by an unmarked net. The adjective context-free is understood in the sense of hyperedge replacement [Hab92], which is also called context-free replacement. In our case, transitions correspond to hyperedges.

Definition 6 (**Context-free Rule**). *A context-free* rule *(short:* rule*) is a tuple* (ϱ, N_l, N_r) *where* $\varrho \in R$ *is the* rule name*,* N_l *is a 1-net, and* N_r *is a net with* $P_l \subseteq P_r$*, and* $\forall p \in P_l, c_l(p) = c_r(p)$*.* N_l *is the* left hand side *and* N_r *the* right hand side *of the rule.*

A *match* of the 1-net N_l in the net N is a mapping $m : P_l \cup \{t_l\} \to P \cup T$ such that m maps the sole transition t_l of N_l to a transition such that $l(m(t_l)) = l_l(t_l)$ and the places $p_l \in P_l$ to places in P such that $\forall p \in P_l, c(m(p)) = c_l(p)$. The notion is extended to matches in marked nets: a *match* of N_l in (N, M) is a match of N_l in N. A *match* of the rule (ϱ, N_l, N_r) on a marked net $\mathcal{N}$ is a match of N_l in N. An *abstract application*, with match m, of ϱ to a marked net $\mathcal{N}$ is a pair $(\mathcal{N}, \mathcal{N}')$ such that if t_l is the transition in N_l, $T' = T - \{m(t_l)\} + T_r$, $P' = P + (P_r - P_l)$, M' coincides with M on the places from P and has value 0 otherwise. The place colours are as in N and N_r,

$$F'^{\pm}(t,p) = \begin{cases} F^{\pm}(t,p) & t \in T, p \in P \\ F_r^{\pm}(t,p) & t \in T_r, p \in (P_r - P_l) \\ F_r^{\pm}(t,p') & t \in T_r, p = m(p') \\ 0 & \text{otherwise.} \end{cases}$$

$\mathcal{N} \overset{\varrho}{\Rightarrow} \mathcal{N}'$ or $\mathcal{N} \overset{\varrho,m}{\Rightarrow} \mathcal{N}'$ is called a *replacement step.*

Note that we always assume $P_r \cup T_r$ to be disjoint from $P \cup T$ in a replacement step (hence the notation "+" instead of "∪" in the above definition). Formally, in a replacement step N_l and N_r are replaced with isomorphic copies whose items do not occur in $P \cup T$, and these copies are used in the rule application.[5]

Remark 2. *Given $\mathcal{N}$, ϱ and a m, the resulting net $\mathcal{N}'$ is uniquely determined.*

The following example shows how a rule is applied to yield a replacement step:

Example 4 (A replacement step)**.** *Figure 2.7 shows a replacement step induced by a rule ϱ and a match m. The top row is the rule. The net in the top left hand corner is the left hand side, which is identified as a subnet in the bottom left hand net. The bottom right hand net is the result:*

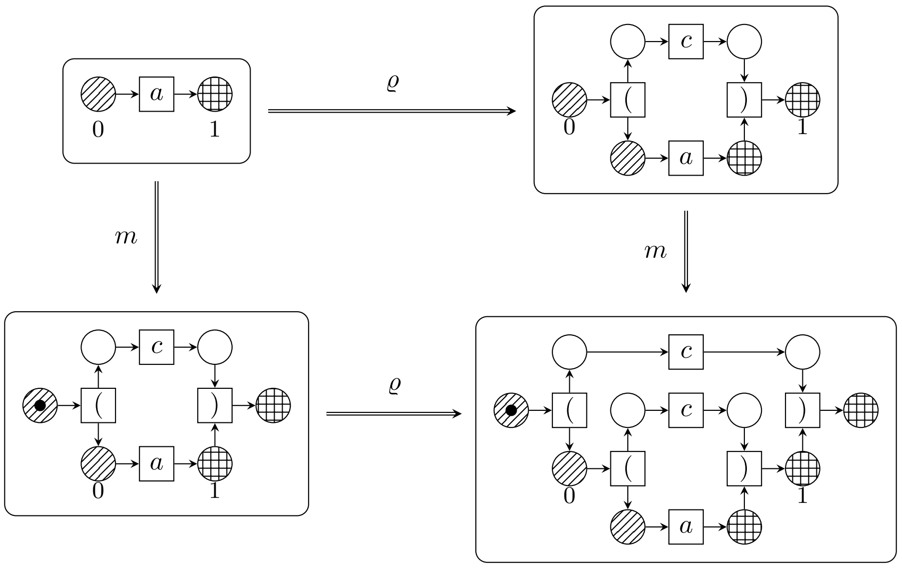


Figure 2.7.: A rule ϱ and a replacement step.

A *step* is either a firing step or a replacement step. We define four functions for extracting the information from a step: for a firing step e, let $\tau(e) := t$, $\lambda(e) := l(t)$, $\text{from}(e) := (N, M)$ and $\text{to}(e) := (N, M')$. For a replacement step, let $\tau(e) := m(t_l)$, $\lambda(e) := \varrho$, $\text{from}(e) := \mathcal{N}$ and $\text{to}(e) := \mathcal{N}'$. The functions to, from, τ and λ canonically extend to sequences.

Definition 7 (Structure-changing Petri nets). *A* structure-changing Petri net *is a tuple $\mathcal{S} = (\mathcal{N}, \mathcal{R})$, where $\mathcal{N}$ is a marked net and $\mathcal{R}$ is a finite set of rules.*

[5]The use of isomorphic copies should not cause a lot of confusion in our setting and we will not stress this issue further.

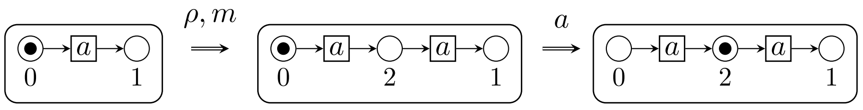
Remark 3. *Every marked net $\mathcal{N}$ may be seen as a structure-changing Petri net $(\mathcal{N}, \emptyset)$ with empty rule set, which will be called a* static *net and by abuse of notation also denoted $\mathcal{N}$. We will not distinguish between $\mathcal{N}$ and $(\mathcal{N}, \emptyset)$.*

A derivation consists of a sequence of steps linking a sequence of marked nets:

Definition 8 (**Derivation**)**.** *A* derivation *of length $n \in \mathbb{N}$ in a structure-changing Petri net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$ is a pair (ξ, σ) of a sequence $\xi_0 \dots \xi_{n-1}$ of steps and a sequence $\sigma_0 \dots \sigma_n$ of marked nets such that* $\mathrm{to}(\xi_i) = \sigma_{i+1}$*,* $\mathrm{from}(\xi_i) = \sigma_i$ *for all $i \in \{0, \dots, n-1\}$ and $\mathcal{N} = \sigma_0$. We write $\sigma_0 \overset{\xi}{\Rightarrow}_{\mathcal{R}} \sigma_n$ and say that (ξ, σ)* starts in *$\mathcal{N} = \sigma_0$ and* ends in *σ_n.*

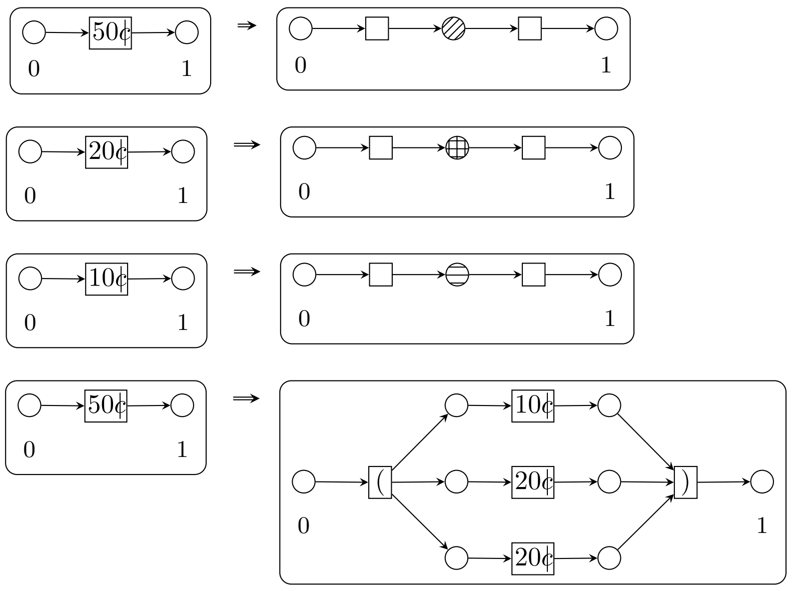
In the following example, the small numbers serve to track places throughout the derivation.

Example 5 (A structure-changing Petri net derivation.)**.**



Another simple example is a vending machine which either returns 50 cent pieces or smaller change, where place colours represent denominations of coins. Its rules are:

Example 6 (Rules for a vending machine.)**.**



A marked net $\mathcal{N}'$ is said to be reachable in $\mathcal{S}$ from $\mathcal{N}$ iff there is a derivation in $\mathcal{S}$ that starts in $\mathcal{N}$ and ends in $\mathcal{N}'$. The marked nets reachable in $\mathcal{S}$ are also called (reachable) *states* of $\mathcal{S}$. We write $\mathcal{RS}(\mathcal{S})$ for the set of all reachable states of $\mathcal{S}$. The reachability graph is the transition system induced by the reachability relation, identifying isomorphic marked nets. In a static net, instead of derivations one considers transition sequences, as they uniquely determine derivations. A net, rule or structure-changing Petri net is *k-coloured* if the highest colour assigned to any place does not exceed $k-1$. A marked net is *m-safe* if any reachable marking has at most m tokens on any of its places.

Remark 4. *Note that the place colouring does not affect the behaviour of the net at all. It will be used in Section 4.1 to specify abstract markings.*

We introduce workflow nets [vdA97], extended by structure-changing rules, are introduced as a special case of structure-changing Petri nets.

Definition 9 (**Workflow Net**). *A* workflow net *is a tuple (N, p_{in}, p_{out}) consisting of a net N and a pair of distinguished places $p_{in}, p_{out} \in P$, the input and output place which have no input respectively no output arcs, subject to the requirement that adding an extra transition from p_{out} to p_{in} would render the net strongly connected.*

The data (p_{in}, p_{out}) need not be made explicit, since these places are easily seen to be uniquely determined in a workflow net. Thus we are justified in treating a workflow net as a special net. The *start marking* of a workflow net N, i.e. the marking where only p_{in} is marked with one token and all other places are not marked, is denoted by ${}^\bullet N$. The *end marking* where only p_{out} is marked with exactly one token is denoted by $N^\bullet$. A workflow net N is *sound* iff from any marking reachable from ${}^\bullet N$, $N^\bullet$ is reachable, and for each transition t there is some marking M reachable from ${}^\bullet N$ such that t is enabled in M.

Definition 10 (**Structure-changing workflow net**). *A* structure-changing workflow net *is a structure-changing Petri net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$ such that*

(1) $\mathcal{N}$ is a sound workflow net marked with its start marking.

(2) for every rule $(\varrho, N_l, N_r) \in \mathcal{R}$, N_l and N_r are sound workflow nets; N_l has two places, one transition, arc weights 1.

(3) $\sum_{t \in T_r} F_r^+(t, p_{out}) = \sum_{t \in T_r} F_r^-(t, p_{in}) = 1$ and the input (resp. output) place of N_r is p_{in} (p_{out}).

Condition (2) implies that only single-input, single-output 1-nets are permitted as left hand sides. The role of condition (3) is to avoid certain complications that otherwise arise in the analysis. Although the restrictions rule out markings with multiplicities, it is now possible to create more instances of a subnet by replacing transitions. Structure-changing workflow nets can still capture situations such as workflows that undergo complications as they are executed, as in Example 6.

Condition (3) in our opinion does not unduly restrict modelling capacity because a right hand side that does not fulfil it can be adapted to yield a similar behaviour while fulfilling (3), as shown in Figure 2.8. In the example, the only difference is that transition t must fire before t_0, t_1, t_2 become activated. The same is done for the output place. It is clear that this construction is applicable in all such cases and will have very limited, predictable effects on the behaviour of the system.

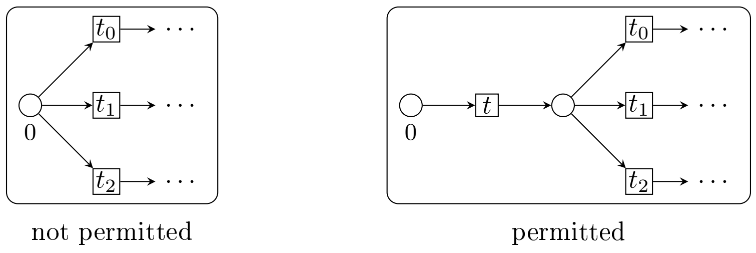


Figure 2.8.: Satisfying condition (3) in the structure-changing workflow net definition with an extra transition.

Every reachable state of a *structure-changing workflow net* is a reachable state of some workflow net. A structure-changing workflow net is called *acyclic* if every reachable state is an acyclic net. It is called *1-safe* if every reachable state is marked with at most one token per place.

2.4. Bibliographic Notes

For an in-depth treatment of the theory of graph transformations, we kindly refer the reader to the monograph of Ehrig et al. [EEPT06]. The definitions presented in this chapter are not the most general possible: in this thesis, we stick to the concrete case of graphs rather than the axiomatic framework of adhesive categories.

For encoding Petri nets as graph transformations, there are several sources such as Corradini [Cor95] and Kreowski [Kre81, KW86]. These encodings have different advantages. Proper care must be taken to reflect concurrent semantics, as researchers have noted [Cor95]. More recent unified treatments focus on the rewriting of Petri nets and the relation of graph rewriting and Petri nets, such as the article of Maximova [MEE12].

For the rewriting of graph-like objects, there are more general formulations such as Ehrig and Prange [EP06], other algebraic approaches founded on category theory such as the single-pushout (Ehrig et al. [EHK$^+$97]) and sesqui-pushout (Corradini et al. [CHHK06]) approach as well as the older approaches which are not founded on category theory (Engelfriet and Rozenberg [ER97]). Delzanno et al. [DS14] study verification problems for reconfigurable systems with whole neighborhood operations, which have somewhat different properties from ours (these systems lend themselves to well-structured analysis for coverability properties). This thesis, however, is exclusively based on the double-pushout approach.

3. Recursively Nested Conditions

'No', said Geraldine, 'that's Lucy. I'm not only your assistant's assistant's sub-assistant, but also the assistant to the assistant to your personal assistant's assistant.'
'Wait', I said, thinking hard, 'that must make you your own assistant.'
— Jasper Fforde, *The Woman who Died a Lot*

Contents

We propose a new specification language for the proof-based approach to verification of graph programs by introducing recursively nested conditions (short μ-conditions) as an alternative to existing formalisms. It will be demonstrated that recursively nested conditions can express many non-local properties of interest.

Our formalism is an extension of *nested conditions* by recursive definitions. While other formalisms can also express many properties of interest, we claim that the recursively nested conditions defined in this thesis offer a viewpoint sufficiently different from existing ones to be worth investigating. The theory of recursively nested conditions offers a weakest precondition calculus that can handle any condition expressible in it (Section 5.1); as opposed to HR* conditions, there is also a proof calculus; as compared M-conditions, which relies more heavily on expressing properties directly in (monadic second-order) logic, ours is more closely related to nested conditions and shares the same basic methodology.

This chapter is structured as follows: Section 3.1 describes the state of the art, Section 3.2 recalls nested graph conditions, Section 3.3 introduces our extension of nested conditions, in Section 3.4 we characterise their expressive power and compare it to other formalisms, Section 3.5 gives concluding remarks and an outlook and Section 3.6 gives bibliographic notes for the chapter.

3.1. State of the Art

The (finite) *nested conditions* or *nested graph conditions* of Habel et al. [HPR06, HP09] are expressions that specify graph properties. They can quantify over subgraphs, contain logical connectives and be nested (hence the name). Nested conditions are known to express the same properties of finite graphs as first-order logic. The rationale for using graph conditions instead of logical formulae facilitates the expression of certain transformations that occur in the so-called predicate transformer approaches to the verification of graph programs.

Nested conditions are well-established by now and can often be used as a drop-in replacement for negative application conditions and other weaker forms of graph condition. As the number of uses for graph conditions grew, nested conditions have been found to be insufficient for some applications: via the equivalence to first order logic, it is known that finite nested conditions can only express properties which are *local* in the logical sense (cf. Gaifman [Gai82]), but application domains often call for the ability to express, for instance, the existence of paths of arbitrary length in a graph, which is a classical example of a non-local property.

To manipulate infinite objects algorithmically, the infinite objects must have a finite representation that is appropriate to compute the necessary transformations. Several extensions of nested graph conditions to non-local conditions have been devised, notably by Radke [Rad13, Rad16] and Poskitt and Plump [PP14].

3.2. Nested Graph Conditions

In this thesis, we build upon a slightly modified version of nested graph conditions, equivalent to those in [Pen09][1]

Definition 11 (**Nested Graph Conditions**)**.** *Let* Cond *be the class of* infinitary nested graph conditions*, defined inductively as follows (where B, C', C are graphs):*

- *If J is a countable set and for all $j \in J$, c_j is a condition (over B), then $\bigwedge_{j \in J} c_j$ is a condition (over B).*
  This includes the case $J = \emptyset$ (for any B), which is the base case.
- *If c is a condition (over B), then $\neg c$ is also a condition (over B).*
- *If $a : B \hookrightarrow C'$ is a monomorphism, $\iota : C \hookrightarrow C'$ is a monomorphism and c' is a condition (over C), then $\exists(a, \iota, c')$ is a condition (over B).*

It is important to distinguish between the infinitary nested graph conditions and an important special case, the *finitary* nested conditions.

[1]For technical reasons, [Pen09] first defines nested conditions with disjunctions indexed over countable sets. These could of course "express" *any* property of finite graphs but these are not really graph conditions in the sense of admitting a *finite* expression. The distinction is made explicit here.

Definition 12 (**Finitary Nested Graph Conditions**). *Same definition as Definition 11, but with the case $\bigwedge_{j\in J} c_j$ restricted to* finite *index sets J only.*

The finitary nested graph conditions are usually called *finite* in the literature and sometimes just *nested graph conditions.* The current usage is introduced here to avoid confusion when defining μ-conditions. Indeed, infinitary nested graph conditions *only* appear as an auxiliary definition. For the purpose of comparing expressiveness later, the term "nested graph condition" will always mean a **finitary** nested graph condition.

Notation. *We call c' a* direct subcondition *of $\exists(\alpha,\iota,c')$, $\neg c'$ and $\bigwedge_{j\in J} c_j$ if one of the c_j is c', and use the term* subcondition *to mean the reflexive and transitive closure of this syntactically defined relation. If c is a condition over B, then B is its* type[2] *and we write $c : B$, and Cond_B is the class of all conditions over B. The usual abbreviations define the other standard operators: $\bigvee_{j\in J} c_j$ is $\neg\bigwedge_{j\in J}\neg c_j$, $\forall$ is $\neg\exists\neg$. All morphisms satisfy the conjunction over the empty index set. To avoid a proliferation of special cases, we introduce $\top$ (true) as a notation for it, and $\bot$ (false) for $\neg\top$. Hence technically for each graph B there is one $\top : B$ and one $\bot : B$. We use the abbreviation $\exists(a)$ for $\exists(a,\iota,\top)$, $\exists(a,c)$ for $\exists(a, id_{\mathrm{cod}(a)}, c)$ and $\exists^{-1}(\iota,c)$ for $\exists(id_{\mathrm{cod}(\iota)},\iota,c)$ (the notation is chosen to symbolise a morphism going the other way).*

The morphism ι serves to unselect[3] a part of C'. This will become necessary later.

Definition 13 (**Satisfaction**). *A monomorphism $f : B \hookrightarrow G$* satisfies *a condition $c : B$, denoted $f \models c$, iff $c = \top$, $c = \neg c'$ and $f \not\models c'$, or $c = \bigwedge_{j\in J} c_j$ and for all $j \in J$, $f \models c_j$, or $c = \exists(a,\iota,c')$ ($a : B \hookrightarrow C'$, $\iota : C \hookrightarrow C'$, $c' : C$) and there exists a monomorphism $q : C' \hookrightarrow G$ such that $f = q \circ a$ and $q \circ \iota \models c'$.*

$$\begin{tikzcd}
\exists B \arrow[r, hook, "a"] \arrow[d, hook, "f"'] & C' \arrow[dl, hook, "q"'] & C, \arrow[l, hook', "\iota"'] \arrow[dll, hook, "q\circ\iota"] \arrow[dll, phantom, "\models"', shift left=4] & \triangleleft\, c'\\
G & & &
\end{tikzcd}$$

A graph G satisfies a condition $c : \emptyset$ iff the unique (mono)morphism $\emptyset \hookrightarrow G$ satisfies c.

Notation. *As one can see in Fig. 3.1, the notation for graph conditions often only depicts source or target graphs of morphisms, in this case the source of a is left implicit at the outer quantifier $\exists(a,\iota,c)$. The numbers show the morphisms' node mappings.*

[2]When we mention "types" in the text, we just mean graphs used as types. This is completely unrelated to the notion of type graph used in the graph transformation literature.

[3]We use the term "unselection" anytime a morphism is used in the inverse direction: in Definition 11, the morphism ι is used to base subconditions on a smaller subgraph, in effect reducing the *selected* subgraph; it will also appear in our definition of graph programs as the name of an operation that reduces the current *selection*, i.e. the subgraph the program is currently working on, similarly "selection".

We also adopt the convention of representing the morphism ι in a situation $\exists(a, \iota, c)$ implicitly: we prefer to annotate the variable's type graph with the images of items under ι in parentheses.

In the diagram of Definition 13, the triangle indicates that C is the type of the subcondition c' which appears nested inside $\exists(\alpha, \iota, c')$.

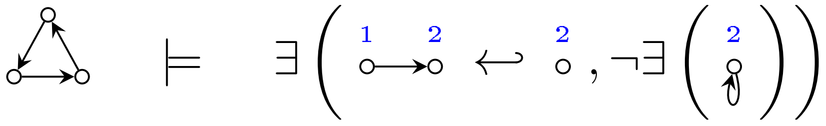


Figure 3.1.: A nested graph condition (stating the existence of two nodes linked by an edge, the second node not having a self-loop) and a graph satisfying it.

As for a logical formula, the most important aspect of a graph condition is not its syntactic form but the property expressed by it.

Definition 14 (Logical equivalence).
The symbol $\equiv$ denotes logical equivalence, i.e. for conditions $c, c' : B$, $c \equiv c'$ iff for all monomorphisms m with domain B, $m \models c \Leftrightarrow m \models c'$.

It is often convenient in later proofs to treat quantification and unselection[3] separately. A condition $\exists(a, \iota, c)$ is equivalent to the nested $\exists(a, \exists^{-1}(\iota, c))$ (recall that $\mathcal{M}$ stands for the collection of all graph monomorphisms):

Lemma 1 (Decomposing "exists"). $\exists(a, \iota, c) \equiv \exists(a, \exists^{-1}(\iota, c))$.

Proof. $f \models \exists(a, \exists^{-1}(\iota, c)) \Leftrightarrow \exists q \in \mathcal{M}, f = q \circ a \wedge q \circ id_{\mathrm{cod}(a)} \models \exists^{-1}(\iota, c)$
$\Leftrightarrow \exists q \in \mathcal{M}, f = q \circ a \wedge \exists q' \in \mathcal{M}, q \circ id_{\mathrm{cod}(a)} = q' \circ id_{\mathrm{cod}(\iota)} \wedge q' \circ \iota \models c'$
$\Leftrightarrow \exists q \in \mathcal{M}, f = q \circ a \wedge \exists q' \in \mathcal{M}, q = q' \wedge q' \circ \iota \models c'$
$\Leftrightarrow \exists q \in \mathcal{M}, f = q \circ a \wedge q \circ \iota \models c' \Leftrightarrow f \models \exists(a, \iota, c)$ □

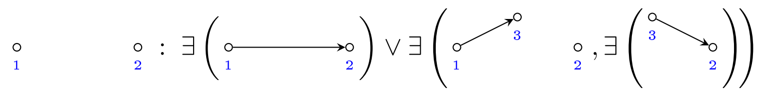
Unselection alone does *not* add expressivity to nested conditions.

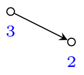
Fact 2 (Equal Expressiveness). *Our conditions with ι are equally expressive as the nested conditions defined in [Pen09].*

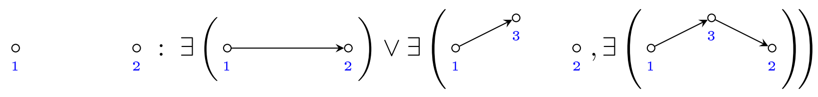
Proof. Structural induction over the nesting, using the defining property of the transformation A and Definition 13 for the case of $\exists^{-1}(\iota, c)$: $f \models \exists^{-1}(\iota, c) \Leftrightarrow f \models A(\iota, c)$. If c, by induction hypothesis, is equivalent to a nested condition without ι, then so is $\exists^{-1}(\iota, c) \equiv A(\iota, c)$. □

We conclude with an example of ι removal.

Example 7 (Paths of length ≤ 2)**.** *This condition expresses the existence of a path of length ≤ 2. It makes use of an unselection. The graph to the left of the colon indicates the condition's type.*



Here is what the condition of Example 7 looks like without the use of unselection. The disjunction created by the A construction has a single member because there is only one possibility to add the node 1 and the edge to the graph  .



At this point, ι could seem to be a notational convenience at best because it really does not add expressiveness. However, it plays an essential role later.

3.3. Recursively Nested Conditions

In this section, we define recursively nested graph conditions, short μ-conditions, on the basis of nested graph conditions. As opposed to nested conditions, the ones defined here can express path and connectivity properties, which frequently arise in the study of the correctness of programs with recursive data structures, or in the modelling of networks.

Nested conditions are a very successful approach to the specification of graph properties for verification. However, they cannot express non-local properties such as connectedness. Our idea is to generalise nested conditions to capture certain non-local properties by adding recursion. The expressiveness of the resulting formalism will later be compared to fixed point logics. The following notations are used throughout the text.

Notation. *Sequences (of graphs, placeholders, morphisms) are written with a vector arrow $\vec{B}$, $\vec{\mathtt{x}}$, $\vec{f}$, and their components are numbered starting from 1. The length of a sequence $\vec{B}$ is denoted by $\|\vec{B}\|$. Indexed typewriter letters $\mathtt{x}_1$ stand for placeholders, i.e. variables. The notation $c : B$ indicating that c has type B is also extended to sequences: $\vec{c} : \vec{B}$ (provided that $\|\vec{c}\| = \|\vec{B}\|$).*

To define fixed point conditions, we need something to take fixed points of, and to enforce existence and uniqueness. The fixed point conditions will be defined as least fixed point solutions of certain systems of equations.

To represent systems of simultaneous equations, we work on tuples of conditions. If $\vec{B} = B_1, \ldots, B_{\|\vec{B}\|}$ is a sequence of graphs, then $\mathrm{Cond}_{\vec{B}}$ is the set of all $\|\vec{B}\|$-tuples $\vec{c}$ of conditions, whose i-th element is a condition over the i-th graph of $\vec{B}$. Satisfaction is extended component-wise: $\vec{f} \models \vec{c}$ if and only if $\forall k \in \{1, \ldots, \|\vec{B}\|\}\ f_k \models c_k$.

Choosing a partial order on $\mathrm{Cond}_{\vec{B}}$, one can define monotonic operators on $\mathrm{Cond}_{\vec{B}}$. The semantics of satisfaction already defines a pre-order: $c \leq c'$ if and only if every morphism that satisfies c also satisfies c', which is obviously transitive and reflexive. As in every pre-order, $\leq \cap \leq^{-1}$ is an equivalence relation compatible with $\leq$ and comparing representants via $\leq$ partially orders its equivalence classes. We introduce variables as placeholders where further conditions can be substituted[4].

Disjunctions $\bigwedge$ and conjunctions $\bigvee$ of countable sets of Cond_B conditions, which by definition exist for any B, are easily seen to be least upper bounds, respectively greatest lower bounds of the sets of conditions. This makes $\mathrm{Cond}_{\equiv B}$ a complete lattice. Let $\mathrm{Cond}_{\vec{B}}$ be ordered with the product order by defining $\vec{f} \models \vec{c}$ to be true when the conjunction holds. This again induces a partial order on the set of equivalence classes, $\mathrm{Cond}_{\equiv \vec{B}}$. Thus, $\mathrm{Cond}_{\equiv \vec{B}}$ is also a complete lattice, and a monotonic operator $\mathcal{F}$ has a least fixed point (*lfp*), given by the limit of $\vec{\mathcal{F}}^n(\overrightarrow{\bot})$ for all $n \in \mathcal{N}$, by the Knaster-Tarski theorem [Tar55]. This is crucial in the definition of a μ-condition. We extend the inductive Definition 11 by placeholders, and define substitutions of conditions for placeholders:

Definition 15 (**Graph Conditions with Placeholders**). *Given a graph B and a finite sequence $\vec{B}$ of graphs, a* (graph) condition with placeholders from $\vec{B}$ *over B is either $\exists(a, \iota, c)$, or $\neg c$, or $\bigwedge_{j \in J} c_j$ with a* finite *index set J, or $\mathbf{x}_i$, $1 \leq i \leq \|\vec{B}\|$ where $\mathbf{x}_i$ is a variable of type B_i.*

A placeholder in a condition $\mathcal{F}$ can be replaced by a condition of same type, which can itself contain more placeholders. Substitution means that all occurrences of a given placeholder are replaced by identical subconditions, and such a replacement is specified for each of the placeholders that appear in $\mathcal{F}$.

Definition 16 (**Substitution**). *If $\mathcal{F}$ is a condition with placeholders $\vec{\mathbf{x}}$ of types $\vec{B}$ and $\vec{c} \in \mathrm{Cond}_{\vec{B}}$, then $\mathcal{F}[\vec{\mathbf{x}}/\vec{c}]$ is obtained by substituting c_i for each occurrence of $\mathbf{x}_i$ for all $i \in \{1,..,\|\vec{B}\|\}$.*

Satisfaction of such a condition by a morphism f is defined relative to *valuations*, which are functions that assign a Boolean value to each monomorphism from the type of each variable to $\mathrm{cod}(f)$

Definition 17 (**Valuation**). *A* valuation *of the variables $\vec{\mathbf{x}}$ is a function that assigns, for every variable $\mathbf{x}_i : B_i$, either $\top$ or $\bot$ to each monomorphism $f : B_i \hookrightarrow \mathrm{cod}(f)$.*

As a stepstone to the definition of μ-condition satisfaction later in this section, we now define satisfaction relative to a valuation.

[4]Note that in our approach variables stand for subconditions, not for attributes or parts of graphs. Wherever confusion with similarly named concepts from the literature could arise, we use the word "placeholder" for "variable".

Definition 18 (**Satisfaction of Conditions with Placeholders**)**.** *If* val *is a valuation, $\mathcal{F}$ is a condition with placeholders $\vec{\boldsymbol{x}}$ and $\boldsymbol{x}_i$ is a variable from $\vec{\boldsymbol{x}}$, then a morphism f (with* $\mathrm{dom}(f) = B_i$ *and $\boldsymbol{x}_i : B_i$) satisfies $\boldsymbol{x}_i$ (written $f \models_{\mathrm{val}} \boldsymbol{x}_i$) iff* $\mathrm{val}(\boldsymbol{x}_i) = \top$*. The other cases for $\mathcal{F}$ are handled as in Definition 13.*

We now turn to least fixed points. As discussed above, a least fixed point shall be defined only up to logical equivalence. To guarantee its existence, the operator must be monotonic ($\vec{c} \leq \vec{d} \Rightarrow \vec{\mathcal{F}}(\vec{c}) \leq \vec{\mathcal{F}}(\vec{d})$ for any $\vec{c}, \vec{d} \in \mathrm{Cond}_{\vec{B}}$).

Notation. *Let $\vec{\mathcal{F}}^0(\vec{\boldsymbol{x}}) := \vec{\boldsymbol{x}}$ and $\vec{\mathcal{F}}^i(\vec{\boldsymbol{x}}) := \vec{\mathcal{F}}(\vec{\mathcal{F}}^{i-1}(\vec{\boldsymbol{x}}))$ for $i \in \mathbb{N} - \{0\}$.*

The following remark is a very useful characterisation of the least fixed point:[5]:

Remark 5. *The least fixed point of $\vec{\mathcal{F}}$ is equivalent to $\bigvee_{n\in N} \vec{\mathcal{F}}^n(\overrightarrow{\bot})$.*

Proof. This is a fixed point because $\vec{\mathcal{F}}(\bigvee_{n\in N} \vec{\mathcal{F}}^n(\overrightarrow{\bot})) = \bigvee_{n\in N-\{0\}} \vec{\mathcal{F}}^n(\overrightarrow{\bot}) = \bot \vee \bigvee_{n\in N-\{0\}} \vec{\mathcal{F}}^n(\overrightarrow{\bot}) = \bigvee_{n\in N} \vec{\mathcal{F}}^n(\overrightarrow{\bot})$. It is the least fixed point because any other fixed point must also be a least upper bound of all $\vec{\mathcal{F}}^n(\overrightarrow{\bot})$ and therefore greater or equal to the one proposed. $\square$

Notation. *$\vec{\mathcal{F}}(\vec{c})$ is a short notation for the result of substituting the conditions $\vec{c}$ for the variables $\vec{\boldsymbol{x}}$ in each component of $\vec{\mathcal{F}}$. The notation $\mathcal{F}(\vec{\boldsymbol{x}})$ merely specifies that any of the list $\vec{\boldsymbol{x}}$ can occur in $\mathcal{F}$, not which ones actually appear or how often. The operator yields new components $\mathcal{F}_i$ which may depend on any of the old components of $\vec{c}$. The operator binds the variables $\vec{\boldsymbol{x}}$.*

We are now ready to define recursively nested conditions, short *μ-conditions.*

Definition 19 (**μ-Condition**)**.** *Given a finite list $\vec{B}$ of graphs and a corresponding list of finitary conditions $\{\mathcal{F}_i\}_{i\in\{1,\ldots,\|\vec{B}\|\}}$ with placeholders $\vec{\boldsymbol{x}} : \vec{B}$ ($\mathcal{F}_i$ having type B_i), then $\mu[\vec{\boldsymbol{x}}]\vec{\mathcal{F}}(\vec{\boldsymbol{x}})$ denotes the least fixed point* (lfp) *of the operator that to any $\vec{c}$ assigns $\vec{\mathcal{F}}[\vec{\boldsymbol{x}}/\vec{c}]$. A recursively nested condition, short μ-*condition*, is a pair (b, l) consisting of a condition with placeholders b, and a finite list of pairs $l = (\boldsymbol{x}_i, \mathcal{F}_i(\vec{\boldsymbol{x}}))$ of a variable $\boldsymbol{x}_i : B_i$ and a condition $\mathcal{F}_i(\vec{\boldsymbol{x}}) : B_i$, with placeholders from $\vec{\boldsymbol{x}}$, for some graph B_i, such that $\vec{\mathcal{F}}$ is monotonic.*

Alternatively, it would be reasonable to define a family of equations without imposing an order on the variables. We chose the list format for reasons of notational convenience.

Notation. *The pair (b, l) is also denoted $(b \mid l)$ or $(b \mid \vec{\boldsymbol{x}} = \vec{\mathcal{F}}(\vec{\boldsymbol{x}}))$ to place emphasis on the interpretation of l as a recursive* specification *of the variables used in b.*

[5]Note that continuity of $\vec{\mathcal{F}}$ is never used or assumed. Only monotonicity matters.

A μ-condition with no variables and equations is a finitary nested condition. Figure 3.2 shows a simple μ-condition that is not a finitary nested condition. It is read as follows: the word "where" stands between main body and equations (we usually represent this as a vertical bar). The only variable is $\mathtt{x}_1$. Its type is indicated in square brackets. The second existential quantifier uses a morphism to unselect node 1 and the sole edge: its source is the type of $\mathtt{x}_1$, which is syntactically required for using the variable in that place. The unselection morphism ι is not written as an arrow, instead it is expressed in compact notation by appending small blue numbers in parentheses to the node numbers in its source graph to specify the mapping. To ease reading, we adopt the convention to always use the same layout for the type of a given variable.

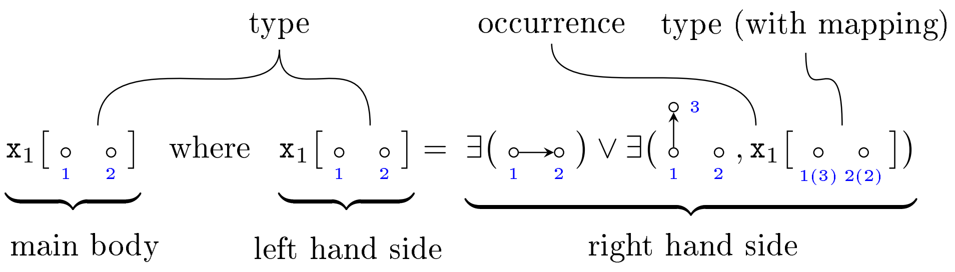

type
occurrence
type (with mapping)
x1 [∘1 ∘2] where x1 [∘1 ∘2] = ∃(∘1 → ∘2) ∨ ∃(∘1 → ∘3 ∘2 , x1 [∘1(3) ∘2(2)])
main body
left hand side
right hand side

Figure 3.2.: The parts of a μ-condition explained

The expressive power of μ-conditions exceeds that of finitary nested conditions, as illustrated by example Figure 3.2. This example can be read on an intuitive level as the result of an indefinitely repeated substitution of the definition of the variable $\mathtt{x}_1$ into itself. In the course of this section, we develop a semantics for μ-conditions that gives a formal meaning.

Fact 3 (**μ-Conditions are More General than Nested**).

1. *μ-conditions generalise finitary nested conditions, consequently all examples for nested conditions are examples for μ-conditions (with no variables or equations).*
2. *μ-conditions are strictly more general than finitary nested conditions: Figure 3.2 expresses the existence of a path of unknown length between two given nodes, which is known not to be expressible as a finitary nested condition.*

In fact, many properties of interest in the verification of graph programs will be shown to be expressible as μ-conditions.

Fact 4 (**Expressible Properties**). *The following properties (families) of graphs can be expressed by μ-conditions, see also Appendix A:*

1. *trees*
2. *binary trees*
3. *balanced binary trees*

4. *acyclic graphs*
5. *connected graphs*

Before giving the formal semantics of μ-conditions, we first develop several technical results that become necessary when fixed point operators are nested. To be able to specify mutually recursive nested fixed points, we allow μ-conditions with open variables (i.e. not occurring as left-hand sides). For a least fixed point of a subset of variables to exist, the system of equations must correspond to a monotonic operator under *any* valuation of the open variables. An operator is said to be *monotonic in* a subset of variables when it is monotonic under any valuation of the remaining variables.

Notation. *We write the list of pairs* $l = (\boldsymbol{x}_i, \mathcal{F}_i(\vec{\boldsymbol{x}}))_{i\in\{1,\dots,\|\vec{B}\|\}}$ *as a system of equations* $\vec{\boldsymbol{x}} = \vec{\mathcal{F}}(\vec{\boldsymbol{x}})$. *We call* b *the* main body *and* l *the* recursive specification *of* $(b \mid l)$ *(and* $\mathcal{F}_i(\vec{\boldsymbol{x}})$ *the* body *or* right hand side *of the variable* $\boldsymbol{x}_i$ *in* l*, or the* i*-th* component *of* $\vec{F}$*). The list* $\vec{\mathcal{F}}$ *is said to* define *the variables* $\vec{\boldsymbol{x}}$*.*

Such systems of equations may be used in a broader sense, to define nested fixed points:

Definition 20 (Transitive Variable Use). *Let* $\{\mathcal{F}_i\}_{i\in I}$ *be a list of conditions as in Definition 19. The* use *relation of* $\mathcal{F}$*,* $\rightsquigarrow_{\mathcal{F}}$*, is defined on literals* $\{\boldsymbol{x}_i, \neg\boldsymbol{x}_i\}_{i\in I}$ *by* $\boldsymbol{x}_i \rightsquigarrow_{\mathcal{F}} \boldsymbol{x}_j$ $(\neg\boldsymbol{x}_j)$ *iff* $\boldsymbol{x}_j$ *occurs as a subcondition under an even (odd) number of negations in* $\mathcal{F}_i$*. The* transitive use paths *of* $\mathcal{F}$ *are all sequences of literals* $\pi_{p_1}\dots\pi_{p_m}$ *such that* $\forall 1\le i<m \left(\pi_{p_i} \rightsquigarrow_{\mathcal{F}} \pi_{p_{i+1}}\right)$ *and* $\forall 1<j<m \left(\pi_{p_m} \neq \pi_{p_j}\right)$*.*

Fixed points can be arbitrarily nested provided that cycles are avoided. The precautions necessary when forming cycles are explained later.

Lemma 2 (Nested Fixed Points). *Given conditions with placeholders* $\{\mathcal{F}_i(\vec{x})\}_{i\in I}$ *that form a monotonic operator* $\mathcal{F}$*, if there is a partitioning* $I = I_1 \uplus I_2$ *with* $\vec{\boldsymbol{x}}_1 = \{\vec{\boldsymbol{x}}_i\}_{i\in I_1}$ *and* $\vec{\boldsymbol{x}}_2 = \{\vec{\boldsymbol{x}}_i\}_{i\in I_2}$ *such that* $\{\mathcal{F}_i\}_{i\in I_1}$ *does not use variables of* $\vec{\boldsymbol{x}}_2$*, then* $\mu[\vec{\boldsymbol{x}}]\vec{\mathcal{F}}(\vec{\boldsymbol{x}})$ *is equivalent to* $\mu[\vec{\boldsymbol{x}}_2]\vec{\mathcal{F}}_{I_2}(\vec{\boldsymbol{x}}_2)$ *with* $\vec{\mathcal{F}}_{I_2}(\vec{\boldsymbol{x}}_2) = \vec{\mathcal{F}}[\vec{\boldsymbol{x}}_1/\mu[\vec{\boldsymbol{x}}_1]\vec{\mathcal{F}}_{I_1}(\vec{\boldsymbol{x}}_1)](\vec{\boldsymbol{x}}_2)$*.*

Proof. Writing $\{\vec{x}_i\}_{i\in I} = \vec{x} = (\vec{x}_1, \vec{x}_2)$ for the sequence of all the variables, $\vec{\bot}$ for sequences of $\bot$ of appropriate lengths, $\hat{\vec{\mathrm{x}}}_1$ for the least fixed point solution $\mu[\vec{\mathrm{x}}_1]\vec{\mathcal{F}}_{I_1}(\vec{\mathrm{x}}_1)$, (1) for $\mu[\vec{\mathrm{x}}]\vec{F}(\vec{\mathrm{x}})$ and (2) for $\mu[\vec{\mathrm{x}}_2]\vec{\mathcal{F}}_{I_2}(\vec{\mathrm{x}}_2)$, the argument starts from the evident fact that $\vec{\bot} \le (\hat{\vec{\mathrm{x}}}_1, \vec{\bot})$. It follows by monotonicity of $\vec{\mathcal{F}}$ and $\vec{\mathcal{F}}_{I_1}$ that (1) $\subseteq$ (2) as all the stages in the induction compare in this way. Conversely, $(\hat{\vec{\mathrm{x}}}_1, \vec{\bot}) \le$ (1) and by monotonicity of $\vec{\mathcal{F}}$, $\vec{\mathcal{F}}^i_{I_2}(\vec{\bot})$ is always smaller or equal than (1), which is a fixed point of $\vec{\mathcal{F}}$. This allows us to conclude that (2) $\le$ (1) too. □

It is permissible for a variable to depend on another variable regardless of monotonicity, but only when special care is taken. Cycles can be avoided by working with a stratified set of variables:

Definition 21 (**Stratification**). *$\mathcal{F}$ is said to be* stratified *if there is a decomposition into $\vec{\mathcal{F}}_{I_1}, ..., \vec{\mathcal{F}}_{I_n}$ such that each $\vec{\mathcal{F}}_{I_m}$ is monotonic in $\vec{\mathbf{x}}_{I_m}$ and there are no variables $\mathbf{x}_i \rightsquigarrow^+_{\mathcal{F}} \mathbf{x}_j$, $j \in I_j$, $i \in I_i$, $j < i$. Such a decomposition is termed a* stratification *of $\vec{\mathcal{F}}$ and the $\vec{F}_{I_m}$ are* strata *of $\vec{\mathcal{F}}$.*

Note that the possible decompositions only depend on the strict partial order of transitive variable use $\rightsquigarrow^+_{\mathcal{F}}$. The order and decomposition of fixed points on $\rightsquigarrow^+_{\mathcal{F}}$-incomparable subsets of variables does not matter (by Lemma 2). Therefore there is no ambiguity in presenting a nested fixed point as a system of equations without explicit stratification. The passage to a single fixed point is depicted in Figure 3.4. Monotonicity and stratification can be enforced syntactically and we only consider such μ-conditions to be well-formed:

Fact 5 (**Positive Variables**). *If there is no transitive use path starting and ending on the same variable and comporting an odd number of negations, then $\vec{\mathcal{F}}$ is stratified.*

Proof. We prove by structural induction that $\mathcal{F}_i(\vec{x})$ is monotonic in $\mathtt{x}_j$ under even numbers of negations and *anti*tonic under odd numbers, i.e. $c \leq d \Rightarrow \mathcal{F}_i[\mathtt{x}_j/c] \leq \mathcal{F}_i[\mathtt{x}_j/d]$ resp. $c \leq d \Rightarrow \mathcal{F}_i[\mathtt{x}_j/c] \geq \mathcal{F}_i[\mathtt{x}_j/d]$. The base case is either $\top$ or $\mathtt{x}_{j'}$, $j \neq j'$ (trivial), or $\mathtt{x}_j$ (monotonic). The other cases are negation, disjunction and existential quantifiers. Examining Definition 13, negation interchanges both even/odd and monotonicity/antitonicity. Disjunction, defined via propositional logic too, is monotonic (and it is not possible that the disjunction contains both monotonic and antitonic uses of $\mathtt{x}_j$ because of the assumption on transitive use paths). Quantifiers $\exists(a, \iota, c')$ are monotonic in c'. Hence the latter two cases do not affect either property. If all components of $\vec{\mathcal{F}}$ are monotonic in $\mathtt{x}_j$, then so is $\vec{F}$. Porting the argument to stratified systems merely requires checking the monotonicity of each stratum. $\square$

Figure 3.3 is a schematic depiction of a condition with creatively nested negations. This condition is well-formed according to Fact 5 in spite of variable $\mathtt{x}_0$ depending negatively on $\mathtt{x}_1$, $\mathtt{x}_1$ on $\mathtt{x}_2$ and $\mathtt{x}_2$ on $\mathtt{x}_1$. It has two strata. Nodes represent variables or connectives.

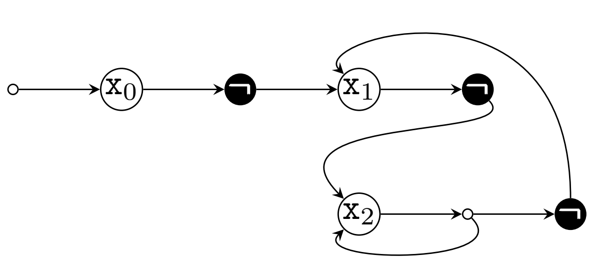


Figure 3.3.: Informal and schematic depiction of a condition with negations.

To illustrate the point of Definition 21, notice how nested fixed points, possibly with open variables in the inner fixed point that are bound only by the outer fixed point operator, can be transformed into a single fixed point. Figure 3.4 informally and schematically depicts the flattening of a mutually recursive condition from a nested fixed point to a single fixed point of two variables. In the first variant, $\mathtt{x}_2$ would be bound by a fixed point operator that does not bind $\mathtt{x}_1$. In the second variant, both variables are bound by the same least fixed point operator.

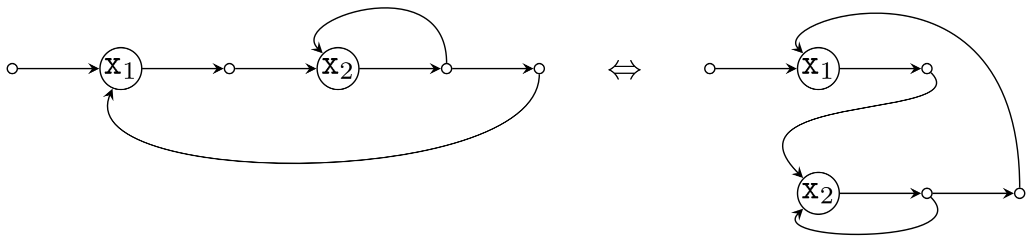


Figure 3.4.: Flattening nested fixed points,

The small nodes in Figure 3.4 stand for operators (Boolean operators, quantifiers) used in the main body and right hand sides and the nodes with variable names for variables. Edges entering variables indicate variable use while edges leaving them point to the respective right hand sides. Any loop in this schematic diagram must run through an even number of negations for a well-formed μ-condition.

Now we can define the semantics of μ-conditions. Any graph condition formalism must define a notion of satisfaction, since graph conditions are used to define properties of graphs. As for nested conditions, satisfaction of a condition by a graph is defined via the auxiliary, more general notion of satisfaction of a condition by a morphism:

Definition 22 (**Satisfaction of μ-Conditions**). *$(b \mid \vec{\mathtt{x}} = \vec{\mathcal{F}}(\vec{\mathtt{x}}))$ with $\vec{\mathtt{x}} : \vec{B}$ is satisfied by f iff $f \models (b[\vec{\mathtt{x}}/\mu[\vec{\mathtt{x}}]\mathcal{F}])$.*

This means that the μ-condition $(b \mid \vec{\mathtt{x}} = \vec{\mathcal{F}}(\vec{\mathtt{x}}))$ can be understood by substituting the least fixed point solution of the system of equations $\vec{\mathtt{x}} = \vec{\mathcal{F}}(\vec{\mathtt{x}})$ in the main body b (for stratified systems, use appropriate nested fixed points). Satisfaction of μ-conditions with open variables is analogous to satisfaction of conditions with placeholders, i.e. requires a valuation to be given.

Remark 6 (**Finite Nesting**). *By the "infinite disjunction" characterisation of the least fixed point, any μ-condition is equivalent to an infinite nested condition. Infinitely deep nesting is not needed because the characterisation in Remark 5 yields a countable disjunction of nested conditions, each one of* finite *nesting depth.*

A morphism satisfies a given μ-condition if and only if it satisfies the finite nested condition obtained by unrolling the recursive specification up to some finite depth:

Proposition 1 (**Satisfaction at Finite Depth**).
$f \models (b \mid \vec{\mathtt{x}} = \vec{\mathfrak{F}}(\vec{\mathtt{x}}))$ iff $\exists n \in \mathbb{N}$, $f \models b[\vec{\mathtt{x}}/\vec{\mathfrak{F}}^n(\overrightarrow{\bot})]$.

Proof. The least fixed point is equivalent to $\bigvee_{i \in N} \vec{\mathfrak{F}}^i(\overrightarrow{\bot})$, which is satisfied by f iff at least one $\vec{\mathfrak{F}}^n(\overrightarrow{\bot})$ is. □

Let us now motivate the necessity of unselection. As an auxiliary notion, the *nesting depth* $n : \mathrm{Cond} \to \mathbb{N}$ is defined as $n(\bigwedge_{j \in J} c_j) = \max(\{0\} \cup \{n(c_j) \mid j \in J\})$, $n(\neg c) = n(c)$, $n(\exists(a, \iota, c)) = n(c) + 1$, $(n(\mathtt{x}_i) = 0)$. The absorption construction allows transformations of conditions that decrease nesting depth when isomorphisms are involved. Define the reduced condition $r_{a,\iota}(c')$ thus:

Definition 23 (**Reduced Condition**). *If $c' = \bigwedge_{j \in J} c_j$, then $r_{a,\iota}(c') = \bigwedge_{j \in J} r_{a,\iota}(c_j)$. If $c' = \neg c''$, then $r_{a,\iota}(c') = \neg r_{a,\iota}(c'')$. If $c' = \exists(a', \iota', c'')$, then $r_{a,\iota}(c') = \exists(a' \circ \iota^{-1} \circ a, c')$.*

Using this definition, isomorphisms can be absorbed:

Lemma 3 (**Absorption**). *Any condition with placeholders $c = \exists(a, \iota, c')$ where a and ι are isomorphisms is equivalent to a condition of smaller nesting depth (or equal if $n(c) = 1$).*

Proof. Directly from Definition 13, $r_{a,\iota}(c') \equiv c$ and at the same time, $n(r_{a,\iota}(c')) = n(c') = n(c) - 1$. The only case where nesting depth does not decrease is when c' is a variable, resulting in nesting depth 1. □

We can use the notion of absorption and Lemma 3 to prove that without nontrivial unselection in the equations, μ-conditions are not more expressive than finitary nested conditions. This, rather than notational convenience, is the real motivation for introducing unselection.

Fact 6 (**Absorbing ι Isomorphisms**). *Any μ-condition $(b \mid \vec{\mathtt{x}} = \vec{\mathfrak{F}}(\vec{\mathtt{x}}))$ where ι is the identity in all subconditions of b and of the components $\mathfrak{F}_i(\vec{\mathtt{x}})$ is equivalent to a nested condition.*

Proof. Decompose $\vec{\mathfrak{F}}$ by Lemma 2 such that each stratum $\vec{\mathfrak{F}}_{I_m}$ only defines variables of the same type. This is indeed possible since with no non-trivial unselection, a variable may transitively depend on itself only via a morphism that is both injective and surjective. Induction over the number of strata: for each $\vec{\mathfrak{F}}_{I_m}$, after each step of the lfp iteration, the nesting level can be reduced by Lemma 3 whenever $\exists(a, \iota, c)$ with a isomorphism occurs. Hence an equivalent condition of nesting level 0 or 1 can be reached. It must be a Boolean combination of conditions of the form $\exists(a, \iota, \mathtt{x}_i)$ with isomorphisms a and ι. Finitely many distinct conditions of this form, hence finitely many distinct Boolean combinations, exist. The monotonic operator $\vec{\mathfrak{F}}_{I_m}$ thus converges after finitely many steps to a finitely deeply nested condition with placeholders, for which the next stratum's lfp, by induction hypothesis possessing the desired property, is substituted. □

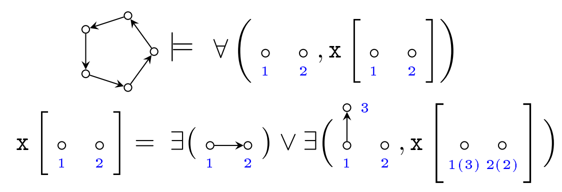
From the definition of satisfaction via fixed points, it is not immediately clear whether satisfaction of μ-conditions is decidable at all. The answer is yes, as witnessed by a polynomial-time algorithm:

Theorem 1 (**Deciding Satisfaction of μ-Conditions**)**.** *Given a morphism $f : B \hookrightarrow G$ and a μ-condition c, it is decidable whether f satisfies c.*

Proof. The following algorithm `CheckMu` decides $f \models c$. For the type B_i of each variable $\mathtt{x}_i$, list all monomorphisms $m_{ik} : B_i \hookrightarrow G$. Build a table which records in each column a Boolean value for each pair $(\mathtt{x}_i, m_{ik})$. The entries in column $j+1$ are computed by evaluating satisfaction of the right hand side corresponding to the row's variable by the morphism m_{ik} associated with the row, under the valuation given by column j. Stop after producing two adjacent columns with the same entries. Output the value of the main body under that valuation. The algorithm is correct because the j-th column corresponds to satisfaction by $\vec{\mathfrak{F}}^j(\overrightarrow{\perp})$, by definition. It terminates because of monotonicity: as values can never change back to $\perp$ from $\top$ while progressing through the columns, there is a finite number $j^* \in \mathbb{N}$ such that $\vec{\mathfrak{F}}^{j^*}$ is satisfied by f iff $\vec{\mathfrak{F}}^{j^*+1}$ is. $\square$

The idea of Theorem 1 is illustrated in Figure 3.5, where part of the table for checking directed connectedness of a 5-cycle is shown.

Figure 3.5 illustrates the algorithm `CheckMu`. The table has 20 rows, for the $2! \cdot \binom{5}{2} = 20$ distinct ways that a discrete graph of two nodes (the type of the sole variable $\mathtt{x}$) can appear as a subgraph of the 5-cycle.



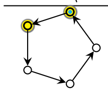
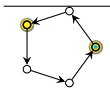
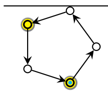
x \	$\perp$	$\mathcal{F}(\perp)$	$\mathcal{F}(\mathcal{F}(\perp))$	$\mathcal{F}^3(\perp)$	$\mathcal{F}^4(\perp)$
	X	✓	✓	✓	✓
	X	X	✓	✓	✓
	X	X	X	✓	✓
⋮					

Figure 3.5.: Illustration of the satisfaction check algorithm.

3.4. Expressiveness of Recursively Nested Conditions

In the following, we compare our μ-conditions to other formalisms for expressing non-local properties: the very powerful grammar-based HR*-conditions of Radke [Rad13, Rad16], and the M-conditions [PP14] of Poskitt and Plump. μ-conditions are defined in a way that extends finite nested conditions. We now compare them to several other concepts found in the literature and present our conclusions.

In this section, HR$^-$-conditions are introduced as are a variant of HR*-conditions with a certain restriction on the graph grammars.

We will show that μ-conditions are:

- equally expressive as first-order logic with least fixed points (Theorem 2).
- able to express non-semilinear unary string languages (Corollary 1).
- incomparable to M-conditions and HR$^-$-conditions:

  there are properties that can be expressed as M-conditions as well as HR$^-$-conditions but not as μ-conditions (Corollary 2, Corollary 6).

  there are properties that can be expressed as μ-conditions but neither as M-conditions nor as HR$^-$-conditions (Corollary 3, Corollary 5).

We also note that some node-counting monadic second-order properties cannot be expressed in HR$^-$ (Fact 7) despite being expressible in HR* [Rad13]. We conjecture that the last statement holds true for unrestricted HR*-conditions instead of HR$^-$-conditions. The results can be summarised in a table, where $\lessgtr$ means incomparable and $=$ means equal expressive power:

μ-conditions vs. HR$^-$-conditions	$\lessgtr$
μ-conditions vs. M-conditions	$\lessgtr$
μ-conditions and FO+lfp	$=$

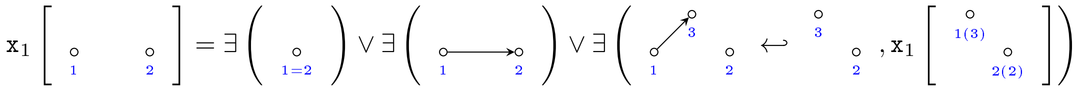
To ease our reasoning about the expressiveness of μ-conditions, we introduce a variant, $\mathcal{A}$-μ-conditions, that allows arbitrary morphisms to appear instead of monomorphisms, both under existential quantifiers and in the definition of satisfaction. This means that the a in $\exists(a, \iota, c)$ may now be an arbitrary morphism, as may the morphisms q that occur in the condition for satisfaction.

We now define $\mathcal{A}$-μ-condition in almost the same way as a μ-condition (Definition 19), except for the restriction that the morphism a has to be a monomorphism in a μ-condition, which is no longer the case for an $\mathcal{A}$-μ-condition.

Definition 24 (**$\mathcal{A}$-μ-condition**). *The definition of $\mathcal{A}$-μ-condition follows Definition 11, Definition 15, Definition 19 with one exception: in the case $\exists(a, \iota, c)$ of the inductive definition, a is no longer restricted to be a monomorphism. Only ι is required to be in $\mathcal{M}$.*

The following example can only be an $\mathcal{A}$-μ-condition due to the non-injective morphism:

Example 8 ($\mathcal{A}$-μ-condition)**.**



We now define $\mathcal{A}$-satisfaction of an $\mathcal{A}$-μ-condition. The definition is almost the same as Definition 13 and Definition 22, except for the case of existential quantifiers, where the required morphism q is now arbitrary.

Definition 25 ($\mathcal{A}$**-Satisfaction**)**.** *$\mathcal{A}$-satisfaction ($f \models_{\mathcal{A}} c$) of an $\mathcal{A}$-μ-condition c by a morphism $f \in \mathcal{A}$ is defined analogously to Definition 13, but the condition is now an $\mathcal{A}$-muGC and in the case of $c = \exists(a, \iota, c')$, q is arbitrary.*

The condition of Example 8 has the meaning that the two given nodes are one and the same, or there exists a path of unknown length between them. In an $\mathcal{M}$-μ-condition, this does not make sense because two nodes in the interface are always distinct.

For the remainder of this section, we use $\mathcal{M}$-μ-condition as a synonym for μ-condition and $\mathcal{M}$-satisfaction as a synonym for satisfaction of μ-conditions. We call $\mathcal{M}$-*semantics* any notion of satisfaction of graph conditions by injective morphisms and $\mathcal{A}$-*semantics* any notion of satisfaction by arbitrary morphisms.

We shall now prove that $\mathcal{M}$- and $\mathcal{A}$-μ-conditions with their respective notions of satisfaction (Definition 13 resp. Definition 25) can express the same properties. The informal justification for the existence of such a transformation is that all $\mathcal{M}$-ness is "local" within the condition. Unlike in a so-to-speak *true* $\mathcal{M}$-semantics, injectivity is not enforced with respect to what has already been unselected, and the size of the interface is globally bounded. While this leads to a somewhat unnatural semantics of simple-looking conditions in detail, as shown later on in Example 15, even the "local" $\mathcal{M}$-ness is useful for computing weakest preconditions (as we will see in Section 5.1). For the problem at hand, we provide two transformations $\mathcal{M}to\mathcal{A}$ and $\mathcal{A}to\mathcal{M}$ to convert between $\mathcal{M}$ and $\mathcal{A}$ for μ-conditions.

Satisfaction of graph conditions by morphisms has originally been introduced as an auxiliary notion to recursively define satisfaction of nested graph conditions. Expressiveness normally refers to objects, not morphisms, otherwise a $\mathcal{M}$-semantics and an $\mathcal{A}$-semantics are never equivalent as the former specifies only injective morphisms. Therefore the domain should be $\emptyset$ when comparing expressiveness.

In the generic framework of [Pen09], several finiteness assumptions are made. All of these assumptions hold in the category of graphs and graph morphisms. Specifically, we shall have recourse to the property that the number of epimorphisms with a given domain is finite (up to isomorphisms of the codomain).

Given therefore a property of graphs expressed as a μ-condition in $\mathcal{M}$-semantics, the same property is expressible in $\mathcal{A}$-semantics (similar to the corresponding theorem in [Rad13]):

We introduce the notation $\mathrm{AllNew}(B)$ for a conjunction of nested conditions stating the non-existence of surjective morphisms of domain B that are not isomorphisms: $\mathrm{AllNew}(B) := \bigwedge_{e:B\twoheadrightarrow X,\ \text{not}\ \cong} \neg\exists(e)$.

Construction 1 ($\mathcal{M}$ **to** $\mathcal{A}$)**.** *For every $\mathcal{M}$-μ-condition $(b \mid \mathcal{F}(\vec{\mathfrak{c}}))$, each right hand side of the transformed μ-condition is $\mathcal{F}'_i(\vec{\mathfrak{x}}) = \mathcal{F}_i(\vec{\mathfrak{c}}) \wedge \mathrm{AllNew}(B_i)$. The variables and their types themselves are not modified. To the main body and right hand sides of a μ-condition $(b \mid \vec{\mathfrak{x}} = \vec{\mathcal{F}}(\vec{\mathfrak{x}}))$, apply this transformation:*

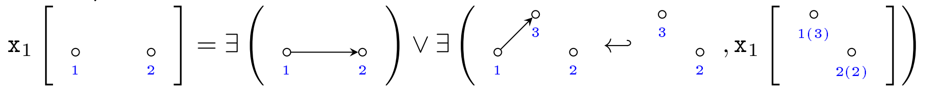
$\mathcal{M}to\mathcal{A}(\exists(a, c)) = \exists(a, \mathcal{M}to\mathcal{A}(c) \wedge \mathrm{AllNew}(\mathrm{cod}(a)))$.

The other cases are not modified, $\mathcal{M}to\mathcal{A}$ is recursively passed down: $\mathcal{M}to\mathcal{A}(\exists^{-1}(\iota, c)) = \exists^{-1}(\iota, \mathcal{M}to\mathcal{A}(c))$, $\mathcal{M}to\mathcal{A}(\bigvee_i c_i) = \bigvee_i \mathcal{M}to\mathcal{A}(c_i)$ and $\mathcal{M}to\mathcal{A}(\neg c) = \neg\mathcal{M}to\mathcal{A}(c)$.

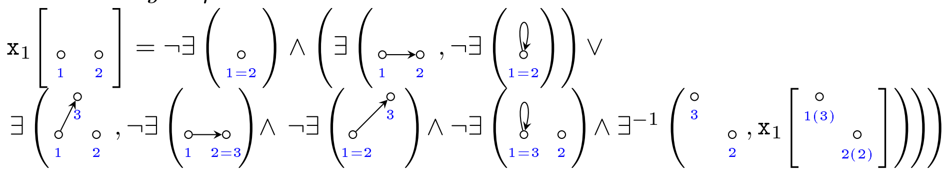
Within each right hand side, the construction corresponds to Construction 3.16 from [Pen09].

Example 9 ($\mathcal{M}to\mathcal{A}$ for recursively nested conditions)**.**

An $\mathcal{M}$-μ-condition:



The resulting $\mathcal{A}$-μ-condition:



The graphical representation uses the convention of indicating non-injective morphisms with equality signs between the identities of the preimage nodes. Types of variables are indicated in square brackets on the left hand side.

It must be shown that the above construction fulfils its purpose.

Proposition 2 (**Correctness of Construction 1**)**.** *Let $(b \mid \vec{\mathfrak{x}} = \vec{\mathcal{F}}(\vec{\mathfrak{x}}))$ be a $\mathcal{M}$-μ-condition whose main body has type B, and $f : B \hookrightarrow G$. Then $f \models (b \mid \mathcal{F})$ iff $f \models_{\mathcal{A}} \mathcal{M}to\mathcal{A}(b \mid \mathcal{F})$.*

Proof. Structural induction over nesting within induction over the fixed point iteration. The interesting case of the structural induction over nesting is $f \models \exists(a, c)$, $f \in \mathcal{M}$. This means $\exists q \in \mathcal{M}, f = q \circ a$. If $f \models \exists(a, c)$ then certainly $f \models_{\mathcal{A}} \exists(a, c)$ since by hypothesis, $f \models_{\mathcal{A}} \mathcal{M}to\mathcal{A}(c)$ iff $f \models c$. Since $a \circ f \in \mathcal{M}$, it satisfies $\mathrm{AllNew}(a)$ since by uniqueness of the epi-mono-factorisation it does not factor through any of the non-trivial epimorphisms.

Conversely, if $f \models_{\mathcal{A}} c$ then either $f \notin \mathcal{M}$ in which case it fails to satisfy the $\mathrm{AllNew}(B_i)$ at the outer nesting level of the right hand side, or $f \in \mathcal{M}$, in which case a structural induction shows that none of the AllNew() conjuncts have any impact on satisfaction. □

Given a property expressed as an $\mathcal{A}$-μ-condition, the same property is expressible in $\mathcal{M}$-semantics:

$$\begin{tikzcd}
B \arrow[r, two heads, "\bar a"] \arrow[d, two heads, "\bar p"] \arrow[dd, bend right, "f"'] & R \arrow[d, two heads] \\
Q \arrow[r, "\cong"] \arrow[d, hook] & Q \arrow[dl, hook] \\
G &
\end{tikzcd}
\qquad
\begin{tikzcd}
B \arrow[rr, hook, "a^\circ"] \arrow[dd, two heads, "\bar p"] \arrow[ddd, bend right, "f"'] & & R \arrow[dl] \arrow[dd, two heads] \\
& D' \arrow[dr, two heads, "e"] & \\
Q \arrow[ur, hook, "a'"] \arrow[rr, hook] \arrow[d, hook] & & E \arrow[dll, hook] \\
G & &
\end{tikzcd}
\qquad
\begin{tikzcd}
B \arrow[d, two heads, "\bar p"] \arrow[dd, bend right] & R \arrow[l, hook', "\iota"'] \arrow[d, two heads, "\bar p'"] \\
Q \arrow[d, hook] & E \arrow[l, hook', "\iota'"'] \arrow[dl, hook] \\
G &
\end{tikzcd}$$

Figure 3.6.: Diagrams pertaining to Proposition 3's proof. Cases from left to right: an epimorphism in the $\mathcal{A}$-condition; a monomorphism; an unselection.

Construction 2 ($\mathcal{A}$ **to** $\mathcal{M}$)**.** *To any μ-condition $(b \mid \vec{\mathcal{x}} = \vec{\mathcal{F}}(\vec{\mathcal{x}}))$ with $b : \emptyset$, assign $(b' \mid \vec{\mathcal{x}}' = \vec{\mathcal{F}}'(\vec{\mathcal{x}}'))$ with $b' = b$ and the following new types: for each B_i, collect all possibilities $\bar{B}_{i,\bar p_{ij}}$ where $\bar p_{ij} : B_i \twoheadrightarrow Q_{ij}$ for some*[6] *Q_{ij}, up to isomorphism of the codomains Q_{ij}. The new variables and equations are $\mathcal{x}'_{i,\bar p_{ij}}[\bar B_{i,\bar p}] = \mathcal{F}'_{i,\bar p_{ij}}(\vec{\mathcal{x}}')$, where $\mathcal{F}'_{i,\bar p_{ij}}(\vec{\mathcal{x}}') = \mathcal{A}to\mathcal{M}_{\bar p_{ij}}(c)$. The latter transformation, which takes one epimorphism $\bar p$ and a condition with placeholders whose type corresponds to* $\mathrm{dom}(\bar p)$*, is defined as follows:*

$\mathcal{A}to\mathcal{M}_{\bar p}(\exists(B \overset{\bar a}{\twoheadrightarrow} R, c))$: if $\bar a$ does not factor through $\bar p$, then $\bot$, else $\mathcal{A}to\mathcal{M}_{\bar p'}(c)$ where $\bar p = \bar p' \circ \bar a$ (which is indeed uniquely determined).

$\mathcal{A}to\mathcal{M}_{\bar p}(\exists(B \overset{a^\circ}{\hookrightarrow} R, c)) = \bigvee_{(e,a',\bar p')} \exists\left(\mathrm{cod}(\bar p) \overset{e\circ a'}{\hookrightarrow}, \mathcal{A}to\mathcal{M}_{\bar p'}(c)\right)$: where $(a', \bar p')$ is the pushout of $(\bar p, a^\circ)$ and e ranges over all epimorphisms with domain $\mathrm{cod}(a')$ *such that $e \circ a' \in \mathcal{M}$.*

$\mathcal{A}to\mathcal{M}_{\bar p}(\exists^{-1}(B \overset{\iota}{\hookleftarrow} R, c)) = \exists^{-1}(Q \overset{\iota'}{\hookleftarrow} Q', \mathcal{A}to\mathcal{M}_{\bar p'}(c))$ where $(\bar p', \iota')$ is the epi-mono-factorisation of $\bar p \circ \iota$.

Boolean combinations: $\mathcal{A}to\mathcal{M}_{\bar p}(\bigvee_i c_i) = \bigvee_i \mathcal{A}to\mathcal{M}_{\bar p}(c_i)$ and $\mathcal{A}to\mathcal{M}_{\bar p}(\neg c) = \neg\mathcal{A}to\mathcal{M}_{\bar p}(c)$,

It must be shown that the above construction fulfils its purpose. Expressiveness referring to a class of graph properties, it is acceptable for the transformed condition to be satisfied by a different morphism of same codomain. This is why a new morphism f', rather than f appears as a satisfying morphism in one direction of the proof.

Proposition 3 (**Correctness of Construction 2**)**.** *Let $c = (b \mid \vec{\mathcal{x}} = \vec{\mathcal{F}}(\vec{\mathcal{x}}))$ be an $\mathcal{A}$-μ-condition. If $f \models_{\mathcal{A}} c$, then there is a morphism f' of same codomain as f such*

[6] $Q_{ij} \cong B_i$ is allowed.

that $f' \models \mathcal{A}to\mathcal{M}_{\bar{p}}(b)$ for some surjective $\bar{p}$. If for a graph G, $G \models \mathcal{A}to\mathcal{M}_{\bar{p}}(b \mid \mathcal{F})$, then $G \models (b \mid \mathcal{F})$.

Proof. Induction over nesting within induction over fixed point iteration. These are the interesting cases for the inner induction step of the proof of $G \models_{\mathcal{A}} c \Rightarrow G \models \mathcal{A}to\mathcal{M}_{\bar{p}}(b)$.

$\mathcal{A}to\mathcal{M}_{\bar{p}}(c)$ where $c = \exists(B \overset{\bar{a}}{\twoheadrightarrow} R)$ and $\bar{p} : B \twoheadrightarrow Q$: Assuming $f \models_{\mathcal{A}} c$, factorised as $f = m \circ e$ by induction assumption. Also $f \models_{\mathcal{A}} c$, hence there is a morphism $f' : R \hookrightarrow G$ such that $f = f' \circ \bar{a}$. If f is to ($\mathcal{M}$-)satisfy $\mathcal{A}to\mathcal{M}_{B \twoheadrightarrow Q}(c)$, there must be a graph Q' and morphisms $u : Q' \hookrightarrow G$ and $\bar{f}' : R \twoheadrightarrow Q'$ such that $u \circ \bar{f}' = f'$ by epi-mono-factorisation. However the epi-mono factorisation of f is unique (cf. [EEPT06]), therefore only $Q \cong Q'$ makes the diagram commute and if $\bar{f}$ fails to factor through $\bar{a}$, this is not possible.

$\mathcal{A}to\mathcal{M}_{B \twoheadrightarrow Q}(c)$ where $c = \exists(B \overset{a^\circ}{\hookrightarrow} R)$: as in the right hand diagram (Figure 3.6), with disjunction over all variants with epimorphism that composes to $\mathcal{M}$ on the side opposite to a° to build the new $\mathcal{M}$-satisfying $\mathcal{M}$-morphism. The pushout again consists of a mono- and an epimorphism[7]. $f \models_{\mathcal{A}} c \Rightarrow \exists q, q \circ a^\circ = f$: assume that $f = f' \circ \bar{p}$ as an epi-mono-factorisation. Note that this assumption is trivial for the interface of the main body and that it automatically holds for $q = x \circ e \circ \bar{p}'$, allowing the induction hypothesis to be used for that decomposition and $\mathcal{A}to\mathcal{M}_{e \circ \bar{p}'}(c)$.

$\mathcal{A}to\mathcal{M}_{\bar{p}}(c)$ where $c = \exists(B \overset{\iota}{\hookleftarrow} R)$ and $\bar{p} : B \twoheadrightarrow Q$: observe that post-composition of ι' with the morphism $p^\circ : Q \hookrightarrow G$ yields the suitable situation to complete the induction step.

At variables, $\mathcal{A}to\mathcal{M}_{B \twoheadrightarrow Q}(\mathtt{x}_i) = \mathtt{x}'_{ij}$ (look up the index j associated with $B \twoheadrightarrow Q$). Per assumption of the outer induction, the new equation $\mathcal{A}to\mathcal{M}_{B_i \twoheadrightarrow Q_{ij}}(\mathtt{x}_i)$ holds.

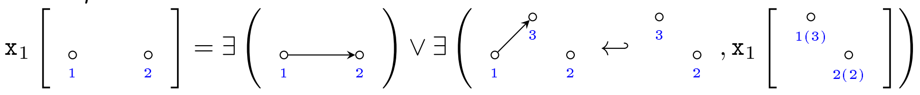
For the converse, observe that if $f \models_{\mathcal{A}} \mathcal{A}to\mathcal{M}_{\bar{p}}(c)$, composition with appropriate epimorphisms from the top part of the diagrams in Figure 3.6 always remains possible and yields satisfaction of c by f. □

We conclude the subsection with an example of an $\mathcal{A}$-μ-condition and the equivalent $\mathcal{M}$-μ-condition obtained using Construction 2. The $\mathcal{M}$-μ-condition has two variables because the nodes in the type of $\mathtt{x}_1$ in the original $\mathcal{A}$-μ-condition need not be distinct, and in $\mathcal{M}$-semantics, this yields two distinct cases.

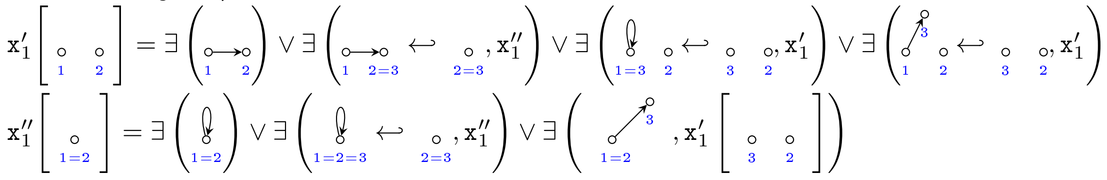
Example 10 ($\mathcal{A}to\mathcal{M}$ for recursively nested conditions)**.**

[7]This is true for graphs and generally in adhesive categories [LS04].

An $\mathcal{A}$-μ-condition:



The resulting $\mathcal{M}$-μ-condition:



3.4.1. Relationship to Fixed Point Logic

In the following, we make precise the relationship between μ-conditions and first-order fixed point logic for graphs. Following Pennemann [Pen09], Rensink [Ren04, Ren06] and Courcelle [Cou97], we speak of first-order *graph* formulae when the theory of graphs is used. By this we mean a certain first-order signature with:

- A ternary relation symbol $\mathrm{inc}(x, y, z)$ (interpreted as the incidence of edge x with source node y and target node z) and
- equality $x = y$

cf. Definition 3.21 in [Pen09], where the predicate symbols $lab_b(x)$ for each b in a finite alphabet of node and edge labels are also introduced. We leave these out for now and discuss unlabelled graphs; it is easy to check that adding labels poses no additional problems.

The unary predicates node and edge can be defined from the inc predicate.

FO+lfp is the extension of first order logic by least fixed points. Formulae are formed as in first-order logic, with the supplementary possibility that if φ is a formula with $\vec{x}$ among its free variables and a free second-order variable R of arity k such that φ is *positive in* R, then $lfp_{R,\vec{x}}(\varphi)(\vec{t})$ is a formula, where $\vec{t}$ is a k-tuple of terms.

A free second-order variable of arity k is a relation symbol of arity k that does not appear in the signature, and φ being positive in R means that R appears only under even numbers of negations in φ. First-order fixed-point graph formulae are then FO+lfp for the theory of graphs.

Satisfaction: $lfp_{R,\vec{x}}(\varphi)(\vec{t})$ is interpreted as a k-ary relation defined as the least fixed point of φ, interpreted as an operator on k-ary relations. Otherwise, the semantics is defined in precisely the same way as for first order logic. Satisfaction of a *graph* formula by a graph follows Pennemann and Courcelle: the graph G is regarded as a structure (D_G, I_G) in the straightforward way by taking $D_G = V_G + E_G$ as the universe and I_G according to the incidence data s_G, t_G.

As mentioned before, we are only concerned with finite graphs. On the logic side this means we are dealing with interpretations over finite structures. Fixed point logic can express transitive closures of binary relations:

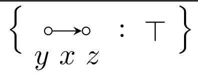
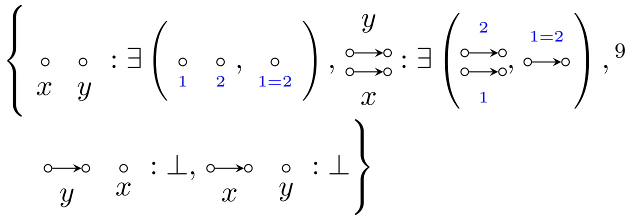
Example 11 (Transitive closure of a binary relation)**.**
$\varphi(x, y) := \mathit{lfp}_{R,xy}\,(E(x, y) \vee \exists z.\, E(x, z) \wedge R(z, y))$
Notice the open variables (x, y) mentioned under the operator. They play the same role as the interface of μ-conditions. The formula φ is interpreted as the least fixed point of the relation R taking the part under the constructor $\mathit{lfp}_{R,xy}$ as the recursive definition.

The above would be an example for a path expression for graphs as they are usually defined in other areas of mathematics, i.e. undirected simple graphs consisting of a vertex set V and an edge set $E \subseteq V \times V$, corresponding to the binary predicate E. Graph formulae, as described above, use variables for both nodes and edges.

[Pen09] (Theorems 3.25 and 3.28) states the correctness of a transformations from formulae in first order graph logic to nested graph conditions and vice versa (Cond resp. Form). In our case, the equivalence likewise relies on translations from μ-conditions to formulae and vice versa which preserve satisfaction.

The transformation of a FO+lfp formula goes along the same general lines as the Cond transformation from [Pen09]. Like Cond, it relies on the transformation to $\mathcal{A}$-satisfaction, and transforms a formula by induction over the structure of the formula. For variables, various possibilities are maintained according to whether it corresponds to a node or to an edge, and its incidence. Conds$_\mu$ differs a little from Cond:[8] Let Var be a countable set of variables used in formulae.

To a formula, it assigns not one but several conditions. The members of Conds$_\mu(\varphi)$ are pairs (c, υ) of a condition c whose type graph B is annotated with a partial mapping $\upsilon : V_B \cup E_B \rightharpoonup \mathrm{Var}$ from nodes and edges to free variables of φ, such that υ^{-1} is a total function. Unlike Cond, Conds$_\mu$ does not immediately produce a disjunction but keeps distinctions between the various possible assignments at a meta-syntactic level until a logical variable is bound.

φ	Conds$_\mu(\varphi)$
inc(x, y, z)	
$x = y$	
$\varphi' \wedge \varphi''$	$\{(c' \wedge d', \upsilon') \mid (c, \upsilon_c) \in \mathrm{Conds}_\mu(\varphi'), (d, \upsilon_d) \in \mathrm{Conds}_\mu(\varphi''), (\iota_c, \iota_d, \upsilon') \in \mathrm{u}(\upsilon_c, \upsilon_d), c' = \exists^{-1}(\iota_c, c), d' = \exists^{-1}(\iota_d, d)\}$ (u defined below)

[8]Chiefly a matter of convenience of presentation, or author's preference. The intuition remains the same.
[9]The nodes are identified in the only possible way.

$\neg\varphi'$	$\{(\neg c, \upsilon) \mid (c, \upsilon) \in \mathrm{Conds}_\mu(\varphi)\}$
$\exists x.\varphi'$	$\{(\exists(\mathrm{dom}(\upsilon') \subseteq \mathrm{dom}(\upsilon), c), \upsilon') \mid (c, \upsilon) \in \mathrm{Conds}_\mu(\varphi')\}$ ($\subseteq$: graph inclusion), where υ' is defined as υ but undefined for all nodes or edges i with $\upsilon(i) = x$; nodes which are no longer the source or target of an edge in $\mathrm{dom}(\upsilon)$, as well as edges which are no longer in $\mathrm{dom}(\upsilon)$, are removed in $\mathrm{dom}(\upsilon')$ (so the type graph contains only those items that are referenced, directly or indirectly, by some logical variable)
$R(\vec{x})$	$\{\mathtt{x}_{R,B} \mid B \in A(\vec{x})\}$ (see below)
$\mathit{lfp}_{R,\vec{x}}(\varphi(R, \vec{x}))$	see below
	(posit equations $\mathtt{x}_{R,B} = \mathrm{RHS}_{R,B}$, then $\mu[\overrightarrow{\mathtt{x}_R}]\mathcal{F}(\overrightarrow{\mathtt{x}_R})$ with these equations)

With a list of k distinct first-order variables $\vec{x}$ we associate the set $A(\vec{x})$ of all pairs (B, υ) such that for each node $v \in V_B$, either $\upsilon(v)$ is defined or it is incident to an edge e such that $\upsilon(e)$ is defined, and for each edge $e \in E_B$, $\upsilon(e)$ is defined, and no unnecessary identifications between unmapped nodes are made. It is simple to show that the number of such pairs (up to graph isomorphisms) is finite.

A term built from a second-order variable $R(\vec{x})$ is thus mapped to a set of condition variables. At last, least fixed point formulae $\mathit{lfp}_{R,\vec{x}}(\varphi(R, \vec{x}))$ are handled by taking for each element $G \in A(R)$ the disjunction of those elements of $\mathrm{Conds}_\mu(\varphi)$ whose types unify with G, thus providing the right hand side $\mathrm{RHS}_{R,G}$ for $\mathtt{x}_{R,G}$ in a system of equations that defines a syntactically positive operator because φ is positive in R.

The transformation u attempts to unify the annotated type graphs by finding a jointly surjective pair of monomorphisms into a common domain that respects the variable assignments. Its output is the list of all such possibilities: $\mathrm{u}(\upsilon, \upsilon') = \{(\iota, \iota', \upsilon'') \mid \exists X, \iota : \mathrm{dom}(\upsilon) \hookrightarrow X, \exists \iota' : \mathrm{dom}(\upsilon') \hookrightarrow X' \text{ jointly surjective}\}$.

In the case of $x = y$, the necessity to add the $\perp$ conditions is motivated by what would happen with the statement $x \neq y$ otherwise, when x happens to be a node and y an edge, such as in the formula $\mathrm{inc}(x, y, z) \wedge x \neq y$. It should evaluate to $\top$, but the corresponding possibility would have gotten thrown out when forming the Cartesian product, resulting in the incorrect condition $\perp$.

Example 12 (An example application of the Conds_μ transformation)**.**
Figure 3.7 shows the transformation of a formula into a condition. The example makes scant use of the more involved constructions and especially not of fixed points, but it demonstrates how the basic recursive definition of the transformation works.

Recall the characterisation of the least fixed point of a monotonic operator as an infinite disjunction, which is valid for the operators in FO+lfp as well.

Lemma 4. *There is a transformation* Conds_μ *such that* $G \models F$ *iff* $G \models_{\mathcal{A}} \mathrm{Conds}_\mu(F)$*.*

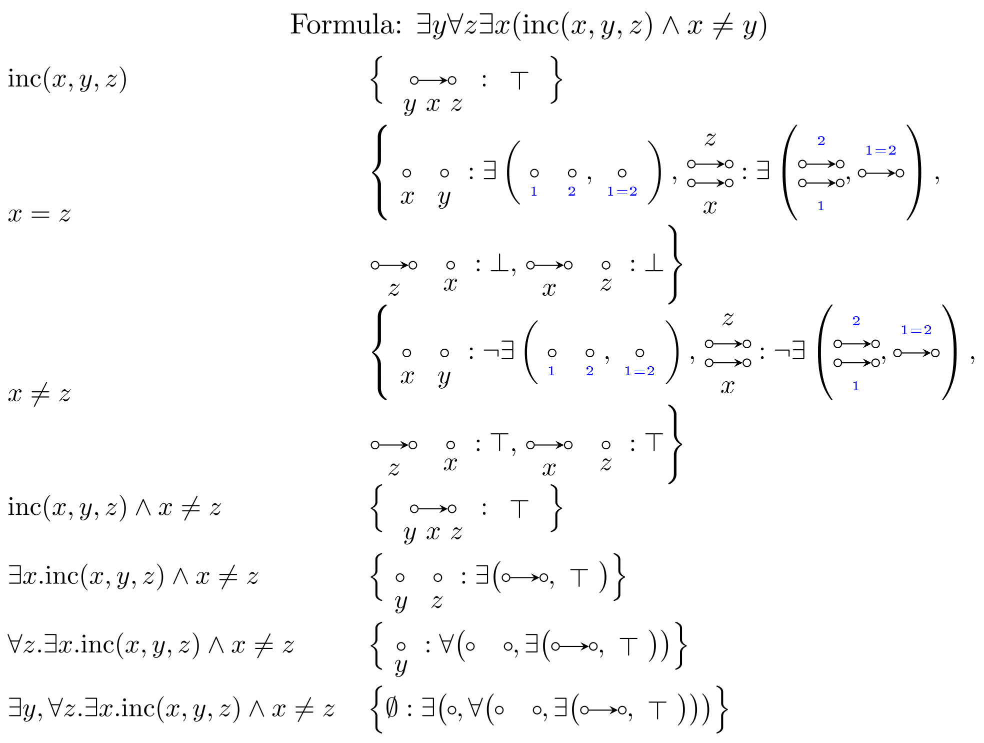

Formula: $\exists y \forall z \exists x(\mathrm{inc}(x,y,z) \wedge x \neq y)$
$\mathrm{inc}(x,y,z)$
$x = z$
$x \neq z$
$\mathrm{inc}(x,y,z) \wedge x \neq z$
$\exists x.\mathrm{inc}(x,y,z) \wedge x \neq z$
$\forall z.\exists x.\mathrm{inc}(x,y,z) \wedge x \neq z$
$\exists y, \forall z.\exists x.\mathrm{inc}(x,y,z) \wedge x \neq z$

Figure 3.7.: An example Conds_μ application with no recursion.

Proof. We treat the fixed points by induction over the fixed point iteration. The base case, $\bot$ on both sides, is trivial. Assume, by way of induction step, that the condition $\mathcal{F}^i_R(\mathtt{x}_{R,B})$ holds for exactly the subgraphs induced by the relation defined by i-fold application of $\varphi(R, \vec{x})$. By induction over the structure of a formula, the right hand sides for $\mathtt{x}_{R,B}$ must be shown to be satisfied by the subgraphs induced by the relation defined by $i+1$-fold application: the cases of $\mathrm{inc}(x,y,z)$ and $x = y$ are straightforward from the semantics: $[\![\mathrm{inc}(x,y,z)]\!] = \{(x,y,z) \in D_G \mid x \in E_G \wedge y,z \in V_G \wedge s_G(x) = y \wedge t_G(x) = z\}$, which corresponds to all the morphisms[10] at which the sole condition in $\mathrm{Conds}_\mu(\mathrm{inc}(x,y,z))$ can be evaluated (to $\top$), and $\mathrm{Conds}_\mu(x = y)$ is likewise satisfied by precisely those morphisms which select twice the same node or edge[11]. For $\varphi' \wedge \varphi''$, the meaning of the construction is just a shift with unselection, subject to consistency of the variable mappings. $\square$

The transformation $\mathrm{Form}_{\mathit{lfp}}(c)$ that we introduce now relies upon a normal form (Fact 3.29 in [Pen09]) where every morphism is decomposed into its finitely many *elementary* nontrivial morphisms, to with gluing of two items $[e = e']_B : B \to B'$ or addition of one node $[u]_B : B \to B'$ or edge $[euv]_B : B \to B'$. It is easy to show that such a decomposition is always finite for graph morphisms (in general, such finiteness assumptions need to be

[10]We would speak of subgraphs, but these are non-injective in general.
[11]The latter with their source and target nodes, as this is unavoidable in graph conditions.

added to the axioms for adhesive categories). The construction is well-defined in spite of the decomposition not being uniquely determined.

In FO+lfp, simultaneous fixed points are syntactic sugar that allows a least fixed point to be expressed by simultaneous definitions: a n-tuple of formulae $\varphi_i(R_1, ..., R_n, \vec{x}_i)$ which have n free second-order variables for defining n relations simultaneously (Def. 3.16 in [Kre02]). To each second-order variable R_i, a list of variables $\vec{x}_i$ of the corresponding arity is allocated. The construction is as follows:

c	$\text{Form}_{\mathit{lfp}}(c)$
$\exists(\text{id}, c)$	$\text{Form}_{\mathit{lfp}}(c)$
$\exists([u], c)$	$\exists u.(\text{node}(u) \wedge \text{Form}_{\mathit{lfp}}(c))$
$\exists([euv], c)$	$\exists e.(\text{edge}(e) \wedge \text{inc}(e, u, v) \wedge \text{Form}_{\mathit{lfp}}(c))$
$\exists([u = v], c)$	$u = v \wedge \text{Form}_{\mathit{lfp}}(c)$
$\exists^{-1}(\iota, c)$	$\text{Form}_{\mathit{lfp}}(c)$
$\neg c$	$\neg\text{Form}_{\mathit{lfp}}(c)$
$\bigvee_{i\in I} c_i$	$\bigvee_{i\in I} \text{Form}_{\mathit{lfp}}(c_i)$
$\mathtt{x}_i$	$R_{\mathtt{x}_i}(\vec{x}_i)$
$\mu[\overrightarrow{\mathtt{x}}]\mathcal{F}(\overrightarrow{\mathtt{x}})$	$\mathit{lfp}_{R_{\mathtt{x}_1},\dots,R_{\mathtt{x}_n}}\{\text{Form}_{\mathit{lfp}}\mathcal{F}_i(\overrightarrow{\mathtt{x}})\}_{i\in\{1\dots\|\overrightarrow{\mathtt{x}}\|\}}$: a simultaneous fixed point

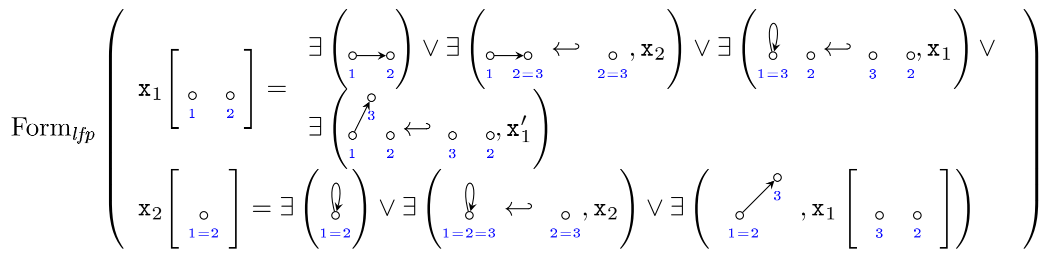
Here, $[u]_P : B \hookrightarrow B'$ (or, leaving B implicit, $[u]$) is defined for any graph B such that $u \notin V_B$ as the inclusion of B into the graph B' which differs from B by $V_{B'} = V_B \cup \{u\}$. $[euv]_B : B \hookrightarrow B'$ is the inclusion of B into the graph B' which has an additional edge $e \notin E_B$ between the nodes $u, v \in V_B$ (i.e. $s_{B'}(e) = u$ and $t_{B'}(e) = v$). If u and v are nodes of B, then $[u = v]_B : B \twoheadrightarrow B'$ is the morphism such that $E_{B'} = E_B$, $V_{B'} = V_B - \{v\}$, $[u = v]_V(v) = u$ and $[u = v]_V(w) = w$ otherwise, $[u = v]_E = id_{E_B}$. $s_{B'}$ and $t_{B'}$ are adapted. If u and v are parallel edges of B, then $[u = v]_B : B \twoheadrightarrow B'$ is the morphism s.t. $V_{B'} = V_B$, $E_{B'} = E_B - \{e\}$, $[u = v]_V = id_{V_B}$ and $[u = v]_E(v) = u$ and $[u = v]_E(w) = w$ otherwise.

Example 13 (An example application of the $\text{Form}_{\mathit{lfp}}$ transformation)**.** *Figure 3.8 shows the transformation of a condition into a formula. The example contains a single fixed point.*

Let us show that the transformation $\text{Form}_{\mathit{lfp}}$ fulfils its purpose.

Lemma 5. *There is a transformation* $\text{Form}_{\mathit{lfp}}$ *such that* $G \models_A c$ *iff* $G \models \text{Form}_{\mathit{lfp}}(c)$.

Proof. By induction over the structure of the condition to be transformed. Most cases are straightforward from the semantics of the logic and of the conditions. Again, the case of a fixed point operator is handled by induction over the fixed point iteration. The simultaneous fixed point formulation allows the stages of the fixed point iteration to be compared, as we can now show by induction that at each stage, the $\|\vec{\mathtt{x}}\|$ relations



$$= lfp_{R_{\mathtt{x}_1},R_{\mathtt{x}_2},x_{11},x_{12},x_{21}} \begin{cases} R_{\mathtt{x}_1} : & \begin{array}{l} \exists e.\,(\mathrm{inc}(e,x_{11},x_{12})) \vee \\ \left(\begin{array}{l} \exists e.\,(\mathrm{inc}(e,y,x_{12}) \wedge R_{\mathtt{x}_2}(x_{12})) \vee \\ \exists e.\,(\mathrm{inc}(e,x_{11},x_{12}) \wedge R_{\mathtt{x}_1}(x_{11}x_{12})) \vee \\ \exists z.\,(\mathrm{node}(z) \wedge \exists e.\,(\mathrm{inc}(e,x_{11},z) \wedge R_{\mathtt{x}_1}(zx_{12}))) \end{array} \right) \end{array} \\ R_{\mathtt{x}_2} : & \begin{array}{l} \exists e.\,(\mathrm{inc}(e,x_{11},x_{11})) \vee \\ \left(\begin{array}{l} \exists e.\,(\mathrm{inc}(e,x_{11},x_{11}) \wedge R_{\mathtt{x}_2}(x_{11})) \vee \\ \exists z.\,(\mathrm{node}(z) \wedge \exists e.\,(\mathrm{inc}(e,x_{11},z) \wedge R_{\mathtt{x}_1}(zx_{11}))) \end{array} \right) \end{array} \end{cases}$$

Figure 3.8.: An example Form$_{lfp}$ application with a single fixed point.

correspond exactly to the sets of morphisms of codomain G that satisfy the components of $\vec{\mathcal{F}}^i(\vec{\mathtt{x}})$. □

Notice that nested fixed points pose no problem in either direction, as it is perfectly admissible for a fixed point operator not to bind all variables at once, both in FO+lfp and in μ-conditions.

Theorem 2. *μ-conditions are equally expressive as first-order logic with least fixed point over finite graphs.*

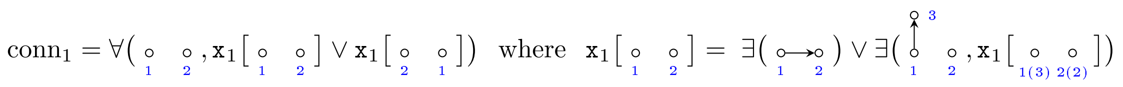
Proof. By Lemma 4 and Lemma 5. □

3.4.2. Expressing String Languages

Strings can be treated as special graphs, allowing graph condition formalisms to be compared on the kinds of string languages they are able to express.

Notation. *Given a graph, we define the* in-degree *of a node v as the number of edges with target v, its* out-degree *as the number of edges with source v and its* degree $\deg(v)$ *as the pair of its in- and out-degree, in that order.*

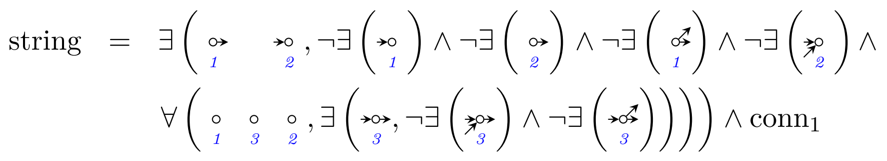
The property of being a string graph can be expressed in every non-local graph condition formalism that has been devised: the condition on the node degrees is already definable in first order, or nested graph conditions. This, together with the right notion of connectedness, defines string graphs. The latter property is just:



conn1 = ∀(, x1[ ] ∨ x1[ ]) where x1[ ] = ∃( ) ∨ ∃(, x1[ ])
1 2 1 2 2 1 1 2 1 2 3 1 2 1(3) 2(2)

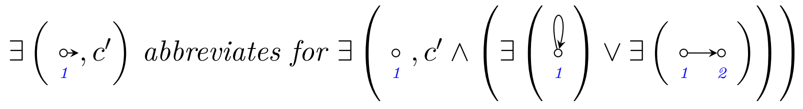
To ensure that a condition is only satisfied by string graphs, it is convenient to conjoin the following to the main body (equivalent versions can be formulated in any of the extensions of nested conditions considered here). The non-connected edges are an abbreviation: they represent disjunctions of possibilities of linking these edges to either the depicted nodes or new nodes.

Construction 3 (String Graph Languages)**.**



string = ∃(, ¬∃( ) ∧ ¬∃( ) ∧ ¬∃( ) ∧ ¬∃( ) ∧
∀(, ∃(, ¬∃( ) ∧ ¬∃( ))) ∧ conn1
1 2 1 2 1 2 1 3 2 3 3 3

where the dangling edges mean that an appropriate disjunction of possibilities[12] *is combined conjunctively with the subcondition:*



By a *unary* language we mean a language over a singleton alphabet. Unary languages are most readily encoded as unlabelled graphs. For all of the graph condition formalisms considered, it is easy to check that the classes of unary string languages that can be expressed do *not* change when working with labelled graphs instead, as labels other than the one in the language's unary alphabet simply fail to match.

Conjunctive context-free grammars (CCFG) are an extension of context-free grammars by conjunctive productions. These grammars express a proper superclass of the context-free languages. The conjunctive context-free languages are a subclass of the FO+lfp definable languages on strings [Okh13]. A direct translation of such grammars to μ-conditions is easily obtained (independently of the translation from and to FO+lfp).

Definition 26 (Conjunctive Context-Free Grammar)**.** *A* conjunctive (context-free) grammar *is a quadruple $G = (\Sigma, N, P, S)$ where Σ and N are disjoint finite sets of* terminal *and* nonterminal *symbols respectively, $S \in N$ is the start symbol, and P is a finite set of ordered pairs (*rules*) of the form $(A, \{\alpha_1, ..., \alpha_n\})$, written as $A \to \alpha_1 \& ... \& \alpha_n$ with $A \in N$ and each* conjunct *α_i is a sequence in $(\Sigma \cup N)^*$.*

The definition is closely related to that of an ordinary context-free grammar. A string is in the language $\mathcal{L}(G)$ if it can be parsed using the rules. [Okh13] gives several equivalent

[12]Not defined here but the abbreviated subcondition can always be constructed similarly to the transformation from $\mathcal{A}$ to $\mathcal{M}$ earlier in this section.

definitions of $\mathcal{L}(G)$. One possibility is to define $\mathcal{L}(\alpha)$ for any $\alpha \in (\Sigma \cup N)^*$ (thus also $\mathcal{L}(S)$) as the set of words whose membership can be deduced using the axiom for terminals, the concatenation rule for conjuncts and the conjunction rule schema (one instance per rule of P) for single nonterminals:

$$\frac{}{s\in\mathcal{L}(s)}\ (s \in \Sigma) \qquad \frac{u\in\mathcal{L}(A) \quad v\in\mathcal{L}(B)}{w\in\mathcal{L}(AB)}\ \begin{array}{c} A \in N \cup T \\ B \in (N \cup T)^* \end{array}$$

$$\frac{w\in\mathcal{L}(\alpha_1) \quad \dots \quad w\in\mathcal{L}(\alpha_n)}{w\in\mathcal{L}(A)}\ (A \to \alpha_1\&\dots\&\alpha_n \in P)$$

A result of Jeż [Jeż07] delivers a striking example of a non-regular unary language expressible by a conjunctive context-free grammar (it is well-known that all regular languages are semilinear):

Example 14 (Non-Semilinear Unary String Language)**.** *We translate the equations from a conjunctive context-free grammar to a recursively nested condition. Each nonterminal becomes a variable of type* $\circ \quad \circ$, *terminals become* $\exists\,(\circ\!\!\longrightarrow\!\!\circ)$ *(labelled, in the case of non-unary alphabets), positive Boolean combinations are represented as such and concatenation* $X \to AB$ *is achieved by*

$$X\begin{bmatrix}\underset{1}{\circ} & \underset{2}{\circ}\end{bmatrix} = \exists\left(\underset{1}{\circ} \quad \underset{3}{\circ} \quad \underset{2}{\circ}, A\begin{bmatrix}\underset{1}{\circ} & \underset{3}{\circ}\end{bmatrix} \wedge B\begin{bmatrix}\underset{3}{\circ} & \underset{2}{\circ}\end{bmatrix}\right)$$

The equations of the counterexample are:

$$\begin{array}{l} A_1 \to A_1A_3\&A_2A_2 \mid a \\ A_2 \to A_1A_1\&A_2A_6 \mid aa \\ A_3 \to A_1A_2\&A_6A_6 \mid aaa \\ A_6 \to A_1A_2\&A_3A_3 \end{array}$$

Translating these equations to conditions with variables results in a μ-condition that is satisfied by the language of (unlabelled) string graphs of lengths $\{4^n \mid n \in \mathbb{N}\}$. For a proof and an explanation of the ingenious construction, we refer to the original paper of Jeż or Okhotin's survey [Okh13].

By Construction 3, we get the following corollary:

Corollary 1 (Unary String Languages of μ-Conditions)**.** *μ-conditions can express non-semilinear unary string languages.*

3.4.3. Relationship to M-Conditions

M-conditions [PP14] have the same expressiveness as monadic second-order graph logic (with quantification over node and edge sets). Hamiltonicity is existence of a Hamiltonian cycle, that is an edge cycle incident once to every node. It is expressible when quantification over edge sets is possible, which it is in M-conditions. Hamiltonicity is not expressible in μ-conditions, by a standard result from finite model theory. Thus we have, at no further expense:

Corollary 2. *There is a property expressible as an M-condition but* ***not*** *as a μ-condition.*

Proof. The existence of a Hamiltonian cycle is not expressible in FO+lfp, hence not μ-conditions, but M-conditions can express it [Lib04]. □

When expressing families of string graphs with M-conditions as defined in [PP14], it is quite sufficient to restrict the conditions syntactically to contain only morphisms whose domain and codomain graphs have nodes of in-degree and out-degree each ≤ 1 because the injective semantics would otherwise not provide any matches: as graphs that are not string graphs must not satisfy the condition, any subcondition violating the node-degree restriction can as well be replaced be $\bot$ or $\top$. This leaves the possibilities $\exists_V X[c]$ which introduces a node set variable X, $\exists_E X[c]$ introducing an edge set variable X, Boolean combinations and $\exists(a \mid \gamma, c')$ where a is an injective graph morphism subject to the node degree restriction (maximum in-degree 1, maximum out-degree 1) and γ is a constraint stating membership of a node in a node set, membership of an edge in an edge set or the existence of a path from a certain node to another, possibly avoiding certain edges. It is well known, and stated in [PP14], that the latter can be expressed using only the remaining means and so can be removed.

Such a restricted M-condition can be transformed into an expression of MSO logic on strings (finite structures of one successor). We sketch the transformation, which starts out by performing the translation into MSO formulae from [PP14], then proceeding to transform conditions $\exists(a \mid \gamma, c)$ with a morphism a requiring the existence of an edge into formulae $\exists a.\varphi$, conditions $\exists(a \mid \gamma, c)$ with a morphism a requiring the existence of a node of degree $(1,1)$ into formulae of the form $a < b$ where a is the logical variable associated with the ingoing edge and b the one associated with the outgoing edge. Hence M-conditions on strings have exactly the same expressiveness as monadic second-order logic on strings. It follows:

Corollary 3. *There is a property expressible as a μ-condition but* ***not*** *as an M-condition.*

Proof. The language $\{a^{4^n} \mid n \in \mathbb{N}\}$, which is not regular and hence not definable in MSO [Tho97] and therefore also not expressible as a M-condition, is a separating example. □

We conclude that a comparison has been achieved:

Corollary 4 ($\mu \lessgtr$ **M**)**.** *M-conditions and μ-conditions are incomparable.*

Proof. By Corollary 2 and Corollary 3. □

The expressiveness of M-conditions itself is known to be strictly included in that of HR*-conditions. In the following subsection, we compare μ-conditions to these.

3.4.4. Relationship to HR*-Conditions

HR* conditions are an extension of nested graph conditions, where the graphs and morphisms that appear in the condition are not fixed, but given by hyperedge replacement grammars. While HR* and μ-conditions seem to have a similar flavour (both being non-local conditions that use variables), the formalisms are unrelated and less similar than they seem at first glance: in the case of HR* the variables stand for subgraphs and in μ-conditions they stand for subconditions, which is a fundamentally different concept.

HR*-conditions are very powerful and subsume M-conditions, even counting-MSO on graphs [Rad13]. We consider HR*-conditions with $\mathcal{M}$-semantics. These are known to be translatable into equivalent $\mathcal{A}$-HR*-conditions ([Rad13], Appendix A). Currently, nothing but second-order logic is known to encompass the whole of HR*-conditions. All attempts to find a tighter bound their expressiveness from above have failed so far. In this subsection, we restrict HR*-conditions slightly and examine their expressive power on unary string languages, as for μ- and M-conditions in the previous section.

Some useful properties lie within this common fragment: both μ-conditions and HR*-conditions are able to express (Boolean combinations of) existential statements about graphs from given hyperedge replacement grammars [Hab92]. Unfortunately, this common fragment is not all that useful because it is not closed under the weakest precondition calculus (a crucial ingredient to our method of proving program correctness, to be introduced in Section 5.1), due to the introduction of new nesting level with a universal quantifier in the weakest precondition with respect to an *unselect* step.

Note however that unlike HR*-conditions with $\mathcal{M}$-semantics, a μ-condition with $\mathcal{M}$-semantics can state the existence of a morphism from a graph generated by the grammar but cannot state the existence *as a subgraph*, that is as an injective morphism: items that are matched by a subcondition $\exists(a, c)$ may be identical to some that had been unselected by a morphism ι at a prior nesting level and this effect cannot be entirely prevented because the size of the type graph of any subcondition and any variable is bounded. HR*-conditions, on the other hand, have a "true" $\mathcal{M}$-semantics.

For the path grammar in Figure 3.9, it turns out not to make a difference since both variants are easily seen to imply each other (Example 15: when a path with repeated nodes exists, then a possibly shorter one without repeated nodes can be constructed from it. When a path without repeated nodes exists as a subgraph, then this is also an example of a path with (zero) repeated nodes). This fact is purely coincidental; it is generally not the case (neither for HR* nor μ-condition) that a condition in $\mathcal{M}$-semantics, even one that uses only existential quantification, has the same meaning as the syntactically identical condition interpreted in $\mathcal{A}$-semantics.

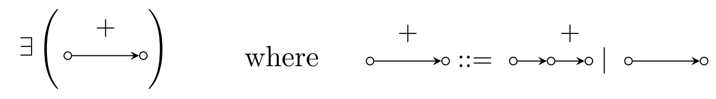


Figure 3.9.: Path grammar and existence of a path in HR*.

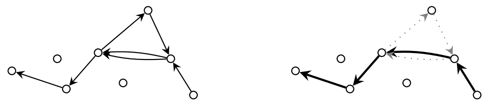


Figure 3.10.: The path-ology of ($\mathcal{M}$-)μ-conditions.

Example 15 (Existence of a path)**.** *Figure 3.10 shows the existence of a path, found with an injective match vs. a non-injective match. In the case of a non-injective match, nothing prevents nodes from being traversed again at a later nesting level or iteration, and the matched path can contain cycles.*

Beyond this common fragment, HR*-conditions can express a host of complex properties. In the following, the definitions are briefly recalled. For thorough definitions, we refer to [Hab92] throughout. Our definitions are somewhat simplified: there is only a single list of pins for a pointed graph or hyperedge.

Notation. *Λ is a finite alphabet of hyperedge labels, Y_G is the set of hyperedges of a hypergraph G.*

To define HR*-conditions, one needs *hyperedge replacement systems*, which are similar to context-free grammars. Nonterminals are so-called *hyperedges* with a list of *attachment points* or *pins*. Replacement is performed on hypergraphs, which are a generalisation of graphs which may also comport hyperedges instead of edges. For our purposes, a hypergraph H is a tuple $(V_H, Y_H, \mathrm{att}_H, \lambda_H)$ of a finite set of nodes V_H and a finite set of hyperedges Y_H, a mapping $\mathrm{att} : Y_H \to V_H^*$ from hyperedges to finite lists of nodes and a labelling function $\lambda_H : Y_H \to \Lambda$. The length of $\mathrm{att}(y)$ is also called the *arity* of the hyperedge $y \in Y_H$.

Remark 7. *A graph can be regarded as a hypergraph whose hyperedges all have arity* 2*.*

A *pointed* hypergraph $(G, p_1, ..., p_n)$ is a hypergraph G together with a list $p_1, ..., p_n$ of some of its nodes, called *pins*. A hyperedge replacement system $\mathcal{R} = (N, T, P)$ consists of a set N of nonterminals and a set T of terminals, and a set P of productions. The components are then also written $N_{\mathcal{R}}$, $T_{\mathcal{R}}$, $P_{\mathcal{R}}$. A production is a pair (A, R) consisting of a nonterminal A (*left hand side*) and a pointed hypergraph R (*right hand side*). One may assume that all hypergraphs R appearing as the left hand side of a given nonterminal have the same number of pins, henceforth also called the *arity* of A and denoted $\mathrm{arity}_{\mathcal{R}}(\mathrm{A})$. In a *derivation step*, some appropriately labelled hyperedge $y \in Y_H$ of the current hypergraph H is replaced with a pointed graph from the right hand side of the production, removing the matched hyperedge and gluing in the right hand side, respecting the ordering of the pins. The resulting hypergraph is denoted $H[y/R]$. A *derivation* is a sequence of derivation steps $H \Rightarrow H[y_1/R_1] \Rightarrow H[y_1/R_1][y_2/R_2]....$ It is said to be *terminal* if the target of its last step contains no nonterminals.

The *language* of a hyperedge x in the hyperedge replacement system $\mathcal{R}$ is the set of hypergraphs[13] obtained by derivations of arbitrary length in $\mathcal{R}$ starting from the hyperedge x and is denoted by $\mathcal{R}(x)$. A hyperedge replacement system together with a (start) hypergraph labelled over $N_{\mathcal{R}} \cup T_{\mathcal{R}}$ is called a *grammar*. Some normal form theorems are known [Hab92], for example it is never necessary for a nonterminal to appear with fused pins and unproductive nonterminals, whose language is empty, can be removed as for context-free grammars.

A HR*-condition is a pair $(c, \mathcal{R})$ of a *condition with nonterminals*[14] c and a hyperedge replacement system $\mathcal{R}$. The *nonterminals* that appear in the graphs are labelled hyperedges, whose labels also appear as left hand sides of productions with the apposite arities in $\mathcal{R}$. The definition of a condition below is based on [Rad13] (in the new reference [Rad16], the presentation was streamlined but the notion is the same):

- If $a : P \hookrightarrow C$ is a monomorphism and c is a HR*-condition over C, then $\exists(a, c)$ is a HR*-condition over P.
- For any two graphs P and C and HR*-condition c over C, $\exists(P \sqsupseteq C, c)$ is a HR*-condition over P.
- As well as the Boolean combinations and true.

The semantics of HR*-conditions is based on *substitutions* obtained from the grammars of the hyperedge replacement system. A graph G satisfies a condition c iff it satisfies the condition *by* a substitution, which is a mapping from the nonterminals to graphs issued from the respective languages. $\Sigma_{\mathcal{R}}$ is the set of all substitutions induced by $\mathcal{R}$ and the image of a labelled hypergraph under the substitution σ, which is obtained by replacing all labelled hyperedges by their images under σ, is denoted P^σ. Substitutions may be partial, which is useful in case some nonterminals do not occur. The noteworthy cases in the semantics are:

- $\exists(a, c)$ for $a : P \hookrightarrow C$ is satisfied *by* a substitution $\sigma \in \Sigma_{\mathcal{R}}$ and a morphism $p : P^\sigma \hookrightarrow G$ if there is a partial substitution τ that agrees with σ on P ($P^\sigma = P^\tau$) and a monomorphism $q : C^\tau \hookrightarrow G$ such that $q \circ a^\tau = p$ and $q \models_\tau c$, where $a^\tau : P^\sigma \hookrightarrow C^\tau$ agrees with a on all items which are not hyperedges and maps all hyperedges to themselves.
- $\exists(P \sqsupseteq C, c)$ is satisfied by σ and a morphism $p : P^\sigma \hookrightarrow G$ if there is a substitution τ that agrees with σ on P, an inclusion $C^\tau \subseteq P^\sigma$ and a monomorphism $q : C^\tau \hookrightarrow G$ such that $q = p\,|_{C^\tau}$ (i.e. the restriction of q to P^σ)[15] and $q \models_\tau c$.
- As well as the Boolean combinations and true.

[13]One can define hyperedge replacement systems that produce families of hypergraphs, but in this section, expressiveness with respect to graphs is examined. All labels of arity other than 2 are nonterminals and the hyperedge replacement system merely serves to define sets of plain (but pointed) graphs.

[14]In the original, these are called *conditions with variables*, which clashed with our terminology. The variables introduced in this chapter stand for conditions, while those used in HR*-conditions indicate non-terminals of graph grammars.

[15]When comparing with [Rad13], notice that the restriction makes sense in this direction.

Note that the condition $\exists(P \sqsupseteq C, c)$ is a source of complication. It does not directly impose any relationship between the items of P and C: any substitution that yields a subgraph will do, and unlike in the semantics $\exists(a, c)$ there is no constraint on the individual nodes, edges, hyperedges. When the same nonterminal is used in P and in C, the switch from σ to τ does not guarantee that for any specific nonterminal $\sigma(x) = \tau(x)$. It does, however, impose an overall constraint via $P^\sigma = P^\tau$ that must not be overlooked.

As a stepstone towards the notoriously powerful HR*-conditions, we introduce HR$^-$-conditions. These are the same as HR*-conditions but subject to these restrictions:

Definition 27 (**HR$^-$-conditions**). *Same definition as HR*, but*

- *all morphisms under $\exists(a, c)$ are injective and only injective matches are permitted ($\mathcal{M}$-semantics[16]).*
- *the hyperedge replacement system generates only graphs with a bounded number of connected components (*non-proliferating*).*

Please note that the proposed restriction leads to a proper subclass in terms of expressible properties, for a reason that also separates node-counting monadic second-order logic (CMSO) from HR$^-$.

Fact 7 (**HR$^-$ $\subsetneq$ HR***). *HR$^-$-conditions cannot express all node-counting monadic second-order properties, because grammars generating nodes with edges are not useful when expressing families of discrete graphs: all productions creating edges can be eliminated by* integrating *[Rad16] the condition of discreteness. A grammar generating arbitrarily large discrete graphs cannot be non-proliferating. A HR$^-$-condition that specifies a set of discrete graphs is merely equivalent to some nested condition.*

On the other hand, paths and even Hamiltonicity can still be expressed, which shows that HR$^-$-conditions remain very powerful.

A *semilinear set* of dimension d is a finite union of sets of the form $\vec{x} + \bigcup_{j \in J} c_j \cdot \vec{y}_j$, where $\vec{x} \in \mathbb{Z}^d$, the index set J is finite and $\forall j \in J,\ c_j \in \mathbb{N} \ \wedge\ \vec{y}_j \in \mathbb{Z}^d$. A language of string graphs is said to be semilinear if the corresponding language of strings is. For string languages, the term refers to the image of the language under the letter count, or Parikh, mapping (cf. Chapter VI of [Hab92]). When the alphabet is unary, the multisets which are the images of the words under the letter count mapping are just integers and semilinear sets are unions of arithmetic progressions. It is well known that Presburger Arithmetic, i.e. first-order logic with integers, addition and $\leq$, defines precisely the semilinear sets.

As was argued for M-conditions in Subsection 3.4.3, it is safe to assume that all hyperedge replacement systems only produce graphs whose nodes have degree at $(0,0)$, $(1,0)$, $(0,1)$ or $(1,1)$: briefly, a hyperedge replacement system can always be transformed into one that produces only disjoint unions of strings (called a *string grammar* in the following

[16]$\mathcal{A}$-semantics was only introduced as a variant in [Rad13], so this is not a restriction at all.

text) by using a larger set of nonterminals annotated with degree information at each pin and then removing all productions which would violate that property. Therefore this property can be assumed for any HR$^-$-condition that is satisfied only by string graphs.

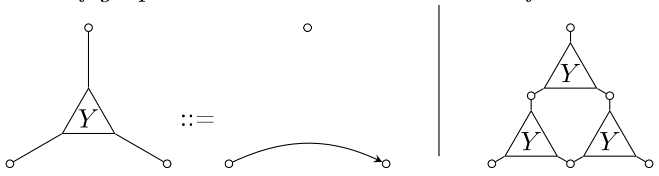
We prove some technical results on hyperedge replacement systems and the property of non-proliferation.

An overloaded notation is used: if Y is a nonterminal of $\mathcal{R} = (N, T, S)$, then Y also denotes the hypergraph that consists of just one hyperedge Y attached to $\mathrm{arity}_{\mathcal{R}}(\mathrm{Y})$ distinct nodes.

Notation. *A* partitioning *of a set A is a set of disjoint nonempty subsets of A whose union equals A. Let $\Pi(A)$ denote the set of all partitionings of A.*

Let us propose an example of a hyperedge replacement system with a single nonterminal and a single rule, which is a string grammar in the above-given sense despite having a nonterminal with three pins:

Example 16 (A hyperedge replacement system)**.** *The following hyperedge replacement system generates a set of graphs with unbounded numbers of connected components:*

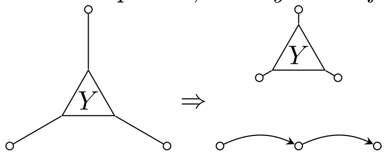


Now for the restriction that distinguishes HR$^-$ from HR*:

Definition 28 (Non-proliferation).
A hyperedge replacement system $\mathcal{R} = (N, T, P)$ is said to be non-proliferating *iff there is a bound $b \in \mathbb{N}$ on the number of connected components of any graph G such that $Y \overset{*}{\Rightarrow} G$ for $Y \in N$.*

A hyperedge replacement system is said to be *proliferating* if there is no such bound. The replacement system of Example 16 is proliferating, as the following derivation can be repeated, yielding an unbounded number of connected components:

Example 17 (Proliferation)**.**
Each application of the rule disconnects the bottom two pins of the nonterminal from the pin at the top, increasing the number of components permanently by 1 for the rest of the derivation. Since the process can be repeated, the system of Example 16 is proliferating.



With an appropriate start graph, this hyperedge replacement system is a string grammar because it generates only graphs of maximum degree $(1, 1)$. Being proliferating, it cannot be used in a HR^--condition.

We now show that non-proliferation is decidable, and we also show how information about the possible configurations of connected components issued from a hyperedge (and which pins are linked by these components) can be extracted. The decidability result holds for all grammars, although it is shown only for string grammars (i.e. grammars that generate disjoint unions of string graphs).

Remark 8 (Normal form)**.** *A hyperedge replacement system can be brought into a Chomsky-esque normal form: a hyperedge replacement system with the same language as a given one (for a given start graph), where each production has either two or zero nonterminals in every right hand side. This is achieved by factoring out a new nonterminal with* $\mathrm{arity}(\mathrm{Y}_1) + \mathrm{arity}(\mathrm{Y}_2)$ *pins for each pair of hyperedges labelled* Y_1*,* Y_2*, or a new nonterminal that generates all the terminal edges present in right hand side, as long as needed. This process clearly terminates, yielding an equivalent system.*

The normal form helps present the following result, which is used in the proof that some μ-condition expressible unary string languages are not expressible by any HR^--condition. The result we are about to prove states that for string grammars as defined above, not only is non-proliferation decidable but one can also extract a formula in Presburger arithmetic that describes the possible lengths of the connected components, both those linked to nodes preserved from the start graph and those (finitely many due to non-proliferation) which are newly created in a derivation. An auxiliary construction analyses the hyperedge replacement system and allows to determine, for each nonterminal:

1. which of its pins can end up being linked by string graphs after a terminal derivation,
2. the possible lengths of these strings.

Construction 4 (**Extracting Link Information**)**.**
Let $(\mathcal{R}, P)$ *be a string grammar. Assume that* $\mathcal{R}$ *is in the abovesaid normal form. Assume by the same technique that* P *contains a single hyperedge* y *of nonterminal label* Y*. We initialise the following* replacement parameters*:*

- *A natural number* k
- *An assignment* $\mathrm{ports} : \{1, ..., \mathrm{arity(y)}\} \to 2^{\{+,-\}}$ *is induced by P: for each pin of* y*, set* $\mathrm{ports} = \{+\}$ *if there is an ingoing terminal-labelled edge in* P *to that pin,* $\{-\}$ *if there is an outgoing edge,* $\emptyset$ *if there are both, or more than one in-/outgoing edge,* $\{+, -\}$ *otherwise.*
- *An injective mapping* $\mathrm{color} : \sigma \uplus \{1, ..., k\} \to \mathbb{N}$.
- *A partitioning* σ *of* $\{1, ..., \mathrm{arity(y)}\}$ *is* valid *with respect to* ports *if it contains only sets of size at most two, and each* $\{p, q\} \in \sigma$ *has* $+ \in \mathrm{ports}(p)$ *and* $- \in \mathrm{ports}(q)$ *or vice versa. Notice that the number of possible* ports *and* σ *is finite for each nonterminal.*

We will define a procedure that, when given this data, attempts to return a formula with free variables $\#1, \#2, ...\#|\sigma|, \#(|\sigma|+1), ..., \#(|\sigma|+k)$ *which describes exactly the lengths of the connected components of graphs generated by* $(\mathcal{R}, P)$ *under the assumption that the pins of* y *are linked up in the way indicated by the partition* σ*. If* $\mathcal{R}$ *is non-proliferating, then this should be the desired result. If it is not, then the computation should abort, reporting the failure.*

Let $\tilde{Y} = \{Y_{k,\text{ports},\sigma} \mid Y \in N_{\text{R}}, k \in \mathbb{N}, \text{ports} \in \{1, ..., \text{arity(y)}\} \to 2^{\{+,-\}}, \sigma \in$ *valid partitions of* $\{1, ..., \text{arity(y)}\}$ *under* $\text{ports}\}$.

For each production $(Y, R) \in P$*, let* $\Theta(Y, \text{ports}, \sigma, \text{color})$ *be the following set of productions:*

- *If* P *contains an edge whose direction*[17] *is inconsistent with* σ *(*$+$ *must mark the source and* $-$ *the target of an edge), then it is empty.*
- *Otherwise, if* R *contains only terminals, then* $\{(Y_{n,\text{ports},\sigma}, \mathfrak{CCC}(R))\}$ *for every possibility for* ports, σ*, where* n *is the number of connected components of* R *which do not include a pin of* R*.* $\mathfrak{CCC}(R)$[18] *differs from* R *in that each terminal* T *of an edge* $e \in Y_R$ *is changed to* (c, T)*,* c *being the color that* color *assigns to the pins of* R *in the connected component of* e*.*
- *Otherwise, there are by assumption two nonterminal hyperedges* u*,* v *with* $\lambda_R(u) = U$*,* $\lambda_R(v) = V$ *in* Y_R*. Return* $\{(Y_k, R') \mid k \in \mathbb{N}, R' \in \text{C}_k(R, \text{ports}, \sigma, \text{color})\}$*. The set* $\text{C}_k(R, \text{ports}, \sigma, \text{color})$ *is defined as follows: for every solution of* $k = l + m + n$ *in positive integers, change the label of* u *to* $U_{l,\text{ports}_u,\sigma_u,\text{color}_u}$ *and* v *to* $V_{l,\text{ports}_v,\sigma_v,\text{color}_v}$*, where the indices* ports_u *and so on run through all choices consistent with the requirement that a string grammar should result.* color *is extended injectively to* $\text{color}_{u'} : \sigma_{u'} \to \mathbb{N}$ *for* $u' \in \{u, v\}$ *subject to the consistency requirement that* $\text{color}_{u'}$ *coincides with* $\text{color}_{v'}$ *on shared or linked pins, and new values do not clash between descendants of* U *and of* V*.*

$\mathcal{R}$ *is non-proliferating iff the languages of all nonterminals whose index* k *exceeds a certain number* $b \in \mathbb{N}$ *are empty: it is clear by construction and can be proven by induction over* $\mathbb{N}$ *that* k *equals to the number of connected components (those not containing a pin of* P*) that will be created in a terminal derivation. To determine that number, search for a cycle* $Y_{k,\text{ports},\sigma} \Rightarrow G$ *with* $y' \in Y_G$*,* $\lambda_G(y') = Y_{k',\text{ports},\sigma}$ *such that after removing* y'*, the remainder of* G *contains a connected component that does not include a pin of* G*. The number of minimal cycles is finite because the search process is independent of* k*,* k' *and there is a finite number of possibilities for* ports, σ*. Consequently, there is an overall bound* $b_{\mathcal{R}}(P)$ *for a non-proliferating grammar* $(\mathcal{R}, P)$*, for any start graph* P*. If* $\mathcal{R}$ *proves to be non-proliferating, then the removal the empty nonterminals is thus computable and yields a finite grammar whose terminals are pairs* (x, y) *where* x *is the number of a connected component and* y *is an edge label of* $\mathcal{R}$*.*

Finally apply the Parikh theorem for hyperedge replacement: Lemma 4.2, Chapter VI of [Hab92] to $(\tilde{Y}, \mathfrak{CCC}(P))$ *to obtain the desired formula.*

[17]Terminals are edges, by assumption

[18]The name stands for "connected component colouring".

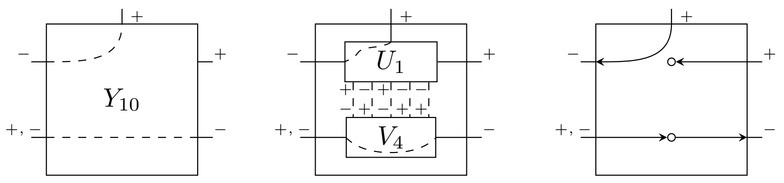


Figure 3.11.: A nonterminal $Y_{10} \in \tilde{Y}$ and the right hand sides of two of its productions.

The idea is that the new nonterminal Y_k corresponds to a case distinction to treat the different possibilities of finding a derivation starting from Y that produces exactly k new connected components not containing any pins of the original Y hyperedge. New components can come into existence in three ways: either both of their end points are within U, or both are within V, or one is within U and the other within V. These correspond to different assignments of the numbers l and m for the former cases, and ports of U and V linked via the right hand side of the production for the latter cases. These ports are then singletons in σ_U and σ_V. An example of such productions is shown in Figure 3.11. The nonterminals Y, U and V have arities five, eight and seven, respectively. Dashed lines at ports depict σ. Dashed lines between U_1 and V_4 denote identification of pins. There would be many more right hand sides, for different choices of σ.

Let $(c, \mathcal{R})$ be a HR$^-$-condition. To each nonterminal $Y \in N_{\mathcal{R}}$, assign the Presburger formulae $\theta_{Y,k,\mathrm{ports},\sigma}(\#1, ..., \#(|\sigma| + k))$ of $|\sigma| + k$ free variables that constrain the open variable n to be in the set of possible lengths of strings generated by that grammar (a construction for computing such a formula is provided by the result on semilinearity of hyperedge replacement graph languages in [Hab92]).

Note that $\mathcal{R}$ may still generate disconnected graphs. In an injective match of a disconnected graph G in a graph H, the connected components of G cannot overlap at all. If G and H are graphs meeting the restriction on node degrees and furthermore H is connected (i.e. H is a string graph), then the length of H must be greater than the sum of the lengths of the connected components of G. To get the length constraint that the presence of a subgraph isomorphic to a graph derived by the grammar $(\mathcal{R}, P)$ imposes in the satisfaction of a HR$^-$-condition $\exists(C \hookrightarrow P, c)$ or $\exists(P \sqsupseteq C, c)$, the formula constraining the integer variable that holds the length of the subgraph must be defined so as to account for the number of gaps created. In a postprocessing of the formula obtained from the Parikh theorem, this length must be the sum of the numbers of edges in the connected components, augmented by the number of connected components, decreased by one. The length of the string graph which is the domain of the satisfying morphism is then constrained from below by the sum of the length of the subgraph and the number of gaps. In $\exists(P \sqsupseteq C, c)$, P^σ and C^τ have a finite number of connected components. The semantics demands that each connected component of C^τ is a subgraph of some connected component of P^σ, but does not constrain this relation further. For each set of connected components of C^τ that lies within one of P^σ, a similar constraint as described above, concerning the lengths and gaps, arises. The proof is sketched here:

Construction 5 (**HR$^-$ Presburger Formulae**). *From a HR$^-$-condition $(c, \mathcal{R})$, we recursively define a Presburger formula and prove that the resulting formula expresses precisely the lengths of string graphs that satisfy c (a relation which we denote $\rightsquigarrow$). Define the formula $\phi_c(n, n_G)$, where n, n_G are open variables (n stands for the size P^σ when c is over P, n_G for the size of the whole graph), for all conditions c and prove that $c \rightsquigarrow \phi_c(n, n_G)$, i.e. $n \in \mathbb{Z}$ makes ϕ_c true if and only if there is a string graph S of length n and a substitution such that $S \models_\sigma n$.*

The next case is $\exists(a, c)$ with $a : P \hookrightarrow C$, best handled via an auxiliary construction: for a condition c over the labelled hypergraph P, let $P \rightsquigarrow \phi(n, n_G) :\equiv \bigvee_{\text{params}} (\exists n_{y_1}, ..., n_{y_{|Y_P|-1}}$ $P \rightsquigarrow_{\text{params}} \phi(n, n_G))$ where params assigns $(k_y \leq b_\mathcal{R}(P), \text{ports}_y, \sigma_y, \text{color}_y)$ to each $y \in Y_P$, $P \rightsquigarrow_{\text{params}} \phi(n, n_G)$ is defined as $P \rightsquigarrow \phi(n, n_G)$ but for a fixed choice of replacement parameters for the grammar $(\mathcal{R}, P)$.

Let $\phi_{\exists(a,c),\text{params}}(n, n_G) := \exists n'.\phi_{c,\text{params}}(n', n_G)[s] \wedge \bigvee_{\mathfrak{I}} \Big(\bigwedge_{y \in Y_{\text{cod}}(a)} \Big(\theta_{k_y,\text{ports}_y,\sigma_y,\text{color}_y}($ $\#1, ..., \#(|\sigma| + k)) \wedge n'_y = \sum(\#i) + k - 1\Big) \wedge \bigwedge_{y \in Y_{\text{dom}}(a)} \Big(n'_y = n_y\Big)\Big) \wedge\ n' < n_G$

In the formula, the subscript $\mathfrak{I}$ abbreviates the set of combinations of $k \leq b_\mathcal{R}(Y, \text{cod}(a))$ and valid $(\text{ports}', \sigma', \text{color}')$ consistent with choices for $\text{dom}(a)$, and $k_y, \text{ports}_y, \sigma_y, \text{color}_y$ are the components of $\text{params}(y)$, the color_y are consistent with each other (pins that are shared or connected via nonterminals of P receive the same colour), the sum $\sum \#i$ ranges over the colours used in derivations based on y only. The substitution s substitutes the primed variables $\{n'_y \mid y \in Y_{\text{cod}(a)}\}$ are for the variables $\{n_y \mid y \in Y_{\text{cod}(a)}\}$ in $\phi_{c,\text{params}}(n', n_G)$.

The final case is $\exists(P \sqsupseteq C, c)$. Let

- *$\mathfrak{I}$ be the set of all possibilities to break up P^τ into connected components. There are finitely many of them because of the restriction that distinguishes HR$^-$ from HR*. More precisely, $\mathfrak{I}_\mathcal{R}(P)$ is the set of all valid (ports, σ) combinations induced by P.*
- *$\mathfrak{J}$ be the analogous set for C^τ.*
- *$\theta_{Y\,I}(\{n_i\}_{i \in I})$ be the Presburger formula obtained from Construction 4 that evaluates to true whenever the list of variables given evaluates to a possible allotment of lengths of the connected components of a graph in $\mathcal{R}(y)$.*

Then, a formula expressing the lengths of $\exists(P \sqsupseteq C, c)$ is:

$\phi_{\exists(P \sqsupseteq C,c)}(n, n_G) := \exists n'.\phi_c(n', n_G) \wedge n' < n \wedge \bigvee_{I \in \mathfrak{I}, J \in \mathfrak{J}} (\exists \{n_i\}_{i \in I}. \exists \{n'_j\}_{j \in J}. \exists f : J \to I,$ $\forall_{i \in I}.\ n_i \geq \Big(\sum_{f(j)=i} n'_j + |\{j \in J \mid f(j) = i\}| - 1\Big)\ n = \sum_{i \in I}(n_i) \wedge n' = \sum_{j \in J}(n'_j) \wedge$ $\theta_{Y\,I}(\{n_i\}_{i \in I}) \wedge \theta_{Y\,J}(\{n_j\}_{j \in J}))$.

This latter formula is computed from $\mathcal{R}$ and P resp. C by Construction 4. Then $\phi_{\exists(P \sqsupseteq C,c),\text{params}}(n, n_G)$ is obtained by restricting $\mathfrak{I}$ to the combinations given by params *and imposing the same restriction on $\mathfrak{J}$ to ensure $P^\sigma = P^\tau$.*

By showing the soundness of Construction 5, it can be shown that HR$^-$-conditions can only express semilinear unary string languages.

Lemma 6 (**HR$^-$ Unary String Lengths**)**.** *All unary languages of string graphs expressible by HR$^-$-conditions are semilinear.*

Proof. Consider Construction 5. It must be shown that the formula it generates describes the correct sets of integers. The base case is correct, as true is satisfied by all graphs and thus imposes only the empty constraint $\phi_{\text{true}}(n, n_G) = \top$: true $\rightsquigarrow \top$. The induction step is also clear for Boolean combinations: from the definition of satisfaction we see that $\neg c \rightsquigarrow \phi_{\neg c} = \neg\phi_c(n)$. The other Boolean combinations are analogous.

For $\exists(a, c)$, assume that $c \rightsquigarrow \phi_c(n, n_G)$. Then for any string graph S, any monomorphism $p : P^\sigma \hookrightarrow S$ fulfilling $p \models_\sigma \exists(a, c)$, adds the edges from the unsubstituted hypergraph which are still present after substitution, as well as string graphs of length n_y edges from each substituted hyperedge (the length being defined in the special way that accounts for disconnected graphs, as discussed earlier and the connected components as specified by params). It is also specified that the hyperedges already present in dom(a) must be substituted in the same way in cod(a). Existence of a suitable substitution is expressed by the outer (finite) disjunction. Thus $\exists(a, c) \rightsquigarrow \phi_{\exists(a,c)}(n, n_G)$.

The final case is $\exists(P \sqsupseteq C, c)$. Again, the lengths given by the formula correspond exactly to the semantics: c must be satisfied and we rename its subgraph length variable to n'. There must be some way of allotting the connected components of C^τ to lie inside those of P^σ, which is expressed by the rest of the formula. Thus $\exists(P \sqsupseteq C, c) \rightsquigarrow \phi_{\exists(P \sqsupseteq C,c)}(n, n_G)$. □

An example for the semilinear expressive power of HR$^-$ over unary string languages:

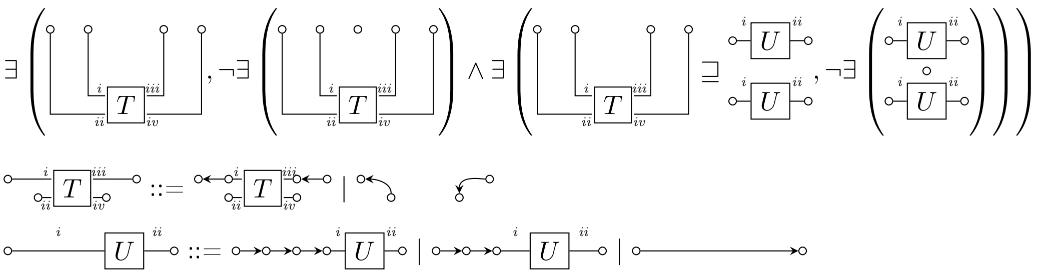


Figure 3.12.: A HR$^-$-condition.

Example 18 (Example for Lemma 6)**.** *For an illustration of the rather laborious workings of Lemma 6, consider the example of Figure 3.12. Schematically, the condition is:*

$c = \exists(\emptyset \hookrightarrow C_1, \neg\exists(C_1 \hookrightarrow C_2, \text{true}) \wedge \exists(C_1 \sqsupseteq C_3, \neg\exists(C_3 \hookrightarrow C_4, \text{true})))$.

There is only one variant for the connected components in $\mathfrak{I}$ for each of the nonterminals of $\mathfrak{R}$. Let us sketch how the formula $\phi_c(n, n_G)$ would be computed from the inside out, naming the subconditions:

$c_0 = \exists(C_3 \hookrightarrow C_4, \text{true})$

$c_1 = \exists(C_1 \sqsupseteq C_3, \neg c_0)$

First of all, notice that there is only one possible arrangement of connected components, numbering 2, for both P and C in the $\exists(C_1 \sqsupseteq, C_3, \neg c_0)$.

$\phi_{c_0}(n, n_G) = \exists n'.\,(n' = 2+n'_{y_1}+n'_{y_2}) \wedge \theta_U(n'_{y_1}) \wedge \theta_U(n'_{y_2}) \wedge n'_{y_1} = n'_{y_1} \wedge n'_{y_2} = n'_{y_2} \wedge n' < n_G$

$\phi_{c_1}(n, n_G) = \exists n'.\,\neg\phi_{c_0}(n', n_G) \wedge n' < n \wedge (\exists n_1, n_2, n'_1, n'_2, n = n_1 + n_2 \wedge\ n' = n'_1 + n'_2 \wedge$ $n_1 = n_2 \wedge n'_1 = n'_2 \wedge\ ((n'_1 < n_1 \wedge n'_2 < n_2) \vee (n'_1 < n_2 \wedge n'_2 < n_1)))$ *(because there is only one possible arrangement of connected components for both P and C, we have simply numbered the edge colours resulting for this one possibility). The full formula is obtained by continuing to gather constraints in this manner. Finally, the computation results in a formula equivalent to $\phi_c(n, n_G) \equiv n = n_g \wedge \exists x, y \in \mathbb{N}, 2 \cdot (3x + 2y) + 1$.*

The comparison to HR$^-$-condition is achieved:

Corollary 5. *There is a property expressible as a μ-condition but **not** as an HR$^-$-condition.*

Proof. From Corollary 1 and Lemma 6. □

Corollary 6. *There is a property expressible as an HR$^-$-condition but **not** as a μ-condition.*

Proof. The existence of a Hamiltonian cycle is not expressible by μ-conditions, but HR$^-$-conditions can express it. We recall the condition (Figure 3.13, replacement system as in Figure 3.9): □

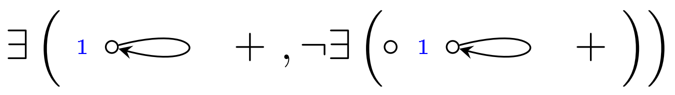


Figure 3.13.: Hamiltonicity as a HR$^-$-condition.

This also yields one direction of the comparison of μ-conditions with HR*-conditions:

Fact 8 ($\mu \subsetneq$ **HR***)**.** *There is a property expressible as an HR*-condition but not as a μ-condition.*

Proof. By Corollary 6, HR$^- \subsetneq \mu$. By definition, HR$^- \subseteq$ HR*. □

We conclude that a comparison has been achieved:

Corollary 7 ($\mu \lessgtr$ **HR**$^-$)**.** *HR$^-$-conditions and μ-conditions are incomparable.*

Proof. By Corollary 5 and Corollary 6. □

Note that we did not restrict the generation of labelled graphs, and one can easily express the intersection of two context-free languages by using a conjunction and two context-free grammars encoded as hyperedge replacement systems. However, this does not contradict our result since it is of no use in expressing unary string languages. One might also think that series-parallel graphs allow the same trick to be encoded in HR*-conditions. However, this would work only for an $\mathcal{A}$-semantics (defined in analogy to the $\mathcal{A}$-semantics for μ-conditions by removing the restriction that quantified morphisms and matches must be in $\mathcal{M}$), whereas HR$^-$ (as well as the standard definition of HR*-conditions) restricts matches and quantification to $\mathcal{M}$.

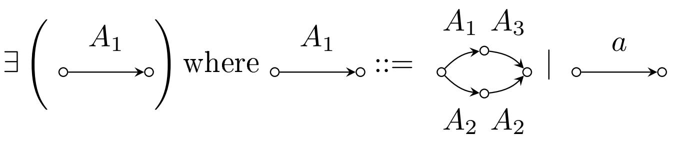


Figure 3.14.: Productions of a hyperedge replacement system for certain series-parallel graphs which, in $\mathcal{A}$-semantics, encodes a CCFG.

Example 19 (Series-parallel graphs in $\mathcal{A}$-semantics)**.** *HR$^-$-conditions in $\mathcal{A}$-semantics can, by stating the non-injective existence of a series-parallel like graph between two nodes, simulate a conjunctive context-free grammar (we have only represented one of the equations in Figure 3.14, as the others are analogous).*

Since the same is not possible with $\mathcal{M}$-semantics by Lemma 6, this construction separates $\mathcal{A}$- and $\mathcal{M}$-HR$^-$conditions.

Corollary 8 (**For HR$^-$**, $\mathcal{A} \neq \mathcal{M}$)**.** *For HR$^-$-conditions, the expressiveness of $\mathcal{A}$-semantics and $\mathcal{M}$-semantics differs, with $\mathcal{A}$-semantics being able to express non-semilinear unary string languages.*

Should μ-conditions also turn out to be incomparable to $\mathcal{A}$-HR$^-$-conditions, then it cannot be for the same reason. However, it is *also* not known whether ($\mathcal{M}$-)HR*-conditions can express non-semilinear unary languages. Perhaps the method for HR$^-$-conditions (Lemma 6) could be extended. The following conjecture is open for proof or disproof:

Conjecture 1 ($\mu \lessgtr$ **HR***)**.** *HR*-conditions and μ-conditions are incomparable.*

3.5. Conclusion and Outlook

We argue that μ-conditions are worth investigating despite the availability of other strong contenders, because our approach to non-local properties provides a new and different generalisation of nested conditions.

In analogy to the equivalence between first-order predicate graph logic and nested graph conditions, we can confirm that μ-conditions have the same expressiveness as fixed point extensions to classical first-order logic for finite graphs. Also, it seemed likely because of the starkly different definitions that the expressiveness of HR* conditions [Rad13] or even MSO differs from that of μ-conditions. The precise relationship remained to be examined; we have provided some answers now. Figure 3.15 shows the relationship between μ- and M-conditions, which is not an inclusion either way. The situation for HR$^-$ vs. μ looks the same, whereas for HR*, it only follows that μ-conditions are **not** as least as expressive as HR* conditions because that much is true for HR$^-$ already (Figure 3.16).

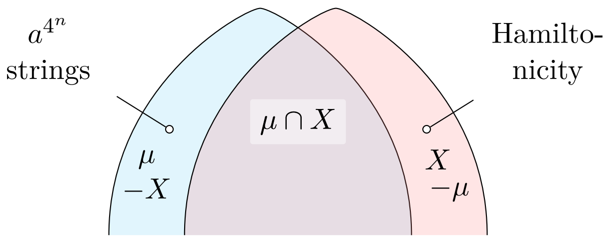


Figure 3.15.: Comparison of expressiveness between μ- and X, for μ-conditions and X-conditions with X in $\{\mathrm{HR}^-, \mathrm{M}\}$. The separating examples are the same in both cases.

Figure 3.16[19] also depicts those comparisons of graph conditions to graph logics that have been established. From Fact 7 it follows that there are some CMSO properties not expressible in HR$^-$, to wit those of discrete graphs.

$\mathcal{A}$-satisfaction is more convenient to relate conditions to logic. $\mathcal{M}$-satisfaction is more convenient in the Wlp calculus [Pen09], a property that carries over gracefully from nested conditions to μ-Conditions. Both express the same properties.

It is still unknown whether any of the the results of Subsection 3.4.4 hold for the full HR* formalism.

Conjecture 2. *HR* conditions (in $\mathcal{M}$-semantics) are not as least as expressive as μ-conditions.*

In the main part of the thesis (Chapter 5), the notion of partial correctness with respect to nested graph conditions will be recalled, core constructions known from nested conditions will be lifted to the new, more expressive conditions and a proof calculus for partial correctness relative to recursively nested conditions will be presented.

3.6. Bibliographic Notes

μ-Conditions were named after the modal μ-calculus [BS06], which is a temporal logic that also uses least (as well as greatest) fixed points. In the μ-calculus, they are used to define eventuality modalities based on a next-step operator ($\diamond\phi = \mu x.\phi \wedge Xx$. In this

[19]Nested [HPR06, HP09], HR* [Rad13], M-conditions [PP14].

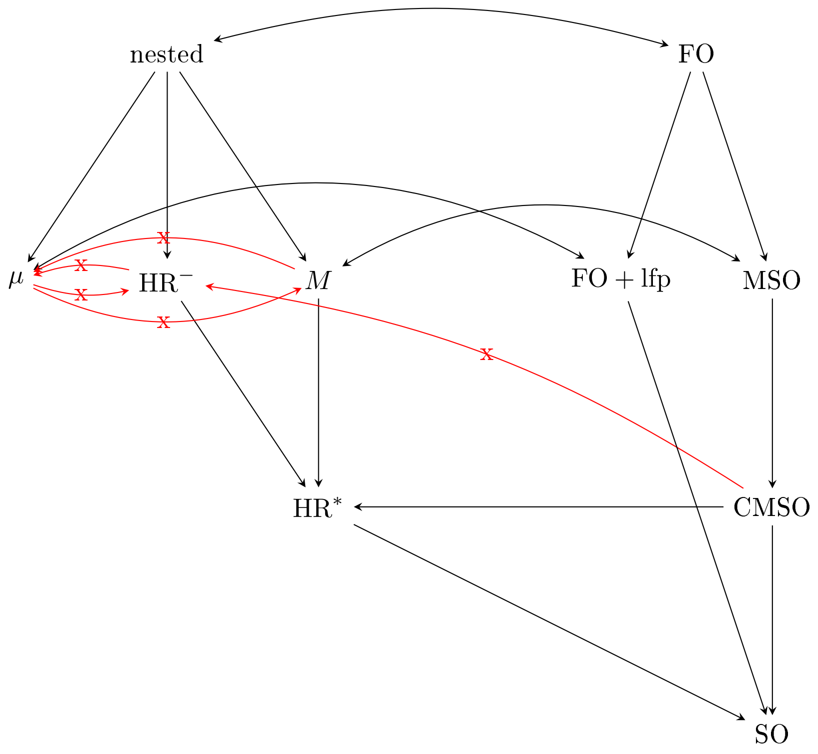


Figure 3.16.: Comparison of expressiveness. A black arrow denotes proper containment of the class of graph properties that can be expressed. A red, dotted arrow with an "x" denotes non-containment.

formula the μ stands for the least fixed point of the monotonic operator $x \mapsto \phi \wedge Xx$ and the formula means that the property ϕ will hold at some definite point in the future, but after an unspecified number of steps). A close relationship otherwise is not implied.

Our extension of nested conditions with unselection is somewhat related to lax conditions [RAB+15]. These look similar to nested conditions and are also built with quantification and Boolean combinations and can be translated to equivalent nested conditions. However, lax conditions are defined on typed attributed graphs and do not contain morphisms at all. Graphs that occur in lax conditions are only related by labellings of nodes and edges, so that like-named nodes in the condition must refer to the same node in the matched graph. The commonality is that labels that occur in the outer nesting level do not necessarily occur in the subcondition, which has an effect not unlike our unselections.

Concerning expressive power, the limits of the ability of first-order logic to express certain properties of finite structures have been of interest for a long time, and the theory is well-established by now [Lib04]. A general notion of locality for logics is due to Gaifman [Gai82]. Hamiltonicity is not definable in FO+lfp in the absence of a predefined linear order. This can be proved using so-called *Pebble games*, which incidentally are an infinite version of the Ehrenfeucht-Fraïssé games that serve to prove that first-order logic cannot express non-local conditions. We do not go into detail since the available literature provides a good exposition, for instance Libkin [Lib04].

For the axiomatization of graphs and results on monadic second-order logic on graphs, cf. the work of Courcelle [Cou97]. For fixed point logics, we used Libkin's textbook [Lib04] and Kreutzer's PhD thesis [Kre02]. String languages are more commonly studied than graph languages. There is also a tradition of examining the expressiveness of logics on strings. For example, it is a well known and celebrated fact that (already existential) monadic second order logic defines exactly the regular string languages [Tho97].

Among related work for graph conditions we feel the need to mention that Cardelli et al. [CGG02] devised what they called a spatial logic for querying graphs (in graph databases), or *graph logic*. Graph conditions have a completely different background. Advantages of graph conditions include the easy availability of weakest precondition constructions and the generic applicability to many kinds of structures, as long as they fulfil the axioms of adhesive categories [LS04] or later refinements [EEPT06].

It is not mentioned in [Rad13] or later work, and seems to be unknown, whether HR^* with $\mathcal{M}$-semantics is less powerful than with $\mathcal{A}$-semantics. If the result Lemma 6 could be lifted to HR^* rather than HR^-, then this would be the case. Nothing but second-order logic is known to encompass the whole of HR^* conditions. The latter inclusion has not been proven to be proper, although it is conjectured by Radke that HR^* is likely not able to express the existence of a nontrivial automorphism. We are inclined to support that conjecture, despite being unable to substantiate it so far.

4. Analysis of Petri Nets with Structure Changes

Self-induced petrification has its drawbacks, though.
— Jasper Fforde, *The Song of the Quarkbeast*

Contents

This chapter is oriented at users of Petri nets to introduce reconfigurations via graph transformations. Its purpose is to provide a contrast to the proof-based methods developed later and show why fully automatic solutions can be expected to work only in restricted special cases. As indicated in the introduction, this chapter is not a prerequisite for the later ones.

In Section 2.3, we recalled that Petri nets are special graph transformation systems, which in turn (Section 2.2) are special graph programs: they can either be viewed directly as graph transformation systems with discrete labelled graphs, or the net structure and marking can be encoded in graph transformation rules. Any definition of structure-changing Petri nets would be situated, from a modelling point of view, midway between Petri nets (which are well suited to modelling resource production and consumption in statically structured processes) and graph transformation systems (which can express structure changes in great generality).

When structure change (or *reconfiguration*) rules are restricted, the resulting formalisms becomes amenable to more analyses than general graph transformation systems and can still express aspects of dynamic structure. On the flip side, the content of this chapter is only loosely connected to the rest of the thesis because the positive results do not generalise to graph programs. The definitions in this chapter are not literally compatible with those of Chapter 2, special-purpose definitions are used to allow a more compact presentation of the results. However, bear in mind that an encoding of structure-changing Petri nets as graph programs is not difficult and the verification methods to be introduced in later chapters also work for structure-changing Petri nets.

The results in this chapter are decision procedures for

- the firing word problem for 1-safe structure-changing workflow nets
- reachability in structure-changing Petri nets
- abstract reachability in 1-safe structure-changing workflow nets

and undecidability of the containment of a structure-changing workflow net (labelled firing sequence) language in a regular language.

The chapter is structured as follows: in Section 4.1 we state our decidability and undecidability results. Section 4.2 concludes the presentation with an outlook and Section 4.3 draws parallels to existing work.

4.1. Analysis of Structure-Changing Workflow Nets

In this section, we define and investigate some decision problems for structure-changing workflow nets. We define the derivation language of a structure-changing Petri net and show that while language containment in a regular language is undecidable, two reachability problems are decidable, and the firing word problem is decidable at least for structure-changing workflow nets. We prove a series of lemmata, corresponding to a local Church-Rosser property for graph grammars [EEPT06], that allow us to equivalently re-arrange derivations, which will be used in the proofs of the first and the third result[1].

Lemma 7 (Independence). *Given a structure-changing Petri net $(\mathcal{N}, \mathcal{R})$, a firing step $(N, M) \overset{t}{\Rightarrow} (N, M'')$ and a replacement step $(N, M) \overset{\varrho,m}{\Rightarrow} (N', M')$ for some $(\varrho, N_l, N_r) \in \mathcal{R}$ and match m with $t \neq m(t_l)$, then there is a firing step $(N', M') \overset{t}{\Rightarrow} (N', M''')$ and a replacement step $(N, M'') \overset{\varrho}{\Rightarrow} (N', M''')$ where $M'''(p)$ takes the value $M''(p)$ for $p \in P$ and 0 for $p \notin P$.*

$$\begin{tikzcd}
 & (N,M) \arrow[dl, Rightarrow, "\varrho"'] \arrow[dr, Rightarrow, "t"] & \\
(N',M') \arrow[dr, Rightarrow, "t"'] & & (N,M'') \arrow[dl, Rightarrow, "\varrho"] \\
 & (N',M''') &
\end{tikzcd}$$

Proof. Immediate from Definition 2.3 and the definition of transition firing. □

Example 20. *Figure 4.1 shows a firing step (horizontal) and a replacement step (vertical), which are independent in the sense of Lemma 7.*

The result (N', M''') is the same only up to an isomorphism, but this will not really impact any of the results. In the scope of this section, two derivations (ξ, σ) and (ξ', σ') are *equivalent* if they are of the same length n and the marked nets σ_n and σ'_n are

[1]One could certainly re-use the result from graph grammars, but that would require us to first formally redefine structure-changing Petri nets as graph transformation systems.

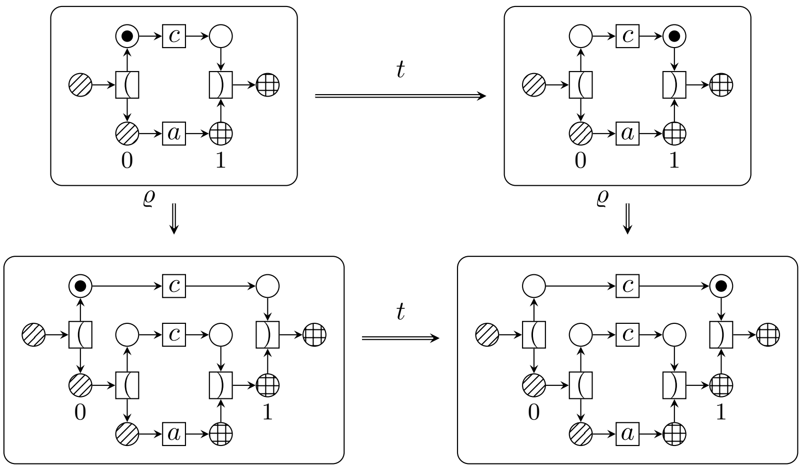


Figure 4.1.: Independent steps.

isomorphic. Two transition sequences are said to be equivalent if they are equivalent as derivations. Transition sequences which only differ by a permutation of the steps are easily seen to be equivalent. In a sound workflow net N, a transition sequence u is said to be *terminal* when $({}^\bullet N)u = N^\bullet$. We provide the relevant lemma:

Lemma 8 (**Permuting Steps**). *If $\mathcal{S}$ is a structure-changing Petri net, for every derivation (ξ, σ) of length n in $\mathcal{S}$, under any of the conditions given below there is a derivation (ξ_π, σ_π) also of length n such that σ_n and $(\sigma_\pi)_n$ are isomorphic, and $\lambda(\xi) = uabv$ and $\lambda(\xi_\pi) = ubav$ for some $u, v \in (R \cup \Sigma)^*$ and $a, b \in (R \cup \Sigma)$. If $\lambda(\xi_i) = a$ and $\lambda(\xi_{i+1}) = b$, $t_a = \tau(\xi_i)$, $t_b = \tau(\xi_{i+1})$,* $\mathrm{from}(\xi_i) = \mathcal{N}$, $\mathrm{to}(\xi_i) = \mathrm{from}(\xi_{i+1}) = \mathcal{N}'$ *and* $\mathrm{to}(\xi_{i+1}) = \mathcal{N}''$*:*

Situation	*Sufficient condition for transposition*
$a \in \Sigma, b \in \Sigma$	${}^\bullet t_b \cap t_a{}^\bullet = \emptyset$
$a \in \Sigma, b \in R$	$t_a \notin T - T'$
$a \in R, b \in \Sigma$	$t_b \notin T' - T$
$a \in R, b \in R$	$t_b \notin T' - T$

Proof. $a, b \in \Sigma$: transitions meeting the requirement can be transposed in an enabled sequence: in a net N, if Mt_1t_2 exists and ${}^\bullet t_2 \cap t_1{}^\bullet = \emptyset$, then t_2 is enabled in M: all places in ${}^\bullet t_2$ hold at least as many tokens as in Mt_1, and t_1 is enabled in Mt_2 because $\forall p \in {}^\bullet t_1 \cap {}^\bullet t_2$ have $M(p) \geq F^-(p, t_1) + F^-(p, t_2)$. By commutativity of addition $Mt_1t_2 = Mt_2t_1$ and hence $\sigma_{|u|+2} = (\sigma_\pi)_{|u|+2}$.

$a \in \Sigma, b \in R$: b can be applied to $\sigma_{|u|}$ because rule applicability is independent of the marking, and by Lemma 7 the same state $\sigma_{|u|+2}$ is reached. The condition is necessary to meet the requirements of Lemma 7: otherwise, t_a is not available after the replacement step.

$a \in R, b \in \Sigma$ is similar: since the preset of the transition t_b is unchanged by the application of R, t_b is enabled in the state $\sigma_{|u|}$ and Lemma 7 applies.

$a, b \in R$: since only the matched transition is replaced and all others remain unchanged, the rules can be swapped, yielding an isomorphic result. In this case and the previous one, the condition is necessary because otherwise the transition t_b is not present prior to the event σ_i. □

The homomorphism $t \mapsto$ (if $t \in A$ then 0 else 1) defines a pre-order $\precapprox_{\boldsymbol{A},\boldsymbol{B}}$ on sequences of $(A+B)^*$ by comparing the images of sequences under the lexicographic order induced by $0 < 1$ on $\{0,1\}^*$. This can be used to re-order a sequence to a normal form:

Lemma 9 (Ordering transitions sequences). *Given a transition sequence u in a marked net with $T = T' + T''$, and $T' \times T''$ contains only pairs (t', t'') where ${}^\bullet t'' \cap t'^\bullet = \emptyset$, then any transition sequence that is equivalent to u and minimal with respect to $\precapprox_{T',T''}$ is in $T'^* T''^*$.*

Proof. A strictly decreasing step along the lexicographic order is possible as long as Lemma 8 can still be applied to a contiguous subsequence $t''t'$, yielding another equivalent transition sequence. □

Definition 29 (Causal Dependency). *A step e_i of a derivation (ξ, σ)* causally precedes *another step e_j, $i < j$, if either $\lambda(\xi_i), \lambda(\xi_j) \in \Sigma$ and $\tau(\xi_i)^\bullet \cap {}^\bullet\tau(\xi_j) \neq \emptyset$, or $\lambda(\xi_i) \in R, \lambda(\xi_j) \in \Sigma$ and $\tau(\xi)_j \in \mathrm{codomain}(m_i)$ or $\lambda(\xi_i), \lambda(\xi_j) \in R$ and $\tau(\xi)_j \in \mathrm{from}(e_i)$. The* causal dependency *relation is the transitive closure of the causal precedence relation.*

Lemma 10 (Ordering and causal dependency). *Any derivation (ξ, σ) equivalent to a given one and minimal with respect to $\precapprox_{\Sigma,R}$ has $\lambda(\xi) = r_0 s_0 ... s_n r_{n+1}$ where $\forall i, s_i \in \Sigma, r_i \in R^*$ and if $\xi_{i,0} ... \xi_{i,n} = r_i$, then $\xi_{i,j}$ causally precedes $\xi_{i,j+1}$, and $\xi_{i,n}$ causally precedes s_{i+1}.*

Proof. Analogous to Lemma 9. As long as there is a contiguous subsequence $\xi_i \zeta \xi_j$ where $\lambda(\xi_i) \in R$, $\lambda(\zeta) \in R^*$, $\lambda(\xi_{i+1}) \in \Sigma$ and ξ_i does not causally precede any of $\zeta \xi_j$, by induction over the length of ζ and using Lemma 7, an equivalent but strictly $\precapprox_{\Sigma,R}$-smaller derivation with $...\zeta \xi_j \xi_i...$ can be constructed. □

4.1.1. Firing Word Problem

A language-theoretic approach to Petri nets looks at their labelled firing sequences. To each transition a letter is assigned and the Petri net is treated as a generator of possible outputs (usually with a termination criterion), cf. Chapter 6 in [PW02].

Definition 30 (Firing language). *Let $\mathfrak{F}$ be the homomorphism that deletes all letters of R and leaves transition labels Σ unchanged. The* firing language *of a structure-changing Petri net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$ is*

$$\mathfrak{F}(\mathcal{S}) := \{\mathfrak{F}(w) \in \Sigma^* \mid \exists \mathcal{N}' : \mathcal{N} \overset{w}{\Rightarrow}_{\mathcal{R}} \mathcal{N}'\}$$

Due to the deleting homomorphism, it is not trivial to see whether a given word w occurs in the firing language of $\mathcal{S}$.

Problem 1 (Firing word problem).

Given:	*a structure-changing Petri net $\mathcal{S}$ and a word $w \in \Sigma^*$*
Question:	*$w \in \mathfrak{F}(\mathcal{S})$? Is w contained in the firing language of $\mathcal{S}$?*

We examine the firing word problem for 1-safe structure-changing workflow nets.

Problem 2 (Problem 1 for 1-safe structure-changing workflow nets).

Given:	*a 1-safe structure-changing workflow net $\mathcal{S}$ and a word $w \in \Sigma^*$*
Question:	*$w \in \mathfrak{F}(\mathcal{S})$? Is w contained in the firing language of $\mathcal{S}$?*

The question is decidable for 1-safe structure-changing workflow nets by the following means: while applying rules, one keeps track of the minimum length of any word in which each transition could occur (the *minimum word length* defined below). Rule matches on transitions that can only be used in words longer than w need not be explored, because under the postulated assumptions the minimum word length is non-decreasing under replacement. Rules are only applied to enabled transitions. The creation of redundant transitions is limited by condition (3) of the definition of a structure-changing workflow net.

In a static net $\mathcal{N} = (N, M)$, let $\| \cdot \|_{\mathcal{N}} : T \rightharpoonup \mathbb{N}$ be the partial function assigning to each transition t of the net the minimum length of any derivation using t. It is undefined if there is no such derivation, otherwise $\|t\|_{\mathcal{N}} = min(\{|ut| \mid ut \in T^*, \exists M' = Mut\})$. We call ut with $|ut| = \|t\|_{\mathcal{N}}$ a *minimum-length t-word* of $\mathcal{N}$. It contains t exactly once at the end, or a proper prefix would suffice. Note also that sequences in T^* rather than label words in Σ^*, are considered. Calculating $\|t\|_{\mathcal{N}}$ might in general not be efficient, but it is in principle possible for all nets. The following lemma relates the minimum lengths of derivations using transitions of the replaced net to those of the original net and the right hand side.

Lemma 11 (Minimum length of derivation using a transition). *Let $\mathcal{S} = (\mathcal{N}, \mathcal{R})$ be a 1-safe structure-changing workflow net. Suppose $\mathcal{N} \overset{*}{\Rightarrow} \mathcal{N}'$, and $\mathcal{N}' \overset{\varrho,m}{\Rightarrow} \mathcal{N}''$ is a replacement step using rule (ϱ, N_l, N_r) (let $t = m(t_l)$). Then each transition $\underline{t} \in T'' - T'$ has $\|\underline{t}\|_{\mathcal{N}''} = \|t\|_{\mathcal{N}'} - 1 + \|\underline{t}\|_{(N_r, {}^\bullet N_r)} \geq \|t\|_{\mathcal{N}'}$, and each transition $\check{t} \in T - \{t\}$ has $\|\check{t}\|_{\mathcal{N}''} \geq \|\check{t}\|_{\mathcal{N}'}$.*

Proof. Let ut be a minimum-length t-word in $\mathcal{N}$ and $v\underline{t}$ a minimum-length $\underline{t}$-word in $(N_r, {}^\bullet N_r)$. Then $uv\underline{t}$ is a minimum-length $\underline{t}$-word in $\mathcal{N}'$.

A transition sequence $v \in T_r^*$ is enabled in some marking in N_r precisely when it is in the corresponding marking (mapping the places via m) in N'' and has the same effect on the places of $P_r - P_l + m(P_l)$ and no effect on all other places. If u is a transition sequence enabled in M', then Lemma 9 guarantees the existence of a transition sequence $\tilde{u}$ that can be decomposed as $\tilde{u} = uv$, where $u \in T'^*$ and $v \in (T'' - T')^*$. For any such uv, because Mu exists and enables t, ut is also a possible transition sequence, and furthermore the shortest one containing t.

As to the second statement, let $u\check{t}$ be a minimum-length $\check{t}$-word of $\mathcal{N}''$. Decomposing u as $u_0 t_0 u_1 ... t_n u_{n+1}$ with $u_i \in T'^*$ and $t_i \in T'' - T'$, there is an equivalent $\precapprox_{T''-T',T'}$-minimal enabled transition sequence $\tilde{u} = \tilde{u}'_0 s_0 \tilde{u}'_1 s_1 ... \tilde{u}'_{m+1}$ with $\tilde{u}'_i \in T'^*$, $s_k \in (T'' - T')^*$ and $s_0 \cdot ... \cdot s_m = t_0 \cdot ... \cdot t_n$, and $\tilde{u}'_0 \cdot ... \cdot \tilde{u}'_{m+1} = u_0 \cdot ... \cdot u_{n+1}$, and s_k is terminal in N_r: assuming the contrary would imply that any equivalent transition sequence of the form $\tilde{u}'_0 s_0 \tilde{u}'_1 s_1 ... \tilde{u}'_m$ has some s_k such that the condition $({}^\bullet N_r)s_k = N_r{}^\bullet$ is violated. Every transition t_i added in the replacement step falls into one of three sets, which we call *start*, *middle* and *end*. A *start* transition is one enabled in ${}^\bullet N_r$. A *end* transition is one that has the end place of N_r in its postset. A *middle* transition is any other transition of N_r. So if $\tilde{u}'_0 s_0 \tilde{u}'_1 s_1 ... \tilde{u}'_m$ is such a sequence where for some $k \in \{0.....m\}$, $({}^\bullet N_r)s_k$ differs from $N_r{}^\bullet$ then the first transition of s_{k+1} (if $k < m - 1$) can neither be a start, nor middle, nor end transition. The latter two lead to a contradiction with $\precapprox_{T''-T',T'}$-minimality, while the former would contradict 1-safety.

Therefore one can construct a $\check{t}$-word of $\mathcal{N}'$, namely $\tilde{\tilde{u}}'' = \tilde{u}'_0 t \tilde{u}'_1 ... \tilde{u}'_m \check{t}$, which is in any case at most as long as $u\check{t}$, thus $\|\check{t}\|_{\mathcal{N}'} \leq \|\check{t}\|_{\mathcal{N}''}$ □

The firing word problem is decidable for 1-safe structure-changing workflow nets in spite of structure changes. The idea of the proof was anticipated in the text following the problem description:

Proposition 4 (**Decidability of Problem 2**). *It is decidable for any 1-safe structure-changing workflow net $\mathcal{S}$ and word $w \in \Sigma^*$ whether $w \in \mathfrak{F}(\mathcal{S})$.*

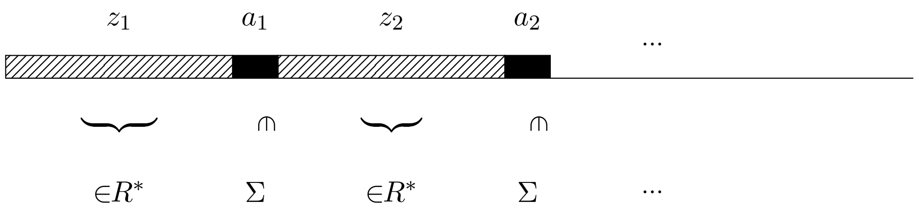
Proof. The algorithm in Figure 4.2 decides the question. j is a natural number, and a pair of marked nets is in the relation $\cong_j$ if the marked nets have the same subnet induced by considering only the transitions that can contribute to words of length up to j, and the places attached to these transitions.

We prove that the algorithm terminates and outputs the correct answer.

The procedure WORD iterates over w. It is correct, by induction on the length of w: the induction basis is witnessed by the trivial case of the empty word. Induction hypothesis: for any $j \leq |w| \in \mathbb{N}$, the prefix u of w of length j is in the language iff St is not empty. Induction step: for any derivation (ξ, σ) with $\mathfrak{F}(\lambda(\xi)) = au$, Lemma 10 yields another derivation (ξ', σ'), with $\lambda(\xi') = zau'$, $z \in R^*$ and $\mathfrak{F}(u') = u$, and each step e_i $(i \leq |z|)$ causally preceded by the previous.

```
procedure Word($\mathcal{N}, w$)
    $St := \{\mathcal{N}\}$
    while $w \neq \epsilon$ do
        $St :=$ Explore($St, |w|$)
        $aw := w$                          ; $a$ was the first letter, $w$ now contains the rest
        $St := \{(N, Ma) \mid (N, M) \in St\}$        ; try successor by firing, if defined
    return $\{(N, M) \in St \mid M = p_{out}\} \neq \emptyset$
procedure Explore($St, j$)
    repeat
        for all $\mathcal{N} \in St$ do              ; gather relevant successors by replacement
            for all enabled transitions $t$ of $\mathcal{N}$ do
                for all matches $m$ of rules $\varrho$ on $t$ do
                    $\mathcal{N}' :=$ result of replacement step          ; $\mathcal{N} \overset{\varrho,m}{\Rightarrow} \mathcal{N}'$
                    if $\nexists \mathcal{N}'' \in St : \mathcal{N}' \cong_j \mathcal{N}''$ then
                        add $\mathcal{N}'$ to $St$
    until no *new* states found
    return $St$
```

Figure 4.2.: Algorithm for the firing word problem



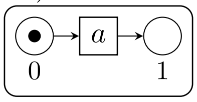
Each of the replacement steps e_i whose corresponding rule names form the z prefix causally precedes the next; unless $|z| = 0$, the first transition to be fired, labelled a, is created in the step $e_{|z|-1}$. In any of the first $|z|$ steps, the matched transition is enabled. This makes it unnecessary to ever directly explore the replacement of disabled transitions.

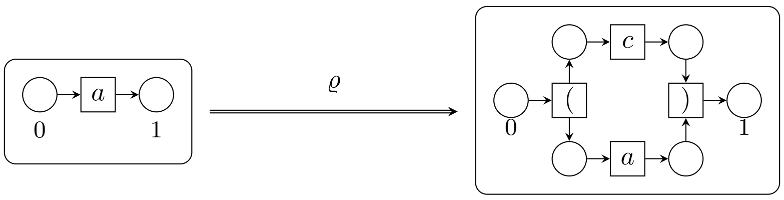
It remains to be seen that the subset of reachable states of $\mathcal{S}$ that actually need to be explored further for the word u is finite: it is always the case that $|z|$, which may well be 0, can be bounded from above by a constant depending only on the rules. The reason why the Explore procedure terminates rests with the comparison that decides whether states are added to St. The comparison is done according to the equivalence relation $\cong_j$. By Lemma 11, minimum word lengths only increase and therefore all transitions with minimum word length exceeding j can be ignored in the exploration.

Only a finite number of equivalence classes of $\cong_j$ occur in the application of replacement steps from $\mathcal{R}$. Let n be the maximum number of items in a right hand side of any rule of $\mathcal{R}$. Regarding each state as a directed graph whose nodes are the places and transitions, and whose arcs are directed edges, let d be the function assigning to each item *it* the length of the shortest directed path from the place p_{in} to *it*. Then by induction on the length of the derivation, no rule application leads to a state which, as a graph, has any node with more than n successors. The number of items with d up to $2j$ is bounded by n^{2j}. There is a finite number of 1-safe nets with these properties.

The set of items of $\mathcal{N}$ with d up to $2j$ clearly includes all transitions t with $\|t\|_{\mathcal{N}} \leq j$. Firing may reduce the minimum word length of any transition by at most 1. This makes it safe to compare states by $\cong_{j-1}$ at the next iteration. □

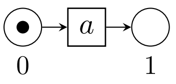
The role of the $\cong_j$ check is that since items that are too far from any marked place can never be used (a notion that was made precise in Lemma 11), nets that differ only in these items can be regarded as equivalent as regards the question at hand. The check is necessary to ensure termination, since otherwise EXPLORE could diverge even before the first firing is attempted in line 6.

Example 21 (Firing word problem)**.** *As an example, consider a structure-changing workflow net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$ with $\mathcal{N} =$*  *and $\mathcal{R} = \{\varrho\}$:*



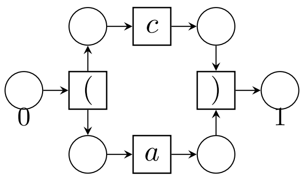
Let us use the algorithm to decide whether ((cca)) $\in \mathfrak{F}(\mathcal{S})$.

Initially, St contains only $\mathcal{N}$.

WORD *initialises its state to St =* {  }.

Since w is not yet empty, EXPLORE *is called on this set St and $|w| = 7$.*

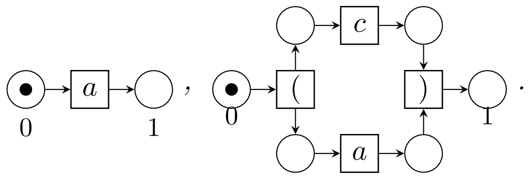
One enabled transition exists in the sole net in St, and it can be replaced once to yield a net shaped like this:



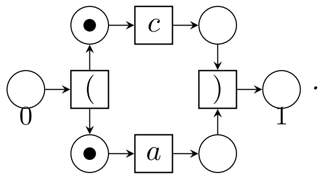
,

which is not $\cong_7$-equivalent to the net in St and therefore added to St.

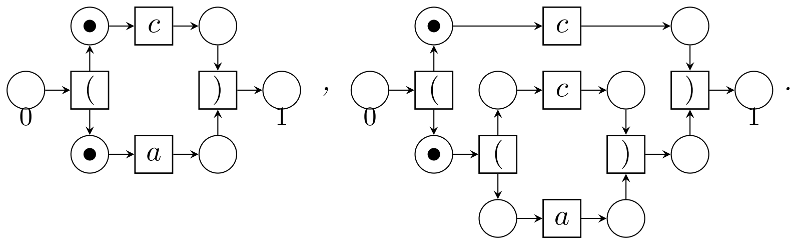
After EXPLORE*, St contains these nets:*



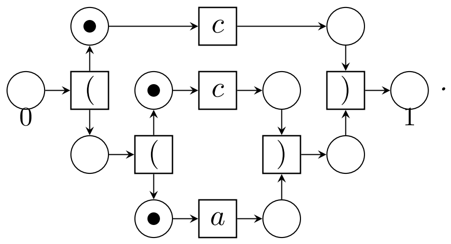
Then w is set to $(cca))$ *and the variable a to the letter* $($*. Firing of a transition labelled* $($ *is attempted, which fails for the first net and yields one successor for the other. Now St contains this net:*



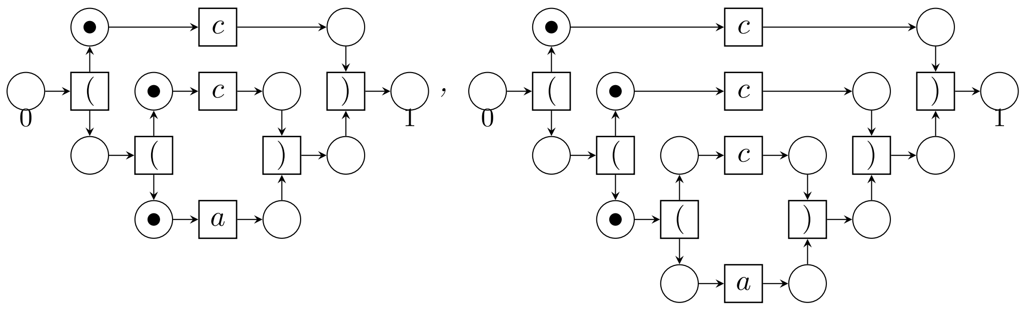
Since $(cca)) \neq \epsilon$, Explore *is called again. After* Explore, *since the new net is not* $\cong_6$*-equivalent to the one net already in St, St contains two nets:*



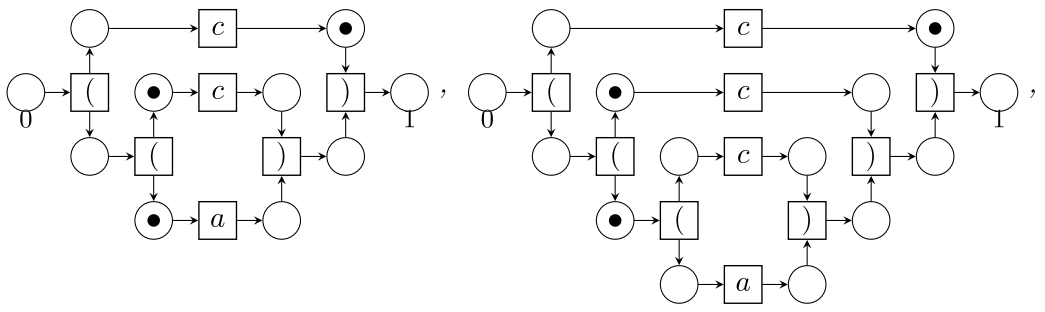
and $a = ($*. After firing of a transition labelled* $($*, St contains one net:*

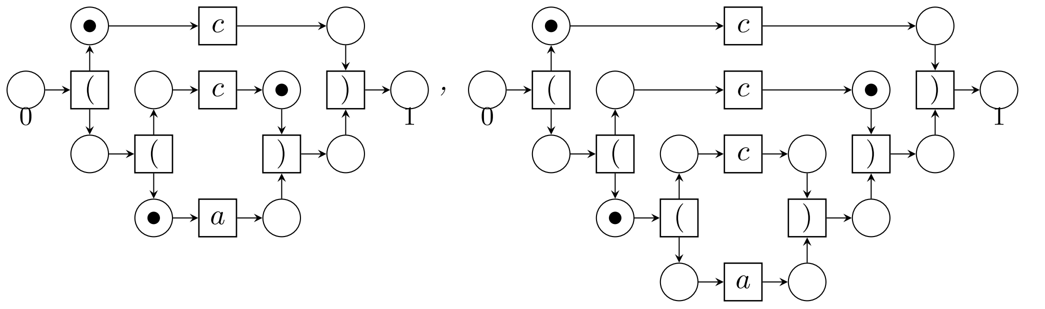


After Explore, *St is*

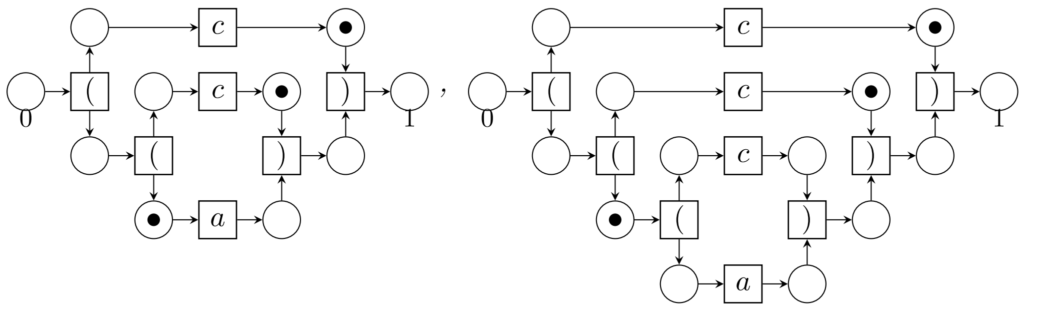


and the variable a holds the letter c*. After firing of a transition labelled* c*: St is*

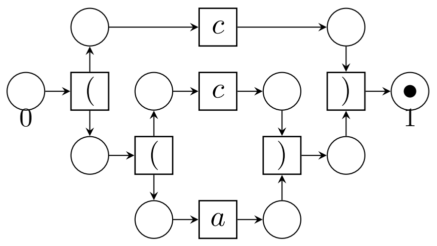




After Explore*, St is unchanged because all the resulting nets are already present. The variable a again holds the letter c. After firing of a transition labelled c, St is*

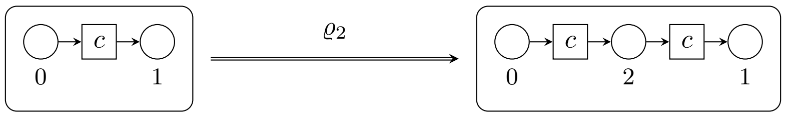


After this, Explore *still adds nothing new. A transition labelled a can be fired in one of the nets. In the last two iterations no replacement rules apply and the remaining letters)) are used for firing. In the end, St contains exactly one net marked with the end marking:*



and the algorithm outputs a positive answer. We conclude that ((cca)) $\in \mathfrak{F}(\mathcal{S})$.

The structure-changing workflow net of Example 21 is too tame to exhibit the type of behaviour that warrants the $\cong_j$ check, because when ϱ is applied to an enabled transition, it *always reduces the number of* enabled and replaceable *transitions by one, since transitions with more than one outgoing edge cannot be replaced by definition of a structure-changing workflow net. The need for the check becomes clearer when there is a second rule:*



When a transition labelled c is enabled and the remaining word to be checked is of the form cu, $u \in T^*$ (or indeed any other non-empty word), the algorithm must make sure not only to check whether the word can be obtained as a firing word by firing the transition c directly, but also whether it could be obtained by applying replacement steps until another transition labelled c comes up. Here, the $\cong_j$ check keeps the search from diverging.

Note also that a variant of the algorithm can be used to check directly for prefixes of firing words:

Fact 9. *By returning $St \neq \emptyset$ instead of $\{(N, M) \in St \mid M = p_{out}\} \neq \emptyset$, the algorithm can be made to check whether w is a* prefix *of any word of $\mathfrak{F}(\mathcal{S})$.*

Proof. The proof follows the same lines as Proposition 4. Instead of checking whether a terminal marking can be reached by a derivation with image w, an arbitrary marking is necessary and sufficient for a positive output of the algorithm. □

4.1.2. Reachability Problems

In this section, before defining abstract reachability we first show that concrete reachability (without using any abstraction) of a given marked net is decidable. Our construction for deciding concrete reachability is similar to the unfolding construction used in [BCKK04] in a different context. The idea is to unfold all possible rule applications until the size of the smallest net obtained using a new match exceeds the queried $\mathcal{N}$, since the size of the net grows monotonically with each rule application and the set of rules is finite.

In the construction, for each possible match, the net is extended the same way as in a rule application but the matched transitions are preserved. To control the simulation, the rule transitions and control places take care of disabling or enabling net transitions according to the simulated rule applications.

Let $\text{match}_\varrho(N) \subseteq T \times P^*$ be the set of all matches of rule ϱ on net N, each match being uniquely determined by target transition and place mapping (we assume here that every left hand side is equipped with an arbitrary total order on the place set, inducing a bijection between place mappings and sequences over P).

Construction 6 (**k-unfolding**). *Given a n-coloured structure-changing Petri net $(\mathcal{N}, \mathcal{R})$, let N_0 be the net obtained from N by adding new places $\{p_t \mid t \in T\}$ and arcs of weight 1 from each t to its respective p_t and back. The new colour n is chosen for all the p_t*

places. To each item created in the construction, we assign a history*, which is a sequence of steps. All transitions t of N_0 have* $\mathrm{hist}(t) = \epsilon$. *Items added in a rule application $e_{\mu,\varrho}$ matching $t \in T_i$ will have history $hist(t) \cdot e_{\mu,\varrho}$.*

For $i \leq k$, the set T_{i+1} is computed from T_i by disjointly adding, for every match μ of any rule $(\varrho, N_l, N_r) \in \mathcal{R}$, the transitions $T_r - T_l$, and one transition for each match, which we call match transition*: $\{t_{\mathrm{hist}(t)\cdot e_{\mu,\varrho}} \mid \mu \in \mathrm{match}_\varrho(N_i)\}$ where $e_{\mu,\varrho}$ is the replacement step determined by ϱ, μ and N_i.*

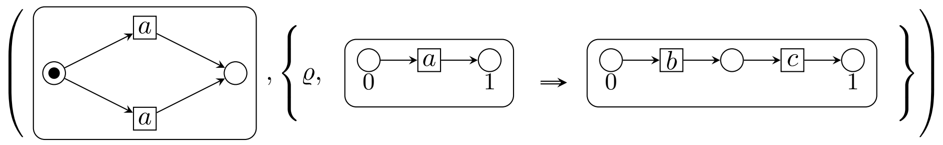
The set P_{i+1} is computed from P_i by disjointly adding, for every match μ of any rule $(\varrho, N_l, N_r) \in \mathcal{R}$, the places of $P_r - P_l$, and one control place *for every right hand side transition: $P_{\mathrm{ctrl}} = \{p_t \mid t \in T_{i+1} - T_i, l(t) \in \Sigma\}$.*

The arc weights in N_{i+1} are as follows:

$$F^{\pm}_{i+1}(t,p) = \begin{cases} F^{\pm}_i(t,p) & t \in T_i, p \in P_i \\ F^{\pm}_r(t,p) & t \in T_r, p \in (P_r - P_l)\ (*) \\ F^{\pm}_r(t,p') & t \in T_r, p = m(p')\ (*) \\ 1 & t \in T_i + T_r,\ p = p_t \\ 1 & t = t_x,\ c(p) = n,\ \mathrm{hist}(p) = xe\ \ (F^{+}\text{only}) \\ 1 & t = t_{xe},\ c(p) = n,\ \mathrm{hist}(p) = x\ \ (F^{-}\text{only}) \\ 0 & \text{otherwise,} \end{cases}$$

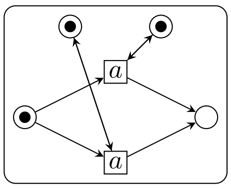
where $(*)$ applies to items obtained from the same match of a rule (ϱ, N_l, N_r), x is any sequence of steps and e is any step. The p_t places are always assigned colour n and the t_x transitions are labelled with the corresponding rule name from R. The marking M_k agrees with M on the places of N_0, has 1 on each p_t place and 0 elsewhere. $\mathcal{N}_k = (N_k, M_k)$ is the k-unfolding *of $(\mathcal{N}, \mathcal{R})$.*

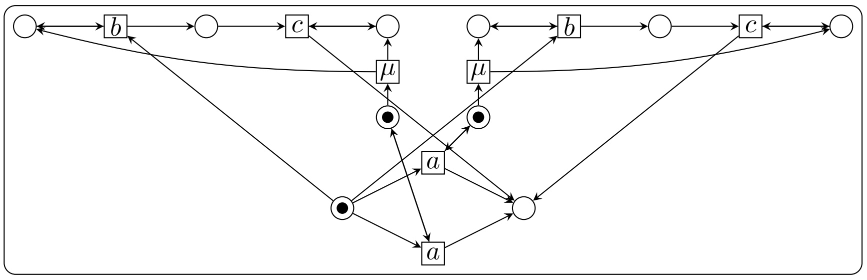
Example 22 (A structure-changing workflow net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$)**.**



Example 23 (The 0-unfolding $\mathcal{N}_0$ and the 1-unfolding $\mathcal{N}_1$ of $\mathcal{S}$[2])**.**

[2]where μ indicates match transitions





Let $\kappa' = \kappa'(\mathcal{S}, \mathcal{N}_Q) \in \mathbb{N}$ be the smallest number such that every marked net $\mathcal{N}$ reachable in any derivation (ξ, σ) without state cycles (i.e. repetition-free σ) and with $|\lambda(\xi)|_R \geq \kappa'$ has no derivation $\mathcal{N} \overset{*}{\Rightarrow}_{\mathcal{R}} (N_Q, M)$ for any M. The number κ' is well-defined because net size increases monotonically along any derivation, and only finitely many nets of any given size are reachable. For the same reason, κ' can be effectively over-approximated by considering all derivations with $|\lambda(\xi)|_\Sigma = 0$ that produce nets of size at most $|P_Q| + |T_Q|$.

Because reconfigurations do not decrease the size of a net, the reachability of a concrete marked net is decidable for any structure-changing Petri net.

Proposition 5 (**Decidability of concrete reachability**). *It is decidable whether a given marked net $\mathcal{N}_Q$ is reachable in a given structure-changing Petri net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$.*

Proof. The idea is to determine the number $\kappa'(\mathcal{S}, \mathcal{N}_Q)$ and reduce the question to reachability in a κ-unfolding $\mathcal{N}_\kappa$ of $\mathcal{S}$ for $\kappa \geq \kappa'(\mathcal{S}, \mathcal{N}_Q)$, the Petri net reachability problem being well-known to be decidable [PW02].

$$\begin{tikzcd}
(\hat{N}, \hat{M}) \arrow[d, Rightarrow, "\varrho \text{ or } t"'] & (N_\kappa, M) \arrow[l, "\text{strip}"'] \arrow[d, Rightarrow, "t_\mu \text{ or } t"] \\
(\hat{N}', \hat{M}') & (N_\kappa, M') \arrow[l, "\text{strip}"]
\end{tikzcd}$$

We prove a mutual simulation of $\mathcal{S}$ and $\mathcal{N}_\kappa$ for any derivation ξ with up to κ rule applications by induction on the length of ξ. As argued in the definition of $\kappa(\mathcal{S}, \mathcal{N}_Q)$, no derivation with $|\lambda(\xi)|_R > \kappa$ yields $\mathcal{N}_Q$. If $|\lambda(\xi)|_R \leq \kappa$, by induction on the length of the derivation, $\mathcal{N}_\kappa$ and $\mathcal{S}$ simulate each other via a mapping strip, defined as follows: the image of (N_κ, M) is the marked net $\hat{\mathcal{N}}$ with transitions $\hat{T} = \{t \in T \mid M(p_t) > 0\}$, places $\hat{P} = (P - P_{\text{ctrl}}) - \{p \in P \mid \forall t \in {}^\bullet p \cup p^\bullet, t \notin \hat{T}\}$ and $\hat{F}^-$, $\hat{F}^+$, $\hat{l}$, $\hat{c}$ are F^-, F^+, l, c with their domains restricted to $\hat{P}$, $\hat{T}$.

If $\hat{\mathcal{N}} \overset{x}{\Rightarrow}_{\mathcal{R}} \hat{\mathcal{N}}'$, if $x \in R$, and by hypothesis, $\hat{\mathcal{N}} = \text{strip}(\mathcal{N}'')$ for $\mathcal{N}'' = (N_\kappa, M)$ for some marking, then $Mt_x = M'$ such that $\text{strip}(N_\kappa, M') = \hat{\mathcal{N}}'$: there is a transition t_x because all rule matches for derivations of up to κ rule applications are represented as transitions in N_κ; the preset of t_x contains exactly the matched transition $\tau(x) = \text{strip}(\hat{t})$ for some $\hat{t} \in \hat{T}$. This transition is not in $\hat{T}'$. Instead, the items of the right hand side are present in strip(N_κ, M'), according to the replacement step x. If $x \in \Sigma$, then the step is also simulated in strip$(\mathcal{N})$.

If in N_κ, $Mt_x = M'$, then it is easy to see that the images under strip of (N_κ, M) and (N_κ, M') are related by the replacement step with match μ. If $Mt = M'$ for a non-match transition, this corresponds via strip to a firing of that transition.

The simulation faithfully preserves all events in derivations of length up to κ. $\square$

Fact 10 (**Bounded reconfigurations**). *If the number of structure changes that will occur during the runtime of the system is $k \in \mathbb{N}$, then the reachability graph of the k-unfolding of any structure-changing Petri net $\mathcal{S}$ is bisimilar to that of $\mathcal{S}$ even in the most general case of Petri net rewriting and only the procedures of Petri net analysis are required.*

Let us now turn our attention to a different notion of reachability, namely reachability of abstract markings. Since the state space of a structure-changing Petri net can be infinite, to specify interesting state properties it is insufficient to specify the number of tokens on specific concrete places. Instead, we fix a finite set of colours and state constraints generically, for all places of a given colour. The place colours may carry a model-specific meaning, or just encode structural information about the net. We will therefore introduce a notion of *abstract marking*: a multiset that counts the total number of tokens on places of each colour in a structure-changing Petri net.

A multiset over a finite set S, or S-multiset, is a function $S \to \mathbb{N}$. The set of S-multisets is denoted $\mathbb{N}^S$. The singleton multiset mapping a to 1 and everything else to 0 is also written a. The *size* $|m|$ of a multiset m is the sum of the values. Multiset addition is component-wise. A sum $\sum_{x \in m} f(x)$ over a multiset $m \in \mathbb{N}^S$ is defined with multiplicities, $\sum_{x \in S} m(x) f(x)$. Multisets are compared using the product order, i.e. $m \leq m'$ iff $\forall s \in S,\ m(s) \leq m'(s)$. In marked nets, we redefine ${}^\bullet t$ and $t^\bullet$ to mean the P-multisets $p \mapsto F^\pm(t, p)$ for $\pm = +, -$, respectively. The definition of enabledness canonically extends to multisets of transitions. For any marked net $\mathcal{N} = (N, M)$ with $N = (P, T, F^-, F^+, l, c)$, we define the *colour abstraction* $\alpha : \mathbb{N} \to \mathbb{N}$ to be the function $i \mapsto \sum_{p \in c^{-1}(i)} M(p)$. For k-coloured nets, we restrict the domain of α to $0, ..., k-1$. The set of images under α of the reachable states of $\mathcal{S}$ is denoted by $\mathcal{ARS}(\mathcal{S})$, for *abstract reachability set*.

If $\mathcal{N} = (N, M)$ is a marked net, define a set of multisets over $\{0, ..., k-1\} + T$, the *extended reachability set* $\mathcal{ER}(\mathcal{N})$, as $\mathcal{ER}(\mathcal{N}) = \mathcal{ARS}(\mathcal{N}) + \{\alpha(m - \sum_{t \in s} {}^\bullet t) + s \mid m \in \mathcal{RS}(\mathcal{N}),\ s \in \mathbb{N}^T \text{ enabled by } m\}$. The set $\mathcal{ER}(\mathcal{N})$ is finite if $\mathcal{N}$ is a sound workflow net with its start marking. Intuitively, it represents all snapshots of the net's state before, after and *during* possibly concurrent firing events. Transitions in a multiset from $\mathcal{ER}(\mathcal{N})$ are said to be *activated* (with multiplicity).

The reason why activated transitions are used in the analysis is that from the point of view of the remainder of the net, an *activated* transition t behaves exactly like an *incompletely executed sound workflow net* that replaces the transition t.

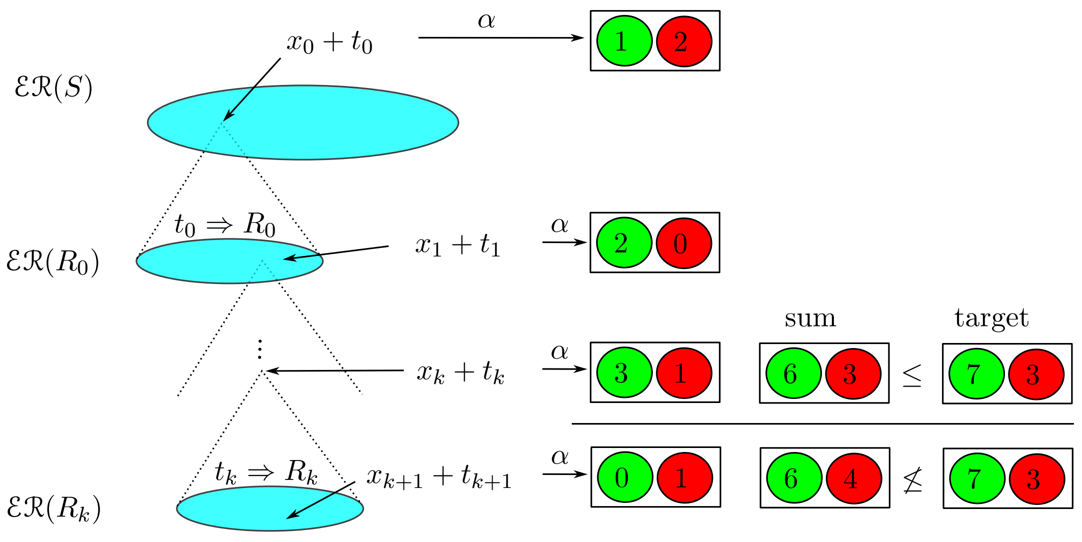


Figure 4.3.: The abstract reachability procedure visualised

Problem 3 (Abstract reachability).

Given:	*a k-coloured structure-changing Petri net $\mathcal{S}$ for some $k \in \mathbb{N}$* $q : \{0, ..., k-1\} \to \mathbb{N}$
Question:	$q \in \mathcal{ARS}(\mathcal{S})$*? Is the abstract marking q reachable?*

Given a structure-changing Petri net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$, we let $\boldsymbol{\mathcal{A}(\mathcal{S})}$ denote the set that comprises $\mathcal{N}$ and the right hand sides of all rules in $\mathcal{R}$. Let $\mathbb{T}(\mathcal{S}) = \biguplus_{N_w \in \mathcal{A}(\mathcal{S})} T_w$ be the set of all transitions occurring in any right hand side or in the start net. ($\biguplus$ stands for a disjoint union of multiple sets). For the intuition behind the following proposition (stating that it is decidable for 1-safe structure-changing workflow nets whether any net with a given abstract marking is reachable), we refer the reader to Figure 4.3.

Proposition 6 (Decidability of Problem 3). *The abstract reachability problem is decidable for 1-safe structure-changing workflow nets.*

Proof. The following algorithm (Figure 4.4), given a structure-changing workflow net $\mathcal{S}$ and $q \in \mathbb{N}^k$, decides whether there is any reachable state $\mathcal{N} \in \mathcal{ARS}(\mathcal{S})$ such that $\alpha(\mathcal{N}) = q$. Let $\mathbb{T} = \mathbb{T}(\mathcal{S})$.

Let us define a directed graph $\mathfrak{W} = (V, E)$ (in the usual mathematical sense, $E \subseteq V \times V$), in general infinite, with a finite set of distinguished root nodes $R \subseteq V$. The node set V is the set of all multisets over $\{0, ..., k-1\} + \mathbb{T}$, and $R = \mathcal{ER}(\mathcal{N})$. The edge set contains all pairs $(t + m, x + m)$ where $t \in \mathbb{T}$, $x \in \mathcal{ER}(N_r)$ and N_r is the right hand side of a rule that matches t (whether a rule can match t can be checked directly from $\mathcal{A}(\mathcal{S})$).

Termination of AbstrReach($\mathcal{S}, q$): observe that multiset size increases monotonically along the edges due to the fact that the empty marking cannot be reachable in a sound

```
procedure AbstrReach($\mathcal{S}, q$)
    Queue := $\mathcal{ER}(\mathcal{N})$; Visited := $\emptyset$
    while Queue not empty do
        $m$ := pop(Queue)
        if $m = q$ then
            return true
        else
            if not ($|m| > |q|$ or $m|_P \not\lesssim q$ or $m \in$ Visited) then
                for all $t, m'$ such that $m = t + m'$ do
                    for all $\{\mu \in match_\varrho(t) \mid (\varrho, N_l, N_r) \in \mathcal{R}\}$ do
                        append(Queue, $\{m' + m \mid m \in \mathcal{ER}(N_r, {}^\bullet N_r\})$)
            Visited := Visited $\cup \{m\}$
    return false
```

Figure 4.4.: Algorithm for the abstract reachability problem

workflow net. To check for the abstract reachability of q, it suffices to explore a spanning forest of the prefix of $\mathfrak{W}$ induced by the nodes of size not exceeding $|q|$, of which there are finitely many. This search is precisely what the algorithm performs. Each node has finitely many descendants due to the finite $\mathcal{ER}$ sets of the right hand side nets. Hence branching is finite and by Kőnig's Lemma each tree in the spanning forest is finite (and the forest has finitely many roots), hence the search terminates.

Correctness of AbstrReach($\mathcal{S}, q$): note that besides those containing at least one activated transition, $\mathfrak{W}$ contains only colour abstractions of reachable markings: a derivation leading to a suitable marked net can simply be read off a path in $\mathfrak{W}$, because in the case of 1-safe structure-changing workflow net it is immaterial *which* transition is replaced, so the abstraction preserves all information needed to construct a suitable derivation. Let us show that the procedure constructs a prefix of $\mathfrak{W}$ breadth-first.

By induction on the number of replacement steps in a $\gtrsim_{\Sigma,R}$-minimal derivation, we show that for $i \in \mathbb{N}$, the procedure always reaches a state where Visited contains the colour abstractions of *all* states reachable within i replacement steps and Queue contains those of all states reachable with $i + 1$ replacement steps.

As long as no rule is applied, the statement holds (the set of states reachable with -1 replacement steps is empty and the set of states reachable with 0 derivation steps equals $\mathcal{ER}(\mathcal{N})$. $\mathcal{ER}(\mathcal{N}) =$ Queue).

Suppose that the induction hypothesis holds for any derivation (ξ, σ) up to a certain length. Any new replacement step $\mathcal{N} \overset{\varrho}{\Rightarrow} \mathcal{N}'$ using a rule ϱ must match some *activated* transition t. Any state obtained by replacing t and executing a non-terminal transition sequence in the right hand side has an abstract marking $m + m''$, where $m + t \in \mathcal{ER}(\mathcal{N})$ and m'' is in the subset of the abstract reachability set of $(\mathcal{N}', \mathcal{R})$ accessible with zero replacement steps. Regardless of the inner loops (which are sure to terminate), the multisets in Queue, which by hypothesis are exactly the abstract states reachable with i replacement steps, are moved one by one into Visited while the successors appended at the back of Queue are precisely the abstract states reachable in $i+1$ replacement steps.

It follows that the algorithm is correct. It explores $\mathfrak{W}$ (in a breadth-first search as measured by the number of replacement steps needed to reach a certain abstract marking) until reaching nodes whose size exceeds $|q|$, and checks whether q is obtained. □

The complexity of the abstract reachability problem is inherited from the corresponding class of static 1-safe nets. For 1-safe nets in general, reachability is PSPACE-complete whereas it is just NP-complete for acyclic 1-safe nets [Esp96].

Proposition 7 (Complexity of acyclic Problem 3). *The abstract reachability problem for acyclic 1-safe structure-changing workflow nets is* NP*-complete.*

Proof. It follows from the proof of Proposition 6 that the problem is not only decidable, but in NP: the answer is positive iff there exists a certain easily checked polynomial-length object, namely the path through $\mathfrak{W}$ at most $O(|\mathcal{R}| \cdot |q|)$ nodes of length $O(|q|)$ each. NP-hardness is straightforwardly shown by a reduction from the corresponding problem for acyclic 1-safe nets. The reachability problem of 1-safe nets can be reduced to the reachability problem of 1-safe workflow nets (see Figure 4.5). This reduction goes as follows: for every place in the 1-safe marked net (N, M_0), a complementary place is added; start end place and $2 + 3|P|$ extra items are added; the simulation is initialised by filling in the start marking and its complement, and it can be aborted without producing erroneous runs by emptying the net. Reachability of M from M_0 becomes reachability of the corresponding marking from the start marking of the workflow net. The workflow net can be seen to be sound. To preserve acyclicity, one removes the complementary places. The reduction of the reachability problem for 1-safe workflow nets to an abstract reachability problem for the corresponding static 1-safe structure-changing workflow net is trivial: places are coloured bijectively. □

Proposition 8 (Complexity of Problem 3). *The abstract reachability problem for 1-safe structure-changing workflow nets is* PSPACE*-complete.*

Proof. PSPACE-hardness again follows from the corresponding problem for 1-safe nets, the reduction using the same construction presented above. The procedure proposed in the proof of Proposition 6 is also easily seen to be in PSPACE because the upper bound on κ is polynomial in $|q|$ and $|R|$. □

As a bonus, $\mathfrak{W}$ can be used to decide coverability of an abstract marking q, i.e. the question whether any state $\mathcal{N}$ with $\alpha(\mathcal{N}) \geq q$ can be reached. Coverability properties are often used as correctness specifications, where an error state is any state that *contains* one of a set of specified *minimal* error states, and a system is correct when it cannot reach such a state (see, for example, [Str14]).

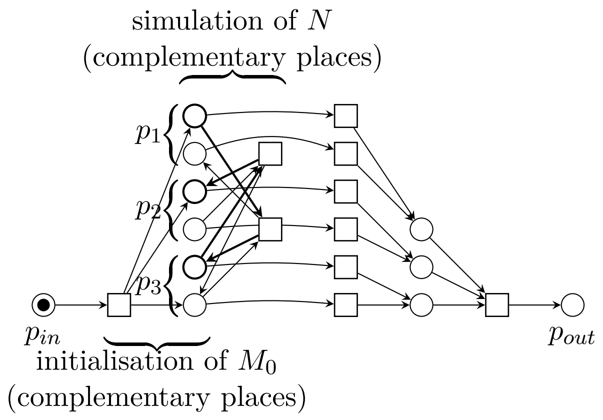


Figure 4.5.: Reduction from 1-safe nets to 1-safe workflow nets.

Proposition 9 (**Decidability of coverability**)**.** *Coverability of an abstract marking is decidable for 1-safe structure-changing workflow nets.*

Proof. $\mathfrak{W}$ is the reachability graph of a net with place set $\{0, ..., k-1\} + \mathbb{T}$ and appropriate transitions that have as preset $t \in \mathbb{T}$ and postset m, for every $m \in \mathcal{ER}(N_r, {}^\bullet N_r)$, where N_r is the right hand side of a rule matching t. Hence coverability is reduced to Petri net coverability, which is decidable (cf. [PW02]). $\square$

4.1.3. Containment Problems

In this section, we study the inclusion of the firing language of a structure-changing workflow net in a given regular language. The motivation is that the regular language can specify all desirable net behaviour, and the problem is to check whether any undesirable firing sequences exist or not.

Problem 4 (**Containment in Regular Language**)**.**

Given:	*a regular language* $L \subseteq \Sigma^*$ *a 1-safe structure-changing Petri net* $\mathcal{S}$
Question:	$\mathfrak{F}(\mathcal{L}(\mathcal{S})) \subseteq L$*?*

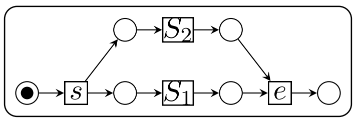
It is well known that the emptiness of the intersection of two context-free languages is undecidable. This problem can be used to show that it is undecidable whether the firing language of even an acyclic 1-safe structure-changing workflow net is contained in a given regular language, which could otherwise be used as a notion of correctness (the regular language specifying correct executions of the net).

Proposition 10 (**Undecidability of abstract language compliance**)**.** *Containment of a structure-changing workflow net language in a regular language is undecidable even* *when restricted to acyclic 1-safe structure-changing workflow nets with at most two tokens in every reachable state.*

Proof. By reduction from the emptiness problem for the intersection of two context-free languages. Let $G_1 = (V_1, T, P_1, S_1)$ and $G_2 = (V_2, T, P_2, S_2)$ be two context-free grammars in Greibach normal form. We assume their non-terminal alphabets disjoint w.l.o.g. ($V_1 \cap V_2 = \emptyset$).

To distinguish the words from the two grammars, we introduce a second terminal alphabet $\hat{T} := \{\hat{a} \mid a \in T\}$ and a function $dup(\epsilon) := \epsilon$, $dup(aw) := a\hat{a} \cdot dup(w)$.

We construct the structure-changing workflow net $\mathcal{S}(G_1, G_2) = (\mathcal{N}, \mathcal{R}_1 \cup \mathcal{R}_2)$ as shown:



For every production $X \to aX_1...X_n$ in G_1, we add a rule replacing a transition labelled X with a linear sequence of transitions labelled $a, X_1, ..., X_n$ to $\mathcal{R}_1$. For productions in G_2, we do the same, but replace the terminal label a with $\hat{a}$ and add the rules to $\mathcal{R}_2$. Disjointness of non-terminal alphabets ensures that the rules in $\mathcal{R}_i$ are only applicable in the subnet resulting from S_i (for $i \in \{1, 2\}$). The words accepted in these subnets hence correspond to those generated by the respective grammars. Let L be $s\{a\hat{a} \mid a \in T\}^*e$ with $\{s, e\} \not\subseteq T$. Now,

$$\begin{array}{rcr} \mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset & \Leftrightarrow & \mathfrak{F}(\mathcal{S}(G_1, G_2)) \subseteq \overline{L} \\ \mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset & \Leftrightarrow & \mathfrak{F}(\mathcal{S}(G_1, G_2)) \cap L = \emptyset \\ \exists w \in \mathcal{L}(G_1) \cap \mathcal{L}(G_2) & \Leftrightarrow & \exists w \in \mathfrak{F}(\mathcal{S}(G_1, G_2)) \cap L \end{array}$$

$\Rightarrow$: if w is generated from both G_1 and G_2, then a derivation for it exists in both grammars. Since context-free derivations in G_1 and G_2 can be translated into sequences of rule applications in the corresponding subnet, there obviously is a reachable net in $\mathcal{S}(G_1, G_2)$ able to accept $s \cdot dup(w)e \in L$.

$\Leftarrow$: All words in L can be written as $s\,dup(w)e$ for some $w \in T^*$. To also be accepted by $\mathcal{S}(G_1, G_2)$, there must be a derivation (ξ, σ) with $f(\lambda(\xi)) = s \cdot dup(w)e$. Now, $w \in \mathcal{L}(G_1)$, since $w = f(\lambda(\xi))$ and the subsequence of $\mathcal{R}_1$-steps in ξ directly corresponds to a derivation in G_1. A symmetric argument holds for G_2.

$\mathcal{S}(G_1, G_2)$ is a 1-safe structure-changing workflow net easily seen to have as reachable states only acyclic nets marked with one or two tokens. $\square$

4.2. Conclusion and Outlook

Reconfigurable Petri nets are of considerable practical interest, mainly in workflow modelling. We have introduced structure-changing Petri nets with context-free transition replacement rules based on graph replacement and studied a number of decision problems that arose. Suitably translated, our results apply to many formalisms that add structure changes to Petri nets. An overview of the results follows. The column headings stand for structure-changing Petri nets and general, 1-safe, acyclic 1-safe structure-changing workflow nets respectively, “yes” means decidable, “no” undecidable, “?” unknown:

	scpn	scwn	1-safe	acyclic 1-safe	
concrete reachability	yes	yes	yes	yes	Proposition 5
abstract reachability	?	?	yes	yes	Proposition 6
firing word problem	?	?	yes	yes	Proposition 4
regular containment	no	no	no	no	Proposition 10

The firing word problem turned out to be nontrivial (we treated the firing word problem for 1-safe structure-changing workflow nets, leaving Problem 1 as an open question). Proposition 10 places severe limitations on the algorithmic analysis of structure-changing Petri nets. Even for systems with a simple structure and a globally bounded token number, language containment questions are undecidable due to the possibility of imposing synchronisation on concurrent context-free processes.

The severe limitations encountered, which preclude automatic analyses in general, are linked to the ability of modelling certain situations. In Petri net theory, it is folklore that “any” proper extension of P/T nets can encode arbitrary computations. This is due to P/T nets coming extremely close to simulating two-counter machines, rendering many questions undecidable. However, one can still attempt to find subclasses such as bounded nets where analogous questions can be answered. Here, a similar phenomenon is encountered for reconfigurations even of bounded nets: under very modest extensions, the formalism is liable to exhibit uncomputable behaviour.

What is the morale?

1. Structure-changing nets with arbitrary replacement rules hardly seem to offer promising analysis methods. It seems most fruitful to investigate classes of nets that behave sufficiently like the “simple” ones presented here, and on the other hand to apply general analysis methods known from graph replacement.
2. Decidable problems for relatively simple subclasses of structure-changing Petri nets and case studies should be performed to evaluate which features are still lacking in order to model real dynamic workflows. We suspect that not all restrictions are necessary and are working to determine the decidability boundary more accurately. We conjecture that the word problem is decidable for general structure-changing workflow nets.

3. In the spirit of *system correctness under adverse conditions*, it will be interesting to turn the reachability problems into game problems by considering some non-trivial scheduling between the two kinds of steps, and universal quantification for replacement steps: until now, rule application and firing cooperate.

Overall, the practical interest of the results in this chapter is to warn of the difficulty of finding interesting decidable subclasses of problems relating to the correctness of graph programs: because the problems posed can be understood as specifications to be checked and structure-changing Petri nets can be encoded as graph programs, we interpret the results as showing the fundamental insufficiency of fully automatic reasoning for the verification of graph programs. The conclusion that one should move on to even *more* general formalisms is only partly paradoxical; on the one hand, it may be possible to obtain further special decidability results, on the other hand fully tackling the verification of structure-changing Petri nets already calls for general (incomplete) methods and therefore little is gained from the restriction in modelling power.

4.3. Bibliographic Notes

Petri nets can be extended with structure changes via graph replacement rules, as in the work of Padberg and Urbášek, Modica et al., Ehrig et al., Padberg [PU03, MGH11, EHP$^+$07, Pad12]. Graph grammars define replacement steps according to rules that serve to reconfigure the Petri net dynamically and can be mixed with transition firings. The work just mentioned, and ours, is closely related to graph grammars and results from graph replacement such as local Church-Rosser and concurrency theorems can be adapted, see Ehrig et al., Modica and Hoffmann. [EHP$^+$07, MGH11] and especially Maximova [MEE12]. This chapter concentrated on Petri net specific aspects such as reachability of markings. Baldan et al. [BCKK04] use Petri net abstractions and unfolding for model checking graph transformation systems. See also Ermel et al. [EE08, EMB$^+$09] and the recent survey of Padberg and Hoffmann [PH15].

Further noteworthy work includes the box calculus of Best [BDH92], the recursive Petri nets of Haddad and Poitrenaud [HP99], the reconfigurable nets of Badouel, Llorens and Oliver [BLO03, LO04] and the open nets of Baldan [BCE$^+$07]. These extensions are much more general and allow structure changes beyond dynamic transition refinement. An alternative would be applying higher-level transformations to the rules, as in Prange et al. [PEHP08].

Safe nets-in-nets (Köhler and Heitmann [KBH10, KBH13]), nested nets (Lomazova [Lom08]) and Hypernets (Mascheroni [Mas11]) represent a different kind of dynamic structure. With unbounded nesting of net tokens, it is possible to simulate a 2-counter machine with zero-tests. The notion of activated transitions and transition refinement are also found in Köhler and Rölke [KR07]. Note that nested Petri net states are also representable as graph transformation systems and a unification of the formalisms could be fruitful.

Finally, in van Hee et al. [vSV03], a notion of refinement for workflow nets was investigated, but with different aims: while literature on lifting results from the analysis of Petri nets to structure-changing Petri nets is relatively scarce, reduction-based approaches for the analysis of Petri nets through graph reductions are sometimes seen.

5. Correctness of Graph Programs

"[...] It is based on rigorous mathematical deduction, which I have gone over myself and which I urge you all to consider. [...]"
— Isaac Asimov, *Foundation's Edge*

Contents

Program verification is the art of establishing the correctness of a program with respect to a specification by means of formal deduction. This chapter constitutes the main part of the thesis. The general verification framework is introduced and instantiated.

We now work in the more general framework of graph transformations. **Sys** and **Env** are re-interpreted as sets of arbitrary graph transformation rules, or more generally graph programs as defined in Chapter 2. Rules or program steps again effect *local* transformations of a *global* state (shared between system and environment), in the sense that the part affected by a rule is of bounded size.

Graph transformations provide a formal way of modelling the graph-based behaviour of a wide range of systems by way of diagrams, amenable to formal verification. One approach to verification proceeds via model checking of abstractions, notably Gadducci et al., Baldan et al., König et al., Rensink et al. [GHK98, BKK03, KK06, RD06]. This can be contrasted with the proof-based approaches of Habel, Pennemann and Rensink [HPR06, HP09] and Poskitt and Plump [PP13]. Here, state properties are expressed by nested graph conditions, and a program can be proved correct with respect to a precondition c and a postcondition d (which are graph conditions). Figure 5.1 on the following page presents a schematic overview of the approach, which is the starting point for this chapter.

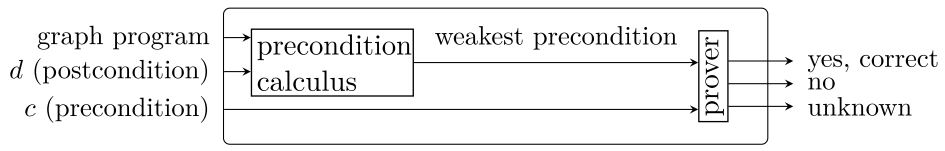


Figure 5.1.: Overview of the proof-based verification approach.

Correctness proofs are performed in the style of Dijkstra's predicate transformer approach (adapted to graph conditions instead of the usual first-order logic with arithmetic) in Pennemann's thesis [Pen09], while Poskitt's thesis [Pos13] features a Hoare logic for partial and total correctness. The fundamental difference to the Hoare approach is that the latter constructs proof trees according to rules for deductive reasoning about specifications, while the Dijkstra approach relies on obtaining a precondition for the whole program first and attempting to solve the implication problem later.

Hoare logic is related to the Dijkstra approach but relies on the construction of proof trees. Dijkstra's approach attempts to construct and present the whole proof by applications of the weakest precondition (or strongest postcondition) transformation, which is a mechanical process on all straight line programs but cannot be fully automated for loops (recursion, iteration). Finally, the implication problem has to be solved to check whether the specified precondition (in the weakest precondition case) implies the weakest precondition (or approximation thereof) obtained from the program and the postcondition.

The above-cited work of Pennemann on the one hand and Plump on the other is based on nested conditions, which cannot express non-local properties of graphs, such as connectivity. In this chapter, we consider correctness with respect to non-local properties, namely the recursively nested conditions of Chapter 3 (with a similar aim as Poskitt and Plump's more recent work [PP14], which however uses Hoare logic and a M-conditions), and we present an extension to the proof calculus from [Pen09].

Nested conditions were first proposed by Habel and Pennemann. They can be used as constraints to specify state properties, or as application conditions to restrict the applicability of a rule. For Petri nets, as in the previous chapter, one can use a logic to specify sets of markings. To be able to express connectivity properties, μ-conditions were introduced in Chapter 3. The next goal is to show that constructions from verification with respect to nested conditions carry over to the new, more expressive conditions. The contributions of this chapter are a generalisation of existing methods to handle the interaction with an environment, and a proof calculus for partial correctness with respect to μ-conditions, together with the proofs and an exemplary application of our method.

In the context of (graph) program correctness, Habel, Pennemann and Rensink [HPR06], Habel and Pennemann [HP09], Pennemann [Pen09] propose a Dijkstra-like [Dij76] notion of *partial correctness*: *Sys* is correct with respect to a pair of nested conditions (c, d) when d holds in every state reachable from a state where c holds. A Hoare calculus for verification of graph programs appears first in Poskitt and Plump [PP12].

In both the Dijkstra-style and the Hoare-style approach, one asks for correctness of a *graph program* with respect to a *program specification* (c, d). Both approaches involve the computation of weakest preconditions of a postcondition relative to a program (the definition is recursive, starting from program steps). The Hoare approach works by having the prover construct proof trees, starting from axioms that reflect the operational semantics of single program steps.

However, a weakest precondition may not be expressible in the assertion language or not computable. The implication check "$\Rightarrow$" may be undecidable. Programs with loops (represented by iteration in graph programs) formally result in infinite disjunctions of weakest preconditions. These infinite objects do not necessarily have a finite representation. To actually verify a program P^*, it is necessary to devise a loop invariant: a valid specification (c, c). These shortcomings are inherent in the verification of arbitrary programs in a Turing-complete language, such as graph programs, and prevent fully automatic reasoning in general.

The structure of the chapter is as follows: we develop the weakest precondition calculus for μ-conditions and prove its soundness in Section 5.1. Then in Section 5.2 we present a proof calculus for μ-conditions, to be used together with the weakest precondition calculus. Section 5.3 concludes the chapter with an outlook and Section 5.4 provides context by collating related work.

5.1. Weakest Liberal Preconditions

The weakest liberal precondition transformation lies at the core of the Dijkstra-style approach. The existence of a construction for this transformation is a desirable property of a graph condition formalism. It can be used to prove correctness:

Definition 31 (**Partial Correctness**). *A graph program P is* (partially) correct *with respect to a precondition c and a postcondition d if and only if for all $(f, f', p) \in [\![P]\!]$, $f \models c$ implies $f' \models d$.*

The related notion of *total* correctness imposes the supplementary requirement that a program must *terminate* from any state satisfying the precondition. Total correctness implies partial correctness. Only partial correctness is considered here. In this section, we present a construction to compute the weakest liberal precondition of a μ-condition with respect to any iteration-free graph program P. "Liberal" means that termination of P is not implied, it is thus the notion of weakest precondition that is adequate for partial correctness. As only iteration causes non-termination, it is redundant in the absence of iteration and weakest precondition and weakest liberal precondition are the same.

Definition 32 (**Weakest Liberal Precondition**). *The weakest liberal precondition (wlp) of d with respect to the program P,* $\mathrm{wlp}(P, d)$*, is the least condition with respect to implication such that $f \models \mathrm{wlp}(P, d) \Rightarrow f' \models d$ if $(f, f', p) \in [\![P]\!]$ for some partial monomorphism p.*

We show that under this assumption there is a μ-condition that expresses precisely the weakest liberal precondition of a given μ-condition with respect to a rule, and it can be computed. The result is similar to the situation for nested conditions. To derive it, we use the *shift* transformation $A_m(c)$ from [Pen09] whose fundamental property is to transform any nested condition c into another nested condition such that $m'' \models A_m(c)$ if and only if $m'' \circ m \models c$ for all composable pairs m'', m of monomorphisms (Lemma 5.4 from [Pen09]). Since this and similar constructions play an important role in this section, we recall it here.

The shift construction and its cousins, to be introduced in this section, require us to consider equivalence classes of epimorphisms. An equivalence class of the relation introduced in Definition 33 is usually called a quotient object:

Definition 33 (**Quotient Objects** ([Bor94], p. 132), $\cong_{\text{Epi}}$)**.** *Let f, g are epimorphisms with* $\text{dom}(f) = \text{dom}(g)$*. $f \cong_{\text{Epi}} g$ holds whenever there is an isomorphism $v : \text{cod}(f) \cong \text{cod}(g)$ such that $v \circ f = g$. Clearly, the relation $\cong_{\text{Epi}}$ is reflexive, symmetric and transitive.*

$$\begin{tikzcd} & B \arrow[dd, "v"] \arrow[dd, phantom, "\cong", swap] \\ A \arrow[ur, "f"] \arrow[dr, "g", swap] & \\ & C \end{tikzcd}$$

The epimorphisms (surjective morphisms) from any given graph G certainly fall into finitely many $\cong_{\text{Epi}}$ equivalence classes since each epimorphism with source G has a codomain that is at most as large as G.

Definition 34 (**Shift Construction** ([Pen09], Lemma 5.4))**.** *Let $m \in \mathcal{M}$ and $c : \text{dom}(m)$. Then $A_m(\neg c') = \neg A_m(c')$ and $A_m(\bigwedge_{j\in J} c_j) = \bigwedge_{j\in J} A_m(c_j)$. The case of existential quantification $c = \exists(a, c')$ is as follows: let (m', a') be the pushout of (m, a). Let* Epi *be the set of all $\cong_{\text{Epi}}$-equivalence classes of epimorphisms e with domain* $\text{cod}(m')$ *whose (arbitrarily chosen) representants compose to monomorphisms $b := e \circ a'$ and $r := e \circ m'$. Then $A_m(\exists(a, c')) = \bigvee_{e\in\text{Epi}} \exists(b, A_r(c'))$.*

Intuitively, the shift construction instantiates the diagram of Figure 5.2 in all possible ways, starting from the morphisms $A \hookrightarrow B$ and $A \hookrightarrow C$ and their pushout.

The fundamental property of the shift over a monomorphism m is that it transforms any condition over $\text{dom}(m)$ into one over $\text{cod}(m)$, preserving and reflecting satisfaction:

Lemma 12 (**Fundamental Property of Shift** ([Pen09], Lemma 5.4))**.** *For all $m \in \mathcal{M}$ and $c : \text{dom}(m)$, we have $\forall m'' \in \mathcal{M}$, $\text{dom}(m'') = \text{cod}(m) \Rightarrow A_m(c) : \text{cod}(m'')$ such that $(m'' \models A_m(c) \Leftrightarrow m'' \circ m \models c)$*

With help of the unselection ι in $\exists(a, \iota, c)$, it is at first glance very easy to exhibit a weakest liberal precondition with respect to $\text{Uns}(y)$: it is sufficient to wrap the main

$$\begin{tikzcd}
A \arrow[rr, hook, "a"] \arrow[dd, hook, "m"'] & & B \arrow[dl, "m'"'] \arrow[dd, hook, "r"] \\
& D \arrow[dr, "e"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & \\
C \arrow[ur, hook, "a'"] \arrow[rr, hook, "b"'] & & E
\end{tikzcd}$$

Figure 5.2.: Diagram for shift

body $b : \mathrm{dom} y$ in an existential quantification, $\exists^{-1}(y, b)$. However, to handle the addition and deletion steps, a construction becomes necessary to make the affected subgraph appear explicitly. This information is crucial to obtain the weakest liberal precondition with respect to $\mathrm{Add}(r)$ and $\mathrm{Del}(l)$ and is of crucial importance at any nesting level in order to obtain the correct result. To that aim, we define a *partial shift* transformation which makes sure that the type of the main body of the transformed μ-condition is never unselected but is instead mapped in a consistent way as a subgraph of the type of each variable.

The following construction serves to obtain the new types; it makes precise the intuition that all possible *intersections* of the old type and the current interface must be taken into account in order to track the changes that must be effected when the interface changes. When the construction is used, the following is given as input: a condition with placeholders $c : B$ or a (left hand side) variable $\mathtt{x}_i : B$; a graph R that will remain constant throughout the construction; a pair of morphisms $x : B \hookrightarrow H$ and $y : R \hookrightarrow H$ with common codomain, (x, y) assumed to be jointly surjective.

Construction 7 (New type for partial shift). *Assume that an arbitrary total order on all graph morphisms is fixed. Given a μ-condition $c = (b \mid \vec{\mathbf{x}} = \mu[\vec{K}]\vec{\mathcal{F}}(\vec{\mathbf{x}}))$, for each variable $\mathbf{x}_i$ of $\vec{K}$, morphisms f_{ij} are obtained from $\vec{B}'$ by collecting arbitrary representants of all $\cong_{\mathrm{Epi}}$-classes of epimorphisms (indexed by $j \in J$, known to be finite in the case of graphs) that compose to monomorphisms with the pushout morphisms in the diagram:*

$$\begin{tikzcd}
\emptyset \arrow[r, hook] \arrow[d, hook] & R \arrow[d, hook] \arrow[dr, hook, bend left, "f_{ij}"] & \\
B_i \arrow[r, hook] \arrow[rr, hook, bend right] & X \arrow[r, two heads] & B'_{ij}
\end{tikzcd}$$

$\mathcal{X}_{R,c}(\mathbf{x}_i)$ is defined as the sequence of morphisms $\vec{f} = \{f_{ij}\}_{j \in J}$ in ascending order. The new list of variables $\vec{K}'$ and their respective types $\vec{B}'$ are obtained by concatenating all $\mathcal{X}_{R,c}(\mathbf{x}_i)$ of the variables of $\vec{K}$, ordered by the outer index i and then by the inner index j.

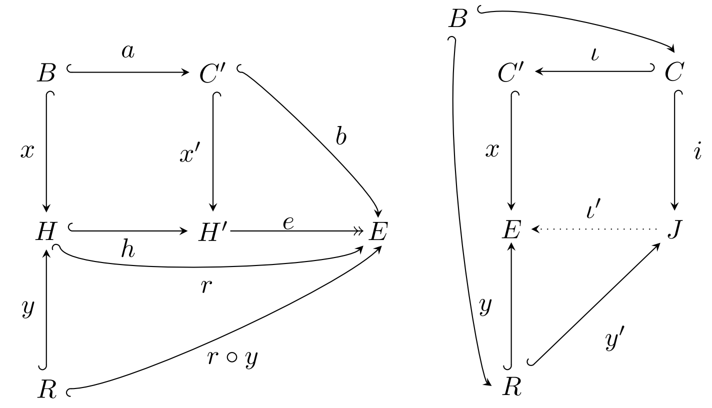
The ordering that appears in the definition is an artefact of our decision to use lists of equations, rather than unordered families. Using the new lists of types and variables, we can now proceed to define the bulk of the partial shift operation, namely its effect on the right hand sides.

Construction 8 (**Partial shift** $\mathcal{P}_{x,y}$)**.** *Given jointly surjective monomorphisms $x : B \hookrightarrow H$ and $y : R \hookrightarrow H$, we define the* partial shift *of $(b \mid \mu[\vec{K}]\mathcal{F})$ with respect to (x, y) as $\mathcal{P}_{x,y}(b \mid \mu[\vec{K}]\mathcal{F}) := (\mathcal{P}_{x,y}(b) \mid \mu[\vec{K'}]\mathcal{F}')$, where the new equations are obtained by applying $\mathcal{P}_{f_{ij},y}$ to the variables of the left hand sides with all possible morphisms f_{ij} from R (cf. Construction 7) and accordingly to the right hand sides. On condition bodies, $\mathcal{P}_{x,y}$ is defined as follows: Boolean combinations of conditions are transformed to the corresponding combinations of the transformed members. The cases of variable occurrences, existential quantification and unselection are as follows:*

For any jointly surjective pair of monomorphisms $x : B \hookrightarrow H$ and $y : R \hookrightarrow H$, $\mathcal{P}_{x,y}(\boldsymbol{x}_i) := \exists(v, \boldsymbol{x}_i^{x,y})$ if $\boldsymbol{x}_i : B$, where $\boldsymbol{x}_i^{x,y} : H$ is a new variable, H is isomorphic to $\mathrm{cod}(y)$, *and $v :$* $\mathrm{cod}(y) \cong H$ *is the isomorphism from* $\mathrm{cod}(y)$ *to the type H* (*which is arbitrarily fixed once and for all for the variable $\boldsymbol{x}_i^{x,y}$*)*. Quantification is processed separately from unselection* (*Lemma 1*)*:*

For any jointly surjective pair of monomorphisms $x : B \hookrightarrow H$ and $y : R \hookrightarrow H$, $\mathcal{P}_{x,y}(\exists(B \overset{a}{\hookrightarrow} C', c')) := \bigvee_{\mathrm{Epi}} \exists(H \hookrightarrow E, \mathcal{P}_{b,r\circ y}(c'))$, where Epi *is the set of all $\cong_{Epi}$-classes of epimorphisms e with domain H' that compose to monomorphisms $r = e \circ x'$ and $b = e \circ h$ with the pushout morphisms* (*see diagram below left*)*.* Epi

For any jointly surjective pair of monomorphisms $x : C' \hookrightarrow E$ and $y : R \hookrightarrow E$, let $\mathcal{P}_{x,y}(\exists^{-1}(C' \overset{\iota}{\hookleftarrow} C, c')) := \exists^{-1}(\iota', \mathcal{P}_{i,y'}(c'))$: form the pullback of $r \circ \iota$ and $b \circ y$, then pushout the obtained morphisms to (y', i) (*see diagram below right*)*.*


$$\begin{tikzcd}
B \arrow[r, hook, "a"] \arrow[d, hook, "x"'] & C' \arrow[d, hook, "x'"'] \arrow[dr, hook, bend left, "b"] & \\
H \arrow[r, hook, "h"'] \arrow[rr, hook, bend right, "r"'] & H' \arrow[r, two heads, "e"] & E \\
R \arrow[u, hook, "y"] \arrow[urr, hook, "r \circ y"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
B \arrow[drr, hook, bend left] \arrow[ddd, hook, bend right] & & \\
 & C' \arrow[d, hook, "x"'] & C \arrow[l, hook', "\iota"'] \arrow[d, hook, "i"] \\
 & E & J \arrow[l, dotted, "\iota'"'] \\
 & R \arrow[u, hook, "y"] \arrow[ur, hook, "y'"'] &
\end{tikzcd}$$

The pair of monomorphisms (x, y) given as parameters for the partial shift are always assumed to be jointly surjective, even if the assumption is not strictly necessary to define Construction 8; this property is guaranteed to carry over to $(b, r \circ y)$ in the case of existential quantification and to (i, y') in the case of unselection. Note that partial shift constructs a new monomorphism $y' := r \circ y : R \hookrightarrow E$, respectively $y' : R \hookrightarrow J$, at each nesting level of the construction with placeholders it processes. This yields, for each subcondition occurrence $c' : B'$ in c, one new monomorphism $y_{c'} : R \hookrightarrow B'$ that is *consistent* with the morphisms occurring in the partially shifted condition in the sense that the compositions commute ($y' = r \circ y$ or $y = y' \circ \iota'$). As the subsequent constructions heavily rely on these monomorphisms $y_{c'}$, they must be viewed as an integral part of partial shift's output, not a temporary result.

Remark 9 (**Ambiguous Variable Contexts**). *In a μ-condition it is not necessarily true that in all contexts where $\mathtt{x}_i$ is used, it appears with the same morphism $R \hookrightarrow B_i$ (where R is the type of b). It is however possible to equivalently transform every μ-condition into a "normal form" that has that property. Applying $\mathcal{P}_{id_R, id_R}$ will by construction result in a μ-condition with unambiguous inclusions $R \hookrightarrow B_i$ for all variables* (*namely the morphisms from the sequences $\mathcal{X}_{R,c}$*)*. μ-condition.*

Proof. By construction, each occurrence of a variable $\mathtt{x}_i^{x,y}$ created in the partial shift has the same type H (not merely an isomorphic one). The monomorphism $y : R \hookrightarrow H$ constructed by Construction 8 must be the same wherever the variable occurs because R is never unselected. □

The property of unambiguous variable contexts is also preserved by the constructions introduced later in this section. Unreachable variables created by $\mathcal{X}$ and $\mathcal{P}$ can be pruned to obtain an equivalent

Equivalence of conditions with placeholders (unlike μ-conditions) is defined for conditions using the same sets of variables, as equivalence in the sense of nested conditions under any valuation. We extend A to conditions with placeholders by defining $A_m(\mathtt{x})$ as $\exists(id_{\mathrm{cod}(m)}, m, \mathtt{x})$ for $\mathtt{x} : B$. We show below that $\mathcal{P}_{x,y}$ is equivalent to A_x. The reason for introducing $\mathcal{P}_{x,y}$ is to gain precise control over the types of the variables in the transformed condition, which should all include the type of the main body. Intuitively, as this corresponds to the currently selected subgraph of a graph program, additions and deletions are applied to that subgraph and one must ensure that the changes apply to the whole μ-condition. Three minor lemmata are required:

Lemma 13 (**Removal of unselection**). *If c' is a condition with placeholders, then $\exists(a, \iota, c') \equiv \exists(a, A_\iota(c'))$ (A as defined in Definition 34).*

Proof. Using the fundamental property of A, the nontrivial case being $m \models \exists(a, \iota, c')$ $\Leftrightarrow \exists q \in \mathcal{M}, q \circ a = m \wedge q \circ \iota \models c' \Leftrightarrow \exists q \in \mathcal{M}, q \circ a = m \wedge q \models A_\iota(c') \Leftrightarrow m \models \exists(a, A_\iota(c'))$. □

Lemma 14 (**Shift composition and decomposition**). *Given two morphisms m'', m', if $m'' \circ m'$ exists, then $A_{m'' \circ m'}(c) \equiv A_{m''}(A_{m'}(c))$ for all conditions (with placeholders) c.*

Proof. $f \models A_{m'' \circ m'}(c) \Leftrightarrow f \circ m'' \circ m' \models c \Leftrightarrow f \circ m'' \models A_{m'}(c) \Leftrightarrow f \models A_{m''}(A_{m'}(c))$ (Lemma 5.4 in [Pen09]) □

Partial shift transforms a (recursively nested) condition to another, equivalent one:

Lemma 15 (**Partial shift**). *The conditions $\mathcal{P}_{x,y}(c)$ and $A_x(c)$ are equivalent.*

Proof. By induction over the recursion depth, and structural induction over c: If c is a variable symbol $\mathtt{x}_i$, then either recursion depth is 0 and the assertion is proved, since it is $\bot$, or it is true because the right hand side of the equation for $\mathtt{x}_i$ has the property. The case $\exists(a, \iota, c')$ of the structural induction is handled as follows (disjunctions ranging over the suitable epimorphisms e and compositions b, r):

$m \models A_x(\exists(a, \iota, c')) \Leftrightarrow m \models A_x(\exists(a, A_\iota(c')))$ according to Lemma 13
$\Leftrightarrow m \models \bigvee \exists(b, A_{e \circ x'}(A_\iota(c')))$ by Lemma 12
$\Leftrightarrow m \models \bigvee \exists(b, A_{e \circ \iota'}(A_i(c'))$ by Lemma 14 (twice)
$\Leftrightarrow m \models \bigvee \exists(b, e \circ \iota', A_i(c'))$ by Lemma 13
$\Leftrightarrow m \models \bigvee \exists(b, e \circ \iota', \mathcal{P}_{i,r'}(c'))$ by induction hypothesis
$\Leftrightarrow m \models \mathcal{P}_{x,r'}(\exists(a, \iota, c'))$ by Construction 8 □

The constructions and proofs in the following section use the pushout-pullback decomposition [HEP06], also known as the special pushout-pullback lemma [EEPT06]. The proof of this standard lemma can be found in [EEPT06]:

Lemma 16 (**Pushout-pullback decomposition** [EEPT06]).
Consider graphs A, B, C, D, E, F and morphisms arranged as in the diagram:

$$\begin{tikzcd}
A \arrow[r, hook, "a"] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & C \arrow[d] \\
D \arrow[r] & E \arrow[r] & F
\end{tikzcd}$$

If $a : A \hookrightarrow B$ is a monomorphism and the square (2) is a pullback and the outer square is a pushout square, then (1) and (2) are both pullback and pushout squares.

5.1.1. The Addition and Deletion Transformations

We introduce two transformations $\delta'_l(c)$, $\alpha'_r(c)$ (based on auxiliary transformations $\delta_{l,y}(c)$ and $\alpha_{r,y}(c)$). These are applied to main body and right hand sides and serve to compute the weakest precondition with respect to addition and deletion, respectively[1], of a μ-condition that has already undergone partial shift. Recall the statement of Remark 9 that partial shift fixes inclusions from the current interface to each graph occurring in the condition (as domain or codomain of a morphism a or ι). When the condition c obtained after partial shift is evaluated on a morphism to check satisfaction, the current interface is never unselected in the recursion but appears included in each variable type. The condition $\alpha'_r(c)$ stipulates the existence of cod(r) (the Add(r) step's input interface)

[1]The letters were chosen to indicate the effect of the transformation: to compute the weakest precondition with respect to *addition*, δ' needs to *delete* portions of the morphisms in the condition, and vice versa.

instead of dom(r) (the output interface), which is intuitively why it yields the correct expression of the weakest precondition of c with respect to Add(r). It might well be that an occurrence of cod(r) cannot have been obtained by a rule application because the pushout demanded by the semantics of Add(r) fails to exist, in which case α' eliminates a branch of the condition. Likewise, in $\delta'_l(c)$, cod(l) takes the place of dom(l) since this corresponds exactly to the effect of the step Del(l).[2]

Construction 9 (**Transformation** δ')**.** *Let $c : B$ be a condition with placeholders. If $r : K \hookrightarrow R$ and $y : R \hookrightarrow B$ are monomorphisms, then $\delta_{r,y}(c)$ is defined as follows: $\delta_{r,y}(\neg c) = \neg\delta_{r,y}(c)$ and $\delta_{r,y}(\bigwedge_{j\in J} c_j) = \bigwedge_{j\in J} \delta_{r,y}(c_j)$. For $c = \exists(a, \iota, c')$, use the decomposition from Lemma 1 and handle the cases of $\exists(a, c)$ and $\exists^{-1}(\iota, c)$ separately:*[3]

$$\underbrace{\begin{tikzcd}
W \arrow[rr, hook, "a'"] \arrow[dr, hook, "r''"] & & X \arrow[dr, hook, "r'"] & \\
& B \arrow[rr, hook, "a"'] & & C' \\
K \arrow[uu, hook, "h"] \arrow[uurr, "h'"] \arrow[dr, hook, "r"'] & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, "y'"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[dr, hook, "r'"] & & V \arrow[ll, hook, "\iota'"'] \arrow[dr, hook, "r'''"] & \\
& C' & & C \arrow[ll, hook, "\iota"] \\
K \arrow[uu, hook, "h"] \arrow[uurr, "h'"] \arrow[dr, hook, "r"'] & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, "y'"'] & &
\end{tikzcd}}_{\delta_{r,y}(\exists(a,c)) \quad \text{and} \quad \delta_{r,y}(\exists^{-1}(\iota,c))}$$

Case of $\delta_{r,y}(\exists(a, c))$: given are $r : K \hookrightarrow R$, $a : B \hookrightarrow C'$, the subcondition $c : C'$ and the monomorphisms y, y', ... obtained in the partial shift construction. The diagrams below depict the steps of the construction, which are detailed below.

$$\begin{tikzcd}
& B \arrow[rr, hook, "a"'] & & C' \arrow[r, phantom, "\lhd"] & c \\
K \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, "y'"'] & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& & X \arrow[dr, hook, "r'"] & & \\
& B \arrow[rr, hook, "a"'] \arrow[dr, phantom, "(\mathrm{PO})"] & & C' \arrow[r, phantom, "\lhd"] & c \\
K \arrow[uurr, "h'"] \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, "y'"'] & & &
\end{tikzcd}$$

[2]In the case of Del(l), it is possible that $\delta'_l(c)$ specifies an occurrence of l which cannot be the input of a Del(l) step. Hence to obtain the actual weakest precondition, a nested condition expressing the applicability of Del(l) must be adjoined to $\delta'_l(c)$.

[3]The morphism y', like y, was obtained during partial shift; the transformations yield corresponding morphisms h' from the new program interface to each graph occurring in the condition body.

$$\begin{tikzcd}
W \arrow[rr, hook, "a'"] \arrow[dr, hook, "r''"'] & & X \arrow[dr, hook, "r'"] & & \\
& B \arrow[rr, hook, "a"'] \arrow[ur, phantom, "(PB)"] & & C' \arrow[r, phantom, "\triangleleft"] & c \\
K \arrow[uurr, hook, "h'"] \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"'] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
W \arrow[rr, hook, "a'"] \arrow[dr, hook, "r''"'] & & X \arrow[dr, hook, "r'"] \arrow[r, phantom, "\triangleleft"] & \delta_{r,y'}(c) & \\
& B \arrow[rr, hook, "a"'] & & C' \arrow[r, phantom, "\triangleleft"] & c \\
K \arrow[uu, hook, dotted, "h"] \arrow[uurr, hook, "h'"] \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"'] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}$$

If no pushout complement of r and $y' = a \circ y$ exists, then $\delta_{r,y}(\exists(a,c)) = \bot$. Otherwise, obtain it as (h', r') ($h' : K \hookrightarrow X$, $r' : X \hookrightarrow C'$) and pullback (a, r') to (a', r'') with source W; this yields a unique morphism h from K to W to make the diagram commute. Apply Lemma 16 to the compositions $h' = a' \circ h$ and $y' = a \circ y$ to see that the left and top squares in the diagram are pushouts. Then $\delta_{r,y}(\exists(a,c)) = \exists(a', \delta_{r,y'}(c))$.

Case of $\delta_{r,y}(\exists^{-1}(\iota, c))$: given are $r : K \hookrightarrow R$, $\iota : C \hookrightarrow C'$, the subcondition $c : C$ and the monomorphisms y, y', ... obtained in the partial shift construction. The diagrams below depict the steps of the construction, which are detailed below.

$$\begin{tikzcd}
& C' & & C \arrow[ll, hook', "\iota"] \arrow[r, phantom, "\triangleleft"] & c \\
K \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[dr, hook, "r'"] & & & & \\
& C' \arrow[l, phantom, "(PO)"] & & C \arrow[ll, hook', "\iota"] \arrow[r, phantom, "\triangleleft"] & c \\
K \arrow[uu, hook, "h"] \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}$$

$$\begin{tikzcd}
X \arrow[dr, hook, "r'"] & & V \arrow[ll, hook', "\iota'"'] \arrow[dr, hook, "r'''"] & & \\
& C' \arrow[ur, phantom, "(PB)"] & & C \arrow[ll, hook', "\iota"] \arrow[r, phantom, "\triangleleft"] & c \\
K \arrow[uu, hook, "h"] \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[dr, hook, "r'"] & & V \arrow[ll, hook', "\iota'"'] \arrow[dr, hook, "r'''"] \arrow[r, phantom, "\triangleleft"] & \delta_{r,y'}(c) & \\
& C' & & C \arrow[ll, hook', "\iota"] \arrow[r, phantom, "\triangleleft"] & c \\
K \arrow[uu, hook, "h"] \arrow[uurr, dotted, "h'"] \arrow[dr, hook, "r"'] & & & & \\
& R \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}$$

Construct a pushout complement (h, r') where $h : K \hookrightarrow X$ and $r' : X \hookrightarrow C'$. If it does not exist, then $\delta_{r,y}(\exists^{-1}(\iota, c)) = \bot$. Otherwise construct a pullback (ι', r''') of (ι, r') with $V := \mathrm{dom}(\iota')$. The pullback property yields existence and uniqueness of $h' : K \to V$ that makes the diagram commute. Then $\delta_{r,y}(\exists^{-1}(\iota, c)) = \exists^{-1}(\iota', \delta_{r,y'}(c))$.

For variables, $\delta_{r,y}(\boldsymbol{x}_i) = \boldsymbol{x}'_i$ is a new variable of type K (see Remark 9).

Note that the construction is well-defined due to Fact 1, since the pushout complement constructed on the left face of the diagram for the *subcondition c* must agree with the pushout complement constructed on the oblique face of the diagram for the condition

$\exists(a, c)$ or $\exists^{-1}(\iota, c)$. But this is only true up to isomorphism of the graph in the pushout complement, W in the case of existential quantification or X in the case of unselection (Boolean combinations pose no problem, as they merely transmit the choice of pushout complement). This minor issue is best resolved by stipulating that the pushout complements must always be chosen consistently, by judicious use of Lemma 3, adapting Construction 9 to compose the appropriate isomorphisms after a' and before ι'.

The transformation α', which handles deletions steps by adding the deleted parts to the condition, is largely analogous, but there are some differences due to the fact that pushouts, rather than pushout complements, are constructed.

Construction 10 (**Transformation** α')**.** *Let $c : B$ be a condition with placeholders. $l : K \hookrightarrow L$ and $y : K \hookrightarrow B$ are monomorphisms, then $\alpha_{l,y}(c)$ is defined as follows: $\alpha_{l,y}(\neg c) = \neg\alpha_{l,y}(c)$ and $\alpha_{l,y}(\bigwedge_{j\in J} c_j) = \bigwedge_{j\in J} \alpha_{l.y}(c_j)$. For $c = \exists(a, \iota, c')$, use the decomposition from Lemma 1 and handle the cases of $\exists(a, c)$ and $\exists^{-1}(\iota, c)$ separately:*

$$\begin{tikzcd}
W \arrow[rr, hook, "a'"] & & X & \\
& B \arrow[ul, hook', "l'"] \arrow[rr, hook, "a"'] & & C' \arrow[ul, hook', "l''"'] \\
L \arrow[uu, hook', "h"] \arrow[uurr, hook, "h'"'] & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
X & & V \arrow[ll, hook', "\iota'"'] & \\
& C' \arrow[ul, hook', "l''"] & & C \arrow[ll, hook', "\iota"] \arrow[ul, hook', "l'''"'] \\
L \arrow[uu, hook', "h"] \arrow[uurr, hook, "h'"'] & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & &
\end{tikzcd}$$

$\alpha_{l,y}(\exists(a, c))$ and $\alpha_{l,y}(\exists^{-1}(\iota, c))$

Case of $\alpha_{l,y}(\exists(a, c))$: given are $l : K \hookrightarrow L$, $a : B \hookrightarrow C'$, the subcondition $c : C'$ and the monomorphisms y, y', ... obtained in the partial shift construction. The diagrams below depict the steps of the construction, which are detailed below.

$$\begin{tikzcd}
& B \arrow[rr, hook, "a"'] & & C' \arrow[r, phantom, "\lhd"] & c \\
L & & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
W & & & & \\
& B \arrow[ul, hook', "l'"] \arrow[rr, hook, "a"'] & & C' \arrow[r, phantom, "\lhd"] & c \\
L \arrow[uu, hook', "h"] \arrow[ur, phantom, "(\mathrm{PO})"] & & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}$$

$$\begin{tikzcd}
W \arrow[rr, hook, "a'"] & & X & & \\
& B \arrow[ul, hook', "l'"] \arrow[rr, hook, "a"'] \arrow[u, phantom, "(\mathrm{PO})"] & & C' \arrow[ul, hook', "l''"'] \arrow[r, phantom, "\lhd"] & c \\
L \arrow[uu, hook', "h"] & & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
W \arrow[rr, hook, "a'"] & & X \arrow[r, phantom, "\lhd"] & \alpha_{l,y'}(c) & \\
& B \arrow[ul, hook', "l'"] \arrow[rr, hook, "a"'] & & C' \arrow[ul, hook', "l''"'] \arrow[r, phantom, "\lhd"] & c \\
L \arrow[uu, hook', "h"] \arrow[uurr, hook, "h'"'] & & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"] \arrow[uurr, hook, bend right, "y'"'] & & &
\end{tikzcd}$$

Construct a pushout (l', h) *of* (y, l) *and a pushout* (l'', a') *of* (l', a)*. The monomorphism* h' *is obtained as* $a' \circ h$ *and the composed square is a pushout by pushout composition. Then* $\alpha_{l,y}(\exists(a, c)) = \exists(a', \alpha_{l,y'}(c))$*.*

Case of $\alpha_{l,y}(\exists^{-1}(\iota, c))$*: given are* $l : K \hookrightarrow L$*,* $\iota : C \hookrightarrow C'$*, the subcondition* $c : C$ *and the monomorphisms* y*,* y'*, ... obtained in the partial shift construction. The diagrams below depict the steps of the construction, which are detailed below.*

$$\begin{tikzcd}
& C' & C \arrow[l, hook', "\iota"] \arrow[r, phantom, "\lhd"] & c \\
L & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"'] \arrow[uur, hook, "y'"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
X & & & \\
& C' \arrow[ul, hook, "l''"'] & C \arrow[l, hook', "\iota"] \arrow[r, phantom, "\lhd"] & c \\
L \arrow[uu, hook, "h"] \arrow[ur, phantom, "(\mathrm{PO})"] & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"'] \arrow[uur, hook, "y'"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
X & & V & & \\
& C' \arrow[ul, hook, "l''"'] & & C \arrow[ll, hook', "\iota"] \arrow[ul, hook, "l'''"'] \arrow[r, phantom, "\lhd"] & c \\
L \arrow[uu, hook, "h"] \arrow[uurr, hook, "h'"'] & & (\mathrm{PO}) & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"'] \arrow[uurr, hook, "y'"'] & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
X & & V \arrow[ll, hook', dotted, "\iota'"'] \arrow[r, phantom, "\lhd"] & \alpha_{l,y'}(c) & \\
& C' \arrow[ul, hook, "l''"'] & & C \arrow[ll, hook', "\iota"] \arrow[ul, hook, "l'''"'] \arrow[r, phantom, "\lhd"] & c \\
L \arrow[uu, hook, "h"] \arrow[uurr, hook, "h'"'] & & & & \\
& K \arrow[ul, hook', "l"] \arrow[uu, hook, "y"'] \arrow[uurr, hook, "y'"'] & & &
\end{tikzcd}$$

Let (h, l'') *be a pushout over* (y, l) *and* (h', l''') *a pushout over* (y', l)*. The commuting morphism from the latter pushout object to* X *is* ι'*. Then* $\alpha_{l,y}(\exists^{-1}(\iota, c)) = \exists^{-1}(\iota', \alpha_{l,y'}(c))$*.*

For variables, $\alpha_l(\boldsymbol{x}_i)$ *is a new variable of type* L *(see Remark 9).*

In contrast to the partial shift $\mathcal{P}$, the transformations α' and δ' leave the number of variables unchanged. Only the types of the variables are modified.

Finally, $\delta'_r(c)$ and $\alpha'_l(c)$ are derived from the more general Construction 9 and Construction 10 by evaluating these construction using the *identity* on the external interface (i.e. the type of the main body) for both morphism arguments.

Definition 35 (Transformations δ' and α').
$\delta'_r(c) = \delta_{r,id}(\mathcal{P}_{id,id}(c))$ *and* $\alpha'_l(c) = \alpha_{l,id}(\mathcal{P}_{id,id}(c))$*.*

We recall that for any $l : K \hookrightarrow L$, there is a *deletability* condition $\Delta(l)$ that expresses the possibility of effecting $\mathrm{Del}(l)$, i.e. $\Delta(l)$ is satisfied exactly by the first components of tuples in $[\![\mathrm{Del}(l)]\!]$. We describe $\Delta(l)$ only informally: $f \models \Delta(l)$ states the non-existence of edges that are in $im(f)$ but incident to a node in $im(f) - im(f \circ l)$.

Now we have all ingredients for a weakest liberal precondition theorem for μ-conditions. The proofs again rely on the general algebraic framework underlying double-pushout rewriting [EEPT06], hence the language of category theory is used for a concise presentation.

Having introduced nested conditions, we make a minor modification of Definition 3 (graph programs). As nested conditions were originally conceived as (negative) application conditions to graph transformation rules, it is convenient to regulate the applicability of a selection by making it dependent on a nested condition.

Definition 36 (Conditional Selection).

Name	*Program P*	*Semantics* $[\![P]\!]$
conditional selection	$Sel(x, c)$	$\{(m_{in}, m_{out}, x) \mid m_{out} \circ x = m_{in}, m_{in} \models c\}$

5.1.2. Correctness of the Constructions

We exhibit and prove the weakest precondition transformation for the four elementary graph programs and the cases of sequential and nondeterministic composition.

Definition 37 (Weakest Precondition Construction). *Define a transformation* Wlp *that accepts a graph program and a condition with placeholders (a μ-condition) and outputs a condition with placeholders (respectively a μ-condition):*

1. $\mathrm{Wlp}(\mathrm{Uns}(y), d) = \exists^{-1}(y, d)$
2. $\mathrm{Wlp}(\mathrm{Add}(r), d) = (\delta_r'(b) \mid \mu\vec{\mathfrak{x}}' = \vec{\mathfrak{F}}'(\vec{\mathfrak{x}}'))$ *where* $d = (b \mid \mu\vec{\mathfrak{x}} = \vec{\mathfrak{F}}(\vec{\mathfrak{x}}))$*, and* $\vec{\mathfrak{x}}'$*,* $\vec{\mathfrak{F}}'$ *are obtained by applying* δ_r' *to the main body and the equations (adapting the variable types).*
3. $\mathrm{Wlp}(\mathrm{Del}(l), d) = (\Delta(l) \Rightarrow \alpha_l'(d) \mid \mu\vec{\mathfrak{x}}' = \vec{\mathfrak{F}}'(\vec{\mathfrak{x}}'))$*, new equations analogous to* $\mathrm{Add}(r)$
4. $\mathrm{Wlp}(\mathrm{Sel}(x, c'), d) = \neg\exists(x, (c' \wedge \neg d))$
5. $\mathrm{Wlp}(P \cup Q, d) = \mathrm{Wlp}(P, d) \wedge \mathrm{Wlp}(Q, d)$
6. $\mathrm{Wlp}(P; Q, d) = \mathrm{Wlp}(p, \mathrm{Wlp}(Q, d))$

Wlp *is ordinarily applied to a condition with placeholders. If d is a μ-condition in case 1 or 4, then* Wlp *is applied to the main body.*

The main result for this chapter can now be stated:

Theorem 3 (Weakest Liberal Precondition). *For each rule ϱ, there is a transformation*[4] Wlp_ϱ *that maps μ-conditions to other μ-conditions and assigns, to each condition c such that $m' \models c$, another condition $\mathrm{Wlp}_\varrho(c)$ such that $m \models \mathrm{Wlp}_\varrho(c)$ whenever $(m, m', p) \in [\![\varrho]\!]$ and $\mathrm{Wlp}_\varrho(c)$ is the least condition, with respect to implication, with this property (weakest liberal precondition). That transformation is Definition 37.*

Proof. The proofs for $\mathrm{Sel}(x)$, $\mathrm{Sel}(x, c')$, $P \cup Q$ and $P; Q$ are exactly as in [Pen09], while correctness of the first step, $\mathrm{Uns}(y)$, is immediate from the semantics.

For deletion and addition step, we proceed by inductively comparing d to $\mathrm{Wlp}(\mathrm{Del}(l), d)$ resp. $\mathrm{Wlp}(\mathrm{Add}(r), d)$, which in turn requires inductively comparing conditions with

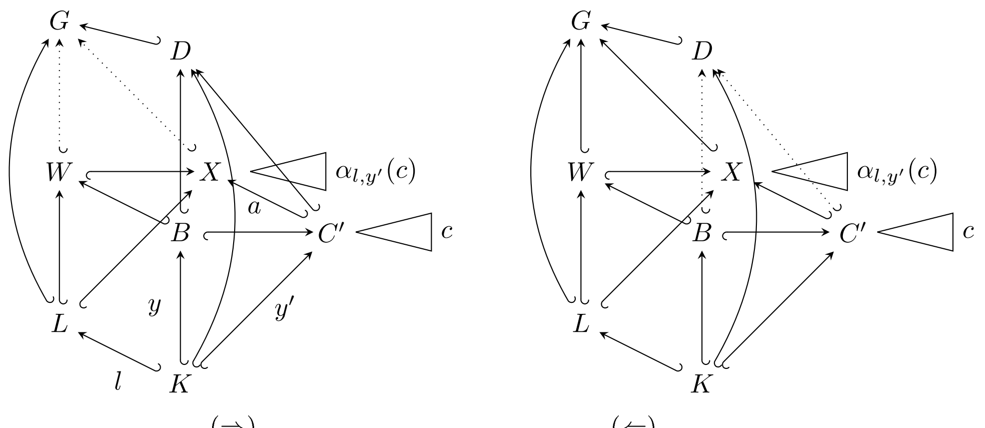
[4]Caution: wlp is the notion as defined in Definition 32 while Wlp is the construction.

placeholders that appear in the main body and right hand sides. The outer induction over $\mathbb{N}$ (see Remark 5) compares the least fixed points. This takes care of the case of variables. The induction hypothesis here states that the valuation at the current iteration satisfies the hypothesis. As the variables satisfy the system of equations, we show by induction over the nesting that the construction is correct for the right hand sides under any valuation satisfying the hypothesis.

For Boolean combinations, the induction step follows directly from the definition of satisfaction under a valuation (Definition 18) and from the definitions $\delta_{r,y}(\neg c) = \neg\delta_{r,y}(c)$ and $\delta_{r,y}(\bigwedge_{j\in J} c_j) = \bigwedge_{j\in J} \delta_{r,y}(c_j)$ (analogously for α').

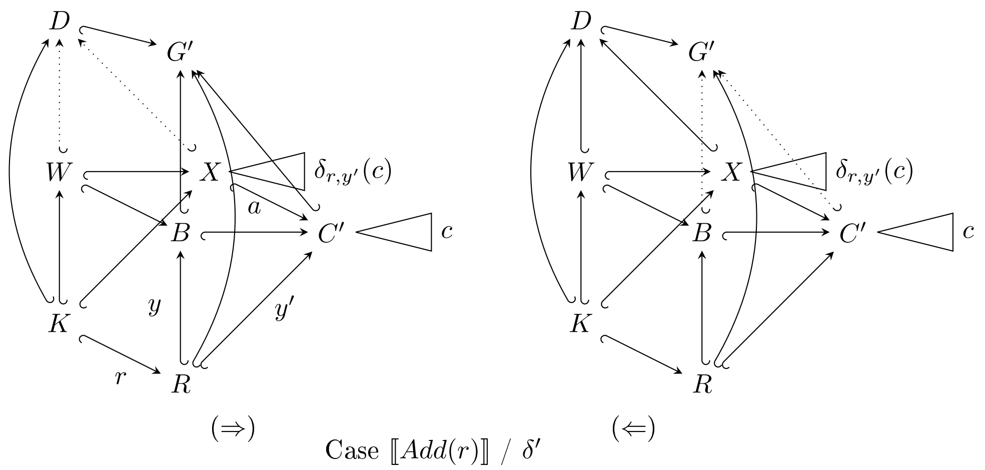
The interesting case of the induction over the nesting lies in comparing satisfaction of $\exists(a,\iota,c)$ to $\alpha_{l,y}(\exists(a,\iota,c))$, resp. $\delta_{r,y}(\exists(a,\iota,c))$. The goal is to obtain bi-implications in both cases: whenever $(m_{in}, m_{out}, p) \in [\![\mathrm{Del}(l)]\!]$, $m_{out} \models c \Leftrightarrow m_{in} \models \mathrm{Wlp}(\mathrm{Del}(l), c)$ and whenever $(m_{in}, m_{out}, p) \in [\![\mathrm{Add}(r)]\!]$, $m_{out} \models c \Leftrightarrow m_{in} \models \mathrm{Wlp}(\mathrm{Add}(r), c)$. We again decompose any subcondition $\exists(a,\iota,c)$ into a quantification of the form $\exists(a,c)$ and an unselection of the form $\exists(\iota,c)$ by Lemma 1. The bi-implications are shown in two parts $\Rightarrow$, $\Leftarrow$ each.

The diagrams below depict the situation in each case (deletion and addition). The lower halves of the diagrams (including the graphs W, X) are the diagrams from the constructions α', δ'. Dotted arrows represent the morphisms whose existence must be shown, solid arrows represent the morphisms from the assumptions. In all four cases, the induction hypothesis asserts correctness of the weakest precondition construction for the morphisms named $y' = a \circ y$ and $a' \circ h$ in Construction 9 and Construction 10 and the induction step concludes correctness at y and h. The definition is recursive over the finite nesting of the right hand side, so we assume that $\alpha_{l,y'}(c)$ resp. $\delta_{r,y'}(c)$ is already defined for the subcondition and the induction hypothesis holds there.



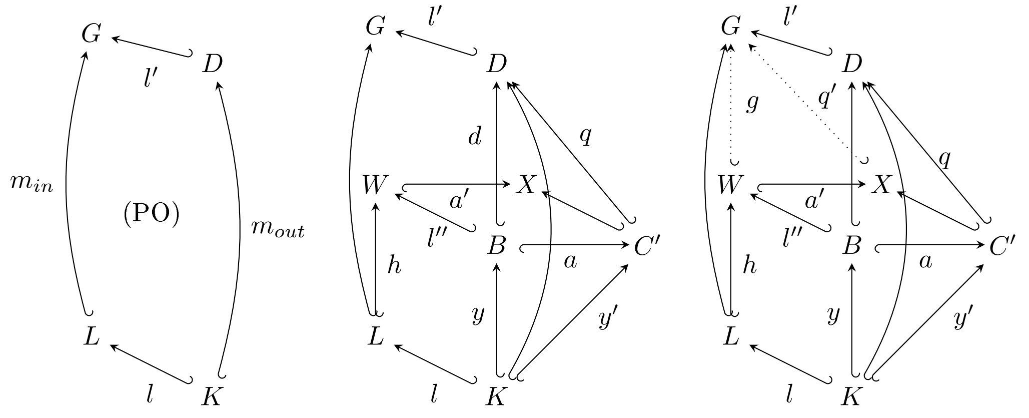
(⇒) (⇐)

Case $[\![Del(l)]\!]$ / α'



Case $[\![Add(r)]\!]$ / δ'

For $(m_{in}, m_{out}, l^{-1}) \in [\![\mathrm{Del}(l)]\!]$ with $m_{out} : K \hookrightarrow D$, $m_{in} : L \hookrightarrow G$, a pushout is given by the semantics. The pushout is depicted in the diagram below. Assume further that $m_{out} \models \exists(a, c)$ with $a : B \hookrightarrow C'$, and that partial shift has been applied, yielding the monomorphisms $y : K \hookrightarrow B$ and $y' : K \hookrightarrow C'$. The diagrams below show the starting situation and the induction step, detailed below.



Consider $\alpha_{l,y}(\exists(a, c))$, where the morphism $y : K \hookrightarrow B$ is obtained from the partial shift construction. Build the pushout over (l'', d) and compose it with the lower pushout square, which must be equal the outer pushout by the uniqueness of pushout composition. The morphism $g : W \hookrightarrow G$ obtained in the pushout and the unique commuting morphism $q' : X \hookrightarrow G$, together with the transformed subcondition $\alpha_{l,y'}(c)$, yield satisfaction of the condition $\alpha_{l,y}(\exists(a, c))$ by m_{in}.

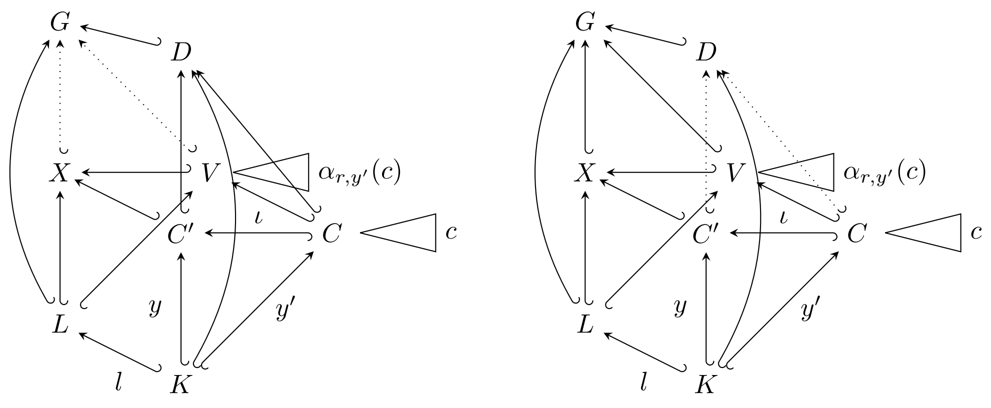
For $(m_{in}, m_{out}, l^{-1}) \in [\![\mathrm{Del}(l)]\!]$ with $m_{out} : K \hookrightarrow D$, $m_{in} : L \hookrightarrow G$ and $m_{in} \models \alpha_{l,y}(\exists(a, c))$ via g: construct a pullback l^* and g with object B', consider the universal morphism from K to B' and conclude that since a canonical isomorphism $B' \cong B$ exists by uniqueness of the pushout complement (since $h : L \hookrightarrow W$ is a monomorphism [EEPT06]) yielding a morphism $d : B \hookrightarrow D$ to complete the pushout square by the special PO-PB lemma.

For $(m_{in}, m_{out}, r) \in [\![\mathrm{Add}(r)]\!]$ with $m_{out} : R \hookrightarrow G'$, $m_{in} : K \hookrightarrow D$ and $m_{out} \models \exists(a, c)$ via g': pullback $D \hookrightarrow G'$ and $B \hookrightarrow G'$ with object W', then use universality to obtain the unique commuting morphism $K \hookrightarrow W'$, which yields a decomposition of the pushout square from the semantics, which by the special PO-PB lemma consists of pushouts and by uniqueness of $\mathcal{M}$-pushout complements implies $W \cong W'$.

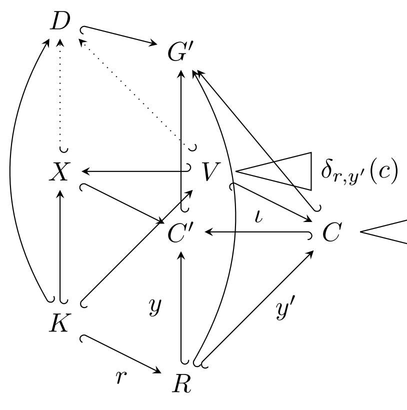
For $(m_{in}, m_{out}, r) \in [\![\mathrm{Add}(r)]\!]$ with $m_{out} : R \hookrightarrow G'$, $m_{in} : K \hookrightarrow D$ and $m_{in} \models \delta_{r,y}(\exists(a, c))$ via g, in the same way as the opposite direction of the $\mathrm{Del}(l)$ case.

In the case of $\mathrm{Del}(l)$, the condition for the pushout complement required by the semantics to exist is precisely $\Delta(l)$. In the case of $\mathrm{Add}(r)$, the construction of δ' asserts the existence of the pushout. From the induction hypothesis and universality of the morphisms constructed to complete the diagrams, the diagrams must commute and we conclude that $m_{out} \models c \Leftrightarrow m_{in} \models \mathrm{Wlp}(\mathrm{Del}(l), c)$, resp. $m_{out} \models c \Leftrightarrow m_{in} \models \mathrm{Wlp}(\mathrm{Add}(r), c)$ under the given circumstances.

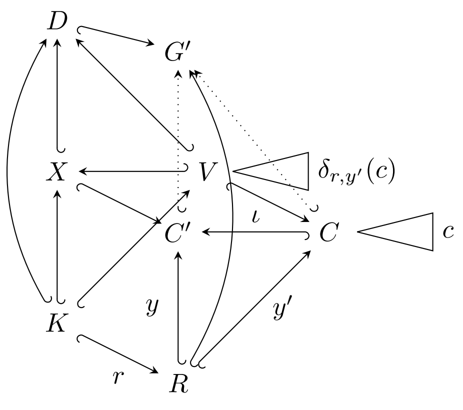
The unselection part is a straightforward adaptation of the proof above: morphisms compose unequivocally and the appropriate morphisms from the program interface to the graphs appearing in the right hand side (B, C'... resp. W, X... in the transformed condition) again exist, as required by the induction hypothesis at the higher nesting level. The proof mirrors the part for quantification:



($\Rightarrow$) Case $[\![Del(l)]\!]$ / α' ($\Leftarrow$)



($\Rightarrow$)



($\Leftarrow$)

Case $[\![Add(r)]\!]$ / δ'

The cases $(\alpha_{l,y}(\exists^{-1}(\iota, c)), \Rightarrow)$ and $(\delta_{r,y}(\exists^{-1}(\iota, c)), \Leftarrow)$ again require only the construction of pushouts and rely on the uniqueness of pushout composition in the same way as the corresponding cases for $\exists(a, c)$, while the remaining two cases are proved by constructing pullbacks and applying Lemma 16, as for $\exists(a, c)$. $\square$

5.1.3. Programs with Iteration

Programs with iteration are problematic because their weakest liberal precondition is neither computable (a graph program can simulate a Turing machine, then consider the wlp of halting with output "yes") nor in general a μ-condition ([Pen09] stated that for nested conditions, it is again a nested condition but meant *infinitary* nested conditions, which can indeed express any property of finite graphs but need not be equivalent to any finitary nested condition or even μ-condition!).

The technique to deal with the iteration P^* of a program P is to attempt to approximate the weakest liberal precondition by some precondition e implied by it, for which $\{e\}P\{e\}$ is correct (i.e. which is an invariant). If $c \Rightarrow e$ and $e \Rightarrow d$ can be established, then the specification is correct. If not, then the answer is unknown. If the answer is unknown, a counterexample search may yet yield a negative answer.

In programs with nested iteration operators, the method sketched here requires successive attempts at finding invariants and demands non-mechanical input for each. This can be regarded as a shortcoming of the Dijkstra method but it is impossible to avoid this issue entirely due to the undecidability of partial correctness.

5.1.4. An Example of a Weakest Liberal Precondition

For purposes of illustration, we construct a weakest liberal precondition of a μ-condition step by step. Figure 5.3 shows a graph program which matches a node with exactly one incoming and one outgoing edge and replaces this by a single edge.

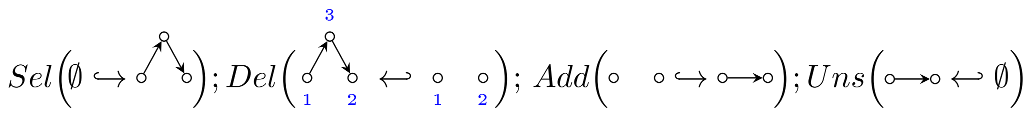


Figure 5.3.: The program $\varrho_{\text{contract}} = \text{Sel}_{\text{c}}; \text{Del}_{\text{c}}; \text{Add}_{\text{c}}; \text{Uns}_{\text{c}}$.

The effect of the rule $\varrho_{\text{contract}}$ is to contract paths, and it can be applied as long as no other edges are attached to the middle node. Figure 5.4 shows a μ-condition whose weakest liberal precondition we wish to compute. It is a typical example of a μ-condition, which evaluates to $\top$ on those graphs that are fully (directed-) connected, i.e. where any pair of nodes is linked by a directed path.

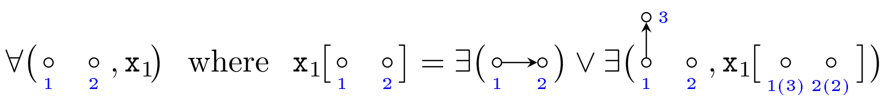


Figure 5.4.: A μ-condition $c_{5.4} = (b \mid l)$ expressing connectedness.

Our example shows a graph program (graph transformation rule), a condition and the weakest precondition with respect to the unselection step of the rule. In $\text{Wlp}(\text{Uns}_{\text{c}}, c_{5.4})$ (Figure 5.5), the nodes under the universal quantifier in Figure 5.5 are not the same as those of the existential one, as these have been unselected: the type of the subcondition $\forall(\underset{1}{\circ}\ \ \underset{2}{\circ}, \mathtt{x}_1)$ is $\emptyset$. The existential quantifier was introduced by the weakest precondition construction for unselection.

The weakest precondition with respect to the unselection is shown in Figure 5.5, before application of the partial shift.

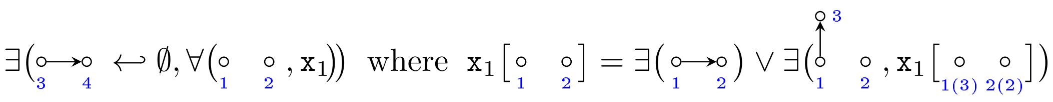


Figure 5.5.: $\text{Wlp}(\text{Uns}_{\text{c}}, c_{5.4})$.

In Figure 5.7 and Figure 5.8, partial shift has been applied to the condition $\text{Wlp}(\text{Uns}_{\text{c}}, c_{5.4})$

node/edge decoration	meaning
●⟶●	items (nodes and edges) selected for $Wlp(Uns(y), c)$
◎⋯→◎	items to be deleted to obtain $Wlp(Add(r), c)$
∘⟹∘	items to be added to obtain $Wlp(Del(l), c)$

Figure 5.6.: Legend for the partial shift and weakest precondition example.

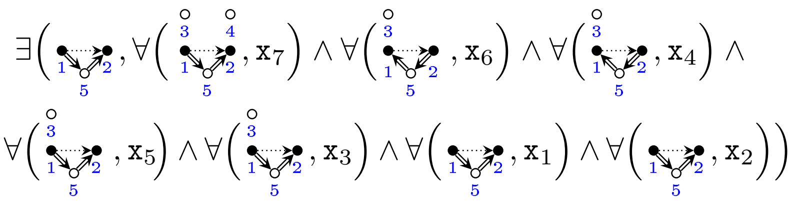


Figure 5.7.: Construction of $\mathrm{Wlp}(\mathrm{Del_c}; \mathrm{Add_c}; \mathrm{Uns_c}, c_{5.4})$.

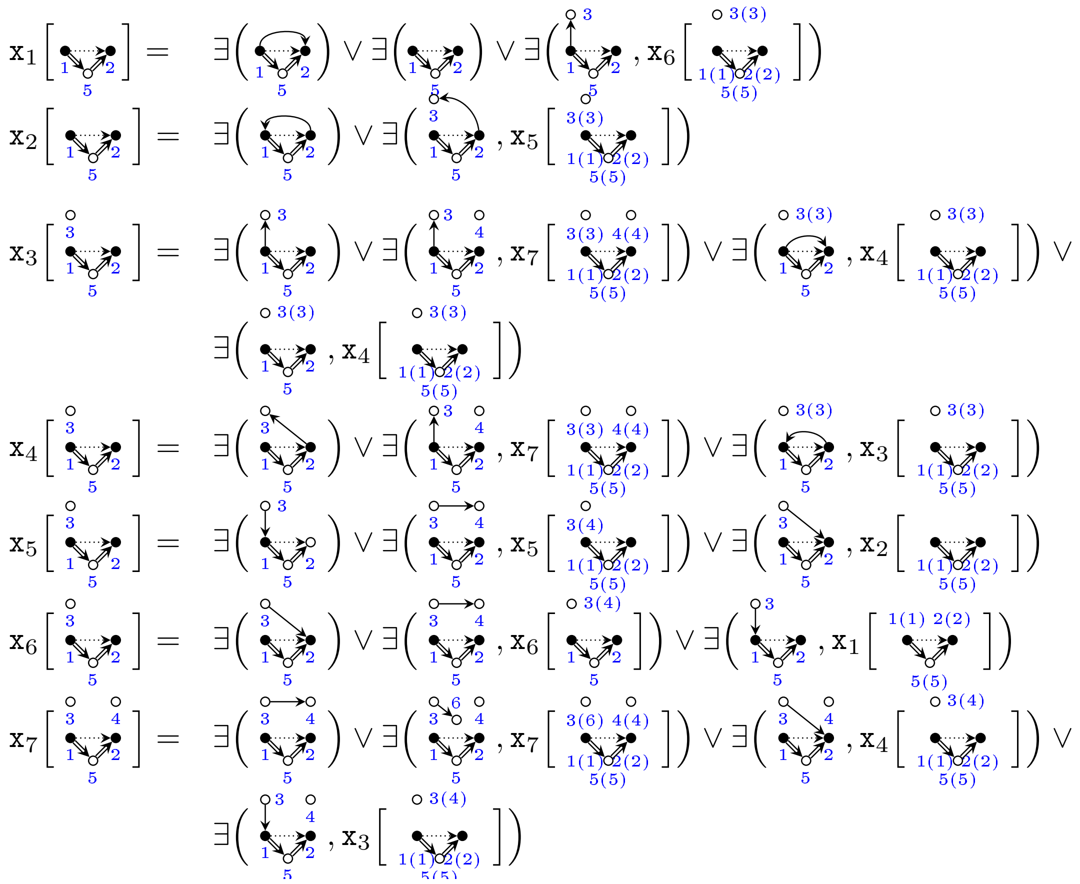


Figure 5.8.: Construction of $\mathrm{Wlp}(\mathrm{Del_c}; \mathrm{Add_c}; \mathrm{Uns_c}, c_{5.4}$: equations for the variables.

of Figure 5.5, and the modifications the condition undergoes in the computation of the weakest precondition with respect to $\mathrm{Add_c}$ and $\mathrm{Del_c}$ (namely, application of δ'_r and α'_l to the main body) are highlighted in various colours (see Figure 5.6 for a legend). Construction 7 has yielded a new list of variables[5], $\mathtt{x}_1, ..., \mathtt{x}_7$, the corresponding equations are shown in Figure 5.8, in abbreviated notation: variable types are suppressed in

[5]Although this particular μ-condition $c_{5.4}$ had only one variable, a partial shift usually yields one with multiple variables.

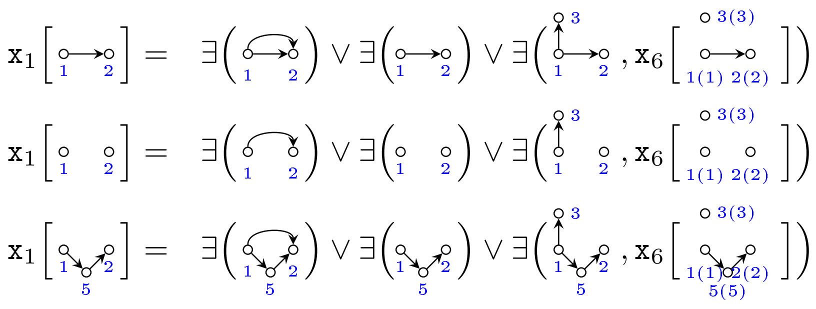


Figure 5.9.: The first equation from Figure 5.8, all three steps in detail.

subconditions $\exists(a, \iota, \mathtt{x}_i)$ if the mapping ι from the type to the target of a is the identity. No other simplifications were applied.

In the computation of the weakest precondition with respect to $\mathrm{Add_c}$ and $\mathrm{Del_c}$, the transformations δ'_r and α'_l are not only applied to the main body, but also to the right hand sides of all the variables obtained via partial shift. We have highlighted the type of the main body of $\mathrm{Wlp}(\mathrm{Uns_c}, c_{5.4})$ throughout Figure 5.7. When following the construction through the nesting levels, please keep in mind that one may sometimes choose among isomorphic pushout objects and furthermore the numbers of new nodes are arbitrary. However, the nodes 1, 2 and (as created by the transformation α') 5 are never "unselected" and therefore present in every type occurring in the weakest preconditions, similarly for the edges (not numbered because their mapping is unambiguous in the example).

Edges drawn as dotted lines are deleted to compute $\mathrm{Wlp}(\mathrm{Add_c}, \mathrm{Wlp}(\mathrm{Uns_c}, c_{5.4}))$ as per Construction 9; the hollow edges and nodes are those that are added in the transformation to compute $\mathrm{Wlp}(\mathrm{Del_c}, \mathrm{Wlp}(\mathrm{Add_c}, \mathrm{Wlp}(\mathrm{Uns_c}, c_{5.4})))$ as per Construction 10. The actual weakest precondition with respect to $\mathrm{Del_c}$ is obtained by adjoining $\Delta(l) \Rightarrow$ to the main body after applying said transformation ($\Delta(l)$ omitted in Figure 5.7 as it is straightforward to compute); the filled nodes are neither deleted nor added (if there were any preserved edges in the examples, these would be drawn as unbroken lines; deleted nodes would be drawn as double circled nodes). A universal quantifier with $\emptyset \hookrightarrow L$ completes the weakest precondition with respect to the rule, as for nested conditions.

Figure 5.9 shows in detail, rather than in the condensed form of Figure 5.8, what happens to the first equation of $\mathrm{Wlp}(\mathrm{Uns_c}, c_{5.4})$ when the transformations Definition 37 are applied. In this example, the deletion step $\mathrm{Del_c}$ deletes only edges, so the check for the existence of a pushout complement (first case of Construction 9) never fails.

5.2. Correctness Relative to Recursively Nested Conditions

We have shown how the weakest liberal precondition construction for nested conditions carries over to μ-conditions. The next task, for which we offer a partial solution in this section, is to develop methods for the deduction of correctness relative to μ-conditions by extending Pennemann's proof calculus $\mathcal{K}$. We recall that $\mathcal{K}$ works on nested conditions which are in conjunctive normal form at each nesting level; it features rules called (supporting) *lift* and (partial) *resolve*: the former serve to lift a member of a conjunction to a deeper nesting level, conjoining its *shift* to the subcondition of an existential quantifier, while the latter seek to derive contradictions. The rule *descend* allows a member $\exists(a, \bot \wedge c)$ of a clause to be replaced by $\bot$.

5.2.1. A Proof Calculus

In a proof calculus we may adopt all rules of $\mathcal{K}$, as their soundness is not impacted by working with recursively nested conditions. However dealing with recursive definitions requires an extension.

The strategy used in Pennemann's ProCon theorem prover [Pen09] (converting the condition to be refuted to a conjunctive normal form at each nesting level and deriving contradictions at the innermost nesting levels) is not applicable in the presence of recursion. In our calculus $\mathcal{K}_\mu$, no such normal form is required. Instead, we add all Boolean manipulations as rules, and propose an induction rule to deal with situations involving fixed points. This proves to be sufficient to handle all situations encountered in the examples.

We employ a sequent notation: the inference rules manipulate *sequents* $\mathcal{F} : \Gamma \vdash \Delta$, where $\mathcal{F}$ is a system of equations, Γ and Δ are sets of μ-condition bodies, with the intended meaning that the disjunction of Δ can be deduced from the conjunction of Γ where the least fixed point solution of $\mathcal{F}$ is substituted for the variables. Additionally, variables are annotated with an arithmetic expression over natural numbers and identifiers $n_1, ...,$ which serve the important purpose of ensuring well-foundedness in the recursive refutation rule. Note that it is always sound to increment an annotation in an inference because by monotonicity, $\mathcal{F}_i^n(\vec{\bot}) \Rightarrow \mathcal{F}_i^{n+1}(\vec{\bot})$. The *context* rule allows access to any subcondition. *Ctx* is a μ-condition syntactically monotonic (or antitonic) in a distinguished open variable $\mathtt{x}$ of same type as c, c' and $\mathcal{F}$ is arbitrary:

$$\frac{\mathcal{F}' : c \vdash c' \quad (\text{resp. } c' \vdash c)}{\mathcal{F} \uplus \mathcal{F}' : Ctx[\mathtt{x}/c] \vdash Ctx[\mathtt{x}/c'] \qquad \text{if } Ctx \text{ is monotonic (antitonic) in } \mathtt{x}} \quad (\textsc{Ctx})$$

Note that variables used in x and x' may have to be renamed in order not to conflict with those in *Ctx*, hence we write $\uplus$. Soundness is then immediate. Another auxiliary rule allows unrolling $\mathtt{x}_i$ to the i-th component $\mathcal{F}_i(\vec{\mathtt{x}})$. When used inside a nested context via Rule Ctx, it replaces a specific occurrence of a variable by its right hand side:

$$\frac{\mathcal{F} : \Gamma \vdash \Delta, \mathtt{x}_i^{(n)}}{\mathcal{F} : \Gamma \vdash \Delta, \mathcal{F}_i(\vec{\mathtt{x}}^{(n-1)}) \qquad \mathcal{F}_i(\vec{\mathtt{x}}) \text{ is the right hand side for } \mathtt{x}_i \text{ in } \mathcal{F}} \quad (\textsc{Unroll}_1)$$

In Rule UNROLL$_1$, the annotations of the variables in the new expression are decremented: when $f \models \mathcal{F}_i^{(n)}(\vec{\bot})$, then it satisfies $\mathtt{x}_i$ in the *next* step of the fixed point iteration (cf. Theorem 1), hence in the conclusion the variables used in the right hand side are all annotated with $(n-1)$. This rule is sound by the fixed point semantics.

The annotations play a crucial role in the proof calculus, because they allow the absurd proposition $\bot$ to be deduced *inductively* by proving that absurdity follows from the premisses at each finite stage of the fixed point iteration. The following recursive refutation rule schema works by exploiting the annotations ($\vec{n'} < \vec{n}$: whatever numbers are substituted for the identifiers of $\vec{n'}$ and $\vec{n}$, the comparison must hold):

$$\frac{\forall i \in I . \mathcal{H}_i(\vec{\mathtt{x}}^{(\vec{n})}) \vdash \vec{\mathcal{G}}(\vec{\mathcal{H}}(\vec{\mathtt{x}}^{(\vec{n'})})) \qquad \vec{\mathcal{G}}(\vec{\bot}) = \vec{\bot}}{\bigvee_{i \in I} . \mathcal{H}_i(\vec{\mathtt{x}}) = \bot \qquad\qquad \vec{n'} < \vec{n}; \quad \vec{\mathcal{G}} \text{ monotonic.}} \qquad (\text{EMPTY})$$

If one can find suitable $I, \vec{\mathcal{H}}, \vec{\mathcal{G}}$, then by well-foundedness of $<$, induction over $\vec{n}$ shows that at any level of the fixed point iteration, the expressions $\mathcal{H}_i(\vec{\mathtt{x}})$ imply absurdity, therefore substituting the least fixed points $\hat{\vec{\mathtt{x}}}$ of the variables must also result in $\mathcal{H}_i(\vec{\mathtt{x}})[\vec{\mathtt{x}}/\hat{\vec{\mathtt{x}}}]$ evaluating to $\bot$:

Lemma 17 (Soundness of Rule Empty). *Rule* EMPTY *is sound.*

Proof. By induction over the well-founded order $< \subseteq \mathbb{N}^{\|I\|} \times \mathbb{N}^{\|I\|}$, at $\vec{n} = \vec{0}$ all variables must evaluate to $\bot$ according to Definition 22. Induction step: assuming that the hypothesis $\vec{\mathcal{H}}(\vec{\mathtt{x}}^{(\vec{n'})}) \equiv \vec{\bot}$ holds at every $\vec{n'} < \vec{n}$, $\vec{\mathcal{H}}(\vec{\mathtt{x}}^{(\vec{n'})}) \equiv \vec{\bot}$ and $\vec{\mathcal{G}}(\vec{\mathcal{H}}(\vec{\mathtt{x}}^{(\vec{n'})})) \equiv \vec{\bot}$, therefore our induction hypothesis also holds at $\vec{n}$. By Proposition 1, all components of $\vec{\mathcal{H}}$ are $\bot$ in the least fixed point. □

A useful instantiation is based on defining conjuncts $\mathcal{H}_{i,j} := \exists^{-1}(\iota_i, \mathtt{x}_i) \wedge \neg\exists^{-1}(\iota_j, \mathtt{y}_j)$ where $\mathtt{x}_i$ and $\mathtt{x}_j$ range over the variables of two μ-conditions whose main bodies have been combined as $b \wedge \neg b'$ (this situation is frequently encountered when attempting to prove that a specified precondition implies a weakest precondition in the Dijkstra approach). The goal is to express the $\mathcal{H}_{i,j}$ in terms of (annotated versions of) each other and then to apply Rule EMPTY to deduce that in the least fixed point, the chosen variable combinations $\mathcal{H}_{i,j}(\vec{x})$ are unsatisfiable.

Several details require attention: Boolean operations must be extended to μ-conditions, which entails variable renaming and union of the systems of equations; rules for exploiting logical equivalences between different Boolean combinations are needed to rewrite conditions into a form suitable for the application of the rules of $\mathcal{K}$ ([Pen09] instead puts each Boolean combination appearing as a subcondition into conjunctive normal form prior to the application of rules). Proof trees in our sequent-style calculus $\mathcal{K}_\mu$ start with instances of the *axiom* ($A \vdash A$ with no antecedents), and make use of all the classical sequent rules [Gen35a] not involving quantifiers. The sequent rules are upgraded to sequents with equations $\mathcal{F}$, as introduced above. Each antecedent may come with equations, but they must not conflict (renaming variables as needed).

$$\frac{\exists(a,c)\wedge d}{\exists(a,c\wedge\exists^{-1}(a,d))}$$

(Supporting)Lift

$$\frac{\neg\exists(a)\qquad\exists(b,d)}{\neg\exists(m^*)}$$

If $\exists m\in\mathcal{M}$, $m\circ b=a$ and (m^*,b^*) is $\mathcal{M}$-pushout complement of (b,m), $d\not\equiv\bot$
PartialResolve

$\mathcal{K}$ (adapted)

$$\frac{\exists(a\circ a',c)}{\exists(a,\exists(a',c))},\quad\frac{\exists(a,\iota\circ\iota',c)}{\exists(a,\iota',\exists^{-1}(\iota,c))},\quad\frac{\exists(id,id,c)}{c},\quad\frac{\exists^{-1}(\iota,c)}{A_\iota(c)},\quad\frac{\exists(a,c)}{r_a(c)}$$

Morphism manipulation rules

$$\frac{\Gamma\vdash\Delta}{D,\Gamma\vdash\Delta}$$

Thinning

$$\frac{D,D,\Gamma\vdash\Delta}{D,\Gamma\vdash\Delta}$$

Contraction

$$\frac{\Delta,D,E,\Gamma\vdash\Theta}{\Delta,E,D,\Gamma\vdash\Theta}$$

Interchange
(all similarly on succedent)

$$\frac{\Gamma\vdash\Theta,D\qquad D,\Delta,\vdash\Lambda}{\Gamma,\Delta\vdash\Theta,\Lambda}$$

Cut

Structural rules

$$\frac{\Gamma\vdash\Theta,A\qquad\Gamma\vdash\Theta,B}{\Gamma\vdash\Theta,A\wedge B}$$

UES

$$\frac{A,\Gamma\vdash\Theta}{A\wedge B,\Gamma\vdash\Theta}$$

UEA

$$\frac{A,\Gamma\vdash\Theta\qquad B,\Gamma\vdash\Theta}{A\vee B,\Gamma\vdash\Theta}$$

OEA

$$\frac{\Gamma\vdash\Theta,A}{\Gamma\vdash\Theta,A\vee B}$$

OES

Logical rules

Figure 5.10.: Resolution-Like ($\mathcal{K}$), Morphism-manipulation and Classical rules.

As well as the major rules presented above, we adapt rules from $\mathcal{K}$ (see box): the partial resolve rule is unchanged, the (supporting) lift rules without automatic application of shift merely make use of unselection. We also use the classical rules for Boolean logic [Gen35a], structural rules for morphism decomposition and removal of trivial nesting ($\frac{\exists(a\circ a',c)}{\exists(a,\exists(a',c))}$, $\frac{\exists(a,\iota\circ\iota',c)}{\exists(a,\iota',\exists^{-1}(\iota,c))}$ and vice versa, $\frac{\exists(id,id,c)}{c}$) (all of these are upgraded to operate on a single condition body on the right side of a sequent). The other rules from $\mathcal{K}$ are adapted: the descent rule $\frac{\exists(a,\bot\wedge c)}{\bot}$ is replaced by a more versatile absorption rule Absorb $\frac{\exists(a,c)}{r_a(c)}$ (mirroring Lemma 3 except that a need not be an isomorphism, r_a is defined as in Definition 23); a ι-removal rule ($\frac{\exists^{-1}(\iota,c)}{A_\iota(c)}$) which is correct by Lemma 13.

The boxes of Figure 5.10 and Figure 5.11 are a synopsis of the rules. Greek capital letters stand for lists of expressions and Latin ones for formulae (here: graph conditions). The rules of the left-hand boxes are applied to a condition on the right hand side of a sequent. The rules of $\mathcal{K}$ have been transmogrified into the new format, which means they no longer need to operate on clauses; furthermore, Lift and SupportingLift are merged into a rule that does not automatically apply A. Thanks to unselection, the application of A is now a separate rule Shift. The Descent rule is no longer necessary because a conjunction with $\bot$ can be transformed to $\bot$ via classical logical rules and Absorb may then remove the quantifier.

$$\frac{\mathcal{F}' : c \vdash c' \quad \text{(resp. } c' \vdash c\text{)}}{\mathcal{F} \uplus \mathcal{F}' : Ctx[\mathtt{x}/c] \vdash Ctx[\mathtt{x}/c'] \qquad \text{if } Ctx \text{ is monotonic (antitonic) in } \mathtt{x}} \quad (\textsc{Ctx})$$

$$\frac{\mathcal{F} : \Gamma \vdash \Delta, \mathtt{x}_i^{(n)}}{\mathcal{F} : \Gamma \vdash \Delta, \mathcal{F}_i(\vec{\mathtt{x}}^{(n-1)}) \qquad \mathcal{F}_i(\vec{\mathtt{x}}) \text{ is the right hand side for } \mathtt{x}_i \text{ in } \mathcal{F}} \quad (\textsc{Unroll}_1)$$

$$\frac{\forall i \in I. \mathcal{H}_i(\vec{\mathtt{x}}^{(\vec{n})}) \vdash \vec{\mathcal{G}}(\vec{\mathcal{H}}(\vec{\mathtt{x}}^{(\vec{n'})})) \qquad \vec{\mathcal{G}}(\vec{\bot}) = \vec{\bot}}{\bigvee_{i \in I} . \mathcal{H}_i(\vec{\mathtt{x}}) = \bot \qquad \vec{n'} < \vec{n};\ \vec{\mathcal{G}} \text{ monotonic.}} \quad (\textsc{Empty})$$

Figure 5.11.: Rule schemata for handling recursion (synopsis).

From amongst the logical rules of the classical sequent calculus, the rules for quantification are not adopted as we do not have logical variables in graph conditions. To access inner nesting levels, we use Ctx instead.

Theorem 4 (Soundness of $\mathcal{K}_\mu$).
The calculus $\mathcal{K}_\mu :=$ rules of Figure 5.10 and Figure 5.11 is sound.

Proof. The soundness of the $\mathcal{K}$ rules has been established in [Pen09], the supplementary rules have been established in the text above. The classical sequent rules without quantification rely on propositional logic only. □

In conclusion, the elements of nesting, Boolean combinations and recursion are handled by the corresponding rules: $\mathcal{K}$ and morphism manipulation rules for nesting, classical sequent rules for Boolean combinations and Empty, Unroll$_1$, Ctx for recursion.

5.2.2. An Example of a Proof

For this subsection, we have opted for a minimalistic first example without the blowup from the weakest liberal precondition in Subsection 5.1.4. The example (Figure 5.12) uses a minimal number of variables to show the calculus $\mathcal{K}_\mu$ and its inductive refutation rule at work. We examine the μ-condition $\mathtt{x}_1 \wedge \neg\mathtt{x}_2$, whose main body has type $[\underset{1}{\circ}\ \ \underset{2}{\circ}]$. Consider the following system $\mathcal{F}$:

Example 24 (System of equations used in the example proof)**.**

$$\mathtt{x}_1[\underset{1}{\circ}\ \ \underset{2}{\circ}] \;=\; \exists(\underset{1}{\circ}\!\longrightarrow\!\underset{2}{\circ}) \vee \exists(\overset{\overset{\circ\,3}{\uparrow}}{\underset{1}{\circ}}\ \ \underset{2}{\circ}, \mathtt{x}_1[\underset{1(3)}{\circ}\ \ \underset{2(2)}{\circ}])$$

$$\mathtt{x}_2[\underset{1}{\circ}\ \ \underset{2}{\circ}] \;=\; \exists(\underset{1}{\circ}\!\longrightarrow\!\underset{2}{\circ}) \vee \exists(\overset{\overset{\circ\,3}{\uparrow}}{\underset{1}{\circ}}\ \ \underset{2}{\circ}, \mathtt{x}_2[\underset{1(3)}{\circ}\ \ \underset{2(2)}{\circ}])$$

While the equations are syntactically identical up to variable renaming, this is not exploited by $\mathcal{K}_\mu$, hence the proof (Figure 5.12) is not a one-liner: it starts by defining a suitable list of auxiliary conditions $\vec{\mathcal{H}}$ (in this case actually a single one, which we name $\mathcal{H}_{1,2}$[6]), unrolling both variables once (1), then uses distributivity of conjunction over disjunction (derivable as a logical rule) to resolve the base case of the right hand side of $\mathtt{x}_1$ (2), then shifts the other branch of $\mathtt{x}_2$ over the corresponding branch of $\mathtt{x}_1$. In a lift and shift step (3), a conjunction of two subconditions is obtained (depending on whether the nodes 3 are identified). In step (4), one of these is dropped and the other is used to obtain $\mathtt{x}_1 \wedge \neg\mathtt{x}_2$, with lower annotations, as a subcondition. Finally we show that the context of this subcondition (monotonic by virtue of being syntactically positive) has least fixed point $\bot$, and apply Rule Empty.

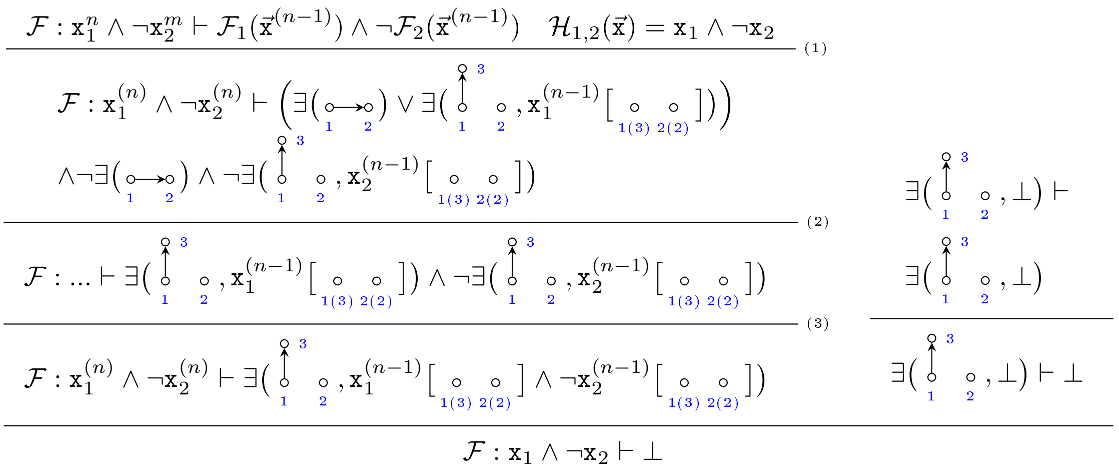

$\mathcal{F} : \mathtt{x}_1^n \wedge \neg\mathtt{x}_2^m \vdash \mathcal{F}_1(\vec{\mathtt{x}}^{(n-1)}) \wedge \neg\mathcal{F}_2(\vec{\mathtt{x}}^{(n-1)})$
$\mathcal{H}_{1,2}(\vec{\mathtt{x}}) = \mathtt{x}_1 \wedge \neg\mathtt{x}_2$
(1)
$\mathcal{F} : \mathtt{x}_1^{(n)} \wedge \neg\mathtt{x}_2^{(n)} \vdash$
(2)
$\mathcal{F} : \ldots \vdash$
(3)
$\mathcal{F} : \mathtt{x}_1^{(n)} \wedge \neg\mathtt{x}_2^{(n)} \vdash$
$\mathcal{F} : \mathtt{x}_1 \wedge \neg\mathtt{x}_2 \vdash \bot$

Figure 5.12.: Deducing a contradiction from $\mathtt{x}_1 \wedge \neg\mathtt{x}_2$ under the system of equations $\mathcal{F}$. Multiple steps have been contracted into single inference lines for brevity.

5.3. Conclusion and Outlook

In this chapter, several results about recursively nested conditions were achieved: a weakest liberal precondition transformation (Theorem 3), a sound proof calculus (Theorem 4). Correctness relative to μ-conditions was defined and discussed and appears to be a fruitful ground for further investigations.

A summary overview of graph conditions for non-local properties is attempted below.

[6] Note that a larger example would likely have required more than one branch to handle each conjunct.

reference	[Pen09]	(here)	[Rad16]	[PP14]
conditions	Nested	μ-	HR*	M
wlp	yes	yes	yes	yes
proof calculus	complete	yes	future work	Hoare logic
theorem prover	yes	future work		

A proof calculus is presented in [PP14] but completeness of a proof calculus has only recently been obtained by Lambers and Orejas [LO14] for nested conditions and remains to be researched for the other approaches.

The different formalisms can be said to occupy different niches. HR* conditions are known to properly contain the monadic second-order definable properties [Rad13] and nested conditions are a special case of each of the other three, and we have provided some results towards separating μ-conditions from M- or HR* conditions in Section 3.4. M-conditions have the advantage of being closely related to MSO logic, just as μ-conditions are closely related to fixed point logic. μ-Conditions have the advantage of a conceptually simple definition and a very graph-condition-like weakest precondition transformation.

As the examples show, our weakest precondition calculus (a weakest precondition calculus now also exists for HR* conditions [Rad16] and is readily available by logical means in the M-conditions formalism [PP14]) produces unwieldy expressions due to partial shift, each time an unselection is encountered in the computation of a weakest precondition. A related blowup is inherited from the weakest precondition calculus of [Pen09], which makes use of the shift construction. In our case however, this construction is a proof rule and can be applied selectively. We can heuristically simplify the expressions and empirically found that many frequently occurring cases can in principle be resolved automatically. Many of these points will be discussed again on practical examples in Chapter 7.

It is an issue with *any* kind of graph conditions that the weakest precondition calculus cannot be said to be *complete*, because that would mean that any graph program and postcondition asserted as a μ-Condition has a corresponding expression for the precondition. This cannot be, as any recursively enumerable graph language would have to be expressible as a μ-condition (with finite index sets at disjunctions!), whereas satisfaction for μ-conditions by a graph is decidable, indeed in polynomial time. It should therefore be stressed that we did not aim for (relative) completeness in the sense of [Coo78], which is defined for Hoare logics as completeness under the assumption that the implication problem of the assertion language could be solved. Because of the lack of expressiveness of the assertion language – in contrast to the usual approach, which uses first-order logic with arithmetic and hence has the required expressiveness, a graph condition can only mention nodes and edges that are actually present in a state and is limited to certain expressible properties for preconditions. Therefore the assertion language does not meet the criterion of expressiveness, and neither completeness nor relative completeness is attainable in our setting without supplementary constructions. This limitation is shared with all graph condition formalisms on the grounds that all properties that can be expressed with (finite) graph conditions are decidable (in the case of μ-conditions, in polynomial time).

Concerning the lack of expressiveness, we believe that it makes sense to *embed* our work in a more general Hoare logic to deal with the graph-like properties that *can* be expressed. Also, an extension of μ-conditions by *temporal operators parameterised on programs might* provide a way to attain completeness, as such a formalism could express weakest preconditions symbolically, similarly to dynamic logic [HKT00].

It indeed appears that μ-conditions might readily generalise to temporal properties, even with the option to nest temporal operators inside quantifiers, which would allow properties such as the preservation of a specific node to be expressed (but require further proof rules). This could be achieved by introducing a *next* operator parameterised on atomic subprograms (the basic steps of Definition 3) and since in the semantics of these program steps the relationship between the interfaces is deterministic, this would again confer an unambiguous *type* to such an expression and make it suitable for use as a subcondition. Whether this offers any new insights remains to be seen.

Future work must include more work on the power of the proof calculus because it would be important to know the limitations of our method.

Future work will also include tool support with special attention to semi-automated reasoning, based on the reasoning engine ENFORCE implemented in [Pen09]. To extend the weakest liberal precondition construction to programs with iteration, one would have to provide, or have the prover attempt to determine, an invariant, as in the original work of Pennemann. To obtain termination proofs for *total* rather than partial correctness, one may proceed as in [Pos13] and prove termination variants.

Eventually, the formalism should be upgraded to express algebraic operations on attributes (labels), extending our work to a practical verification method that separates the graph specific concerns from other aspects and allows proofs of properties that depend on both, for example involving data structures whose elements should remain ordered. This proposed extension should be easy to implement, as a body of research is already available on attribution concepts for graph transformations.

Finally, the limitations imposed by undecidability prompt the search for classes of programs and conditions where correctness is decidable.

5.4. Bibliographic Notes

In the following, bibliographic notes on the various aspects treated in this chapter are listed. These concern program correctness (the general notion from software verification), proof calculi and related constructions for non-local graph conditions, other subjects in the verification of graph programs.

The original reference for Dijkstra's predicate transformer approach is [Dij76], Hoare's axiomatic approach to program correctness is [Hoa83]. In Hoare logic, much attention has always been given to completeness [Coo78]. On the subject of proving correctness in Hoare logic (axiomatic semantics), we refer to the monograph of Apt and Olderog [AO97]. A Hoare calculus for the verification of graph programs is found in Poskitt and Plump [PP14], which was already mentioned in Chapter 3. Partial correctness can be complemented with termination analyses such as [BKZ14], or well-founded termination rules as in [Pos13], to yield total correctness.

An approach related to ours, based on Hoare logic, is separation logic. Separation logic appears in a growing number of publications in the context of software verification. It was devised because, in the word of Reynolds [Rey02], *"in logics, sharing is the default and non-sharing tedious to declare and because methods for reasoning about shared mutable data structures weren't practical"*. Separation logic is not a graph condition formalism and has a different scope. Our definitions can in principle be used for many kinds of graph-like structures. However, a comparison or even a combination would certainly be interesting when our work is applied to software verification.

It has indeed been demonstrated that graph transformation systems have the potential to be a practical tool in the verification of operations on data structures [BCE$^+$05]. There is an ongoing effort to base the verification of concurrent object-oriented programs on shape analysis (for example Yahav and Sagiv [YS10]), which has been related to graph transformation based methods in the work Rensink [RD06] and Zambon and Rensink [ZR11, RZ12]. Even more recently Heußner et al., Corrodi et al. [HPCM15, CHP16] have published work on the verification of concurrent object-oriented programs using graph programs, using the GROOVE model checker [KR06]. For a treatment of data values, attributed graphs can be used. The monograph of Ehrig et al. [EEPT06] offers a treatment of attributed graph transformations, which is also an active area of research. On the specification side, graph conditions should at least be able to express constraints on attributes, such as the E-conditions of Orejas [Ore11]. Inference rules must account for the attributes as well.

Apart from the proof-based approach to verification of graph transformation systems, another notion of correctness is via abstract model checking of the transition systems. The state space of the graph transformation system, which is in general infinite, is condensed into a finite number of abstract states; specifications are given as temporal logic formulas whose propositions are graph conditions. References are Gadducci et al., Baldan et al., König et al., Rensink et al. [GHK98, BKK03, KK06, RD06]. These could in principle be adapted to graph programs: in Section 6.1, graph programs will be given an operational semantics, which is a prerequisite for applying this method.

Shape analysis is an approach to software verification that works with graph abstractions of pointer structures [SRW99, LZC13, DN03]. The latter work uses reduction by graph rewriting systems to define valid pointer structures. We envisage shape analysis as a possible future application of our work. The link between the abstract model checking mentioned above, and deductive methods such as ours, could be provided by abstraction refinement [CGJ$^+$03], which has to our knowledge not yet been researched for graph programs.

The PhD thesis of Radke [Rad16] contains new methods and results for HR* conditions, such as an integration into rules, which is based on a shift construction. Poskitt and Plump [PP14] have presented a weakest precondition calculus for another extension of nested conditions (monadic second-order conditions) and demonstrated its use in a Hoare logic. The method is arguably closer to reasoning directly in a logic and less graph condition like, but seems successful at solving some of the same problems in a different way. Strecker et al. [Str08, PST13] have performed verification of graph transformation system within general-purpose theorem proving environments, with positive path conditions. Dyck and Giese [DG15] automatically check certain kinds of inductive invariants of graph transformation systems.

As for the proof calculus presented in this chapter, the original purpose of Gentzen's sequent calculus [Gen35a, Gen35b] was to provide deep results in proof theory, the famous Hauptsatz. Our usage is quite humble in comparison and serves mainly to formulate Rule Empty as an inference rule.

The soundness of $\mathcal{K}$ in the context of nested conditions has been established in the publications introducing them; recently a tableaux based completeness proof of $\mathcal{K}$ has been found by Lambers and Orejas [LO14]. The resolution-style proof rules of $\mathcal{K}$ are clearly sound for μ-conditions as well. Navarro, Pino, Orejas and Lambers are developing a formalism [MN16] supporting (specifically) path conditions and proofs with nested tableaux, which does not seem to be far removed from our work. It will certainly prove worthwhile to compare their proof calculus, ours and those for temporal logics (cf. e.g. Studer [Stu08]). Sound and complete tableau-like proof systems are also known for the propositional μ calculus [Cle90] and propositional linear temporal logic [GHL$^+$07]. These logics make use of recursive definitions and tableaux may have loops. There is an undeniable structural resemblance between tableaux with loops and proofs using Rule Empty.

6. Correctness under Adverse Conditions

Bureaucrat Number 1.0: *You are technically correct: the best kind of correct!*
— Futurama Ser. 2, Ep. 11

Contents

In the study of structure-changing Petri nets and structure-changing workflow nets in Chapter 4, system and environment are treated on an equal footing. Both are modelled as rule sets whose rules could be interleaved in arbitrary order.

Adverse conditions are only modelled insofar as the two rule sets contained different kinds of rules (firing and reconfiguration, respectively) and reconfiguration could be said to be an adverse condition for the correct termination of a workflow. The leading idea is to imagine the interaction between system and environment as *interleaving* or *parallelism* between their respective actions.

Simply partitioning the rule set is insufficient to express the distinction between a *controlled* system and an *uncontrolled* environment. The study of correctness of graph programs under adverse conditions is also concerned with this distinction.

The notion of correctness presented in this chapter is abstract in the sense that the problems of controller design or program synthesis are excluded, but it is usable within the proof-based framework of Chapter 5 and allows proofs to be carried out in the same way as for the programs of Chapter 5.

An adverse condition model formally describes the possible influences of the environment. It may include assumptions on the frequency of faults and conditions under which the various faults may occur. Assumptions on their frequency are modelled in our framework as part of **Asm**. Assumptions on the precise circumstances under which a fault can occur however are part of **Env** because they are best represented as application conditions in graph rules, if the faults are represented as transformations rules too. Thus the possible interplays between system and environment are determined both by the possible schedulings (frequency of faults) and shapes of rules.

In the systems we consider, state spaces (though never the individual states) easily become very large or even countably infinite. The ability to formulate abstractions and perform computations on them is essential to deal with these systems. When we employ the term "error state", we mean a very simple instance of abstraction building. It implies a lumping of all conceivable states of the system into two meta-states: error and non-error. Figure 6.1 shows an abstract state space of a system with reversible (repairable) errors.

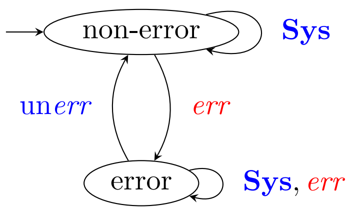


Figure 6.1.: Reversible errors: abstract transition system over-approximating the possible behaviours. The transitions are not always enabled in a concrete state.

The meta-states error and non-error are represented symbolically, as formulæ in a logic or as graph conditions. The proof goal then is to show that the transition labelled un*err* is always possible.

This chapter is structured as follows: Section 6.1 briefly recalls the special case of structure-changing Petri nets to motivate our notion of correctness under adverse conditions. In Section 6.2, a formal framework is presented that distinguishes system from environment actions according to a simple control model that allows system actions to be chosen arbitrarily to satisfy the specification, while the remaining nondeterminism is adverse. This framework is instantiated in concrete terms and we show how to extend the weakest precondition calculus to such systems. This enables an investigation of the interaction of system and environment under the assumption of intermittent faults within the basic framework of partial correctness. Section 6.3 concludes with an outlook and Section 6.4 lists bibliographic notes.

6.1. Adverse Reconfigurations

As a motivation, let us reconsider the case of structure-changing Petri nets investigated in Chapter 4. The reachability problems asked for the possibility of reaching a certain state, or abstract state (which was a set of states specified by a multiset of place colours), under any choice of firings and reconfigurations. Such an abstract state may model an error state, for instance, and the structure-changing workflow net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$ is "correct" if such a state cannot be reached. We may instead envisage a new abstract reachability problem "under adversity", asking for the existence of a sequence of firing steps that are interleaved with finite sequences of uncontrolled reconfigurations. After each **Sys** firing step, **Env** performs a number (according to model assumptions, either arbitrary or constrained) of reconfiguration steps. The nondeterminism of the environment is then modelled by defining $\Rightarrow'_{\mathcal{S}} := \{(\mathcal{N}, \mathcal{N}') \mid \exists \mathcal{N}'', t \in \Sigma, \varrho \in R^*, \mathcal{N} \overset{t}{\Rightarrow} \mathcal{N}'' \overset{\varrho}{\Rightarrow} \mathcal{N}'\}$ and extending the relation to sets as $\Rightarrow''_{\mathcal{S}} := \{(\mathfrak{N}, \mathfrak{N}') \mid \mathfrak{N}' = \bigcup_{\mathcal{N} \in \mathfrak{N}} \{\mathcal{N}' \mid \mathcal{N} \Rightarrow'_{\mathcal{S}} \mathcal{N}'\}\}$.

Now the situation is slightly different. As reconfigurations have been introduced to model *adverse* conditions, they really should be uncontrolled. In the behaviour of (**Sys**‖**Env**), adverse rules should be universally quantified: the question is not only whether a certain derivation exists, but whether *all* the derivations that differ in the choice of environment steps can still be controlled by system steps so as to fulfil the specification.

We may formulate a modified abstract reachability problem such as the following: given a k-coloured structure-changing Petri net $\mathcal{S} = (\mathcal{N}, \mathcal{R})$ for some $k \in \mathbb{N}$ and some multiset $q : \{0, ..., k-1\} \to \mathbb{N}$, is there a set $\mathfrak{N}$ such that $\{\mathcal{N}\} \Rightarrow''^{*}_{\mathcal{S}} \mathfrak{N}$ and $\alpha(\mathfrak{N}) = \{q\}$? This is an example *in nuce* of our idea of correctness under adverse conditions, to be introduced in this chapter. However, we will *not* pursue the specific case of structure-changing Petri nets any further and instead formulate correctness under adverse conditions in the more general framework of graph programs and graph conditions, which can be specialised to structure-changing Petri nets should the need arise: by virtue of the translation sketched in Section 2.3 and by encoding Definition 6 as a graph transformation rule, structure-changing Petri nets are a special case of graph transformation systems. The negated abstract reachability problem is then easily seen to fit our program correctness framework introduced in the previous chapter. However, the non-negated version asks for reachability rather than safety and the modified one sketched in this section requires an interleaving of safety and reachability. In the following, we propose an extension of the formalism to allow such properties to be verified.

6.2. Modelling Control: Own and Adverse Steps

System and environment actions (steps) must have a different semantics because the former are controlled (the composition of **Sys** and **Env** being correct as long as there exists a choice of next action satisfying the postcondition), while the latter are not, in the sense that to be correct, all possibilities must lead into a state satisfying the postcondition.

It is worth noting that in the verification methods for graph programs we base our work on, neither the specification consisting of pre- and postconditions nor the semantics (Definition 3) of graph programs makes any reference to intermediary states. This is unlike the work on model checking listed in Section 5.4, since temporal specifications rely on such a notion: in order for temporal modalities to be interpretable, a step semantics must be defined instead. The same is true for this section.

We stress that the extended formalism presented in this chapter represents a point in a design space and different design choices are possible to fit into the framework:

- The choice of program operators (we construct program terms using the Kleene operators).

- The treatment of parallelism (we defer the treatment of parallelism for the time being)
- The separate interfaces which will be introduced

6.2.1. 2-Player Programs and Their Semantics

One can view the controller **Sys** and the environment **Env** as two participants in a game. We start by proposing a modified version of Definition 3 (graph programs, now also called *(program) terms* because of the new, step-wise execution semantics detailed in the following part). The definition is exactly as Definition 3 apart from the index $\pm \in \{+,-\}$[1] The only formal difference is a superscript that labels each program with one of these two symbols.

Definition 38 (**2-Player Graph Programs**).
A two-player graph program is one of the following:

- *elementary terms* $\mathrm{Sel}^{\pm}(x)$, $\mathrm{Del}^{\pm}(l)$, $\mathrm{Add}^{\pm}(r)$, $\mathrm{Uns}^{\pm}(y)$ *for* $\pm \in \{+,-\}$,
- $skip^{\pm}$ *for* $\pm \in \{+,-\}$,
- *If* P *and* Q *are two-player graph programs then so are* $P \cup^{+} Q$ *and* $P \cup^{-} Q$ *if* $\mathrm{in}(P) = \mathrm{in}(Q)$ *and* $\mathrm{out}(P) = \mathrm{out}(Q)$, $P^{*^{+}}$ *and* $P^{*^{-}}$ *if* $\mathrm{in}(P) = \mathrm{out}(P)$, $P;Q$ *if* $\mathrm{in}(Q) = \mathrm{out}(P)$ (in *and* out *are defined below*).

To every program an input and an output interface are unambiguously assigned. The sequential composition $P;Q$ *is only well-formed when* P *and* Q *agree on their interfaces. These input and output interfaces are defined as* (*for* $\pm \in \{+,-\}$)*:*

program P	$\mathrm{in}(P)$	$\mathrm{out}(P)$
$\mathrm{Sel}^{\pm}(x)$	$\mathrm{dom}(x)$	$\mathrm{cod}(x)$
$\mathrm{Del}^{\pm}(l)$	$\mathrm{cod}(l)$	$\mathrm{dom}(l)$
$\mathrm{Add}^{\pm}(r)$	$\mathrm{dom}(r)$	$\mathrm{cod}(r)$
$\mathrm{Uns}^{\pm}(y)$	$\mathrm{cod}(y)$	$\mathrm{dom}(y)$
$P;Q$	$\mathrm{in}(P)$	$\mathrm{out}(Q)$
$P^{*^{\pm}}$	$\mathrm{in}(P)$	$\mathrm{out}(P)$

The interfaces of programs can be checked statically as it is determined by the morphisms used. The interfaces of $skip^{\pm}$ are indeterminate, subject to the condition that input equals output, so there is really one $skip^{+}_{P}$ and one $skip^{-}_{P}$ for each graph P.

For example, Figure 6.2 below is a two-player program with input and output interface $\emptyset$. Its intuitive meaning is as follows: the environment chooses how often the loop is executed. At each iteration, the controller decides whether an addition or a deletion of a loop should take place. The choice of match is again up to the environment, i.e. the controller cannot decide which loop is deleted or where it is added.

[1] We use $\pm$ when a definition is analogous for $+$ and $-$. We use the symbol $+$ to indicate system actions (program terms, sets ...) and $-$ for all such belonging to the environment.

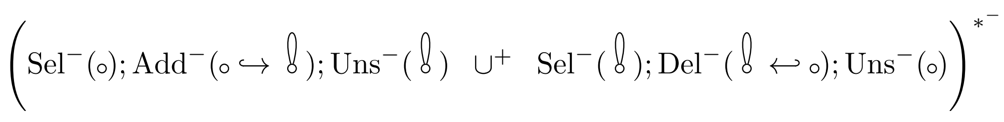


Figure 6.2.: A two-player program.

Notation. *All morphisms in the diagrams of the following pages are mono, even if represented as straight arrows.*

To lift programs with interface to a two-player setting, a conscious design choice is in order. The modelling of the interaction between two processes can be made more natural by changing the definition of $[\![\cdot]\!]$. Instead of just a graph with a single selection, the states reached by a program are now graphs annotated with two, possibly intersecting, selections. One subgraph selection $B^+ \hookrightarrow G$ belongs to the ocntrollre and the other selection $B^- \hookrightarrow G$ belongs to the environment, and the selections can overlap.

Definition 39 (2-Player Interfaces). *$\mathcal{Y}$ is the class of all diagrams of the following type, where the two morphisms (y^+, y^-) are jointly surjective:*

$$\begin{tikzcd}
B^+ \arrow[dr, hook, "y^+"'] & & B^- \arrow[dl, hook', "y^-"] \\
& B \arrow[d, hook, "g"'] & \\
& G &
\end{tikzcd}$$

Formally, instead of tuples of two monomorphisms and one partial monomorphism, the elements in our new semantics of two-player programs are quadruples $(Y_{\text{in}}, Y_{\text{out}}, p^+, p^-) \in \mathcal{Y} \times \mathcal{Y} \times \mathcal{PM} \times \mathcal{PM}$ where $Y_{\text{in}} = ({y_{\text{in}}}^+, {y_{\text{in}}}^-, g_{\text{in}})$, $Y_{\text{out}} = ({y_{\text{out}}}^+, {y_{\text{out}}}^-, g_{\text{out}})$ and p^+ is a partial monomorphism from $\text{dom}({y_{\text{in}}}^+)$ to $\text{dom}({y_{\text{out}}}^+)$ and p^- from $\text{dom}({y_{\text{in}}}^-)$ to $\text{dom}({y_{\text{out}}}^-)$.

The idea is that the intuitive meaning and the formal semantics should deviate as little as possible from the plain (non two-player) case so we can adapt the proofs from Section 5.1. The condition itself is oblivious of the individual selections.[2] Satisfaction is not changed at all, it simply ignores the individual interfaces:

Definition 40 (Satisfaction). *If $Y = (y^+, y^-, f) \in \mathcal{Y}$ and c is a μ-condition, then $Y \models c$ iff $f \models c$.*

We can now define a semantics for two-player graph programs, closely related to the semantics of graph programs as defined in Chapter 2: a selection step is still a selection step, although it changes only the selection whose superscript ($+$ or $-$) corresponds to

[2]This, too, is only a design choice to simplify the presentation.

that of the program term. An addition step is still an addition step and changes the graph as well the selection of one player. An unselection step unselects part of one player's selection. There are no new impediments except for deletion, which is forbidden when an item that should be deleted is still selected by the other player.

Definition 41 (Semantics of 2-Player Programs). *Let $Y_{\mathrm{in}} = ({y_{\mathrm{in}}}^+, {y_{\mathrm{in}}}^-, g_{\mathrm{in}})$ and $Y_{\mathrm{out}} = ({y_{\mathrm{out}}}^+, {y_{\mathrm{out}}}^-, g_{\mathrm{out}})$ throughout.*

$[\![\mathrm{Sel}^+(x)]\!]_\pm = \{(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \mid (Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \in (\mathcal{Y} \times \mathcal{Y} \times \mathcal{PM} \times \mathcal{PM})\}$ *such that $x : {B_{\mathrm{in}}}^+ \hookrightarrow {B_{\mathrm{out}}}^+$ is a monomorphism, ${B_{\mathrm{in}}}^- = {B_{\mathrm{out}}}^-$, (x'', y') is a pushout of $({y_{\mathrm{in}}}^+, x)$, e a surjective morphism with domain* $\mathrm{cod}(y')$ *(in the diagram E stands for* $\mathrm{cod}(e)$*) and $(x', {y_{\mathrm{out}}}^+) = (e \circ x'', e \circ y')$, and there exists a morphism $g_{\mathrm{out}} : \mathrm{cod}(x') \hookrightarrow G$ to make the diagram commute, then Y_{out} is defined with ${y_{\mathrm{out}}}^+ = x' \circ {y_{\mathrm{in}}}^+$, and $p^+ = x$.*

$[\![\mathrm{Sel}^-(x)]\!]_\pm$ *is exactly analogous, i.e.* $\{(Y_{\mathrm{in}}, Y_{\mathrm{out}}, \mathrm{id}_{{B_{\mathrm{in}}}^+}, p^-) \mid (Y_{\mathrm{in}}, Y_{\mathrm{out}}, \mathrm{id}_{{B_{\mathrm{in}}}^+}, p^-) \in (\mathcal{Y} \times \mathcal{Y} \times \mathcal{PM} \times \mathcal{PM}\}$ *with all $+$ and $-$ swapped subsequently.*

$[\![\mathrm{Uns}^+(y)]\!]_\pm = \{(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \mid (Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \in (\mathcal{Y} \times \mathcal{Y} \times \mathcal{PM} \times \mathcal{PM})\}$ *such that $({y_{\mathrm{out}}}^+, {y_{\mathrm{out}}}^-)$ is the pushout over the pullback of $({y_{\mathrm{in}}}^-, {y_{\mathrm{in}}}^+ \circ y)$; unique morphism g_{out} from the pushout object (this is necessarily jointly surjective, as required).* Uns^- *is defined analogously. The situation for* Uns^+ *is represented below:*

$$\begin{tikzcd}
 & B'' \arrow[dl] \arrow[dr] & \\
B^- \arrow[dr] & \text{(PB)} & B'^+ \arrow[dl] \arrow[d] \\
 & B & B^+ \arrow[l]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & B'' \arrow[dl] \arrow[dr] & \\
B^- \arrow[dr] & \text{(PO)} & B'^+ \arrow[dl] \\
 & B' &
\end{tikzcd}$$

$[\![\mathrm{Add}^+(r)]\!]_\pm = \{(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \mid (Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \in (\mathcal{Y} \times \mathcal{Y} \times \mathcal{PM} \times \mathcal{PM})\}$ *such that (g', p') is pushout of $(r, g_{\mathrm{in}} \circ {y_{\mathrm{in}}}^+)$, $({y_{\mathrm{out}}}^+, r')$ is pushout of $(r, {y_{\mathrm{in}}}^+)$, g_{out} is obtained as the morphism from the inner pushout object* $\mathrm{cod}(r')$ *to* $\mathrm{cod}(r') = G'$ *and ${y_{\mathrm{out}}}^-$ is $r' \circ {y_{\mathrm{in}}}^-$.*

$[\![\mathrm{Del}^+(l)]\!]_\pm = \{(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \mid (Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, \mathrm{id}_{{B_{\mathrm{in}}}^-}) \in (\mathcal{Y} \times \mathcal{Y} \times \mathcal{PM} \times \mathcal{PM})\}$ *where $p^+ = l^{-1}$ such that $(l', {y_{\mathrm{in}}}^+)$ is a pushout over $(l, {y_{\mathrm{out}}}^+)$ and (l'', g_{in}) is a pushout over (l', g_{out}).*

Composition is analogous to Definition 3 (existence of two *composite partial monomorphisms) and $P \cup Q$, P^* are also as before: $[\![P \cup Q]\!]_\pm = [\![P]\!]_\pm \cup [\![Q]\!]_\pm$, $[\![P; Q]\!]_\pm = \{(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+; p'^+, p^-; p-^+ \mid \exists (Y_{\mathrm{in}}, Y', p^+, p^-) \in [\![P]\!]_\pm, (Y', Y_{\mathrm{out}}, p'^+, p'^-) \in [\![Q]\!]_\pm\}$.*

An *atomic program* or *action* is any of the elementary steps.

In the following illustration Figure 6.3 of Definition 41, the diagrams depict the situations for the atomic 2-player . The input interface $Y_{\mathrm{in}} = ({y_{\mathrm{in}}}^+, {y_{\mathrm{in}}}^-, g_{\mathrm{in}})$ is highlighted in a brighter shade and the output interface $Y_{\mathrm{out}} = ({y_{\mathrm{out}}}^+, {y_{\mathrm{out}}}^-, g_{\mathrm{out}})$ is highlighted in a darker shade. Recall that two-player interfaces are Y-shaped diagrams. In order not to disturb the symmetry between the semantics of the (+)-steps and the (−)-steps, only the Y diagrams of the bottom row are oriented as in **??** with the y^+ morphism starting at the top left hand corner, while the Y diagrams in the top row are mirrored on the vertical axis. Note that all morphisms are monomorphisms even though normal arrows are used.

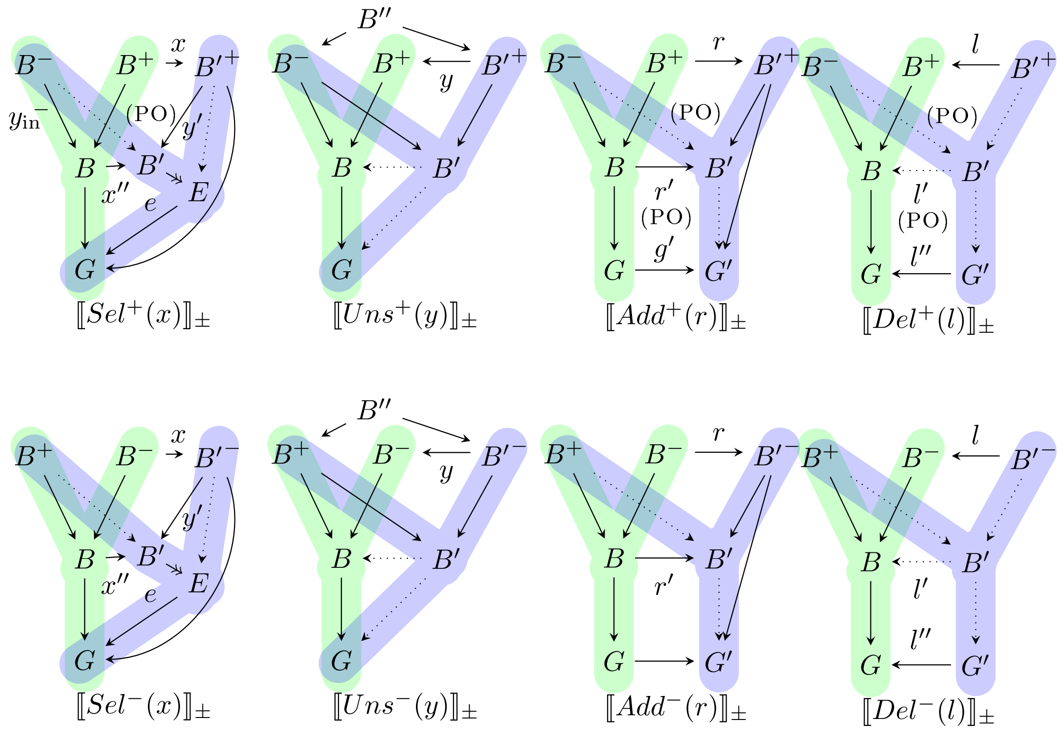


Figure 6.3.: Atomic 2-player graph programs. Input and output interfaces are highlighted.

The justification for postulating the existence of a factorisation in $[\![Del^\pm]\!]$ is that a deletion step must not be able to damage the other player's interface. This peculiarity will be reflected in the Wlp-like construction.

The second, more important change vis-à-vis the one-player situation is that a control model must be introduced and intermediary steps defined because *choice* of action now becomes important. So far, correctness meant that *no* execution starting from a state satisfying c resulted in a bad state satisfying $\neg d$. In other words, *all* choice was considered to be adverse. When some actions are controlled, it might be possible to avoid bad states by resolving nondeterminism in a safe way (if one exists). The semantics of a two-player program is a transition system, and in game-theoretic terms our focus is on *safety* games. Introducing a notion of control will distinguish some cases X^+, X^- that are otherwise handled identically in Definition 41 such as $X \cup^+ Y$ vs. $X \cup^- Y$.

The plan is now to introduce abstract process steps in the manner of a process algebra, then combine these with the $[\![\cdot]\!]$ semantics of the steps, similarly to the notion of Structural Operational Semantics introduced by Plotkin [Plo04]. Finally, the appropriate generalisation of the weakest precondition calculus will be developed. A transition system (labelled over a transition label alphabet Σ) consists of a set of states S, a relation $\Xi \subseteq S \times \Sigma \times S$ and a start state s_0.

Notation. *Let $\Sigma(P)$ be the set of all atomic programs which syntactically appear in the term P. The symbol $\checkmark$ stands for the empty program skip and τ stands for skip as an atomic action:*

Definition 42 ((+)- and (−)-programs)**.** *Let $\mathcal{X}^+$ be the collection of all program terms whose topmost operator (in a sequence $P;Q$, defined as the topmost operator of P) has the superscript $(+)$ and $\mathcal{X}^-$ analogously for superscript $(-)$.*

By this definition, every program belongs either to $\mathcal{X}^+$ or $\mathcal{X}^-$. We introduce a semantics that assigns control of the next step to the respective player (controller **Sys** for $(+)$, environment **Env** for $(-)$).

Definition 43 (Abstract Process Steps)**.** *The labelled transition system $\mathcal{X}(P) = (\mathcal{X}(P), \xi, P)$, with labels from $\Xi = \Sigma_P \times \{+,-\}$ where $\mathcal{X}(P)$ is the set of all reachable terms given by the rules below, $\xi \subseteq \mathcal{X}(P) \times \Xi \times \mathcal{X}(P)$. Its set of transitions ξ is given by the rules in the table below:*

Atomic:	$\frac{}{X \xrightarrow{X}_+ \checkmark}$ *for atomic* $X \in \mathcal{X}^+$	$\frac{}{X \xrightarrow{X}_- \checkmark}$ *for atomic* $X \in \mathcal{X}^-$.
Choice:	$\frac{}{P \cup^+ Q \xrightarrow{\tau}_+ P}$ $\quad \frac{}{P \cup^+ Q \xrightarrow{\tau}_+ Q}$	$\frac{}{P \cup^- Q \xrightarrow{\tau}_- Q}$ $\quad \frac{}{P \cup^- Q \xrightarrow{\tau}_- Q}$
Sequence:	$\frac{P \xrightarrow{X}_+ P'}{P;Q \xrightarrow{X}_+ P';Q}$ $\quad \frac{P \xrightarrow{X}_+ \checkmark}{P;Q \xrightarrow{X}_+ Q}$	$\frac{P \xrightarrow{X}_- P'}{P;Q \xrightarrow{X}_- P';Q}$ $\quad \frac{P \xrightarrow{X}_- \checkmark}{P;Q \xrightarrow{X}_- Q}$
Iteration:	$\frac{}{P^{*+} \xrightarrow{\tau}_+ \checkmark}$ $\quad \frac{}{P^{*+} \xrightarrow{\tau}_+ P;P^{*+}}$	$\frac{}{P^{*-} \xrightarrow{\tau}_- \checkmark}$ $\quad \frac{}{P^{*-} \xrightarrow{\tau}_- P;P^{*-}}$

Notation. Next(P) *is the set of all atomic programs X such that $P \xrightarrow{X}_\pm P'$ for some $P', \pm$ according to Definition 43.*

Note that by Definition 43, each program term unambiguously assigns control to one of the players because the transitions leaving it can easily be seen to be all of one type (+, −). This matches Definition 42 and so we define subsets of $\mathcal{X}^+$ and $\mathcal{X}^-$ associated with a program P:

Definition 44 (State Ownership)**.** *For each two-player graph program P, let*

$\mathcal{X}^+(P) := \{Q \mid P \xrightarrow{*} Q\} \cap \mathcal{X}^+$

$\mathcal{X}^-(P) := \{Q \mid P \xrightarrow{*} Q\} \cap \mathcal{X}^-$

Some of the abstract transitions are never accompanied by any change of the state, they correspond to a choice being made. Luckily, because of their regular nature, our program terms present no additional difficulty.

Lemma 18 (Finite Abstract Transition Systems).
The transition system $(\mathcal{X}(P), \Xi, \xi, P)$ is finite for any two-player graph program P.

Proof. The transitions are derivatives of a regular expression in the sense of Brzozowski [Brz64]. The claim follows using Theorem 4.3 in [Brz64]. □

The steps of this transition system are abstract and when starting from an actual state, there may be zero or more transitions actually possible. A *configuration* of P is an ordered pair $(s, X) \in \mathcal{Y} \times \mathcal{X}(P)$:

Definition 45 (Step Semantics).
$(s, P) \to_\pm (s', P')$ if $P \overset{X}{\to} \pm P'$ according to Definition 43 and $(s, s', p^+, p^-) \in [\![X]\!]$ for some partial monomorphisms p^+, p^-.

6.2.2. Weakest Preconditions

Having defined programs with ownership and endowed them with a step semantics compatible with previous definitions, we proceed to define a precondition construction and a suitable extension of the notion of weakest precondition. The new definitions are designed to extend the definitions from Chapter 6 to the two-player setting. Since conditions and their satisfaction only depends on the common interface P and not P^- and P^+ due to the way it was defined, intuitively a weakest precondition construction looks like Figure 6.4 for each of the atomic programs. The "proc" in the name of the construction stands for "process".

Furthermore, the weakest-precondition-like construction must reflect the two types of nondeterminism. The following definition is arranged to highlight the analogy between the (+) and (−) cases. The (−) case is similar to the constructions of Chapter 6 but must now also account for the effect on the player interfaces; this is the only difference. The (+) case mirrors the (−) case and it will subsequently be shown that this construction is appropriate, after introducing the extended notion of correctness.

Construction 11 (2-Player "Weakest Precondition"). *The cases in the following table defines a transformation $\mathrm{Wlproc}_P : \mathcal{Y} \to \mathcal{Y}$ for each elementary two-player program P.*

$\mathrm{Wlproc}_{\mathrm{Sel}^+(x,c')}(y_0, y_1, d) =$ $(y_0', y_1', \exists(x', c' \wedge d))$ *where* (y_0', x') *is a pushout complement of* (x, y_0) *and* $y_1 = x' \circ y_1'$. *If no such factorisation of* y_1 *exists or if the pushout complement fails to exist, then* $\mathrm{Wlproc}(y_0, y_1, \mathrm{Sel}_+(x, c'), d) = \bot$.	$\mathrm{Wlproc}_{\mathrm{Sel}^-(x,c')}(y_0, y_1, d) =$ $(y_0, y_1', \forall(x', c' \wedge d))$ *(constructed analogously to* Sel^+ *case)*

$\mathrm{Wlproc}_{\mathrm{Del}^+(l)}(y_0, y_1, d) = (y'_0, y'_1, d')$ *where* $d' = \Delta(l') \wedge \alpha'_{l'}(d)$ *(similarly, but* $y'_1 = l' \circ y_1$*):* (y'_0, l') *pushout of* (y_0, l)*, then* (g, l'') *by pushout* (g', l')*.* l'' *is discarded.*	$\mathrm{Wlproc}_{\mathrm{Del}^-(l)}(y_0, y_1, d) = (y'_0, y'_1, d')$ *where* $d' = \Delta(l') \Rightarrow \alpha'_{l'}(d) = \mathrm{Wlp}(\mathrm{Del}(l'), d)$ *(otherwise analogous to* Del^+*)*
$\mathrm{Wlproc}_{\mathrm{Add}^+(r)}(y_0, y_1, d) = (y'_0, y'_1, \mathrm{Wlp}(\mathrm{Add}(r'), d))$ *(*y'_1 *as for* Sel *again)*	$\mathrm{Wlproc}_{\mathrm{Add}^-(r)}(y_0, y_1, d) = (y'_0, y'_1, \mathrm{Wlp}(\mathrm{Add}(r'), d))$
$\mathrm{Wlproc}_{\mathrm{Uns}^+(y)}(y_0, y_1, d) = (y'_0, y'_1, \mathrm{Wlp}(\mathrm{Uns}(y'), d))$ *(*y'_1 *as for* Del *again)*	$\mathrm{Wlproc}_{\mathrm{Uns}^-(y)}(y_0, y_1, d) = (y'_0, y'_1, \mathrm{Wlp}(\mathrm{Uns}(y'), d))$
$\mathrm{Wlproc}_{P \cup^- Q}(y_0, y_1, d) = (y'_0, y'_1, x \wedge y)$ *where* $(y'_0, y'_1, x) = \mathrm{Wlproc}(P, y_0, y_1, d)$ *and* $(y'_0, y'_1, y) = \mathrm{Wlproc}(Q, y_0, y_1, d)$ *(according to definition,* P *and* Q *must agree on input and output interfaces for* $P \cup Q$ *to be defined)*	$\mathrm{Wlproc}_{P \cup^+ Q}(y_0, y_1, d) = (y'_0, y'_1, x \vee y)$ *where* $(y'_0, y'_1, x) = \mathrm{Wlproc}(P, y_0, y_1, d)$ *and* $(y'_0, y'_1, y) = \mathrm{Wlproc}(Q, y_0, y_1, d)$

Sequence is handled recursively: $\mathrm{Wlproc}_{P;Q}(y_0, y_1, d) = \mathrm{Wlproc}_P(\mathrm{Wlproc}_Q(y_0, y_1, d))$

The cases of P^{*^+} and P^{*^-} would be infinite Boolean combination, which are not allowed. The construction can be formally extended to these cases to yield expressions which are not μ-conditions and can in general only be approximated by μ-conditions. Several more steps are needed before we can formulate and prove the main result of the chapter, which is the soundness of Construction 11:

Lemma 19 (Steps and Denotation).
$[\![P]\!]_\pm = \bigcup_{P \overset{X_1 \ldots X_n}{\to}{}^* \checkmark} [\![X_1; \ldots; X_n]\!]_\pm$*, where* $X_1, \ldots X_n$ *are atomic steps.*

Proof. By structural induction over P. The base case for atomic steps holds by Definition 43. $[\![P \cup Q]\!]_\pm = [\![P]\!]_\pm \cup [\![Q]\!]_\pm$ while $P \cup Q \to R$ iff $P \to R$ or $Q \to R$, thus the statement holds by assumption on P and Q and set union; $[\![P; Q]\!]_\pm$: assuming the statement holds for Q and P', it holds because $P; Q \xrightarrow{X} P'; Q$ iff $P \xrightarrow{X} P'$ and $\bigcup_{P \xrightarrow{X} P'} [\![X; P'; Q]\!]_\pm = [\![X]\!]_\pm; [\![P'; Q]\!]_\pm$. $[\![P^{*^\pm}]\!]_\pm$ is defined as $\{skip^\pm\} \cup^\pm \{P\} \cup^\pm \{P; P\} \ldots$, induction over the natural numbers and statement for $\cup$. □

This notion of semantics is not quite adequate for the modelling of controlled actions.

Definition 46 (Strategies). *A* strategy *is a function* $\chi : \mathcal{X}^+(P) \times \mathcal{Y} \to \Sigma_P \times (\mathcal{Y} \times \mathcal{Y} \times \mathcal{PM} \times \mathcal{PM})$ *which selects an action* X *from* $\mathrm{Next}(P)$ *and tuple* (y, y', p^+, p^-) *from* $[\![X]\!]$*.*

A strategy thus selects, given the pair (P, y) of program term and current state, one possible action X such that $P \xrightarrow{X}_+ P'$, and one element from the set $[\![X]\!]$. It can be

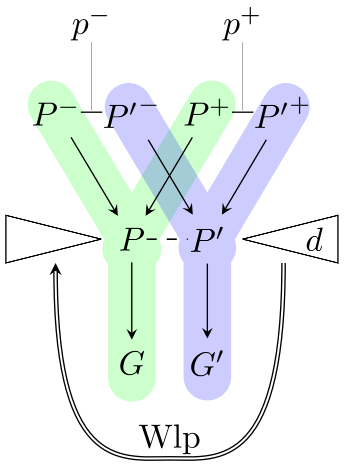


Figure 6.4.: Towards adapting weakest preconditions.

said to *resolve the nondeterminism* inherent in the definition of the semantics of graph programs.

In other words, a strategy seeks to *ensure* a certain behaviour by picking a successor configuration at each (s, P) where $s \in \mathcal{X}^+$. If there is no suitable configuration to ensure that the specified postcondition holds, there is also no suitable strategy and the specification will be incorrect. The asymmetry between the (+) and (−) cases is intentional.

Definition 47 (χ**-Ensured semantics**). *Let* $[\![P]\!]_\chi$ *be defined recursively as follows:*

- *If* $P \in \mathcal{X}^-$, *then* $[\![P]\!]_\chi := \bigcup_{P \xrightarrow{X}_- P'} [\![X]\!]; [\![P']\!]_\chi$
  (using composition of partial monomorphisms).
- *If* $P \in \mathcal{X}^+$, *then* $[\![P]\!]_\chi = [\![X]\!]; [\![P']\!]_\chi$ *where* $(P, X, P') = \chi(P)$.

A specification $\{c\}P\{d\}$ is now said to be *correct* if there exists a strategy that ensures a choice of correct successor states.

Definition 48 (**Correctness**). *If* P *is a two-player graph program, a specification* $\{c\}P\{d\}$ *is* correct *if there exists a strategy* $\chi : \mathcal{X}^+(P) \to \Sigma_P$ *such that* $y' \models d$ *whenever* $(y, y', p^+, p^-) \in [\![P]\!]_\chi$ *with* $y \models c$.

The role of χ is to resolve nondeterminism at (concrete) (+)-owned configurations by providing a choice of system-controlled actions for each possible configuration, while nondeterminism at (−)-owned states is not resolved. The choice of action and match for (+) is constrained only by the semantics $[\![\cdot]\!]_\pm$. The programs $P \cup^- Q$ and $P \cup^+ Q$ represent choice points without any state transitions. From general considerations on game theory, its is known that considering *positional strategies* are sufficient even for parity games [GTW02].

The case of a program without any + steps or operators trivially reduces to the weakest precondition and correctness notion of the previous chapter. That is,

Fact 11 (Adverse-only correctness). *A program built from only $(-)$-steps and adverse choice $\cup^-$ and adverse iteration P^{*^-} is correct with respect to a pre- and condition (c, d) iff the program in the sense of Definition 3 obtained by stripping the $-$ superscripts is correct in the sense of Definition 31.*

Proof. From Definition 46, the requirement of existence of a strategy is vacuous in this case. From Construction 11, the selection y^- is always empty and by joint surjectivity, y^+ is the identity. Therefore the definition of correctness reduces to that of one-player programs as defined in Definition 3, Definition 31. □

The crucial part, as for the graph programs and conditions of the previous chapter, is that the weakest precondition construction is correct. Now one can re-use the reasoning of Theorem 3, adapting the proofs for the plain case to choice and two-player interfaces.

Theorem 5 (Weakest Precondition). $\mathrm{Wlproc}_P(d)$ *is the least condition with respect to implication such that* $Y \models \mathrm{Wlproc}_P(d) \Rightarrow Y' \models d$ *if and only if there is a strategy* $\chi : \mathcal{X}^+(P) \to \Sigma_P$ *such that* $(Y, Y', p^+, p^-) \in [\![P]\!]_\chi$ *for some pair of partial monomorphisms* (p^+, p^-).

Proof. Case of a sequence of atomic steps: induction over the length of the sequence with $P = X; P'$. Assume that the property holds for $\mathrm{Wlproc}_{P'}$ and conclude that also for P, $\mathrm{Wlproc}_P(d)$ is implied by a precondition c if and only if $\{c\}P\{d\}$ is correct:

- If $Y_{\mathrm{in}} \models \mathrm{Wlproc}_X(d)$, $X \in \mathcal{X}^+$, there must exist a $(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, p^-) \in [\![X]\!]$ such that $Y_{\mathrm{out}} \models d$.
- If $Y_{\mathrm{in}} \models \mathrm{Wlproc}_X(d)$, $X \in \mathcal{X}^-$, then for all $(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, p^-) \in [\![X]\!]$, $Y_{\mathrm{out}} \models d$ must hold.

This must be shown for every program.

$\mathrm{Sel}^-(x, c')$: since the semantics (see Definition 41) does not add anything to selection of other player, if $Y' \models d$, then the condition on the required factoring of y_1 and the existence of the pushout complement holds at any valid precondition because the semantics contains only $(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, p^-)$ related in this way. Then use the proof of Theorem 3 on the lower part of the diagram (disregarding the individual selections y_{in}, y_{out}) and transform d to $\mathrm{Wlp}(\mathrm{Sel}(x, c'), d)$.

$\mathrm{Sel}^+(x, c')$ is analogous but for the quantification: according to Definition 46 and Definition 48, for any correct specification, there exists a choice $(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p, p)$ such that $Y_{\mathrm{out}} \models d$. The existential quantifier states the existence of a suitable extension of the current ${y_{\mathrm{in}}}^+$, determining a suitable tuple from the semantics.

$\mathrm{Del}^-(l)$: in the semantics, use the special PO-PB lemma to see that the existence of two pushout squares is postulated. The requirement that whenever $(Y_{\mathrm{in}}, Y_{\mathrm{out}}, p^+, p^-)$ with

$Y_{\text{in}} \models c$ ($\Leftrightarrow g_{\text{in}} \models c$ per definition), $Y_{\text{out}} \models d$ ($g_{\text{out}} \models d$) is that of a plain weakest liberal precondition (Theorem 3).

$\text{Del}^+(l)$: same as $\text{Del}^-(l)$, but with $\Delta(l') \wedge \alpha'_l(d)$ instead of $\Delta(l') \Rightarrow \alpha'_l(d)$: there is no choice because the program term has only one outgoing transition, but the definition of wlproc states that a transition must nevertheless exist.

$\text{Add}^-(r)$ and $\text{Add}^+(r)$ are analogous to $\text{Del}^-(r)$ and $\text{Del}^+(r)$: as addition steps never fail, the $(-)$ and the $(+)$ case are alike except for the quantifier.

$\text{Uns}^-(r)$ and $\text{Uns}^+(r)$: the reduction to Wlp is again analogous to the previous cases. Unselection steps never fail and there is no choice, hence $(-)$ and $(+)$ are alike.

$P \cup^- Q$ and $P \cup^+ Q$: in the former case, the outgoing transitions from $(P \cup^- Q, s)$ lead to (P, s) and (Q, s) and each must satisfy the respective $\text{Wlproc}_P(d)$ resp. $\text{Wlproc}_Q(d)$. In the latter case, only one must satisfy it because χ can be chosen accordingly to assign one of (P, s) or (Q, s) to $(P \cup_+ Q, s)$. □

Example 25 (An example with labelled discrete graphs)**.** *The* wlproc *transformation has been illustrated on labelled discrete graphs in Figure 6.5, which is found at the end of the chapter due to its bulk. We have abstained from constructing an example with more complicated graphs or conditions, but the techniques from this chapter are designed to be combined with those from the previous one.*

Deadlocks, i.e. the absence of steps from a certain configuration, have different consequences according to whether the deadlocked configuration belongs to the system or the environment. This concerns the situation of a program whose next step to be executed is a selection or a deletion, both of which may fail under certain conditions stated in the semantics. What seems like an asymmetric treatment of the players is indeed asymmetric, but it is merely an artefact of our definitions and we see no obstacles to refining these for a more satisfactory treatment.

Remark 10. *If $s \models c$ and (s, P) has no outgoing transition with the $(+)$-program P, then the* wlproc *of d will be unsatisfiable together with c and the specification will be incorrect. If it belongs to a $(-)$-program however, the specification is correct.*

6.3. Conclusion and Outlook

The main statement of this chapter is Theorem 5, the correctness of the weakest precondition construction for two-player graph programs, built upon and extending Theorem 3. In the literature, the distinction introduced in this chapter is also known as *angelic* versus *demonic* nondeterminism (cf., for instance [CvW03]).

We have demonstrated that one and the same framework is useful for describing systems under adverse conditions. While the basic idea is clear, there are many degrees of freedom in designing the formalism. We proposed one and demonstrated how it fits into the framework of Dijkstra-style partial correctness of graph programs. Some aspects are not fully satisfactory: it falls short of handling true concurrency; unintuitive treatment of deadlocks; conditions only speak about the graph part of a configuration and lack the means to distinguish configurations that differ by the remaining program. We feel confident that these shortcomings can easily be remedied in further work.

Defining the operational semantics of graph programs as a process algebra without concurrency, whose terms are partitioned according to ((+) and (−)) ownership is a deliberate choice in order to simplify the definitions and results in this chapter, though it is not the only conceivable possibility.

There are many possibilities for extensions. For example:

- Controller synthesis, which extends the scope of this thesis
- Properties beyond safety (when not reaching bad states is not enough). Reachability games would be a first step in that direction. When methods for proving total correctness become available in future work, these could likely be extended to reachability in the case of adverse conditions.
- A combination with temporal modalities would also allow for greater flexibility in the formulation of specifications. Our step semantics already offers one of the prerequisites.
- Concurrency. Philosophically, being forced to work with the notion of a *global*, always synchronously viewed state is a rather unsatisfactory feature of a formalism that describes local state changes, especially in view of the fact that the machinery for true concurrency in graph transformations is well-developed. Asynchronous games have also been researched as an abstract notion, see Rideau and Winskel [RW11]. Winskel's asynchronous games could be instantiated with graph transformations, using an event structure semantics [CEL$^+$96].

Systems modelled by graph transformations may be large, distributed and accordingly exhibit concurrent behaviour for which a satisfactory treatment already exists (notions of *locality* and *parallel independence* [EEPT06] in graph transformation). A future version of this work would make use of the existing theory to describe, and reason about, concurrent behaviour.

A parallel composition acquires a meaning only when intermediary steps are distinguished, otherwise the only ways two graph programs could interact would be considering whether their *track morphisms* [EEPT06] are *parellelly independent* (see ibid.). But in modelling interaction between system and environment, it is desirable to have a fine enough semantics to be able to detect deadlocks. Although such an operator is not defined in this work, it is a candidate for a future extension.

Rules could also interact in other ways, such as amalgamation (due to Boehm, Fonio and Habel [BFH87]). This may be a better model of faulty executions of actions when the occurrences of a certain error in a real world system are always concomitant with the execution of a system action. A rule is executed during the normal functioning of the system, but it malfunctions and some error rule is sometimes executed at the same time. This can model situations that do not occur when rules cannot directly interact. Amalgamable pairs of rule applications contain a common subrule and no *delete/keep* conflicts may exist, however a common deletion may be "factored out", leaving two parallel independent steps. We mention it because it provides an alternative to concurrency.

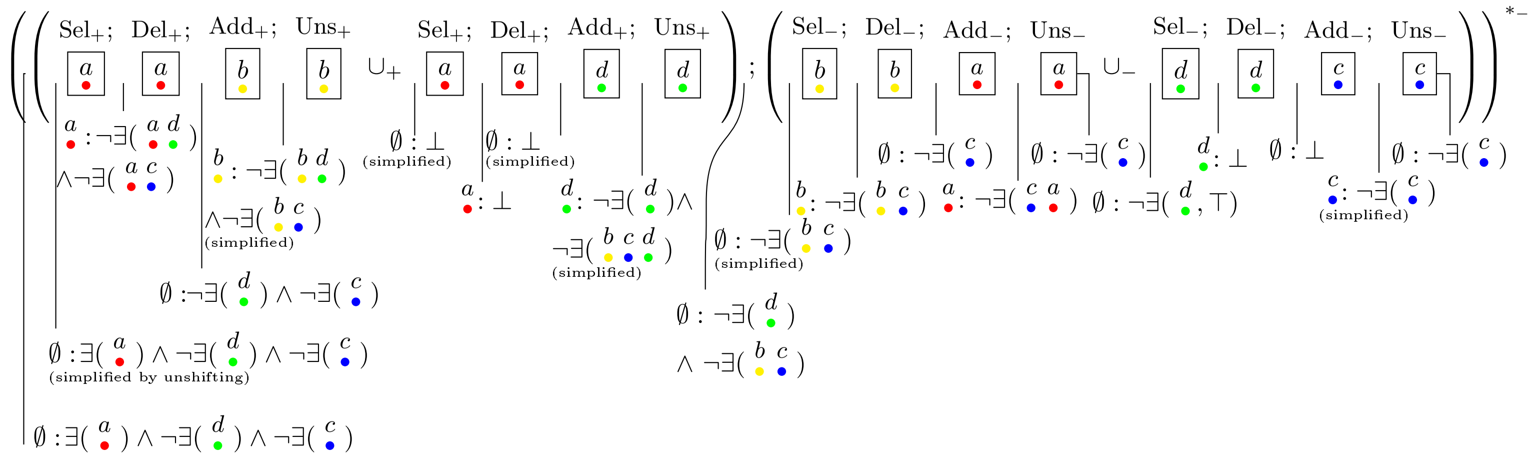
As an explicit justification of further design choices made in this chapter: the two-player interfaces are in principle entirely dispensable for the second part. It would have been entirely possible to declare only whole graph transformation rules as atomic steps ϱ, which have $\text{in}(\varrho) = \text{out}(\varrho) = \emptyset$. However, we wanted to stick closely to Chapter 5 and had to introduce the interfaces for improved modelling, the easier to keep own and adverse program parts from interfering in undesired ways.

6.4. Bibliographic Notes

Specifically for the verification of graph transformation systems, Heckel et al. [Hec98, HEWC97], Koch [Koc99], Baldan et al. and Rensink [BCKL07, Ren08] work with temporal logic, which is also based on a step semantics (of graph transformation systems, i.e. rule applications).

The interaction between system and environment can be understood as a game. Structure rewriting games have been introduced by Kaiser [Kai09]. They are based on hypergraph rewriting. Literature on games on graph transformation systems or Petri nets is otherwise scarce [RSVB05], with little use of true concurrency either. The new Petri games of Olderog et al., however, are defined such that strategies can only view the causal history of a local state [Old14, FO14, FGO15]. For control structures including transactional rules in graph rewriting see also Baldan et al. [BCD$^+$08]. For process algebras see Fokkink [Fok00].

Classes of games on graph transformation systems have also been studied (often without employing the terminology and machinery of the graph transformation community) in the group of Wolfgang Thomas by Radmacher et al. [RT08], Gross et al. [GRT10], Klein et al. [KRT12], Gruner et al. [GRT13], presenting detailed complexity results for specific graph transformation games whose players have asymmetric roles (routing on a network vs. changing network structure).



In this example we have shown the computation of a wlproc construction step by step. The whole program is a $(-)$-iteration of a sequence of two subprograms, the first controlled by the system and the second by the environment. In this example, we have just represented the preconditions with their types and not the morphisms y^+, y^-, as one of the two player interfaces is always empty and it can be seen from the program what y^+ and y^- are. We further verify in the same way that

$$\mathrm{wlproc}(\neg\exists(\overset{d}{\bullet})) = \exists(\overset{a}{\bullet}) \wedge \neg\exists(\overset{d}{\bullet}) \qquad \mathrm{wlproc}(\exists(\overset{a}{\bullet})) = \exists(\overset{a}{\bullet})$$

and therefore there is an invariant for the iterated program, that can be ensured by correct choice of rule by *Sys*:

$$\exists(\overset{a}{\bullet}) \wedge \neg\exists(\overset{d}{\bullet}) \wedge \neg\exists(\overset{c}{\bullet})$$

Figure 6.5.: An example with labelled discrete graphs.

7. Applications

KRYTEN: *Let me think of a cogent paradigm, Sir.*
CAT: *I'd rather have a good example.*
— Red Dwarf, Series VII (Nanarchy)

Contents

This chapter contains a series of basic examples to demonstrate how our method works. It is structured as follows. Section 7.1 is an example outlining all steps in the verification of a small program. The focus is on the weakest precondition and only a selected branch of the proof search is shown because the whole process is somewhat repetitive. Section 7.2 demonstrates the power of $\mathcal{K}_\mu$ to prove the equivalence of various path conditions. Detailed discussion of full $\mathcal{K}_\mu$-proofs can be found in that section and the corresponding appendix. Section 7.3 concludes with an outlook.

Only the plain notion of correctness from Chapter 5 rather than the two-player variant is used in this chapter, because the procedure and all the essential features of the method are the same whether using the basic definitions or the extended, two-player ones.

The weakest preconditions and rule applications have been computed using our graph conditions toolkit, which is available under the GNU General Public License v3. Features implemented include an example generator for graph constructions, satisfaction of μ-conditions, the weakest precondition calculus for μ-conditions and graph programs (also two-player) and computer-aided application of $\mathcal{K}_\mu$ proof steps.

7.1. Simple Expanding Network

Our first example is a sanity check to show that our method can prove the preservation of connectivity under repeated applications of an appropriate graph transformation rule. The example program Expand expands edges of a graph that represents a network, perhaps a very simple model of a railroad network where the edges represent track segments. It is the iteration P (Figure 7.1). P itself removes a single track (edge) between two waypoints (nodes) and glues in a new waypoint and two new tracks. Expand $= P^*$ repeats this operation for an indefinite number of steps.

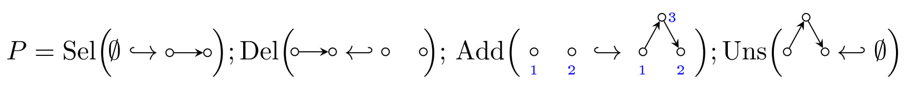


Figure 7.1.: A program Expand for expanding an edge.

As pre- and postcondition, we consider directed connectedness (condition conn:) the existence of a path between any pair of nodes (Figure 7.2). The goal is to show that Expand is correct relative to (conn, conn). Intuitively, this statement holds. In the following, we apply our methods and compute the weakest precondition (the illustrations are in Appendix B. Then, we must establish the desired invariant by proving the implication conn $=$ Wlp(P, conn).

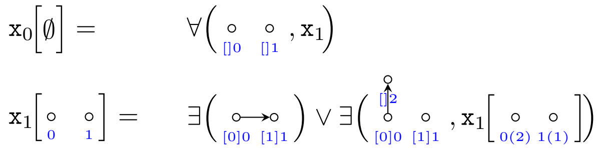


Figure 7.2.: The pre- and postcondition conn: directed connectedness.

Notation. *In this whole chapter, the use of the cumbersome sequent notation, which is relevant only for the formal definition of Rule* Empty*, is avoided. Instead, all proofs are presented as sequences of logical inferences whose soundness follows from the soundness of* $\mathcal{K}_\mu$.

The plan of attack is always to derive the contradiction by eventually invoking Rule Empty. Recall that invoking Rule Empty requires identifying a suitable set of expressions $\mathcal{H}_i$ depending on new variables, which are themselves defined in terms of $\vec{\mathtt{x}}$ via equations $\vec{\mathcal{G}}$. The proof structure is not unlike a *tableau*, unfolding branches until a promising choice of $\mathcal{H}_i$ and $\vec{\mathcal{G}}$ becomes apparent. Much like a tableau proof, the prover seeks to *close* all branches, by reducing them – not directly to contradictions, but to versions of each other's root expressions with strictly lower annotations, and Rule Empty then allows to see the desired contradiction by well-foundedness. A selected part of the proof is presented and discussed in Section C.1 in the appendix.

7.2. Path Assertions

The existence of a path between two nodes is perhaps the simplest conceivable example of a non-local graph condition. We have seen how to express it recursively as a μ-condition. The variant first mentioned in Fact 3 and used throughout this text as an example is by no means the only possibility. The condition shown in Figure 7.3, here called π, is merely the usual path condition after partial shift: while one can also prove this fact using $\mathcal{K}_\mu$, let it suffice to say that by Lemma 15, π is equivalent to the originally defined path condition. The difference is that both the first and the last node are never unselected, whereas in the original condition only the last node is never unselected. But it is intuitively clear that the existence of a path can be expressed in widely disparate ways. For example, one may just as well trace the path backwards, starting at the second node (condition τ, Figure 7.6).

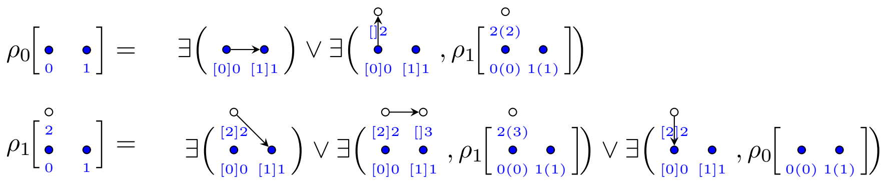


Figure 7.3.: Path condition π with fixed external interface.

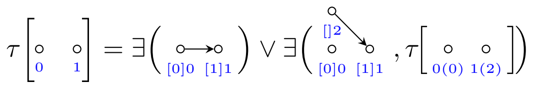


Figure 7.4.: Path condition τ, starting at the last node.

The question of this section is:

$\boxed{\text{Can } \mathcal{K}_\mu \text{ prove the equivalence of the syntactically distinct path conditions } \pi,\ \tau?}$

Answering this question would help gauge the power of $\mathcal{K}_\mu$. The interest of the question is independent from the plausibility of such a proof obligation coming up in a real correctness proof (which is hard to foretell).

7.2.1. A Proof of Equivalence

Before attempting to prove the equivalence of π and τ, consider the condition Figure 7.5. It is a slightly modified variant of Figure 7.3 where the first node is not used twice in a path. Is $\mathcal{K}_\mu$ able to prove *this* equivalence?

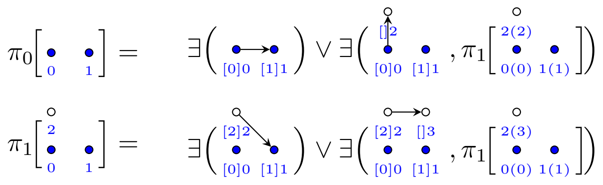


Figure 7.5.: Path condition ρ with fixed external interface, never re-using the first node.

$\rho \Leftrightarrow \pi$ is established by deriving $\bot$ from the μ-conditions $\pi \wedge \neg\rho$ and $\rho \wedge \neg\pi$ (complete, detailed, commented proofs in Subsection C.2.1 in the appendix). As in the previous section, the keystone of the proof is a single application of Rule Empty.

7.2.2. Forwards, Backwards and Tree Recursive Paths

Another possibility for expressing the existence of a path is, as suggested at the beginning of this section, to build it backwards. Yet another possibility is to opt for a tree recursive formulation (Figure 7.6, Figure 7.7)).

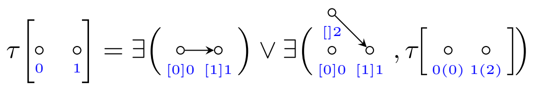


Figure 7.6.: Path condition τ, starting at the last node.

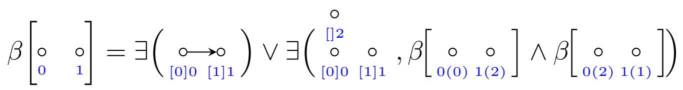


Figure 7.7.: Tree recursive path condition β.

It is intuitively clear that these conditions all express the same property. Can $\mathcal{K}_\mu$ derive the equivalences? The plan for the remainder of this section is to prove the equivalence of π and β, from which the equivalence of τ and β will follow on grounds of symmetry, completing the picture. From now on, π will no longer denote the partially shifted condition but the original one because the proofs are shorter in that case.

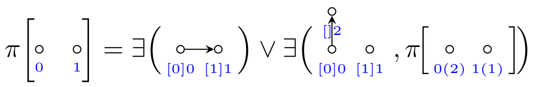


Figure 7.8.: Path condition π, starting at the first node.

The complete, detailed, commented proofs establishing that $\pi \Leftrightarrow \beta$ can be found in Subsection C.2.1. It follows by symmetry that $\tau \Leftrightarrow \beta$, because the proofs for $\pi \Rightarrow \beta$ and $\beta \Rightarrow \pi$ can be read as proofs for $\tau \Rightarrow \beta$ and $\beta \Rightarrow \tau$ respectively, by reversing all edges[1]. It follows that $\pi \Leftrightarrow \tau$.

We conclude that $\mathcal{K}_\mu$ is indeed able to prove equivalences between the disparately constructed path conditions.

7.3. Conclusion

In this chapter, we have demonstrated the effectiveness of our method on small examples: an invariant of a simple graph program and several equivalences of conditions were established. The latter means that our proof calculus $\mathcal{K}_\mu$ meets the expectation that it

[1]This should be formulated as a lemma, but the fact is immediate.

should be able to prove the equivalence of all the proposed path conditions. All proofs of this chapter were made using tool support, but still completely manually directed. The user experience would be much improved by an automatic search, which would be the most urgent future work for this part. This nonwithstanding, $\mathcal{K}_\mu$ also proved quite suitable for back-of-the-envelope reasoning, provided that one uses an abbreviated notation. and reasons at an intermediate level, for example effecting several lifts at once.

In the future, we plan to undertake larger case studies that involve realistic programs that express, for instance, distributed algorithms or concurrent access to shared data structures. An example of how to specify data structures with μ-conditions can be found in Appendix A.

Comparisons between data values are left as a future extension, since we found this aspect orthogonal to the problems addressed in this work. However, operations on data such as said value comparisons could be easily implemented by using a graph attribution concept such as [PH16] (which was designed with the aim of preserving much of the theory of ordinary graph transformation), since the theory of graph transformations and graph conditions applies generically to a wide range of structures beyond labelled graphs. It would then be sufficient to work in a suitable category of attributed graphs and straightforwardly add proof rules dealing with the operations of the attribute algebra. Thus in a hybrid proof the graph-like aspects would be covered by the $\mathcal{K}_\mu$ rules while operations on the attributes would be covered by supplementary rules specific to the attribute algebra.

8. Conclusion and Outlook

'And in case it crossed your mind to wonder, as I can see how it possibly might, I am completely sane. Which is why I call myself Wonko the Sane, just to reassure people on this point.'
— Douglas Adams, *So Long and Thanks for All the Fish*

Contents

We have examined multiple aspects of formally proving the correctness of graph programs under adverse conditions, concentrating on a proof-based approach based on recursively nested graph conditions.

8.1. Summary

Goals that have been achieved:

- A novel formalism of recursively nested graph conditions, short μ-conditions (Chapter 3), which is conceptually simple yet expressive enough to cover non-local properties of practical relevance, such as the balancedness of a binary tree (Figure A.3).
- A weakest precondition construction for μ-conditions, which is complete in the absence of iteration (Theorem 3).
- A sound proof calculus $\mathcal{K}_\mu$ for μ-conditions (Chapter 5).
- A demonstration that the framework can be used for correctness under adverse conditions: formulation of (one variant of) adversity for safety properties (Chapter 6).
- A detailed comparison of μ-conditions to related formalisms in terms of expressiveness.
- Some decidability results for structure-changing workflow nets (Chapter 4), which can be regarded as a restricted class of graph programs.

The choice to forgo the more general formulation of the results based on the axiomatic framework of adhesive categories, sticking to graphs instead, is mostly a matter of presentation. We encountered no obstacles to generalising our results to more general cases. Indeed, many proofs are already expressed in categorial language.

8.2. Future Work

Since graphs are an abstraction of data structures and graph programs model their modification by an imperative program, we envisage potential **applications** to software verification, for example in the framework of Hoare logics for a dynamically typed, object oriented programming language presented in the PhD thesis of Engelmann [Eng16]. Although the emerging standard approach would admittedly be to use separation logic [Rey02] for reasoning about pointer structures or references between objects, graph transformations have also recently been used in that domain [HPCM15].

Some ideas for future work:

- Addressing incompleteness issues. Is the restriction to expressible preconditions a problem in practice?
- Integration with Hoare logic, thus gaining flexibility and doing away with possibly lacking modularity of the Dijkstra approach noted by Poskitt [Pos13].
- Under what conditions might verification methods based on μ-conditions hold an advantage over separation logic? A question to be examined from the theoretical and empirical angle.
- Making the methods developed available as a shape analysis tool for the verification of imperative programs (software verification).
- Constructing an automatic refutation search and a search to constructively detect the existence of models. Certainly an automatic theorem prover for $\mathcal{K}_\mu$ would be a worthwhile goal in the context of the present work.
- A more thorough look into the power of the proof calculus, most likely drawing on literature on proof calculi for modal logics. Gaining an accurate idea of the power of $\mathcal{K}_\mu$ is an open challenge.
- A comparison of the expressiveness of μ-conditions and full HR* instead of HR$^-$ (Subsection 3.4.4). It is an open question whether HR* is more expressive than μ-conditions.
- Working out the generalisation to more general types of structures beyond graphs, by porting all the results to adhesive categories having the appropriate finiteness property (finite number of epimorphisms with a given domain, up to isomorphisms of the codomain), and decidability properties.
- A fuller treatment of adverse conditions, including synthesis of correct controllers, would be highly desirable.
- Reachability and liveness properties as well as safety would be highly interesting because typical requirements of systems that interact with an environment require such properties, which cannot yet be specified. On a related note, total correctness.
- There exists work on the model checking of temporal graph properties. On the other hand the modal μ calculus [BS06] defines temporal modalities using an immediate

next-step operator and least fixed points. It should be possible to develop a system that encompasses both and is endowed with a sound and useful proof calculus.

- Defining modes of interaction beyond those formulated in Chapter 6, such as asynchronous interactions with conflict and concurrency between system and environment actions, and extending the proof method to them, would be very desirable considering that concurrent semantics for graph transformations are otherwise well-established.
- The adverse conditions formulation of Chapter 6 models perfect knowledge of a global system state and omniscience on the part of the system player. From among the three aspects of adverse conditions mentioned in the introduction (limited knowledge, unpredictable behaviour, changing system environment and structure), the aspect of limited knowledge of the system state is left to future work.
- *Timed* and *probabilistic* systems could well be the subject of future investigations, as both types of behaviour constitute refinements of the discrete, nondeterministic systems considered here.

A. Expressible Conditions

In this appendix we present properties that can be expressed by μ-conditions.

The property of *being a tree* can be expressed as the conjunction of the existence of a node with no ingoing edges (the root node) from which every other node is reachable and the absence of any nodes with two ingoing edges. Clearly, this can be done with μ-conditions. π is the path condition defined as in , and the remaining constraints are just nested conditions.

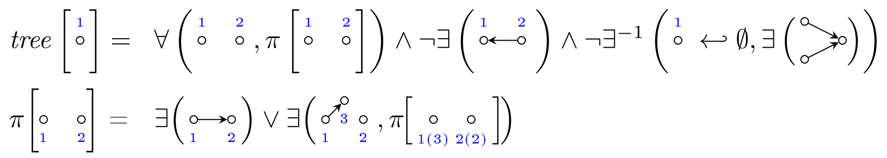


Figure A.1.: μ-Condition specifying trees.

For binary trees, the condition is adapted to labelled graphs: binary trees are encoded as graphs with edge label alphabet $\{l, \bar{l}, r, \bar{r}\}$. A node is decorated with a self-loop labelled $\bar{l}$ if it has no left child node, $\bar{r}$ if it has no right child node. Otherwise, an edge labelled l denotes a left child and r a right child. In Figure A.2, quantification over graphs with unlabelled edges should be read as an abbreviation for a disjunction over all possible instantiations with labels from $\{l, r\}$ (the disjunction is always finite because our label alphabets are always finite):

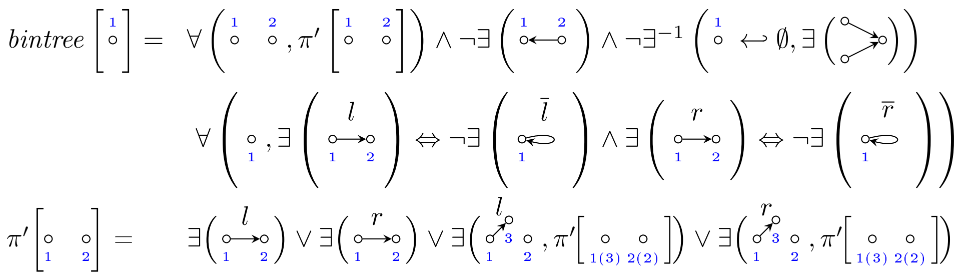


Figure A.2.: μ-Condition specifying binary trees.

Figure A.3 exhibits a graph condition that expresses an important property of binary trees. Self-balancing binary search trees are data structures that have become indispensable for many applications, see Knuth [Knu97]:

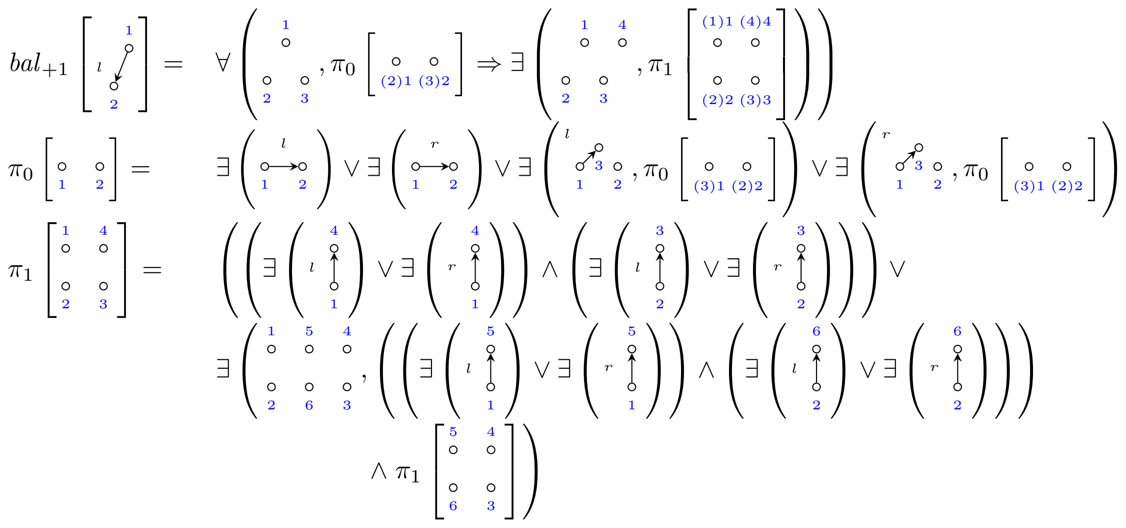
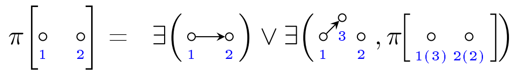
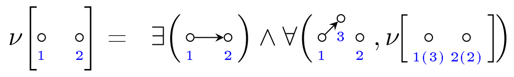
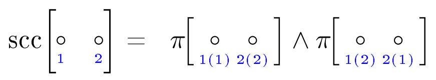
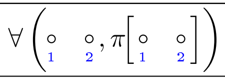
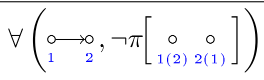
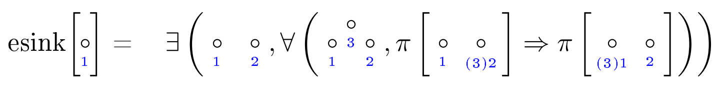
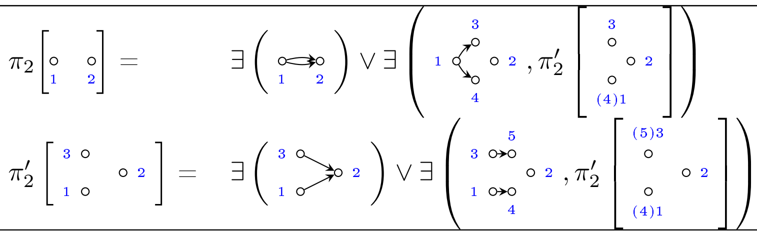


Figure A.3.: Balancedness of a binary tree.

The property *bal*$_{+1}$ (Figure A.3) says that for every path (through the tree) starting at the left child node of 1, there is a path of the same length starting from the node 1 itself. Because the successor nodes on the parallel paths are always disjoint by $\mathcal{M}$-semantics, this can only be a path through the right subtree. When universally quantified, this μ-condition constrains the depth of any left subtree of a node to be at most one more than the depth of the right subtree. Together with the symmetric condition obtained by swapping l and r, this property is *balancedness* of a binary tree. The size of the condition is considerable, which is partly due to the fact that the property is complicated. We suggest that by treating labels symbolically as in the work of Orejas et al. [OEP10], the notation could be made more concise.

A table of expressible properties is provided for perusal and reference. Definitions are re-used throughout the table (shared definitions are not re-stated every time):

Existence of a path between two given nodes 1, 2	
Each node reachable from 1 has an edge to 2	
1 and 2 lie in the same strongly connected component	
Directed connectedness	

Absence of cycles (other than loops)	
Existence of a sink node for all paths from 1	
Existence of two paths of equal length ($\mathcal{A}$-semantics!)	

It is important to understand the limitations of path constraints expressed by μ-conditions. Paths cannot be forced not to contain cycles and multiple paths cannot be forced to be disjoint by way of μ-conditions alone, except if the graph itself is constrained in such a way. Restrictions on node degrees can be used to specify graphs that not to contain any cycles or crossings of paths. Edge labels can be used to rule out only cycles or crossings on paths with certain labels.

Known limitations were noted in Section 3.4. The most important point is that any property that is not computable in polynomial time is also not expressible as a μ-condition. For a different reason, Hamiltonicity, still not known to be outside of polynomial-time, cannot be expressed either.

B. Computed Weakest Preconditions

This appendix lists the computed weakest liberal preconditions from Section 7.1. The weakest liberal precondition with respect to the Add; Del; Uns part of the program is computed from the postcondition in several intermediate steps, which are shown in Figure B.1 to Figure B.6 (including several simplifications to decrease clutter). First $\mathrm{Wlp}(\mathrm{Uns}, d)$ is computed including the partial shift, then $\mathrm{Wlp}(\mathrm{Del}; \mathrm{Uns}, d) = \mathrm{Wlp}(\mathrm{Del}, \mathrm{Wlp}(\mathrm{Uns}, d))$ and so forth. The weakest precondition $\mathrm{Wlp}(P, d)$ is shown in Figure B.7). The implication problem $c \Rightarrow \mathrm{Wlp}(P, d)$ is shown in Figure B.8 and represents the final result of the computation. If the condition of Figure B.8 can be refuted using the $\mathcal{K}_\mu$ calculus, then the program is correct with respect to the specification.

In detail, the steps are as follows:

After applying the Wlp construction including partial shift to the unselection step of the rule, the condition becomes as in Figure B.1 and Figure B.2. The resulting condition is much bulkier than d itself because the unselection step is implicitly followed by a partial shift. A partial shift of a condition c results in a condition of $O(m \times n \times o)$ variables, where m is the size of the largest graph appearing as a variable type in c, n is the size of the type of the main body of c and o is the number of variables in c. Steps other than unselect by definition never increase the number of variables. By inlining the right hand sides of variables that directly depend on just one different variable or on none at all, the number can be further reduced (this was performed in Figure B.5).

The weakest precondition with respect to the addition is in Figure B.3 and Figure B.4. This precondition can be simplified by removing the subconditions $\perp$ that sprung into existence because the construction δ' outputs $\perp$ under certain circumstances, by removing nestings with identity morphisms and by simplifying chain variables that have just one other variable on their right hand side (Figure B.5).

Figure B.6 shows the result of applying the α' transformation and modifying the main body with the deletability condition $\Delta(l) \Rightarrow$ (which turns out to be trivial, since no nodes are deleted). Finally, the selection is handled by wrapping the condition in a universal quantification of the selected morphism. The whole expression was further simplified by removing redundant variables Figure B.7. The next step after having computed the weakest precondition $\mathrm{Wlp}(P, d)$ is to attack the implication problem Figure B.8 using $\mathcal{K}_\mu$. The variables have been suitably renumbered. If d is indeed an invariant that can be proven by $\mathcal{K}_\mu$, one should be able to derive a refutation of $c \wedge \neg\mathrm{Wlp}(P, d)$.

Notation. *Recall that the small blue numbers are node identities, the numbers in square brackets fix the morphism under a quantifier by indicating the preimages, for example* *in [2]1, 1 is the node identity in the target of the morphism and 2 is its preimage in the source graph. Empty square brackets mean that a node of the target graph has no preimage. For variable types and unselections, the notation 1(2) means that the node 2 in parentheses is mapped to the node 1 in the type, respectively that the node previously known as 2 is now identified by the number 1.*

Please note that these pictures were automatically generated by a layout algorithm that does not respect the convention that the target graph of a morphism should have a similar layout as the source graph. Mappings are conveyed by node numbers only.

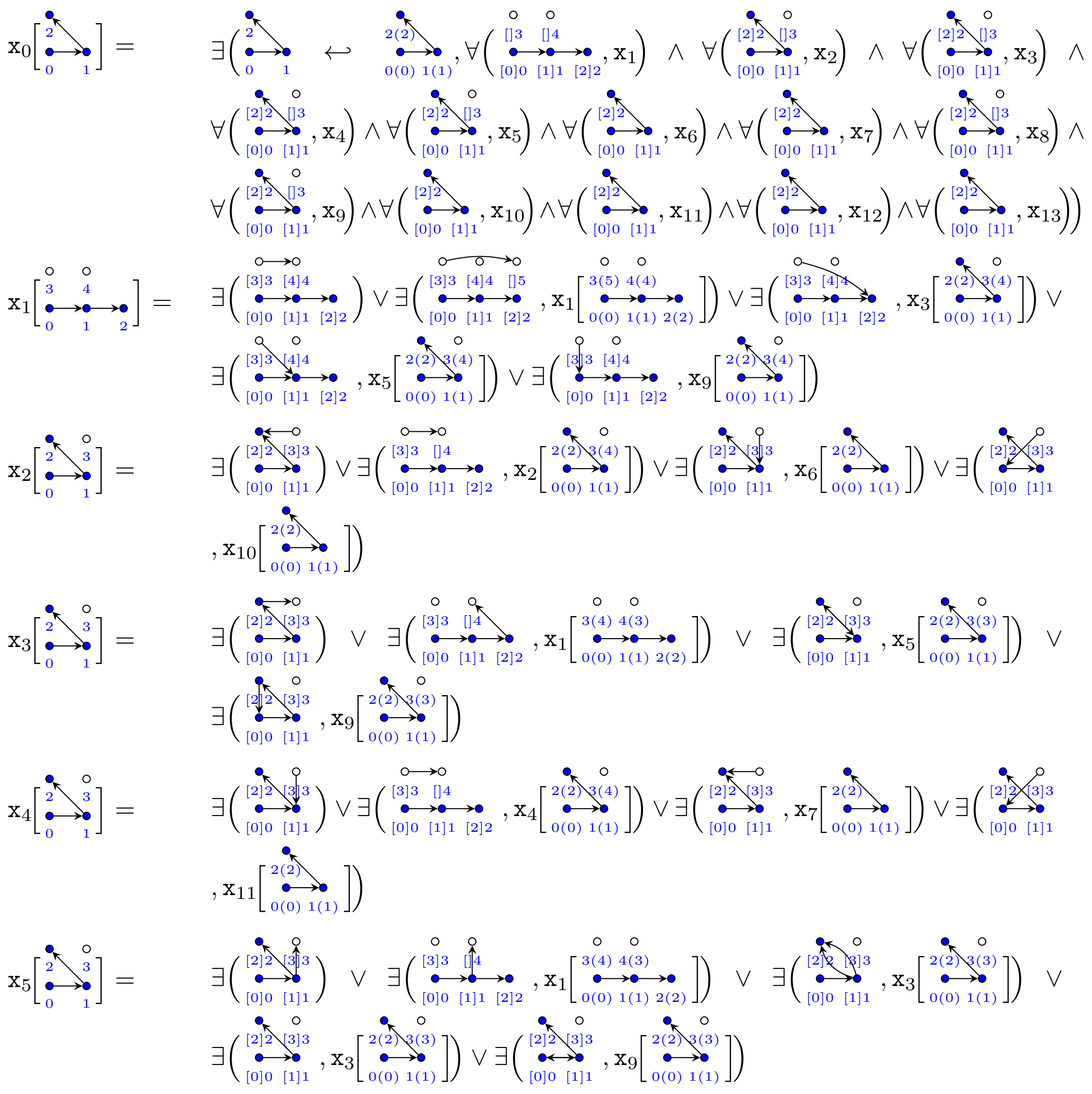


Figure B.1.: Wlp(Uns, d) (part I). The nodes of the external interface (of the main body, conventionally called x_0) are highlighted in blue.

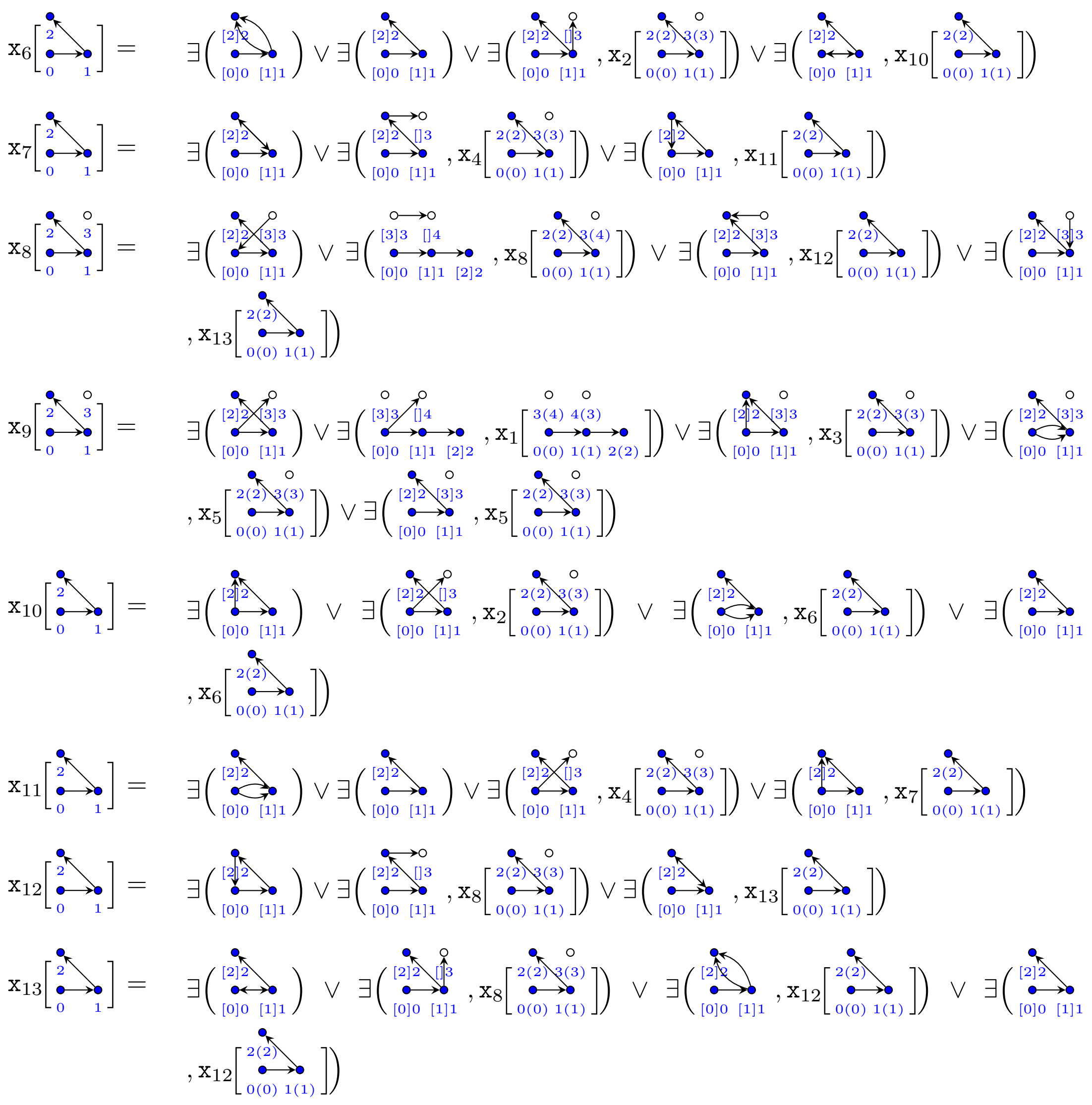


Figure B.2.: Wlp(Uns, d) (part II).

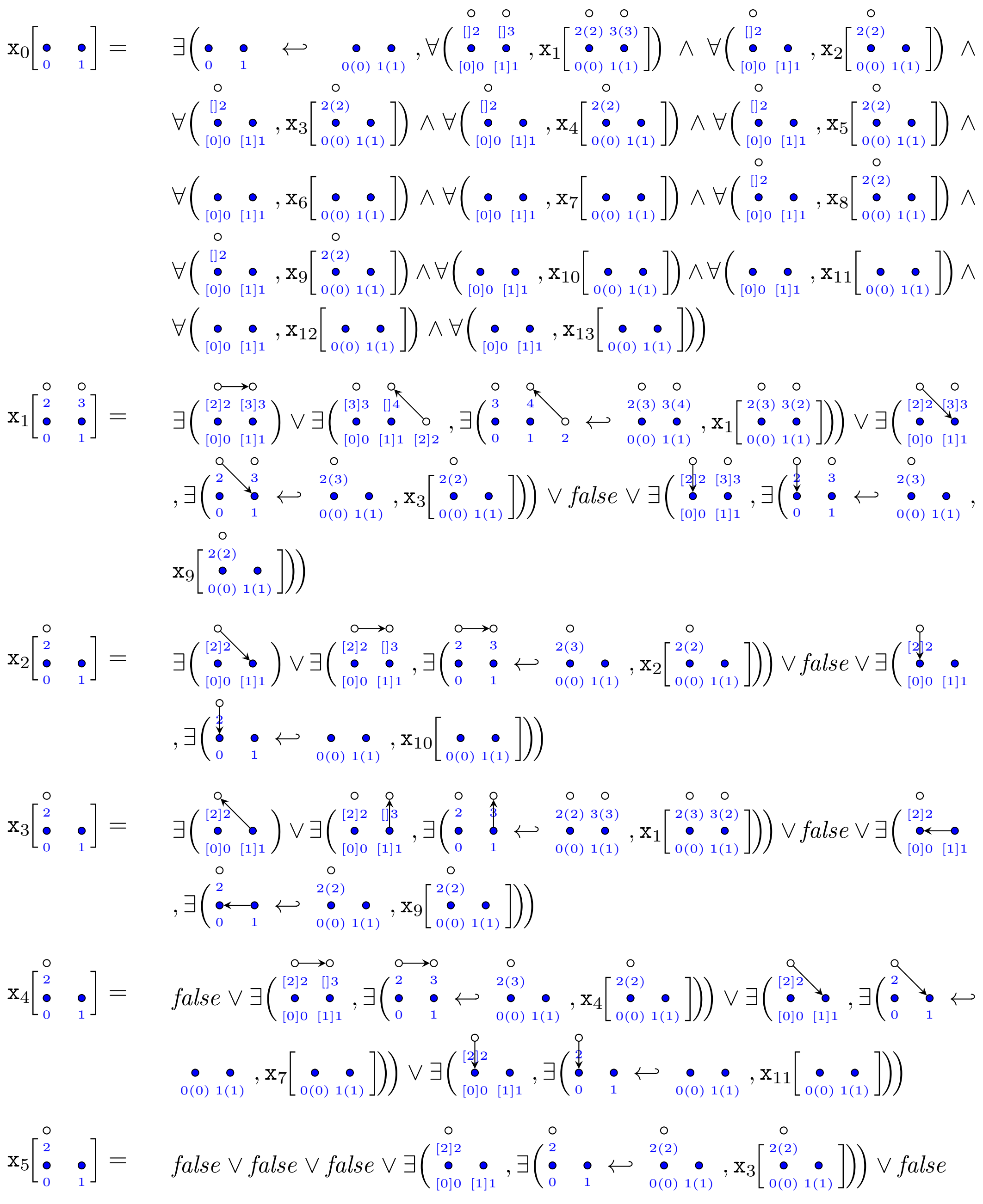


Figure B.3.: Wlp(Del; Uns, d) (part I)

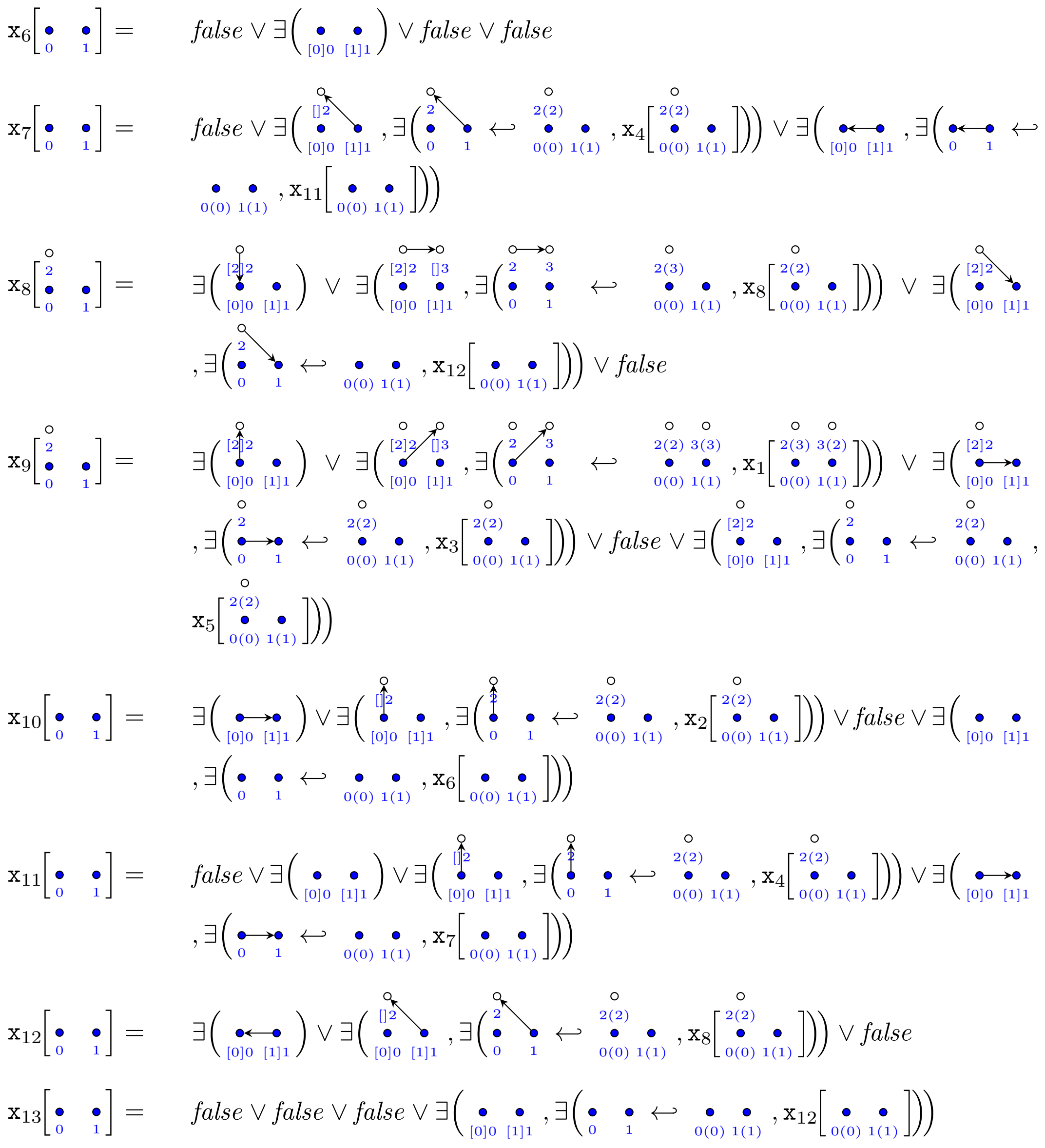


Figure B.4.: Wlp(Del; Uns, d) (part II)

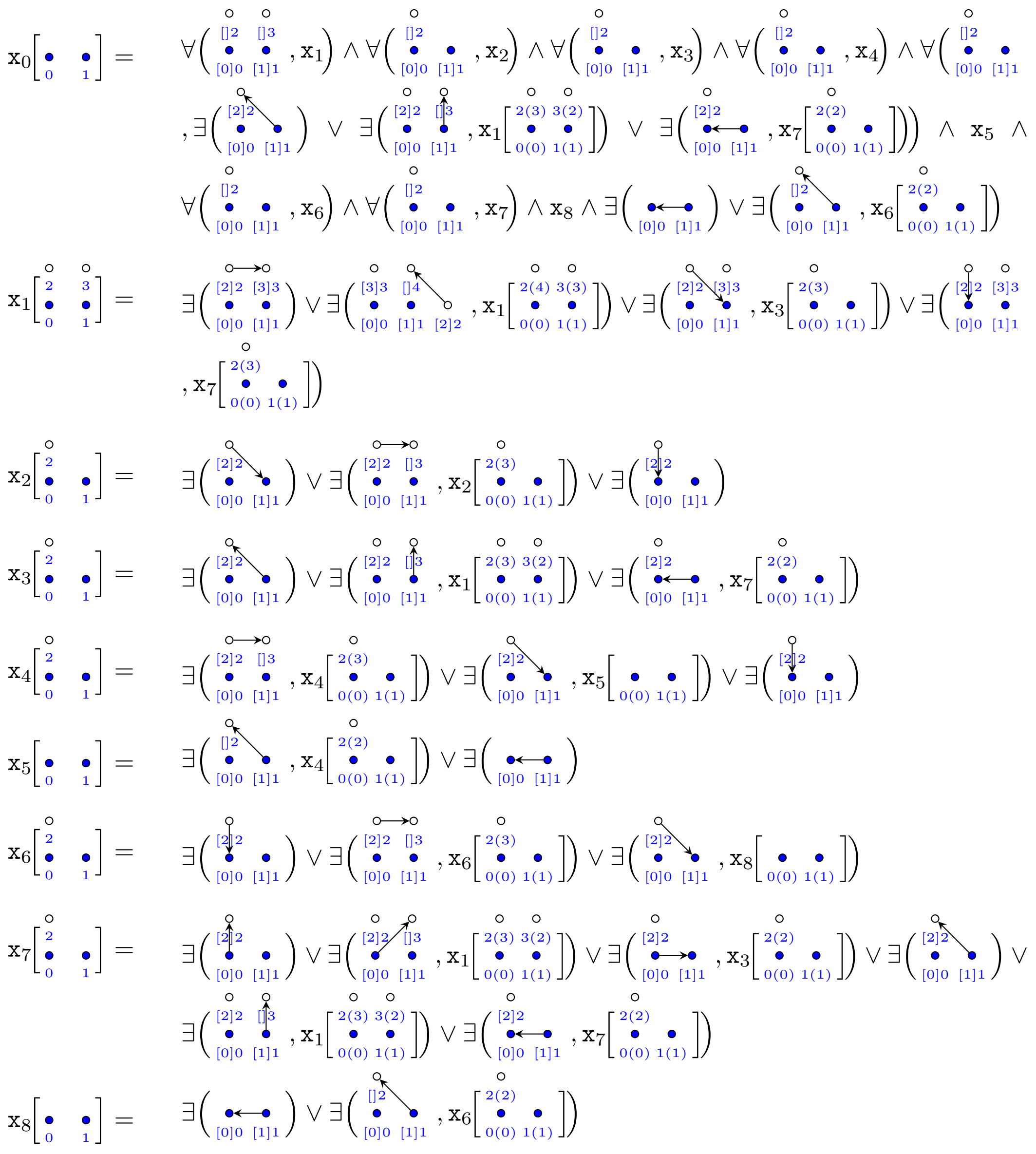


Figure B.5.: Wlp(Del; Uns, d) (simplified)

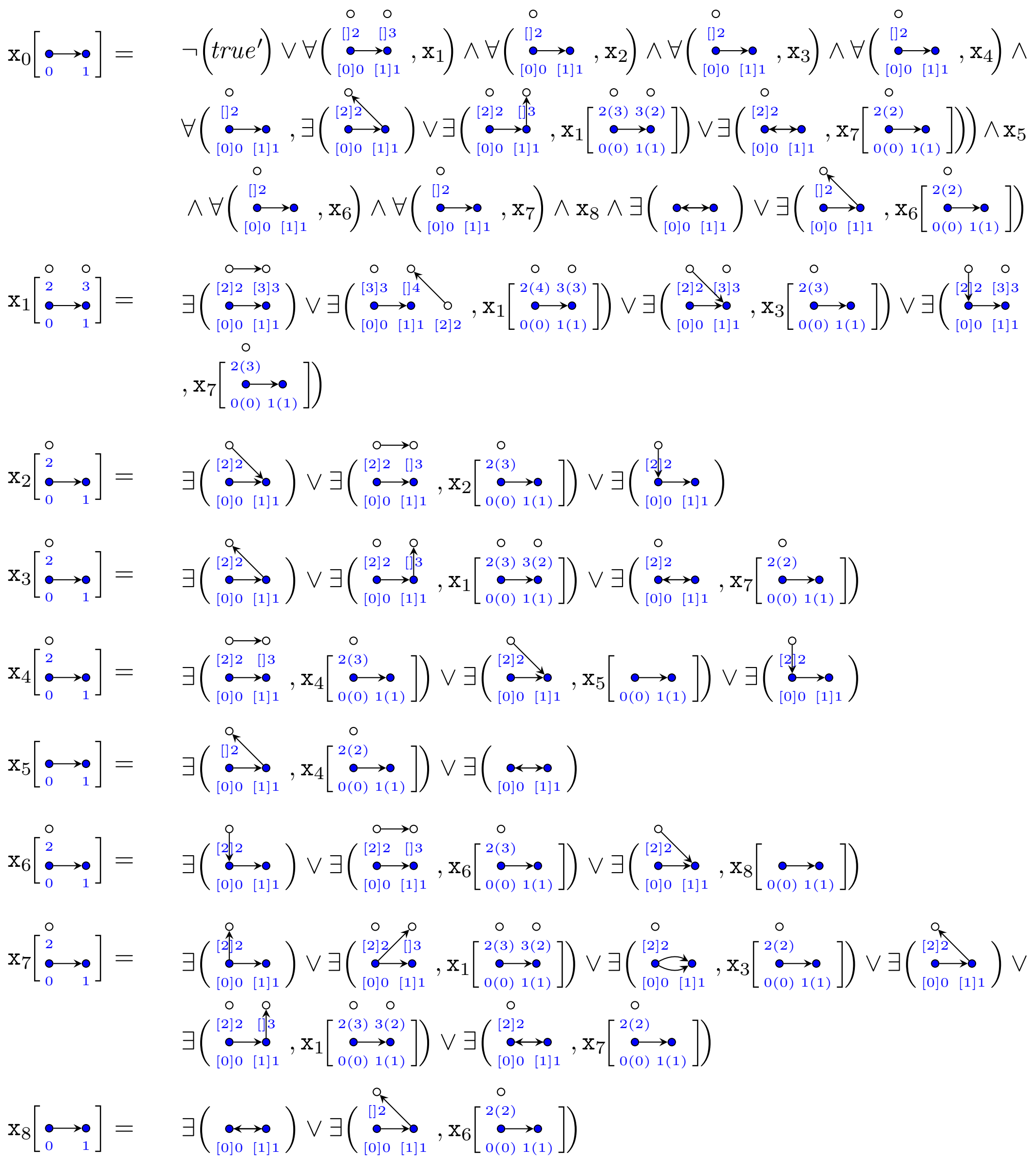


Figure B.6.: Wlp(Add; Del; Uns, d)

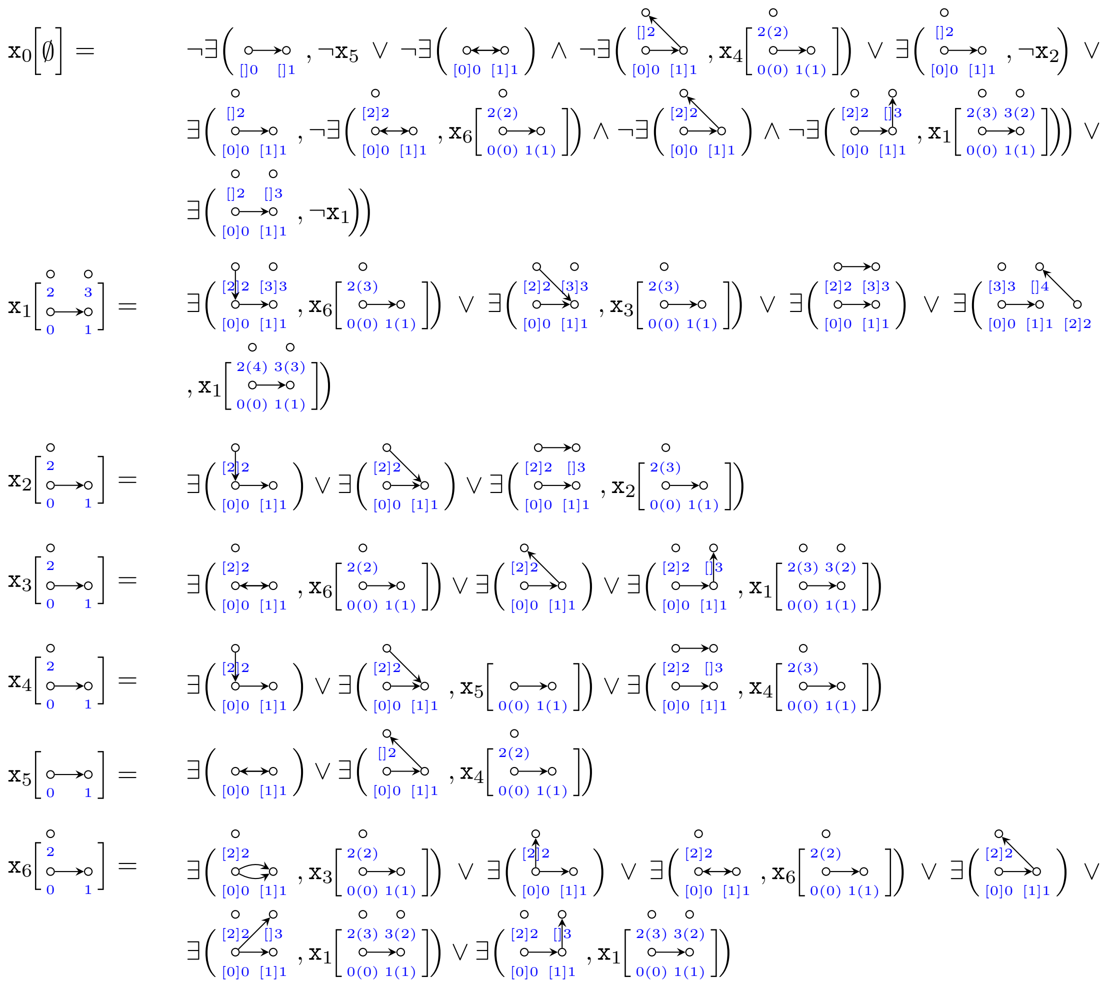


Figure B.7.: Wlp(P, d) (simplified further)

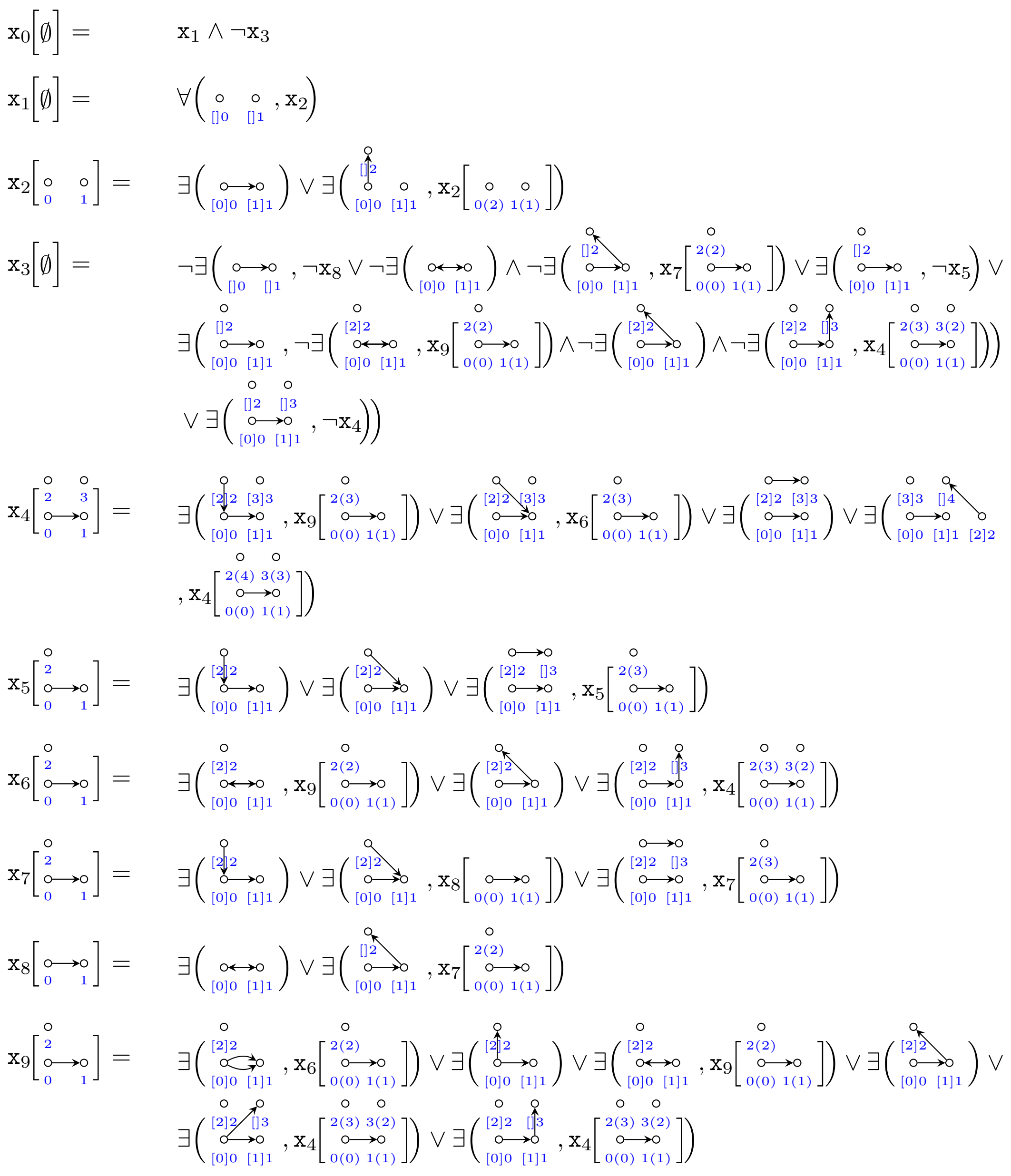


Figure B.8.: The implication problem: disprove this ($\neg(c \Rightarrow \mathrm{Wlp}(P, d))$)

C. Example Proofs

This appendix contains commented listings of the proofs from Appendix B.

C.1. Simple Expanding Network

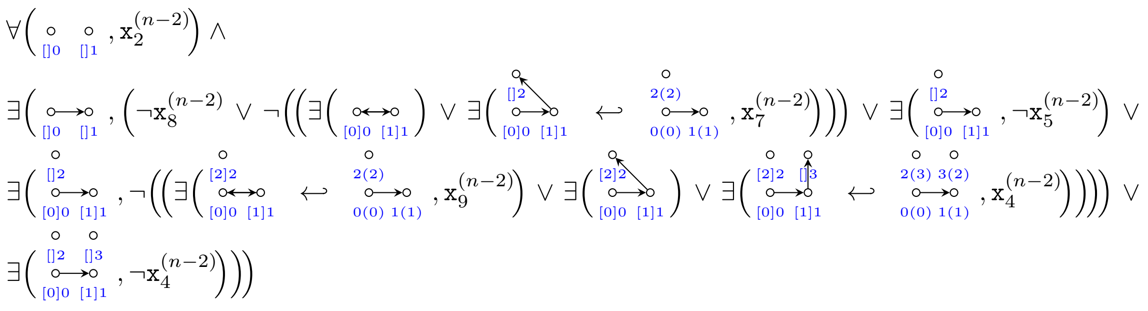
The proof obligation is to derive $\bot$ from the combination of variables, whose definitions are given in Figure B.8:

$$\mathtt{x}_1^{(n-1)} \wedge \neg\mathtt{x}_3^{(n-1)}$$

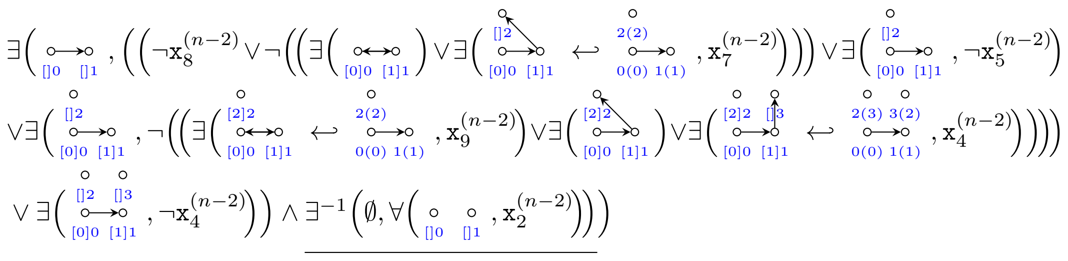
Figure C.1.: The initial proof obligation.

This is indeed possible and therefore the program provably fulfils the specification. Only a selected branch of the proof search is shown, because providing the full proof here would probably not provide any supplementary insight (see Section 7.2 for examples including full $\mathcal{K}_\mu$ proofs).

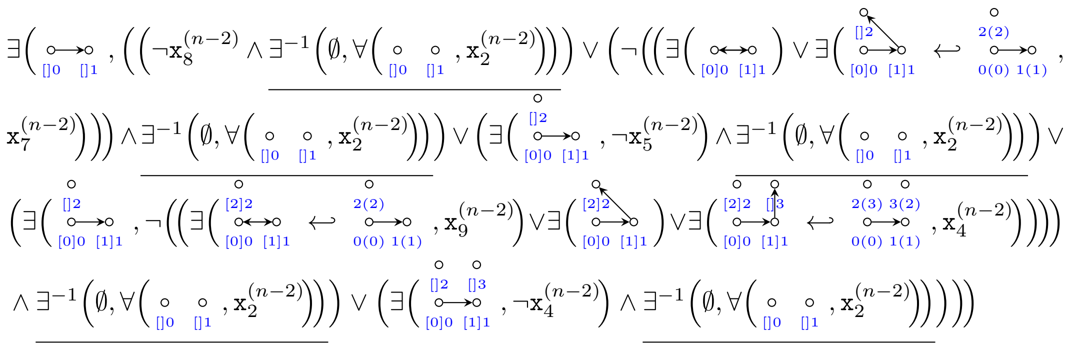
From the condition of Figure C.1, a larger expression is obtained after unrolling the definitions of the variables using Rule Unroll$_1$. It is still a conjunction of two expressions:



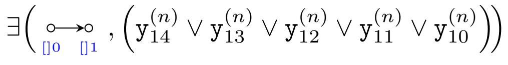
Then the first conjunct of the outer conjunction is lifted over the existential quantifier of the second conjunct, without applying the A construction (Shift). The lifted subcondition is underlined:



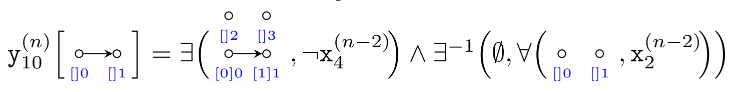
Distributivity is applied, which was not explicitly formulated as a rule, but follows from the logical rules of $\mathcal{K}_\mu$.



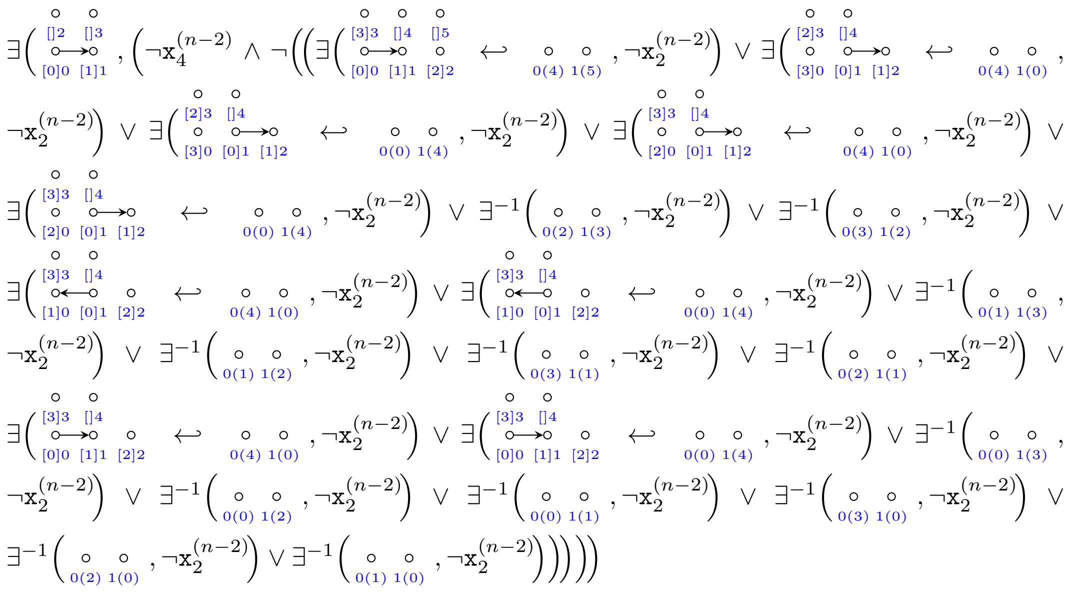
The five branches of the disjunction are named and handled separately. These named branches play the role of the $\mathcal{G}$ in Rule Empty, while the expressions respectively deduced from them by use of inference rules constitute the components of $\mathcal{H}$. **This exposition focuses on the first and last one only, which are typical. The full proof must handle all five.** We adopt the convention of naming the new variables defined by the conditions $\mathcal{G}_i(\vec{\mathtt{x}})$ with the new letter $\mathtt{y}$, so as not to confuse them with the variables of the original μ-condition:



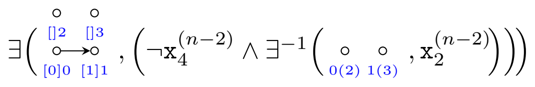
Let us examine the last branch of the disjunction.



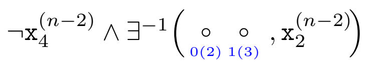
After lifting the second conjunct inside the quantifier of the first one and applying A, a great number of cases are created.



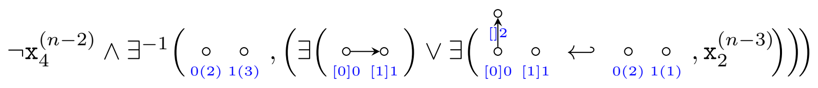
Only one of the cases created in the lift is relevant, the others are quickly discarded by logical rules (the case distinction is a disjunction under an odd number of negations, therefore subconditions which can be shown to be equivalent to ⊥ can be removed).



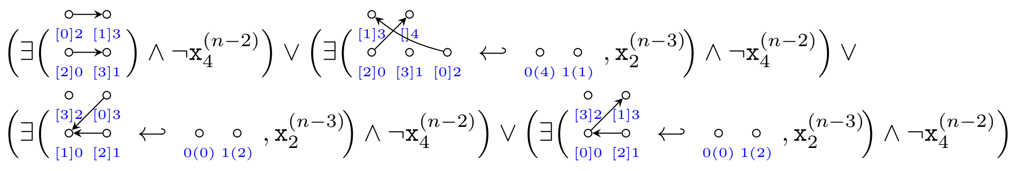
The outer nesting level will not be modified any more and we concentrate on the direct subcondition:



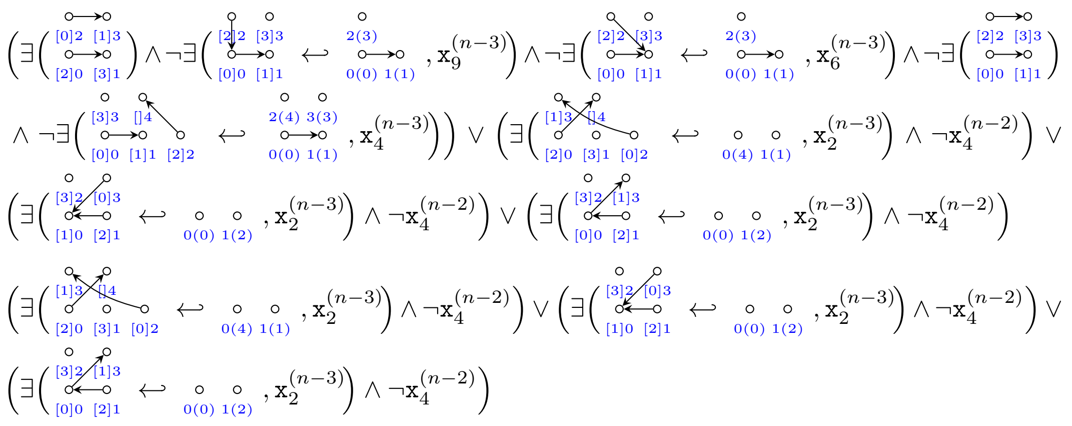
To proceed, the variable x_2 can be unrolled,



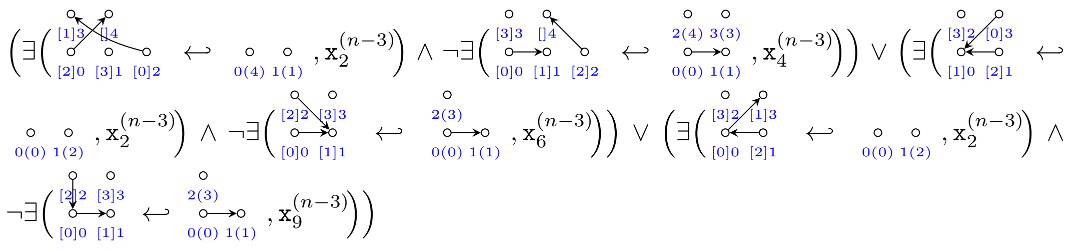
Then a shift is applied to get rid of the unselection, then the conjunction with $\neg x_4^{(n-2)}$ is distributed over the disjunction.



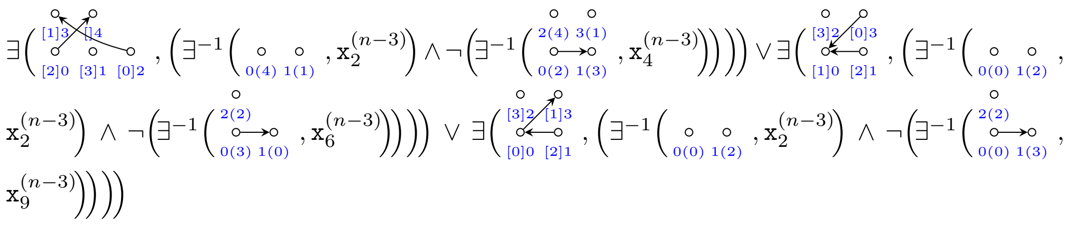
In the latter expression, the first branch of the outer disjunction created from distributivity can be seen to contain a subcondition without variables (a “base case” of a recursive condition) and its negation. This can be resolved using the rules from $\mathcal{K}$. The following two pictures show the situation before resolving the “base case” subcondition, and after thus resolving the first branch:



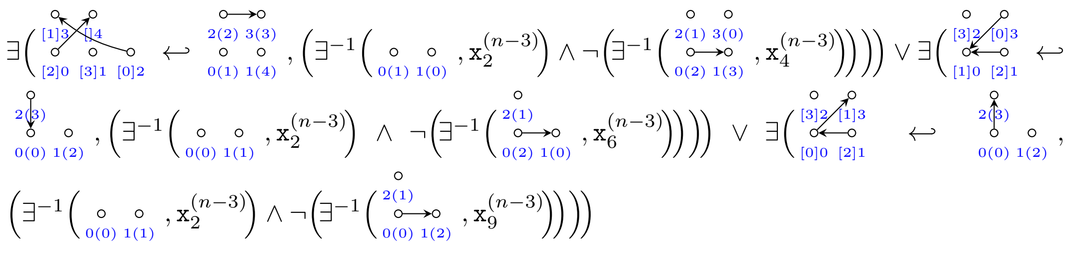
When all the occurrences of the variables x_4 are unrolled after lifting and only the relevant branches kept, the following condition results.



Further rearrangements by inference rules (lifts have been applied to gather subconditions; A has been applied and irrelevant possibilities created in the lifts have been pruned using logical rules. After lifting, unrolling x_4 and pruning irrelevant branches):



Common unselections are factored out by ABSORB:



Focusing on the first branch of the previously obtained expression by naming it (**the others have to be addressed separately** to arrive at the full proof) (Figure C.2), then unrolling the occurrence of x_2, then applying distributivity yields this:

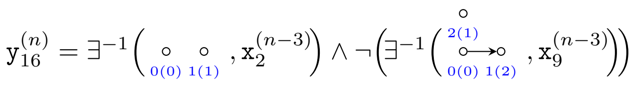
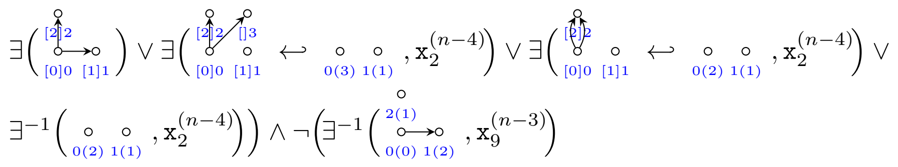
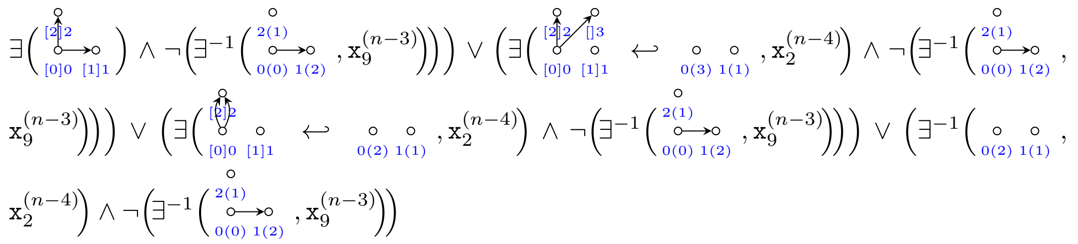
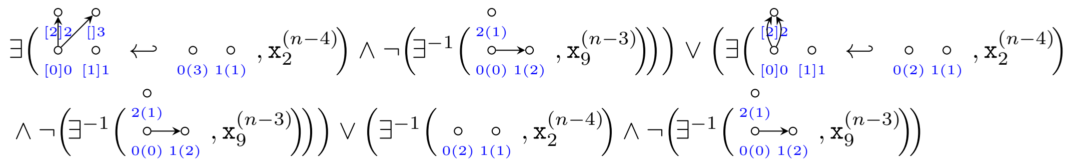


Figure C.2.: The first branch of the previously obtained consequence of $y_{10}^{(n)}$.





One branch of the condition can again be resolved by unrolling x_9, keeping only its base case which happens, after lifting over the unselection, to be the negation of the first subcondition $\exists(a, \top)$. One branch is thus resolved:



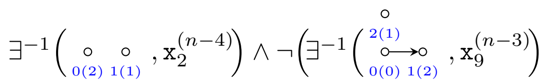
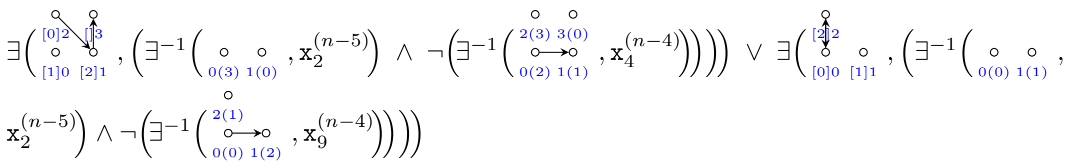


Figure C.3.: One of the remaining branches of the previous expression.

Applying several inference rules (distributivity, unrolling) to one of the remaining branches (Figure C.3):



Finally we have obtained a previously encountered expression (Figure C.2, not the similar-looking but subtly different Figure C.3!), with strictly decreased annotations. The significance of this is that $y_{15}^{(n)}$:= Figure C.2 implies some condition which has a (syntactically positive[1]) subcondition equivalent to $\mathtt{y}_{15}^{(n-1)}$.

In order to complete the proof, the same procedure must be applied to every branch until each named branch $\mathtt{y}_i = \mathcal{G}_i(\mathtt{x})$ is seen to be a monotonic combination $\mathcal{H}_i(\vec{\mathtt{y}})$ of all the other named branches. Then, if $\mathcal{H}(\vec{\bot}) = \bot$ (which can even be detected automatically in a typical proof, especially when the components $\mathcal{H}_i$ have no subcondition where no variable occurs), it follows from the soundness of Rule Empty that all of the $\mathcal{G}_i(\mathtt{x})$ are unsatisfiable in the least fixed point solution.

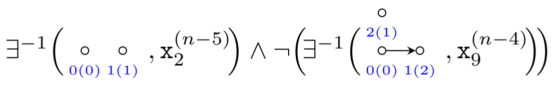


Figure C.4.: Branch done.

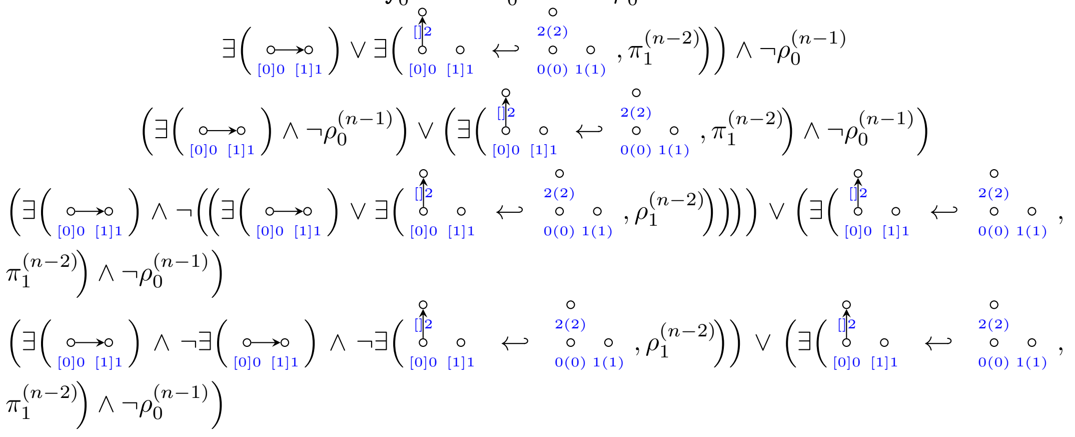
C.2. Path Assertions

This section contains the detailed proofs for showing the equivalence of the various path conditions from Section 7.2.

C.2.1. A Proof of Equivalence

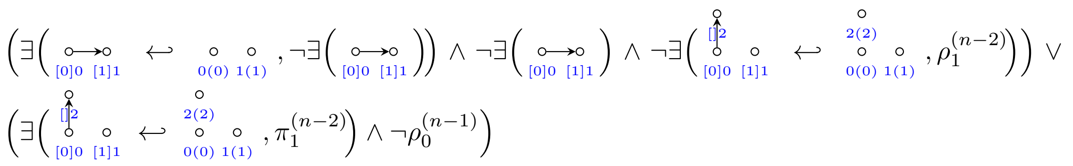
We present the proof for $\pi \Rightarrow \rho$, then $\rho \Rightarrow \pi$. First $\pi \Rightarrow \rho$: in the following list of steps, several rules are applied to the main body of the condition: the sole occurrence of π_0 is unrolled, distributivity is applied, a subcondition $\neg\rho_0$ is unrolled and transformed to a conjunction. To recapitulate: unroll π_0, distributivity, unroll one occurrence of ρ_0, apply De Morgan's rule:

$$\mathtt{y}_0^{(n-1)} = \pi_0^{(n-1)} \wedge \neg\rho_0^{(n-1)}$$

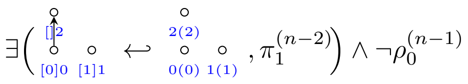


[1]As one may check because all branches that were followed up appeared under zero negations.

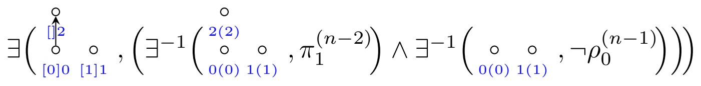
At this point, the whole condition is a disjunction of two members. One of these contains a subcondition with no variables in conjunction with its negation. These can be resolved to $\bot$, eliminating one branch of the disjunction:

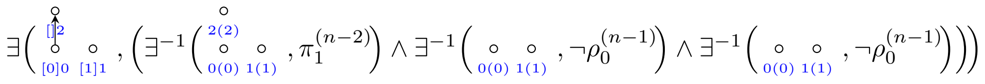


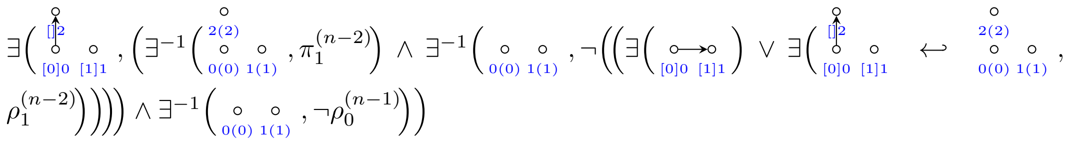
The first conjunct, after effecting the shift (construction A), resolves by PartialResolve, leaving this behind:



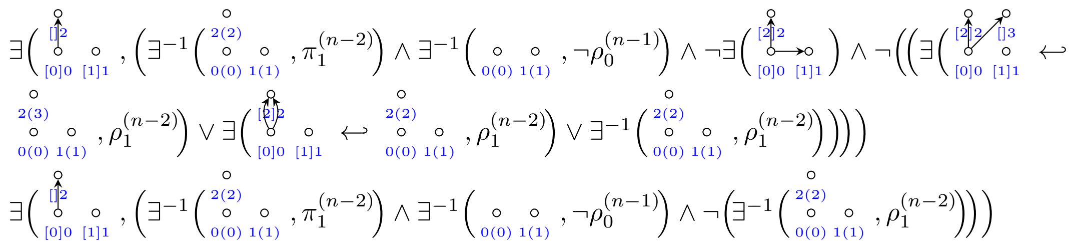
Next, the second conjunct is lifted into the quantifier of the first one, then duplicated for later use, then one occurrence of ρ_0 is unrolled.



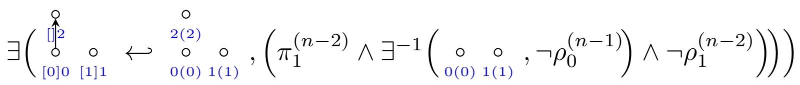


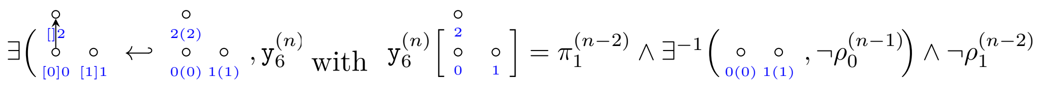


A shift is applied and unwanted conjuncts removed.

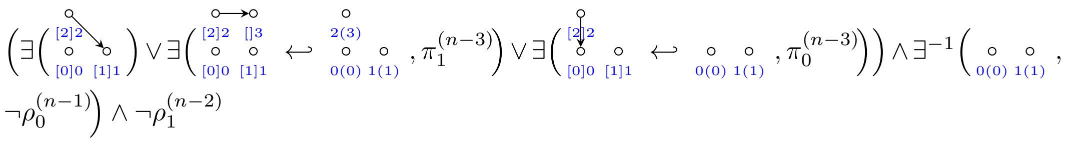


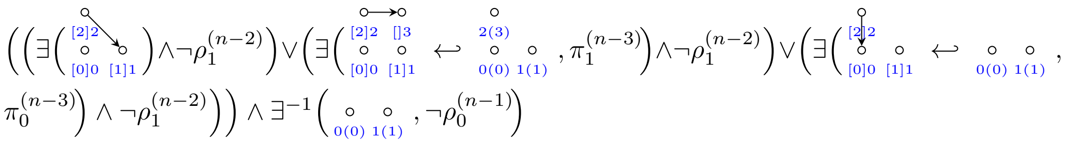
In this step, a common unselection is factored out (Absorbed, in fact) and the resulting subcondition is given a name. Effectively a new variable is introduced.

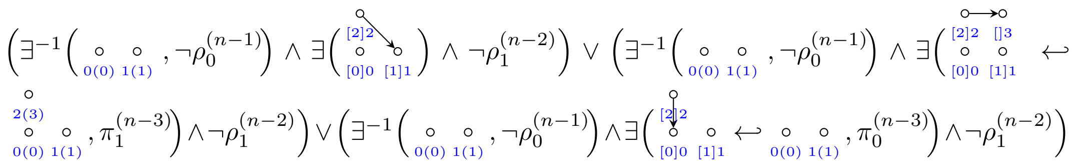




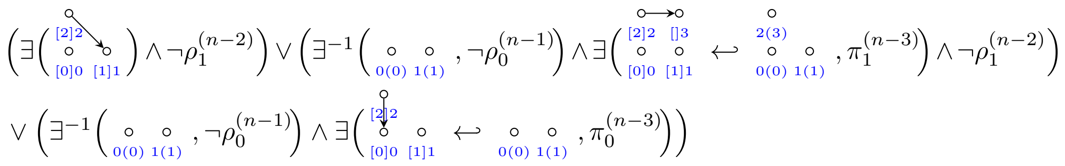
We continue working on the new named subcondition, unrolling π_1, then applying distributivity, then also applying distributivity to the $\exists(\iota, \neg\rho_0^{(n-1)})$ subcondition.

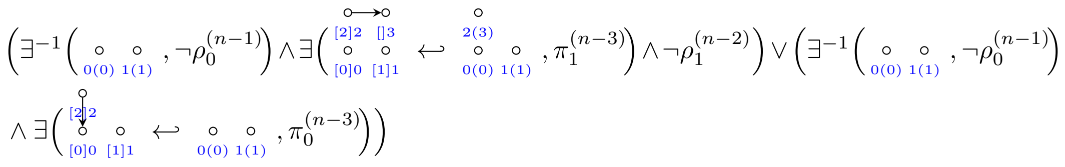




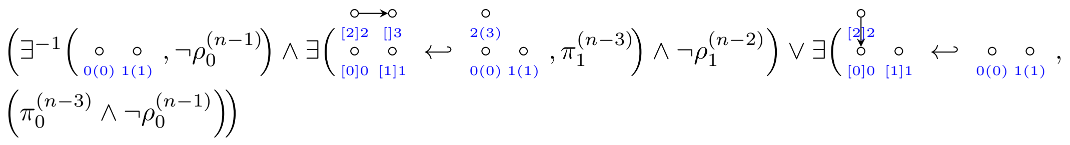


At every conjunct, we keep only what is relevant. Then we eliminate the "base case" in the first conjunct by unrolling that occurrence ρ_1 and cancelling, as described before in similar cases:

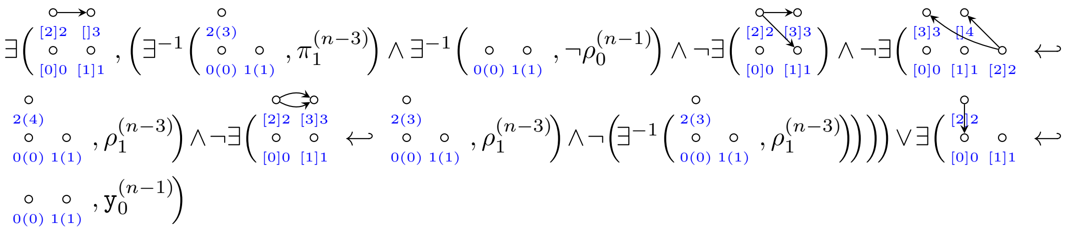




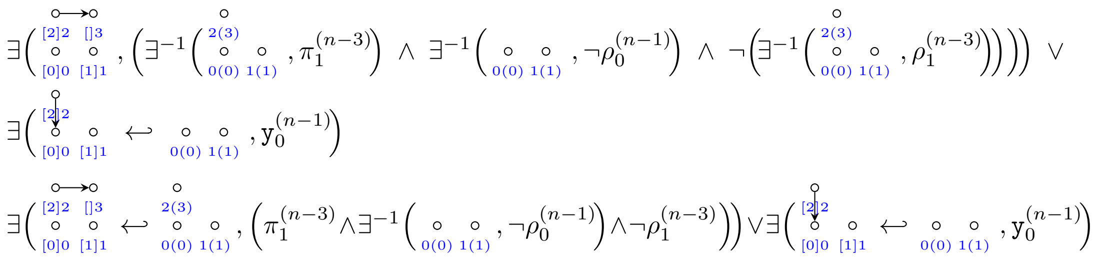
The latter of the two remaining conjuncts is treated with a Lift.



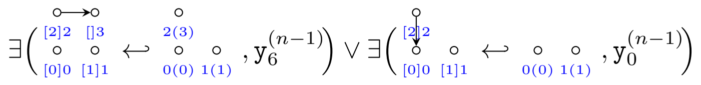
The previous step resulted in a subcondition that matches with y_0, with decreased indices. This situtation is desirable, as it leads towards the applicability of Rule Empty.



Eliminating conjuncts that are irrelevant to the situation at hand, and factoring out a common unselection:

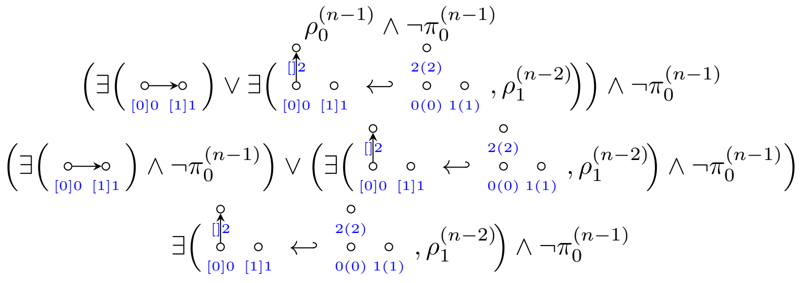


Matching the previously named y_6 completes this branch of the proof:

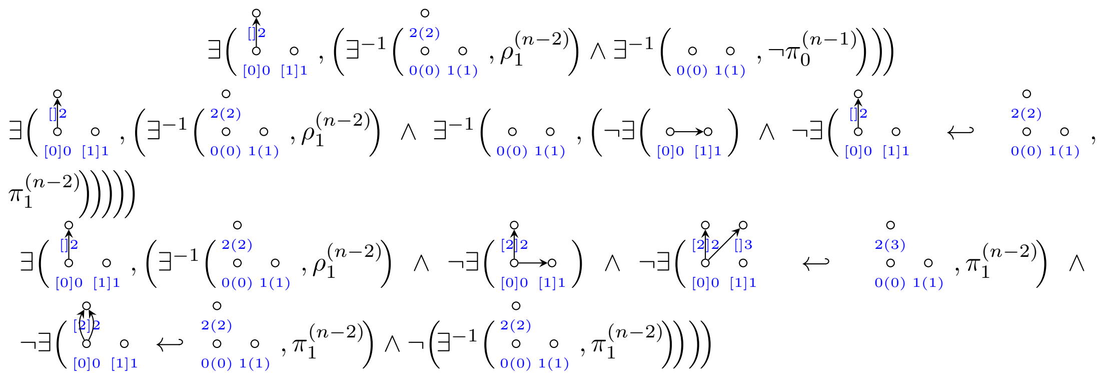


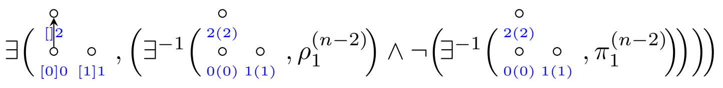
At this moment, one checks that both y_0 and y_6 are now positive combinations of y_0 and y_6, which can be checked by substitution of $\bot$ to imply $\bot$ in the least fixed point solution. Since y_0 is by definition equivalent to the main body of the condition to be refuted, this has been shown to imply $\bot$, hence the refutation was successful and $\pi \Rightarrow \rho$ is established.

Let us now prove the other direction, $\rho \Rightarrow \pi$, which requires fewer steps. The beginning is analogous to our proof of the direction $\pi \Rightarrow \rho$. The remainder of the proof is simpler because for this direction there is no need to "presciently" duplicate a subcondition at this point.

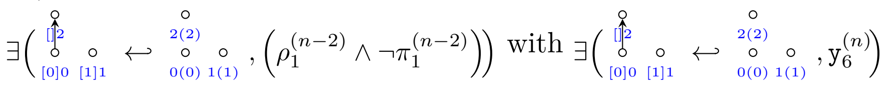


Now we lift, unroll π_0, apply a shift and discard the unwanted conjuncts, which in this case again happen to be those that do not perform a maximal amount of node identification.

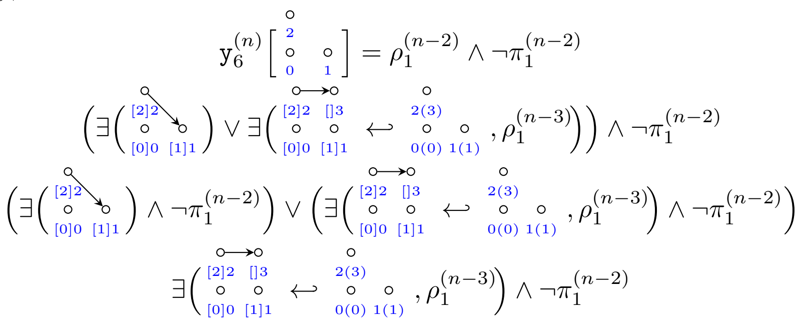




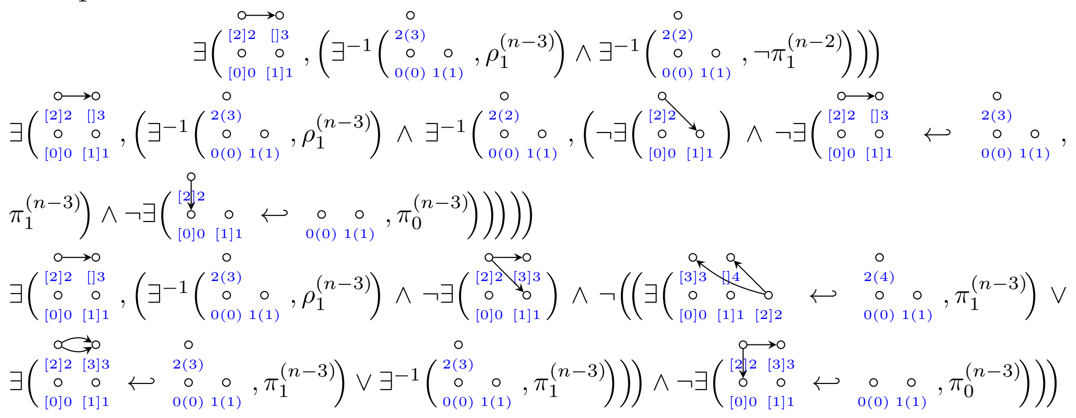
Again, we factor out common unselection and name the resulting subcondition. This decision is prompted by the possibility of creating a conjunction of “naked” occurrences of ρ and $\neg\pi$, which looks like a decent candidate to be used in Rule Empty.



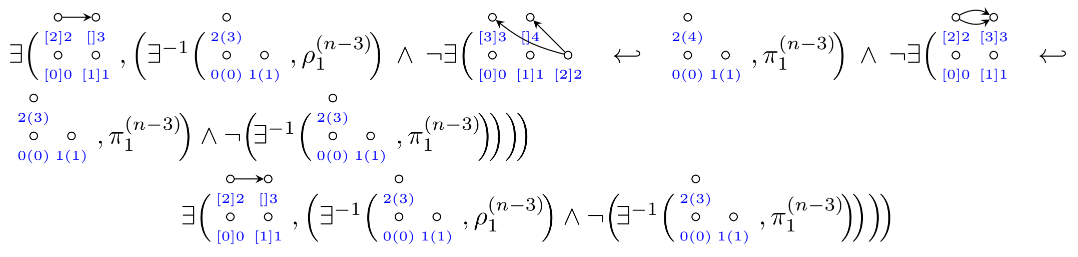
Having named y_6, it is necessary to keep working on it in the usual fashion: unroll, apply distributivity, eliminate the base case.

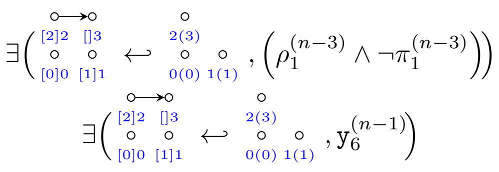


The $\neg\pi_1^{(n-2)}$ subcondition is lifted inside the quantifier, unrolled and a shift is applied.



After eliminating unneeded conjuncts and factoring out a common unselection, y_6 is matched with decreased annotations, concluding the proof since now both y_0 and y_6 are seen to imply positive combinations $\vec{\mathcal{H}}$ of y_0 and y_6 with $\vec{\mathcal{H}}(\vec{\bot}) = \vec{\bot}$.

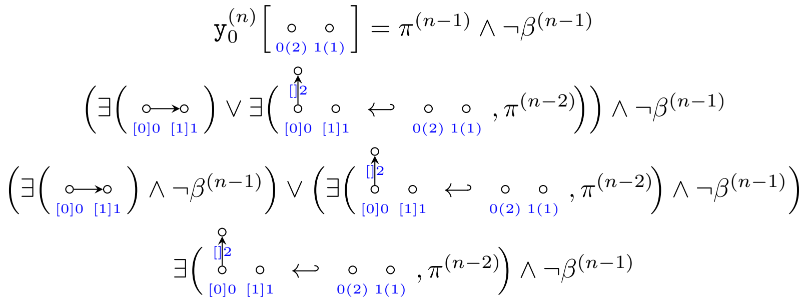




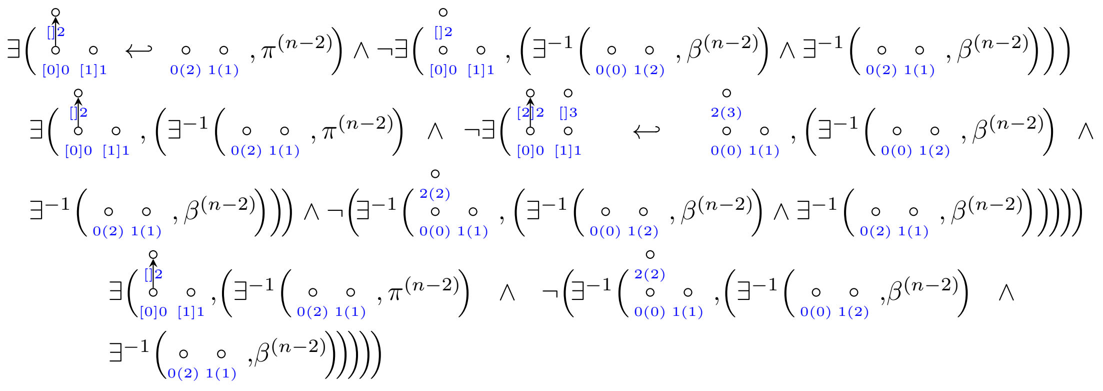
$\rho \Rightarrow \pi$ is established.

C.2.2. Forwards, Backwards and Tree Recursive Paths

First, $\pi \Rightarrow \beta$. This direction is straightforward using the experience from the previous proofs. The beginning is analogous to the previous proofs: the variable for the condition π that appears positively is unrolled once and its base case eliminated.



Next, β is unrolled. Its base case is of no use here, so it is simply dropped from the conjunction. Then the subcondition issued from the right hand side of β is lifted over the quantifier and shifted. One of the conjuncts resulting from the shift, positing the existence of a fourth node, does not appear to lead anywhere and is dropped. The useful case is that with only three nodes, where the "middle" node (which appears with number 2 in Figure 7.7) is identified with the target of the edge in the right hand side of π.



After some simplification of the unselections and manipulations of the Boolean structure to gather the cases (shown below), the first subcondition is seen to resolve against the existence of the edge in the outer quantifier: after unrolling the first occurrence of β, the base case is kept and used in the resolution.

$$\exists\Big(\underset{[0]0}{\overset{\overset{\circ}{\uparrow{\scriptstyle []2}}}{\circ}}\;\underset{[1]1}{\circ}\;,\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\pi^{(n-2)}\Big)\wedge\neg\Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}\;,\beta^{(n-2)}\Big)\wedge\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\Big)\Big)$$

$$\exists\Big(\underset{[0]0}{\overset{\overset{\circ}{\uparrow{\scriptstyle []2}}}{\circ}}\;\underset{[1]1}{\circ}\;,\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\pi^{(n-2)}\Big)\wedge\Big(\neg\Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\vee\neg\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\Big)\Big)\Big)$$

$$\exists\Big(\underset{[0]0}{\overset{\overset{\circ}{\uparrow{\scriptstyle []2}}}{\circ}}\;\underset{[1]1}{\circ}\;,\Big(\Big(\neg\Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\wedge\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\pi^{(n-2)}\Big)\Big)\vee\Big(\neg\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\wedge$$
$$\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\pi^{(n-2)}\Big)\Big)\Big)\Big)$$

The gathering of unselections (intermediary steps implicit between the first and second line: after unrolling the first β and discarding all but the base case, resolution using rules from $\mathcal{K}$ can ensue:

$$\exists\Big(\underset{[0]0}{\overset{\overset{\circ}{\uparrow{\scriptstyle []2}}}{\circ}}\;\underset{[1]1}{\circ}\;,\Big(\neg\Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\vee\Big(\neg\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\wedge\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\pi^{(n-2)}\Big)\Big)\Big)\Big)$$

$$\exists\Big(\underset{[0]0}{\overset{\overset{\circ}{\uparrow{\scriptstyle []2}}}{\circ}}\;\underset{[1]1}{\circ}\;,\Big(\neg\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\beta^{(n-2)}\Big)\Big)\wedge\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\pi^{(n-2)}\Big)\Big)\Big)$$

$$\exists\Big(\underset{[0]0}{\overset{\overset{\circ}{\uparrow{\scriptstyle []2}}}{\circ}}\;\underset{[1]1}{\circ}\;\hookleftarrow\;\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\Big(\neg\beta^{(n-2)}\wedge\pi^{(n-2)}\Big)\Big)$$

The end of the proof is reached by matching y_0 with lower annotations:

$$\exists\Big(\underset{[0]0}{\overset{\overset{\circ}{\uparrow{\scriptstyle []2}}}{\circ}}\;\underset{[1]1}{\circ}\;\hookleftarrow\;\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}\;,\mathrm{y}_0^{(n-1)}\Big)$$

$\pi \Rightarrow \beta$ is thus established.

The direction $\beta \Rightarrow \pi$ is not as straightforward. It is best attacked by establishing a lemma first: β will be shown to be imply the condition γ (Figure C.5), which wraps β in an outer condition syntactically closer to π. Then the goal becomes to establish $\gamma \Rightarrow \pi$.

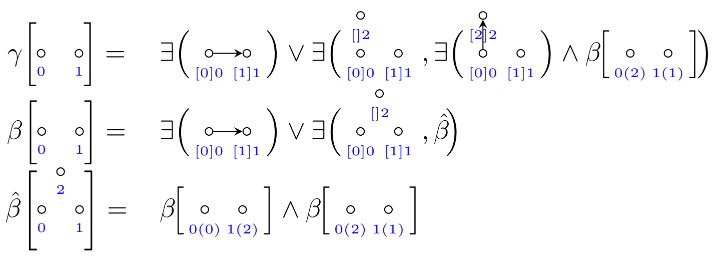
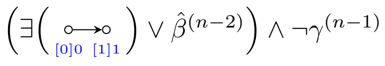


Figure C.5.: The auxiliary condition γ.

As usual, we start with $\beta \wedge \neg\gamma$ and eliminate one branch of the unrolled condition β. The $\neg\gamma$ subcondition is duplicated for later use. To distinguish between the occurrences of β originating from the subcondition β initially present and those originating from γ, the latter are primed. For $\mathcal{K}_\mu$, it makes no difference whether otherwise syntactically equal conditions use a different set of variables.

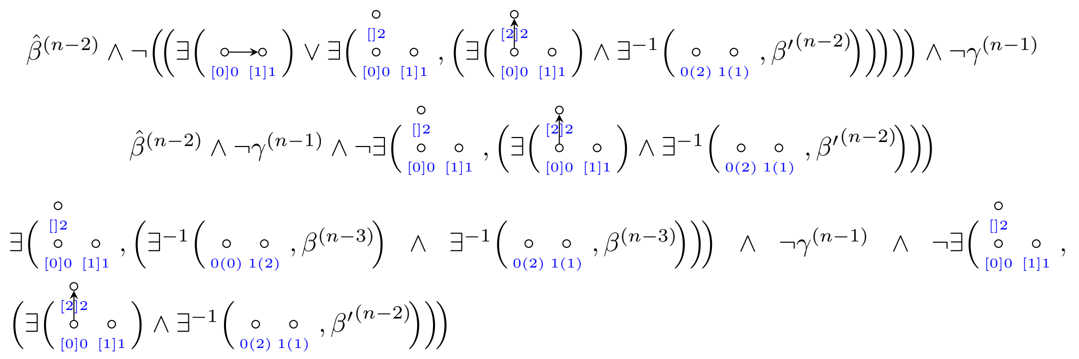
$$\beta^{(n-1)} \wedge \neg\gamma^{(n-1)}$$



$$\hat{\beta}^{(n-2)} \wedge \neg\gamma^{(n-1)}$$

$$\hat{\beta}^{(n-2)} \wedge \neg\gamma^{(n-1)} \wedge \neg\gamma^{(n-1)}$$

After unrolling the occurrence of β and one of the occurrences of γ, the condition is:



Further manipulations lead to a situation where the duplicate $\neg\gamma$ (with the unselection it has acquired from a LIFT), the positive occurrences of β and a disjunction of two (negated) members form a conjunction inside the quantifier that was inherited from the right hand side of the positive occurrence of $\hat{\beta}$:

$$\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}, \Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ}, \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}, \beta^{(n-3)}\Big) \;\wedge\; \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta^{(n-3)}\Big) \;\wedge\; \neg\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ},$$
$$\exists\Big(\overset{\overset{\circ}{\uparrow[2]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta'^{(n-2)}\Big)\Big)\Big)\Big)\Big) \wedge \neg\gamma^{(n-1)}$$

$$\Big(\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}, \Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}, \beta^{(n-3)}\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta^{(n-3)}\Big) \wedge \neg\Big(\exists\Big(\overset{\overset{\circ}{\uparrow[2]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},$$
$$\beta'^{(n-2)}\Big)\Big)\Big)\Big) \wedge \neg\gamma^{(n-1)}\Big)$$

$$\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}, \Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}, \beta^{(n-3)}\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta^{(n-3)}\Big) \wedge \Big(\neg\exists\Big(\overset{\overset{\circ}{\uparrow[2]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big) \vee \neg\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},$$
$$\beta'^{(n-2)}\Big)\Big)\Big)\Big)\Big) \wedge \neg\gamma^{(n-1)}$$

$$\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}, \Big(\exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ}, \neg\gamma^{(n-1)}\Big) \;\wedge\; \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}, \beta^{(n-3)}\Big) \;\wedge\; \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta^{(n-3)}\Big) \;\wedge$$
$$\Big(\neg\exists\Big(\overset{\overset{\circ}{\uparrow[2]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big) \vee \neg\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta'^{(n-2)}\Big)\Big)\Big)\Big)\Big)$$

One of the two members of the disjunction can be addressed straight away, by noticing that its β can be cancelled out by an occurrence of β' that appears inside an identical unselection.[2]

$$\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}, \Big(\Big(\neg\exists\Big(\overset{\overset{\circ}{\uparrow[2]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big)\Big) \vee \Big(\neg\Big(\exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta'^{(n-2)}\Big)\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta^{(n-3)}\Big)\Big)\Big)$$
$$\wedge\, \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ}, \neg\gamma^{(n-1)}\Big) \wedge \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}, \beta^{(n-3)}\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta^{(n-3)}\Big)\Big)\Big)$$

$$\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}, \Big(\Big(\neg\exists\Big(\overset{\overset{\circ}{\uparrow[2]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big)\Big) \vee \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \Big(\underline{\neg\beta'^{(n-2)} \wedge \beta^{(n-3)}}\Big)\Big)\Big) \wedge \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},$$
$$\neg\gamma^{(n-1)}\Big) \wedge \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}, \beta^{(n-3)}\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \beta^{(n-3)}\Big)\Big)\Big)$$

Let us name the other branch too because the outer nesting level will not be affected by any of the following steps.

$$\exists\Big(\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}, \Big(\mathrm{y}_9^{(n)} \vee \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ}, \mathrm{y}_8^{(n)}\Big)\Big)\Big)$$

Continuing work on the newly named subcondition, the first occurrence of β is unrolled and its base case resolved. Fast-forwarding these steps yields:

$$\mathrm{y}_9^{(n)}\Big[\overset{\overset{\circ}{[]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big] = \neg\exists\Big(\overset{\overset{\circ}{\uparrow[2]2}}{\underset{[0]0}{\circ}}\;\underset{[1]1}{\circ}\Big) \wedge \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ}, \neg\gamma^{(n-1)}\Big) \wedge \exists^{-1}\Big(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ}, \beta^{(n-3)}\Big) \wedge \exists^{-1}\Big(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},$$
$$\beta^{(n-3)}\Big)$$

[2]Instead of explicitly performing the cancellation, let it be stated (without proof) that two syntactically equal μ-conditions can always be resolved against each other using $\mathcal{K}_\mu$.

$$\neg\exists\left(\begin{array}{cc}\underset{[2]2}{\circ}&\\ \uparrow&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array}\right)\wedge\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\neg\gamma^{(n-1)}\right)\wedge\left(\exists\left(\begin{array}{cc}\underset{[1]2}{\circ}&\\ \underset{[0]0}{\circ}\longrightarrow&\underset{[2]1}{\circ}\end{array}\right)\vee\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ},\hat{\beta}^{(n-4)}\right)\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-3)}\right)$$

$$\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ},\hat{\beta}^{(n-4)}\right)\wedge\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\neg\gamma^{(n-1)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-3)}\right)\wedge\neg\exists\left(\begin{array}{cc}\underset{[2]2}{\circ}&\\ \uparrow&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array}\right)$$

Unrolling $\hat{\beta}$ and shifting yields an expression with a disjunction of two cases. We apply distributivity to the other three conjuncts to move the disjunction to the top level:

$$\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ},\exists\left(\begin{array}{cc}\underset{[]2}{\circ}&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array},\left(\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ},\beta^{(n-5)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-5)}\right)\right)\right)\right)\wedge\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\neg\gamma^{(n-1)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-3)}\right)\wedge\neg\exists\left(\begin{array}{cc}\underset{[2]2}{\circ}&\\ \uparrow&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array}\right)$$

$$\left(\exists\left(\begin{array}{cc}\underset{[1]2}{\circ}&\underset{[]3}{\circ}\\ \underset{[0]0}{\circ}&\underset{[2]1}{\circ}\end{array}\hookleftarrow\begin{array}{cc}\underset{2(3)}{\circ}&\\ \underset{0(0)}{\circ}&\underset{1(1)}{\circ}\end{array},\left(\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ},\beta^{(n-5)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-5)}\right)\right)\right)\vee\exists^{-1}\left(\begin{array}{cc}\underset{2(1)}{\circ}&\\ \underset{0(0)}{\circ}&\underset{1(2)}{\circ}\end{array},\left(\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ},\beta^{(n-5)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-5)}\right)\right)\right)\right)\wedge\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\neg\gamma^{(n-1)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-3)}\right)\wedge\neg\exists\left(\begin{array}{cc}\underset{[2]2}{\circ}&\\ \uparrow&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array}\right)$$

Naming the two direct subconditions of the disjunctions to treat them in turn:

$$\mathtt{y}_{11}^{(n)}\vee\mathtt{y}_{10}^{(n)}$$

Let us attempt to handle the subcondition with three nodes first:

$$\mathtt{y}_{11}^{(n)}\left[\begin{array}{cc}\underset{2}{\circ}&\\ \underset{0}{\circ}&\underset{1}{\circ}\end{array}\right]=\exists^{-1}\left(\begin{array}{cc}\underset{2(1)}{\circ}&\\ \underset{0(0)}{\circ}&\underset{1(2)}{\circ}\end{array},\left(\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(2)}{\circ},\beta^{(n-5)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-5)}\right)\right)\right)\wedge\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\neg\gamma^{(n-1)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-3)}\right)\wedge\neg\exists\left(\begin{array}{cc}\underset{[2]2}{\circ}&\\ \uparrow&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array}\right)$$

Composing the unselections as far as possible reveals that one occurrence of β occurs under the same unselection as $\neg\gamma$, which leads back to $\mathtt{y}_0$ with a lower annotation for β. The annotation of γ is irrelevant because it occurs negatively.

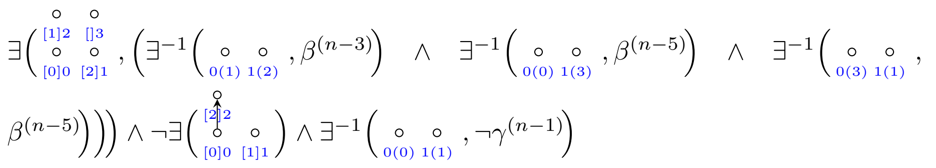
$$\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\neg\gamma^{(n-1)}\right)\wedge\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-3)}\right)\wedge\neg\exists\left(\begin{array}{cc}\underset{[2]2}{\circ}&\\ \uparrow&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array}\right)\wedge\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-5)}\right)\wedge\exists^{-1}\left(\underset{0(1)}{\circ}\;\underset{1(2)}{\circ},\beta^{(n-5)}\right)$$

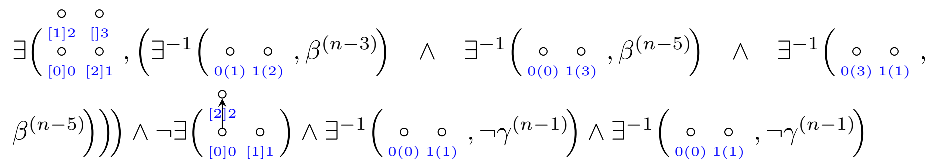
$$\exists^{-1}\left(\underset{0(2)}{\circ}\;\underset{1(1)}{\circ},\beta^{(n-3)}\right)\wedge\neg\exists\left(\begin{array}{cc}\underset{[2]2}{\circ}&\\ \uparrow&\\ \underset{[0]0}{\circ}&\underset{[1]1}{\circ}\end{array}\right)\wedge\exists^{-1}\left(\underset{0(1)}{\circ}\;\underset{1(2)}{\circ},\beta^{(n-5)}\right)\wedge\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\left(\neg\gamma^{(n-1)}\wedge\beta^{(n-5)}\right)\right)$$

$$\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\neg\gamma^{(n-1)}\wedge\beta^{(n-5)}\right)$$

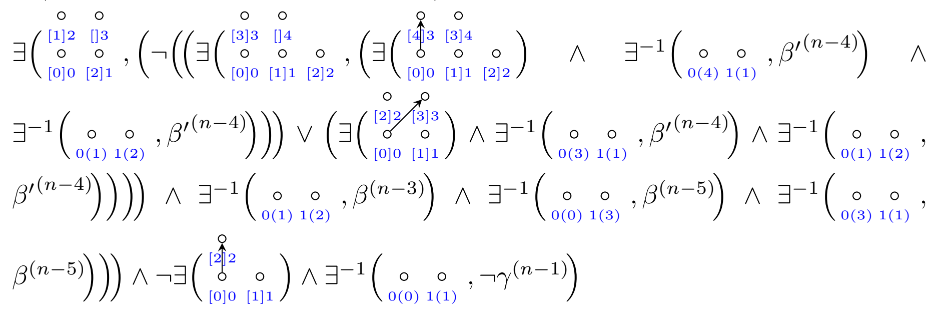
$$\exists^{-1}\left(\underset{0(0)}{\circ}\;\underset{1(1)}{\circ},\mathtt{y}_0^{(n-4)}\right)$$

The remaining subcondition turns out to be considerably more difficult to resolve. We first apply a Lift, duplicate the subcondition with γ and unroll one occurrence:





Applying Shift to the unrolled right hand side of γ, the condition expands to the following (some branches already pruned):



There are no more common unselections to be factored out. The obvious way forward would be a lift to attempt to combine subconditions, but the only existential quantifier (rather than unselection) appears in negated form and cannot be accessed. Therefore, we choose to extend the big conjunction by one more conjunct $\exists(a) \wedge \neg\exists(a)$ (where a is a morphism that posits the existence of a fifth node, see Figure C.6). This is equivalent to $\top$. Thus we use the classical law of the excluded middle, then distributivity and lift the recalcitrant negated subcondition over $\exists(a)$.

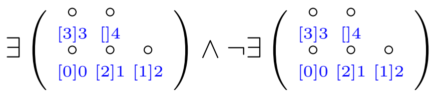
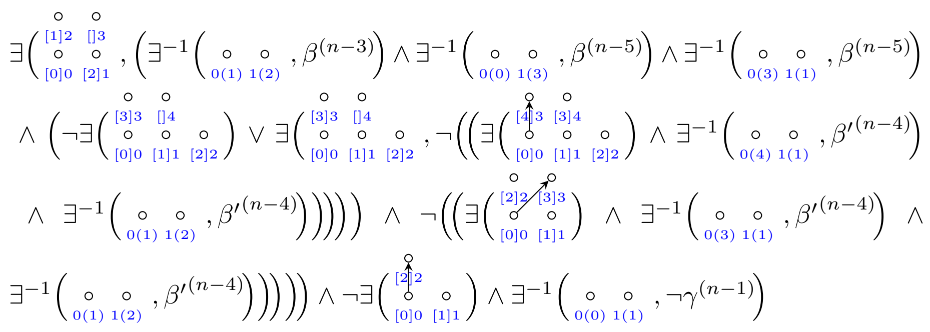
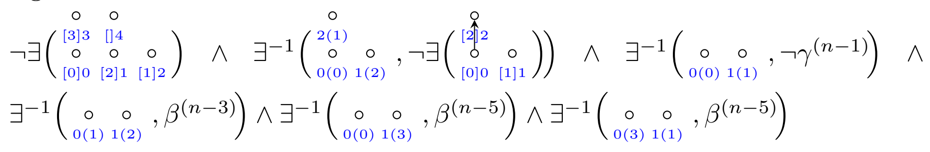


Figure C.6.: Excluded middle.

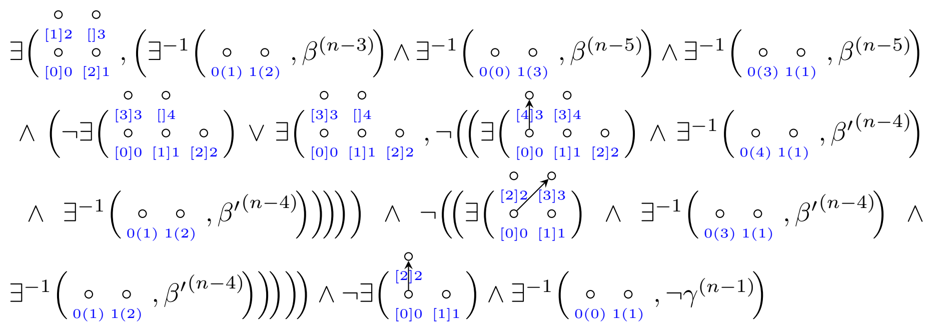
This way, the fifth node, which before only appeared under a negated quantifier, becomes accessible to lifting of subconditions from outside. First, we use distributivity to obtain a disjunction of two subconditions, one with $\exists(a)$ and the other with $\neg\exists(a)$ in conjunction with the remaining conjuncts of the current condition.



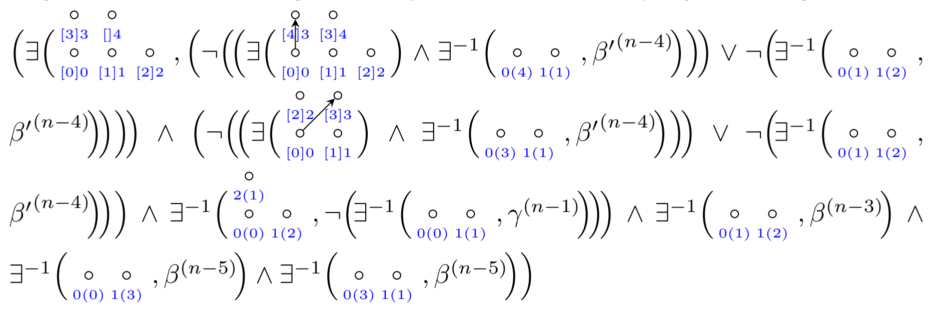
The case with $\neg\exists(a)$ reduces to a finite case distinction, because repeated unrolling of β is unnecessary: early use of $\neg\exists(a)$ removes all branches that have five nodes or more, then each of the resulting cases can be resolved by unrolling γ as far as necessary. We consider the branch finished because $\mathcal{K}_\mu$ can clearly detect its unsatisfiability without even using Rule Empty.



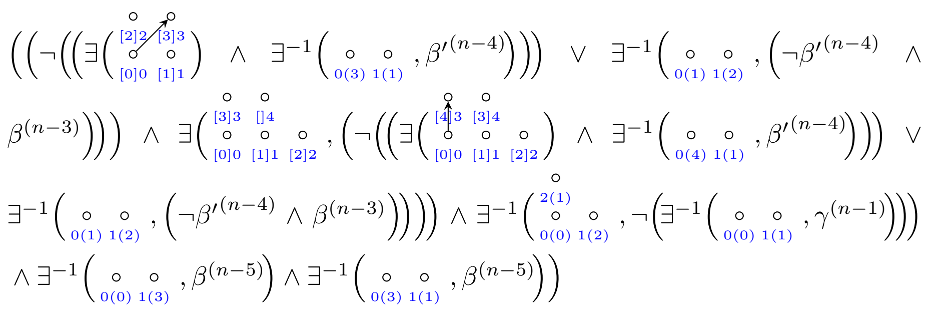
The case with $\exists(a)$ is more interesting:



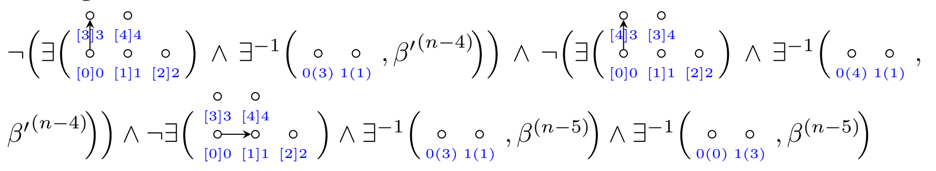
Regrouping the members of negated conjunctions and applying De Morgan's rule to both,



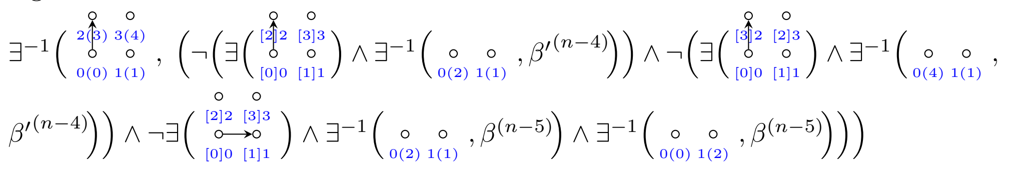
we see that each of the resulting disjunctions has a branch that can be resolved using one of the positive occurrences of β:



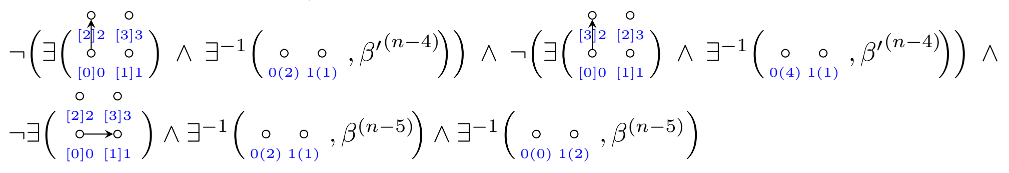
The remaining cases are:



Paying attention to the morphisms, one cannot fail but notice that the unselection of the node 2 can be moved all the way to the top of the subcondition corrently under investigation.



The final step is to notice that the direct subcondition of the outer unselection is equivalent to y_0 with a lower annotation for β, shifted over the existence of a fourth node. Thus, the proof is completed at this point (modulo minor rearrangements and UNROLL, with attention to the annotations):



It remains to show $\gamma \Rightarrow \pi$. Since the lemma $\beta \Rightarrow \gamma$ has been established, there is no obstacle to starting from the assumption $\beta^{(n-1)} \Rightarrow \neg\pi$, using CTX to change the subcondition $\beta^{(n-1)}$ to $\gamma^{(n-1)}$, performing the first steps as in all the previous proofs and obtaining a subcondition $\beta^{(n-2)} \Rightarrow \neg\pi$ that occurs positively as the only branch, thus establishing $\beta \Rightarrow \pi$ and reaching the goal.

D. Table of Symbols

The following table lists most symbols introduced, together with (if applicable) the number of their definition and the page of first appearance.

symbol	usage
structure-changing Petri nets	
Σ	alphabet of transition labels (page 18)
R	alphabet of rule names (page 18)
$N = (P, T, F^-, F^+, l, c)$ (possibly with subscripts $N_x = (P_x, ...)$)	a Petri net: places, transitions, in- and outgoing arc weights of transitions, transition labels, place colours (Definition 5, page 18)
$\mathcal{N} = (N, M)$ (possibly with subscripts)	a marked net (Definition 5, page 18)
$\mathcal{S} = (\mathcal{R}, \mathcal{N}_0)$	a structure-changing Petri net (Definition 7, page 20)
(ξ, σ)	derivation in a structure-changing workflow net (Definition 8, page 21)
$\mathfrak{F}$	homomorphism leaving Σ letters unchanged, deleting R letters (page 69)
$\mathfrak{F}(\mathcal{S})$	firing language of the structure-changing Petri net $\mathcal{S}$ (Definition 30, page 69)
graph programs and μ-conditions	
$\mathcal{M}$	the class of graph monomorphisms (page 11)
$\mathcal{A}$	the class of all graph morphisms (page 11)
$f : F \to G$	f is a graph morphism from F to G (page 11)
$f : F \hookrightarrow G$	f is a graph monomorphism from F to G (page 11)
$f : F \subseteq G$	f is a graph inclusion from F to G (page 11)
$f : F \twoheadrightarrow G$	f is a graph epimorphism from F to G (page 11)
$f : F \cong G$	f is a graph isomorphism from F to G (page 11)
$\mathcal{PM}$	the class of all partial monomorphisms ($= \mathcal{M} \times \mathcal{M}$) (page 14)
$B, C, G, H, ...$	some graphs (all plain capital letters may designate graphs)
P	some programs
$\text{Sel}(x)$, $\text{Del}(l)$, $\text{Add}(r)$, $\text{Uns}(y)$, *skip*	elementary programs (Definition 3, page 15)
$P; Q$, $P \cup Q$, P^*	composite programs
$\cong_{\text{Epi}}$	quotient equivalence relation (Definition 33, page 91)
a, f, g (plain letters)	some graph morphisms
c (plain letters)	some graph conditions (any kind)
$c : B$	c is of type B (page 26)

$\mathtt{x}_i$ (typewriter letters)	graph condition variables (placeholders). Its type conventionally has the same index $\mathtt{x}_i : B_i$ (Definition 15, page 29)
$\mathcal{F}$	condition with placeholders (Definition 15, page 29)
$\mathcal{F}_i$	condition with placeholders as right hand side of the equation for a recursively specified variable (page 32)
$\vec{\mathtt{x}}$	list of variables (page 28)
Cond	class of infinitary nested conditions (Definition 11, page 25)
$\mathrm{Cond}_{\vec{B}}$	class of sequences of infinitary nested conditions of types $\vec{B}$ (page 26)
$\mathcal{F}[\vec{\mathtt{x}}/\vec{c}]$	substitution (Definition 16, page 29)
$\vec{\mathcal{F}}, \vec{\mathcal{G}}$...	some operators (page 29)
$(b \mid l)$	a μ-condition with main body b and recursive specification l (page 30)
μ-conditions: semantics and constructions	
$f \models c$	satisfaction of condition $c : B$ by monomorphism $f \in \mathcal{M}$, $\mathrm{dom}(f) = B$ (Definition 13, page 26)
$f \models_{\mathcal{A}}$	$\mathcal{A}$-satisfaction of $\mathcal{A}$-condition $c : B$ by morphism $f \in \mathcal{A}$, $\mathrm{dom}(f) = B$ (Definition 25, page 38)
Wlp	weakest precondition transformation (Definition 37, page 100)
$\delta'_r(c)$	deletion transformation used to compute Wlp of *addition* (Construction 10, page 98)
$\alpha'_r(c)$	addition transformation used to compute Wlp of *deletion* (Construction 9, page 96)
$[\![P]\!]$	semantics of program P (Definition 3, page 15)
$A_m(c)$	shift of $c : \mathrm{dom}(m)$ over $m \in \mathcal{M}$ (Definition 34, page 91)
$\mathcal{P}_{x,y}$	partial shift with respect to jointly surjective $x : B \hookrightarrow H$ and $y : R \hookrightarrow H$ (Construction 8, page 93)
2-player programs	
$\mathcal{Y}$	the class of all two-player interfaces (Definition 39, page 121)
Wlproc	two-player extension of Wlp (Construction 11, page 125)
χ	strategy (Definition 46, page 126)
$\mathcal{X}^-$	terms owned by $(-)$ (Definition 42, page 124)
$\mathcal{X}^+$	terms owned by $(+)$ (Definition 42, page 124)
P, P', Q, Q' (sometimes)	some program terms
$\mathrm{Sel}^+(x)$ ... (with superscript $+$ or $-$)	program terms (Definition 38, page 120)
X	some atomic program term
$[\![P]\!]_\chi$	χ-ensured semantics of program P (Definition 47, page 127)

E. List of Publications

[EOF15] Björn Engelmann, Ernst-Rüdiger Olderog, and Nils Erik Flick. Closing the gap – formally verifying dynamically typed programs like statically typed ones using Hoare logic – extended version. preprint CoRR, arXiv:1501.02699 [cs.PL], 2015.

[FE15] Nils Erik Flick and Björn Engelmann. Properties of Petri nets with context-free structure changes. In *Selected Revised Papers from the Workshop on Graph Computation Models (GCM 2015)*, volume 71 of *ECEASST*, 2015.

[Fli13] Nils Erik Flick. Derivation languages of graph grammars. *ECEASST*, 61, 2013.

[Fli15] Nils Erik Flick. Quotients of unbounded parallelism. In *proceedings of ICTAC 2015*, volume 9399 of *LNCS*, pages 241–257, 2015.

[Fli16] Nils Erik Flick. Correctness of graph programs relative to recursively nested conditions. *ECEASST*, 73, 2016.

[KF13] Manfred Kudlek and Nils Erik Flick. A hierarchy of languages with catenation and shuffle. *Fundamenta Informaticae*, 128(1-2):113–128, 2013.

[KF14] Manfred Kudlek and Nils Erik Flick. Properties of languages with catenation and shuffle. *Fundamenta Informaticae*, 129(1-2):117–132, 2014.

Bibliography

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs.* Springer, 1997.

[BCD$^+$08] Paolo Baldan, Andrea Corradini, Fernando Luís Dotti, Luciana Foss, Fabio Gadducci, and Leila Ribeiro. Towards a notion of transaction in graph rewriting. *Electronic Notes in Theoretical Computer Science*, 211:39–50, 2008.

[BCE$^+$05] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICAH*, volume 5, page 04, 2005.

[BCE$^+$07] Paolo Baldan, Andrea Corradini, Hartmut Ehrig, Reiko Heckel, and Barbara König. Bisimilarity and behaviour-preserving reconfigurations of open Petri nets. In *Algebra and Coalgebra in Computer Science*, volume 4624 of *Lecture Notes in Computer Science*, pages 126–142. 2007.

[BCKK04] Paolo Baldan, Andrea Corradini, Barbara König, and Bernhard König. Verifying a behavioural logic for graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 104:5–24, 2004.

[BCKL07] Paolo Baldan, Andrea Corradini, Barbara König, and Alberto Lluch Lafuente. A temporal graph logic for verification of graph transformation systems. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *Lecture Notes in Computer Science*, pages 1–20. 2007.

[BDH92] Eike Best, Raymond Devillers, and Jon Hall. The box calculus: A new causal algebra with multi-label communication. In *Advances in Petri Nets*, volume 609 of *Lecture Notes in Computer Science*, pages 21–69. 1992.

[BFH87] Paul Boehm, Harald-Reto Fonio, and Annegret Habel. Amalgamation of graph transformations: a synchronization mechanism. *Journal of Computer and System Sciences*, 34(2):377–408, 1987.

[BKK03] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In *Static Analysis*, pages 255–272. 2003.

[BKZ14] H. J. Sander Bruggink, Barbara König, and Hans Zantema. Termination analysis for graph transformation systems. In *IFIP TCS 2014*, volume 8705 of *Lecture Notes in Computer Science*, pages 179–194, 2014.

[BLO03] Eric Badouel, Marisa Llorens, and Javier Oliver. Modeling concurrent systems: Reconfigurable nets. In *Proceedings of Int. Conf. on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2003.

[Bor94] Francis Borceux. *Handbook of Categorical Algebra*. Number 50 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994. 3 volumes.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[BS06] Julian Bradfield and Colin Stirling. Modal μ-calculi. In Johan van Benthem Patrick Blackburn and Frank Wolter, editors, *Handbook of Modal Logic*, pages 721–756. Elsevier, 2006.

[CEL$^+$96] Andrea Corradini, Hartmut Ehrig, Michael Löwe, Ugo Montanari, and Francesca Rossi. An event structure semantics for graph grammars with parallel productions. In *Graph Grammars and their Application to Computer Science*, Lecture Notes in Computer Science, pages 240–256, 1996.

[CGG02] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A spatial logic for querying graphs. In *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 597–610. 2002.

[CGJ$^+$03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[CHHK06] Andrea Corradini, Tobias Heindel, Frank Hermann, and Barbara König. Sesqui-pushout rewriting. In *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 30–45. 2006.

[CHP16] Claudio Corrodi, Alexander Heußner, and Christopher M. Poskitt. A graph-based semantics workbench for concurrent asynchronous programs. In *Proc. Intl. Conf. Fundamental Approaches to Software Engineering (FASE) 2016*, volume 9633 of *Lecture Notes in Computer Science*, pages 31–48, 2016.

[Cle90] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, 1990.

[Coo78] Stephen A Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[Cor95] Andrea Corradini. Concurrent computing: from Petri nets to graph grammars. volume 2 of *Electronic Notes in Theoretical Computer Science*, pages 56–70. 1995.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. volume 1, pages 313–400. World Scientific, 1997.

[CvW03] Orieta Celiku and Joakim von Wright. Implementing angelic nondeterminism. In *Proceedings APSEC 2003*, pages 176–185, 2003.

[DG15] Johannes Dyck and Holger Giese. Inductive invariant checking with partial negative application conditions. In *Proccedings Intl. Conf. on Graph Transformation*, volume 9151 of *Lecture Notes in Computer Science*, pages 237–253, 2015.

[Dij76] Edsger Wybe Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

[DN03] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 310–324, 2003.

[DR98] Jörg Desel and Wolfgang Reisig. Place/transition Petri nets. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 122–173. 1998.

[DS14] Giorgio Delzanno and Jan Stückrath. Parameterized verification of graph transformation systems with whole neighbourhood operations. In *Reachability Problems*, volume 8762 of *Lecture Notes in Computer Science*, pages 72–84. 2014.

[EE08] Claudia Ermel and Karsten Ehrig. Visualization, simulation and analysis of reconfigurable systems. In *Applications of Graph Transformations with Industrial Relevance*, pages 265–280. 2008.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. Springer, 2006.

[EHK$^+$97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation. Part II: Single-pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, pages 247–312. World Scientific, 1997.

[EHP$^+$07] Hartmut Ehrig, Kathrin Hoffmann, Julia Padberg, Ulrike Prange, and Claudia Ermel. Concurrency in reconfigurable P/T systems: Independence of net transformations and token firing in reconfigurable P/T systems. In *Proc. ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 104–123. 2007.

[EMB$^+$09] Claudia Ermel, Tony Modica, Enrico Biermann, Hartmut Ehrig, and Kathrin Hoffmann. Modeling multicasting in communication spaces by reconfigurable high-level Petri nets. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 47–50, 2009.

[Eng16] Björn Engelmann. *Verification Techniques for Dynamically Typed Programs.* PhD thesis, Universität Oldenburg, 2016.

[EP06] Hartmut Ehrig and Ulrike Prange. Weak adhesive high-level replacement categories and systems: A unifying framework for graph and Petri net transformations. volume 4060 of *Lecture Notes in Computer Science*, pages

235–251. 2006.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. volume 1, pages 1–94. World Scientific, 1997.

[Esp96] Javier Esparza. Decidability and complexity of Petri net problems - an introduction. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. 1996.

[FE15] Nils Erik Flick and Björn Engelmann. Properties of Petri nets with context-free structure changes. In *Selected Revised Papers from the Workshop on Graph Computation Models (GCM 2015)*, volume 71 of *Electronic Communications of the EASST*, 2015.

[FGO15] Bernd Finkbeiner, Manuel Gieseking, and Ernst-Rüdiger Olderog. Adam: Causality-based synthesis of distributed systems. In *CAV 2015*, volume 9206 of *Lecture Notes in Computer Science*, pages 433–439, 2015.

[Fli16] Nils Erik Flick. Correctness of graph programs relative to recursively nested conditions. *Electronic Communications of the EASST*, 73, 2016.

[FO14] Bernd Finkbeiner and Ernst-Rüdiger Olderog. Petri games: Synthesis of distributed systems with causal memory. In *Proc. GandALF*, volume 161 of *Electronic Proceedings in Theorecitcal Computer Science*, pages 217–230, 2014.

[Fok00] Wan Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000.

[Gai82] Haim Gaifman. On local and non-local properties. *Studies in Logic and the Foundations of Mathematics*, 107:105–135, 1982.

[Gen35a] Gerhard Gentzen. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[Gen35b] Gerhard Gentzen. Untersuchungen über das logische Schließen. II. *Mathematische Zeitschrift*, 39(1):405–431, 1935.

[GHK98] Fabio Gadducci, Reiko Heckel, and Manuel Koch. A fully abstract model for graph-interpreted temporal logic. In *TAGT'98*, volume 1764 of *Lecture Notes in Computer Science*, pages 310–322, 1998.

[GHL$^+$07] Joxe Gaintzarain, Montserrat Hermo, Paqui Lucio, Marisa Navarro, and Fernando Orejas. A cut-free and invariant-free sequent calculus for PLTL. In *Computer Science Logic*, pages 481–495, 2007.

[GRT10] James Gross, Frank G. Radmacher, and Wolfgang Thomas. A game-theoretic approach to routing under adversarial conditions. In *Proc. IFIP TCS*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 355–370. 2010.

[GRT13] Sten Grüner, Frank G. Radmacher, and Wolfgang Thomas. Connectivity games over dynamic networks. *Theor. Comp. Sci.*, 493:46–65, 2013.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*, 2002.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. 1992.

[Hec98] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. *Fundamental Approaches to Software Engineering*, pages 138–153, 1998.

[HEP06] Julia Padberg Hartmut Ehrig, Annegret Habel and Ulrike Prange. Adhesive high-level replacement systems: a new categorical framework for graph transformation. *Fundamenta Informaticae*, (74), 2006.

[HEWC97] Reiko Heckel, Hartmut Ehrig, Uwe Wolter, and Andrea Corradini. Integrating the specification techniques of graph transformation and temporal logic. *Mathematical Foundations of Computer Science*, pages 219–228, 1997.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* Foundations of Computing. MIT press, 2000.

[Hoa83] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 26(1):53–56, 1983.

[HP99] Serge Haddad and Denis Poitrenaud. Theoretical aspects of recursive Petri nets. In *Application and Theory of Petri Nets*, volume 1639 of *Lecture Notes in Computer Science*, pages 228–247. 1999.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Math. Struct. in Comp. Sci.*, 19(2):245–296, 2009.

[HPCM15] Alexander Heußner, Christopher M. Poskitt, Claudio Corrodi, and Benjamin Morandi. Towards practical graph-based verification for an object-oriented concurrency model. In *Proc. Graphs as Models (GaM 2015)*, volume 181 of *Electronic Proceedings in Theorecitcal Computer Science*, pages 32–47, 2015.

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Proceedings of the Intl. Conf. on Graph Transformation*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460, 2006.

[Jeż07] Artur Jeż. Conjunctive grammars can generate non-regular unary languages. In *Developments in Language Theory 2007*, volume 4588 of *Lecture Notes in Computer Science*, pages 242–253. 2007.

[Kai09] Łukasz Kaiser. Synthesis for structure rewriting systems. In *Mathematical Foundations of Computer Science*, volume 5734 of *Lecture Notes in Computer Science*, pages 415–426. 2009.

[KBH10] Michael Köhler-Bußmeier and Frank Heitmann. Safeness for object nets.

Fundamenta Informaticae, 101(1):29–43, 2010.

[KBH13] Michael Köhler-Bußmeier and Frank Heitmann. Complexity results for elementary hornets. In *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *Lecture Notes in Computer Science*, pages 150–169. 2013.

[KK06] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. volume 3920 of *Lecture Notes in Computer Science*, pages 197–211. 2006.

[Knu97] Donald Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, third edition edition, 1997.

[Koc99] Manuel Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems.* PhD thesis, TU Berlin, 1999.

[KR06] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In *Model Checking Software, Proc. 13th International SPIN Workshop*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305, 2006.

[KR07] Michael Köhler and Heiko Rölke. Web service orchestration with super-dual object nets. In *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 263–280, 2007.

[Kre81] Hans-Jörg Kreowski. A comparison between Petri-nets and graph grammars. In *Graph-Theoretic Concepts in Computer Science*, volume 100 of *Lecture Notes in Computer Science*, pages 306–317. 1981.

[Kre02] Stephan Kreutzer. *Pure and Applied Fixed-Point Logics.* PhD thesis, RWTH Aachen, 2002.

[KRT12] Dominik Klein, Frank G. Radmacher, and Wolfgang Thomas. Moving in a network under random failures: A complexity analysis. *Sci. of Comp. Prog.*, 77(7):940–954, 2012.

[KW86] Hans-Jörg Kreowski and Anne Wilharm. Net processes correspond to derivation processes in graph grammars. *Theoretical Computer Science*, 44:275–305, 1986.

[Lib04] Leonid Libkin. *Elements of finite model theory.* Springer, 2004.

[LO04] Marisa Llorens and Javier Oliver. Structural and dynamic changes in concurrent systems: reconfigurable Petri nets. *Computers, IEEE Transactions on*, 53(9):1147–1158, 2004.

[LO14] Leen Lambers and Fernando Orejas. Tableau-based reasoning for graph properties. In *Graph Transformation*, volume 8571 of *Lecture Notes in Computer Science*, pages 17–32. 2014.

[Lom08] Irina A. Lomazova. Nested Petri nets for adaptive process modeling. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science*, volume 4800 of *Lecture Notes in Computer Science*, pages

460–474. 2008.

[LS04] Stephen Lack and Paweł Sobociński. Adhesive categories. In *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288, 2004.

[LZC13] Zhao-Peng Li, Yu Zhang, and Yi-Yun Chen. A shape graph logic and a shape system. *Journal of Computer Science and Technology*, 28(6):1063–1084, 2013.

[Mas11] Marco Mascheroni. *Hypernets: a class of hierarchical Petri nets.* PhD thesis, Università degli Studi di Milano-Bicocca, 2011.

[MEE12] Maria Maximova, Hartmut Ehrig, and Claudia Ermel. Transfer of local confluence and termination between Petri net and graph transformation systems based on M-functors. *Electronic Communications of the EASST*, 51, 2012.

[MGH11] Tony Modica, Karsten Gabriel, and Kathrin Hoffmann. Formalization of Petri nets with individual tokens as basis for DPO net transformations. In *Proceedings PNGT 2010*, volume 40 of *Electronic Communications of the EASST*. 2011.

[MN16] Fernando Orejas Leen Lambers Marisa Navarro, Elvira Pino. A logic of graph conditions extended with paths. In *PreProc. 7th Int. Workshop on Graph Computation Models (GCM)*, 2016.

[OEP10] Fernando Orejas, Hartmut Ehrig, and Ulrike Prange. Reasoning with graph constraints. *Formal Aspects of Computing*, 22(3-4):385–422, 2010.

[Okh13] Alexander Okhotin. Conjunctive and boolean grammars: The true general case of the context-free grammars. *Comp. Sci. Review*, 9:27 – 59, 2013.

[Old14] Ernst-Rüdiger Olderog. Von Petri-Spielen zu endlichen Automaten. In *44. Jahrestagung der GI*, volume 232 of *LNI*, page 2209. GI, 2014.

[Ore11] Fernando Orejas. Symbolic graphs for attributed graph constraints. *Journal of Symbolic Computation*, 46(3):294–315, 2011.

[Pad12] Julia Padberg. Abstract interleaving semantics for reconfigurable Petri nets. *Electronic Communications of the EASST*, 51, 2012.

[PEHP08] Ulrike Prange, Hartmut Ehrig, Kathrin Hoffmann, and Julia Padberg. Transformations in reconfigurable place/transition systems. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 96–113. 2008.

[Pen09] Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems.* PhD thesis, Universität Oldenburg, 2009.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn, 1962.

[PH15] Julia Padberg and Kathrin Hoffmann. A survey of control structures for reconfigurable Petri nets. *Journal of Computer and Communications*, 3:20–28, 2015.

[PH16] Christoph Peuser and Annegret Habel. Attribution of graphs by composition of $\mathcal{M},\mathcal{N}$-adhesive categories. *Electronic Communications of the EASST*, 73, 2016.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[Pos13] Christopher M. Poskitt. *Verification of Graph Programs.* PhD thesis, University of York, 2013.

[PP12] Christopher M Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1):135–175, 2012.

[PP13] Christopher M. Poskitt and Detlef Plump. Verifying total correctness of graph programs. *Electronic Communications of the EASST*, 61, 2013.

[PP14] Christopher M. Poskitt and Detlef Plump. Verifying monadic second-order properties of graph programs. In *Proceedings of the Intl. Conf. on Graph Transformation*, volume 8571 of *Lecture Notes in Computer Science*, pages 33–48. 2014.

[PST13] Christian Percebois, Martin Strecker, and Hanh Nhi Tran. Rule-level verification of graph transformations for invariants based on edges' transitive closure. In *SEFM 2013*, volume 8137 of *Lecture Notes in Computer Science*, pages 106–121, 2013.

[PU03] Julia Padberg and Milan Urbášek. Rule-based refinement of Petri nets: A survey. In *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 161–196. 2003.

[PW02] Lutz Priese and Harro Wimmel. *Petri-Netze.* Springer, 2002.

[RAB$^+$15] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Gabriele Taentzer. Translating essential OCL invariants to nested graph constraints focusing on set operations. In *Proceedings Intl. Conf. on Graph Transformation*, volume 9151 of *Lecture Notes in Computer Science*, pages 155–170. 2015.

[Rad13] Hendrik Radke. HR* graph conditions between counting monadic second-order and second-order graph formulas. *Electronic Communications of the EASST*, 61, 2013.

[Rad16] Hendrik Radke. *A Theory of HR* Graph Conditions and their Application to Meta-Modeling.* PhD thesis, Universität Oldenburg, 2016.

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, 157:39–59, 2006.

[Ren04] Arend Rensink. Representing first-order logic using graphs. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations (ICGT'04)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335. 2004.

[Ren06] Arend Rensink. Nested quantification in graph transformation rules. In *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 1–13. 2006.

[Ren08] Arend Rensink. Explicit state model checking for graph grammars. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132. 2008.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74, 2002.

[RSVB05] Jean-François Raskin, M. Samuelides, and Laurent Van Begin. Games for counting abstractions. *Electronic Notes in Theoretical Computer Science*, 128(6):69–85, 2005.

[RT08] Frank G. Radmacher and Wolfgang Thomas. A game theoretic approach to the analysis of dynamic networks. *Electronic Notes in Theoretical Computer Science*, 200(2):21–37, 2008.

[RW11] Silvain Rideau and Glynn Winskel. Concurrent strategies. In *LICS*, pages 409–418, 2011.

[RZ12] Arend Rensink and Eduardo Zambon. Pattern-based graph abstraction. In *Graph Transformations - Proceedings Intl. Conf. on Graph Transformation*, volume 7562 of *Lecture Notes in Computer Science*, pages 66–80, 2012.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118. ACM, 1999.

[Str08] Martin Strecker. Modeling and verifying graph transformations in proof assistants. *Electronic Notes in Theoretical Computer Science*, 203(1):135–148, 2008.

[Str14] Tim Strazny. *An algorithmic framework for checking coverability in well-structured transition systems.* PhD thesis, Universität Oldenburg, 2014.

[Stu08] Thomas Studer. On the proof theory of the modal mu-calculus. *Studia Logica*, 89(3):343–363, 2008.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3: Beyond Words, pages 389–455. Springer, 1997.

[vdA97] Wil M. P. van der Aalst. Verification of workflow nets. In *Applications and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. 1997.

[vdA01] Wil M. P. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Sys. Frontiers*, 3(3):297–317, 2001.

[vSV03] Kees van Hee, Natalia Sidorova, and Marc Voorhoeve. Soundness and separability of workflow nets in the stepwise refinement approach. In *Applications and Theory of Petri Nets*, volume 2679 of *Lecture Notes in Computer Science*, pages 337–356. 2003.

[WRRM08] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering*, 66(3):438–466, 2008.

[YS10] Eran Yahav and Shmuel Sagiv. Verifying safety properties of concurrent heap-manipulating programs. *ACM Trans. Program. Lang. Syst.*, 32(5):18:1–18:50, 2010.

[ZR11] Eduardo Zambon and Arend Rensink. Using graph transformations and graph abstractions for software verification. *Electronic Communications of the EASST*, 38, 2011.

Index