Verifying Monadic Second-Order Properties of Graph Programs

Christopher M. Poskitt[1] and Detlef Plump[2]

[1] Department of Computer Science, ETH Zürich, Switzerland
[2] Department of Computer Science, The University of York, UK

Abstract. The core challenge in a Hoare- or Dijkstra-style proof system for graph programs is in defining a weakest liberal precondition construction with respect to a rule and a postcondition. Previous work addressing this has focused on assertion languages for first-order properties, which are unable to express important global properties of graphs such as acyclicity, connectedness, or existence of paths. In this paper, we extend the nested graph conditions of Habel, Pennemann, and Rensink to make them equivalently expressive to monadic second-order logic on graphs. We present a weakest liberal precondition construction for these assertions, and demonstrate its use in verifying non-local correctness specifications of graph programs in the sense of Habel et al.

1 Introduction

Many problems in computer science and software engineering can be modelled in terms of graphs and graph transformation, including the specification and analysis of pointer structures, object-oriented systems, and model transformations; to name just a few. These applications, amongst others, motivate the development of techniques for verifying the functional correctness of both graph transformation rules and programs constructed over them.

A recent strand of research along these lines has resulted in the development of *proof calculi* for graph programs. These, in general, provide a means of systematically proving that a program is correct relative to a specification. A first approach was considered by Habel, Pennemann, and Rensink [8, 14], who contributed weakest precondition calculi – in the style of Dijkstra – for simple rule-based programs, with specifications expressed using *nested conditions* [7]. Subsequently, we developed Hoare logics [18, 17] for the graph transformation language GP 2 [16], which additionally allows computation over labels, and employed as a specification language an extension of nested conditions with support for expressions.

Both approaches suffer from a common drawback, in that they are limited to first-order structural properties. In particular, neither of them support proofs about important *non-local* properties of graphs, e.g. acyclicity, connectedness, or the existence of arbitrary-length paths. Part of the difficulty in supporting such assertions is at the core of both approaches: defining an effective construction for

the weakest property guaranteeing that an application of a given rule will establish a given postcondition (i.e. the construction of a *weakest liberal precondition* for graph transformation rules).

Our paper addresses exactly this challenge. We define an extension of nested conditions that is equivalently expressive to monadic second-order (MSO) logic on graphs [3]. For this assertion language, and for graph programs similar to those of [8, 14], we define a weakest liberal precondition construction that can be integrated into Dijkstra- and Hoare-style proof calculi. Finally we demonstrate its use in verifying non-local correctness specifications (properties including that the graph is bipartite, acyclic) of some simple programs.

The paper is organised as follows. In Section 2 we provide some preliminary definitions and notations. In Section 3 we define an extension of nested conditions for MSO properties. In Section 4 we define graph programs, before presenting our weakest liberal precondition construction in Section 5, and demonstrating in Section 6 its use in Hoare-style correctness proofs. Finally, Section 7 presents some related work before we conclude the paper in Section 8.

Proofs omitted from this paper are available in an extended version [20].

2 Preliminaries

Let $\mathbb{B} = \{\text{true}, \text{false}\}$ denote the set of Boolean values, $\text{Vertex}, \text{Edge}$ denote (disjoint) sets of node and edge identifiers (which shall be written in lowercase typewriter font, e.g. $\mathtt{v}, \mathtt{e}$), and $\text{VSetVar}, \text{ESetVar}$ denote (disjoint) sets of node- and edge-set variables (which shall be written in uppercase typewriter font, e.g. $\mathtt{X}, \mathtt{Y}$).

A *graph* over a label alphabet $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ is defined as a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where $V_G \subset \text{Vertex}$ and $E_G \subset \text{Edge}$ are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G \colon E_G \to V_G$ are the *source* and *target* functions for edges, $l_G \colon V_G \to \mathcal{C}_V$ is the node labelling function and $m_G \colon E_G \to \mathcal{C}_E$ is the edge labelling function. The *empty graph*, denoted by $\emptyset$, has empty node and edge sets. For simplicity, we fix the label alphabet throughout this paper as $\mathcal{L} = \langle \{\square\}, \{\square\} \rangle$, where $\square$ denotes the blank label (which we render as $\bullet$ and $\longrightarrow$ in pictures). We note that our technical results hold for any fixed finite label alphabet.

Given a graph G, the *(directed) path predicate* $\text{path}_G \colon V_G \times V_G \times 2^{E_G} \to \mathbb{B}$ is defined inductively for nodes $v, w \in V_G$ and sets of edges $E \subseteq E_G$. If $v = w$, then $\text{path}_G(v, w, E)$ holds. If $v \neq w$, then $\text{path}_G(v, w, E)$ holds if there exists an edge $e \in E_G \setminus E$ such that $s_G(e) = v$ and $\text{path}_G(t_G(e), w, E)$.

A *graph morphism* $g \colon G \to H$ between graphs G, H consists of two functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources, targets and labels; that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $l_H \circ g_V = l_G$, and $m_H \circ g_E = m_G$. We call G, H the *domain* (resp. *codomain*) of g. Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x. It is *injective* (*surjective*) if g_V and g_E are injective (surjective). It is an *isomorphism* if it is both injective and surjective. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

3 Expressing Monadic Second-Order Properties

We extend the nested conditions of [7] to a formalism equivalently expressive to MSO logic on graphs. The idea is to introduce new quantifiers for node- and edge-set variables, and equip morphisms with constraints about set membership. The definition of satisfaction is then extended to require an interpretation of these variables in the graph such that the constraint evaluates to true. Furthermore, constraints can also make use of a predicate for explicitly expressing properties about directed paths. Such properties can of course be expressed in terms of MSO expressions, but the predicate is provided as a more compact alternative.

Definition 1 (Interpretation; interpretation constraint). Given a graph G, an *interpretation I in G* is a partial function $I : \mathrm{VSetVar} \cup \mathrm{ESetVar} \to 2^{V_G} \cup 2^{E_G}$, such that for all variables $\mathtt{X}$ on which it is defined, $I(\mathtt{X}) \in 2^{V_G}$ if $\mathtt{X} \in \mathrm{VSetVar}$ (resp. 2^{E_G}, ESetVar). An *(interpretation) constraint* is a Boolean expression that can be derived from the syntactic category Constraint of the following grammar:

Constraint ::= Vertex '∈' VSetVar | Edge '∈' ESetVar
| `path` '`(`' Vertex '`,`' Vertex ['`,`' `not` Edge {'`|`' Edge}] '`)`'
| `not` Constraint | Constraint (`and` | `or`) Constraint | `true`

Given a constraint γ, an interpretation I in G, and a morphism q with codomain G, the value of $\gamma^{I,q}$ in $\mathbb{B}$ is defined inductively. If γ contains a set variable for which I is undefined, then $\gamma^{I,q} = \text{false}$. Otherwise, if γ is `true`, then $\gamma^{I,q} = \text{true}$. If γ has the form $x \in \mathtt{X}$ with x a node or edge identifier and $\mathtt{X}$ a set variable, then $\gamma^{I,q} = \text{true}$ if $q(x) \in I(\mathtt{X})$. If γ has the form `path(`v`,`w`)` with v, w node identifiers, then $\gamma^{I,q} = \text{true}$ if the predicate $\mathrm{path}_G(q(v), q(w), \emptyset)$ holds. If γ has the form `path(`v`,`w`,not` e_1`|`$\ldots$`|`e_n`)` with v, w node identifiers and $e_1, \ldots, e_n$ edge identifiers, then $\gamma^{I,q} = \text{true}$ if it is the case that the path predicate $\mathrm{path}_G(q(v), q(w), \{q(e_1), \ldots, q(e_n)\})$ holds. If γ has the form `not` γ_1 with γ_1 a constraint, then $\gamma^{I,q} = \text{true}$ if $\gamma_1^{I,q} = \text{false}$. If γ has the form γ_1 `and` γ_2 (resp. γ_1 `or` γ_2) with γ_1, γ_2 constraints, then $\gamma^{I,q} = \text{true}$ if both (resp. at least one of) $\gamma_1^{I,q}$ and $\gamma_2^{I,q}$ evaluate(s) to true. □

Definition 2 (M-condition; M-constraint). An *MSO condition* (short. *M-condition*) over a graph P is of the form `true`, $\exists_{\mathtt{V}}\mathtt{X}[c]$, $\exists_{\mathtt{E}}\mathtt{X}[c]$, or $\exists(a \mid \gamma, c')$, where $\mathtt{X} \in \mathrm{VSetVar}$ (resp. ESetVar), c is an M-condition over P, $a\colon P \hookrightarrow C$ is an injective morphism (since we consider programs with injective matching), γ is an interpretation constraint over items in C, and c' is an M-condition over C. Furthermore, Boolean formulae over M-conditions over P are also M-conditions over P; that is, $\neg c$, $c_1 \wedge c_2$, and $c_1 \vee c_2$ are M-conditions over P if c, c_1, c_2 are M-conditions over P.

An M-condition over the empty graph $\emptyset$ in which all set variables are bound to quantifiers is called an *M-constraint*. □

For brevity, we write `false` for $\neg$`true`, $c \Rightarrow d$ for $\neg c \vee d$, $c \Leftrightarrow d$ for $c \Rightarrow d \wedge d \Rightarrow c$, $\forall_{\mathtt{V}}\mathtt{X}[c]$ for $\neg\exists_{\mathtt{V}}\mathtt{X}[\neg c]$, $\forall_{\mathtt{E}}\mathtt{X}[c]$ for $\neg\exists_{\mathtt{E}}\mathtt{X}[\neg c]$, $\exists_{\mathtt{V}}\mathtt{X}_1, \ldots \mathtt{X}_n[c]$ for

$\exists_{\mathtt{V}}\mathtt{X}_1\,[\,\ldots\exists_{\mathtt{V}}\mathtt{X}_n\,[c]\,\ldots\,]$ (analogous for other set quantifiers), $\exists(a \mid \gamma)$ for $\exists(a \mid \gamma, \mathtt{true})$, $\exists(a, c')$ for $\exists(a \mid \mathtt{true}, c')$, and $\forall(a \mid \gamma, c')$ for $\neg\exists(a \mid \gamma, \neg c')$.

In our examples, when the domain of a morphism $a\colon P \hookrightarrow C$ can unambiguously be inferred, we write only the codomain C. For instance, an M-constraint $\exists(\emptyset \hookrightarrow C, \exists(C \hookrightarrow C'))$ can be written as $\exists(C, \exists(C'))$.

Definition 3 (Satisfaction of M-conditions). Let $p : P \hookrightarrow G$ denote an injective morphism, c an M-condition over P, and I an interpretation in G. We define inductively the meaning of $p \models^I c$, which denotes that p *satisfies* c *with respect to* I. If c has the form $\mathtt{true}$, then $p \models^I c$. If c has the form $\exists_{\mathtt{V}}\mathtt{X}[c']$ (resp. $\exists_{\mathtt{E}}\mathtt{X}[c']$), then $p \models^I c$ if $p \models^{I'} c'$, where $I' = I \cup \{\mathtt{X} \mapsto V\}$ for some $V \subseteq V_G$ (resp. $\{\mathtt{X} \mapsto E\}$ for some $E \subseteq E_G$). If c has the form $\exists(a\colon P \hookrightarrow C \mid \gamma, c')$, then $p \models^I c$ if there is an injective morphism $q\colon C \hookrightarrow G$ such that $q \circ a = p$, $\gamma^{I,q} = \text{true}$, and $q \models^I c'$.

A graph G *satisfies* an M-constraint c, denoted $G \models c$, if $i_G\colon \emptyset \hookrightarrow G \models^{I_\emptyset} c$, where $I_\emptyset$ is the *empty interpretation in* G, i.e. undefined on all set variables. □

We remark that model checking for both first-order and monadic second-order logic is known to be PSPACE-complete [5]. However, the model checking problem for monadic second-order logic on graphs of bounded treewidth can be solved in linear time [2].

Example 1. The following M-constraint *col* (translated from the corresponding formula §1.5 of [1]) expresses that a graph is 2-colourable (or bipartite); i.e. every node can be assigned one of two colours such that no two adjacent nodes have the same one. Let $\gamma_{\mathtt{col}}$ denote `not (v∈X and w∈X) and not (v∈Y and w∈Y)`.

$$\exists_{\mathtt{V}}\mathtt{X,Y}\ [\ \forall(\ \bullet_{\mathtt{v}}\ ,\ \exists(\ \bullet_{\mathtt{v}}\ \mid \mathtt{(v{\in}X\ or\ v{\in}Y)\ and\ not\ (v{\in}X\ and\ v{\in}Y)})) \\ \wedge\ \forall(\ \bullet_{\mathtt{v}}\ \ \bullet_{\mathtt{w}}\ ,\exists(\ \bullet_{\mathtt{v}}\!\!\rightarrow\!\!\bullet_{\mathtt{w}}\ )\Rightarrow\exists(\ \bullet_{\mathtt{v}}\ \ \bullet_{\mathtt{w}}\ \mid \gamma_{\mathtt{col}}))\ ]$$

A graph G will satisfy *col* if there exist two subsets of V_G such that: (1) every node in G belongs to *exactly one* of the two sets; and (2) if there is an edge from one node to another, then those nodes are not in the same set. Intuitively, one can think of the sets `X` and `Y` as respectively denoting the nodes of colour one and colour two. If two such sets do not exist, then the graph cannot be assigned a 2-colouring. □

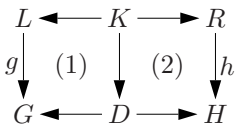
Theorem 1 (M-constraints are equivalent to MSO formulae). The assertion languages of M-constraints and MSO graph formulae are equivalently expressive: that is, given an M-constraint c, there exists an MSO graph formula φ such that for all graphs G, $G \models c$ if and only if $G \models \varphi$; and vice versa. □

4 Graph Programs

In this section we define rules, rule application, and graph programs. Whilst the syntax and semantics of the control constructs are based on those of GP 2 [16],

the rules themselves follow [8, 14], i.e. are labelled over a fixed finite alphabet, and do not support relabelling or expressions. We equip the rules with application conditions (M-conditions over the left- and right-hand graphs), and define *rule application* via the standard double-pushout construction [4].

Definition 4 (Rule; direct derivation). A *plain rule* $r' = \langle L \hookleftarrow K \hookrightarrow R \rangle$ comprises two inclusions $K \hookrightarrow L$, $K \hookrightarrow R$. We call L, R the left- (resp. right-) hand graph and K the interface. An *application condition* $\mathrm{ac} = \langle \mathrm{ac}_L, \mathrm{ac}_R \rangle$ for r' consists of two M-conditions over L and R respectively. A *rule* $r = \langle r', \mathrm{ac} \rangle$ is a plain rule r' and an application condition ac for r'.



For a plain rule r' and a morphism $K \hookrightarrow D$, a *direct derivation* $G \Rightarrow_{r',g,h} H$ (short. $G \Rightarrow_{r'} H$ or $G \Rightarrow H$) is given by the pushouts (1) and (2). For a rule $r = \langle r', \mathrm{ac} \rangle$, there is a *direct derivation* $G \Rightarrow_{r,g,h} H$ if $G \Rightarrow_{r',g,h} H$, $g \models^{I_\emptyset} \mathrm{ac}_L$, and $h \models^{I_\emptyset} \mathrm{ac}_R$. We call g, h a *match* (resp. *comatch*) for r. Given a set of rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_{r,g,h} H$ for some $r \in \mathcal{R}$. □

It is known that, given a (plain) rule r, graph G, and morphism g as above, there exists a direct derivation if and only if g satisfies the *dangling condition*, i.e. that no node in $g(L) \setminus g(K)$ is incident to an edge in $G \setminus g(L)$. In this case, D and H are determined uniquely up to isomorphism, constructed from G as follows: first, remove all edges in $g(L) \setminus g(K)$ obtaining D. Then add disjointly all nodes and edges from $R \setminus K$ retaining their labels. For $e \in E_R \setminus E_K$, $s_H(e) = s_R(e)$ if $s_R(e) \in V_R \setminus V_K$, otherwise $s_H(e) = g_V(s_R(e))$, (targets defined analogously) resulting in the graph H.

We will often give rules without the interface, writing just $L \Rightarrow R$. In such cases we number nodes that correspond in L and R, and establish the convention that K comprises exactly these nodes and that $E_K = \emptyset$ (i.e. K can be completely inferred from L, R). Furthermore, if the application condition of a rule is $\langle \mathtt{true}, \mathtt{true} \rangle$, then we will only write the plain rule component.

We consider now the syntax and semantics of graph programs, which provide a mechanism to control the application of rules to some graph provided as input.

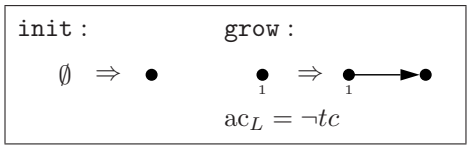
Definition 5 (Graph program). *(Graph) programs* are defined inductively. First, every rule (resp. rule set) $r, \mathcal{R}$ and `skip` are programs. Given programs C, P, Q, we have that $P; Q$, $P!$, `if` C `then` P `else` Q, and `try` C `then` P `else` Q are programs. □

Graph programs are *nondeterministic*, and their execution on a particular graph could result in one of several possible outcomes. That outcome could be a graph, or it could be the special state "fail" which occurs when a rule (set) is not *applicable* to the current graph.

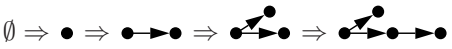
A full structural operational semantics is given in [20], but the informal meaning of the constructs is as follows. Let G denote an input graph. Programs

$r, \mathcal{R}$ correspond to rule (resp. rule set) application, returning H if there exists some $G \Rightarrow_r H$ (resp. $G \Rightarrow_{\mathcal{R}} H$); otherwise fail. Program $P; Q$ denotes sequential composition. Program $P!$ denotes as-long-as-possible iteration of P. Finally, the conditional programs execute the first or second branch depending on whether executing C returns a graph or fail, with the distinction that the `if` construct does not retain any effects of C, whereas the `try` construct does.

Example 2. Consider the program `init`; `grow`! defined by the rules:



where tc is an (unspecified) M-condition over L expressing some termination condition for the iteration (proving termination is not our concern here, see e.g. [19]). The program, if executed on the empty graph, nondeterministically constructs and returns a tree. It applies the rule `init` exactly once, creating an isolated node. It then iteratively applies the rule `grow` (each application adding a leaf to the tree) until the termination condition tc holds. An example program run, with $tc = \exists(\bullet_1 \bullet \bullet \bullet)$, is:



□

5 Constructing a Weakest Liberal Precondition

In this section, we present a construction for the *weakest liberal precondition* relative to a rule r and a postcondition c (which is an M-constraint). In our terminology, if a graph satisfies a weakest liberal precondition, then: (1) any graphs resulting from applications of r will satisfy c; and (2) there does not exist another M-constraint with this property that is weaker. (Note that we do not address termination or existence of results in this paper.)

The construction is adapted from the one for nested conditions in [7], and as before, is broken down into a number of stages. First, a translation of postconditions into M-conditions over R (transformation "A"); then, from M-conditions over R into M-conditions over L (transformation "L"); and finally, from M-conditions over L into an M-constraint expressing the weakest liberal precondition (via transformations "App" and "Pre").

First, we consider transformation A, which constructs an M-condition over R from a postcondition (an M-constraint) by computing a disjunction over all the ways that the M-constraint and comatches might "overlap".

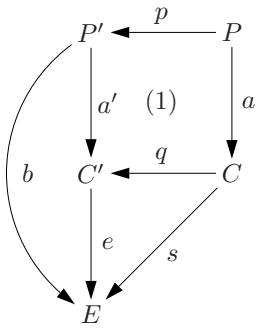
Theorem 2 (M-constraints to M-conditions over R). There is a transformation A, such that for all M-constraints c, all rules r with right-hand side R, and all injective morphisms $h\colon R \hookrightarrow H$,

$$h \models^{I_\emptyset} \mathrm{A}(r, c) \text{ if and only if } H \models c.$$

Construction. Let c denote an M-constraint, and r a rule with right-hand side R. We define $\mathrm{A}(r, c) = \mathrm{A}'(\emptyset \hookrightarrow R, c)$ where A' is defined inductively as follows. For injective graph morphisms $p\colon P \hookrightarrow P'$ and M-conditions over P, define:

$$\begin{aligned}
\mathrm{A}'(p, \texttt{true}) &= \texttt{true},\\
\mathrm{A}'(p, \exists_{\mathtt{V}}\mathtt{X}\texttt{[}c'\texttt{]}) &= \exists_{\mathtt{V}}\mathtt{X}\texttt{[}\mathrm{A}'(p, c')\texttt{]},\\
\mathrm{A}'(p, \exists_{\mathtt{E}}\mathtt{X}\texttt{[}c'\texttt{]}) &= \exists_{\mathtt{E}}\mathtt{X}\texttt{[}\mathrm{A}'(p, c')\texttt{]},\\
\mathrm{A}'(p, \exists(a\colon P \hookrightarrow C \mid \gamma, c')) &= \textstyle\bigvee_{e\in\varepsilon}\exists(b\colon P' \hookrightarrow E \mid \gamma, \mathrm{A}'(s\colon C \hookrightarrow E, c')).
\end{aligned}$$

The final equation relies on the following. First, construct the pushout (1) of p and a leading to injective graph morphisms $a'\colon P' \hookrightarrow C'$ and $q\colon C \hookrightarrow C'$. The disjunction then ranges over the set ε, which we define to contain every surjective graph morphism $e\colon C' \to E$ such that $b = e \circ a'$ and $s = e \circ q$ are injective graph morphisms (we consider the codomains of each e up to isomorphism, hence the disjunction is finite).



The transformations A, A' are extended for Boolean formulae over M-conditions in the usual way, that is, $\mathrm{A}(r, \neg c) = \neg\mathrm{A}(r, c)$, $\mathrm{A}(r, c_1 \wedge c_2) = \mathrm{A}(r, c_1) \wedge \mathrm{A}(r, c_2)$, and $\mathrm{A}(r, c_1 \vee c_2) = \mathrm{A}(r, c_1) \vee \mathrm{A}(r, c_2)$ (analogous for A'). □

Example 3. Recall the rule `grow` from Example 2. Let c denote the M-constraint:

$$\exists_{\mathtt{V}}\mathtt{X,Y}\texttt{[}\ \forall(\ \bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ , \exists(\ \bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ \mid \mathtt{path(v,w)}) \Rightarrow \exists(\ \bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ \mid \gamma))\ \texttt{]}$$

for $\gamma =$ `(v ∈ X and w ∈ Y) and not (v ∈ Y or w ∈ X)`, which expresses that there are two sets of nodes X, Y in the graph, such that if there is a path from some node v to some node w, then v belongs only to X and w only to Y. Applying transformation A:

$$\begin{aligned}
&\mathrm{A}(\mathtt{grow}, c)\\
&= \mathrm{A}'(\emptyset \hookrightarrow\ \bullet_{\mathtt{1}}\!\!\rightarrow\!\!\bullet_{\mathtt{2}}\ , c)\\
&= \exists_{\mathtt{V}}\mathtt{X,Y}\texttt{[}\ \mathrm{A}'(\emptyset \hookrightarrow\ \bullet_{\mathtt{1}}\!\!\rightarrow\!\!\bullet_{\mathtt{2}}\ , \forall(\ \bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ ,\\
&\qquad \exists(\ \bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ \mid \mathtt{path(v,w)}) \Rightarrow \exists(\ \bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ \mid \gamma)))\ \texttt{]}\\
&= \exists_{\mathtt{V}}\mathtt{X,Y}\texttt{[}\ \textstyle\bigwedge_{i=1}^{7}\forall(\ \bullet_{\mathtt{1}}\!\!\rightarrow\!\!\bullet_{\mathtt{2}}\ \hookrightarrow E_i, \exists(E_i \mid \mathtt{path(v,w)}) \Rightarrow \exists(E_i \mid \gamma))\ \texttt{]}
\end{aligned}$$

where the graphs E_i are as given in Figure 1. □

Transformation L, adapted from [7], takes an M-condition over R and constructs an M-condition over L that is satisfied by a match if and only if the

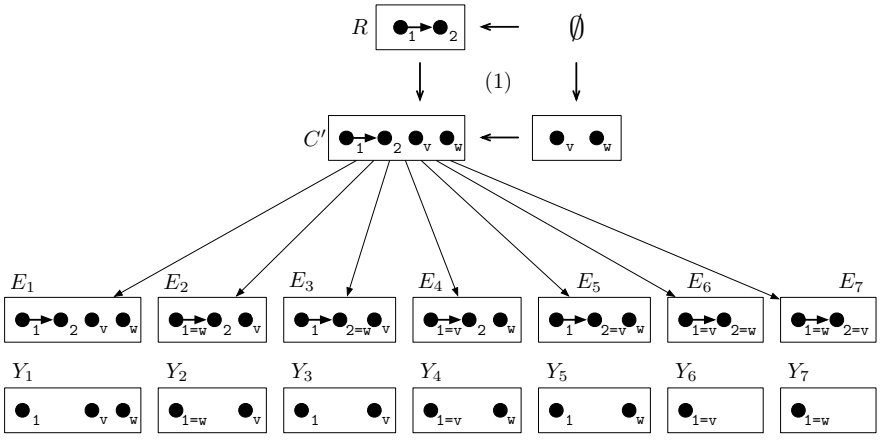


Fig. 1. Applying the construction in Examples 3 and 4

original is satisfied by the comatch. The transformation is made more complex by the presence of path and MSO expressions, because nodes and edges referred to on the right-hand side may no longer exist on the left. For clarity, we separate the handling of these two types of expressions, and in particular, define a *decomposition* LPath of path predicates according to the items that the rule is creating or deleting. For example, if an edge is created by a rule, a path predicate decomposes to a disjunction of path predicates collectively asserting the existence of paths to and from the nodes that will eventually become its source and target; whereas if an edge is to be deleted, the predicate will exclude it.

Proposition 1 (Path decomposition). There is a transformation LPath such that for every rule $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$, direct derivation $G \Rightarrow_{r,g,h} H$, path predicate p over R, and interpretation I,

$$\mathrm{LPath}(r, p)^{I,g} = p^{I,h}.$$

Construction. Let $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ and $p = \texttt{path}(v, w, \texttt{not } E)$. For simplicity, we will treat the syntactic construct E as a set of edges and identify $\texttt{path}(v, w, \texttt{not } E)$ and $\texttt{path}(v, w)$ when E is empty. Then, define:

$$\mathrm{LPath}(r, p) = \mathrm{LPath}'(r, v, w, E^{\ominus}) \texttt{ or } \mathrm{FuturePaths}(r, p).$$

Here, $E^{\ominus}$ is constructed from E by adding edges $e \in E_L \setminus E_R$, i.e. that the rule will delete. Furthermore, $\mathrm{LPath}'(r, v, w, E^{\ominus})$ decomposes to path predicates according to whether v and w exist in K. If $\mathrm{path}_R(v, w, E^{\ominus})$ holds, then $\mathrm{LPath}'(r, v, w, E^{\ominus})$ returns `true`. Otherwise, if both $v, w \in V_K$, then it returns $\texttt{path}(v, w, \texttt{not } E^{\ominus})$. If $v \notin V_K, w \in V_K$, it returns:

$$\texttt{false or path}(x_1, w, \texttt{not } E^{\ominus}) \texttt{ or path}(x_2, w, \texttt{not } E^{\ominus}) \texttt{ or} \ldots$$

for each $x_i \in V_K$ such that $\mathrm{path}_R(v, x_i, E^{\ominus})$. Case $v \in V_K, w \notin V_K$ analogous. If $v, w \notin V_K$, then it returns $\texttt{false or path}(x_i, y_j, \texttt{not } E^{\ominus})\ \texttt{or}\ \ldots$ for all $x_i, y_j \in V_K$ such that $\mathrm{path}_R(v, x_i, E^{\ominus})$ and $\mathrm{path}_R(y_j, w, E^{\ominus})$.

Finally, FuturePaths(r, p) denotes `false` in disjunction with:

$$(\mathrm{LPath}'(r, v, x_1, E^{\ominus})\ \texttt{and path}(y_1, x_2, \texttt{not } E^{\ominus}) \ldots \texttt{and path}(y_i, x_{i+1}, \texttt{not } E^{\ominus}) \ldots \texttt{and}\ \mathrm{LPath}'(r, y_n, w, E^{\ominus}))$$

over all non-empty sequences of distinct pairs $\langle\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle\rangle$ drawn from:

$$\{\langle x, y\rangle \mid x, y \in V_K \wedge \mathrm{path}_R(x, y, E^{\ominus}) \wedge \neg \mathrm{path}_L(x, y, E^{\ominus})\}.$$

□

In addition to paths, transformation L must handle MSO expressions that refer to items present in R but absent in L. To achieve this, it computes a disjunction over all possible "future" (i.e. immediately after the rule application) set memberships of these missing items. The idea being, that if a set membership exists for these missing items that satisfies the interpretation constraints *before* the rule application, then one will still exist once they have been created. The transformation keeps track of such potential memberships via sets of pairs as follows.

Definition 6 (Membership set). A *membership set* M is a set of pairs $(x, \mathtt{X})$ of node or edge identifiers x with set variables of the corresponding type. Intuitively, $(x, \mathtt{X}) \in M$ encodes that $x \in \mathtt{X}$, whereas $(x, \mathtt{X}) \notin M$ encodes that $x \notin \mathtt{X}$.

□

Theorem 3 (From M-conditions over R to L). There is a transformation L such that for every rule $r = \langle\langle L \hookleftarrow K \hookrightarrow R\rangle, \mathrm{ac}\rangle$, every M-condition c over R (with no free variables, and distinct variables for distinct quantifiers), and every direct derivation $G \Rightarrow_{r,g,h} H$,

$$g \models^{I_\emptyset} \mathrm{L}(r, c) \ \text{ if and only if } \ h \models^{I_\emptyset} c.$$

Construction. Let $r = \langle\langle L \hookleftarrow K \hookrightarrow R\rangle, \mathrm{ac}\rangle$ denote a rule and c an M-condition over R. We define $\mathrm{L}(r, c) = \mathrm{L}'(r, c, \emptyset)$. For such an r, c, and membership set M, the transformation L' is defined inductively as follows:

$$\begin{aligned}
\mathrm{L}'(r, \texttt{true}, M) &= \texttt{true}, \\
\mathrm{L}'(r, \exists_{\mathtt{V}}\mathtt{X}[c'], M) &= \exists_{\mathtt{V}}\mathtt{X}[\ \bigvee_{M' \in 2^{M_{\mathtt{V}}}} \mathrm{L}'(r, c', M \cup M')\ ] \\
\mathrm{L}'(r, \exists_{\mathtt{E}}\mathtt{X}[c'], M) &= \exists_{\mathtt{E}}\mathtt{X}[\ \bigvee_{M' \in 2^{M_{\mathtt{E}}}} \mathrm{L}'(r, c', M \cup M')\ ]
\end{aligned}$$

where $M_{\mathtt{V}} = \{(v, \mathtt{X}) \mid v \in V_R \setminus V_L\}$ and $M_{\mathtt{E}} = \{(e, \mathtt{X}) \mid e \in E_R \setminus E_L\}$.

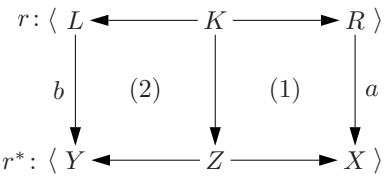
For case $c = \exists(a \mid \gamma, c')$, we define:

$$\mathrm{L}'(r, \exists(a \mid \gamma, c'), M) = \texttt{false}$$

if $\langle K \hookrightarrow R, a\rangle$ has no pushout complement; otherwise:

$$\mathrm{L}'(r, \exists(a \mid \gamma, c'), M) = \exists(b \mid \gamma_M, \mathrm{L}'(r^*, c', M))$$

which relies on the following. First, construct the pushout (1), with $r^* = \langle Y \hookleftarrow Z \hookrightarrow X\rangle$ the "derived" rule obtained by constructing pushout (2). The interpretation constraint γ_M is obtained from γ as follows. First, consider each predicate $x \in \mathtt{X}$ such that $x \notin Y$. If $(y, \mathtt{X}) \in M$ for some $y = x$, replace the predicate with `true`; otherwise `false`. Then, replace each path predicate p with $\mathrm{LPath}(r^*, p)$.



The transformation L is extended for Boolean formulae in the usual way, that is, $\mathrm{L}(r, \neg c) = \neg\mathrm{L}(r, c)$, $\mathrm{L}(r, c_1 \wedge c_2) = \mathrm{L}(r, c_1) \wedge \mathrm{L}(r, c_2)$, and $\mathrm{L}(r, c_1 \vee c_2) = \mathrm{L}(r, c_1) \vee \mathrm{L}(r, c_2)$ (analogous for L'). □

Example 4. Take `grow`, c, γ and $\mathrm{A}(\mathtt{grow}, c)$ as considered in Example 3. Applying transformation L:

$$\begin{aligned}
&\mathrm{L}(\mathtt{grow}, \mathrm{A}(\mathtt{grow}, c)) = \mathrm{L}'(\mathtt{grow}, \mathrm{A}(\mathtt{grow}, c), \emptyset)\\
&= \exists_{\mathtt{V}}\mathtt{X},\mathtt{Y}[\ \textstyle\bigvee_{M' \in 2^{M_{\mathtt{V}}}} \mathrm{L}'(\mathtt{grow}, \bigwedge_{i=1}^{7} \forall(\ \bullet_{\mathtt{1}} \rightarrow \bullet_{\mathtt{2}}\ \hookrightarrow E_i, \exists(E_i \mid \mathtt{path(v,w)})\\
&\qquad \Rightarrow \exists(E_i \mid \gamma)), M')\ ]\\
&= \exists_{\mathtt{V}}\mathtt{X},\mathtt{Y}[\ \textstyle\bigvee_{M' \in 2^{M_{\mathtt{V}}}} (\bigwedge_{i \in \{1,2,4\}} \forall(\ \bullet_{\mathtt{1}}\ \hookrightarrow Y_i, \exists(Y_i \mid \mathtt{path(v,w)}) \Rightarrow \exists(Y_i \mid \gamma))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}, \exists(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}\ \mid \mathtt{path(v,1)}) \Rightarrow \exists(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}\ \mid \gamma_{M'}, \mathrm{L}'(\mathtt{grow}, \mathtt{true}, M')))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{w}}, \mathtt{false} \Rightarrow \exists(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{w}}\ \mid \gamma_{M'}, \mathrm{L}'(\mathtt{grow}, \mathtt{true}, M')))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1=v}}, \mathtt{true} \Rightarrow \exists(\ \bullet_{\mathtt{1=v}}\ \mid \gamma_{M'}, \mathrm{L}'(\mathtt{grow}, \mathtt{true}, M')))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1=w}}, \mathtt{false} \Rightarrow \exists(\ \bullet_{\mathtt{1=w}}\ \mid \gamma_{M'}, \mathrm{L}'(\mathtt{grow}, \mathtt{true}, M'))))\ ]\\
&= \exists_{\mathtt{V}}\mathtt{X},\mathtt{Y}[\ \textstyle\bigvee_{M' \in 2^{M_{\mathtt{V}}}} (\bigwedge_{i \in \{1,2,4\}} \forall(\ \bullet_{\mathtt{1}}\ \hookrightarrow Y_i, \exists(Y_i \mid \mathtt{path(v,w)}) \Rightarrow \exists(Y_i \mid \gamma))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}, \exists(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}\ \mid \mathtt{path(v,1)}) \Rightarrow \exists(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}\ \mid \gamma_{M'}))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1=v}}, \exists(\ \bullet_{\mathtt{1=v}}\ \mid \gamma_{M'})))\ ]\\
&= \exists_{\mathtt{V}}\mathtt{X},\mathtt{Y}[\ \textstyle\bigwedge_{i \in \{1,2,4\}} \forall(\ \bullet_{\mathtt{1}}\ \hookrightarrow Y_i, \exists(Y_i \mid \mathtt{path(v,w)}) \Rightarrow \exists(Y_i \mid \gamma))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}, \exists(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}\ \mid \mathtt{path(v,1)}) \Rightarrow \exists(\ \bullet_{\mathtt{1}}\bullet_{\mathtt{v}}\ \mid \mathtt{v} \in \mathtt{X\ and\ not\ v} \in \mathtt{Y}))\\
&\quad \wedge \forall(\ \bullet_{\mathtt{1=v}}, \exists(\ \bullet_{\mathtt{1=v}}\ \mid \mathtt{v} \in \mathtt{X\ and\ not\ v} \in \mathtt{Y}))\ ]
\end{aligned}$$

where the graphs E_i and Y_i are as given in Figure 1 and $M_{\mathtt{V}} = \{(2, \mathtt{X}), (2, \mathtt{Y})\}$. Here, only one of the subsets ranged over yields a satisfiable disjunct: $M' = \{(2, \mathtt{Y})\}$, i.e. $\gamma_{M'} =$ `(v ∈ X and true) and not (v ∈ Y or false)` for $\mathtt{w} = 2$. □

Transformation App, adapted from Def in [14], takes as input a rule set $\mathcal{R}$ and generates an M-constraint that is satisfied by graphs for which $\mathcal{R}$ is applicable.

Theorem 4 (Applicability of a rule). There is a transformation App such that for every rule set $\mathcal{R}$ and every graph G,

$$G \models \mathrm{App}(\mathcal{R}) \text{ if and only if } \exists H.\, G \Rightarrow_{\mathcal{R}} H.$$

Construction. If $\mathcal{R}$ is empty, define $\mathrm{App}(\mathcal{R}) = \texttt{false}$; otherwise, for $\mathcal{R} = \{r_1, \ldots, r_n\}$, define:

$$\mathrm{App}(\mathcal{R}) = \mathrm{app}(r_1) \vee \cdots \vee \mathrm{app}(r_n).$$

For each rule $r = \langle r', \mathrm{ac}\rangle$ with $r' = \langle L \hookleftarrow K \hookrightarrow R\rangle$, we define $\mathrm{app}(r) = \exists(\emptyset \hookrightarrow L, \mathrm{Dang}(r') \wedge \mathrm{ac}_L \wedge \mathrm{L}(r, \mathrm{ac}_R))$. Here, $\mathrm{Dang}(r') = \bigwedge_{a \in A} \neg\exists a$, where the index set A ranges over all injective graph morphisms $a \colon L \hookrightarrow L^{\oplus}$ (up to isomorphic codomains) such that the pair $\langle K \hookrightarrow L, a\rangle$ has no pushout complement; each $L^{\oplus}$ a graph that can be obtained from L by adding either (1) a loop; (2) a single edge between distinct nodes; or (3) a single node and a non-looping edge incident to that node. □

Finally, transformation Pre (adapted from [8]) combines the other transformations to construct a weakest liberal precondition relative to a rule and postcondition.

Theorem 5 (Postconditions to weakest liberal preconditions). There is a transformation Pre such that for every rule $r = \langle\langle L \hookleftarrow K \hookrightarrow R\rangle, \mathrm{ac}\rangle$, every M-constraint c, and every direct derivation $G \Rightarrow_r H$,

$$G \models \mathrm{Pre}(r, c) \text{ if and only if } H \models c.$$

Moreover, $\mathrm{Pre}(r, c) \vee \neg\mathrm{App}(\{r\})$ is the *weakest liberal precondition* relative to r and c.

Construction. Let $r = \langle\langle L \hookleftarrow K \hookrightarrow R\rangle, \mathrm{ac}\rangle$ denote a rule and c denote an M-constraint. Then:

$$\mathrm{Pre}(r, c) = \forall(\emptyset \hookrightarrow L, (\mathrm{Dang}(r) \wedge \mathrm{ac}_L \wedge \mathrm{L}(r, \mathrm{ac}_R)) \Rightarrow \mathrm{L}(r, \mathrm{A}(r, c))).$$

□

Example 5. Take $\texttt{grow}$, c, γ and $\mathrm{L}(\texttt{grow}, \mathrm{A}(\texttt{grow}, c))$ as considered in Example 4. Applying transformation Pre:

$$\begin{aligned}
&\mathrm{Pre}(\texttt{grow}, \mathrm{L}(\texttt{grow}, \mathrm{A}(\texttt{grow}, c))) \\
&= \forall(\ \bullet_{\texttt{1}}\ , \mathrm{ac}_L \Rightarrow \exists_{\texttt{V}}\texttt{X,Y[}\ \textstyle\bigwedge_{i \in \{1,2,4\}} \forall(\ \bullet_{\texttt{1}}\ \hookrightarrow Y_i, \exists(Y_i \mid \texttt{path(v,w)}) \Rightarrow \exists(Y_i \mid \gamma)) \\
&\quad \wedge \forall(\ \bullet_{\texttt{1}}\bullet_{\texttt{v}}\ , \exists(\ \bullet_{\texttt{1}}\bullet_{\texttt{v}}\ \mid \texttt{path(v,1)}) \Rightarrow \exists(\ \bullet_{\texttt{1}}\bullet_{\texttt{v}}\ \mid \texttt{v} \in \texttt{X and not v} \in \texttt{Y})) \\
&\quad \wedge \forall(\ \bullet_{\texttt{1=v}}\ , \exists(\ \bullet_{\texttt{1=v}}\ \mid \texttt{v} \in \texttt{X and not v} \in \texttt{Y}))\ \texttt{]})
\end{aligned}$$

where the graphs Y_i are as given in Figure 1. This M-constraint is only satisfied by graphs that do not have any edges between distinct nodes, because of the assertion that every match (i.e. every node) must be in $\texttt{X}$ and not in $\texttt{Y}$. Were an edge to exist – i.e. a path – then the M-constraint asserts that its target is in $\texttt{Y}$; a contradiction. □

6 Proving Non-Local Specifications

In this section we show how to systematically prove a non-local correctness specification using a Hoare logic adapted from [18, 17]. The key difference is the use of M-constraints as assertions, and our extension of Pre in constructing weakest liberal preconditions for rules. (We note that one could just as easily adapt the Dijkstra-style systems of [8, 14].)

We will specify the behaviour of programs using *(Hoare) triples*, $\{c\}\ P\ \{d\}$, where P is a program, and c, d are *pre-* and *postconditions* expressed as M-constraints. We say that this specification holds in the sense of *partial correctness*, denoted by $\models \{c\}\ P\ \{d\}$, if for any graph G satisfying c, every graph H resulting from the execution of P on G satisfies d.

For systematically proving a specification, we present a *Hoare logic* in Figure 2, where c, d, e, inv range over M-constraints, P, Q over programs, r over rules, and $\mathcal{R}$ over rule sets. If a triple $\{c\}\ P\ \{d\}$ can be instantiated from an axiom or deduced from an inference rule, then it is *provable* in the Hoare logic and we write $\vdash \{c\}\ P\ \{d\}$. Proofs shall be displayed as trees, with the specification as the root, axiom instances as the leaves, and inference rule instances in-between.

$$[\text{ruleapp}]_{\text{wlp}}\quad \{\text{Pre}(r, c) \vee \neg\text{App}(\{r\})\}\ r\ \{c\} \qquad [\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$[\text{comp}]\ \frac{\{c\}\ P\ \{e\} \qquad \{e\}\ Q\ \{d\}}{\{c\}\ P; Q\ \{d\}} \qquad [!]\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\text{App}(\mathcal{R})\}}$$

$$[\text{cons}]\ \frac{c \Rightarrow c' \quad \{c'\}\ P\ \{d'\} \quad d' \Rightarrow d}{\{c\}\ P\ \{d\}}$$

Fig. 2. A Hoare logic for partial correctness

For simplicity in proofs we will typically treat $[\text{ruleapp}]_{\text{wlp}}$ as two different axioms (one for each disjunct). Note that we have omitted, due to space, the proof rules for the conditional constructs. Note also the restriction to rule sets in [!], because the applicability of arbitrary programs cannot be expressed in a logic for which the model checking problem is decidable [17].

Theorem 6 (Soundness). Given a program P and M-constraints c, d, we have that $\vdash \{c\}\ P\ \{d\}$ implies $\models \{c\}\ P\ \{d\}$. □

The remainder of this section demonstrates the use of our constructions and Hoare logic in proving non-local specifications of two programs. For the first, we will consider a property expressed in terms of MSO variables and expressions, whereas for the second, we will consider properties expressed in terms of `path` predicates. Both programs are simple, as our focus here is not on building intricate proofs but rather on illustrating the main novelty of this paper: a Pre construction for MSO properties.

Example 6. Recall the program `init; grow!` of Example 2 that nondeterministically constructs a tree. A known non-local property of trees is that they can be assigned a 2-colouring (i.e. they are bipartite), a property that the M-constraint *col* of Example 1 precisely expresses. Hence we will show that $\vdash \{emp\}\ \texttt{init; grow!}\ \{col\}$, where $emp = \neg\exists(\ \bullet\ )$ expresses that the graph is empty. A proof tree for this specification is given in Figure 3, where the interpretation constraints γ_1 and γ_2 in $\mathrm{Pre}(\texttt{grow}, col)$ are respectively `(v∈X or v∈Y) and not (v∈X and v∈Y)` and `not (v∈X and w∈X) and not (v∈Y and w∈Y)`.

$$\dfrac{\dfrac{\{\mathrm{Pre}(\texttt{init}, col)\}\ \texttt{init}\ \{col\}}{\{emp\}\ \texttt{init}\ \{col\}} \qquad \dfrac{\dfrac{\{\mathrm{Pre}(\texttt{grow}, col)\}\ \texttt{grow}\ \{col\}}{\{col\}\ \texttt{grow}\ \{col\}}}{\{col\}\ \texttt{grow!}\ \{col \wedge \neg \mathrm{App}(\{\texttt{grow}\})\}}}{\vdash \{emp\}\ \texttt{init;grow!}\ \{col\}}$$

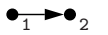
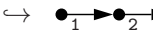
$$\mathrm{Pre}(\texttt{init}, col) \equiv col$$

$$\begin{aligned}\mathrm{Pre}(\texttt{grow}, col) \equiv \forall(\ \bullet_1\ , \neg tc \Rightarrow \exists_{\mathtt{v}}\texttt{X,Y[}\\ &\forall(\ \bullet_1\bullet_{\mathtt{v}}\ , \exists(\ \bullet_1\bullet_{\mathtt{v}}\ \mid \gamma_1)) \wedge\ \forall(\ \bullet_{1=\mathtt{v}}\ , \exists(\ \bullet_{1=\mathtt{v}}\ \mid \gamma_1))\\ &\wedge \forall(\ \bullet_1\bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ , \exists(\ \bullet_1 \xrightarrow[\mathtt{v}]{} \bullet_{\mathtt{w}}\ ) \Rightarrow \exists(\ \bullet_1\bullet_{\mathtt{v}}\bullet_{\mathtt{w}}\ \mid \gamma_2))\\ &\wedge \forall(\ \bullet_{1=\mathtt{v}}\bullet_{\mathtt{w}}\ , \exists(\ \bullet_{1=\mathtt{v}} \xrightarrow[\mathtt{v}]{} \bullet_{\mathtt{w}}\ ) \Rightarrow \exists(\ \bullet_{1=\mathtt{v}}\bullet_{\mathtt{w}}\ \mid \gamma_2))\\ &\wedge \forall(\ \bullet_{1=\mathtt{w}}\bullet_{\mathtt{v}}\ , \exists(\ \bullet_{1=\mathtt{w}} \xrightarrow[\mathtt{w}]{} \bullet_{\mathtt{v}}\ ) \Rightarrow \exists(\ \bullet_{1=\mathtt{w}}\bullet_{\mathtt{v}}\ \mid \gamma_2))\\ &\wedge (\forall(\ \bullet_{1=\mathtt{v}}\ , \exists(\ \bullet_{1=\mathtt{v}}\ \mid \texttt{not v} \in \texttt{X}))\\ &\quad \vee \forall(\ \bullet_{1=\mathtt{v}}\ , \exists(\ \bullet_{1=\mathtt{v}}\ \mid \texttt{not v} \in \texttt{Y})))\ \texttt{]})\end{aligned}$$

Fig. 3. Trees are 2-colourable

Observe that $\mathrm{Pre}(\texttt{grow}, col)$ is essentially an "embedding" of the postcondition *col* within the context of possible matches for `grow`. The second line expresses that every node (whether the node of the match or not) is coloured `X` or `Y`. The following three conjuncts then express that any edges in the various contexts of the match connect nodes that are differently coloured. The final conjunct is of the same form, but is "pre-empting" the creation of a node and edge by `grow`. To ensure that the graph remains 2-colourable, node `1` of the match must not belong to both sets; this, of course, is already established by the first nested conjunct. Hence the first implication arising from instances of [cons], $col \Rightarrow \mathrm{Pre}(\texttt{grow}, col)$, is valid. The second implication, $emp \Rightarrow \mathrm{Pre}(\texttt{init}, col)$, is also valid since a graph satisfying *emp* will not have any nodes to quantify over. □

Example 7. An *acyclic graph* is a graph that does not contain any *cycles*, i.e. non-empty paths starting and ending on the same node. One way to test for acyclicity is to apply the rule $\texttt{delete} = \langle\langle\ \bullet_1 \to \bullet_2\ \Rightarrow\ \bullet_1\bullet_2\ \rangle, \mathrm{ac}_L\rangle$ for as long as possible; the resulting graph being edgeless if the input graph was acyclic. Here, ac_L denotes the left application condition $\neg\exists(\ \bullet_1 \to \bullet_2\ \hookrightarrow\ \bullet \to \bullet_1 \to \bullet_2\ ) \vee$

$\neg\exists($  $\hookrightarrow$ ), expressing that in matches, either the source node has indegree 0 or the target node has outdegree 0 (we do not consider the special case of looping edges for simplicity). Note that nodes *within* a cycle would not satisfy this: if a source node has an indegree of 0 for example, there would be no possibility of an outgoing path ever returning to the same node.

We prove two claims about this rule under iteration: first, that it deletes all edges in an acyclic graph; second, that if applied to a graph containing cycles, the resulting graph would not be edgeless. That is, $\vdash \{\neg c\}$ `delete!` $\{e\}$ and $\vdash \{c\}$ `delete!` $\{\neg e\}$, for M-constraints c (for cycles), e (for edgeless), $\gamma_c =$ `path(v,w,not e) and path(w,v,not e)`, and proofs as in Figure 4.

$$\dfrac{\dfrac{\dfrac{\{\mathrm{Pre}(\texttt{delete},\neg c)\}\ \texttt{delete}\ \{\neg c\}}{\{\neg c\}\ \texttt{delete}\ \{\neg c\}}}{\{\neg c\}\ \texttt{delete!}\ \{\neg c \wedge \neg\mathrm{App}(\{\texttt{delete}\})\}}}{\vdash \{\neg c\}\ \texttt{delete!}\ \{e\}} \qquad \dfrac{\dfrac{\dfrac{\{\mathrm{Pre}(\texttt{delete},c)\}\ \texttt{delete}\ \{c\}}{\{c\}\ \texttt{delete}\ \{c\}}}{\{c\}\ \texttt{delete!}\ \{c \wedge \neg\mathrm{App}(\{\texttt{delete}\})\}}}{\vdash \{c\}\ \texttt{delete!}\ \{\neg e\}}$$

$c = \exists($ $\mid \texttt{path(v,w) and path(w,v)})$

$e = \neg\exists($  $)$

$\mathrm{Pre}(\texttt{delete},\neg c) = \forall($ $, \mathrm{ac}_L \Rightarrow$

$\neg\exists($ $\mid \gamma_c) \wedge \neg\exists($ $\mid \gamma_c)$

$\wedge\neg\exists($ $\mid \gamma_c) \wedge \neg\exists($ $\mid \gamma_c)$

$\wedge\ \neg\exists($ $\mid \gamma_c) \wedge \neg\exists($ $\mid \gamma_c)$

$\wedge\ \neg\exists($  $\mid \gamma_c))$

$\mathrm{App}(\{\texttt{delete}\}) = \exists($ $, \mathrm{ac}_L)$

Fig. 4. Acyclity (or lack thereof) is invariant

First, observe that $\mathrm{Pre}(\texttt{delete},\neg c)$ is essentially an "embedding" of the postcondition $\neg c$ within the context of possible matches for `delete`. The path predicates in γ_c now additionally assert (as a result of the L transformation) that paths do not include images of edge `e`: this is crucially important for establishing the postcondition because the rule deletes the edge. For space reasons we did not specify $\mathrm{Pre}(\texttt{delete}, c)$, but this can be constructed from $\mathrm{Pre}(\texttt{delete},\neg c)$ by replacing each $\wedge$ with $\vee$ and removing each $\neg$ in the nested part.

The instances of [cons] give rise to implications that we must show to be valid. First, $\neg c \Rightarrow \mathrm{Pre}(\texttt{delete},\neg c)$ is valid: a graph satisfying $\neg c$ does not contain any cycles, hence it also does not contain cycles outside of the context of matches for `delete`. Second, $\neg c \wedge \neg\mathrm{App}(\{\texttt{delete}\}) \Rightarrow e$ is valid: a graph satisfying the antecedent does not contain any cycles and also no pair of incident nodes for which ac_L holds. If the graph is not edgeless, then there must be some such pair satisfying ac_L; otherwise the edges are within a cycle. Hence the graph must be edgeless, satisfying e.

In the second proof tree, $c \Rightarrow \mathrm{Pre}(\texttt{delete}, c)$ is valid. A graph satisfying c contains a cycle: clearly, no edge (with its source and target) in this cycle satisfies

ac_L; hence the graph satisfies the consequent, since images of edge `e` cannot be part of the cycle in the graph. Finally, $c \wedge \neg\mathrm{App}(\{\texttt{delete}\}) \Rightarrow \neg e$ is valid: if a graph satisfies the antecedent, then it contains a cycle, the edges of which `delete` will never be applicable to because of ac_L; hence the graph cannot be edgeless, and satisfies $\neg e$. □

7 Related Work

We point to a few related publications addressing the verification of non-local graph properties through proofs / theorem proving and model checking.

Habel and Radke have considered HR conditions [9], an extension of nested conditions embedding hyperedge replacement grammars via graph variables. The formalism is more expressive than MSO logic on graphs (it is able, for example, to express node-counting MSO properties such as "the graph has an even number of nodes" [21]) but it is not yet clear whether an effective construction for weakest liberal preconditions exists. Percebois et al. [15] demonstrate how one can verify global invariants involving paths, directly at the level of rules. Rules are modelled with (a fragment of) first-order logic on graphs in the interactive theorem prover Isabelle. Inaba et al. [10] address the verification of type-annotated Core UnCAL – a query algebra for graph-structured databases – against input/output graph schemas in MSO. They first reformulate the query algebra itself in MSO, before applying an algorithm that reduces the verification problem to the validity of MSO over trees.

The GROOVE model checker [6] supports rules with paths in the left-hand side, expressed as a regular expression over edge labels. One can specify such rules to match only when some (un)desirable non-local property holds, and then verify automatically that the rule is never applicable. Augur 2 [11] also uses regular expressions, but for expressing forbidden paths that should not occur in any reachable graph.

8 Conclusion

This paper has contributed the means for systematic proofs of graph programs with respect to non-local specifications. In particular, we defined M-conditions, an extension of nested conditions equivalently expressive to MSO logic on graphs, and defined for this assertion language an effective construction for weakest liberal preconditions of rules. We demonstrated the use of this work in some Hoare-style proofs of programs relative to non-local invariants, i.e. the existence of 2-colourings, and the existence of arbitrary-length cycles. Some interesting topics for future work include: extending M-conditions and Pre to support other useful predicates (e.g. an *undirected* path predicate), adding support for attribution (e.g. along the lines of [18, 17]), implementing the construction of Pre, and generalising the resolution- and tableau-based reasoning systems for nested conditions [13, 12] to M-conditions.

Acknowledgements. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

References

1. Courcelle, B.: Graph rewriting: An algebraic and logic approach. In: Handbook of Theoretical Computer Science, vol. B, chap. 5. Elsevier (1990)
2. Courcelle, B.: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. Information and Computation 85(1), 12–75 (1990)
3. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach. Cambridge University Press (2012)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
5. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer (2006)
6. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Software Tools for Technology Transfer 14(1), 15–40 (2012)
7. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19(2), 245–296 (2009)
8. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer (2006)
9. Habel, A., Radke, H.: Expressiveness of graph conditions with variables. In: GraMoT 2010. Electronic Communications of the EASST, vol. 30 (2010)
10. Inaba, K., Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Graph-transformation verification using monadic second-order logic. In: PPDP 2011. pp. 17–28. ACM (2011)
11. König, B., Kozioura, V.: Augur 2 - a new version of a tool for the analysis of graph transformation systems. In: GT-VMT 2006. ENTCS, vol. 211, pp. 201–210. Elsevier (2008)
12. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: ICGT 2014. LNCS, vol. 8571, pp. 17–32. Springer (2014)
13. Pennemann, K.H.: Resolution-like theorem proving for high-level conditions. In: ICGT 2008. LNCS, vol. 5214, pp. 289–304. Springer (2008)
14. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Doctoral dissertation, Universität Oldenburg (2009)
15. 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. Springer (2013)
16. Plump, D.: The design of GP 2. In: WRS 2011. EPTCS, vol. 82, pp. 1–16 (2012)
17. Poskitt, C.M.: Verification of Graph Programs. Ph.D. thesis, University of York (2013)
18. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
19. Poskitt, C.M., Plump, D.: Verifying total correctness of graph programs. In: GCM 2012. Electronic Communications of the EASST, vol. 61 (2013)
20. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs: Extended version (2014), `http://arxiv.org/abs/1405.5927`
21. Radke, H.: HR* graph conditions between counting monadic second-order and second-order graph formulas. In: GCM 2012. Electronic Communications of the EASST, vol. 61 (2013)