On Correctness of Graph Programs Relative to Recursively Nested Conditions

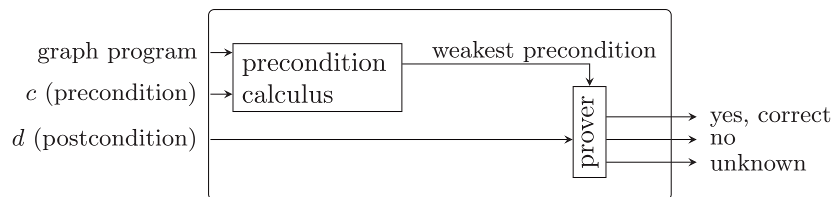
Nils Erik Flick*

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 path properties. 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.

1 Introduction

Graph transformations provide a formal way to model the graph-based behaviour of a wide range of systems by way of diagrams. Such systems can be formally verified. One approach to verification proceeds via model checking of abstractions, notably Gadducci et al., Baldan et al., König et al., Rensink et al. [4, 1, 10, 18]. This can be contrasted with the proof-based approaches of Habel, Pennemann and Rensink [7, 6] and Poskitt and Plump [15]. 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:



The correctness proof is done in the style of Dijkstra's [2] predicate transformer approach in Pennemann's thesis [12], while Poskitt's thesis [14] features a Hoare [8] 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 [12].

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

Our formalism is an extension of nested conditions by recursive definitions. While several extensions of nested graph conditions to non-local conditions already exist (Radke [17], Poskitt and Plump [16]), we argue that as opposed to the former, ours already 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. Therefore μ-conditions, albeit still work in progress, offer a new viewpoint that may be sufficiently different from existing ones to be worth investigating.

The outline of the paper is as follows: Section 2 recalls graph programs and conditions. Sections 3 and 4 introduce μ-conditions and correctness under μ-conditions, respectively, together with an exemplary application of the method, Section 5 provides context by listing related work and Section 6 concludes with an outlook. After the main text, there is an appendix with the proofs.

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. [3], and the basic notions of category theory. For standard definitions and more details, we refer the reader to Ehrig et al. [3]. For an in-depth introduction to nested conditions and graph programs and practical approaches to semi-automatic theorem proving in this context, we refer the reader to Pennemann [12].

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 (monomorphisms) are distinguished typographically by a curly arrow $f : G \hookrightarrow H$ while double-tailed arrows $f : G \twoheadrightarrow H$ denote surjective ones (epimorphisms). We use the symbol $\mathcal{M}$ to denote the class of all graph monomorphisms. A *partial morphism* is a pair of monomorphisms with the same domain. The empty graph is denoted by $\emptyset$.

All graphs in this paper are assumed to be finite.

A brief review of nested conditions follows. Nested graph conditions were proposed by Habel and Pennemann. Finite nested conditions were later shown to be equally expressive as graph-interpreted first-order predicate logic. Graph conditions can be used as constraints to specify state properties, or as application conditions to restrict the applicability of a rule.

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).

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.

Notation. If c is a condition over P, then P is its *type*[1] and we write $c : P$, and Cond_P is the class of all conditions over P. We may write $\exists(a, c)$ instead of $\exists(a, id_{\mathrm{cod}(a)}, c)$. 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. We introduce *false* as a notation for it, and *true* for $\neg$*false*. We may omit the subcondition *true* (together with ι), writing $\exists(a)$ for $\exists(a, \iota, \mathit{true})$.

When all the index sets are finite, one obtains the *finite* nested conditions. The morphism ι serves to unselect[2] a part of C', which will become necessary later.

Definition 2 (Satisfaction). A monomorphism $f : P \hookrightarrow G$ *satisfies* a condition $c : P$, denoted $f \models c$, iff $c = \mathit{true}$, $c = \neg c'$ 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 $c = \exists(a, \iota, c')$ ($a : P \hookrightarrow C'$, $\iota : C \hookrightarrow C'$, $c' : C$) and there exists a monomorphism $q : C' \hookrightarrow G$ such that $f = q \circ a$ and $q \circ \iota \models c'$.

$$\exists(P \xhookrightarrow{a} C' \xhookleftarrow{\iota} C, \triangleleft c' \,) \qquad f\downarrow \; q \quad q \circ \iota \quad \models \qquad G$$

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

In the diagram of Def. 2, the triangle indicates that C is the type of the subcondition c' which appears nested inside $\exists(\alpha, \iota, c')$.

Remark 1 (No Added Expressivity)**.** Our conditions with ι are equally expressive as the nested conditions defined in [12]. The proof, which we omit here, relies on the transformation A from [12].

Definition 3. $\equiv$ denotes logical equivalence, i.e. for conditions $c, c' : P$, $c \equiv c'$ iff for all monomorphisms m with domain P, $\Rightarrow m \models c \Leftrightarrow m \models c'$.

Notation. As one can see in Fig. 1, the notation for graph conditions customarily only depicts source or target graphs of morphisms. The tiny blue numbers

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

[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".

$\models \quad \exists\left(\overset{1}{\circ}\!\rightarrow\!\overset{2}{\circ} \hookleftarrow \overset{2}{\circ}, \neg\exists\left(\overset{2}{\circ} \right)\right)$

Fig. 1. A nested graph condition (stating the existence of two nodes linked by an edge, where the second node does not have a self-loop) and a graph satisfying it.

show the morphisms' node mappings. We also adopt the convention of not explicitly representing the morphism ι in a situation $\exists(a, \iota, \mathtt{x}_i)$; we prefer to annotate the variable's type graph with the images of items under ι in parentheses.

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 by sets of graph transformation rules.

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 program respectively, and the partial morphism records the changes effected by the program. Our programs are a proper subset of those in Pennemann [12], and use the same semantics.

Definition 4 (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 m_{in} and m_{out} are called *interfaces* and universally quantified in the set comprehensions that appear in the definitions below.

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, then so are their disjunctive $\{P, Q\}$ and sequential $(P; Q)$ composition. 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 composition $p'; p''$ of partial morphisms $p' = G_1 \overset{l_1}{\hookleftarrow} D_1 \overset{r_1}{\hookrightarrow} H_1$, $p'' = H_1 \overset{l_2}{\hookleftarrow} D'_2 \overset{r_2}{\hookrightarrow} H_2$ is defined as $G \overset{l_1 \circ l'_2}{\hookleftarrow} D'' \overset{r_2 \circ r'_1}{\hookrightarrow} H_2$ using the pullback (r'_1, l'_2) of (r_1, l_2). If P is a graph program, then 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 $P^0 = skip$.

Remark 2. The definitions generalise the state transitions in plain graph transformation, a rule $\varrho = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ being precisely 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. These are capable of expressing 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 are unable to 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 will be similar to first order fixed point logics, see e.g. Kreutzer [9]. The reader might want to compare our μ-conditions to a distinct formalism towards expressing non-local properties, the very powerful grammar-based HR* conditions of Radke [17]. We argue that μ-conditions are worth looking into despite the availability of strong contenders for the extension of nested conditions to non-local properties, such as MSO-conditions [16] because μ-conditions provide a new and different generalisation of nested conditions, and neither is it immediately clear how the respective expressivities compare. The related work section, Sec. 5, contains a summary on different non-local graph condition formalisms. Specifically, we will show in this section that the weakest liberal precondition transformation, core of the Dijkstra-style approach, can be adapted.

Notation. Sequences (of graphs, placeholders, morphisms) are written as bold letters $\boldsymbol{P}$, $\mathbf{x}$, $\boldsymbol{f}$, and their components are numbered starting from 1. The length of a sequence $\boldsymbol{P}$ is denoted by $\|P\|$. 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: $\boldsymbol{c} : \boldsymbol{P}$ (provided $\|c\| = \|P\|$).

To define fixed point conditions, we need something to take fixed points of, and to ascertain that the fixed point exists. Choosing a partial order on $\mathrm{Cond}_{\boldsymbol{P}}$, one can define monotonic operators on $\mathrm{Cond}_{\boldsymbol{P}}$. The semantics of satisfaction already defines a pre-order: $c \leq c'$ iff 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[3].

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

To represent systems of simultaneous equations, we work on tuples of conditions. If $\boldsymbol{P} = P_1, \ldots, P_{\|P\|}$ is a sequence of graphs, then $\mathrm{Cond}_{\boldsymbol{P}}$ is the set of all $\|P\|$-tuples $\boldsymbol{c}$ of conditions, whose i-th element is a condition over the i-th graph of $\boldsymbol{P}$. Satisfaction is defined component-wise: $\boldsymbol{f} \models \boldsymbol{c}$ iff $\forall k \in \{1, \ldots, \|P\|\}\ f_k \models c_k$.

By definition, $\bigwedge$ and $\bigvee$ of countable sets of Cond_P conditions exist for any P, and they are easily seen to be least upper and greatest lower bounds of the sets. This makes $\mathrm{Cond}_{\equiv P}$ a complete lattice. Let $\mathrm{Cond}_{\boldsymbol{P}}$ be ordered with the product order by defining $\boldsymbol{f} \models \boldsymbol{c}$ to be true when the conjunction holds. This again induces a partial order on the set of equivalence classes, $\mathrm{Cond}_{\equiv \boldsymbol{P}}$. Thus, $\mathrm{Cond}_{\equiv \boldsymbol{P}}$ is also a complete lattice, and monotonic operators have least fixed points by the Knaster-Tarski theorem [20], given by the limit of $\boldsymbol{\mathcal{F}}^n(\boldsymbol{false})$ for all $n \in \mathcal{N}$. This ensures that systems of equations as defined below yield least fixed point solutions, which is crucial in the definition of a μ-condition. We extend the inductive definition from Def. 1 as follows:

Definition 5 (Graph Conditions with Placeholders). Given a graph P and a finite sequence $\boldsymbol{P}$ of graphs or morphisms, a *condition with placeholders from* $\boldsymbol{P}$ 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 $\mathtt{x}_i$, $1 \leq i \leq \|\boldsymbol{P}\|$ where $\mathtt{x}_i$ is a variable of type P_i.

A condition can be substituted for a variable of same type:

Definition 6 (Substitution). If $\boldsymbol{P}$ is a list of graphs and $\mathcal{F}$ is a condition with placeholders $\mathtt{x}$ over $\boldsymbol{P}$, then if $\boldsymbol{c} \in \mathit{Cond}_{\boldsymbol{P}}$, then $\mathcal{F}[\mathtt{x}/\boldsymbol{c}]$ is obtained by substituting each occurrence of $\mathtt{x}_i$ by c_i for all $i \in \{1, ..., \|\mathcal{P}\|\}$.

Satisfaction of conditions with placeholders by a morphism f is defined in the obvious way relative to a *valuation*, which is an assignment of *true* or *false* to each monomorphism of the type graph of the variable into $\mathrm{cod}(f)$.

As discussed above, a least fixed point will be sought only up to logical equivalence. To guarantee existence of the least fixed point, the operator must be monotonic ($\boldsymbol{c} \leq \boldsymbol{d} \Rightarrow \boldsymbol{\mathcal{F}}(\boldsymbol{c}) \leq \boldsymbol{\mathcal{F}}(\boldsymbol{d})$ for any $\boldsymbol{c}, \boldsymbol{d} \in \mathit{Cond}_{\boldsymbol{P}}$). Monotonicity can be enforced syntactically for substitutions by never placing a variable under an odd number of negations, which is proved by structural induction as in fixed point logics or the modal μ calculus.

Definition 7 (μ-Condition). Given a finite list $\boldsymbol{P}$, if $\{\mathcal{F}_i\}_{i \in \{1,\ldots,\|P\|\}}$ are conditions with placeholders from $\boldsymbol{P}$, over the graphs of $\boldsymbol{P}$ respectively, then $\mu[\boldsymbol{P}]\boldsymbol{\mathcal{F}}$ denotes the least fixed point of the operator $\boldsymbol{c} \mapsto \boldsymbol{\mathcal{F}}[\mathtt{x}/\boldsymbol{c}]$.

A *μ-condition* is a pair (b, l) consisting of a condition with placeholders b, and a finite list of pairs $l = (\mathtt{x}_i, \mathcal{F}_i(\boldsymbol{x}))$ of a variable $\mathtt{x}_i : P_i$ and a condition $\mathcal{F}_i(\boldsymbol{x}) : P_i$, with placeholders from $\boldsymbol{x}$, for some graph P_i, such that $\boldsymbol{\mathcal{F}}$ is monotonic.

Notation. we write the list of pairs $l = (\mathtt{x}_i, \mathcal{F}_i(\boldsymbol{x}))$ as a system of equations $\boldsymbol{x} = \boldsymbol{\mathcal{F}}(\boldsymbol{x})$. We call b the *main body* and l the *recursive specification* of (b, l), and $\boldsymbol{\mathcal{F}}(\boldsymbol{c})$ is understood as substitution of conditions $\boldsymbol{c}$ for the variables $\mathtt{x}$.

Thus each condition with placeholders typed over $\boldsymbol{P}$ defines a unary operator on $Cond_{\boldsymbol{P}}$.

Remark 3 (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.

$$\mathtt{x}_1[\ \underset{1}{\circ}\quad \underset{2}{\circ}\ ] \text{ where } \mathtt{x}_1[\ \underset{1}{\circ}\quad \underset{2}{\circ}\ ] = \exists(\underset{1}{\circ}\!\to\!\underset{2}{\circ}) \lor \exists(\underset{1}{\circ}\!\to\!\overset{3}{\circ}\quad \underset{2}{\circ}, \mathtt{x}_1[\ \underset{1(3)}{\circ}\quad \underset{2(2)}{\circ}\ ])$$

It is read as follows: the word "where" separates the main body from the equations. Here, $\mathtt{x}_1$ is the only variable, and 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 indeed syntactically required for using the variable in that place. The unselection morphism ι is implicit in the notation, and is only expressed by adding small blue numbers in parentheses to the node numbers in its source graph to specify the mapping. This compact notation for ι is why the second existential quantifier in the example has only two fields. To ease reading and writing, we adopt the convention to always use precisely the same layout for the type graph of a given variable.

The following statement is not needed in the proofs that will follow, but it helps motivate the use of the "unselection" morphisms. We therefore view it as justified to leave the proof as an exercise:

Remark 4 (Why ι)**.** A μ-condition where ι is the identity in all subconditions of the main body and of the components $\mathcal{F}_i(\boldsymbol{x})$ is equivalent to a nested condition.

The following fact is well-known:

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

Definition 8 (Satisfaction). The μ-condition $b \mid \boldsymbol{x} = \boldsymbol{\mathcal{F}}(\boldsymbol{x})$ with $\boldsymbol{x} : \boldsymbol{P}$ is satisfied by a morphism f iff $f \models b[\boldsymbol{x}/\mu[\mathcal{P}]\mathcal{F}]$.

Remark 6 (No Infinite Nesting)**.** By the characterisation of the least fixed point as an infinite disjunction, every μ-condition is equivalent to an infinite nested condition. Infinitely deep nesting does not arise, because the characterisation in Remark 5 yields a countable disjunction of *finitely* deeply nested conditions.

A morphism satisfies a given μ-condition iff it satisfies the finite nested condition obtained by unrolling the recursive specification up to some finite depth and substituting the resulting nested conditions into the main body:

Proposition 1 (Satisfaction at Finite Recursion Depth). $f \models b \mid \boldsymbol{x} = \boldsymbol{\mathcal{F}}(\boldsymbol{x})$ *iff* $\exists n \in \mathbb{N},\ f \models b[\boldsymbol{x}/\boldsymbol{\mathcal{F}}^n(\boldsymbol{false})]$.

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

3.2 Weakest Liberal Preconditions of μ-conditions

In this subsection, we present a construction to compute the weakest liberal precondition of any given μ-condition with respect to any graph program P that does *not* use iteration ("liberal" means that termination of P is not implied, and is redundant in the absence of iteration, as only iteration causes non-termination).

Definition 9 (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 will 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 [12] whose fundamental property is to transform any nested condition c into another nested condition such that $m'' \models A_m(c)$ iff $m'' \circ m \models c$ for all composable pairs m'', m of monomorphisms (Lemma 5.4 from [12]). Since this and similar constructions play an important role in this section, we recall here 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 $A_m(\exists(a, c')) = \bigvee_{e \in Epi} \exists(b, A_r(c'))$.

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 containing the type of the main body:

Construction 1 (New type graphs for partial shift)**.**

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

The morphisms f are obtained from $\boldsymbol{P'}$ by collecting all epimorphisms that compose to monomorphisms with the pushout morphisms in the diagram:

$$\begin{array}{ccccc} \emptyset & \hookrightarrow & P_i & & \\ \downarrow & & \downarrow & \searrow & \\ R & \hookrightarrow & X & \twoheadrightarrow & P'_j \end{array} \qquad f : R \to P'_j$$

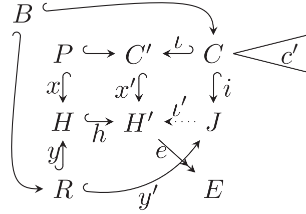
Construction 2 (Partial shift)**.**

Given monomorphisms $x : G \hookrightarrow H$ and $y : R \hookrightarrow H$,

$\mathcal{P}_{x,y}(b \mid \mu[\boldsymbol{K}]\mathcal{F}) := \mathcal{P}_{x,y}(b) \mid \mu[\boldsymbol{K'}]\mathcal{F}'$, where the new list of variables $\boldsymbol{K'}$ and their respective types $\boldsymbol{P'}$ are obtained by concatenating all $\mathcal{X}_{R,c}(\mathtt{x_i})$ of the variables of $\boldsymbol{K}$ in order. where the new variables and 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.

$\mathcal{P}_{x,y}(\mathtt{x}_i) = \mathtt{x}_i^y$ if $\mathtt{x}_i : G$, where $\mathtt{x}_i^y : H$ is a new variable, $H = \mathrm{cod}(y)$.

$\mathcal{P}_{x,y}(\exists(P \overset{a}{\hookrightarrow} C' \overset{\iota}{\hookleftarrow} C, c'))$ is constructed as follows: Let Epi be 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. $\mathcal{P}_{x,y}(\exists(P \hookrightarrow C' \hookleftarrow C, c') = \bigvee_{Epi} \exists(H \hookrightarrow E \hookleftarrow J, \mathcal{P}_{i,y'}(c'))$: for each member of the disjunction, form the pullback of $r \circ \iota$ and $b \circ y$, then pushout the obtained morphisms to (y', i) as in the diagram:



Boolean combinations of conditions are transformed to the corresponding combinations of the transformed members.

Remark 7 (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, but sometimes smaller μ-condition.

Equivalence of conditions with placeholders (unlike μ-conditions) is only defined for conditions using the same sets of variables, as equivalence in the sense of nested conditions for each 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 $\mathtt{x} : P$.

One can show that $\mathcal{P}_{x,y}$ is equivalent to A_x. The reason for introducing $\mathcal{P}_{x,y}$ is that it allows precise control over the types of the variables in the transformed condition, which should 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 make sure that the changes apply to the whole μ-condition to obtain the correct result.

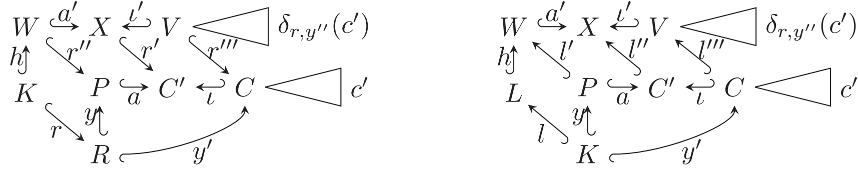
Lemma 1. The conditions $\mathcal{P}_{x,y}(c)$ and $A_x(c)$ are equivalent.

We introduce the transformations $\delta'_m(c)$, $\alpha'_m(c)$ (based on auxiliary transformations $\delta_{m,y}(c)$ and $\alpha_{m,y}(c)$, repsectively), which are used in the computation of the weakest liberal precondition (with respect to addition and deletion, respectively[4]), of a μ-condition that has already undergone partial shift:

Definition 10 (Transformations δ' and α').

Let $c : G$ be a condition with placeholders. If $r : K \hookrightarrow R$ and $y : R \hookrightarrow G$ (resp. $l : K \hookrightarrow L$ and $y : K \hookrightarrow G$) 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 $c = \exists(a, \iota, c')$, the following constructions are used:



Case of $\delta_{m,y}(c')$: If the pushout complement of r and $a \circ y$ does not exist, then $\delta_{m,y}(c) = false$. Otherwise, obtain it as x' and r' and pullback (a, r') to (a', r'') with source W; this yields a morphism h from K to W to make the diagram commute and the special PO-PB lemma [3] applicable. Pullback (ι, r') to (ι', r'''), $(x' = a' \circ h)$ and let $\delta_{m,y}(c) = \exists(a', \iota', \delta_{m,y''}(c'))$ (the pullback property yields existence and uniqueness of y'' between K and V to make it commute).

Case of $\alpha_{l,y}(c)$: pushout (y, l) to (l', h); pushout (l', a) to (l'', a'); pullback $(a' \circ h, l'' \circ \iota)$ and pushout to (y'', l''') over the pullback (not drawn in the figure except for the morphism l''''). The commuting morphism from the pushout object V to X fills in to yield $\exists(a', \iota', \alpha_{l,y}(c'))$. The commuting morphism from L to V is y''.

For variables, $\delta_{m,y}(\mathtt{x_i}) = \mathtt{x'_i}$ is a new variable of type K, likewise $\alpha_m(\mathtt{x_i})$ has type L (see Rem. 7). Finally, $\delta'_m(c) = \delta_{m,id}(\mathcal{P}_{id,id}(c))$ and $\alpha'_m(c) = \alpha_{m,id}(\mathcal{P}_{id,id}(c))$.

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 $im(f) - im(f \circ l)$.

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 $\mathrm{Wlp}_\varrho(c)$

[4] The letters were chosen so as 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.

with the property 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 having this property.

3.3 A Weakest Liberal Precondition Example

In this subsection, we construct a weakest liberal precondition of a μ-condition step by step. Figure 2 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.

$$Sel(\emptyset \hookrightarrow \cdots); Del(\cdots \hookleftarrow \cdots); Add(\cdots \hookrightarrow \cdots); Uns(\cdots \hookleftarrow \emptyset)$$

Fig. 2. A path-contracting rule $\varrho_{\mathrm{contract}} = Sel_{\mathrm{c}}; Del_{\mathrm{c}}; Add_{\mathrm{c}}; Uns_{\mathrm{c}}$.

Figure 3 shows a μ-condition whose weakest liberal precondition we wish to compute. It is a typical example of a μ-condition, which evaluates to *true* on those graphs that contain some node which has a path to every other node.

$$\exists(\circ_1, \forall(\circ_1\ \circ_2, \mathtt{x}_1)) \quad \text{where} \quad \mathtt{x}_1[\circ_1\ \circ_2] = \exists(\circ_1 \to \circ_2) \vee \exists(\circ_1 \to \circ_3\ \ \circ_2, \mathtt{x}_1[\circ_{1(3)}\ \circ_{2(2)}])$$

Fig. 3. A μ-condition $c_3 = (b, l)$ expressing the existence of a node from which there exists a path to every other node.

$$\exists(\circ_1 \to \circ_2 \hookleftarrow \emptyset, \forall(\circ_1\ \circ_2, \mathtt{x}_1)) \quad \text{where} \quad \mathtt{x}_1[\circ_1\ \circ_2] = \exists(\circ_1 \to \circ_2) \vee \exists(\circ_1 \to \circ_3\ \ \circ_2, \mathtt{x}_1[\circ_{1(3)}\ \circ_{2(2)}])$$

Fig. 4. $\mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)$. Note that the nodes under the universal quantifier are not the same as those of the outer existential quantifier, as these have been unselected: the type of the subcondition $\forall(...)$ is $\emptyset$.

In Figures 5 and 6, a partial shift has been applied to the condition of Figure 4 ($\mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)$), 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 Figure 7 for a legend). Construction 1 has yielded a new list of variables[5], $\mathtt{x}_1, ..., \mathtt{x}_7$, the corresponding equations are shown in 6. Note that the representation is somewhat further abbreviated: type graphs of variables are suppressed in the notation in subconditions $\exists(a, \iota, \mathtt{x}_i)$, when the mapping ι from the type graph to the target of a is the identity. No other simplifications were applied. We have highlighted in yellow and red the type of the main body of $\mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)$ throughout Figure 5; edges that are highlighted in red are deleted to compute $\mathrm{Wlp}(Add_{\mathrm{c}}, \mathrm{Wlp}(Uns_{\mathrm{c}}, c_3))$ as per Def. 10; the edges and nodes highlighted in red are not present initially, but added to compute $\mathrm{Wlp}(Del_{\mathrm{c}}, \mathrm{Wlp}(Add_{\mathrm{c}}, \mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)))$ as per Def. 10, which is obtained by conjoining $\Delta(l)$ to the main body (which we have not represented, as it is easy to compute and would only encumber the illustration). In the end, a universal

[5] Although the original μ-condition needed only one variable, the partial shift yields a μ-condition with multiple variables in general.

Fig. 5. Construction of $\mathrm{Wlp}(Del_c; Add_c; Uns_c, \varrho_c)$: main body (variables: see below).

Fig. 6. Construction of $\mathrm{Wlp}(Del_c; Add_c; Uns_c, \varrho_c)$: equations for the variables.

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 $Wlp(Del(l), c)$

Fig. 7. Legend for Figure 5.

quantification with morphism $\emptyset \hookrightarrow L$ completes the weakest precondition with respect to the rule, as in the construction for nested conditions [12].

The outer existential quantifier in Fig. 5, where the unselection morphism is not shown in the abbreviated representation, is really as in Fig. 8:

Fig. 8. Outer nesting level of the conditions in Fig. 5

When following the construction through the nesting levels, please keep in mind that one may sometimes choose among isomorphic pushout objects and 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 is to develop methods for deducing correctness relative to μ-conditions and extend the proof calculus, for which we offer a partial solution in this section.

The soundness of Pennemann's calculus $\mathcal{K}$ has been established in the publications introducing them, and recently a tableaux based completeness proof of $\mathcal{K}$ was published [11]. The proof rules of $\mathcal{K}$ are easily seen to be sound for μ-conditions as well, however the recursive definitions requires an extension.

For our calculus $\mathcal{K}_\mu$, we adopt the resolution-style rules of $\mathcal{K}$ and add an induction principle to deal with certain situations involving fixed points. This proved to be sufficient to handle all the situations encountered in the examples.

We employ a sequent notation: the inference rules manipulate objects $Ctx : \Gamma \vdash \Delta$, with the intended meaning that the disjunction of Δ can be deduced from the conjunction of Γ in the context Ctx. The context Ctx is a pair of a left hand side of a sequent and an operator on μ-conditions. Γ and Δ are sets of expressions, which differ from conditions in that identifiers can be used for the main bodies of μ-conditions and both these and variables can be annotated with their recursion depth, an implicitly universally quantified natural number. $\mathtt{x}_i^{(n)}$ then stands for $\mathcal{F}_i^{(n)}(\mathit{false})$ and an auxiliary rule permits to unroll it to the i-th component of $\boldsymbol{\mathcal{F}}$ applied to $\mathtt{x}^{(n-1)}$.

The induction rule announced above is (where $\mathcal{H}_{i,j}$ for each $i \in \{1, ..., \|I\|\}$, $j \in \{1, ..., \|J\|\}$ is any condition with placeholders):

$$\frac{\overbrace{\begin{array}{l}\mathtt{x}_i^{(m)} \wedge \neg \mathtt{y}_j^{(n)} \vdash \\ \mathcal{H}_{i,j}(\mathtt{x}_1^{(m-1)} \wedge \neg \mathtt{y}_1^{(n)}, ..., \mathtt{x}_{\|I\|}^{(m-1)} \wedge \neg \mathtt{y}_{\|J\|}^{(n)})\end{array}}^{\forall i,j} \qquad \begin{array}{l}\vdash \\ \forall i,j\ \mathcal{H}_{i,j}(\boldsymbol{\mathit{false}}) = \boldsymbol{\mathit{false}}\end{array}}{\bigvee \mathtt{x}_i \wedge \neg \mathtt{y}_i \vdash \mathit{false}}$$

(IndMuEmpty)

Theorem 3. $\mathcal{K}_\mu := \mathcal{K} \cup \{$IndMuEmpty$\}$ is sound.

There are a number of details hidden in the discussion above: Boolean operations must be lifted to μ-conditions, which entails variable renaming and union of the

systems of equations; rules for exploiting logical equivalences between different Boolean combinations are necessary to equivalently transform conditions into a form suitable for the application of the rules of $\mathcal{K}$; in [12], each Boolean combination appearing inside a nested condition is put into conjunctive normal form prior to the application of rules. Proof trees in the sequent-style calculus $\mathcal{K}_\mu$ start with instances of the *axiom* ($A \vdash A$ is derivable by a rule with no antecedents), and make use of all the classical sequent rules [5] not involving quantifiers.

Our handling of nested contexts relies on substitutions: a context is a pair of a left hand side of a sequent, and a graph condition with a special variable. The rule for manipulating the context is usable both ways:

$$\frac{\vdash Ctx(x)}{Ctx \vdash x} \qquad (\textsc{Ctx})$$

5 Related Work

Recently, Poskitt and Plump [16] have presented a weakest precondition calculus for a different extension of nested conditions (monadic second-order conditions) and demonstrated how to use it 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 [17] are another approach towards the same goal; they have already been mentioned in the main text and recently there has been an effort at extending the weakest precondition calculus to a subclass including path expressions. Verification of graph transformation system has also been performed within general-purpose theorem proving environments by Strecker et al. [19, 13], with positive path conditions. Verification of graph transformation systems via model checking of abstractions, as opposed to the prover-based approach pursued here, can be found in Gadducci et al., Baldan et al., König et al., Rensink et al. [4, 1, 10, 18].

A summary overview of graph conditions for non-local properties is attempted below (a proof calculus is presented in [16] but completeness of a proof calculus has only recently been obtained by Lambers and Orejas [11] 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 [17] 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	[12]	(here)	[17]	[16]
conditions	Nested	μ-	HR*	MSO-
wlp	yes	yes	incomplete[6]	yes
complete proof calculus	yes	future work		
theorem prover	yes	future work		

[6] Radke, personal communication.

6 Conclusion and Outlook

We have introduced μ-conditions and achieved several results, mainly a weakest liberal precondition transformation (Theorem 2), soundness of a proof calculus (Theorem 3), and discussed correctness relative to μ-conditions, which appears to be a fruitful ground for further investigations.

In analogy to the equivalence between first-order predicate graph logic and nested graph conditions, we are investigating whether μ-conditions have the same expressivity as fixed point extensions to classical first-order logic for finite graphs.

Also, the expressivity of HR* conditions [17] or even MSO likely surpasses that of μ-conditions, but the precise relationship remains to be examined. As the examples show, the weakest precondition calculus (which is still a research question for HR* conditions [17] but readily available by logical means in the MSO-conditions formalism [16]) produces quite unwieldy expressions due to partial shift. The blowup is exponential in the size of the interface graphs used in the rule, and seems unavoidable because of the need to use a fixed set of type graphs for the finitely many variables (and a blowup is also inherited from the weakest precondition calculus of [12]). Rule IndMuEmpty also contributes because it involves a Cartesian product between variable sets. We have devised heuristics to simplify the expressions, but even if many of the cases can be resolved automatically, this issue still raises concerns as to the practical applicability.

Future work will also include tool support with special attention to semi-automated reasoning, based on the reasoning engine Enforce implemented in [12]. 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, one could proceed as in [14] and prove a termination variant. We plan a further generalisation to correctness under adverse conditions, i.e. systems subject to environmental interference, also modelled as a graph program. Furthermore, 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 by introducing a temporal *next* operator parameterised on atomic subprograms (the basic steps of Def. 4) 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. Eventually, we would also like to deal with algebraic operations on attributes and extend 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. Finally, the limitations imposed by undecidability prompt the search for of restricted decidable classes.

Acknowledgements

We thank Annegret Habel, many other members of SCARE and the anonymous reviewers for constructive criticism of the approach and the paper.

References

1. Baldan, P., König, B., König, B.: A logic for analyzing abstractions of graph transformation systems. In: Static Analysis, pp. 255–272. Springer (2003)
2. Dijkstra, E.W.: A discipline of programming. Prentice Hall (1976)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
4. Gadducci, F., Heckel, R., Koch, M.: A fully abstract model for graph-interpreted temporal logic. In: TAGT'98. LNCS, vol. 1764, pp. 310–322 (1998)
5. Gentzen, G.: Untersuchungen über das logische Schließen. I. Mathematische Zeitschrift 39(1), 176–210 (1935)
6. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. in Comp. Sci. 19(2), 245–296 (2009)
7. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: ICGT 2006. LNCS, vol. 4178, pp. 445–460 (2006)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 26(1), 53–56 (1983)
9. Kreutzer, S.: Pure and Applied Fixed-Point Logics. Ph.D. thesis, Dissertation thesis, RWTH Aachen (2002)
10. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. LNCS, vol. 3920, pp. 197–211 (2006)
11. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: Graph Transformation, LNCS, vol. 8571, pp. 17–32 (2014)
12. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Universität Oldenburg (2009)
13. Percebois, C., Strecker, M., Tran, H.N.: Rule-level verification of graph transformations for invariants based on edges' transitive closure. In: SEFM 2013. LNCS, vol. 8137, pp. 106–121 (2013)
14. Poskitt, C.M.: Verification of Graph Programs. Ph.D. thesis, University of York (2013)
15. Poskitt, C.M., Plump, D.: Verifying total correctness of graph programs. Electronic Communications of the EASST 61 (2013)
16. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: Graph Transformation, LNCS, vol. 8571, pp. 33–48 (2014)
17. Radke, H.: HR* graph conditions between counting monadic second-order and second-order graph formulas. Electronic Communications of the EASST 61 (2013)
18. Rensink, A., Distefano, D.: Abstract graph transformation. ENTCS 157(1), 39–59 (2006)
19. Strecker, M.: Modeling and verifying graph transformations in proof assistants. ENTCS 203(1), 135–148 (2008)
20. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5(2), 285–309 (1955)