How Much Are Your Geraniums? Taking Graph Conditions Beyond First Order

Arend Rensink(✉)

University of Twente, Enschede, The Netherlands
arend.rensink@utwente.nl

Abstract. Previous work has shown how first-order logic can equivalently be expressed through nested graph conditions, also called *condition trees*, with surprisingly few ingredients. In this paper, we extend condition trees by adding set-based operators such as sums and products, calculated over operands that are themselves characterised by first-order logic formulas. This provides a greatly improved way to specify computations such as: *given that the price of a geranium plant equals* 2 *per flower petal, return the average price of all geraniums with at least one flower*.

We claim the same level of expressive equivalence as before between (extended) condition trees and a certain class of logic formulas; we show that the latter go beyond what can be expressed in first-order logic.

On the practical side, we evaluate the performance and usability of set-based operators by specifying and comparing the example geranium property, with and without set-based operators, in the graph transformation tool GROOVE.

1 Introduction

Graph transformation is a formalism that can be used for different purposes: to define graph languages (as a generalisation of string grammars; see for instance [8]), to define binary relations and functions over graphs (as a generalisation of term rewriting; see for instance [22]) or as a rule-based formalism to describe the behviour of a system (as a generalisation of, for instance, Petri nets; see [18]). In each of these settings, it is of interest how powerful the graph transformation rules are. Here a balance must be struck between sticking to the simplest kind of rules, which have very well-defined properties but provide only low-level building blocks, and allowing more elaborate, powerful rules, which enable one to specify the language, relation or dynamic system at hand more directly and concisely but whose effect is correspondingly harder to analyze.

In this paper, we follow the algebraic approach to graph transformation pioneered by the late Hartmut Ehrig.[1] A good reference work for the theoretical

[1] Ehrig was also a leading researcher in *algebraic data specification*, i.e. [7]; in that capacity he was one of the founders of ACT-ONE, the data type specification language of LOTOS, the standardisation of which in [20] was one of Ed Brinksma's early scientific achievements.

J.-P. Katoen et al. (Eds.): Brinksma Festschrift, LNCS 10500, pp. 191–213, 2017.
DOI: 10.1007/978-3-319-68270-9_10

background of the approach in general is the book [5]. The simplest kind of rules, working on the simplest kind of graphs, consist only of a *left hand side* and a *right hand side*, both of which are graphs. Applying such a rule to a given *host graph* roughly consists of the following two steps:

- Finding a match of the left hand side to the host graph, in the form of a graph morphism;
- Replacing the image of the left hand side in the host graph that was identified by the match by a copy of the right hand side, while preserving some context.

From the point of view of analyzability, one of the advantages of these simple rules is that the condition for their applicability as well as the scope of their application are fixed by the left hand side: the rule tests for the presence of a certain substructure in the host graph, and all ensuing changes are applied within this substructure. This, however, is simultaneously a disadvantage when one wants to specify a transformation consisting of an a priori unknown number of small, identical changes. A good example is the effect of firing a transition in a Petri net: in this situation, *all* input places should contain a token which, moreover, should be consumed; and *all* output places should receive a token. There is no bound on the number of input places or output places a transition in an actual Petri net may have; so to achieve the desired effect using only simple graph transformation rules, one has to take refuge to one of the following solutions:

- Devise a fairly complicated protocol of simple rule applications in which the input places are individually tested for the presence of a token, after which those are also individually removed and tokens are placed on the individual output places.
- Create one rule for every combination of m input places and n output places. However, apart from the fact that this gives rise to an infinitary family of rules, one also has to ensure that the rule for $i \times j$ input/output places is not applicable to a transition with $m \times n$ input/output places with $m > i$ or $n > j$. This requires a test for the absence, rather than the presence, of certain structures in the host graph, which in itself is beyond our simple rules as well.

Because limitations of this kind were found to severely hamper the practical use of graph transformation in practice, mechanisms to generalise and extend both the applicability condition of rules as well as their effect have been studied for quite some time. For instance:

- [14] proposes to enrich rules with so-called *negative application conditions* (NACs) that test for the absence of structure, nullifying one of the obstacles to the second solution discussed above.
- [6,23] generalise NACs to *nested graph conditions*, using which any first-order property of graphs can be used as a rule applicability condition. In particular, the nesting structure mimics the concept of *alternating quantifiers*.

- [12,26] study the concept of *amalgamation* of rules, which is a composition mechanism that allows building complex rules such as the Petri net firing rule out of arbitrarily many copies of small "simple" rules. Such composed rules are sometimes called *multi-rules*.
- [24,25] present *nested rules*, which can be seen as a marriage of nested graph conditions and amalgamation: in terms of [26], the proof of the nested graph condition serves as the amalgamation scheme. A nested rule can contain universal quantification not only within its applicability condition, but also within sub-rules, which then have an effect wherever in the host graph the corresponding applicability sub-condition is satisfied.

However, first-order logic has its limitations, and there are in fact applicability conditions and multi-rules that can still not easily be expressed using any of the above techniques. For instance, there are cases where it is relevant to know or compute a *collective* value for a set of sub-graphs characterised by some property. This is where the current paper comes in. As an example, consider the following task:

> Given that the price of a geranium plant equals 2 per flower petal, compute the average price of all geraniums with at least one flower.

In mathematical notation, this can be expressed as follows:

$$(\sum_{g \in G} \mathit{price}(g))/|G| \quad \text{where} \quad \begin{array}{r} \mathit{price} : g \mapsto \sum_{f \in F_g} 2 * \mathit{petals}(f) \\ G = \{g \mid \mathbf{Geranium}(g) \wedge F_g \neq \varnothing\} \\ F_g = \{f \mid \mathbf{Flower}(f) \wedge \mathsf{has}(g, f)\} \end{array} \tag{1}$$

This has the following noteworthy features:

- $\mathbf{Geranium}(x)$, $\mathbf{Flower}(y)$ and $\mathsf{has}(x, y)$ are predefined predicates expressing, respectively, that x is a geranium, y is a flower, and y is a flower of x,
- $\mathit{petals}(x)$ is a predefined partial function returning the number of petals of x, if x is a **Flower**.
- G and F_g (for $g \in G$) are defined as sets of, respectively, all geraniums and all flowers of geranium g;
- $\mathit{price}(g)$ is the price of geranium g, defined as twice the number of petals of all flowers of g.

The need to use sets of entities (G and F_g) and set-based operators ($\sum$ and the cardinality $|G|$) take this beyond what can be expressed in first-order logic. Consequently, the computation of a formula such as the above is essentially as tricky to specify using graph transformation, even with nested rules as in [25], as the firing of a Petri net transition is with only simple rules: again, one has to sum up first the flower petals and subsequently the individual geranium prices one by one.

This paper proposes a way to directly support set-based operators. We present this in three aspects: 1. a generalisation of nested graph conditions; 2. an extension of first-order logic; 3. an experiment showing the performance and conciseness gain with respect to the encoding of (1) using simple rules. We claim that

this extension is not just theoretically interesting but also practically useful: in the course of time, we have received multiple feature requests for our graph transformation tool GROOVE [11] to support functionality of this kind, most lately in the context of [16].

The remainder of this paper is structured as follows: in Sect. 2 we present the necessary background from algebra and graph theory; in Sect. 3 we recall the theory behind nested graph conditions. Section 4 contains the main technical contribution, viz. the extension to set-based operators. In Sect. 5 we report on the experiment of encoding formula (1) above in terms of set-based operators; and in Sect. 6 we summarise the findings and present related and future work.

2 Definitions

In this section, we set the stage by recalling some basic notions from algebra, graph theory and logic that we need to present our contribution.

2.1 Algebra

We restrict ourselves to the domain of integers. This means we work with a fixed signature Σ with consisting of the standard arithmetic operators, such as add (addition), mul (multiplication) and div (division). Each operator has an *arity* $\nu(o) \in \mathbb{N}$. Constants are included as nullary operators. We also use a universe of variables $\mathcal{V}$. From these ingredients, we define *terms* through the following grammar:

$$t ::= x \mid o(t_1, \ldots, t_{\nu(o)}) \tag{2}$$

where $x \in \mathcal{V}$ and $o \in \Sigma$. The set of terms over a given set of variables $V \subseteq \mathcal{V}$ is denoted $\mathbb{T}(V)$.

Definition 1 (algebra, homomorphism). *An* algebra *is a tuple* $A = \langle D, (F^o)_{o \in \Sigma} \rangle$ *consisting of*

- *a value domain* D*;*
- *a partial function* $F^o \colon D^{\nu(o)} \to D$ *for each* $o \in \Sigma$*.*

Given two algebras A_1, A_2*, a* homomorphism $h \colon A_1 \to A_2$ *is a partial function* $h \colon D_1 \to D_2$ *such that for all operators* $o \in O$ *and all* $v_i \in D$ $(1 \leq i \leq \nu(o))$*:*

$$h(F_1^o(v_1, \ldots, v_{\nu(o)})) = F_2^o(h(v_1), \ldots, h(v_{\nu(o)}))$$

provided that all function applications are defined.

The functions F^o are allowed to be partial so that division by zero can be accounted for. Henceforth we identify an algebra A with its value domain and just talk about the values of A, rather than of the domain of A. In fact, in this paper we restrict ourselves to two particular (families of) algebras:

- The term algebra $\mathbb{T}(V)$ for arbitrary finite sets of variables $V \subseteq \mathcal{V}$, where each function F^o is total and constructs a new term by applying the corresponding operator o to the operand terms.
- The natural algebra $\mathbb{I}$ consisting of the "actual" integers and the "actual" functions for addition, multiplication, etc. (and division by 0 is undefined).

2.2 Attributed Graphs

In the following, $\mathcal{N}$ denotes a universe of nodes, and $\mathcal{L}$ a universe of graph (edge) labels.

Definition 2 (graph, morphism). *A* graph *is a tuple* $G = \langle N, E \rangle$ *where*

- $N \subseteq \mathcal{N}$ *is a set of nodes;*
- $E \subseteq N \times \mathcal{L} \times N$ *is a set of edges.*

Graph G *is called* attributed *over an algebra* A *if* $A \subseteq N_G$*. The class of all graphs is denoted* $\mathbf{Graph}$ *and the subclass of attributed graphs* $\mathbf{Graph}^A$*.*

Given two graphs G_1, G_2*, a* morphism $f\colon G_1 \to G_2$ *is a function* $f\colon N_1 \to N_2$ *such that:*

$$f : (n_1, a, n_2) \mapsto (f(n_1), a, f(n_2)).$$

Morphism f *is called* attributed *if the* G_i *are attributed over* A_i *(*$i = 1, 2$*) and* $f \restriction A_1$ *is a homomorphism from* A_1 *to* A_2*. The class of all morphism is denoted* $\mathbf{Morph}$*.*

Morphism f *is called* injective *if* $f(n_1) = f(n_2)$ *implies* $n_1 = n_2$*.*

We use $src(e), lab(e)$ and $tgt(e)$ to denote the source, label and target of an edge e, and $dom(f), cod(f)$ to denote the domain and codomain of a morphism f. If G is attributed over A, we call the elements of $A \subseteq N_G$ *data nodes* and the elements of $N_G \setminus A$ *pure nodes*. We also refer to a graph that is not attributed as a *pure graph*.

Note that the node set of an attributed graph is typically infinite, because algebra domains are. The only data nodes of such a graph we are usually interested in (and that are included in figures) are those that are connected by some edge to a pure node.

In practice, the only attributed morphisms f we will consider are such that either $dom(f)$ and $cod(f)$ are graphs over $\mathbb{T}(V_1)$ and $\mathbb{T}(V_2)$ for some $V_1 \subseteq V_2 \subseteq \mathcal{V}$ and $f \restriction \mathbb{T}(V_1)$ is the identity homomorphism, or such that $dom(f)$ is attributed over $\mathbb{T}(V)$ for some $V \subseteq \mathcal{V}$ and $cod(f)$ is attributed over $\mathbb{I}$.

2.3 First-Order Logic

In its purest form, first-order logic (**FOL**) reasons about arbitrary structures, using formulas composed from some predefined set of n-ary predicates. Here we restrict to binary predicates, which coincide with the set of edge labels $\mathcal{L}$ introduced above.

Furthermore, we also use a set of variables $\mathcal{X}$, which for now is not connected to the data variables $\mathcal{V}$ used above. The grammar of **FOL** is then given by:

$$\phi ::= a(x, y) \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \forall x : \phi \mid \exists x : \phi \tag{3}$$

for arbitrary $a \in \mathcal{L}$ and $x, y \in \mathcal{X}$. We also use the notation $\mathsf{Q}\, X : \phi$ with $\mathsf{Q} \in \{\exists, \forall\}$ and finite $X \subseteq \mathcal{X}$ to denote the simultaneous existential or universal

quantification over all variables in X. We use $fv(\phi)$ to denote the free variables of a formula ϕ, defined in the usual way; we call formula ϕ *closed* if $fv(\phi) = \varnothing$.

Formulas are evaluated over *interpretations*, which define a domain of discourse as well as actual relations for all predicate symbols. In our setting, interpretations coincide with graphs, although in keeping with tradition we will denote them I rather than G: for a given predicate symbol $a \in \mathcal{L}$, the actual binary relation as defined by a given interpretation (i.e., graph) I is nothing but the set of pairs $(src(e), tgt(e))$ for all edges $e \in E_I$ with $lab(e) = a$.

We also need the concept of an *assignment* α, which is a partial mapping $\alpha \colon \mathcal{X} \to N_G$ mapping at least all free variables of ϕ to nodes in the interpretation. For given assignments α, β, we use the following constructions:

$$\alpha[x \leftarrow n] : y \mapsto \begin{cases} n & \text{if } x = y \\ \alpha(y) & \text{otherwise.} \end{cases} \qquad \alpha[\beta] : y \mapsto \begin{cases} \beta(y) & \text{if } y \in dom(\beta) \\ \alpha(y) & \text{otherwise.} \end{cases}$$

The semantics of ***FOL*** is given by a relation $I, \alpha \vDash \phi$ expressing "I satisfies ϕ under α," defined as follows:

$$\begin{array}{ll} I, \alpha \vDash a(x, y) & \text{if } (\alpha(x), a, \alpha(y)) \in E_I \\ I, \alpha \vDash \phi_1 \wedge \phi_2 & \text{if } I, \alpha \vDash \phi_1 \text{ and } I, \alpha \vDash \phi_2 \\ I, \alpha \vDash \phi_1 \vee \phi_2 & \text{if } I, \alpha \vDash \phi_1 \text{ or } I, \alpha \vDash \phi_2 \\ I, \alpha \vDash \neg\phi & \text{if not } I, \alpha \vDash \phi \\ I, \alpha \vDash \forall x : \phi & \text{if } I, \alpha[x \leftarrow n] \vDash \phi \text{ for all } n \in N_I \\ I, \alpha \vDash \exists x : \phi & \text{if there is a } n \in N_I \text{ such that } I, \alpha[x \leftarrow n] \vDash \phi. \end{array}$$

α may be omitted if ϕ is closed.

3 Nested Graph Conditions

In this section, we recall nested graph conditions and their equivalence (in expressive power) to ***FOL***.

3.1 Conditions as Graphs

In order to understand in what sense a graph can represent a property also expressible in ***FOL***, consider that any graph C can be seen as a pattern that occurs or does not occur in another graph G—where "occurring in" means that there exists a morphism $\gamma \colon C \to G$. For an equivalent ***FOL*** formula, we first have to establish a correspondence of nodes (of C) to variables (in $\mathcal{X}$). For this purpose, we fix a mapping $\xi \colon \mathcal{N} \to \mathcal{X}$ that associates a variable with every node.

Definition 3 (condition graph). *A* condition graph *is a graph C such that ξ is injective on N_C.*

We use $\xi_C = \xi \restriction N_C$ to denote the restriction of ξ to the nodes of condition graph C (hence ξ_C^{-1} is well-defined). We also denote $x_n = \xi(n)$.

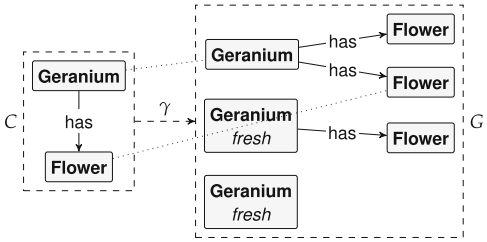


Fig. 1. Example graph condition with satisfying morphism γ (The dotted lines indicate the node mapping)

Whether or not a function $\gamma\colon N_C \to N_G$ is a morphism from C to G is a property of G that is equivalently expressed by the following formula, evaluated over $I = G$ with assignment $\alpha = \gamma \circ \xi_C^{-1}$:

$$\phi_C \;:=\; \bigwedge_{e \in E_C} lab(e)(x_{src(e)}, x_{tgt(e)}) \tag{4}$$

This equivalence is formally stated by the following proposition.

Proposition 1. *Let C be a condition graph, G a graph and α an assignment to N_C.*

1. $\alpha \circ \xi_C$ is a morphism from C to G if and only if $G, \alpha \vDash \phi_C$.
2. There exists a morphism from C to G if and only if $G \vDash \exists\, \xi(N_C) : \phi_C$.

Clearly, morphisms in the satisfaction of condition graphs take over the role of assignments in the satisfaction of ***FOL*** formulas, viz., to bind nodes of the target graph [the interpretation] to nodes [variables] of the source graph [the formula].

Example 1. As an example, consider the graphs C and G in Fig. 1. Inscribed node labels are syntactic sugar for self-edges with that label. C specifies that there is a flowering geranium, which is equivalently expressed by

$$\exists x, y : \mathbf{Geranium}(x, x) \wedge \mathbf{Flower}(y, y) \wedge \mathsf{has}(x, y).$$

(In keeping with the decision to restrict ***FOL*** to binary predicates, this uses $\mathbf{Geranium}(x, x)$ rather than $\mathbf{Geranium}(x)$ as in (1) to express that x is a geranium; however, there is no conceptual difference between the two.) G actually has 3 pairs of **Geranium**-**Flower**-pairs that satisfy this condition, one of which is identified by the morphism γ in the figure.

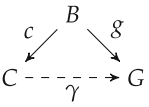
The compositional definition of $\vDash$ over ***FOL*** gradually builds up the assignment α. To also take pre-existing bindings into account in the case of graphs, we will use graph morphisms, rather than graphs, both for the conditions themselves and for the combination of interpretation and assignment.

3.2 Morphisms as Conditions

In the following we consider morphisms $g\colon B \to G$, where B stands for the *bound graph*, which can be seen as a sub-pattern of a condition graph that has already been "found". The nodes of B will turn out to correspond to free variables in the formula to be checked. Accordingly, rather than just condition graphs C, we consider *condition morphisms*.

Definition 4 (condition morphism). *A condition morphism is an injective morphism $c\colon B \to C$ between condition graphs, such that $\xi_B = \xi_C \circ c$.*

Morphism $g\colon B \to C$ satisfies condition morphism $c\colon C \to D$ if (just as in the case of condition graphs, see Proposition 1) a third morphism $\gamma\colon C \to G$ exists, which however (in addition) satisfies $\gamma \circ c = g$; in other words, γ has to respect the image of the bound graph. This is represented by the commuting diagram.



Notationally, this is expressed by the (overloaded) relation $\models$:

$$g \models c \text{ if there is some } \gamma\colon cod(c) \to cod(g) \text{ such that } \gamma \circ c = g. \tag{5}$$

In fact, γ is a *witness* or *proof* that c exists in (the codomain of) g. We also use $[\![c]\!]$ to denote the semantic function of c that maps any $g\colon B \to G$ to the set of proofs of c on g, thus:

$$[\![c]\!] : g \mapsto \{\gamma \in \textbf{\textit{Morph}} \mid \gamma \circ c = g\}. \tag{6}$$

A condition morphism $c\colon B \to C$ is equivalent to the following formula:

$$\phi_c \;:=\; \exists \xi(N_C \setminus c(N_B)) : \phi_C \tag{7}$$

The equivalence is formally stated by the following proposition:

Proposition 2 (condition morphism equivalence). *Let $c\colon B \to C$ be a condition morphism and $g\colon B \to G$ a morphism; then $g \models c$ if and only if $G, g \circ \xi_B^{-1} \models \phi_c$.*

It should be noted that the injectivity of c is required for this to work: if there were distinct $n_1, n_2 \in N_B$ such that $c(n_1) = c(n_2)$, then ϕ_c would have to include a predicate equating x_{n_1} and x_{n_2}. For simplicity we have chosen to omit equality from the version of ***FOL*** used in this paper and restrict condition morphisms to injective ones; in the conclusions we briefly discuss what would be required to generalise the setup.

Vice versa, (we claim without proof that) any formula ϕ from the following fragment of ***FOL*** can be easily encoded into an equivalent graph condition:

$$\phi ::= a(x, y) \;\mid\; \phi_1 \wedge \phi_2 \;\mid\; \exists x : \phi.$$

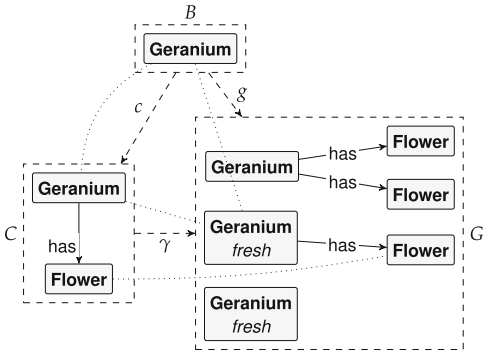


Fig. 2. Example condition morphism with satisfying morphism: $g \vDash c$

Example 2. Figure 2 is a variation on Fig. 1 where the bound graph B pre-identifies the particular **Geranium** of which we want to know whether it has a **Flower**. A proof γ of the satisfaction of c on g is drawn in.

3.3 Trees as Conditions

In the following, we use *diagrams*, which are special kinds of graphs, of which the nodes are labelled with elements of ***Graph*** and the edges are labelled with elements of ***Morph*** with domain and codomain corresponding to the edge source and target. We will apply the usual trick of identifying diagram nodes and edges with their labels; in the context of this paper this will not give rise to ambiguities. If G is a node in a diagram $\boldsymbol{D}$, $out_{\boldsymbol{D}}(G) = \{f \in \boldsymbol{D} | dom(f) = G\}$ denotes the set of morphisms in $\boldsymbol{D}$ with domain G. Such a diagram $\boldsymbol{D}$ is *tree-shaped* if it is acyclic and has a single node $rt(\boldsymbol{D})$ with no incoming edges, whereas all other nodes have precisely one incoming edge. If $\boldsymbol{D}$ is tree-shaped, then for any node G of $\boldsymbol{D}$ we use $\boldsymbol{D}[G]$ to denote the subtree of $\boldsymbol{D}$ rooted in G.

Definition 5 (condition tree). *A condition tree $\boldsymbol{C}$ is a tree-shaped diagram consisting of condition graphs and condition morphisms, in which, moreover, every graph C is labelled with a boolean operator $\mathsf{O}^C \in \{\bigvee, \bigwedge\}$ and every morphism c with a quantor $\mathsf{Q}^c \in \{\forall, \exists\}$.*

$\boldsymbol{C}$ is called closed *if $rt(\boldsymbol{C})$ is the empty graph.*

Satisfaction of a condition tree is once more expressed by a relation $\boldsymbol{C} \vDash g$, where $dom(g) = rt(\boldsymbol{C})$, recursively defined as follows:

$$g \vDash \boldsymbol{C} \quad \text{if} \quad \mathsf{O}^{rt(\boldsymbol{C})}{}_{c \in out_{\boldsymbol{C}}(rt(\boldsymbol{C}))} \; \mathsf{Q}^c \gamma \in [\![c]\!](g) : \gamma \vDash \boldsymbol{C}[cod(c)]. \tag{8}$$

In this definition, we use a notational trick by which the quantors Q and logical connectives O are actually used in their natural-language meanings.

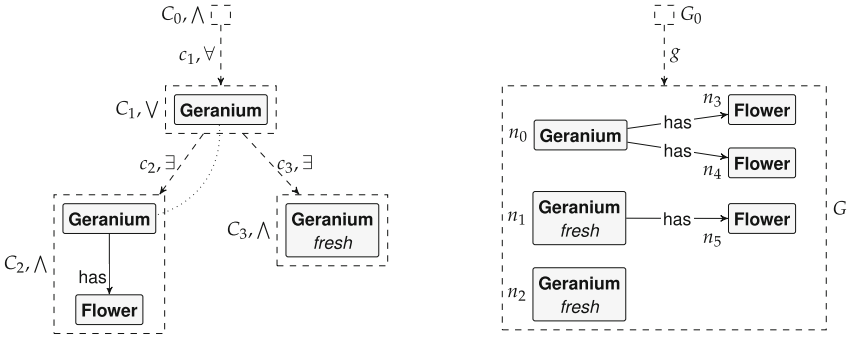


Fig. 3. Example condition tree $\boldsymbol{C}$ and morphism g such that $g \models \boldsymbol{C}$

We call a ***FOL*** formula ϕ and a condition tree $\boldsymbol{C}$ *equivalent* if for any graph morphism $g\colon rt(\boldsymbol{C}) \to G$:

$$G, g \circ \xi^{-1}_{rt(\boldsymbol{C})} \models \phi \quad \text{if and only if} \quad g \models \boldsymbol{C}. \tag{9}$$

Example 3. Figure 3 shows a condition tree $\boldsymbol{C}$ that expresses the property "all geraniums either have a flower or are fresh", and a morphism g for which $\boldsymbol{C}$ is satisfied. Note that C_2 and C_3, which are childless in $\boldsymbol{C}$, are labelled $\bigwedge$: this specifies that all their outgoing morphisms must be covered, which is vacuously satisfied. An equivalent ***FOL*** formula is:

$$\forall x : \mathbf{Geranium}(x, x) \to ((\exists y : \mathbf{Flower}(y, y) \wedge \mathsf{has}(x, y)) \vee \mathit{fresh}(x, x)).$$

A key result, reformulated from [25], is that conditions trees and ***FOL*** formulas are expressively equivalent. This generalises Proposition 2 to condition trees.

Theorem 1 (condition tree equivalence).

1. *For every* ***FOL*** *formula* ϕ*, there is an equivalent condition tree* $\boldsymbol{C}_\phi$*;*
2. *For every condition tree* $\boldsymbol{C}$*, there is an equivalent* ***FOL*** *formula* $\phi_{\boldsymbol{C}}$*.*

3.4 Proof Trees

For the extension to set-based operators in the next section, it is useful to present an alternative characterisation of the satisfaction of condition trees defined in (8). Given a morphism $g\colon B \to G$ and a condition morphism $c\colon B \to C$ in a condition tree $\boldsymbol{C}$, a set of morphisms $\Gamma \subseteq (C \to G)$ is said to *cover* c if $\mathsf{Q}^c = \exists$ and $|\Gamma \cap [\![c]\!](g)| > 0$, or $\mathsf{Q}^c = \forall$ and $[\![c]\!](g) \subseteq \Gamma$.

A *proof* of a condition tree $\boldsymbol{C}$ on a morphism $g\colon rt(\boldsymbol{C}) \to G$ is itself a tree-shaped diagram $\boldsymbol{P}$, with a mapping π to $\boldsymbol{C}$ that preserves all graphs and morphisms of $\boldsymbol{P}$, and for every graph P a morphism $\gamma_P\colon P \to G$, such that

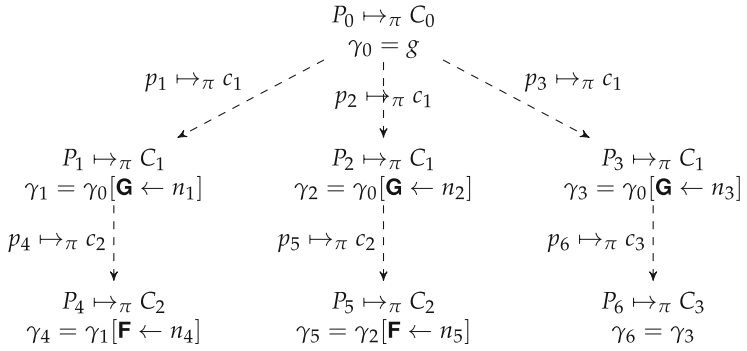


Fig. 4. Proof for $g \vDash \boldsymbol{C}$ in Fig. 3. **G** is the **Geranium**-node in C_1 and **F** the **Flower**-node in C_2

(i) $\gamma_{rt(\boldsymbol{P})} = g$
(ii) $\gamma_{cod(p)} \circ p = \gamma_{dom(p)}$ for all edges p in $\boldsymbol{P}$
(iii) If $\mathsf{O}^{\pi(P)} = \bigvee$ for P in $\boldsymbol{P}$, then $out_{\boldsymbol{P}}(P)$ covers *some* $c \in out_{\boldsymbol{C}}(\pi(P))$;
(iv) If $\mathsf{O}^{\pi(P)} = \bigwedge$ for P in $\boldsymbol{P}$, then $out_{\boldsymbol{P}}(P)$ covers *all* $c \in out_{\boldsymbol{C}}(\pi(P))$.

$\boldsymbol{P}$ is called a *minimal proof* if it is no longer a proof when a single branch is removed. In practice, we always use minimal proofs.

Example 4. Figure 4 shows a proof $\boldsymbol{P}$ of the satisfaction $g \vDash \boldsymbol{C}$ in Example 3. The morphisms γ_i for the nodes P_i of $\boldsymbol{P}$ are constructed step by step, starting with the empty morphism ($\gamma_0 = g$) and adding images along the way. Note that this is not the only proof of $\boldsymbol{C}$ on g: from P_1, another mapping from **F** exists to the other flower n_3 of **Geranium**-node n_0, whereas from P_2, the fact that **Geranium**-node n_1 is *fresh* means that it is also possible to cover c_2 rather than c_3.

The following proposition states that a condition tree is satisfied by a morphism g if and only if there exists a proof of this kind. In other words, proofs are witnesses of the satisfaction of a condition tree.

Proposition 3. *Let $\boldsymbol{C}$ be a condition tree and g a morphism with $dom(g) = rt(\boldsymbol{C})$; then $g \vDash \boldsymbol{C}$ if and only if there exists a proof $\boldsymbol{P}$ for $\boldsymbol{C}$ on g.*

4 Set-Based Operators

All of the theory in Sect. 3 can be extended without any problem whatsoever to attributed graphs, with the understanding that condition graphs C are always attributed over $T_\Sigma(V_C)$ for some fixed $V_C \subseteq \mathcal{V}$ and host graphs are attributed over $\mathbb{I}$, that $\mathcal{V} \subseteq \mathcal{X}$ (all data variables are logical variables) and that ξ is the identity over $\mathcal{V}$ (hence every algebra variable stands for itself). To re-establish the connection to ***FOL*** (Theorem 1), all that is required is to extend basic predicates

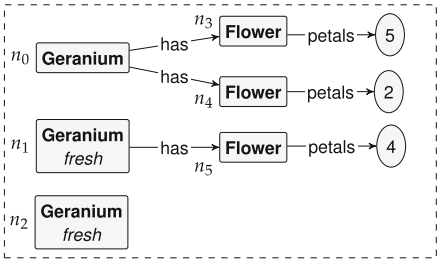


Fig. 5. Flowering geraniums, subject to the price computation $\mathsf{sum}(\mathsf{mul}(2, x) \mid x, f_0 : \mathsf{has}(g_0, f_0) \wedge \mathsf{petals}(f_0, x))$

to range over terms rather than just variables, i.e., so that they are of the form $a(t_1, t_2)$, and to extend the semantics accordingly to

$$I, \alpha \vDash a(t_1, t_2) \qquad \text{if } (h_\alpha(t_1), a, h_\alpha(t_2)) \in E_I$$

where $h_\alpha \colon \mathbb{T}(V_I) \to \mathbb{I}$ is uniquely determined by the assignment α.

This sets the stage for the introduction of set-based operators. Essentially, these arise from commutative and associative binary operators in Σ; essentially, they are operators that can be applied to arbitrary finite (in some case only non-empty) multisets of operands without regard for their ordering. Examples are:

- card, which just returns the number of operands;
- sum and mul, which compute the product, respectively sum;
- max and min, which compute the maximum, respectively minimum.

We use $\Sigma^{\mathbb{O}}$, ranged over by $\mathbb{O}$, to denote the collection of set-based operators. Clearly, each of the $\mathbb{O}$ has a corresponding operation $F^{\mathbb{O}} \colon \mathbf{2}^{\mathbb{I}} \to \mathbb{I}$. The core idea of this paper is that the operands can be computed over sets of graphs that are themselves characterised through universal quantification. This gives rise to terms of the form $\mathbb{O}(t \mid X : \phi)$, which computes t for all assignments to the (hitherto free) variables in X that cause ϕ to be satisfied.

Example 5. Let $t = \mathsf{sum}(\mathsf{mul}(2, x) \mid x, f_0 : \mathsf{has}(g_0, f_0) \wedge \mathsf{petals}(f_0, x))$ be a term, to be computed over the graph in Fig. 5. Note that g_0 is a free variable in t, which should be assigned one of the **Geranium**-nodes in the graph before the term can be evaluated. If $\alpha : g_0 \mapsto n_0$, then $\mathsf{has}(g_0, f_0) \wedge \mathsf{petals}(f_0, x)$ can be satisfied by assigning n_3 to f_0 and 5 to x, or by assigning n_4 to f_0 and 2 to x; hence t evaluates to $2 * 5 + 2 * 2 = 14$. Alternatively, if $\alpha : g_0 \mapsto n_1$ then t evaluates to 8, and if $g_0 \mapsto n_2$ then t becomes 0.

To formalise this, we extend and combine the term grammar (2) and the *FOL* grammar (3):

$$\begin{aligned} t &::= x \mid o(t_1, \ldots, t_{\nu(o)}) \mid \mathbb{O}(t \mid X : \phi) \\ \phi &::= a(t_1, t_2) \mid \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \forall x : \phi \mid \exists x : \phi. \end{aligned} \tag{10}$$

The function fv is extended with $fv(\mathbb{O}(t \mid X : \phi)) = fv(t) \setminus X$. We use $\mathbb{S}$ to denote the set of terms according to this grammar, and $\mathbb{S}(V)$ for those terms that take their free variables from V; and we call the resulting logic *set-based operator logic* (***SBOL***).

Note that (10) has a recursive dependency between the rules for t and ϕ; hence, to interpret ***SBOL***, we need to simultaneously extend the notion of homomorphism as well as the semantics of ***FOL***. Clearly, to evaluate a set-based operator application, we need to have an interpretation available; hence we use "extended homomorphisms" $h_{I,\alpha}$ to map $\mathbb{S}$ to $\mathbb{I}$, where I is an interpretation and α an assignment:

$$h_{I,\alpha} : \mathbb{O}(t \mid X : \phi) \mapsto F^{\mathbb{O}} | h_{I,\alpha[\beta]}(t)\beta \colon X \to I,\ I, \alpha[\beta] \vDash \phi.$$

Here, β may assign any of the elements of the interpretation I to the variables in X. This should be compared to the semantics of universal quantification as in $\forall X : \phi$, where the assignment is likewise extended to all variables in X before ϕ is evaluated. The ***SBOL*** semantic rule for predicates is then straightforward:

$$I, \alpha \vDash a(t_1, t_2) \qquad \text{if } (h_{I,\alpha}(t_1), a, h_{I,\alpha}(t_2)) \in E_I.$$

$\mathbb{O}$-terms are encoded in condition trees by treating them as data variables with additional constraints, namely that the value they are assigned equals the outcome of the corresponding set-based operation. The tricky part is that the subterm t in an $\mathbb{O}$-term $\mathbb{O}(t \mid X : \phi)$ must be evaluated in the same context as ϕ, which corresponds to a child of the condition tree node in which the $\mathbb{O}$-term itself appears. This is illustrated by the following example.

Example 6. Consider the following predicate, based on the term t from Example 5:

$$\phi \ = \ \mathsf{price}(g_0, \mathsf{sum}(\mathsf{mul}(2, x) \mid x, f_0 : \mathsf{has}(g_0, f_0) \wedge \mathsf{petals}(f_0, x))).$$

This formula is satisfied by a graph G if the node assigned to g_0 has a price-labelled edge to the value of the sum-term. However, the subterm $\mathsf{mul}(2, x)$ used in computing the sum must be evaluated in a child of the condition graph encoding ϕ itself, with a corresponding universal quantification of the variables x and f_0. This dependency is encoded by the additional, sum-labelled dotted line in the condition tree of Fig. 6.

Definition 6 (extended condition tree). *An extended condition tree* $\boldsymbol{X}$ *is a condition tree* $\boldsymbol{C}$ *with, for every* $\wedge$*-labelled graph* C *in* $\boldsymbol{C}$*, a partial mapping* $\tau_C \colon V_C \to (\Sigma^{\mathbb{O}} \times \mathbb{T} \times \boldsymbol{Morph})$ *mapping some of the variables in* V_C *to triples* $\langle \mathbb{O}, t, c \rangle$*, where* $c \in out_{\boldsymbol{C}}(C)$ *is* $\forall$*-labelled and* $t \in N_{cod(c)}$*.*

Thus, τ_C identifies which data nodes in C encode terms of the form $\mathbb{O}(t \,|\, X : \phi)$, indicating what is the operator $\mathbb{O}$, what is the term t and which child in the condition tree corresponds to the subformula ϕ. For instance, in Fig. 6, τ_B maps node $p \in N_B$ to $\langle \mathsf{sum}, \mathsf{mul}(2, x), c \rangle$.

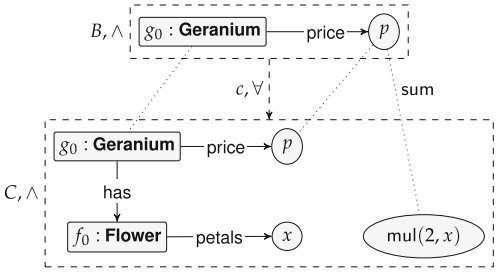


Fig. 6. Extended condition tree for $\mathsf{price}(g_0, \mathsf{sum}(\mathsf{mul}(2,x) \mid x, f_0 : \mathsf{has}(g_0, f_0) \wedge \mathsf{petals}(f_0, x)))$

The satisfaction of an extended condition tree $\boldsymbol{X}$ by a morphism g is determined by a minimal proof $\boldsymbol{P}$ of the condition tree $\boldsymbol{C}$ underlying $\boldsymbol{X}$ which should satisfy conditions (i)–(iv) in Sect. 3.4 and in addition

(v) For all nodes P in $\boldsymbol{P}$ and $x \in N_P$ such that $\tau_{\pi(P)}(x) = \langle \mathbb{O}, t, c \rangle$, if $\Gamma \subseteq out_{\boldsymbol{P}}(P)$ covers c then $\gamma_P(x) = F^{\mathbb{O}}|\gamma_{cod(p)}(t)p \in \Gamma$.

In other words, to evaluate a set-based operator node x in a given proof tree node P, where x is labelled by $\tau_{\pi(P)}$ as $\langle \mathbb{O}, t, c \rangle$, we collect the outgoing morphisms of P that cover (i.e., prove) the $\forall$-quantified condition morphism c, and for all of these we look up the value of t in the concrete host graph. The concrete operation $F^{\mathbb{O}}$ is applied to the resulting (multi)set. Thus, x effectively encodes the term $\mathbb{O}(t \mid X : \phi_{\boldsymbol{X}[cod(c)]})$, where $X = \xi(N_{cod(c)} \setminus N_P)$ is the set of variables fresh in $cod(c)$.

Example 7. Figure 7 shows a proof tree for the extended condition tree of Fig. 6. The two children of the root, P_1 and P_2, assign n_3, resp. n_4 to f_0, and accordingly, 5 resp. 2 to x; the term $\mathsf{mul}(2,x)$ hence evaluates to 10, resp. 4. Now condition (v) above kicks in, emposing the constraint

$$\gamma_0(p) = F^{\mathsf{sum}}\{\gamma_1(\mathsf{mul}(2,x)), \gamma_2(\mathsf{mul}(2,x))\} = F^{\mathsf{sum}}\{10, 4\} = 14.$$

It is interesting to note that where proof trees are ordinarily built from parents to children, condition (v) has a dependency in the other direction: the value of a set-based operation can only be computed after a covering set of children has been established.

This is the core ingredient in the following main theorem of this paper, extending Theorem 1, which we present here without proof.

Theorem 2 (extended condition tree equivalence).

1. *For every* ***SBOL*** *formula ϕ, there is an equivalent extended condition tree $\boldsymbol{X}_\phi$;*
2. *For every extended condition tree $\boldsymbol{X}$, there is an equivalent* ***SBOL*** *formula $\phi_{\boldsymbol{X}}$.*

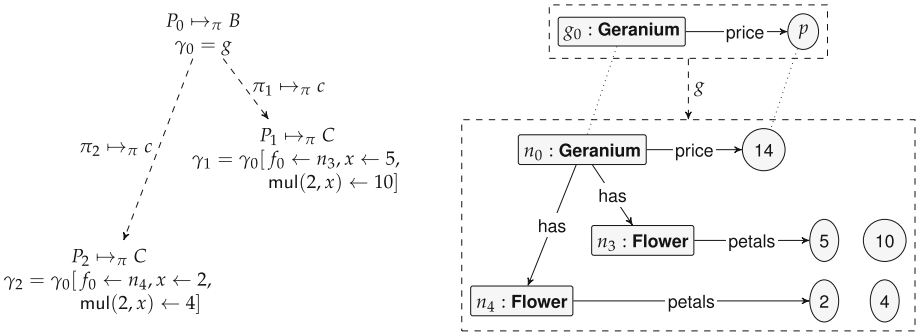


Fig. 7. Proof for the extended condition tree of Fig. 6

Both directions can be proved by induction, on (respectively) the structure of ***SBOL*** formulas and the depth of extended condition trees. The interesting case for Clause 1 is, obviously, how to deal with terms of the form $\mathbb{O}(t \mid X : \phi)$: this requires the introduction of a $\forall$-labelled child which a τ-mapping. Vice versa, for Clause 2, every τ-mapping gives rise to an $\mathbb{O}$-term.

5 The Geranium Experiment

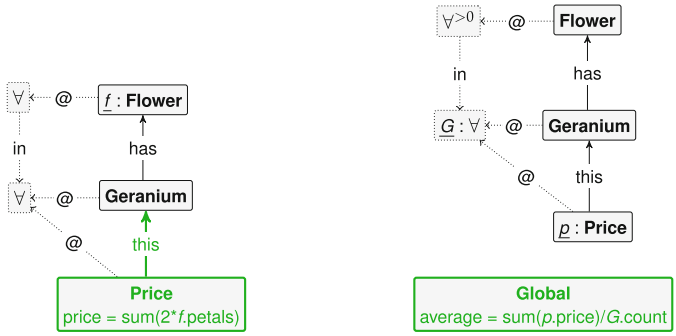
The original motivation of this paper was not to develope a new theoretical concept but to extend the existing tool GROOVE [11] with a feature allowing the use of set-based operators. We will now illustrate the capabilities of the tool. It should be noted that, because the implementation preceded the theoretical foundation exposed in this paper, notations in the tool are not identical to the ones in the previous sections.

5.1 Two-Step Computation

Figure 8 shows two rules in GROOVE that specify, successively, the simultaneous computation of the price of all geraniums in the world, and the computation of the average price of geraniums with at least one flower. The applicability condition of these rules is given by condition trees with set-based operators.

The figure shows two nested rules in GROOVE syntax, the complete explanation of which is out of scope here. An important aspect is that the condition tree is flattened in that all graphs are combined; the structure of the tree is recovered through quantifier nodes.

- The root of the condition tree is always the empty graph—i.e., condition trees are always closed;
- The root has only a single child, called the *base*, which is existentially quantified and consists of all nodes without @-connector to any quantifier node, plus all edges between them;



(a) Simultaneous computation of individual geranium prices

(b) Global average price of flowering geraniums

Fig. 8. Two-step computation of average geranium prices using multi-rules. (Color figure online)

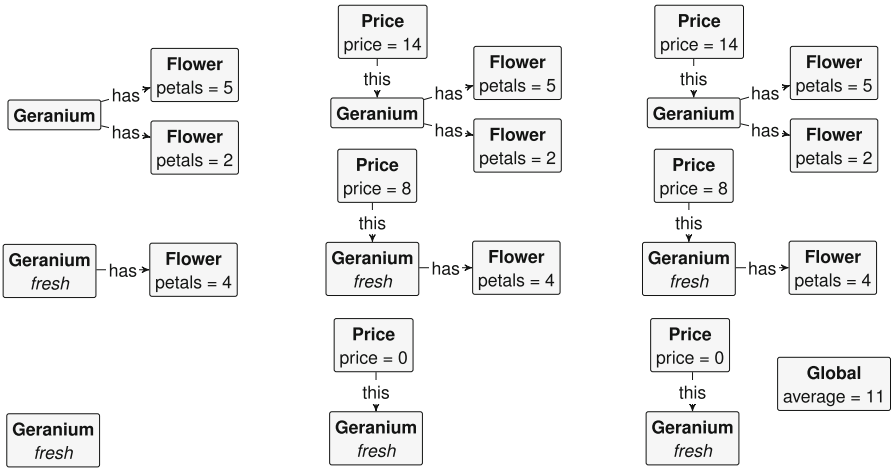
- Quantifier nodes point to their parents in the condition tree through in-connectors; quantifier nodes without outgoing in-connectors are children of the base;
- Except for the base, every level of the condition tree consists of the nodes linked by @-connectors to the corresponding quantifier node or one of its ancestors, plus all edges between them;
- As an additional feature, not formalised in Definition 5, one of the quantifier nodes in Fig. 8b is labelled $\forall^{>0}$ rather than $\forall$. This indicates that, in a proof tree, a covering of this level should contain at least one element (corresponding to the fact that, in (1), we only want the average price of flower-bearing geraniums).

The green nodes and edges in Fig. 8 (the **Price**-node in Fig. 8a and the **Global**-node in Fig. 8b) are *created* when the rule is applied; if, as in the case of Fig. 8a, there is also a @-connector from such a creator node to a quantifier, that means an instance is that node is created for every match of the corresponding condition tree level.

The computation of the average price consists of applying these two rules in succession. Figure 9 shows an example computation, starting with the host graph in Fig. 9a; the application of the rule in Fig. 8a results in Fig. 9b, after which the rule in Fig. 8b results in Fig. 9c.

5.2 One-Step Computation

The above solution still does not actually express the whole expression (1) in a single go: instead, the function *price* is first computed, then used. A single rule that encodes the entire condition is given in Fig. 10.



(a) Start state

(b) Intermediate state (price computed)

(c) Final state (average computed)

Fig. 9. Example average price computation

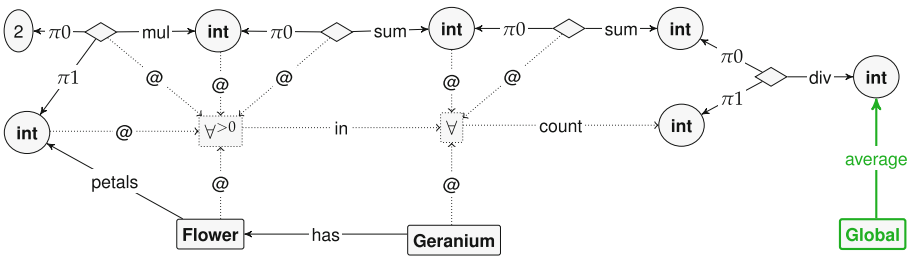


Fig. 10. One-step computation of average geranium prices

This rule uses more primitive syntax for expressions, which requires some further explanation.[2] Elliptical nodes are data nodes, which can be either data values or variables. Diamond-shaped ones correspond to operations: their outgoing π_i-labelled edges collect a list of arguments, and the remaining outgoing edge applies an operator to that list, depositing the result in its target node. Examples of such operators are: mul, which stands for multiplication (in this case, the multiplication of 2 to the number of petals of a **Flower**), 2 instances of sum, which is the set-based summation, and div, which divides the summed-up geranium prices by the number of geraniums involved—the latter being made available through an outgoing count-edge of the universally quantified level.

[2] The reason for reverting to more primitive syntax is simply that the GROOVE's expression parser cannot yet cope with the nested set-based operators used in this rule.

In terms of Fig. 9, the rule in Fig. 10 can be applied to the start graph Fig. 9a, directly leading to a graph that corresponds to Fig. 9c without the **Price**-labelled nodes.

5.3 Iterative Computation

In the absence of set operators, one would have to encode the computation of the individual geranium prices as well as their average price using a sequence of simple rules that iteratively add up the numbers of petals of each geranium to get the correct value of *price*, and the prices and count of the flowering geraniums to compute the average. Besides requiring more rules as well as a way to schedule them, this also introduces bookkeeping into the graph to keep track of which flowers or geraniums were already counted. We do not show the solution here, as this would entail explaining much more about the GROOVE tool; however, it is packaged together with the others and available online.[3]

5.4 Experimental Results

Table 1 reports the performance of the three implemented solutions: one-step (Sect. 5.2), two-step (Sect. 5.1) and iterative (Sect. 5.3). The computation has been applied on host graphs ranging in size from 60 nodes (essentially 10 disjoint copies of the start graph in Fig. 9a) to 600 nodes (100 disjoint copies of that same graph), by running the relevant rules 1000 times and taking the average time for a single computation. GROOVE is written in Java; to ged rid of some of the known issues in measuring the performance of Java programs, we started all experiments with a "warm-up run" to allow the Just-In-Time compilation to kick in.

Table 1. Performance of the geranium experiment

Graph size (#nodes)	One-step (ms)	Two-step (ms)	Iterative (ms)
60	0.59	0.72	5
120	0.99	1.29	17
180	1.32	1.57	37
240	1.44	2.40	67
300	1.48	2.90	108
360	1.71	3.18	140
420	1.96	3.52	190
480	1.90	3.99	249
540	2.11	4.64	316
600	2.39	5.15	395

[3] See groove.cs.utwente.nl/downloads/grammars/.

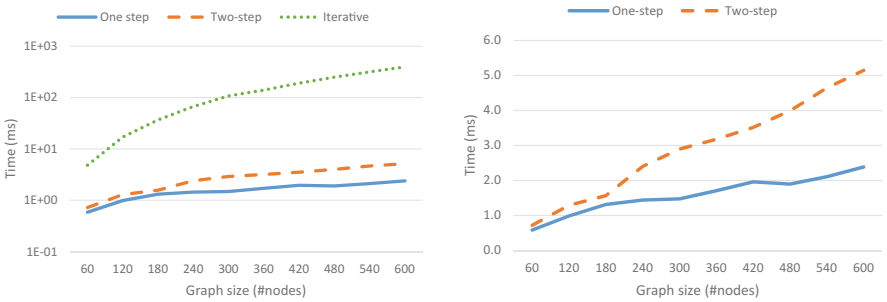


Fig. 11. Relative performance and scaling of the three implemented solutions

Even so, the precise run-time figures deviate within a margin of about 10% when repeating the same experiment, without, however, affecting the qualitative outcome and conclusions in any meaningful way.

The experiment has been conducted on a laptop with an Intel i5-6300U CPU running at 2.40 GHz, using Java 8 with sufficient memory to avoid major garbage collections. The GROOVE rule system used is available online and can be run in the newest version of the tool GROOVE.[4]

The results are shown graphically in Fig. 11. The left hand side compares all three approaches on a logarithmic scale, the second is limited to the one-step and two-step solutions that use the set operators and uses a linear time scale.

5.5 Evaluation

It can be seen from the data provided that not only are the one-step and two-step, set operator-based solutions several orders of magnitude faster than the iterative solution, they also scale much better: on the sample size of our experiment, the trend seems linear for the set operator-based solutions whereas the degradation in performance is clearly worse for the iterative solution.

A second observation is that the one-step solution performs better than the two-step solution, even at small problem sizes; the difference becomes more pronounced at larger sizes, seeing that the slope of the approximately linear trend is shallower for the former.

Both of these observations can be explained by a superficial analysis of the run-time effort involved. There are two effects in play when the graph size grows: matching becomes harder and transformation sequences become longer.

– For the set operator-based solutions: the *number* of matches remains the same (all rules have exactly 1 match) but the *size* of that match—which is nothing else than a proof tree—grows linearly. The length of the transformation sequence is not affected: it is always 1 for the one-step solution and 2 for the two-step solution. All in all, it makes sense that the running time of the computation increases linearly with the size of the host graph.

[4] See sf.net/projects/groove.

– For the iterative solution: here the *numer* of matches grows linearly with the size of the host graph, but the *size* of each match is constant. Since (for the purpose of this experiment) the exploration is set to linear (meaning that there is no backtracking in the exploration of the rule system), only 1 of the n matches is selected each time; however, the total number of steps grows linearly in the size of the graph. Concretely, in our solution, computing the solution for the largest graph (size 600) takes 703 steps. All in all, based on these observations, one may expect the running time to increase (at least) quadratically with the size of the host graph.

Besides a difference in performance, there is also a clear difference in conciseness of our three solutions: the single rule of Fig. 10 is (somewhat) smaller than the two rules of Fig. 8, whereas our iterative solution consists of 5 (smaller) rules plus a control program that schedules them. Moreover, all solutions except the one-step need to add elements for the graph for bookkeeping purposes: in the case of the two-step solution, this consists of the **Price**-nodes created by Fig. 8a and used by Fig. 8b, whereas the iterative solution does not only use such **Price**-nodes but also counted-markers for those flowers and geraniums that have already been taken into account.

Finally, we claim that there is also a difference in understandability. Though the primitive syntax of Fig. 10 is not ideal, this is a matter of supporting further syntactic sugar. The main difficulty in understanding the set-operator based solutions lies in the concept of condition trees, which can admittedly be tricky in practice, but once mastered is (to our opinion) quite usable. In contrast, the iterative solution requires an understanding of the way the 5 smaller rules work together, which is far from straightforward.

6 Conclusion

To summarise: the contribution reported in this paper consists of

– The extension of nested graph conditions to set-based operators such as sum, product and cardinality, increasing their expressiveness beyond first-order logic;
– The extension of first-order logic to set-based operator logic ***SBOL***, which we claim to be expressively equivalent to the extended graph conditions;
– The implementation of set-based operators in GROOVE, leading to a clear gain in performance, conciseness and understandability in rule systems where set-based operators play a major role, as illustrated by a single case study.

6.1 Related Work

Within graph transformation, patterns and techniques to specify multi-rules have been studied for some time, leading to concepts such as *star operators* [19], *subgraph operators* [3], *collection operators* [13], *cloning rules* [15] and *set-valued transformations* [10]. More recently, *pattern rewriting* [17] has been developed

and applied in the context of chemical reactions [1]. We believe that all these concepts can be seen as special forms of amalgamation as proposed first in [26] and generalised later in [12], as can our own nested rules from [24,25]. However, the treatment of set-based operators presented in this paper, though inspired by the mechanisms of amalgamation, goes beyond it in expressiveness.

On the tool front, many of the graph transformation tools that are currently being maintained support some form of multi-rules. Examples are FUJABA [21] which features *set nodes*, and HENSHIN [2] which knows the concept of an *amalgamation unit*. Again, we believe that these are limited to (essentially) the first-order level and cannot directly express set-based operators.

It should also be recognised that there are alternative ways to achieve the effect of multi-rules. For instance, VIATRA2 [4] allows the specification of recursive patterns through the control language, and FUJABA can specify some degree of parallel rule application through *storyboards*.

6.2 Future Work

In the theoretical exposition in this paper, we have kept things simple wherever we could. In particular, we have restricted our algebra to integers only, and our condition morphisms to injective ones. On the first count, we foresee no difficulty to extend to other datatypes, using multi-sorted algebras to ensure well-typedness. On the second count, we have already shown in [23] that allowing condition morphisms to be non-injective corresponds to introducing equality as a basic predicate in the logic; we have no reason to believe that the same correspondence fails to hold in the extended case of this paper.

On the logic side, it would be interesting to know whether ***SBOL*** as introduced in (10) actually corresponds to a known fragment of logic. Clearly it is well within monadic second-order logic, since all that can be done with sets in ***SBOL*** is applying set-based operators to them; this is far less than the ability to use sets as first-class values. As one reviewer suggested, the fact that the effect of set-based operators can be mimicked by an iterative solution justifies a hypothesis that the introduction of fixpoints of some kind into ***FOL*** may be sufficient to cover ***SBOL***—although we feel that this will quite likely be more powerful, maybe even quite a bit so.

On the implementation side, the support of set-based operators in GROOVE can certainly be further improved, especially by providing further syntactic sugar for nested set-based operators as used in Fig. 10 (see Footnote 2). It should be noted that GROOVE does support other datatypes besides integers, and also supports non-injectivity in condition morphisms.

Acknowledgement. My scientific career got underway under the inspiring supervision of Ed Brinksma, who instilled and shared a fascination with the maths behind it all, without losing sight of intuition and pragmatics. Even if I do remember throwing a frustrated pen at him at one occasion, I am glad to have had Ed as mentor and friend.

References

1. Andersen, J.L., Flamm, C., Merkle, D., Stadler, P.F.: Inferring chemical reaction patterns using rule composition in graph grammars. CoRR abs/1208.3153 (2012). http://arxiv.org/abs/1208.3153
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16145-2_9
3. Balasubramanian, D., Narayanan, A., Neema, S., Shi, F., Thibodeaux, R., Karsai, G.: A subgraph operator for graph transformation languages. In: Ehrig and Giese [9]. http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/72
4. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Haddad, H. (ed.) Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), pp. 1280–1287. ACM (2006)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006). doi:10.1007/3-540-31188-2
6. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.: Theory of constraints and application conditions: from graphs to high-level structures. Fundam. Inform. **74**(1), 135–166 (2006). http://content.iospress.com/articles/fundamenta-informaticae/fi74-1-07
7. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification, Part I and II. Monographs in Theoretical Computer Science. An EATCS Series, vols. 6 and 21. Springer, Heidelberg (1985, 1990). doi:10.1007/978-3-642-69962-7, doi:10.1007/978-3-642-61284-8
8. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: an algebraic approach. In: Symposium on Switching and Automata Theory, pp. 167–180. IEEE Computer Society (1973). https://doi.org/10.1109/SWAT.1973.11
9. Ehrig, K., Giese, H. (eds.): Graph Transformation and Visual Modeling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 6 (2007)
10. Fuss, C., Tuttlies, V.E.: Simulating set-valued transformations with algorithmic graph transformation languages. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 442–455. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89020-1_30
11. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. STTT **14**(1), 15–40 (2012)
12. Golas, U., Ehrig, H., Habel, A.: Multi-amalgamation in adhesive categories. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 346–361. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15928-2_23
13. Grønmo, R., Krogdahl, S., Møller-Pedersen, B.: A collection operator for graph transformation. Softw. Syst. Model. **12**(1), 121–144 (2013). doi:10.1007/s10270-011-0190-3
14. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. **26**(3/4), 287–313 (1996). doi:10.3233/FI-1996-263404
15. Hoffmann, B., Janssens, D., Eetvelde, N.V.: Cloning and expanding graph transformation rules for refactoring. In: Karsai, G., Taentzer, G. (eds.) Graph and Model Transformation (GraMoT). Electronic Notes in Theoretical Computer Science, vol. 152, pp. 53–67 (2006). https://doi.org/10.1016/j.entcs.2006.01.014

16. Junges, S., Guck, D., Katoen, J., Rensink, A., Stoelinga, M.: Fault trees on a diet: automated reduction by graph rewriting. Formal Asp. Comput. **29**(4), 651–703 (2017). doi:10.1007/s00165-016-0412-0
17. Kissinger, A., Merry, A., Soloviev, M.: Pattern graph rewrite systems. In: Löwe, B., Winskel, G. (eds.) International Workshop on Developments in Computational Models. EPTCS, vol. 143, pp. 54–66 (2014)
18. Kreowski, H.-J.: A comparison between petri-nets and graph grammars. In: Noltemeier, H. (ed.) WG 1980. LNCS, vol. 100, pp. 306–317. Springer, Heidelberg (1981). doi:10.1007/3-540-10291-4_22
19. Lindqvist, J., Lundkvist, T., Porres, I.: A query language with the star operator. In: Ehrig and Giese [9]. http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/55
20. LOTOS: A formal description technique based on the temporal ordering of observational behaviour. ISO/IEC Infernational Standard 8807, International Organization for Standardization (1989). https://www.iso.org/standard/16258.html
21. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) 22nd International Conference on on Software Engineering (ICSE), pp. 742–745. ACM (2000)
22. Plump, D.: Essentials of term graph rewriting. In: GETGRATS Closing Workshop. Electronic Notes in Theoretical Computer Science, vol. 51, pp. 277–289 (2001). https://doi.org/10.1016/S1571-0661(04)80210-X
23. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30203-2_23
24. Rensink, A.: Nested quantification in graph transformation rules. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 1–13. Springer, Heidelberg (2006). doi:10.1007/11841883_1
25. Rensink, A., Kuperus, J.: Repotting the geraniums: on nested graph transformation rules. In: Boronat, A., Heckel, R. (eds.) Graph Transformation and Visual Modeling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 18 (2009). http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/260
26. Taentzer, G.: Parallel high-level replacement systems. Theor. Comput. Sci. **186**(1–2), 43–81 (1997). doi:10.1016/S0304-3975(96)00215-0