Verifying Graph Programs with Monadic Second-Order Logic

Gia S. Wulandari*[1,2] and Detlef Plump[1]

[1] Department of Computer Science, University of York, UK
[2] School of Computing, Telkom University, Indonesia

Abstract. To verify graph programs in the language GP 2, we present a monadic second-order logic with counting and a Hoare-style proof calculus. The logic has quantifiers for GP 2's attributes and for sets of nodes or edges. This allows to specify non-local graph properties such as connectedness, k-colourability, etc. We show how to construct a strongest liberal postcondition for a given graph transformation rule and a precondition. The proof rules establish the total correctness of graph programs and are shown to be sound. They allow to verify more programs than is possible with previous approaches. In particular, many programs with nested loops are covered by the calculus.

1 Introduction

GP 2 is a programming language based on graph transformation rules which aims to facilitate formal reasoning. Graphs and rules in GP 2 can be attributed with heterogeneous lists of integers and character strings. The language has a simple formal semantics and is computationally complete [15].

The verification of graph programs with various Hoare-style calculi is studied in [17,18,16,19] based on so-called E-conditions or M-conditions as assertions. E-conditions are an extension of nested graph conditions [7,13] with attributes (list expressions). They can express first-order properties of GP 2 graphs, while M-condition can express monadic second-order properties (without counting) of non-attributed GP 2 graphs. In both cases, verification is restricted to the class of graph programs whose loop bodies and branching guards are rule-set calls.

In this paper, we introduce a monadic second-order logic (with counting) for GP 2. We define the formulas based on a standard logic for graphs enriched with GP 2 features such as list attributes, indegree and outdegree functions for nodes, etc. We prefer to use standard logic because we believe it is easier to comprehend by programmers that are not familiar with graph morphisms and commuting diagrams. Another advantage of a standard logic is the potential for using theorem proving environments such as Isabelle [10,11], Coq [12], or Z3 [1].

In [21] we show how to prove programs partially correct by using closed first-order formulas as assertions. The class of graph programs that can be verified with the calculi of [21] consists of the so-called control programs. These programs

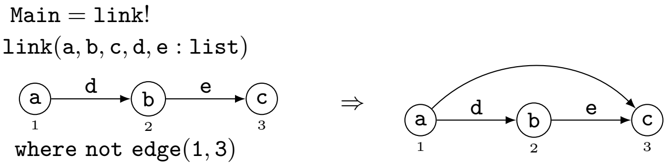
* Supported by Indonesia Endowment Fund for Education (LPDP)

may contain certain nested loops and branching commands with arbitrary loop-free programs as guards. Hence, the class of programs that can be handled is considerably larger than the class of programs verifiable with [16].

Here, we continue that work and show how to prove total correctness of control programs in the sense that programs are both partially correct and terminating. Also, to generalise the program properties that can be verified, we use closed monadic second-order formulas as assertions. This allows to prove non-local properties such as connectedness or k-colourability. Our main technical result is the construction of a strongest liberal postcondition from a given precondition and a GP 2 transformation rule. This operation serves as the axiom in the proof calculus of Section 5.

2 The Graph Programming Language GP 2

GP 2 programs transform input graphs into output graphs, where graphs are directed and may contain parallel edges and loops. Formally, a graph G is a system $\langle V_G, E_G, s_G, t_G, l_G, m_G, p_G\rangle$ comprising two finite sets of vertices and edges, source and target functions, a partial node labelling function, an edge labelling function, and a partial root function. Nodes v for which $l_G(v)$ or $p_G(v)$ is undefined may only exist in the interface of GP 2 rules, but not in host graphs. Nodes and edges are labelled with lists consisting of integers and character strings. This includes the special case of items labelled with the empty list which may be considered as "unlabelled".



```
Main = link!
link(a, b, c, d, e : list)
where not edge(1, 3)
```

Fig. 1: Graph program `transitive-closure` [14]

The principal programming construct in GP 2 are conditional graph transformation rules labelled with expressions. For example, the rule `link` in Fig. 1 has five formal parameters of type `list`, a left-hand graph and a right-hand graph which are specified graphically, and a textual condition starting with the keyword `where`. Node identifiers are written below the nodes, and all other text in the graphs consists of labels. Parameters are typed as `list`, `atom`, `int`, `string`, or `char`, where `atom` stands for the union of integers and strings, and lists are arbitrary sequences of atoms.

Besides carrying expressions, nodes and edges can be marked red, green or blue. Also, nodes can be marked grey and edges can be dashed. An example with red and grey nodes and a dashed edge can be seen in Fig. 5 of Section 6.

Rules are applied to host graphs in a two-stage process. First a rule is instantiated by replacing all variables with values of the same type, evaluating all expressions in the right-hand side of the rule, and checking the application condition. This yields a standard rule in the so-called double-pushout approach with relabelling [8]. Next, the instantiated rule is applied to the host graph by constructing two suitable natural pushouts [2].

A program consists of declarations of conditional rules and (non-recursive) procedures, including a distinct procedure named `Main`. Next we briefly describe GP 2's major control constructs.

A *rule-set call* $\{r_1, \ldots, r_n\}$ non-deterministically applies one of the applicable rules to the host graph. The call *fails* if none of the rules is applicable to the host graph.

The *sequential composition* of programs P and Q is written $P; Q$.

The command `if` C `then` P `else` Q is executed on a host graph G by first executing C on a copy of G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The command `try` C `then` P `else` Q has a similar effect, except that P is executed on the result of C's execution.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body was entered for the last time. The `break` command inside a loop terminates that loop and transfers control to the command following the loop.

In general, the execution of a program P on a host graph G may result in different graphs, fail, or diverge. This is formally defined by the operational semantics of GP 2 which assigns to P and G the set $[\![P]\!]G$ of all possible execution outcomes. See, for example, [15].

3 Monadic Second-Order Formulas for Graph Programs

We define MSO formulas which specify classes of GP 2 host graphs. The abstract syntax of formulas is shown in Fig. 2, where type names, arithmetic operators, and special operators such as `edge`, `root`, `indeg`, `outdeg`, etc. are inherited from the GP 2 syntax. The category Char is the set of all printable ASCII characters except '"', and Digit is the set $\{0, \ldots, 9\}$. All variables are typed, with associated domains as in Table 1.

Table 1: Variable types and their domain over a graph G

type	Node	Edge	SetNode	SetEdge	List	Atom	Int	String	Char
domain	V_G	E_G	2^{V_G}	2^{E_G}	$(\mathbb{Z} \cup \mathsf{Char}^*)^*$	$\mathbb{Z} \cup \mathsf{Char}^*$	$\mathbb{Z}$	Char^*	Char

The types for labels form a subtype hierarchy, given by `list` $\supset$ `atom` $\supset$ `int`, `string` and `string` $\supset$ `char`, where atoms are considered as lists of length one and characters are considered as strings of length one. Hence list variables may have integer, string, or character values. Such restrictions can be enforced by subtype predicates. For example, the list variable x can be constrained to hold an integer value by the predicate $\mathsf{int}(\mathsf{x})$.

For brevity, we write $c \Rightarrow d$ for $\neg c \vee d$, $c \Leftrightarrow d$ for $(c \Rightarrow d) \wedge (d \Rightarrow c)$, $\forall_\mathsf{V}\mathsf{x}(c)$ for $\neg\exists_\mathsf{v}\mathsf{x}(\neg c)$, and similarly with $\forall_\mathsf{e}\mathsf{x}(c), \forall_\mathsf{l}\mathsf{x}(c), \forall_\mathsf{V}\mathsf{X}(c)$, and $\forall_\mathsf{E}\mathsf{X}(c)$. We also sometimes write $\exists_\mathsf{v}\mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{n}(\mathsf{c})$ for $\exists_\mathsf{v}\mathsf{x}_1(\exists_\mathsf{v}\mathsf{x}_2(\ldots\exists_\mathsf{v}\mathsf{x}_\mathsf{n}(\mathsf{c})\ldots))$ (also for other

Formula ::= true | false | Elem | Cond | Equal
| Formula (‘$\wedge$’ | ‘$\vee$’) Formula | ‘$\neg$’Formula | ‘(’Formula‘)’
| ‘$\exists_\mathsf{v}$’ NodeVar ‘(’Formula‘)’ | ‘$\exists_\mathsf{e}$’ EdgeVar ‘(’Formula‘)’
| ‘$\exists_\mathsf{l}$’ (ListVar) ‘(’Formula‘)’
| ‘$\exists_\mathsf{V}$’ SetNodeVar ‘(’Formula‘)’ | ‘$\exists_\mathsf{E}$’ SetEdgeVar ‘(’Formula‘)’
Number ::= Digit {Digit}
Elem ::= Node (‘$\in$’ | ‘$\notin$’) SetNodeVar | EdgeVar (‘$\in$’ | ‘$\notin$’) SetEdgeVar
Cond ::= (int | char | string | atom) ‘(’Var‘)’
| Lst (‘=’ | ‘$\neq$’) Lst
| Int (‘>’ | ‘>=’ | ‘<’ | ‘<=’) Int
| edge ‘(’ Node ‘,’ Node [‘,’ Label] [‘,’ EMark] ‘)’
| path ‘(’ Node ‘,’ Node [‘,’ SetEdgeVar] ‘)’
| root ‘(’ Node ‘)’
Var ::= ListVar | AtomVar | IntVar | StringVar | CharVar
Lst ::= empty | Atm | Lst ‘:’ Lst | ListVar | l_V ‘(’Node‘)’ | l_E ‘(’EdgeVar‘)’
Atm ::= Int | String | AtomVar
Int ::= [‘-’] Number | ‘(’Int‘)’ | IntVar
| Int (‘+’ | ‘-’ | ‘*’ | ‘/’) Int
| (indeg | outdeg) ‘(’Node‘)’
| length ‘(’AtomVar | StringVar | ListVar‘)’
| card‘(’(SetNodeVar | SetEdgeVar)‘)’
String ::= ‘ “ ’ Char ‘ ” ’ | CharVar | StringVar | String ‘.’ String
Node ::= NodeVar | (s | t) ‘(’ EdgeVar‘)’
EMark ::= none | red | green | blue | dashed | any | m_E‘(′EdgeVar‘)′
VMark ::= none | red | blue | green | grey | any | m_V‘(′Node‘)
Equal ::= Node (’=’ | ‘$\neq$’) Node | EdgeVar (’=’ | ‘$\neq$’) EdgeVar
| Lst (’=’ | ‘$\neq$’) Lst | VMark (’=’ | ‘$\neq$’) VMark
| EMark (’=’ | ‘$\neq$’) EMark

Fig. 2: Abstract syntax of monadic second-order formulas

quantifiers). Terms in MSO formulas are defined as usual and may contain function symbols, constants and variables.

Example 1 (Monadic second-order formulas).
1) $\exists_\mathsf{V}\mathsf{X}(\forall_\mathsf{v}\mathsf{x}(\mathsf{x} \in \mathsf{X} \Rightarrow \mathsf{m_V}(\mathsf{x}) = \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2)$ expresses “there exists at least two unmarked nodes”.
2) $\exists_\mathsf{V}\mathsf{X}(\forall_\mathsf{V}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \Leftrightarrow \mathsf{x} \in \mathsf{X}) \wedge \exists_\mathsf{l}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2 * \mathsf{n}))$ expresses “the number of grey nodes is even”.

Note that the first-order formula $\exists_\mathsf{v}\mathsf{x},\mathsf{y}(\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \wedge \mathsf{m_V}(\mathsf{y}) = \mathsf{none} \wedge \mathsf{x} \neq \mathsf{y})$ is equivalent to the first formula. But it is unlikely that the second formula can be expressed in the first-order fragment of our MSO logic because pure first-order logic on graphs (without built-in functions and relations) cannot specify that the number of nodes is even [6].

The truth value of an MSO formula over a graph is defined via assignments, which are functions mapping free variables to their domains.

Definition 1 (Assignment). Consider an MSO formula c. Let A, B, C, D, E be the set of free node, edge, list, node-set, and edge-set variables in c, respectively. Given a free variable x, we write $\mathsf{dom}(\mathsf{x})$ for the domain of x as defined by

Table 1. A *formula assignment* for c over a host graph G is a pair $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ where $\alpha_G = \langle \alpha_V : A \rightarrow V_G, \alpha_E : B \rightarrow E_G, \alpha_{2V} : D \rightarrow 2^{V_G}, \alpha_{2E} : E \rightarrow 2^{E_G}) \rangle$ and $\alpha_{\mathbb{L}} : C \rightarrow \mathbb{L}$, such that for each free variable x, $\alpha(\mathsf{x}) \in \mathsf{dom}(\mathsf{x})$. We denote by c^α the (first-order) formula resulting from c after replacing each term y with y^α, where y^α is defined inductively as follows:

1. If y is a free variable, $y^\alpha = \alpha(y)$;
2. If y is a constant, $y^\alpha = y$;
3. If $y = \mathsf{length}(\mathsf{x})$ for some list variable x, y^α equals to the number of characters in x^α if x is a string variable, 1 if x is an integer variable, or the number of atoms in x^α if x is a list variable;
4. If $y = \mathsf{card}(\mathsf{X})$ for some node-set or edge-set variable X, y^α is the number of elements in X^α;
5. If y is the functions $\mathsf{s}(\mathsf{x}), \mathsf{t}(\mathsf{x}), \mathsf{l_E}(\mathsf{x}), \mathsf{m_E}(\mathsf{x}), \mathsf{l_V}(\mathsf{x}), \mathsf{m_V}(\mathsf{x}), \mathsf{indeg}(\mathsf{x})$, or $\mathsf{outdeg}(\mathsf{x})$, y^α is $s_G(\mathsf{x}^\alpha)$, $t_G(\mathsf{x}^\alpha)$, $\ell^E_G(\mathsf{x}^\alpha), m^E_G(\mathsf{x}^\alpha), \ell^V_G(\mathsf{x}^\alpha), m^V_G(\mathsf{x}^\alpha)$, indegree of x^α in G , or outdegree of x^α in G, respectively;
6. If $y = \mathsf{x}_1 \oplus \mathsf{x}_2$ for $\oplus \in \{+, -, *, /\}$ and integers ${\mathsf{x}_1}^\alpha, {\mathsf{x}_2}^\alpha$, $y^\alpha = \mathsf{x}_1 \oplus_{\mathbb{Z}} \mathsf{x}_2$;
7. If $y = \mathsf{x}_1.\mathsf{x}_2$ for some terms ${\mathsf{x}_1}^\alpha, {\mathsf{x}_2}^\alpha$, y^α is string concatenation x_1 and x_2;
8. If $y = \mathsf{x}_1 : \mathsf{x}_2$ for some lists ${\mathsf{x}_1}^\alpha, {\mathsf{x}_2}^\alpha$, y^α is list concatenation x_1 and x_2 □

A graph G satisfies a formula c, denoted by $G \models c$, if there exists an assignment α for c over G such that c^α is true. Table 2 shows how the truth value of c^α is determined.

Table 2: Truth value of c^α in graph G

$\mathbf{c}^\alpha$	**true iff**
true	true
false	false
$\mathsf{int}(\mathsf{x})$	$\mathsf{x} \in \mathbb{Z}$
$\mathsf{char}(\mathsf{x})$	$\mathsf{x} \in \mathrm{Char}$
$\mathsf{string}(\mathsf{x})$	$\mathsf{x} \in \mathrm{Char}^*$
$\mathsf{atom}(\mathsf{x})$	$\mathsf{x} \in \mathbb{Z} \cup \mathrm{Char}^*$
$\mathsf{root}(\mathsf{x})$	$p_G(\mathsf{x}) = 1$
$\mathsf{t}_1 \otimes \mathsf{t}_2$	$\mathsf{t}_1 \otimes_{\mathbb{Z}} \mathsf{t}_2$
$\mathsf{X} \oslash \mathsf{Y}$	$\mathsf{X} \oslash_{\mathbb{Z}} \mathsf{Y}$
$\mathsf{x} \in \mathsf{X}$	$\mathsf{x} \in_{\mathbb{Z}} \mathsf{X}$

$\mathbf{c}^\alpha$	**true iff**
$\mathsf{edge}(\mathsf{x}, \mathsf{y}, \mathsf{l}, \mathsf{m})$	$s_G(e) = \mathsf{x}$ and $t_G(e) = \mathsf{y}$ for some $e \in E_G$ where $l^E_G(e) = \mathsf{l}$ and $m^E_G(e) = \mathsf{m}$

$\mathsf{path}(\mathsf{x}, \mathsf{y}, \mathsf{E})$	for some $e_1, \ldots, e_n \in E_G - \mathsf{E}$,
	$s_G(e_1) = a$, $s_G(e_n) = b$,$t_G(e_i) = s_G(e_{i+1})$
	for every $i = 1, \ldots, n-1$

$\mathsf{t}_1 \ominus \mathsf{t}_2$	if t_1 (or t_2) is any: t_2 (or t_1) $\ominus_{\mathbb{B}}$$\mathsf{blue}, \mathsf{red}, \mathsf{green}, \mathsf{gray}$, or dashed; otherwise: $\mathsf{t}_1 \ominus_{\mathbb{B}} \mathsf{t}_2$

$\mathbf{c}^\alpha$	**true iff**
$\neg \mathsf{b}$	b is false in G
$\mathsf{b}_1 \vee \mathsf{b}_2$	b_1 is true in G or b_2 is true in G
$\mathsf{b}_1 \wedge \mathsf{b}_2$	both b_1 and b_2 are true in G
$\exists_\mathsf{v} \mathsf{x}(\mathsf{b})$	$\mathsf{b}^{[\mathsf{x} \mapsto \mathsf{v}]}$ is true in G for some $v \in V_G$
$\exists_\mathsf{e} \mathsf{x}(\mathsf{b})$	$\mathsf{b}^{[\mathsf{x} \mapsto \mathsf{e}]}$ is true in G for some $e \in E_G$
$\exists_\mathsf{l} \mathsf{x}(\mathsf{b})$	$\mathsf{b}^{[\mathsf{x} \mapsto \mathsf{l}]}$ is true in G for some $l \in \mathbb{L}$
$\exists_\mathsf{V} \mathsf{X}(\mathsf{b})$	$\mathsf{b}^{[\mathsf{X} \mapsto \mathsf{V}]}$ is true in G for some $V \in 2^{V_G}$
$\exists_\mathsf{E} \mathsf{X}(\mathsf{b})$	$\mathsf{b}^{[\mathsf{X} \mapsto \mathsf{E}]}$ is true in G for some $E \in 2^{E_G}$

In the table, $\otimes \in \{>, >=, <, <=\}$, $\ominus \in \{=, \neq\}$, $\oslash \in \{=, \neq, \subset, \subseteq\}$, $\otimes_{\mathbb{Z}}$ is the integer operation represented by $\otimes$, and $\ominus_{\mathbb{B}}$ (or $\oslash_{\mathbb{B}}$) is the Boolean operation represented by $\ominus$ (or $\oslash$). Also, given a Boolean expression b, a (set) variable x, and a constant i, we denote by $b^{[x \mapsto i]}$ the expression obtained from b by changing every occurrence of x to i.

4 Constructing a Strongest Liberal Postcondition

In this section, we present a construction that can be used to obtain a strongest liberal postcondition from a given precondition and a rule schema. Here, we limit the precondition to closed MSO formulas.

Definition 2 (Strongest liberal postcondition over a conditional rule schema). An assertion d is a *liberal postcondition* w.r.t. a conditional rule schema r and a precondition c, if for all host graphs G and H, $G \vDash c$ and $G \Rightarrow_r H$ implies $H \vDash d$. A *strongest liberal postcondition* w.r.t. c and r, denoted by $\mathrm{SLP}(c, r)$, is a liberal postcondition w.r.t. c and r that implies every liberal postcondition w.r.t. c and r. □

In [21], we show how to construct a strongest liberal postcondition over FO formulas. Here, we use the same approach in the construction, that is, by obtaining a left-application condition, which then be used to obtain a right-application condition, so that finally we can obtain a strongest liberal postcondition.

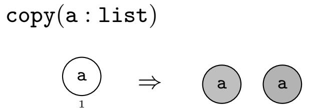


Fig. 3: GP 2 conditional rule schema copy

As a running example, let us consider the rule schema copy of Fig. 3 and the MSO formula e expressing “the number of grey nodes is even”:
$e \equiv \exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m}_{\mathsf{V}}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})) \wedge \exists_{\mathsf{I}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2 * \mathsf{n}))$.
Note that the interface of the rule copy is the empty graph. We intentionally do not preserve the node 1 and have two new nodes instead to see the effect of both removal and addition of an element in the construction of a strongest liberal postcondition.

Remark 1. In the following subsections we explain the transformations involved in the construction of a strongest liberal postcondition. For this purpose, we consider a generalised form of MSO formulas called *conditions*, which may contain node and edge constants. Also, we consider a generalised form of rule schemata which have both a left and a right application condition, where the conditions can be more expressive than the application conditions of GP 2 rule schemata.

4.1 From Precondition to Left-Application Condition

We start with the transformation of a precondition to a left-application condition with respect to a conditional rule schema $r = \langle L \leftarrow K \rightarrow R, \mathit{\Gamma} \rangle$. Intuitively, the transformation is done by:

1. Expressing the dangling condition as a condition over L, denoted by $\mathrm{Dang}(r)$.
2. Finding all possibilities of variables in c representing nodes/edges in a match of L and of forming a disjunction from all possibilities, denoted by $\mathrm{Split}(c, r)$.
3. Evaluating terms and Boolean expression we can evaluate in $\mathrm{Split}(c, r)$, $\mathrm{Dang}(r)$, and Γ with respect to the left-hand graph of the given rule, then form a conjunction from the result of evaluation, and simplify the conjunction.

4.1.1 Condition Dang

The dangling condition must be satisfied by an injective morphism g if $G \Rightarrow_{r,g} H$ for some rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ and host graphs G, H. A graph G with an injective morphism $g : L \rightarrow G$ satisfies the dangling condition if every node $v \in g(L-K)$ is not incident to any edge outside $g(L)$. That is, all edges incident to a deleted node must be in $g(L)$. This means that the indegree and outdegree of each deleted node $g^{-1}(v) \in L-K$ are the same as the indegree and outdegree of v in G.

Definition 3 (Condition Dang). Consider a rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ where $\{v_1, \cdots, v_n\}$ is the set of nodes in $L - K$. Let $indeg_L(v)$ and $outdeg_L(v)$ denote the indegree and outdegree of a node v in L. The condition $\mathrm{Dang}(r)$ is defined as follows:

1. if $V_L - V_K = \emptyset$ then $\mathrm{Dang}(r) = \mathsf{true}$
2. if $V_L - V_K \neq \emptyset$ then
   $\mathrm{Dang}(r) = \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{indeg}(\mathsf{v_i}) = indeg_L(v_i) \wedge \mathsf{outdeg}(\mathsf{v_i}) = outdeg_L(v_i)$ □

Example 2.
For the rule $r = \mathsf{copy}$ (see Fig. 3): $\mathrm{Dang}(r) = \mathsf{indeg}(1) = 0 \wedge \mathsf{outdeg}(1) = 0$

4.1.2 Transformation Split

A node (or edge) variable x in c can represent any node (or edge) in an input graph, in the sense that we can substitute any node (or edge) in G to check the truth value of c in G (see point 5 and 6 of Definition 5). Also, a node (or edge) set variable X in c can represent any set of nodes (or edges) in the input graph, where each node (or edge) in the image of a match may or may not be an element of the set (see point 8 and 9 of Definition 5).

To express that a set of nodes/edges in L is a subset of a set of nodes/edges represented by a set variable, we define subset formulas.

Definition 4 (Subset Formula). Given a set of nodes $N = \{v_1, \ldots, v_n\}$, a *subset formula* for N with respect to a node set variable X has the form $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ where for $i = 1, \ldots, n$, $c_i = v_i \in X$ or $v_i \notin X$. The formula true is the only subset formula for the empty set with respect to any set variable. □

Definition 5 (Transformation Split). Let us consider a rule schema $r = \langle L \leftarrow K \rightarrow R, \Gamma\rangle$, where $V_L = \{v_1, \ldots, v_n\}$ and $E_L = \{e_1, \ldots, e_m\}$. Let $\{V_1, \ldots, V_{2^n}\}$ be the power set of V_L, and $d_1, \ldots, d_{2^n}$ be subset formulas of V_L w.r.t. X where for every

$i = 1, \ldots, 2^n$, d_i represents V_i. Similarly, let $\{E_1, \ldots, E_{2^m}\}$ be the power set of E_L, and $a_1, \ldots, a_{2^m}$ be subset formulas of E_L w.r.t. X where for every $i = 1, \ldots, 2^m$, a_i represents E_i.

Let c be a condition over L sharing no variables with r (note that it is always possible to replace the label variables in c with new variables that are distinct from variables in r). We define the condition $\mathrm{Split}(c, r)$ over L inductively as follows, where c_1, c_2 are conditions over L:

1) If c is either true, false, a predicate $\mathsf{int(t)}, \mathsf{char(t)}, \mathsf{string(t)}, \mathsf{atom(t)}, \mathsf{root(t)}$ for some term t, in the form $\mathsf{t_1} \ominus \mathsf{t_2}$ for $\ominus \in \{= . \neq . <, \leq, >, \geq\}$ and some terms $\mathsf{t_1}, \mathsf{t_2}$, or in the form $\mathsf{x} \in \mathsf{X}$ or $\mathsf{x} \notin \mathsf{X}$,
$$\mathrm{Split}(c, r) = c$$
2) $\mathrm{Split}(c_1 \vee c_2, r) = \mathrm{Split}(c_1, r) \vee \mathrm{Split}(c_2, r)$,
3) $\mathrm{Split}(c_1 \wedge c_2, r) = \mathrm{Split}(c_1, r) \wedge \mathrm{Split}(c_2, r)$,
4) $\mathrm{Split}(\neg c_1, r) = \neg\mathrm{Split}(c_1, r)$,
5) $\mathrm{Split}(\exists_\mathsf{v}\mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i=1}}^{\mathsf{n}} \mathrm{Split}(c_1^{[x \mapsto v_i]}, r)) \vee \exists_\mathsf{V}\mathsf{x}(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{x} {\neq} \mathsf{v_i} \wedge \mathrm{Split}(c_1, r)$,
6) $\mathrm{Split}(\exists_\mathsf{e}\mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i=1}}^{\mathsf{m}} \mathrm{Split}(c_1^{[x \mapsto e_i]}, r)) \vee \exists_\mathsf{e}\mathsf{x}(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}} \mathsf{x} {\neq} \mathsf{e_i} \wedge \mathrm{inc}(c_1, r, x))$,
where
$$\begin{aligned} \mathrm{inc}(c_1, r, x) = {} & \textstyle\bigvee_{\mathsf{i=1}}^{\mathsf{n}}(\bigvee_{\mathsf{j=1}}^{\mathsf{n}} \mathsf{s(x)} = \mathsf{v_i} \wedge \mathsf{t(x)} = \mathsf{v_j} \wedge \mathrm{Split}(c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_i}, \mathsf{t(x)} \mapsto \mathsf{v_j}]}, r)) \\ & \vee (\mathsf{s(x)} = \mathsf{v_i} \ \wedge \ \textstyle\bigwedge_{\mathsf{j=1}}^{\mathsf{n}} \mathsf{t(x)} \neq \mathsf{v_j} \wedge \mathrm{Split}(c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_i}]}, r)) \\ & \vee (\textstyle\bigwedge_{\mathsf{j=1}}^{\mathsf{n}} \mathsf{s(x)} \neq \mathsf{v_j} \ \wedge \ \mathsf{t(x)} = \mathsf{v_i} \wedge \mathrm{Split}(c_1^{[\mathsf{t(x)} \mapsto \mathsf{v_i}]}, r)) \\ & \vee (\textstyle\bigwedge_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{s(x)} \neq \mathsf{v_i} \ \wedge \ \textstyle\bigwedge_{\mathsf{j=1}}^{\mathsf{n}} \mathsf{t(x)} \neq \mathsf{v_j} \wedge \mathrm{Split}(c_1, r)) \end{aligned}$$
7) $\mathrm{Split}(\exists_\mathsf{l}\mathsf{x}(c_1), r) = \exists_\mathsf{l}\mathsf{x}(\mathrm{Split}(c_1, r))$
8) $\mathrm{Split}(\exists_\mathsf{V}\mathsf{X}(c_1), r) = \exists_\mathsf{V}\mathsf{X}(\bigwedge_{i=1}^{2^n} d_i \Rightarrow \mathrm{Split}(c_1, r))$
9) $\mathrm{Split}(\exists_\mathsf{E}\mathsf{X}(c_1), r) = \exists_\mathsf{E}\mathsf{X}(\bigwedge_{i=1}^{2^m} a_i \Rightarrow \mathrm{Split}(c_1, r))$

where $c^{[a \mapsto b]}$ for a variable or function a and constant b represents the condition c after the replacement of all occurrence of a with b. □

Intuitively, we only need to consider substituting nodes in L for each term in c representing a node (a node variable or a source or target function), and similarly, edges in L for all edge variables in c. In addition, we need to consider all possible ways in which nodes/edges in L are elements of a set in c.

Example 3.
Consider again the precondition e from our running example:
$\exists_\mathsf{V}\mathsf{X}(\neg\exists_\mathsf{v}\mathsf{x}((\mathsf{m_V(x)} = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V(x)} \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})) \wedge \exists_\mathsf{l}\mathsf{n}(\mathsf{card(X)} = 2 * \mathsf{n}))$
has the form of $\exists_\mathsf{V}\mathsf{X}(c_1)$. From point 8 and 3 of Definition 5, for
$d = \exists_\mathsf{v}\mathsf{x}((\mathsf{m_V(x)} = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V(x)} \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))$, we have
$$\begin{aligned} \mathrm{Split}(e, r) = {} & \exists_\mathsf{V}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \mathrm{Split}(\neg\mathsf{d}, \mathsf{r}) \wedge \mathrm{Split}(\exists_\mathsf{l}\mathsf{n}(\mathsf{card(X)} = 2 * \mathsf{n}), \mathsf{r})) \\ & \wedge (1 \notin \mathsf{X} \Rightarrow \mathrm{Split}(\neg\mathsf{d}, \mathsf{r}) \wedge \mathrm{Split}(\exists_\mathsf{l}\mathsf{n}(\mathsf{card(X)} = 2 * \mathsf{n}), \mathsf{r}))). \end{aligned}$$
We know that $\mathrm{Split}(\exists_\mathsf{l}\mathsf{n}(\mathsf{card(X)} = 2 * \mathsf{n}), r)$ is equal to $\exists_\mathsf{l}\mathsf{n}(\mathsf{card(X)} = 2 * \mathsf{n})$ (see point 7 of Definition 5), while $\mathrm{Split}(\neg d, r) = \neg\mathrm{Split}(d, r)$ (see point 4 of Definition 5). Then from point 5 of Definition 5, we have
$$\begin{aligned} \mathrm{Split}(d, r) = {} & (\mathsf{m_V}(1) = \mathsf{grey} \wedge 1 \notin \mathsf{X}) \vee (\mathsf{m_V}(1) \neq \mathsf{grey} \wedge 1 \in \mathsf{X}) \\ & \vee \ \exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{m_V(x)} = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V(x)} \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})) \end{aligned}$$
so that
$$\begin{aligned} \mathrm{Split}(e, r) = {} & \exists_\mathsf{V}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg\mathrm{Split}(\mathsf{d}, \mathsf{r}) \wedge \exists_\mathsf{l}\mathsf{n}(\mathsf{card(X)} = 2^*\mathsf{n})) \\ & \wedge (1 \notin \mathsf{X} \Rightarrow \neg\mathrm{Split}(\mathsf{d}, \mathsf{r}) \wedge \exists_\mathsf{l}\mathsf{n}(\mathsf{card(X)} = 2^*\mathsf{n}))) \end{aligned}$$

4.1.3 Transformation Val

The condition resulting from transformation Split, the condition Dang, and the rule schema condition Γ may contain node/edge identifiers of the given left-hand graph. To simplify the conditions, we can check if there is a disjuntion with a true disjunc or a conjunction with a false conjunct so that we can ruled out because of its value in the left-hand graph. For a simple example, a conjunct condition $\mathsf{m_V}(1) = \mathsf{grey}$ can be replaced with false if node 1 in the given left-hand graph is not grey.

Let us consider a rule schema $r = \langle L \leftarrow K \rightarrow R, \Gamma\rangle$, a condition c over L, a host graph G, and a premorphism $g : L \rightarrow G$. Let c share no variables with L. To simplify c w.r.t. L, we apply the transformation $\mathrm{Val}(c, r)$ as follows:

1. Obtain c' from c by replacing terms involving $\mathsf{s}, \mathsf{t}, \mathsf{l_V}, \mathsf{l_E}, \mathsf{m_V}, \mathsf{m_E}$, indeg and outdeg, that do not have node/edge variables as arguments, with their values in L. In addition, we also replace integer, string, and list operations with their values if their arguments are only constants.
   Note that the values of indeg and outdeg depend on the host graph, while here we evaluate them in the left-hand graph. Hence, we use the terms $\mathsf{incon(v)}$ and $\mathsf{outcon(v)}$ as constants representing the indegree resp. outdegree of $g(v)$ minus indegree resp. outdegree of v in L.
2. Obtain c'' from c' by evaluating Boolean operations $=, \neq, \leq, \geq, \mathsf{root}$, if their arguments only consists of constants, to their values in L.
3. Consider any implication of the form $a \Rightarrow d$ for some subset formula a and condition d to $a \Rightarrow d^T$. d^T is obtained from d by changing every subcondition of the form $i \in X$ for $i \in V_L$, $i \in E_L$ and set variable X to true if $i \in X$ is implied by a or false otherwise.
4. Simplify c''' by simplifying conjunct disjunct involving true or false. Also, change the subconditions of the forms $\neg\,\mathsf{true}$,$\neg(\neg\, a)$, $\neg(a \vee b)$, $\neg(\mathsf{a} \wedge \mathsf{b})$, and $\mathsf{a} \Rightarrow \mathsf{false}$ for some conditions a, b to $\mathsf{false}, \mathsf{a}, \neg\mathsf{a} \wedge \neg\mathsf{b}, \neg\mathsf{a} \vee \neg\mathsf{b}, \neg\mathsf{a}$ resp. □

The formal definition of $\mathrm{Val}(c, r)$ is rather long [22] because the expressions we have in a condition may be nested. Hence, we do not present it in this paper.

Example 4. Let $f = \mathrm{Split}(e, r)$ from Example 3. That is,

$$f = \exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg\mathrm{Split}(\mathsf{d},\mathsf{r}) \wedge \exists_{\mathsf{I}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n}))$$
$$\wedge\, (1 \notin \mathsf{X} \Rightarrow \neg\mathrm{Split}(\mathsf{d},\mathsf{r}) \wedge \exists_{\mathsf{I}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n})))$$

where $\mathrm{Split}(d, r) = (\mathsf{m_V}(1) = \mathsf{grey} \wedge 1 \notin \mathsf{X}) \vee (\mathsf{m_V}(1) \neq \mathsf{grey} \wedge 1 \in \mathsf{X})$
$$\vee\, \exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))$$

Since node 1 in the left-hand graph of r is unmarked, then we can replace $\mathsf{m_V}(1) = \mathsf{grey}$ with false, and $\mathsf{m_V}(1) \neq \mathsf{grey}$ with true.

We also replace $1 \in \mathsf{X}$ and $1 \notin \mathsf{X}$ with true or false, based on the premise in the conjunct of f. That is, replace $1 \in \mathsf{X}$ and $1 \notin \mathsf{X}$ with true and false (resp.) for the first conjunct of f, and with false and true (resp.) for the second conjunct. Hence, we obtain the following condition

$$\exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg((\mathsf{false} \wedge \mathsf{false}) \vee (\mathsf{true} \wedge \mathsf{true}) \vee \mathsf{b}) \wedge \exists_{\mathsf{I}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n}))$$
$$\wedge\, (1 \notin \mathsf{X} \Rightarrow \neg((\mathsf{false} \wedge \mathsf{true}) \vee (\mathsf{true} \wedge \mathsf{false}) \vee \mathsf{b}) \wedge \exists_{\mathsf{I}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n})))$$

where $b = \exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))$.

Finally, we simplify $\neg((\mathsf{false} \wedge \mathsf{false}) \vee (\mathsf{true} \wedge \mathsf{true}) \vee \mathsf{b}) \wedge \exists_{\mathsf{I}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^{*}\mathsf{n})$ to false. Also, $\neg((\mathsf{false} \wedge \mathsf{true}) \vee (\mathsf{true} \wedge \mathsf{false}) \vee \mathsf{b})$ to $\neg\mathsf{b}$. Hence, we finally obtain $\mathrm{Val}(f, r) = \ \exists_{\mathsf{V}}\mathsf{X}(1 \notin \mathsf{X} \Rightarrow \neg\mathsf{b} \wedge \exists_{\mathsf{I}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^{*}\mathsf{n}))$

4.1.4 Transformation Lift

Finally, we define the transformation Lift, which takes a precondition and a rule schema as an input and gives a left-application condition as an output.

Definition 6 (Transformation Lift). Let $r = \langle L \leftarrow K \rightarrow, \Gamma\rangle$ be a rule schema, c be a precondition, and $\mathrm{Lift}(c, r)$ is a left application condition w.r.t. c and r. Then, $\mathrm{Lift}(c, r) = \mathrm{Val}(\mathrm{Split}(c \wedge \Gamma, r) \wedge \mathrm{Dang}(r), r)$.

Example 5.
For the rule schema $r = \mathsf{copy}$, $\Gamma = \mathsf{true}$ and $\mathrm{Dang}(r) = \mathsf{indeg}(1) = 0 \wedge \mathsf{outdeg}(1) = 0$ such that $\mathrm{Val}(\mathrm{Dang}(r), r) = \mathsf{true}$ and $\mathrm{Split}(e \wedge \Gamma, r) = \mathrm{Split}(e, r)$. Hence, $\mathrm{Lift}(e, r^{\vee}) = \mathrm{Val}(\mathrm{Split}(e, r), r)$.

In [22], we show that by using the described construction, we can obtain a left-application condition that is satisfied by every possible match of the given rule schema.

Let us consider the transformation Split. From point 8 and 9 of Definition 5, we know that Split may gives us conjunction of implications in specific form (i.e. implications with subset formula as premise), and such form will still be exist in the resulting condition of the transformation Lift. From now on, we say that the obtained application condition (from Lift) is in 'lifted form'.

4.2 From Left- to Right-Application Condition

To obtain a right-application condition from a left-application condition, we need to consider what properties could be different in the initial and the result graphs. Recall that in constructing a left-application condition, we evaluate all functions with a node/edge constant argument and change them with constant.

4.2.1 Transformation Adj

Due to the deletion of nodes/edges by a rule schema, properties that hold in the initial graph may not hold anymore in the output graph. Hence, we need to adjust the obtained left application condition so that we can have a condition that can be satisfied by a comatch.

For example, the Boolean value for $\mathsf{x} = \mathsf{i}$ for any node/edge variable x and node/edge constant i that gets deleted must be false in the resulting graph. Analogously, $\mathsf{x} \neq \mathsf{i}$ is always true. Also, all variables in the left-application condition should not represent any new nodes and edges in the right-hand side. In addition, we also need to consider the case where we have set variables.

In a lifted form, we may have subformulas of the form $\exists_{\mathsf{V}}\mathsf{X}(\bigwedge_{i=1}^{2^n} d_i \Rightarrow \mathrm{Split}(c_1, r))$ (or similar for edges), where each d_i represent the condition where a subset of V_L is a subset of the set represented by X. A node in V_L may or may not exist in the output graph. Hence, we need to do adjustment by use a property in standard logic.

Definition 7 (Transformation Adj). Given a rule schema $r = \langle L \leftarrow K \rightarrow R, \Gamma\rangle$ where $V_L = \{v_1, \dots, v_n\}$, $E_L = \{e_1, \dots, e_m\}$, $V_K = \{u_1, \dots, u_k\}$, $V_R = \{w_1, \dots, w_p\}$, and $E_R = \{z_1, \dots, z_q\}$, where $v_i \neq w_j$ (or $e_i \neq z_j$) for all v_i and w_j (or e_i and x_j) not in K. Let $\{V_1, \dots, V_{2^n}\}$ be the power set of V_L, and $d_1, \dots, d_{2^n}$ be subset formulas of V_L w.r.t. X where for every $i = 1, \dots, 2^n$, d_i represents V_i. Similarly, let $\{U_1, \dots, U_{2^k}\}$ be the power set of V_K, and $b_1, \dots, b_{2^k}$ be subset formulas of V_K w.r.t. X where for every $i = 1, \dots, 2^k$, b_i represents U_i. Also, let $\{E_1, \dots, E_{2^m}\}$ be the power set of E_L, and $a_1, \dots, a_{2^m}$ be subset formulas of E_L w.r.t. X where for every $i = 1, \dots, 2^m$, a_i represents E_i.

For a condition c over L in lifted form, the *adjusted* condition of c w.r.t. r is defined inductively as below, where $c_1, \dots, c_s$ are conditions over L, for $s \geq 2^m$ and $s \geq 2^n$:

1. If c is the formulas true or false,
   $\mathrm{Adj}(c, r) = c$
2. If c is predicate $\mathsf{int(x), char(x), string(x)}$, or $\mathsf{atom(x)}$ for some list variable x,
   $\mathrm{Adj}(c, r) = c$
3. If c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a list and neither contains free node/edge variable,
   $\mathrm{Adj}(c, r) = c$
4. If c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a node (or edge) and neither contains free node/edge variable or node/edge constant,
   $\mathrm{Adj}(c, r) = c$
5. If c is a Boolean operation $f_1 \diamond f_2$ for $\diamond \in \{=, \neq, <, \leq, >, \geq\}$ and some terms f_1 and f_2 representing integers and neither contains free node/edge variable or any set variables, $\mathrm{Adj}(c, r) =$
   $$\begin{cases} \mathsf{false} & \text{, if } \ominus \in \{=\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ \mathsf{true} & \text{, if } \ominus \in \{\neq\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ c' & \text{, otherwise} \end{cases}$$
6. If c is a Boolean operation $\mathsf{x} \in \mathsf{X}$ for a bounded set variable X and bounded edge variable x, or a bounded set variable X and a bounded node variable x, $\mathsf{x} = \mathsf{s(y)}$ or $\mathsf{x} = \mathsf{t(y)}$ for some bounded edge variable y,
   $\mathrm{Adj}(c, r) = c$
7. If $c = \exists_{\mathsf{l}}\mathsf{x}(\mathsf{c}_1$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}(c, r) = \exists_{\mathsf{l}} x(\mathrm{Adj}(c_1, r))$
8. If $c = \exists_{\mathsf{v}}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c}_1\right)$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}(c, r) = \exists_{\mathsf{v}} x(\bigwedge_{i=1}^{p}, x \neq w_i \wedge \mathrm{Adj}(c_1, r))$
9. If $c = \exists_{\mathsf{e}}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c}_1\right)$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}(c, r) = \exists_{\mathsf{e}} x(\bigwedge_{i=1}^{q}, x \neq z_i \wedge \mathrm{Adj}(c_1, r))$
10. If $c = \exists_{\mathsf{V}}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^n}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form or contains $\mathsf{card(X)}$
    $\mathrm{Adj}(c, r) = \exists_{\mathsf{V}} X(\bigwedge_{v \in V_R - V_K} v \notin X \bigwedge_{i=1}^{2^k}(b_i \Rightarrow \bigvee_{j \in W_i} c'_j))$
    where $c'_j = \mathrm{Adj}(c_j, r)^{[\mathsf{card(X)} \mapsto \mathsf{card(X)} + |(\mathsf{V_L} - \mathsf{V_K}) \cap \mathsf{V_j}|]}$ and for $i = 1, \dots, 2^k$, W_i is a subset of $\{1, \dots, 2^n\}$ such that for all $j \in \{1, \dots, 2^n\}$, $j \in W_i$ iff d_j implies b_i

11. If $c = \exists_\mathsf{E}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^m}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form, construction of $\mathrm{Adj}(c, r)$ is analogous to point 10
12. If $c = c_1 \vee c_2$ for some conditions c_1, c_2 over L in lifted form, $\mathrm{Adj}(c, r) = \mathrm{Adj}(c_1, r) \vee \mathrm{Adj}(c_2, r)$
13. If $c = c_1 \wedge c_2$ for some conditions c_1, c_2 over L in lifted form, $\mathrm{Adj}(c, r) = \mathrm{Adj}(c_1, r) \wedge \mathrm{Adj}(c_2, r)$
14. If $c = \neg c_1$ for some condition c_1 over L in lifted form, $\mathrm{Adj}(c, r) = \neg\mathrm{Adj}(c_1, r)$

Example 6. Let us consider $\mathrm{Lift}(e, r), r)$ from Example 5. That is, the condition
$\exists_\mathsf{V}\mathsf{X}(1 \notin \mathsf{X} \Rightarrow \neg\mathsf{b} \wedge \exists_\mathsf{I}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n}))$
where $b = \exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))$.
From point 10 of Definition 7, we get $\mathrm{Adj}(\mathrm{Lift}(e, r))$ is
$\exists_\mathsf{V}\mathsf{X}(2 \notin \mathsf{X} \wedge 3 \notin \mathsf{X} \wedge (\mathsf{true} \Rightarrow \neg\mathrm{Adj}(\mathsf{b}, \mathsf{r}) \wedge \exists_\mathsf{I}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n})))$
where $\mathrm{Adj}(b, r)$ is
$\exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 2\mathsf{x} \neq 3 \wedge ((\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))$ (see point 5 and 8 of Definition 7. Hence,

$$\mathrm{Adj}(b, r) {=} \exists_\mathsf{V}\mathsf{X}(2 \notin \mathsf{X} \wedge 3 \notin \mathsf{X} \wedge \exists_\mathsf{I}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n}) \\ \wedge \neg\exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 2\mathsf{x} \neq 3 \wedge ((\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \wedge \mathsf{x} \notin \mathsf{X}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))))$$

4.2.2 Condition Spec and Transformation Shift

To have a right application condition that yield to strongest liberal postcondition, we need to have a condition that express properties of right-hand graph, in addition to the condition that derived from the given precondition. Hence, we need a condition that explicitly express the structure, labels, marks of the right-hand graph. Also, the right-application condition should express the dangling condition for any co-match.

To express the structure and properties of R, we use the condition $\mathrm{Spec}(R)$, which specify the right-hand graph uniquely up to the node/edge IDs and name of variables. $\mathrm{Spec}(R)$ is defined as the condition

$$\bigwedge_{\mathsf{i=1}}^{\mathsf{k}} \mathrm{Type}(x_i) \;\wedge\; \bigwedge_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{l_V}(\mathsf{v_i}) = \ell_R^V(v_i) \;\wedge\; \mathsf{m_V}(\mathsf{v_i}) {=} m_R^V(v_i) \;\wedge\; \mathrm{Root}_R(v_i) \\ \wedge\; \bigwedge_{\mathsf{i=1}}^{\mathsf{m}} \mathsf{s}(\mathsf{e_i}) {=} s_R(e_i) \;\wedge\; \mathsf{t}(\mathsf{e_i}) {=} t_R(e_i) \;\wedge\; \mathsf{l_E}(\mathsf{e_i}) = \ell_R^E(e_i) \;\wedge\; \mathsf{m_E}(\mathsf{e_i}) {=} m_R^E(e_i)$$

where $\mathrm{Type}(x)$ for $x \in X$ is $\mathsf{int}(\mathsf{x})$, $\mathsf{char}(\mathsf{x})$, $\mathsf{string}(\mathsf{x})$, $\mathsf{atom}(\mathsf{x})$, or true if x is an integer, char, string, atom, or list variable respectively, and $\mathrm{Root}_L(v)$ for $v \in V_L$ is a function such that $\mathrm{Root}_L(v) = \mathsf{root}(\mathsf{v})$ if $p_L(v) = 1$, and $\mathrm{Root}_L(v) = \neg\mathsf{root}(\mathsf{v})$ otherwise.

Definition 8 (Shifting). Consider a rule schema $r = \langle L \leftarrow K \rightarrow R, \Gamma\rangle$, and a precondition c. The right-application condition w.r.t. c and r, denoted by $\mathrm{Shift}(c, r)$, is defined as:

$$\mathrm{Shift}(c, r) = \mathrm{Adj}(\mathrm{Lift}(c, r), r) \;\wedge\; \mathrm{Spec}(R) \;\wedge\; \mathrm{Dang}(r^{-1}) \qquad \Box$$

Example 7.
$\mathrm{Adj}(\mathrm{Lift}(c, r), r)$ has been obtained from Example 6, where $\mathrm{Spec}(R)$ is the condition $\mathsf{m_V}(2) = \mathsf{grey} \wedge \mathsf{m_V}(3) = \mathsf{grey} \wedge \mathsf{l_V}(2) = \mathsf{a} \wedge \mathsf{l_V}(3) = \mathsf{a}$.
Also, $\mathrm{Dang}(r^{-1}) {=} \mathsf{indeg}(2) = 0 \wedge \mathsf{indeg}(3) = 0 \wedge \mathsf{outdeg}(2) = 0 \wedge \mathsf{outdeg}(3) = 0$ (see Definition 3). Hence, $\mathrm{Shift}(e, r)$ is

$$
\begin{array}{l}
\exists_{\mathsf{V}}\mathsf{X}(\exists_{\mathsf{l}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^{*}\mathsf{n}) \wedge 2 \notin \mathsf{X} \wedge 3 \notin \mathsf{X} \\
\quad \wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_V}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))) \\
\wedge \mathsf{m_V}(2) = \mathsf{grey} \wedge \mathsf{m_V}(3) = \mathsf{grey} \wedge \mathsf{l_V}(2) = \mathsf{a} \wedge \mathsf{l_V}(3) = \mathsf{a} \\
\wedge\, \mathsf{indeg}(2) = 0 \wedge \mathsf{indeg}(3) = 0 \wedge \mathsf{outdeg}(2) = 0 \wedge \mathsf{outdeg}(3) = 0
\end{array}
$$

4.3 From Right-Application Condition to Postcondition

The right-application condition we obtain from transformation Shift is strong enough to express properties of the result graph, w.r.t the comatch. To turn the condition c obtained from Shift to a postcondition, we only need to generalised the condition by the transformation $\mathrm{Var}(c)$, which is obtained from c by substituting fresh variables to node/edge identifiers and adding a constraint that different fresh variables represent different nodes/edges that there is no two new variables express the same node/edge. Finally, we need to bind all free variables to obtain a closed MSO formula.

Definition 9 (Slp). Given a rule $r = \langle r, \Gamma\rangle$ for a rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ and a precondition c. A postcondition w.r.t. c and r, denoted by $\mathrm{Slp}(c, r)$, is the MSO formula $\exists_{\mathsf{v}} x_1, \ldots, x_n(\exists_{\mathsf{e}} y_1, \ldots, y_m(\exists_{\mathsf{l}} z_1, \ldots, z_k(\mathrm{Var}(\mathrm{Shift}(c, r)))))$, where $\{x_1, \ldots, x_n\}$, $\{y_1, \ldots, y_m\}$, and $\{z_1, \cdots, z_k\}$ denote the set of free node, edge, and label (resp.) variables in $\mathrm{Var}(\mathrm{Shift}(c, r))$.

Example 8. First, we need to obtain $\mathrm{Var}(\mathrm{Shift}(e, r))$ by substituting fresh variables to node/edge identifiers in $\mathrm{Shift}(e, r)$ of Example 7. The condition $\mathrm{Shift}(e, r)$ has two node variables, that are 2 and 3. We can then to y and z respectively because we do not both variables in $\mathrm{Shift}(e, r)$. In addition, we also need to add a constraint that $\mathsf{y} \neq \mathsf{z}$. Hence, we have

$$
\begin{array}{ll}
\mathrm{Var}(\mathrm{Shift}(e, r)) = & \mathsf{y} \neq \mathsf{z} \\
& \wedge\exists_{\mathsf{V}}\mathsf{X}(\exists_{\mathsf{l}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2^{*}\mathsf{n}) \wedge \mathsf{y} \notin \mathsf{X} \wedge \mathsf{z} \notin \mathsf{X} \\
& \qquad \wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \neq \mathsf{z} \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_V}(\mathsf{x}) = \mathsf{grey}) \\
& \qquad \text{‘} \qquad\qquad\qquad\qquad \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))) \\
& \wedge \mathsf{m_V}(\mathsf{y}) = \mathsf{grey} \wedge \mathsf{m_V}(\mathsf{z}) = \mathsf{grey} \wedge \mathsf{l_V}(\mathsf{y}) = \mathsf{a} \wedge \mathsf{l_V}(\mathsf{z}) = \mathsf{a} \\
& \wedge\, \mathsf{indeg}(\mathsf{y}) = 0 \wedge \mathsf{indeg}(\mathsf{z}) = 0 \wedge \mathsf{outdeg}(\mathsf{y}) = 0 \wedge \mathsf{outdeg}(\mathsf{z}) = 0
\end{array}
$$

so that
$\mathrm{Slp}(e, r) = \exists_{\mathsf{v}}\mathsf{y}, \mathsf{z}(\exists_{\mathsf{l}}\mathsf{a}(\mathrm{Var}(\mathrm{Shift(e,r)}))))$

Theorem 1 (Strongest liberal postconditions). Given a precondition c and a conditional rule schema $r = \langle\langle L \leftarrow K \rightarrow R\rangle, \Gamma\rangle$. Then, $\mathrm{Slp}(c, r)$ is a strongest liberal postcondition w.r.t. c and r.

In [22], we prove Theorem 1 by showing that $\mathrm{Lift}(c, r)$ and $\mathrm{Shift}(c, r)$ must be satisfied by every match and comatch (resp.).

5 Proof Calculus

In this section, we define a syntactic proof calculus in the sense of total correctness, called SYN.

5.1 The Calculus

Our calculus is a total correctness calculus, which means that a Hoare triple $\{c\}\ P\ \{d\}$ is totally correct if the execution of P on G satisfying c either yields a proper graph or fails (divergence is excluded).

Definition 10 (Partial and total correctness [17]). Consider a precondition c and a postcondition d. A graph program P is *partially correct* with respect to c and d, denoted by $\vDash_{\text{par}} \{c\}\ P\ \{d\}$, if for every host graph G and every graph H in $[\![P]\!]G$, $G \models c$ implies $H \models d$. The triple $\{c\}\ P\ \{d\}$ is *totally correct*, denoted by $\vDash_{\text{tot}} \{c\}\ P\ \{d\}$, if it is partially correct and if for every host graph G satisfying c, P does not diverge or get stuck.

A program can get stuck if it contains a command `if/try` C `then` P `else` Q where C can diverge from a graph G, or it contains a loop $B!$ whose body B can diverge from a graph G. Hence, getting stuck is always a signal of divergence. To prove that a program does not diverge, we use a termination function # which assigns a natural number to every host graph. The proof rule for loops will require that loop bodies decrease the #-value of graphs satisfying the loop invariant. This concept was introduced in [18], but only for loop bodies that are rule set calls.

Definition 11 (Termination function; #-decreasing). A *termination function* is a mapping $\#\colon \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ from host graphs to natural numbers. Given an assertion c and a graph program P, we say that P is *#-decreasing* (under c) if for all graphs $G, H \in \mathcal{G}(\mathcal{L})$ such that $G \vDash c$,

$$\langle P, G\rangle \to^* H \text{ implies } \#G > \#H.$$

To define a proof calculus, we need assertions that can express preconditions of failing or successful executions. For this, we also use the assertion Success and Fail as defined in [21] which can be defined if we consider the classes loop-free programs and iteration commands. A loop-free program simply is a program that has no loop, while an iteration command is inductively defined as: 1) every loop-free program and non-failing command is an iteration command, and 2) a command in the form $C; P$ is an iteration command if C is a loop-free program and P is an iteration command.

Theorem 2. For any loop-free program P and precondition c, there exists MSO formula $\text{Success}(P)$ and $\text{Slp}(c, P)$ such that a graph $G \vDash \text{Success}(P)$ if and only if there exists a host graph $H \in [\![P]\!]G$ and $G \vDash \text{Slp}(c, P)$ if and only if G is a strongest liberal postcondition w.r.t c and P. Also, for any iteration command S, there exists MSO formula $\text{Fail}(P)$ such that $G \vDash \text{Fail}(S)$ if and only if $\text{fail} \in [\![P]\!]G$.

Intuitively, MSO formulas $\text{Success}(P)$ and $\text{Fail}(P)$ are preconditions that assert the existence of successful and failing (resp.) execution of P. In addition, we consider the predicate $\text{Break}(c, P, d)$ for graph command P and assertions c, d as a predicate that is true if and only if for all derivations $\langle P, G\rangle \to^* \langle \texttt{break}, H\rangle, G \vDash c$ implies $H \vDash d$.

From [21], we know that we have constructions for Slp, Success, and Fail as mentioned in Theorem 2 if we have the construction of a strongest liberal postcondition over a rule schema. Since we have it, we can can define the constructions of $\text{Slp}(c, P)$, $\text{Success}(P)$, and $\text{Fail}(P)$ to prove the theorem. As an example, for $\text{Slp}(c, P)$, we can define it inductively as: (i) if P is a rule set call $\mathcal{R} = \{r_1, \ldots, r_n\}$ then $\text{Slp}(c, P)$=$\text{Slp}(c, S) = \text{Slp}(c, r_1) \vee \ldots \vee \text{Post}(c, r_n)$, (ii) if $P = Q$ $\texttt{or}$ S for some programs Q, S then $\text{Slp}(c, P)$=$\text{Slp}(c, Q) \vee \text{Slp}(c, S)$, (iii) if $P = Q; S$ then $\text{Slp}(c, p)$=$\text{Slp}(\text{Slp}(c, Q), S)$, (iv) if $P = \texttt{if}\, C\, \texttt{then}\ \ Q\, \texttt{else}\, S$ for some program C then $\text{Slp}(c, P)$=$\text{Slp}(c \wedge \text{Success}(C), Q) \vee \text{Slp}(c \wedge \text{Fail}(C), S)$, and (v) if $P = \texttt{try}\, C\, \texttt{then}\, Q\, \texttt{else}\, S$ then $\text{Slp}(c, P)$=$\text{Slp}(c \wedge \text{Success}(C)\ \ , C; Q) \vee \text{Slp}(c \wedge \text{Fail}(C), S)$. The construction for Success and Fail can be seen in Appendix.

Definition 12 (Proof rules). The total correctness proof rules is defined in Fig. 4, where c, d, and d' are any conditions, r is any conditional rule schema, $\mathcal{R}$ is any set of rule schemata, C is any loop-free program, P and Q are any control commands, and S is any iteration command.

$$[\text{ruleapp}]_{\text{slp}}\ \frac{}{\{c\}\ r\ \{\text{Slp}(c, r)\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$[\text{comp}]\ \frac{\{c\}\ P\ \{e\} \quad \{e\}\ P\ \{d\}}{\{c\}\ P; Q\ \{d\}}$$

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

$$[\text{if}]\ \frac{\{c \wedge \text{Success}(C)\}\ P\ \{d\} \quad \{c \wedge \text{Fail}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]\ \frac{\{c \wedge \text{Success}(C)\}\ C; P\ \{d\} \quad \{c \wedge \text{Fail}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{alap}]\ \frac{\{c\}\ P\ \{c\} \qquad P \text{ is \#-decreasing under } c \qquad \text{Break}(c, P, d)}{\{c\}\ P!\ \{(c \wedge \text{Fail}(S)) \vee d\}}$$

Fig. 4: Total correctness proof rules of calculus SYN

The proof rules are used to construct proof trees.

Definition 13 (Provability; proof tree[16]). A triple $\{c\}\ P\ \{d\}$ is provable in the calculus, denoted by $\vdash \{c\}\ P\ \{d\}$, if one can construct a *proof tree* from the axioms and inference rules of the calculus with that triple as the root. If $\{c\}\ P\ \{d\}$ is an instance of an axiom X then $(X\ \frac{}{\{c\}\ P\ \{d\}})$ is a proof tree, and $\vdash \{c\}\ P\ \{d\}$. If $\{c\}\ P\ \{d\}$ can be instantiated from the conclusion of an inference rule Y, and there are proof trees $T_1, \ldots, T_n$ with conclusions that are instances of the n premises of Y, then $(Y\ \frac{T_1 \quad \ldots \quad T_n}{\{c\}\ P\ \{d\}})$ is a proof tree, and $\vdash \{c\}\ P\ \{d\}$.

5.2 Soundness

In [21], we show that our partial correctness calculus is sound. Now, we extend it to total correctness calculus, which is also proven to be sound in [23]. We prove the soundness by considering the induction on proof trees.

Theorem 3 (Soundness of the calculus). Given graph program P and MSO formulas c, d. Then, $\vdash \{c\}\ P\ \{d\}$ implies $\models_{\text{tot}} \{c\}\ P\ \{d\}$.

In the calculus, we use $[\text{ruleapp}]_{\text{slp}}$ as an axiom. Alternatively, we can change the axiom to $[\text{ruleapp}]_{\text{wlp}}\ \dfrac{}{\{\neg\text{Slp}(\neg d, r^{-1})\}\ r\ \{d\}}$ and we still have a sound proof calculus [23].

However, relative completeness of the calculus is still an open problem. If we consider FO Hoare-triples, there is a strong evidence that we may have a correct FO Hoare-triple but we can not prove it by our FO proof calculus (see [21]) while we can prove it if by MSO proof calculus, which shows that the expressiveness of assertions play important role in relative completeness.

Courcelle [4,5] has proven that the following properties are not expressible in MSO logic without counting (either with set of node or set of edges quantifier):

1. The graph has even number of nodes
2. The number of nodes in a graph is a prime number
3. The graph has the same number of red nodes and grey nodes

However, we can express the three properties by the following MSO formulas, respectively:

1. $\exists_{\mathsf{V}}\mathsf{X}(\forall_{\mathsf{v}}\mathsf{x}(\mathsf{x} \in \mathsf{X}) \wedge \exists_{\mathsf{l}}\mathsf{n}(\mathsf{card}(\mathsf{x}) = 2 * \mathsf{n}))$
2. $\exists_{\mathsf{V}}\mathsf{X}(\forall_{\mathsf{v}}\mathsf{x}(\mathsf{x} \in \mathsf{X}) \wedge \neg\exists_{\mathsf{l}}\mathsf{n}, \mathsf{m}(\mathsf{n} \neq 1 \wedge \mathsf{m} \neq 1 \wedge \mathsf{card}(\mathsf{x}) = \mathsf{n} * \mathsf{m}))$
3. $\exists_{\mathsf{V}}\mathsf{X}, \mathsf{Y}(\forall_{\mathsf{v}}\mathsf{x}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{red} \Leftrightarrow \mathsf{x} \in \mathsf{X}) \wedge \forall_{\mathsf{v}}\mathsf{x}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{grey} \Leftrightarrow \mathsf{x} \in \mathsf{Y}) \wedge \mathsf{card}(\mathsf{X}) = \mathsf{card}(\mathsf{Y}))$

With the existence of function card, our formula can express more properties if we compare it with counting MSO logic in [5] because we can compare cardinality between two sets with ours. However, what kind of properties can not be expressed by our formulas is still an open problem in this paper. Hence, the relative completeness of our MSO Hoare-triple is still unknown.

6 Case Study

In this section, we present the graph programs `is-connected` [3] and we verify the graph program with respect to the given specifications. Due to page limitation, we do not show the proof of implications in this paper. The proof can be found in [22] and other examples can be found in [22].

```
Main = try init then (DFS!; Check)
DFS = forward!; try back else break
Check = if match then fail
```

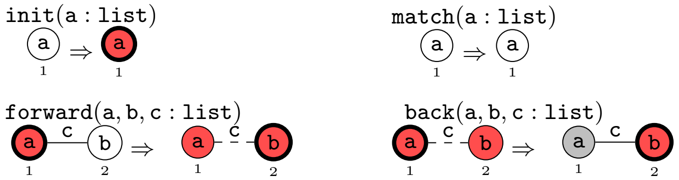


Fig. 5: Graph program `is-connected`

Here we consider the graph program `is-connected` as seen in Fig. 5. The program is executed by checking the existence of an unrooted node with no marks and change it to a red rooted node. The program then execute depth first-search procedure by finding unrooted node that is adjacent with the red rooted node and change the node to red, swap the rootedness, and mark the edge between them by dashed and repeat it as long as possible. The procedure continue by searching a red node that adjacent to red unrooted node by dashed edge and change the mark of the rooted node to grey while unmarking it, and move the root to the other node, then reply the procedure. Finally, the program checks if there still exists an unmarked node. If so, then the program yields fail.

For the specification, here we consider the case where the input graph is connected. For the case with disconnected graph, please see [22].

Precondition: *All nodes and edges are unmarked, and all nodes are unrooted. Also, the graph is connected, that is, for every nodes x, y, there exists an undirect path from x to y)*
Postcondition: *Either the graph is empty, or there is a node that is marked with red and is rooted while other nodes are grey and unrooted. All edges are unmarked, and the graph is connected.*

Now let us consider loops we have in the program `is-connected`. There are two loops: `forward!` and `DFS!`. For the former, we can consider #-function that count the number of unmarked nodes. By the application of the rule schema `forward`, the number of unmarked nodes obviously decreasing. Hence `forward` is #-decreasing. For `DFS!`, we can consider a #-function that count unmarked nodes and red nodes. From the initial graph, the application of `forward!` will not change the value of #, while `try` either will decrease the value of # by 1 or make us reach `break`. Hence, `DFS` is #-decreasing as well.

The total correctness proof for this case study is given by the proof tree of Fig. 6. We refer to [22] for the assertions in the proof tree, which we omit here because of the lack of space. For the same reason, we omit #-decreasing requirement in the premise of proof rule [alap].

From the proof tree we know the triple $\{pre\}$ `init` $\{c\}$ and $\{c\}$ `DFS`! $\{post\}$ are totally correct so that by the proof rule [comp] we can conclude that $\{pre\}$ `init`; `DFS`! $\{post\}$ is totally correct as well. Implication $post \Rightarrow \neg\text{Fail}(\texttt{match})$ must be true because the postcondition assert that there is no unmarked node. Hence, we can conclude that the execution of the program on a graph satisfying Precondition cannot fail and must resulting a graph satisfying Postcondition.

7 Conclusion

Poskitt and Plump [17] have defined a calculus to verify graph programs by using a so-called E-conditions [16] and M-conditions [19] as assertions. E-conditions are only able to express FO properties of GP 2 graph, while M-conditions can express properties of MSO properties of non-attributed graph (not all GP 2 graphs).

$$[\text{try}]\ \dfrac{\text{Subtree A} \qquad [\text{cons}]\ \dfrac{[\text{skip}]\ \dfrac{}{\{\ pre \wedge \text{Fail}(\texttt{init})\}\ \texttt{skip}\ \{pre \wedge \text{Fail}(\texttt{init})\}}}{\{\ pre \wedge \text{Fail}(\texttt{init})\}\ \texttt{skip}\ \{post\}}}{\{\ pre\ \}\ \texttt{try init then}\ (\texttt{DFS!};\texttt{Check})\ \{\ post\ \}}$$

where subtree A is:

$$[\text{cons}]\ \dfrac{\text{comp}\ \dfrac{[\text{cons}]\ \dfrac{[\text{ruleapp}]_{slp}\ \dfrac{}{\{\ pre\ \}\ \texttt{init}\ \{\ \text{Slp}(pre, \texttt{init})\ \}}}{\{\ pre\ \}\ \texttt{init}\ \{\ c\ \}} \qquad [\text{cons}]\ \dfrac{[\text{alap}]\ \dfrac{[\text{comp}]\ \dfrac{[\text{cons}]\ \dfrac{[\text{alap}]\ \dfrac{[\text{cons}]\ \dfrac{[\text{ruleapp}]_{slp}\ \dfrac{}{\{\ c\ \}\ \texttt{forward}\ \{\ \text{Slp}(c, \texttt{forward})\ \}}}{\{\ c\ \}\ \texttt{forward}\ \{\ c\ \}}}{\{\ c\ \}\ \texttt{forward!}\ \{\ c \wedge \text{Fail}(\texttt{forward})\ \}}}{\{\ c\ \}\ \texttt{forward!}\ \{\ d\ \}} \qquad \text{Subtree A1}}{\{\ c\ \}\ \texttt{DFS}\ \{\ c\ \}} \qquad \text{Break}(c, \texttt{DFS}, post)}{\{\ c\ \}\ \texttt{DFS!}\ \{\ (c \wedge \text{Fail}(\texttt{DFS})) \vee post\ \}}}{\{\ c\ \}\ \texttt{DFS!}\ \{\ post\ \}} \qquad \text{Subtree A2}}{\{\ pre\ \}\ \texttt{init};\texttt{DFS!};\texttt{Check}\ \{\ post\ \}}}{\{\ pre \wedge \text{Success}(\texttt{init})\ \}\ \texttt{init};\texttt{DFS!};\texttt{Check}\ \{\ post\ \}}$$

for Subtree A1:

$$[\text{try}]\ \dfrac{[\text{cons}]\ \dfrac{[\text{ruleapp}]_{slp}\ \dfrac{}{\{\ d \wedge \text{Success}(\texttt{back})\ \}\ \texttt{back}\ \{\ \text{Slp}(d \wedge \text{Success}(\texttt{back}), \texttt{back})\ \}}}{\{\ d \wedge \text{Success}(\texttt{back})\ \}\ \texttt{back}\ \{\ c\ \}} \qquad [\text{cons}]\ \dfrac{[\text{break}]\ \dfrac{}{\{\ d \wedge \text{Fail}(\texttt{back})\ \}\ \texttt{break}\ \{\ d \wedge \text{Fail}(\texttt{back})\ \}}}{\{\ d \wedge \text{Fail}(\texttt{back})\ \}\ \texttt{break}\ \{\ c\ \}}}{\{\ d\ \}\ \texttt{try back else break}\ \{\ c\ \}}$$

and Subtree A2:

$$[\text{if}]\ \dfrac{[\text{cons}]\ \dfrac{[\text{fail}]\ \dfrac{}{\{\ \textsf{false}\ \}\ \texttt{fail}\ \{\ \textsf{false}\ \}}}{\{\ post \wedge \text{Success}(\texttt{match})\ \}\ \texttt{fail}\ \{\ \text{post}\ \}} \qquad [\text{cons}]\ \dfrac{[\text{skip}]\ \dfrac{}{\{\ post \wedge \text{Fail}(\texttt{match})\ \}\ \texttt{skip}\ \{\ post \wedge \text{Fail}(\texttt{match})\ \}}}{\{\ post \wedge \text{Fail}(\texttt{match})\ \}\ \texttt{skip}\ \{\ post\ \}}}{\{\ post\ \}\ \texttt{if match then fail}\ \{\ post\ \}}$$

Fig. 6: Proof tree for `is-connected`

However, there are only limited graph programs that can be verified by the calculus (e.g. programs with no nested loop).

E-condition is an extension of nested graph conditions [7]. Pennemann [13] shows how to obtain a weakest liberal precondition (wlp) w.r.t a graph condition and a program and introduced a theorem prover to prove implication between a precondition and the obtained wlp. However, graph conditions also only able to express FO properties of a non-attributed graph. Habel and Radke [9] then introduced HR* conditions, which extend the graph conditions by introducing graph variables that represent graphs generated by hyperedge-replacement systems. Radke [20] showed that HR* conditions is somewhere between node-counting MSO graph formulas and SO graph formulas and showed how to construct a wlp w.r.t the conditions. However, theorem prover for this condition is not available yet, and we believe that having a wlp alone is not enough for program verifications.

In this paper, we have defined MSO formulas that can express local properties of GP 2 graphs, even properties that can not be expressed in counting MSO graph formulas [6]. By using the MSO formulas as assertions, we show that we can construct a strongest liberal postcondition (Slp) over a rule schema. Moreover, we also can use the construction to obtain Slp over a loop-free program, precondition Success(P) (or Fail(P)) that asserts the existence of a proper graph (or path to failure) in the execution of loop-free program P (or iteration command S). With this result, we can define a proof calculus to verify total correctness of graph programs with nested loops in certain forms.

As usual for Hoare calculi, our calculus does not cover implications between assertions. Currently, we have started to experiment of the use of SMT solver Z3 [1] to prove the implication.

References

1. N. Bjørner, L. de Moura, L. Nachmanson, and C. M. Wintersteiger. Programming Z3. In *SETSS 2018*, volume 11430 of *LNCS*, pages 148–201. Springer, 2018.
2. G. Campbell. Efficient graph rewriting. BSc thesis, Department of Computer Science, University of York, 2019. ArXiv e-print arXiv:2010.03993.
3. G. Campbell, B. Courtehoute, and D. Plump. Fast rule-based graph programs. *ArXiv e-prints*, arXiv:2012.11394, 2020. 47 pages.
4. B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.
5. B. Courcelle. Monadic second-order graph transductions. In *Proc. CAAP '92*, volume 581 of *LNCS*, pages 124–144. Springer, 1992.
6. B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach.* Cambridge University Press, 2012.
7. A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19:245–296, 2009.
8. A. Habel and D. Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 135–147. Springer-Verlag, 2002.

9. A. Habel and H. Radke. Expressiveness of graph conditions with variables. *Electronic Communications of the EASST*, 30, 2010.
10. T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer Academic Publishers, 2002.
11. T. Nipkow and G. Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.
12. C. Paulin-Mohring. Introduction to the Coq proof-assistant for practical software verification. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification*, volume 7682, pages 45–95. Springer, 2012.
13. K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.
14. D. Plump. The graph programming language GP. In *Proc. CAI 2009*, volume 5725 of *LNCS*, pages 99–122. Springer, 2009.
15. D. Plump. From imperative to rule-based graph programs. *Journal of Logic and Algebraic Methods in Programming*, 88:154–173, 2017.
16. C. M. Poskitt. *Verification of Graph Programs*. PhD thesis, The University of York, 2013.
17. C. M. Poskitt and D. Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
18. C. M. Poskitt and D. Plump. Verifying total correctness of graph programs. In *Graph Computation Models (GCM 2012), Revised Selected Papers*, volume 61 of *Electronic Communications of the EASST*, 2013.
19. C. M. Poskitt and D. Plump. Verifying monadic second-order properties of graph programs. In *Proc. ICGT 2014*, volume 8571 of *LNCS*, pages 33–48. Springer, 2014.
20. H. Radke. *A Theory of HR* Graph Conditions and their Application to Meta-Modeling*. PhD thesis, University of Oldenburg, Germany, 2016.
21. G. Wulandari and D. Plump. Verifying graph programs with first-order logic. In *Proc. GCM 2020*, volume 330 of *EPTCS*, pages 181–200, 2020.
22. G. S. Wulandari. Verification of graph programs with monadic second-order logic. Submitted PhD thesis, 2021.
23. G. S. Wulandari and D. Plump. Verifying graph programs with monadic second-order logic (extended version). Technical report, University of York, 2021. https://uoycs-plasma.github.io/GP2/publications.