Proving Correctness of Logically Decorated Graph Rewriting Systems*

Jon Haël Brenas[1], Rachid Echahed[2], and Martin Strecker[3]

1 **CNRS and Université Grenoble Alpes, Saint Martin d'Hères, France**
`Jon-Hael.Brenas@imag.fr`
2 **CNRS and Université Grenoble Alpes, Saint Martin d'Hères, France**
`Jon-Hael.Brenas@imag.fr`
3 **IRIT – Université de Toulouse, Toulouse, France**
`martin.strecker@iri.fr`

Abstract

We first introduce the notion of logically decorated rewriting systems where the left-hand sides are endowed with logical formulas which help to express positive as well as negative application conditions, in addition to classical pattern-matching. These systems are defined using graph structures and an extension of combinatory propositional dynamic logic, $\mathcal{CPDL}$, with restricted universal programs, called $\mathcal{C}2\mathcal{PDL}$. In a second step, we tackle the problem of proving the correctness of logically decorated graph rewriting systems by using a Hoare-like calculus. We introduce a notion of specification defined as a tuple (Pre, Post, R, S) with Pre and Post being formulas of $\mathcal{C}2\mathcal{PDL}$, R a rewriting system and S a rewriting strategy. We provide a sound calculus which infers proof obligations of the considered specifications and establish the decidability of the verification problem of the (partial) correctness of the considered specifications.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases Graph Rewriting, Hoare Logic,Combinatory PDL, Rewrite Strategies, Program Verification

Digital Object Identifier 10.4230/LIPIcs.FSCD.2016.14

1 Introduction

Rewriting techniques and particularly term rewriting systems have been very successful in different areas such as theorem proving or declarative programming languages. Term rewriting systems have a wide range of interesting results such as confluence analysis, termination orderings and even very powerful proof techniques based, in particular, on equational reasoning and structural induction. However, the structure of terms (trees) is not well suited to specify easily problem handling graph structures, unless one uses cumbersome encodings.

In this paper we will focus on a class of rewriting systems that manipulate graphs. Graphs are data structures that have become ubiquitous. In addition to discrete mathematics and computer science, they are also used to model data in various fields such as biology, geography, physics etc. The transformation of graphs is nowadays a domain of research in its own. One may distinguish two main streams, in the literature, for graph transformations : (i) the algorithmic approaches, which describe explicitly the algorithms involved in the application

* This work has been funded by the project CLIMT(ANR/(ANR-11-BS02-016).

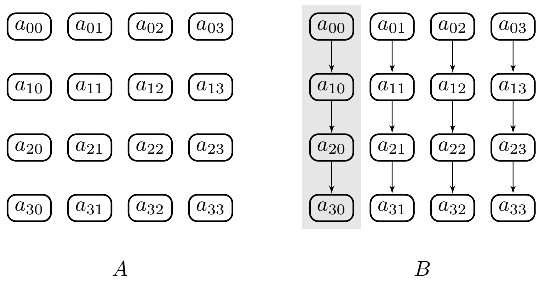

Figure 1 A – A Sudoku grid. B – An illustration of the correct definition of columns. We omit the labels of the edges.

of a rule to a graph, and (ii) the algebraic approaches which define abstractly a graph transformation step using basic constructs borrowed from category theory.

In this paper, we follow the algorithmic approach as proposed in [7] and consider rewrite rules of the form $lhs \to \alpha$ where lhs is a graph and α is a sequence of elementary actions that perform the desired transformations on the subgraph matched by lhs. We define the notion of *logically decorated graph rewriting systems* (LDRS) where the left-hand sides of rules are graphs attributed by formulas in a dynamic logic, called $\mathcal{C}2\mathcal{PDL}$[5], and whose models are also graphs. Such formulas, within the left-hand sides, could be seen as additional conditions to be fulfilled when matching a subgraph.

After the introduction of LDRS systems, we tackle the problem of their verification with the objective of building a decidable procedure. For that we define a Hoare-like calculus the aim of which is to prove that a transformation is correct, i.e., given a set of rewriting rules, a strategy stating how to apply them, a pre-condition indicating what are the properties to be satisfied before the application of the transformations and a post-condition stating which property is to be verified after the transformations, whether for any graph G satisfying the pre-condition every graph obtained by transforming G will satisfy the post-condition. To do so, we define a calculus that generates weakest-pre-conditions and verification conditions for each intermediate step of the strategy. This infers a weakest pre-condition and a verification condition for the whole transformations. That weakest pre-condition is then compared to the given pre-condition. We show that the proposed Hoare-like calculus is sound and that the considered correctness problem is decidable.

▶ **Example 1.** To clarify what is our aim and how our system works, we will be using a running example: we propose to study a simple program dealing with Sudoku grids. For reason of clarity and conciseness, we will study 4x4 grids instead of the normal 9x9. Nonetheless, the example can be easily extended to the normal Sudoku. An example of such a grid is shown in Figure 1.A. The goal is to fill each blank cell with a number between 1 and 4 such that the same number doesn't appear twice on a line, a column or a square. The goal of this example is not to show that graph transformations are efficient for solving Sudokus but just to provide a rather simple and common example in order to illustrate how to carry out the correctness proof of a program defined as a graph rewrite system.

The paper is organized as follows. In the following section, the dynamic logic $\mathcal{C}2\mathcal{PDL}$ used to express pre- and post-conditions is presented briefly. Then, the class of logically decorated rewriting systems is defined in Section 3 together with a notion of rewrite strategies. In Section 4, we start by setting the verification problem we consider and show how the proof obligations are generated. We also prove that the presented verification process is sound and decidable. An overview of related work is given in Section 5. Section 6 concludes the paper.

2 The dynamic logic $\mathcal{C}2\mathcal{PDL}$

In this section, we introduce $\mathcal{C}2\mathcal{PDL}$ [5], a logic that we use to specify assertions. It is a mix of Converse Propositional Dynamic Logic [8] and Combinatory Propositional Dynamic Logic [15], both commonly known as $\mathcal{CPDL}$. $\mathcal{C}2\mathcal{PDL}$ contains elements of Propositional Dynamic Logic, that allows one to define complex role constructors, and Hybrid Logic, which allows one to use the power of nominals. $\mathcal{C}2\mathcal{PDL}$ further extends the $\mathcal{CPDL}$s in two ways: it splits the universe into elements that are part of the model and elements that may be created by an action or that have been deleted; it also extends the notion of universal role to "total" roles over subsets of the universe in order to be able to deal with these modifications of the universe.

▶ **Definition 2** (Syntax of $\mathcal{C}2\mathcal{PDL}$). Given three countably infinite and pairwise disjoint alphabets Σ, the set of *names*, Φ_0, the set of *atomic propositions*, Π_0, the set of *atomic programs*, the language of $\mathcal{C}2\mathcal{PDL}$ is composed of *formulas* and *programs*[1]. We partition the set of names Σ into two countably infinite alphabets Σ_1 and Σ_2 such that $\Sigma_1 \cup \Sigma_2 = \Sigma$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$. Formulas ϕ and programs α are defined as:

$$\begin{array}{lll} \phi & := & i \mid \phi_0 \mid \neg\phi \mid \phi \vee \phi \mid \langle\alpha\rangle\phi \\ \alpha & := & \alpha_0 \mid \nu_S \mid \alpha;\alpha \mid \alpha \cup \alpha \mid \alpha^* \mid \alpha^- \mid \phi? \end{array}$$

where $i \in \Sigma$, $\phi_0 \in \Phi_0$, $\alpha_0 \in \Pi_0$ and $S \subseteq \Sigma$.

We denote by Π the set of programs and by Φ the set of formulas. As ususal, $\phi \wedge \psi$ stands for $\neg(\neg\phi \vee \neg\psi)$ and $[\alpha]\phi$ stands for $\neg(\langle\alpha\rangle\neg\phi)$.

For now, the splitting of Σ seems artificial. It is actually grounded in the use we want to make of the logic. Roughly speaking, Σ_1 stands for the names that are used in "the" current model whereas Σ_2 stands for the names that may be used in the future (or have been used in the past but do not participate in the current model).

▶ **Definition 3** (Model). A *model* is a tuple $\mathcal{M} = (M, R, \chi, V)$ where M is a set called the *universe*, $\chi : \Sigma \to M$ is a surjective mapping such that $\chi(\Sigma_1) \cap \chi(\Sigma_2) = \emptyset$, $R : \Pi \to \mathcal{P}(M^2)$ and $V : \Phi \to \mathcal{P}(M)$ are mappings such that:

- For each $\alpha_0 \in \Pi_0$, $R(\alpha_0) \in \mathcal{P}(\chi(\Sigma_1)^2)$
- $R(\nu_S) = \chi(S)^2$ for $S \subseteq \Sigma$
- $R(\alpha \cup \beta) = R(\alpha) \cup R(\beta)$
- $R(A?) = \{(s, s) | s \in V(A)\}$
- $R(\alpha^-) = \{(s, t) | (t, s) \in R(\alpha)\}$
- $R(\alpha^*) = \bigcup_{k<\omega} R(\alpha^k)$ where α^k stands for the sequence $\alpha; \ldots ; \alpha$ of length k
- $R(\alpha; \beta) = \{(s, t) | \exists v.((s, v) \in R(\alpha) \wedge (v, t) \in R(\beta))\}$
- For each $i \in \Sigma$, $V(i) = \{\chi(i)\}$
- For each $\phi_0 \in \Phi_0$, $V(\phi_0) \in \mathcal{P}(\chi(\Sigma_1))$
- $V(\neg A) = M \backslash V(A)$
- $V(A \vee B) = V(A) \cup V(B)$
- $V(\langle\alpha\rangle A) = \{s | \exists t \in M.((s, t) \in R(\alpha) \wedge t \in V(A))\}$

In the following, we write $sR_\alpha t$ for $(s, t) \in R(\alpha)$.

$\mathcal{C}2\mathcal{PDL}$ will be used in this paper to label nodes and to express properties of attributed graphs.

[1] This notion of *programs* is borrowed from PDL logic.

▶ **Definition 4** (Attributed Graph). An *attributed graph* G is a tuple $(N,E,\mathcal{C},\mathcal{R},L_N,L_E,s,t)$ where N is the set of *nodes*, E is the set of *edges*, $\mathcal{C}$ is the set of *node labels* or *concepts*, $\mathcal{R}$ is the set of *edge labels* or *roles*, L_N is the *node labeling* (total) function, $L_N : N \to \mathcal{P}(\mathcal{C})$, L_E is the *edge labeling* (total) function, $L_E : E \to \mathcal{R}$, s is the *source function* $s : E \to N$ and t is the *target function* $t : E \to N$.

From now on, we will only consider graphs such that $\mathcal{C} = \Phi$ and $\mathcal{R} = \Pi$. Given a graph $G = (N, E, \mathcal{C}, \mathcal{R}, L_N, L_E, s, t)$ and a formula ϕ, we say that $G \models \phi$ if there exists $n \in N$ such that $n \models \phi$ where the universe M is the set of nodes N. R and V are defined as usual: for $\phi_0 \in \Phi_0$ (resp. $\pi_0 \in \Pi_0$), $V(\phi_0) = \{x \in N | \phi_0 \in L_N(x)\}$ (resp. $R(\pi_0) = \{(x,y) \in N^2 | \exists e \in E.s(e) = x \wedge t(e) = y \wedge L_E(e) = \pi_0\}$). R and V are then extended to non-atomic propositions and programs following the same rules defined in the models. As usual, a formula ϕ is satisfiable if there exists a graph G such that $G \models \phi$ and unsatisfiable otherwise and it is valid if for all models G, $G \models \phi$ and invalid otherwise. We denote by S a subset of $\mathcal{C}$ which consists of names such that for each name $s \in S$ there is at most one node $n \in N$ such that $n \models s$. One may remark that all models can be considered as graphs. The converse is false.

Often, we will write $i : C$ instead of $i : \{C\}$ to say that node i is labelled with the formula C. In Figure 2, we give an example of models depicted as graphs.

Attributed graphs where all nodes are named will be called *named graphs*. This notion of graphs will be used in the proof section.

▶ **Definition 5** (Named Graph). A *named graph* G is an attributed graph such that the set of names $S \subseteq C$ satisfies:

(a) $\forall s \in S.\ \exists n \in N.\ s \in L_N(n)$,
(b) $\forall n \in N.\ L_N(n) \cap S \neq \emptyset$,
(c) $\forall n, n' \in N, n \neq n',\ L_N(n) \cap L_N(n') \cap S = \emptyset$.

Notation: From (a), (b) and (c), it is obvious that, as each name labels at least one node, each node is labeled by at least one name and each name labels at most one node, it is possible to define two functions θ and μ such that $\forall s \in S, \theta(s)$ is the node named s and $\forall n \in N, \mu(n)$ is a name of n. This allows to define the surjective mapping of models χ and thus named graphs and models are equivalent structures. We will thus consider from now on that formulae are interpreted over named graphs.

▶ **Example 6.** Going back to the Sudoku example, we will use $\mathcal{C2PDL}$ to state some properties. We are going to use a name for each cell of the grid (a_{ij} with $0 \leq i, j \leq 3$ where i is the row and j the column in which the cell can be found). We will also use three atomic programs R (resp. C and SQ) to state which cells are on the same row (resp. column and square). Finally, we will need eight atomic propositions 1 (resp. 2,3 and 4) and P_1 (resp. P_2,P_3 and P_4) to state that a cell *is known to* contain 1 (resp. 2, 3 or 4) and that a cell *may* contain 1 (resp. 2, 3 or 4). Thus we require that $\{a_{ij} | i, j \in [0,3]\} \subset \Sigma$, $\{i | i \in [1,4]\} \cup \{P_i | i \in [1,4]\} \subset \Phi_0$ and $\{R, C, SQ\} \subset \Pi_0$. As we do not create or delete nodes, we do not make use of the possibility to change the set of definition of the total program. For this example, ν stands for ν_{Σ_1}. We define below a few relevant formulae.

- The atomic program C should describe columns, as shown in Figure 1.B:
  $c_j = \langle\nu\rangle(a_{0j}\ \wedge \langle C\rangle(a_{1j}\ \wedge \langle C\rangle(a_{2j}\ \wedge \langle C\rangle a_{3j})))$ for $j \in [0,3]$. c_j describes the successive elements of a column.
  $\bar{c}_j = \langle\nu\rangle(a_{0j}\ \wedge [C](a_{1j}\ \wedge [C](a_{2j}\ \wedge [C](a_{3j} \wedge [C]\bot))))$ for $j \in [0,3]$. $\bar{c}_j$ says that there are no more elements in a column than those specified by c_j. Thus a column is specified by $c_j \wedge \bar{c}_j$.

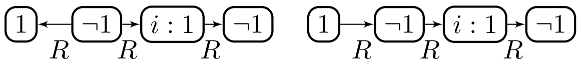

Figure 2 Model and counter-model. All nodes of the left graph satisfy the formula $[\nu](1 \Rightarrow [R^-][R^{-*}]\neg 1)$. This is not the case for the graph given on the right since $i \not\models (1 \Rightarrow [R^-][R^{-*}]\neg 1)$.

- A cell should contain, at most, one value: $u_{I,J} = [\nu](I \Rightarrow \neg J)$ for $I, J \in [1,4]$, $I \neq J$ and there is no doubt about it (e.g., once a cell is assigned the value I, it is no longer a candidate for any future potential assignement P_J): $\overline{u}_{I,J} = [\nu](I \Rightarrow \neg P_J)$ for $I, J \in [1,4]$
- If a cell has a value J, J cannot be the value of any other cell on the same row, column or square $v_{r,J} = [\nu](J \Rightarrow (([r][r^*]\neg J) \wedge ([r^-][r^{-*}]\neg J)))$ for $J \in [1,4]$ and $r \in \{R, C, SQ\}$. Figure 2 shows an example of a model and a counter-model of part of this expression.

▶ **Theorem 7.** *Given a formula ϕ of $\mathcal{C2PDL}$, the satisfiability and the validity of ϕ are decidable.*

More on $\mathcal{C2PDL}$ including the proof of this theorem can be found in [5].

3 Logically Decorated Rewriting Systems LDRS

In this section we introduce the notion of *logically decorated rewriting systems*, LDRS. These are extensions of graph rewriting systems defined in [7] where graphs are attributed with $\mathcal{C2PDL}$ formulas. The left-hand sides of the rules are thus logically decorated graphs whereas the right-hand sides are defined as sequences of elementary actions. These actions constitute a set of elementary transformations used in graph transformation processes. The operational way the right-hand sides are defined in this paper departs from those classically used in algebraic approaches [18] such as simple pushout or double pushout where rules are defined by means of graph morphisms.

▶ **Definition 8** (Elementary Action, Action)**.** An *elementary action*, say a, has one of the following forms:

- a *concept addition* $add_C(i, c)$ (resp. *concept deletion* $del_C(i, c)$) where i is a node and c is a basic concept (a proposition name) in Φ_0. It adds the node i to (resp. removes the node i from) the valuation of the concept c.
- a *role addition* $add_R(i, j, r)$ (resp. *role deletion* $del_R(i, j, r)$) where i and j are nodes and r is an atomic role (edge label) in Π_0. It adds the pair (i, j) to (resp. removes the pair (i, j) from) the valuation of the role r.
- a *node addition* $add_I(i)$ (resp. *node deletion* $del_I(i)$) where i is a new node (resp. an existing node). It creates the node i. i has no incoming nor outgoing edge and there is no basic concept (in Φ_0) such that i belongs to its valuation (resp. it deletes i and all its incoming and outgoing edges).
- a *global edge redirection* $i \gg j$ where i and j are nodes. It redirects all incoming edges of i toward j.

The result of performing the elementary action α on a graph $G = (N^G, E^G, C^G, R^G, L_N^G, L_E^G, s^G, t^G)$, written $G[\alpha]$, produces the graph $G' = (N^{G'}, E^{G'}, C^{G'}, R^{G'}, L_N^{G'}, L_E^{G'}, s^{G'}, t^{G'})$ as defined in Figure 3. An *action*, say α, is a sequence of elementary actions of the form $\alpha = a_1; a_2; \ldots; a_n$. The result of performing α on a graph G is written $G[\alpha]$. $G[a; \alpha] = (G[a])[\alpha]$ and $G[\epsilon] = G$, ϵ being the empty sequence.

If $\alpha = add_C(i,c)$ **then:**
$N^{G'} = N^G, E^{G'} = E^G, C^{G'} = C^G, R^{G'} = R^G, L_E^{G'} = L_E^G$
$$L_N^{G'}(n) = \begin{cases} L_N^G(n) \cup c & \text{if } n = i \\ L_N^G(n) & \text{if } n \neq i \end{cases}$$
$s^{G'} = s^G$, $t^{G'} = t^G$

If $\alpha = add_R(i,j,r)$ **then:**
$N^{G'} = N^G$, $C^{G'} = C^G, R^{G'} = R^G$, $L_N^{G'} = L_N^G$
$E^{G'} = E^G \cup e$ where e is a new element
$$L_E^{G'}(e') = \begin{cases} r & \text{if } e' = e \\ L_E^G(e') & \text{if } e' \neq e \end{cases}$$
$$s^{G'}(e') = \begin{cases} i & \text{if } e' = e \\ s^G(e') & \text{if } e' \neq e \end{cases}$$
$$t^{G'}(e') = \begin{cases} j & \text{if } e' = e \\ t^G(e') & \text{if } e' \neq e \end{cases}$$

If $\alpha = add_I(i)$ **then:**
$N^{G'} = N^G \cup i$ where i is a new node
$C^{G'} = C^G$, $R^{G'} = R^G$
$$L_N^{G'}(n') = \begin{cases} \emptyset & \text{if } n' = i \\ L_N^G(n') & \text{if } n' \neq i \end{cases}$$
$E^{G'} = E^G$, $L_E^{G'} = L_E^G, s^{G'} = s^G$, $t^{G'} = t^G$

If $\alpha = i \gg j$ **then:**
$N^{G'} = N^G$, $E^{G'} = E^G$, $C^{G'} = C^G$
$R^{G'} = R^G$, $L_N^{G'} = L_N^G$, $L_E^{G'} = L_E^G, s^{G'} = s^G$
$$t^{G'}(e) = \begin{cases} j & \text{if } t^G(e) = i \\ t^G(e) & \text{if } t^G(e) \neq i \end{cases}$$

If $\alpha = del_C(i,c)$ **then:**
$N^{G'} = N^G, E^{G'} = E^G, C^{G'} = C^G, R^{G'} = R^G$, $L_E^{G'} = L_E^G$
$$L_N^{G'}(n) = \begin{cases} L_N^G(n) \backslash c & \text{if } n = i \\ L_N^G(n) & \text{if } n \neq i \end{cases}$$
$s^{G'} = s^G$, $t^{G'} = t^G$

If $\alpha = del_R(i,j,r)$ **then:**
$N^{G'} = N^G$, $C^{G'} = C^G, R^{G'} = R^G$, $L_N^{G'} = L_N^G$
$E^{G'} = E^G \backslash \{e | s^G(e) = i \wedge t^G(e) = j \wedge L_E^G(e) = r\}$
$L_E^{G'}$ is the restriction of L_E^G to $E^{G'}$
$s^{G'}$ is the restriction of s^G to $E^{G'}$
$t^{G'}$ is the restriction of L_E^G to $E^{G'}$

If $\alpha = del_I(i)$ **then:**
$E^{G'} = E^G \backslash \{e | s^G(e) = i \vee t^G(e) = i\}$
$N^{G'} = N^G \backslash i$, $C^{G'} = C^G, R^{G'} = R^G$
$L_N^{G'}$ is the restriction of L_N^G to $N^{G'}$
$L_E^{G'}$ is the restriction of L_E^G to $E^{G'}$
$s^{G'}$ is the restriction of s^G to $E^{G'}$
$t^{G'}$ is the restriction of L_E^G to $E^{G'}$

Figure 3 Summary of the effects of elementary actions.

▶ **Definition 9** (Rule, LDRS). A *rule* ρ is a pair (lhs, α) where lhs, called the left-hand side, is an attributed graph with $\mathcal{C}2\mathcal{PDL}$ formulae as attributes and α, called the right-hand side, is an action. Rules are usually written $lhs \rightarrow \alpha$. A logically decorated rewriting system, LDRS, is a set of rules.

It is noteworthy that the left-hand side of a rule is an attributed graph, that is it can contain nodes labeled with $\mathcal{C}2\mathcal{PDL}$ formulae. This is not insignificant. Indeed, these formulae can express reachability expression (closure of a program), non-local properties (universal program), ... These node labelings allow to write more concise and simpler rewriting systems. For instance, Figure 4 provides two LDRSs, GRS_0 and GRS_1 which remove unreachable nodes from a start state (labeled by S) of an automaton. Without the closure constructor $(*)$, one would tag that the start states are reachable (label R) (rule ρ_{00}), then say that every neighbor of a reachable node is reachable (rule ρ_{01}) and finally that all nodes that have not been reached by applying the first two rules as much as possible are to be removed (rule ρ_{02}). GRS_0 requires the explicit computation of the reachability making the algorithm more complex. On the other hand, GRS_1 only uses one rule which says that all nodes that are not reachable from a start state are to be removed (rule ρ_1).

It is worth noting that the impact of $\mathcal{C}2\mathcal{PDL}$ formulae in node labels is not limited to graph rewriting. Indeed, let us consider the term rewriting system of integer arithmetic with multiplication. The classical way to deal with 0s in such a case is to have rules saying that $0 \times x \leadsto 0$ and $x \times 0 \leadsto 0$. Considering terms as trees, and thus as graphs, it is also possible to improve on this set of rules by using, for instance, the rule $i : \times \wedge \langle((L \cup R); \times?)^*\rangle 0 \leadsto i : 0$. This rule states that if a node i is such that $i : \times \wedge \langle((L \cup R); \times?)^*\rangle 0$, that is to say, node i is labeled by the multiplication operator $(i : \times)$ and there is a path of left- or right-operands $(L \cup R)$, crossing nodes labeled by the multiplication operator $(\times?)$, that leads to a node labeled by 0 $(i : \langle((L \cup R); \times?)^*\rangle 0)$, then node i could be labeled by 0 $(i : 0)$.

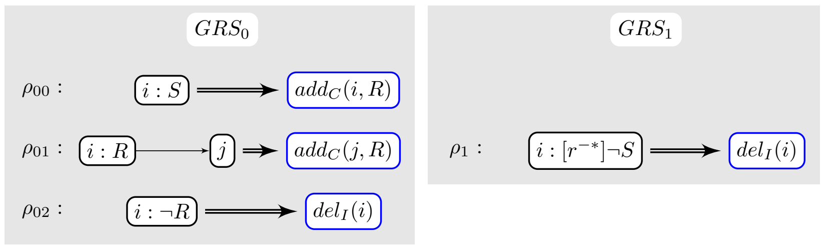

Figure 4 Two LDRS dealing with the suppression of unreachable nodes. Rules with atomic formulae are on the left. The rule with a non-atomic formula is on the right.

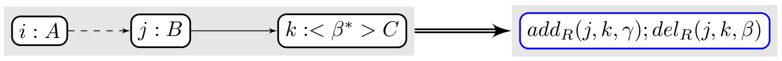

Figure 5 An example of LDRS. The dashed line represents the program (label) α and the plain line the program β. The first edge labeled β after an access α on a path toward a C gets a new tag γ.

In Figure 5, we provide an additional toy example which is used later. It consists of one rule which relabels the edge going from node j to k with label (program) γ whenever node k has access to "information" C. This rule may have different interpretations such as the modification of access policy to information tagged C.

▶ **Example 10.** Back to our running example, we provide a very simple graph rewriting system $\mathcal{R}$ that tries to produce a full and correct grid. It contains 16 rules, that are summarized in Figure 6. The rules $\rho_{r,J}$ make sure that when a line (resp. a column or a square) contains a cell with value J, P_J is no longer available for all the cells on the line (resp. column or square). The rules ρ_J are used to pick one choice among those that are available.

▶ **Definition 11** (Match). A *match* h between a left-hand side lhs and a graph G is a pair of functions $h = (h_N, h_E)$, with $h_N : N_{lhs} \to N_G$ and $h_E : E_{lhs} \to E_G$ such that:

1. $\forall n \in N_{lhs}, \forall c \in L_{N_{lhs}}(n), h_N(n) \models c$
2. $\forall e \in E_{lhs}, L_{E_{lhs}}(e) = L_{E_G}(h_E(e))$
3. $\forall e \in E_{lhs}, s_G(h_E(e)) = h_N(s_{lhs}(e))$
4. $\forall e \in E_{lhs}, t_G(h_E(e)) = h_N(t_{lhs}(e))$

The third and the fourth conditions are classical and say that the source and target functions and the match have to agree. The first condition says that for every node n of the left-hand side, the node to which it is associated, $h(n)$, in G has to satisfy every concept that n satisfies. This condition clearly expresses additional negative and positive conditions which are added to the "structural" pattern matching. The second one ensures that the match respects edge labeling.

▶ **Definition 12** (Rule Application). A graph G rewrites to graph G' using a rule $\rho = (lhs, \alpha)$ iff there exists a match h from lhs to G. G' is obtained from G by performing actions in $h(\alpha)^2$. Formally, $G' = G[h(\alpha)]$. We write $G \to_\rho G'$ or $G \to_{\rho,h} G'$.

Confluence of graph rewriting systems is not easy to establish. For instance, orthogonal graph rewrite systems are not always confluent, see e.g.,[7]. That is why we use a notion rewrite strategies to control the use of possible rules. Informally, a strategy specifies the

2 $h(\alpha)$ is obtained from α by replacing every node name,n, of lhs by $h(n)$.

$$\rho_{r,J}: \boxed{i : P_J \wedge \langle (r \cup r^-)^* \rangle J} \Longrightarrow \boxed{del_C(i, P_J)}$$

$$\rho_J: \boxed{i : P_J} \Longrightarrow \boxed{add_C(i, J); del_C(i, P_1); del_C(i, P_2); del_C(i, P_3); del_C(i, P_4)}$$

Figure 6 A summary of the rules of $\mathcal{R}$. In $\rho_{r,J}$, r must be replaced by either R, C or SQ and J by 1, 2, 3 or 4. In ρ_J, J must be replaced by 1, 2, 3 or 4.

$$\frac{G \to_\rho G'}{G \Rightarrow_\rho G'} \text{ (Rule)} \qquad \frac{G \Rightarrow_{s_0} G'' \quad G'' \Rightarrow_{s_1} G'}{G \Rightarrow_{s_0;s_1} G'} \text{ (Strategy composition)}$$

$$\frac{G \Rightarrow_{s_0} G'}{G \Rightarrow_{s_0 \oplus s_1} G'} \text{ (Choice left)} \qquad \frac{G \Rightarrow_{s_1} G'}{G \Rightarrow_{s_0 \oplus s_1} G'} \text{ (Choice right)}$$

$$\frac{\neg \mathbf{App}(s, G)}{G \Rightarrow_{s^*} G} \text{ (Closure false)} \qquad \frac{G \Rightarrow_s G'' \quad G'' \Rightarrow_{s^*} G' \quad \mathbf{App}(s, G)}{G \Rightarrow_{s^*} G'} \text{ (Closure true)}$$

Figure 7 Strategy application rules.

application order of different rules. It does not point to where the matches are to be found nor does it ensure the unicity of the reduction outcome.

▶ **Definition 13** (Strategy). Given a graph rewriting system $\mathcal{R}$, a *strategy* is a non-empty word of the following language defined by s:

$s := \quad \rho \quad \text{(Rule)} \mid s; s \quad \text{(Composition)} \mid s \oplus s \quad \text{(Choice)} \mid s^* \quad \text{(Closure)}$

where ρ is any rule in $\mathcal{R}$.

We write $G \Rightarrow_{\mathcal{S}} G'$ when G rewrites to G' following the rules given by the strategy $\mathcal{S}$.

Informally, the strategy "$\rho_1; \rho_2$" means that rule ρ_1 should be applied first, followed by the application of rule ρ_2. The strategy "$\rho_0^*; (\rho_1 \oplus \rho_2)$" means that rule ρ_0 is applied as far as possible, then followed either by ρ_1 or ρ_2. It is worth noting that the closure is the standard "while" construct: if the strategy we use is s^*, the strategy s is used as long as it is possible and not an undefined number of times.

▶ **Example 14.** For the Sudoku example, a possible strategy $\mathcal{S}_1$ could be: "As long as one can eliminate possibilities, do it. Then, when it is no longer the case, make a choice in one of the blank cells and go back to the first step". $\mathcal{S}_1$ may be defined as $\mathcal{S}_1 = (\bigoplus_{r \in \{R,C,SQ\}, J \in [1,4]} \rho_{r,J})^*; ((\bigoplus_{J \in [1,4]} \rho_J); (\bigoplus_{r \in \{R,C,SQ\}, J \in [1,4]} \rho_{r,J})^*)^*$.

In Figure 7, we provide the rules that specify how strategies are used to rewrite a graph. For that, we use the predicate $\mathbf{App}(s, G)$ which holds whenever graph G can be rewritten by the strategy s. It is defined as follows:

$\mathbf{App}(\rho, G) = true$ iff there exists a match h from the left-hand side of ρ to G

$\mathbf{App}(s_0 \oplus s_1, G) = \mathbf{App}(s_0, G) \vee \mathbf{App}(s_1, G)$

$\mathbf{App}(s_0^*, G) = true$

$\mathbf{App}(s_0; s_1, G) = \mathbf{App}(s_0, G)$

It is worth noting that $\mathbf{App}(s, G)$ is not meant to denote that the whole strategy can be applied to G, just that the next step can be applied. Indeed, let's assume the strategy $s = s_0; s_1$ where s_0 can be applied but may lead to a state where s_1 cannot. In this case $\mathbf{App}(s, G)$ will hold saying that the strategy s can be applied on G.

$$
\begin{array}{ll}
wp(add_C(i,c),\ Q) = Q[add_C(i,c)] & wp(del_C(i,c),\ Q) = Q[del_C(i,c)] \\
wp(add_R(i,j,r),\ Q) = Q[add_R(i,j,r)] & wp(del_R(i,j,r),\ Q) = Q[del_R(i,j,r)] \\
wp(add_I(i),\ Q) = Q[add_I(i)] & \bullet\ wp(del_I(i),\ Q) = Q[del_I(i)] \\
wp(i >> j,\ Q) = Q[i >> j] & wp(\epsilon,\ Q) = Q \\
wp(a;\alpha,\ Q) = wp(a, wp(\alpha, Q)) &
\end{array}
$$

Figure 8 Weakest pre-conditions for actions.

$$
\begin{array}{ll}
wp(s_0; s_1,\ Q) = wp(s_0, wp(s_1,\ Q)) & wp(s_0 \oplus s_1,\ Q) = wp(s_0, Q) \wedge wp(s_1, Q) \\
wp(s^*,\ Q) = inv_s & wp(\rho,\ Q) = App(tag(\rho)) \Rightarrow wp(tag(\alpha_\rho), Q)
\end{array}
$$

Figure 9 Weakest pre-conditions for strategies.

4 Proving Correctness of LDRS's

Equational reasoning and structural induction method represent the main core of proof techniques dedicated to reason about term rewrite systems. Unfortunately, graph rewrite systems do not benefit yet from such established techniques. For example, generalization of equational reasoning to graph rewriting systems is not complete [6]. In this section, we propose a way to specify properties of LDRSs for which we establish a decidable proof procedure.

▶ **Definition 15** (Specification). A *specification* SP is a tuple $(Pre,\ Post,\ \mathcal{R},\ \mathcal{S})$ where Pre and $Post$ are $\mathcal{C2PDL}$ formulas, $\mathcal{R}$ is a graph rewriting system and $\mathcal{S}$ is a strategy.

A specification SP is said to be *correct* iff for all graphs G, G' such that $G \Rightarrow_{\mathcal{S}} G'$ and $G \models Pre$, then $G' \models Post$.

In order to show the correctness of a specification, we follow a Hoare-calculus style and compute the weakest pre-condition $wp(\mathcal{S}, Post)$. For that, we define the weakest pre-conditions of a formula $Post$ induced by a strategy, a rule, an action and an elementary action. They are presented in Figure 8 for the elementary actions and Figure 9 for strategies.

The weakest pre-condition of an elementary action, say a, and a post-condition Q is defined as $wp(a, Q) = Q[a]$ where $Q[a]$ stands for the pre-condition consisting of Q to which is applied a substitution induced by the action a that we denote by $[a]$. The notion of substitution used here is the one of Hoare-calculi [11].

▶ **Definition 16** (Substitutions). To each elementary action a is associated a *substitution*, written $[a]$, such that for any graph G, $(G \models \phi[a]) \Leftrightarrow (G[a] \models \phi)$.

It is worth noting that substitutions are, in all generality, not defined as formulae of $\mathcal{C2PDL}$. They are defined as a new constructor whose meaning is that the weakest pre-conditions as defined above are correct. There is no reason whatsoever to think that the addition of a constructor for substitutions is harmless, in general. It is a very interesting problem to figure out for which logics they can be introduced as it is a strong indication that such a logic may be suitable to study the correction of programs. It is one of the central results of [5] that $\mathcal{C2PDL}$ is closed under substitutions allowing us to use them freely in the following.

$$tag(\rho_J): \boxed{i:\{i_a, P_J\}} \Longrightarrow \boxed{add_C(i_a, J); del_C(i_a, P_1); del_C(i_a, P_2); del_C(i_a, P_3); del_C(i_a, P_4)}$$

Figure 10 The rule ρ_J is modified into $tag(\rho_J)$ with $tag(i) = i_a$.

Apart from $wp(\rho,\ Q)$, the weakest pre-conditions defined in here are the usual ones. As in any other framework using Hoare logic, it requires the definition of an invariant(inv) for each loop that has to be provided by the user.

The weakest pre-condition of a rule $\rho = (lhs_\rho, \alpha_\rho)$ and a post-condition Q is given by $wp(\rho,\ Q) = App(tag(\rho)) \Rightarrow wp(tag(\alpha_\rho), Q)$ where $tag : N_{lhs_\rho} \to \Sigma$ is a function which associates to every node, n, of the left-hand side of rule ρ, a fresh name in Σ. These new names are used to keep track of the matching locations within potential graphs rewritten by different instances of ρ during the execution of a strategy. $tag(\rho) = (tag(lhs_\rho), tag(\alpha_\rho))$ where $tag(lhs_\rho)$ is a named graph which consists of the graph lhs_ρ where the node labeling function is augmented by tag, i.e., for all nodes, n, of lhs_ρ, $L_{N_{tag(lhs_\rho)}}(n) = L_{N_{lhs_\rho}}(n) \cup tag(n)$. $tag(\alpha_\rho)$ is obtained from α_ρ by substituting every node (in N_{lhs_ρ}), say i, by $tag(i)$. Figure 10 gives an example turning the left-hand side of a rule into a named graph via a function tag.

The formula $App(tag(\rho))$ expresses the applicability of the rule $tag(\rho)$. In other words, it expresses the existence of a match of the left-hand side of $tag(\rho)$. More precisely $App(tag(\rho))$ is defined as follows:

$$App(tag(\rho)) = \phi_{nodes} \wedge \phi_{edges}$$

where

$$\phi_{nodes} = \bigwedge_{u \in tag(N_{lhs_\rho})} \langle \nu_{\Sigma_1} \rangle (u \wedge \bigwedge_{\phi \in L_N^{\flat(lhs_\rho)}(\theta(u))} \phi)$$

and

$$\phi_{edges} = \bigwedge_{e \in E | s(e) = \theta(u)} \langle \nu_{\Sigma_1} \rangle (u \wedge \langle L_E^{\flat(lhs_\rho)}(e) \rangle tag(t(e)) \wedge$$
$$\bigwedge_{e \in E | t(e) = \theta(u)} \langle \nu_{\Sigma_1} \rangle (u \wedge \langle L_E^{\flat(lhs_\rho)}(e)^- \rangle tag(s(e))\,.$$

ϕ_{nodes} states that the first condition of Definition 11 is satisfied i.e., all formulae satisfied by a node of the left-hand side have to be satisfied by its image in the graph. ϕ_{edges} does the same with edges: the label of the corresponding edges are the same and the source and target functions fulfill the matching compatibly condition.

Tagging a rule may seem to reduce its applicability. Indeed, by choosing a new name for each node of the left-hand side, the rule can now be applied only at the nodes of the graph named accordingly. Let ρ be a rule such that $LHS_\rho = (N_\rho = \{i_0, \ldots i_n\}, E_\rho, \mathcal{C}_\rho, \mathcal{R}_\rho, L_{N_\rho}, L_{E_\rho}, s_\rho, t_\rho)$ is its left-hand side. In order to prove that the application of ρ on a graph $G = (N_G, E_G, \mathcal{C}_G, \mathcal{R}_G, L_{N_G}, L_{E_G}, s_G, t_G)$ is correct, one has to verify that for every match $h = (h_N, h_E)$ (as defined in Definition 11), the post-condition is satisfied after the transformation associated with ρ is applied at the $h_N(i_k)$'s. Instead of showing that, the verification procedure proves that for any graph, if the rule can be applied at the $\theta(u)$'s, where θ is the function that associates to each name of Σ a node of G and $u \in tag(\rho)$, the post-condition is satisfied after performing the transformation. As the u's are fresh names, they do not have any impact on the previous characterization of the graphs. Thus, the validity of

$$
\begin{array}{ll}
vc(\rho,\ Q) = \top & vc(s_0; s_1,\ Q) = vc(s_0, wp(s_1,\ Q)) \wedge vc(s_1, Q) \\
vc(s_0 \oplus s_1,\ Q) = vc(s_0, Q) \wedge vc(s_1, Q) & \\
vc(s^*,\ Q) = (inv_s \wedge NApp(s) \Rightarrow Q) & \wedge\ (inv_s \Rightarrow wp(s, inv_s)) \wedge vc(s, inv_s)
\end{array}
$$

Figure 11 Verification conditions.

$wp(\rho, Post)$ actually states that whatever the choice of θ is, if the rule can be applied, then the post-condition will be satisfied after it is fired.

The weakest pre-condition associated to the closure of a strategy, say s^*, and a post-condition Q is defined as $wp(s^*,\ Q) = inv_s$ where inv_s is an invariant ($\mathcal{C}2\mathcal{PDL}$ formula) associated to the strategy s. Roughly speaking s^* could be seen as a while-loop which needs invariants associated with additional proof obligations defined by means of the function vc (verification conditions) given in Figure 11.

Informally, the predicate *NApp(s)*, used in the definition of vc, the verification conditions function, says that the strategy s cannot be applied. To express *NApp(s)*, one may wonder whether it is possible to use the negation of $App(s)$. The answer is negative since *App* has been defined using dedicated names, that is to say a rule is applied at a specific place defined by the added names introduced by the function *tag*. Intuitively, if one wants to express that there is no match for a rule whose left-hand side contains a cycle (e.g., the second graph of Figure 12), then universal quantification is required. For instance, in that case it would be $\forall i, j.[\nu_{\Sigma 1}](i \Rightarrow [R](j \Rightarrow [R]\neg i))$. One of them can be discarded to produce the expression $\forall i.[\nu_{\Sigma 1}](i \Rightarrow [R][R]\neg i)$. Alas, this cannot be expressed in $\mathcal{C}2\mathcal{PDL}$. The names only allow to express existential quantifiers. Indeed, to express universal quantifiers, one needs to extend the logic with the binder $\downarrow$ of hybrid logic which is enough to make the logic undecidable[2].

To express formally *NApp(s)* in $\mathcal{C}2\mathcal{PDL}$, one needs to introduce some additional definitions first.

▶ **Definition 17** (Explicitly Named Nodes, Non-Oriented Cycles)**.** An *explicitly named node* is a node such that each disjunct of the disjunctive normal form of its label contains a name. A *non-oriented cycle*, c, is a finite list of nodes $c = [n_0, \dots, n_k]$ such that (i) $n_k = n_0$ and (ii) $\forall \kappa \in [0, k-1], \exists r \in \Pi_0$ such that $(n_\kappa, n_{\kappa+1}) \in R(r)$ or $(n_{\kappa+1}, n_\kappa) \in R(r)$.

▶ **Definition 18** (Grove and Thicket)**.** A *grove* is a disjoint union of thickets. A *thicket* is a connected graph such that it does not contain any non-oriented cycle composed only of non explicitely named nodes. We call a strategy $\mathcal{S}$ relative to a rewriting system $\mathcal{R}$ a *grove-strategy* if the left-hand sides of all the rules appearing under a closure are groves.

Let lhs be a left-hand side. Let us split N_{lhs} into T_E, the set of explicitly named nodes, and $T_I = N_{lhs} \backslash T_E$. For each maximally connected subgraph composed only of nodes in T_I, a distinguished node r_i is selected. If there is a maximally connected subgraph composed only of explicitly named nodes, an r_i is also picked for it. Now, everything is ready to define *NApp(s)*.

1. $NApp(\rho) = \bigvee_{r_i} [\nu_{\Sigma_1}] NA(r_i, \emptyset)$
2. $NA(n, V) = (\bigvee_{\phi \in L_N(n)} \neg\phi) \vee (\bigvee_{e \in E | s(e) = n | s(e) \in T_E \cup (T_I \backslash V)} [L_E(e)] NA(t(e), V \cup \{n\})) \vee (\bigvee_{e \in E | t(e) = n | t(e) \in T_E \cup (T_I \backslash V)} [L_E(e)^-] NA(s(e), V \cup \{n\}))$ if $n \notin V$
3. $NA(n, V) = \neg\mu(n)$ if $n \in T_E \cap V$
4. $NApp(s_0 \oplus s_1) = NApp(s_0) \wedge NApp(s_1)$
5. $NApp(s^*) = false$
6. $NApp(s_0; s_1) = NApp(s_0)$

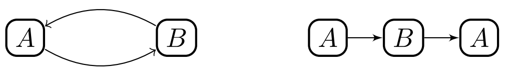

Figure 12 These two graphs are indistinguishable without names.

Rule 2 is the most involved. $NA(n, V)$ is used to describe what has to be true for a node different from n. V is used to track which nodes have already been visited. $\bigvee_{\phi \in L_N(n)} \neg\phi$ states that there is at least one of the formulae satisfied by n that is not satisfied while $\bigvee_{e\in E|s(e)=n|s(e)\in T_E \cup (T_I \setminus V)}[L_E(e)]NA(t(e), V \cup \{n\})$ means that there is a neighbor of n using an outgoing edge that cannot find a match. $\bigvee_{e\in E|t(e)=n|t(e)\in T_E \cup (T_I \setminus V)}[L_E(e)^-]NA(s(e), V \cup \{n\})$ has the same signification but for incoming edges. Rule 3 is used to recall the names of the explicitly named nodes that have already been visited without creating a cycle in the execution. Rules 4 to 6 are exactly the negation of $App(s, G)$. Rule 1 says that for at least one r_i, it is not possible to find a node that would be a match. It is noteworthy that there is no rule for $N \in T_I \cap V$ since the considered left-hand side has to be a grove.

▶ **Example 19.** The rules of the Sudoku example are simple, having only one-node left-hand side, and thus not very interesting as far as $NApp$ is concerned. We will thus consider the rule, say ρ, of Figure 5. There is only one connected subgraph so one has to pick one distinguished node r_0. Let us choose, randomly, the node j as r_0. Then $NApp(\rho) = [\nu_{\Sigma_1}]NA(j, \emptyset)$. As j is not explicitly named, $NA(j, \emptyset) = \neg B \vee [\beta]NA(k, \{j\}) \vee [\alpha^-]NA(i, \{j\})$. As i is not explicitly named either, $NA(i, \{j\}) = \neg A$ as the only neighbor of i is j. Finally, $NA(k, \{j\}) = [\beta^*]\neg C$ as the only neighbor of k is j. Thus $NApp(\rho) = [\nu_{\Sigma_1}](\neg B \vee [\beta][\beta^*]\neg C \vee [\alpha^-]\neg A)$.

Given a specification SP, the correctness of SP amounts to verify the validity of the formula $Corr_{SP} = vc(\mathcal{S}, post) \wedge (pre \Rightarrow wp(\mathcal{S}, post))$. It can be shown that $Corr_{SP}$ is a $\mathcal{C}2\mathcal{PDL}$ formula due to the closure by substitutions of $\mathcal{C}2\mathcal{PDL}$ [5]. Therefore we can state the soundness of our calculus in the following theorem.

▶ **Theorem 20.** *Let $SP = (Pre,\ Post,\ \mathcal{R},\ \mathcal{S})$ be a specification where Pre and Post are $\mathcal{C}2\mathcal{PDL}$ formulas, $\mathcal{S}$ is a grove-strategy relative to LDRS $\mathcal{R}$. If the formula $Corr_{SP} = vc(\mathcal{S}, post) \wedge (pre \Rightarrow wp(\mathcal{S}, post))$ is valid then for all graphs G and G', if $G \Rightarrow_{\mathcal{S}} G'$, then $G \models pre$ implies $G' \models post$.*

$Corr_{SP}$ being a formula of $\mathcal{C}2\mathcal{PDL}$, one gets the following corollary from theorem 7.

▶ **Corollary 21.** *Let $SP = (Pre,\ Post,\ \mathcal{R},\ \mathcal{S})$ be a specification where Pre and Post are $\mathcal{C}2\mathcal{PDL}$ formulas, $\mathcal{S}$ is a grove-strategy relative to LDRS $\mathcal{R}$. The verification of the correctness of SP is decidable.*

▶ **Example 22.** The example of Figure 13 contains one rule that is applied as long as possible. As the closure is used, an invariant has to be specified. We choose $\langle\nu_{\Sigma_1}\rangle x$ which is obviously an invariant of the loop. The specifications differ on the post-condition, the first one being what we would expect, that is all elements are labelled A, and the other one being the exact opposite. Both specifications are deemed partially correct. Indeed, as $wp(\rho_1^*, Post) = inv = \langle\nu_{\Sigma_1}\rangle x$, $Pre \Rightarrow wp(\rho_1^*, Post) = \top$. Furthermore, as $vc(\rho_1^*, Post) = (inv \wedge App(\rho_1) \Rightarrow inv[add_C(i, A)]) \wedge (inv \wedge NApp(\rho_1) \Rightarrow Post) \wedge vc(\rho_1, inv)$ and $vc(\rho_1, inv) = \top$ and $inv \Rightarrow inv = \top$, $vc(\rho_1^*, Post) = inv \wedge NApp(\rho_1) \Rightarrow Post)$. But then, as $NApp(\rho_1) = [\nu_{\Sigma_1}]\bot$, $inv \wedge NApp(\rho_1) = \bot$ and thus $vc(\rho_1^*, Post) = \top$ independently of the post-condition. The fact that two specifications leading to opposite results would be both considered correct

$$\rho_1 : \quad \begin{tikzcd} \boxed{i} \arrow[r, Rightarrow] & \boxed{add_C(i, A)} \end{tikzcd}$$

$$SP_{10} = (\langle \nu_{\Sigma_1} \rangle (x \wedge \neg A), [\nu_{\Sigma_1}]A, \{\rho_1\}, \rho_1^*)$$

$$SP_{11} = (\langle \nu_{\Sigma_1} \rangle (x \wedge \neg A), \langle \nu_{\Sigma_1} \rangle \neg A, \{\rho_1\}, \rho_1^*)$$

Figure 13 Scholastic examples of specification. Both of the specifications are partially correct.

seems to be an error but the fact is that the program will never stop and, actually, no result is ever reached.

It is noteworthy that the choice of the invariant is, as usual in Hoare logic [11], far from innocent. Let's assume we chose $inv = \top$ instead. Then, the only difference is that now $inv \wedge NApp(\rho_1) = NApp(\rho_1) = [\nu_{\Sigma_1}]\bot$. But then, $inv \wedge NApp(\rho_1) \Rightarrow [\nu_{\Sigma_1}]\neg A$ is true but not $inv \wedge NApp(\rho_1) \Rightarrow \langle \nu_{\Sigma_1} \rangle A$. A poor choice of inv can prevent someone from proving a program correct.

The example of the Sudoku can be proven to be valid with a suitable choice of *Pre*, *Post* and *inv*.

5 Related work

One of the main features of the class of rewriting systems we introduce in this paper consists in specifying application conditions as logic formulas that label nodes of the left-hand sides. This is, to our knowledge, a new approach to specify application conditions. In [9] the idea of additional (negative) application conditions has been introduced in a framework based on category theory.

Our second contribution is to introduce a decidable verification procedure for the considered rewrite systems and the logic used to express system properties. The problem of reasoning about graph transformations is known to be complex in its full generality [12].

One approach to program verification, as exemplified by [10, 16], is similar to the one we pursue here, i.e., the goal is to generate the weakest pre-condition for a condition to be satisfied. Our method strongly diverges from theirs in very key points, though. First and foremost, their rewriting systems are based on algebraic methods whereas ours are purely algorithmic. Secondly, our graphs are logically attributed, that is each node is labeled with formulas. Another difference lies in the considered logics. Indeed, the logic presented in [16] is equivalent to monadic second-order logic. It thus allows to express second-order quantification which is out of the scope of our logic. The one in [10] is equivalent to first-order graph formulas whose expressivity is not comparable to ours (i.e., there are formulas that can be expressed in one logic but not in the other and vice versa). Both monadic second-order and first-order graph logics are undecidable. On the other hand, we have carefuly chosen an extension of dynamic logic that preserves decidability. Last but not least, the way they compute the pre-conditions is also much different from ours. We use the notion of substitutions whereas their conditions are built incrementally on the rules.

In [13], the authors also use a stronger logic, monadic second-order logic again, and the verification problem is decidable. This is due to the fact that their transformations are bisimulation-generic whereas ours are not. Furthermore they do not label nodes. In both cases, the core of the reflexion is centered in modifications of the structure of the data (that is how general classes of nodes and edges interact). On the other hand, we are able to do the core of these transformation but our focus is on localized modifications that is a change at the instance level.

Another approach to the verification of graph transformation is the classical model-checking applied in the case where states are specified by graphs [17]. This approach, which needs the development of a possibly infinite transition systems, departs from ours.

A verification method of graph transformation is tackled in [14] where a set of forbidden graphs is given and where context-free graph grammars are used instead of logics to define post-conditions. This method is quite different from ours.

In [3], a logic dedicated to mimic graph transformations has been introduced. That logic can also be used to specify properties over graph transformations. The expressive power of that logic is very rich but its validity problem is not decidable.

In [6], an extension of equational logic to graph transformation has been investigated. Theories generated by such logic are not recursively enumerable in general. Thus no completeness nor decidability results can be expected. In addition, in the considered logic bisimilar graphs cannot be distinguished.

The modification of Knowledge Bases is a much different but very active field. Belief revision [1] deals with the addition of new knowledge and how to modify the Knowledge Base so that it is still consistent. That means that a lot of modifications may be hidden in one update action. This is quite a different approach as we want, on the other hand, to know exactly what actions are performed and use that to define our new knowledge.

The work in [4] is heading toward the same goal as the present paper but both the programming language, that is slightly less expressive and imperative, and the considered logic are different.

6 Conclusion

We have presented a new class of graph rewrite systems, LDRSs, where left-hand sides can express additional application conditions defined as $\mathcal{C2PDL}$ formulas. We defined computations with these systems by means of rewrite strategies. There is certainly much work to be done around such systems with logically decorated left-hand sides. For instance, the extension to narrowing derivations, which is a matter of future work, would use an involved unification algorithm taking into account the underlying logic. We have also presented a sound Hoare-like calculus for specifications with pre- and post-conditions in $\mathcal{C2PDL}$ and show that the considered correctness problem is decidable. These positive achievements deserve to be extended to other logics which we intend to investigate in the future.

References

1 Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symb. Log.*, 50(2):510–530, 1985. doi:10.2307/2274239.

2 Carlos Areces and Balder ten Cate. Hybrid logics. In *Studies in Logic and Practical Reasoning*, volume 3, pages 821–868. Elsevier, 2007.

3 Philippe Balbiani, Rachid Echahed, and Andreas Herzig. A dynamic logic for termgraph rewriting. In *5th International Conference on Graph Transformations (ICGT)*, volume 6372 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2010.

4 Jon Haël Brenas, Rachid Echahed, and Martin Strecker. A Hoare-like calculus using the SROIQ σ logic on transformations of graphs. In *Theoretical Computer Science – 8th IFIP TC 1/WG 2.2 International Conference, TCS 2014, Rome, Italy, September 1-3, 2014. Proceedings*, pages 164–178, 2014. doi:10.1007/978-3-662-44602-7_14.

5 Jon Hael Brenas, Rachid Echahed, and Martin Strecker. *C2PDLS*: A combination of combinatory and converse PDL with substitutions. 2015. URL: `http://lig-membres.imag.fr/echahed/c2pdls.pdf`.

6 Ricardo Caferra, Rachid Echahed, and Nicolas Peltier. A term-graph clausal logic: Completeness and incompleteness results. *Journal of Applied Non-classical Logics*, 18(4):373–411, 2008.

7 Rachid Echahed. Inductively sequential term-graph rewrite systems. In *4th International Conference on Graph Transformations, ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2008.

8 Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979. `doi:10.1016/0022-0000(79)90046-1`.

9 Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996. `doi:10.3233/FI-1996-263404`.

10 Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *MSCS*, 19:245–296, 2009.

11 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. `doi:10.1145/363235.363259`.

12 Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin / Heidelberg, 2004. URL: `http://www.cs.umass.edu/~immerman/pub/cslPaper.pdf`.

13 Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Graph-transformation verification using monadic second-order logic. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 17–28, 2011. `doi:10.1145/2003476.2003482`.

14 Barbara König and Javier Esparza. Verification of graph transformation systems with context-free specifications. In *Graph Transformations – 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 – October 2, 2010. Proceedings*, pages 107–122, 2010. `doi:10.1007/978-3-642-15928-2_8`.

15 Solomon Passy and Tinko Tinchev. An essay in combinatory dynamic logic. *Inf. Comput.*, 93(2):263–332, 1991. `doi:10.1016/0890-5401(91)90026-X`.

16 Christopher M. Poskitt and Detlef Plump. Verifying monadic second-order properties of graph programs. In *Graph Transformation – 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings*, pages 33–48, 2014. `doi:10.1007/978-3-319-09108-2_3`.

17 Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 – October 2, 2004, Proceedings*, pages 226–241, 2004. `doi:10.1007/978-3-540-30203-2_17`.

18 Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.