Electronic Communications of the EASST
Volume 73 (2016)

Graph Computation Models
Selected Revised Papers from GCM 2015

Proving Correctness of Graph Programs Relative to Recursively Nested Conditions

Nils Erik Flick

20 pages

Guest Editors: Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Proving Correctness of Graph Programs Relative to Recursively Nested Conditions

Nils Erik Flick*

Carl von Ossietzky Universität, 26111 Oldenburg, Germany,
flick@informatik.uni-oldenburg.de

Abstract: We propose a new specification language for the proof-based approach to verification of graph programs by introducing μ-conditions as an alternative to existing formalisms which can express many non-local properties of interest. The contributions of this paper are the lifting of constructions from nested conditions to the new, more expressive conditions and a proof calculus for partial correctness relative to μ-conditions. Most importantly, we prove the correctness of a new construction to compute weakest preconditions with respect to finite graph programs.

Keywords: correctness, graph programs, non-local graph conditions, weakest precondition calculus, proof calculus

1 Introduction

Graph transformations provide a formal way to model 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. The following figure presents a schematic overview of the approach, which is also our starting point:

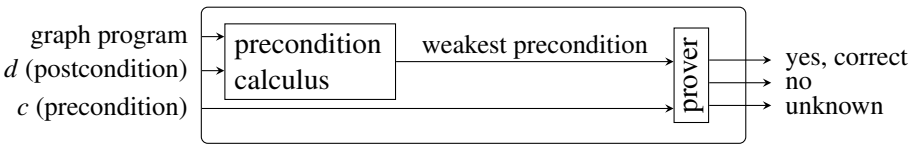


Figure 1: Overview of the proof-based verification approach

Correctness proofs are done in the style of Dijkstra's [Dij76] predicate transformer approach in Pennemann's thesis [Pen09], while Poskitt's thesis [Pos13] features a Hoare [Hoa83] logic for partial and total correctness. Both works are based on nested conditions, which cannot express non-local properties of graphs, such as connectivity. In this paper, we consider non-local properties, and we present an extension to the proof calculus from [Pen09]. Our formalism is an extension of nested conditions by recursive definitions. Several extensions of nested graph

* This work is supported by the German Research Foundation (DFG), grant GRK 1765 (Research Training Group – System Correctness under Adverse Conditions)

conditions to non-local conditions already exist (Radke [Rad13], Poskitt and Plump [PP14]). We argue that as opposed to the former, ours offers a weakest precondition calculus that can handle any condition expressible in it. As compared to the latter, 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. Thus μ-conditions offer a viewpoint sufficiently different from existing ones to be worth investigating.

This paper is structured as follows: Section 2 recalls graph programs and conditions. Section 3 introduces μ-conditions and Section 4 defines correctness under μ-conditions and the proof calculus, together with proofs of the results and a (small) exemplary application of the method. Section 5 provides context by listing related work. Section 6 concludes with an outlook.

2 Graph Conditions and Programs

In this section, we introduce graph conditions and graph programs. We assume familiarity with graph transformation systems in the sense of Ehrig et al. [EEPT06], and the basic notions of category theory. For practical approaches to semi-automatic theorem proving in this context, we also refer the reader to Pennemann [Pen09]. In this paper, all graphs are assumed to be finite.

Notation *The domain and codomain of a morphism* $f : G \to H$ *are denoted by* $\mathrm{dom}(f) = G$ *and* $\mathrm{cod}(f) = H$. *Curly arrows* $f : G \hookrightarrow H$ *denote injective morphisms (*monomorphisms*) while double-ended arrows* $f : G \twoheadrightarrow H$ *denote surjective ones (*epimorphisms*). $\mathcal{M}$ denotes the class of all graph monomorphisms. An* isomorphism *of graphs is both a mono- and an epimorphism. A* partial morphism *is a pair of monomorphisms with the same domain.* $\emptyset$ *denotes the empty graph.*

Let us recall nested conditions, initially due to Habel and Pennemann. Finite nested conditions are known to be equally expressive as graph-interpreted first-order predicate logic.

Definition 1 (Nested Graph Conditions) Let Cond be the class of nested conditions, defined inductively as follows (where P, C', C are graphs):

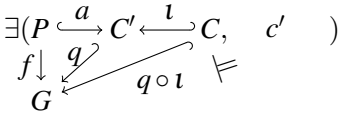
- If J is a countable set and for all $j \in J$, c_j is a condition (over P), then $\bigvee_{j \in J} c_j$ is a condition (over P). This includes the case $J = \emptyset$ (for any P).
- If c is a condition (over P), then $\neg c$ is also a condition (over P).
- If $a : P \hookrightarrow C'$ is a monomorphism,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

Notation *We call* c' *a* direct subcondition *of* $\exists(\alpha, \iota, c')$, $\neg c'$ *and* $c' \vee c''$ *and use* subcondition *for the reflexive and transitive closure of this syntactically defined relation. If* c *is a condition over* P, *then* P *is its* type[1], *denoted* $c : P$, *and* Cond_P *is the class of all conditions over* P. *The usual abbreviations define the other standard operators:* $\bigwedge$ *is* $\neg \bigvee \neg$, $\forall$ *is* $\neg \exists \neg$. *No morphism satisfies the disjunction over the empty index set. To avoid special cases, we write it as* $\bot$ *(false), and* $\top$ *(true) for* $\neg \bot$, *though technically for each graph* P *there is one* $\bot : P$ *and one* $\top : P$. *We write* $\exists(a)$ *for* $\exists(a, \iota, \top)$, $\exists(a, c)$ *for* $\exists(a, id_{\mathrm{cod}(a)}, c)$ *and* $\exists^{-1}(\iota, c)$ *for* $\exists(id_{\mathrm{cod}(\iota)}, \iota, c)$.

[1] When we mention "type graphs" in the text, we just mean graphs used as types.

With finite index sets, one obtains the *finite* nested conditions. The morphism ι serves to unselect[2] a part of C'. Our extension is similar to lax conditions [RAB+15], but slighter in scope.

Definition 2 (Satisfaction) A monomorphism $f : P \hookrightarrow G$ *satisfies* a condition $c : P$, denoted $f \models c$, iff $c = \top$, $c = \neg c'$ and $f \not\models c'$, or $c = \bigvee_{j \in J} c_j$ and there is a $j \in J$ such that $f \models c_j$, or $c = \exists(a, \iota, c')$ (where $a : P \hookrightarrow C'$, $\iota : C \hookrightarrow C'$, $c' : C$) and there exists a monomorphism $q : C' \hookrightarrow G$ such that $f = q \circ a$ and $q \circ \iota \models c'$.



A graph G satisfies a condition $c : \emptyset$ if and only if the unique morphism $\emptyset \hookrightarrow G$ satisfies c.

Notation *The symbol $\equiv$ denotes logical equivalence, i.e. for conditions $c, c' : P$, $c \equiv c'$ if and only if for all monomorphisms m with domain P, $\Rightarrow m \models c \Leftrightarrow m \models c'$.*

Remark 1 (No Added Expressivity) *Our conditions with ι are equally expressive as the nested conditions defined in [Pen09]. The proof, omitted, relies on the transformation A from [Pen09].*

Notation *As one can see in Fig. 2, the notation for graph conditions often only depicts source or target graphs of morphisms. The small blue numbers show the morphisms' node mappings. We also adopt the convention of representing the morphism ι in a situation $\exists(a, \iota, \mathrm{x}_i)$ implicitly: we prefer to annotate the variable's type graph with the images of items under ι in parentheses.*

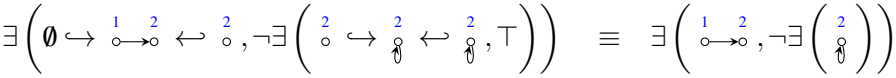


Figure 2: A nested graph condition, in full and in abbreviated notation (stating the existence of two nodes linked by an edge, the second node not having a self-loop).

Next, we introduce graph transformations. We follow the double pushout approach with injective rules and injective matches. For technical reasons, we 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. skip is 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. Its only complication is that programs can only be composed if they agree on the currently selected subgraph, called the *interface*. This ensures that an addition is performed in the same place as the deletion. A graph transformation rule is then nothing else than a selection; a deletion; an addition; an unselection.

The semantics of a graph program is a triple of two monomorphisms and one partial morphism. The two monomorphisms represent the selected subgraphs before and after the execution of the

[2] We will use the term "unselection" anytime a morphism is used in the inverse direction: in Def. 1, 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 for "selection". Unselection is an indispensable part of our formalism.

program respectively, and the partial morphism records the changes effected by the program. Our programs are a proper subset of those in Pennemann [Pen09], and use the same semantics.

Definition 3 (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 *interfaces* m_{in} (*input*) and m_{out} (*output*) are universally quantified in the set comprehensions that appear in the definitions below. Each triple $(m_{in}, m_{out}, (p_l, p_r))$ has $\mathrm{cod}(p_l) = \mathrm{dom}(m_{in})$ and $\mathrm{cod}(p_r) = \mathrm{dom}(m_{out})$.

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

If P and Q are graph programs, so are their disjunction $\{P, Q\}$ and sequence $P;Q$. The semantics of disjunction is a set union $[\![P]\!] \cup [\![Q]\!]$ 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 partial morphisms $p' = (l_1, r_1)$, $p'' = (l_2, r_2)$ 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). If P is a graph program, so is its *iteration* P^*: $[\![P^*]\!] = \bigcup_{j \in \mathbb{N}} [\![P^j]\!]$ where $P^j = P; P^{j-1}$ for $j \geq 1$ and $P^0 = \mathrm{skip}$.

Remark 2 *The definitions generalise the state transitions in plain graph transformation: a rule* $\rho = (L \xhookleftarrow{l} K \xhookrightarrow{r} R)$ *is exactly simulated by the program* $Sel(\emptyset \hookrightarrow L); Del(l); Add(r); Uns(\emptyset \hookrightarrow R)$.

3 μ-Conditions

In this section, we define 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. We then define and prove the correctness of some basic constructions. An example is provided at the end of this section to illustrate the constructions step by step.

3.1 Defining μ-Conditions

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 resulting formalism is similar to first order fixed point logics, see e.g. Kreutzer [Kre02].

The reader might want to compare our μ-conditions to a distinct formalism for expressing non-local properties, the very powerful grammar-based HR* conditions of Radke [Rad13]. We argue that μ-conditions are worth looking into despite the availability of strong contenders for an extension of nested conditions to non-local properties, such as MSO-conditions [PP14] because our approach provides a new and different generalisation of nested conditions, also is it not obvious how the respective expressivities compare.[3] Specifically, we show in this section that

[3] See Section 5 for a summary of related work on non-local graph condition formalisms.

the weakest liberal precondition transformation, core of the Dijkstra-style approach, carries over.

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

To define fixed point conditions, we need something to take fixed points of, and to enforce existence and uniqueness. Choosing a partial order on $\mathrm{Cond}_{\vec{P}}$, one can define monotonic operators on $\mathrm{Cond}_{\vec{P}}$. 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]. To represent systems of simultaneous equations, we work on tuples of conditions. If $\vec{P} = P_1, \ldots, P_{\|\vec{P}\|}$ is a sequence of graphs, then $\mathrm{Cond}_{\vec{P}}$ is the set of all $\|\vec{P}\|$-tuples $\vec{c}$ of conditions, whose i-th element is a condition over the i-th graph of $\vec{P}$. Satisfaction is defined component-wise: $\vec{f} \models \vec{c}$ if and only if $\forall k \in \{1, \ldots, \|\vec{P}\|\}\ f_k \models c_k$.

Disjunctions $\bigwedge$ and conjunctions $\bigvee$ of countable sets of Cond_P conditions, which by definition exist for any P, are easily seen to be least upper resp. greatest lower bounds of the sets. This makes $\mathrm{Cond}_{\equiv P}$ a complete lattice. Let $\mathrm{Cond}_{\vec{P}}$ 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, $\mathrm{Cond}_{\equiv \vec{P}}$. Thus, $\mathrm{Cond}_{\equiv \vec{P}}$ 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 Def. 1 by placeholders, and define substitutions of conditions for placeholders:

Definition 4 (Graph Conditions with Placeholders) Given a graph P and a finite sequence $\vec{P}$ of graphs, a *condition with placeholders from* $\vec{P}$ over P is a (graph) condition with placeholders is either $\exists(a, \iota, c)$, or $\neg c$, or $\bigvee_{j \in J} c_j$, or $\mathtt{x}_i$, $1 \leq i \leq \|\vec{P}\|$ where $\mathtt{x}_i$ is a variable of type P_i.

Definition 5 (Substitution) If $\mathcal{F}$ is a condition with placeholders $\vec{\mathtt{x}}$ of types $\vec{P}$ and $\vec{c} \in Cond_{\vec{P}}$, then $\mathcal{F}[\vec{\mathtt{x}}/\vec{c}]$ is obtained by substituting c_i for each occurrence of $\mathtt{x}_i$ for all $i \in \{1,...,\|\vec{P}\|\}$.

Satisfaction of such a condition by a morphism f is defined relative to a *valuation* val, which is an assignment of $\top$ or $\bot$ to each monomorphism of the type graph of the variable into $\mathrm{cod}(f)$, by $f \models \mathtt{x}_i$ iff $\mathrm{val}(\mathtt{x}_i) = \top$ (where $\mathrm{dom}(f) = P_i$ and $\mathtt{x}_i : P_i$). As discussed above, a lfp 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 $\vec{c}, \vec{d} \in Cond_{\vec{P}}$). The following remark is very useful:

Remark 3 *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 $\vec{\mathcal{F}}(\bigvee_{n \in N} \vec{\mathcal{F}}^n(\overrightarrow{\bot})) = \bigvee_{n \in N-\{0\}} \vec{\mathcal{F}}^n(\overrightarrow{\bot}) = \bot \vee \bigvee_{n \in N-\{0\}} \vec{\mathcal{F}}^n(\overrightarrow{\bot})$

[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".

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

Definition 6 (μ-Condition) Given a finite list $\vec{P}$ and conditions $\{\mathcal{F}_i\}_{i\in\{1,\dots,\|\vec{P}\|\}}$ with placeholders $\vec{\mathtt{x}} : \vec{P}$ ($\mathcal{F}_i$ having type P_i and so on), then $\mu[\vec{\mathtt{x}}]\vec{\mathcal{F}}(\vec{\mathtt{x}})$ denotes the least fixed point (*lfp*) of the operator that to any $\vec{c}$ assigns $\vec{\mathcal{F}}[\vec{\mathtt{x}}/\vec{c}]$. A *$\mu$-condition* is a pair (b,l) consisting of a condition with placeholders b, and a finite list of pairs $l = (\mathtt{x}_i, \mathcal{F}_i(\vec{\mathtt{x}}))$ of a variable $\mathtt{x}_i : P_i$ and a condition $\mathcal{F}_i(\vec{\mathtt{x}}) : P_i$, with placeholders from $\vec{\mathtt{x}}$, for some graph P_i, such that $\vec{\mathcal{F}}$ is monotonic.

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 $l = (\mathtt{x}_i, \mathcal{F}_i(\vec{\mathtt{x}}))_{i\in\{1,\dots,\|\vec{P}\|\}}$ as a system of equations $\vec{\mathtt{x}} = \vec{\mathcal{F}}(\vec{\mathtt{x}})$. We call b the* main body *and l the* recursive specification *of (b,l) (and $\mathcal{F}_i(\vec{\mathtt{x}})$ the* body *or* right hand side *of the variable x_i in l, or the i-th* component *of $\vec{F}$), and $\vec{\mathcal{F}}(\vec{c})$ is understood as substitution of conditions $\vec{c}$ for the variables $\vec{\mathtt{x}}$. $\vec{\mathcal{F}}$ is said to* define *the variables $\vec{\mathtt{x}}$.*

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

Definition 7 (Transitive Variable Use) Let $\{\mathcal{F}_i\}_{i\in I\}}$ be a list of conditions as in Def. 6. The *use* relation of $\mathcal{F}$, $\rightsquigarrow_{\mathcal{F}}$, is defined on literals $\{\mathtt{x}_i, \neg\mathtt{x}_i\}_{i\in I}$ by $\mathtt{x}_i \rightsquigarrow_{\mathcal{F}} \mathtt{x}_j$ ($\neg\mathtt{x}_j$) iff $\mathtt{x}_j$ 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 $\forall 1 \le i < m.\pi_{p_i} \rightsquigarrow_{\mathcal{F}} \pi_{p_{i+1}}$ and $\forall 1 < j < m.\pi_{p_m} \neq \pi_{p_j}$.

Lemma 1 (Nested Fixed Points) *Given conditions with placeholders $\{\mathcal{F}_i(\vec{x})\}_{i\in I}$, if there is a partitioning $I = I_1 \uplus I_2$ with $\vec{\mathtt{x}}_1 = \{\vec{\mathtt{x}}_i\}_{i\in I_1}$ and $\vec{\mathtt{x}}_2 = \{\vec{\mathtt{x}}_i\}_{i\in I_2}$ such that $\{\mathcal{F}_i\}_{i\in I_1}$ does not use variables of $\vec{\mathtt{x}}_2$, then if $\mu[\vec{\mathtt{x}}]\vec{\mathcal{F}}(\vec{\mathtt{x}})$ exists it is equivalent to $\mu[\vec{\mathtt{x}}_1]\vec{\mathcal{F}}_{I_1}(\vec{\mathtt{x}}_1)$ with $\vec{\mathcal{F}}_{I_1}(\vec{\mathtt{x}_1}) = \mu[\vec{\mathtt{x}}_2]\vec{\mathcal{F}}_{I_2}(\vec{\mathtt{x}}_2, \vec{\mathtt{x}}_1)$.*

Proof. Immediate from the definition of least fixed point. □

Definition 8 (Stratification) $\mathcal{F}$ is said to be *stratified* if there is a decomposition into $\vec{\mathcal{F}}_{I_1}, \dots, \vec{\mathcal{F}}_{I_n}$ such that each $\vec{\mathcal{F}}_{I_m}$ is monotonic in $\vec{\mathtt{x}}_{I_m}$ and there are no variables $\mathtt{x}_i \rightsquigarrow^+_{\mathcal{F}} \mathtt{x}_j$, $j \in I_j$, $i \in I_i$, $j < i$. 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 1. Therefore there is no ambiguity in presenting a nested fixed point as a system of equations without explicit stratification. Monotonicity and stratification can be enforced syntactically and we only consider such μ-conditions to be well-formed:

Remark 4 (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 Def. 2, negation interchanges both even/odd and monotonicity/antitonicity. Disjunction, defined via propositional logic too, is monotonic, 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. □

Remark 5 (First Example: μ-Conditions are More General than Nested)

1. *μ-conditions generalise 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 nested conditions: the following expresses the existence of a path of unknown length between two given nodes.*

$$\mathtt{x}_1\Big[\underset{1}{\circ}\quad\underset{2}{\circ}\Big] \text{ where } \mathtt{x}_1\Big[\underset{1}{\circ}\quad\underset{2}{\circ}\Big] = \exists\Big(\underset{1}{\circ}\!\!\longrightarrow\!\!\underset{2}{\circ}\Big) \vee \exists\Big(\begin{array}{c}\circ^{\,3}\\ \uparrow\\ \circ_1\end{array}\quad\underset{2}{\circ}, \mathtt{x}_1\Big[\underset{1(3)}{\circ}\quad\underset{2(2)}{\circ}\Big]\Big)$$

It reads as follows: the word "where" stands between main body and equations. The only variable is $\mathtt{x}_1$. Its type graph 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 graph 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 graph of a given variable.

Let us motivate the necessity of "unselection". The *nesting depth* $n : \mathrm{Cond} \to \mathbb{N}$ is defined as $n(\bigvee_{j\in J} c_j) = \max(\{0\} \cup \{n(c_j) \mid j \in J\})$, $n(\neg c) = n(c)$, $n(\exists(a,\iota,c)) = n(c)+1$, $(n(\mathtt{x}_i) = 0)$.

Lemma 2 (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. Define the reduced condition $r_{a,\iota}(c')$ thus: if $c' = \bigvee_{j\in J} c_j$, then $r_{a,\iota}(c') = \bigvee_{j\in J} r_{a,\iota}(c_j)$. If $c' = \neg c''$, then $r_{a,\iota}(c') = \neg r_{a,\iota}(c'')$. If $c' = \exists(a',\iota',c'')$, then $r_{a,\iota}(c') = \exists(a' \circ \iota^{-1} \circ a, c')$. Directly from Def. 2, $r_{a,\iota}(c') \equiv c$ 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. □

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

Proof. Decompose $\vec{\mathcal{F}}$ by Lemma 1 such that each stratum $\vec{\mathcal{F}}_{I_m}$ 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 $\vec{\mathcal{F}}_{I_m}$, after each step of the lfp iteration, the nesting level can be reduced by Lemma 2 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 $\exists(a, \iota, \mathrm{x}_i)$ with isomorphisms a and ι. Finitely many distinct conditions of this form, hence finitely many distinct Boolean combinations, exist. The monotonic operator $\vec{\mathcal{F}}_{I_m}$ 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. □

Definition 9 (Satisfaction) $b \mid \vec{\mathrm{x}} = \vec{\mathcal{F}}(\vec{\mathrm{x}})$ with $\vec{\mathrm{x}} : \vec{P}$ is satisfied by f iff $f \models b[\vec{\mathrm{x}}/\mu[\mathcal{P}]\mathcal{F}]$.

This means that the μ-condition $b \mid \vec{\mathrm{x}} = \vec{\mathcal{F}}(\vec{\mathrm{x}})$ can be understood by substituting the lfp solution of the system of equations $\vec{\mathrm{x}} = \vec{\mathcal{F}}(\vec{\mathrm{x}})$ 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 7 (Finite Nesting) *By the "infinite disjunction" characterisation of the lfp, any μ-condition is equivalent to an infinite nested condition. Infinitely deep nesting is not needed because the characterisation in Rem. 3 yields a countable disjunction of* finitely deeply *nested conditions.*

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) $f \models b \mid \vec{\mathrm{x}} = \vec{\mathcal{F}}(\vec{\mathrm{x}})$ *iff* $\exists n \in \mathbb{N}$, $f \models b[\vec{\mathrm{x}}/\vec{\mathcal{F}}^n(\overrightarrow{\perp})]$.

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

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

Proof. The following algorithm `CheckMu` decides $f \models c$. For the type graph P_i of each variable x_i, list all monomorphisms $m_{ik} : P_i \hookrightarrow G$. Build a table which records in each column a Boolean value for each pair (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 $\vec{\mathcal{F}}^j(\overrightarrow{\perp})$, by definition. It terminates because of monotonicity: as values never change back to $\perp$ from $\top$ while progressing through the columns, there is a finite number $j^* \in \mathbb{N}$ such that $\vec{\mathcal{F}}^{j^*}$ is satisfied by f iff $\vec{\mathcal{F}}^{j^*+1}$ is. □

3.2 Weakest Liberal Preconditions of μ-conditions

In this subsection, 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 termination of P is not implied. It is redundant in the absence of iteration, as only iteration causes non-termination).

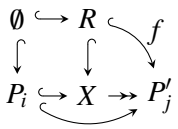
Definition 10 (Weakest Liberal Precondition) The weakest liberal precondition (wlp) of c with respect to the program P, $\mathrm{Wlp}(P, c)$, is the least condition with respect to implication such that $f' \models c \Rightarrow f \models \mathrm{Wlp}(P, c)$ if $(f, f', p) \in [\![P]\!]$ for some partial morphism 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 the case $c = \exists(a,c')$: if (m',a') is the pushout of (m,a), let Epi be the set of all epimorphisms e with domain $\mathrm{cod}(m')$ that compose to monomorphisms $b := e \circ a'$ and $r := e \circ m'$. Then $A_m(\exists(a,c')) = \bigvee_{e \in \mathrm{Epi}} \exists(b, A_r(c'))$.

With help of the unselection ι in $\exists(a,\iota,c)$, it is at first glance trivial to exhibit a weakest liberal precondition with respect to $Uns(y)$. However, to handle the addition and deletion steps, a construction becomes necessary that makes the affected subgraph explicit again. This information is crucial to obtain the weakest liberal precondition with respect to $Add(r)$ and $Del(l)$ and must not be forgotten at any nesting level in order to obtain the correct result. To that aim, we define a *partial shift* construction which makes sure that the type graph of the main body is never unselected in the μ-condition but is instead mapped in a consistent way as a subgraph of the type graph of each variable. The following serves to obtain the new type graphs:

Construction 1 (New type graphs for partial shift) *We assume that an arbitrary total order on all graph morphisms is fixed. If $c = b \mid \vec{x} = \mu[\vec{K}]\vec{\mathcal{F}}(\vec{x})$ is a μ-condition, then for a variable $\mathtt{x}_i$ of $\vec{K}$, $\mathcal{X}_{R,c}(\mathtt{x}_i)$ is defined as the sequence of morphisms $\vec{f}$ obtained as below, in ascending order.*

The new list of variables $\vec{K}'$ and their respective types $\vec{P}'$ are obtained by concatenating all $\mathcal{X}_{R,c}(\mathtt{x_i})$ of the variables of $\vec{K}$ in order. The morphisms f are obtained from $\vec{P}'$ by collecting all epimorphisms that compose to monomorphisms with the pushout morphisms in the diagram:

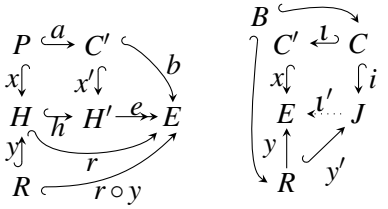


Lemma 3 (Decomposing "exists") $\exists(a,\iota,c) \equiv \exists(a,\exists^{-1}(\iota,c))$

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

Construction 2 (Partial shift $\mathcal{P}_{x,y}$) *Given monomorphisms $x : P \hookrightarrow H$ 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 $\mathcal{P}_{f,y}$ to the variables of the left hand sides with all possible morphisms f from R, as below, 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. $\mathcal{P}_{x,y}(\mathtt{x}_i) := \mathtt{x}_i^{x,y}$ if $\mathtt{x}_i : P$, where $\mathtt{x}_i^{x,y} : H$ is a new variable, $H = \mathrm{cod}(y)$. Quantification is processed separately from unselection (Lemma 3): $\mathcal{P}_{x,y}(\exists(P \overset{a}{\hookrightarrow} C', c')) = \bigvee_{\mathrm{Epi}} \exists(H \hookrightarrow E, \mathcal{P}_{b,r\circ y}(c'))$, where* Epi *is the set of all epimorphisms e with*

domain H' that compose to monomorphisms $r = e \circ x'$ and $b = e \circ h$ with the pushout morphisms (diagram below left). $\mathcal{P}_{x,y}(\exists^{-1}(C' \xleftarrow{\iota} C, c')) = \exists^{-1}(\iota', \mathcal{P}_{i,y'}(c'))$: form the pullback of $r \circ \iota$ and $b \circ y$, then pushout the obtained morphisms to (y', i) (diagram below right):



Remark 8 ((Un)ambiguous Variable Contexts) *Note that 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 $R \hookrightarrow P_i$ (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 $R \hookrightarrow P_i$ for all variables (namely the morphisms from the sequences $\mathcal{X}_{R,c}$), and this property 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 μ-condition.*

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 $A_m(\mathtt{x})$ to be $\exists(id_{\mathrm{cod}(m)}, m, \mathtt{x})$ if $\mathtt{x} : P$. 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 graph 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 4 (Removal of Unselection) *If c' is a condition with placeholders, then $\exists(a, \iota, c') \equiv \exists(a, A(\iota, c'))$ (A being Pennemann's shift as described earlier in this subsection).*

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 5 (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]) □

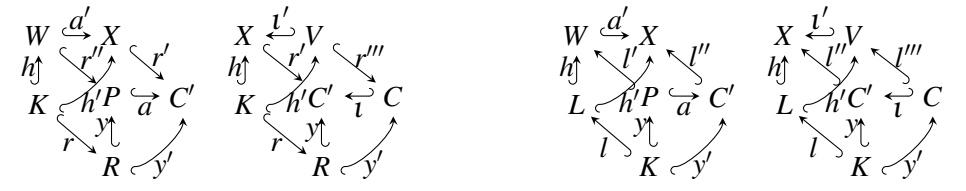
Lemma 6 *The conditions $\mathcal{P}_{x,y}(c)$ and $A_x(c)$ are equivalent.*

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 4
$\Leftrightarrow m \models \bigvee \exists(b,A_{e\circ x'}(A_\iota(c')))$ by Lemma 5.4 from [Pen09]
$\Leftrightarrow m \models \bigvee \exists(b,A_{e\circ\iota'}(A_i(c'))$ by Lemma 5 (twice) $\Leftrightarrow m \models \bigvee \exists(b,e\circ\iota',A_i(c'))$ by Lemma 4
$\Leftrightarrow m \models \bigvee \exists(b,e\circ\iota',\mathcal{P}_{i,r'}(c'))$ by induction hypothesis $\Leftrightarrow m \models \mathcal{P}_{x,r'}(\exists(a,\iota,c'))$ by Constr. 2 □

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 wlp with respect to addition and deletion, respectively[5], of a μ-condition that has already undergone partial shift. Recall the statement of Rem. 8 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 $\mathrm{cod}(r)$ (the $Add(r)$ step's input interface) instead of $\mathrm{dom}(r)$ (the output interface), which is intuitively why it yields the correct expression of the wlp of c with respect to $Add(r)$. It might well be that an occurrence of $\mathrm{cod}(r)$ 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 $\delta'_l(c)$, $\mathrm{cod}(l)$ takes the place of $\mathrm{dom}(l)$ since this corresponds exactly to the effect of the step $Del(l)$.[6]

Definition 11 (Transformations δ' and α') Let $c : P$ be a condition with placeholders. If $r\colon K \hookrightarrow R$ and $y\colon R \hookrightarrow P$ (resp. $l\colon K \hookrightarrow L$ and $y\colon K \hookrightarrow P$) are monomorphisms, then $\delta_{r,y}(c)$ ($\alpha_{l,y}(c)$) is defined as follows: $\delta_{r,y}(\neg c) = \neg\delta_{r,y}(c)$ and $\delta_{r,y}(\bigvee_{j\in J} c_j) = \bigvee_{j\in J}\delta_{r,y}(c_j)$ (respectively: $\alpha_{l,y}(\neg c) = \neg\alpha_{l,y}(c)$ and $\alpha_{l,y}(\bigvee_{j\in J} c_j) = \bigvee_{j\in J}\alpha_{l,y}(c_j)$). For $c = \exists(a,\iota,c')$, decompose using Lemma 3:[7]



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

Case of $\delta_{r,y}(\exists(a,c))$: if no pushout complement of r and $y' = a\circ y$ exists, then $\delta_{r,y}(c) = \bot$. Otherwise, obtain it as (x',h') 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 the special pushout-pullback lemma [EEPT06] 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. $\delta_{r,y}(\exists(a,c)) = \exists(a',\delta_{r,y'}(c'))$. Case of $\delta_{r,y}(\exists^{-1}(\iota,c))$: Pullback (ι,r') to (ι',r'''). The pullback property yields existence and uniqueness of $h' : K \to V$ to make the diagram commute. $\delta_{r,y}(\exists^{-1}(a,c)) = \exists^{-1}(\iota',\delta_{r,y'}(c))$.

Case of $\alpha_{l,y}(\exists(a,c))$: pushout (y,l) to (l',h); pushout (l',a) to (l'',a'). h' is obtained as $a'\circ h$

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

[6] 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 wlp, a nested condition expressing the applicability of $Del(l)$ must be adjoined to $\delta'_l(c)$.

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

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

For variables, $\delta_{r,y}(\mathtt{x_i}) = \mathtt{x'_i}$ is a new variable of type K, likewise $\alpha_l(\mathtt{x_i})$ has type L (see Rem. 8). Finally, $\delta'_r(c) = \delta_{r,id}(\mathcal{P}_{id,id}(c))$ and $\alpha'_l(c) = \alpha_{l,id}(\mathcal{P}_{id,id}(c))$.

In contrast to $\mathcal{P}$, the transformations α' and δ' leave the number of variables unchanged. Only the types of the variables are modified. We recall that for any $l : K \hookrightarrow L$, there is a condition $\Delta(l)$ that expresses the possibility of effecting $Del(l)$, i.e. $\Delta(l)$ is satisfied exactly by the first components of tuples in $[\![Del(l)]\!]$. 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 theoretical framework of double-pushout rewriting [EEPT06].

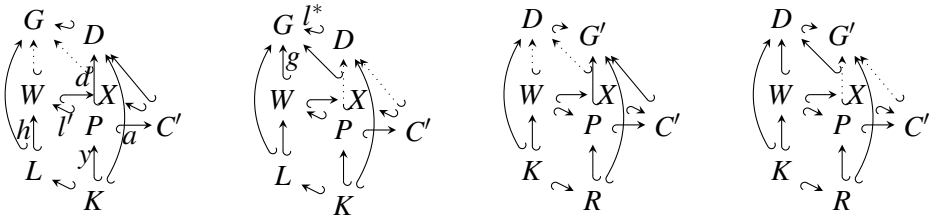
Theorem 2 (Weakest Liberal Precondition for μ-conditions) *For each rule ρ, there is a transformation Wlp_ρ that transforms μ-conditions to μ-conditions and assigns to each condition c such that $m' \models c$ another condition $\mathrm{Wlp}_\rho(c)$ such that $m \models \mathrm{Wlp}_\rho(c)$ whenever $(m,m',p) \in [\![\rho]\!]$ and $\mathrm{Wlp}_\rho(c)$ is the least condition with respect to implication having this property.*

Proof. We exhibit and prove the transformation in four steps, which compose to the full rule.

1. $\mathrm{Wlp}(Uns(y),c) = \exists^{-1}(y,c)$
2. $\mathrm{Wlp}(Add(r),c) = \delta'_r(b) \mid \mu\vec{\mathtt{x}}' = \vec{\mathcal{F}}'(\vec{\mathtt{x}}')$ where $c = b \mid \mu\vec{\mathtt{x}} = \vec{\mathcal{F}}(\vec{\mathtt{x}})$, and $\vec{\mathtt{x}}'$, $\vec{\mathcal{F}}'$ are obtained by applying δ'_r to the main body and the equations (adapting the variable types).
3. $\mathrm{Wlp}(Del(l),c) = \Delta(l) \Rightarrow \alpha'_l(c) \mid \mu\vec{\mathtt{x}}' = \vec{\mathcal{F}}'(\vec{\mathtt{x}}')$, new equations analogous to $Add(r)$
4. $\mathrm{Wlp}(Sel(x,c'),c) = \neg\exists(x,(c' \wedge \neg c))$

The proof for $Sel(x,c')$ is exactly as in [Pen09], while correctness of the first step, $Uns(y)$, is immediate from the semantics. For steps 2 and 3, we proceed by inductively comparing c to $\mathrm{Wlp}(Del(l),c)$ resp. $\mathrm{Wlp}(Add(r),c)$, which in turn requires inductively comparing conditions with placeholders that appear in the main body and right hand sides. The outer induction over $\mathbb{N}$ (see Rem. 3) 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.

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. The diagrams depict the situation in each case (deletion and addition). The names of the morphisms are as in Def. 11. Dotted arrows represent the morphisms whose existence must be shown. In all four cases, the induction hypothesis asserts correctness of the weakest precondition construction for the morphisms named $a \circ y$ and $a' \circ h$ in Def. 11 and the induction step concludes correctness at y and h; the ι part is much easier: composition is unequivocal and the appropriate morphisms from the program interface to the graphs $P, C' \ldots$ $(W, X \ldots)$ again exist to advance the induction.



Case $[\![Del(l)]\!]$ / α' Case $[\![Add(r)]\!]$ / δ'

$(m_{in}, m_{out}, l^{-1}) \in [\![Del(l)]\!]$, $m_{out} : K \hookrightarrow D$, $m_{in} : L \hookrightarrow G$ (as depicted in the diagram) and $m_{out} \models \exists(a, \iota, c)$ via d: consider $\alpha_{l,y}(c)$, where y is the morphism $K \hookrightarrow P$ obtained from the partial shift construction. Build the pushout over (l', d) and compose it with the lower pushout square, which yields the outer pushout by uniqueness. The morphism $g : W \hookrightarrow G$ obtained in the pushout and the unique commuting morphism $q : X \hookrightarrow G$ yield satisfaction.

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

$(m_{in}, m_{out}, r) \in [\![Add(r)]\!]$, $m_{out} : R \hookrightarrow G'$, $m_{in} : K \hookrightarrow D$ and $m_{out} \models \exists(a, \iota, c)$ via g': pullback $D \hookrightarrow G'$ and $P \hookrightarrow G'$ with object W', then use 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 $W \cong W'$.

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

In the case of $Del(l)$, the condition for the pushout complement required by the semantics to exist is precisely $\Delta(l)$. In the case of $Add(r)$, 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 $m_{out} \models c \Leftrightarrow m_{in} \models \mathrm{Wlp}(Del(l), c)$, resp. $m_{out} \models c \Leftrightarrow m_{in} \models \mathrm{Wlp}(Add(r), c)$ under the given circumstances. □

3.3 A Weakest Liberal Precondition Example

In this subsection, we construct a weakest liberal precondition of a μ-condition step by step. Fig. 3 shows a single-rule graph program which matches a node with exactly one incoming and one outgoing edge and replaces this by a single edge. The effect of the rule is to contract paths, and it can be applied as long as no other edges are attached to the middle node. Fig. 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. In Fig. 7 and Fig. 8, a partial shift has been applied to the condition $(\mathrm{Wlp}(Uns_c, c_4))$ of Fig. 5, and the modifications the condition undergoes in the computation of the weakest precondition with respect to Add_c and Del_c are highlighted in various colours (see Fig. 6 for a legend). Constr. 1 has yielded a new list of variables[8], $\mathtt{x}_1, \ldots, \mathtt{x}_7$,

[8] Although the original μ-condition had only one variable, partial shift usually yields one with multiple variables.

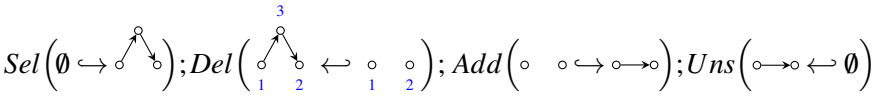


Figure 3: A path-contracting rule $\rho_{\text{contract}} = Sel_c; Del_c; Add_c; Uns_c$.

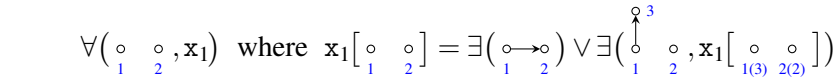


Figure 4: A μ-condition $c_4 = (b, l)$ expressing connectedness.

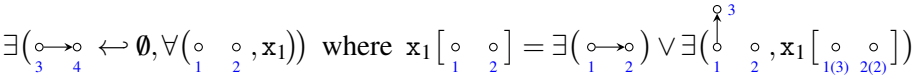


Figure 5: $\text{Wlp}(Uns_c, c_4)$. The nodes under the universal quantifier are not the same as those of the existential one, as these have been unselected: the type of the subcondition $\forall(...)$ is $\emptyset$.

the corresponding equations are shown in Fig. 8, in abbreviated notation: variable types are suppressed in subconditions $\exists(a, \iota, x_i)$ if the mapping ι from the type graph to the target of a is the identity. No other simplifications were applied. We have highlighted the type of the main body of $\text{Wlp}(Uns_c, c_4)$ throughout Fig. 7: edges drawn in red are deleted to compute $\text{Wlp}(Add_c, \text{Wlp}(Uns_c, c_4))$ as per Def. 11; the green edges and nodes, not present initially, are added to compute $\text{Wlp}(Del_c, \text{Wlp}(Add_c, \text{Wlp}(Uns_c, c_4)))$ as per Def. 11, which is obtained by adjoining $\Delta(l) \Rightarrow$ to the main body ($\Delta(l)$ omitted in Fig. 7 as it is straightforward to compute); the yellow nodes belong to both the red and green sets. A universal quantifier with $\emptyset \hookrightarrow L$ completes the weakest precondition with respect to the rule, as for nested conditions [Pen09].

When following the construction through the nesting levels, please keep in mind that one may sometimes choose among isomorphic pushout objects and that the numbers of new nodes are arbitrary, but the nodes 1, 2 and (as created by the transformation α') 5 are never "unselected" and therefore present in every type graph occurring in the weakest preconditions, similarly for the edges (not numbered because their mapping is unambiguous in the example).

4 Correctness Relative to μ-conditions

In the previous section, 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 $\bot$.

4.1 A Proof Calculus for μ-conditions

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 published [LO14]. The resolution-style proof rules of $\mathcal{K}$ are clearly sound for μ-conditions as well. In our calculus

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 6: Legend for the partial shift and weakest precondition example.

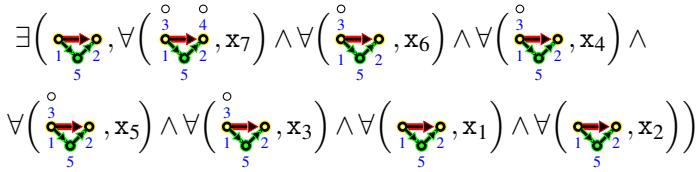


Figure 7: Construction of $\mathrm{Wlp}(Del_c;Add_c;Uns_c,\rho_c)$: application of ${\delta'}_r$ and ${\alpha'}_l$ to the main body.

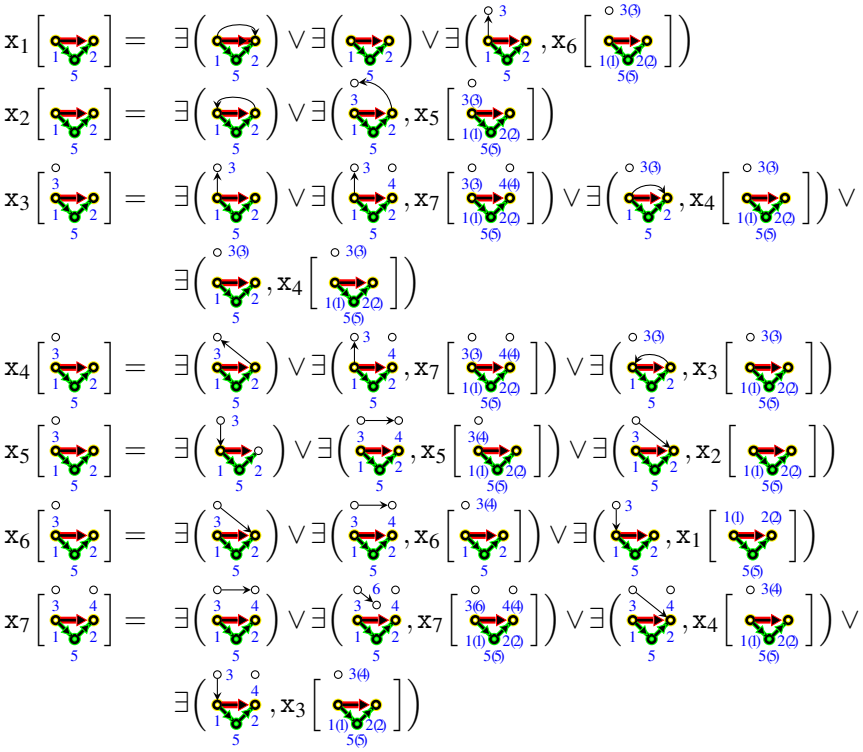


Figure 8: Construction of $\mathrm{Wlp}(Del_c;Add_c;Uns_c,\rho_c)$: equations for the variables.

$\mathcal{K}_\mu$ we adopt all rules of $\mathcal{K}$. 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 deducing contradictions at the innermost nesting levels) is not applicable in the presence of recursion. Instead, we add all Boolean manipulations as rules, and propose an induction rule to deal with situations involving fixed points. This proved 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 expres-

sion 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 x of same type as c, c':

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

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, sound by the fixed point semantics, allows unrolling x_i to the i-th component $\mathcal{F}_i(\vec{\mathrm{x}})$. When used inside a nested context via Rule Ctx, it replaces a specific occurrence of a variable by its right hand side:

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

In this rule, the annotations of the variables in the new expression are decremented: when $f \models \mathcal{F}_i^{(n)}(\vec{\bot})$, then it satisfies x_i in the *next* step of the fixed point iteration (cf. Th. 1), hence in the conclusion the variables used in the right hand side are all annotated with $(n-1)$. We detect absurdity 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) in a recursive refutation rule:

$$\frac{\forall i \in I . \mathcal{H}_i(\vec{\mathrm{x}}^{(\vec{n})}) \vdash \vec{\mathcal{G}}(\vec{\mathcal{H}}(\vec{\mathrm{x}}^{(\vec{n'})})) \qquad \vec{\mathcal{G}}(\vec{\bot}) = \vec{\bot}}{\bigvee_{i \in I} . \mathcal{H}_i(\vec{\mathrm{x}}) = \bot \qquad \vec{n'} < \vec{n};\ \vec{\mathcal{G}} \text{ monotonic}; < \text{ well-founded.}} \quad (\textsc{Empty})$$

Rule Empty is sound: if one can to find suitable $I, \vec{\mathcal{H}}, \vec{\mathcal{G}}$, then induction over $\vec{n}$ shows that at any level of the fixed point iteration, the expressions $\mathcal{H}_i(\vec{x})$ imply absurdity.

A useful instantiation is based on defining conjuncts $\mathcal{H}_{i,j} := \exists^{-1}(\iota_i, \mathrm{x}_i) \wedge \neg\exists^{-1}(\iota_j, \mathrm{y}_j)$ where x_i and 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 lfp, 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 [Gen35] not involving quantifiers.

As well as the major rules presented above, we use rules from $\mathcal{K}$: the partial resolve rule is unchanged ($\frac{\neg\exists(a) \wedge \exists(b,d)}{\exists(m^*)}$ for $a = m \circ b$ and (m^*, b^*) the $\mathcal{M}$-pushout complement of (b,m), $d \not\equiv \bot$), the (supporting) lift rules without automatic application of shift are merely $\frac{\exists(a,c) \wedge d}{\exists(a,c \wedge \exists^{-1}(a,d))}$. We also

use the classical rules for Boolean logic [Gen35], 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 $\frac{\exists(a,c)}{r_a(c)}$ (mirroring Lemma 2, r_a is defined as in the proof of that lemma except that a need not be an isomorphism); a partial shift rule ($\frac{\exists^{-1}(\iota,c)}{A(\iota,c)}$) which is correct by Lemma 4.

Theorem 3 *The calculus* $\mathcal{K}_\mu := \mathcal{K} \cup \{\textsc{Empty}, \textsc{Unroll}_1\} \cup (\text{classical+structural rules})$ *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. □

4.2 A Proof Example

For this subsection, we have opted for a new minimal example without the blowup from the weakest liberal precondition in Subsection 3.3. The example (Fig. 9) 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 $x_1 \wedge \neg x_2$, whose main body has type $\left[\begin{smallmatrix}\circ & \circ\\ 1 & 2\end{smallmatrix}\right]$. Consider the following system $\mathcal{F}$:

$$x_1\left[\begin{smallmatrix}\circ & \circ\\ 1 & 2\end{smallmatrix}\right] = \exists\left(\begin{smallmatrix}\circ\!\!\rightarrow\!\!\circ\\ 1\quad 2\end{smallmatrix}\right) \vee \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_1\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right) \;;\; x_2\left[\begin{smallmatrix}\circ & \circ\\ 1 & 2\end{smallmatrix}\right] = \exists\left(\begin{smallmatrix}\circ\!\!\rightarrow\!\!\circ\\ 1\quad 2\end{smallmatrix}\right) \vee \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_2\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right)$$

While the equations are syntactically identical up to variable renaming, this is not exploited by $\mathcal{K}_\mu$, hence the proof is not a one-liner: it starts by defining a suitable list of auxiliary conditions

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{F} : x_1^n \wedge \neg x_2^m \vdash \mathcal{F}_1(\vec{x}^{(n-1)}) \wedge \neg \mathcal{F}_2(\vec{x}^{(n-1)}) \quad (\mathcal{H}_{1,2}(\vec{x}) = x_1 \wedge \neg x_2)}{\mathcal{F} : x_1^{(n)} \wedge \neg x_2^{(n)} \vdash \left(\exists\left(\begin{smallmatrix}\circ\!\!\rightarrow\!\!\circ\\ 1\quad 2\end{smallmatrix}\right) \vee \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_1^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right)\right) \wedge \neg\exists\left(\begin{smallmatrix}\circ\!\!\rightarrow\!\!\circ\\ 1\quad 2\end{smallmatrix}\right) \wedge \neg\exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_2^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right)}\,(1)}{\mathcal{F}' : \ldots \vdash \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_1^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right) \wedge \neg\exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_2^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right)}\,(2)}{\mathcal{F}' : \ldots \vdash \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_1^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right] \wedge \neg\exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_2^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right)\right)}\,(3) \qquad \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, \bot\right) \vdash \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, \bot\right)}{\mathcal{F}' : x_1^{(n)} \wedge \neg x_2^{(n)} \vdash \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, x_1^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right] \wedge \neg x_2^{(n-1)}\left[\begin{smallmatrix}\circ & \circ\\ 1(3) & 2(2)\end{smallmatrix}\right]\right)}\,(4) \qquad \exists\left(\begin{smallmatrix}\overset{3}{\circ}\\ \uparrow\\ 1\end{smallmatrix}\ \begin{smallmatrix}\circ\\ 2\end{smallmatrix}, \bot\right) \vdash \bot}{\mathcal{F} : x_1 \wedge \neg x_2 \vdash \bot}$$

Figure 9: Deducing a contradiction from $x_1 \wedge \neg x_2$ under the system of equations $\mathcal{F}$. Multiple steps have been contracted into single inference lines for the sake of brevity.

$\vec{\mathcal{H}}$ (in this case actually a single one, which we name $\mathcal{H}_{1,2}$[9]), unrolling both variables once (1), then uses distributivity of conjunction over disjunction to resolve the base case of the right hand side of x_1 (2), then shifts the other branch of x_2 over the corresponding branch of 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 $x_1 \wedge \neg x_2$, with lower annotations, as a subcondition. Finally we show that the context of this subcondition (monotonic by virtue of being syntactically positive) has lfp $\bot$, and apply Rule EMPTY.

5 Related Work

A summary overview of graph conditions for non-local properties is attempted below (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). Note that while 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, we have not yet been able to separate μ-conditions from MSO or HR*:

reference	[Pen09]	(here)	[Rad13]	[PP14]
conditions	Nested	μ-	HR*	MSO-
wlp	yes	yes	incomplete[10]	yes
theorem prover	yes	future work		
complete proof calculus	yes	future work		

Recently, 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. HR* conditions [Rad13] are another approach towards the same goal; they have already been mentioned in the main text; there is an ongoing effort at extending the weakest precondition calculus to a subclass including path expressions. 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. Verification of graph transformation systems via abstract model checking, as opposed to the prover-based approach, can be found in Gadducci et al., Baldan et al., König et al., Rensink et al. [GHK98, BKK03, KK06, RD06].

6 Conclusion and Outlook

We have introduced μ-conditions and achieved a weakest liberal precondition transformation (Th. 2) and a sound proof calculus (Th. 3) for correctness relative to μ-conditions, which seems a fruitful ground for further investigations. In analogy to the equivalence between first-order

[9] Note that a larger example would likely have required more than one branch to handle each conjunct.
[10] Radke, personal communication: construction only partially defined.

graph logic and nested graph conditions, we conjecture that μ-conditions have the same expressivity as fixed point extensions to first-order logic on finite graphs. Also, the expressivity of HR* conditions [Rad13] or MSO likely differs from μ-conditions, but this remains to be examined. As the examples show, our weakest precondition calculus (still a research question for HR* conditions [Rad13] but readily available by logical means in the MSO-conditions formalism [PP14]) produces unwieldy expressions due to partial shift. The blowup is exponential in the interface size (a related blowup is inherited from the weakest precondition calculus of [Pen09]). We can heuristically simplify the expressions and hope that many cases can be resolved automatically.

Future work includes more proof theory and tool support with special attention to semi-automated reasoning, based on the reasoning engine ENFORCE implemented in [Pen09]. To extend the wlp construction to programs with iteration, one would have to provide (or have the prover attempt to find) an invariant, as in the original work of Pennemann; for termination, one could proceed as in [Pos13] and prove termination variants. It 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 via a *next* operator parameterised on atomic subprograms (the basic steps of Def. 3) and since in the semantics of programs 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, and allow the expression of eventualities as in the modal μ-calculus[BS07]. Whether this offers any new insights remains to be seen. We plan to deal with algebraic operations on attributes and extend our work to a 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 with ordered elements.

Acknowledgements: We thank Annegret Habel, many other members of SCARE as well as several anonymous reviewers for providing valuable criticism of the approach and the text.

Bibliography

[BKK03] P. Baldan, B. König, B. König. A logic for analyzing abstractions of graph transformation systems. In *Static Analysis*. Pp. 255–272. Springer, 2003.

[BS07] J. Bradfield, C. Stirling. Modal μ-calculi. *Studies in Logic and Practical Reasoning* 3:721–756, 2007.

[DG15] J. Dyck, H. Giese. Inductive Invariant Checking with Partial Negative Application Conditions. In *ICGT*. LNCS 9151, pp. 237–253. 2015.

[Dij76] E. W. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.

[Gen35] G. Gentzen. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift* 39(1):176–210, 1935.

[GHK98] F. Gadducci, R. Heckel, M. Koch. A Fully Abstract Model for Graph-Interpreted Temporal Logic. In *TAGT'98*. LNCS 1764, pp. 310–322. 1998.

[Hoa83] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM* 26(1):53–56, 1983.

[HP09] A. Habel, K.-H. Pennemann. Correctness of High-Level Transformation Systems Relative to Nested Conditions. *Math. Struct. in Comp. Sci.* 19(2):245–296, 2009.

[HPR06] A. Habel, K.-H. Pennemann, A. Rensink. Weakest Preconditions for High-Level Programs. In *ICGT 2006*. LNCS 4178, pp. 445–460. 2006.

[KK06] B. König, V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. LNCS 3920, pp. 197–211. 2006.

[Kre02] S. Kreutzer. *Pure and Applied Fixed-Point Logics*. PhD thesis, Dissertation thesis, RWTH Aachen, 2002.

[LO14] L. Lambers, F. Orejas. Tableau-Based Reasoning for Graph Properties. In *Graph Transformation*. LNCS 8571, pp. 17–32. 2014.

[Pen09] K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.

[Pos13] C. M. Poskitt. *Verification of Graph Programs*. PhD thesis, University of York, 2013.

[PP13] C. M. Poskitt, D. Plump. Verifying Total Correctness of Graph Programs. *Electronic Communications of the EASST* 61, 2013.

[PP14] C. M. Poskitt, D. Plump. Verifying Monadic Second-Order Properties of Graph Programs. In *Graph Transformation*. LNCS 8571, pp. 33–48. 2014.

[PST13] C. Percebois, M. Strecker, H. N. Tran. Rule-Level Verification of Graph Transformations for Invariants Based on Edges' Transitive Closure. In *SEFM 2013*. LNCS 8137, pp. 106–121. 2013.

[RAB+15] H. Radke, T. Arendt, J. Becker, A. Habel, G. Taentzer. Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations. In *Proc. ICGT*. LNCS 9151, pp. 155–170. 2015.

[Rad13] H. Radke. HR* Graph Conditions Between Counting Monadic Second-Order and Second-Order Graph Formulas. *Electronic Communications of the EASST* 61, 2013.

[RD06] A. Rensink, D. Distefano. Abstract graph transformation. *ENTCS* 157:39–59, 2006.

[Str08] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *ENTCS* 203(1):135–148, 2008.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5(2):285–309, 1955.