

A SAT-Based Graph Rewriting and Verification Tool Implemented in Haskell

Marcus Ermler

University of Bremen, Department for Mathematics and Computer Science

WFLP 2013, September 12, 2013

Motivation

Main motivation: Tool support in graph rewriting

Aim: A tool for graph rewriting and verification

Questions:

1. How to tackle the nondeterminism of graph rewriting, especially in case of NP-complete graph problems?
2. What could be a useful programming language in this context?

Answers:

1. heuristics, exhaustive search, parallelization, **SAT solving**
⇒ chip design, term rewriting, UML/OCL models
2. Java, C++, Python, **Haskell**
⇒ formulas, graphs, and rules are near to their mathematical description

Tool history

- translation of graph transformational derivation process into propositional formulas (presented on ICGT 2010)
- first implementation in the author's diploma thesis (2010)
- introducing SATaGraT (**SAT** solver **assists** **G**raph **T**ransformation Engine) on AGTIVE 2011
- **today**: three processing steps, verification of WFLP2013a, first steps to translations into CSP and SMT, more examples

SATaGraT - main components

- **Graph rewriting:** Modules for graphs, graph morphisms, rules, control conditions, and graph transformation units
- **Propositional formulas:** Three different translations
 - ICGT 2010
 - AGTIVE 2011
 - WFLP 2013plus first steps for translations into CSP and SMT
- **Solvers:** SAT solvers MiniSat, Limboole, and Funsat; CSP solver Sugar; SMT solver Yices
- **Verification:** existentially quantified graph properties and all quantified properties over terms
- **Examples:** Hamiltonian path problem, job-shop scheduling, ...

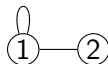
Graphs

- edge labeled directed graphs without multiple edges and with a finite node set over a set Σ of labels: $G = (V, E)$ where $V = \{1, \dots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$
- injective graph morphisms $g: G \rightarrow H$ for matching of subgraphs (structure- and label-preserving)
 \Rightarrow these morphisms are injective mappings between the node sets of G and H : $g_V: V_G \rightarrow V_H$

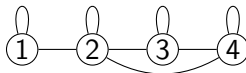
Graphs

- edge labeled directed graphs without multiple edges and with a finite node set over a set Σ of labels: $G = (V, E)$ where $V = \{1, \dots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$
- injective graph morphisms $g: G \rightarrow H$ for matching of subgraphs (structure- and label-preserving)
 \Rightarrow these morphisms are injective mappings between the node sets of G and H : $g_V: V_G \rightarrow V_H$

member

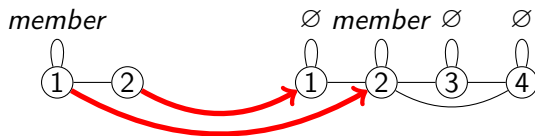


\emptyset *member* \emptyset \emptyset



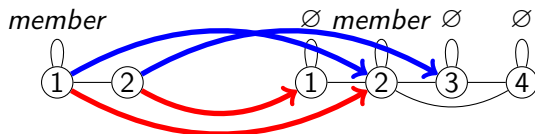
Graphs

- edge labeled directed graphs without multiple edges and with a finite node set over a set Σ of labels: $G = (V, E)$ where $V = \{1, \dots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$
- injective graph morphisms $g: G \rightarrow H$ for matching of subgraphs (structure- and label-preserving)
 \Rightarrow these morphisms are injective mappings between the node sets of G and H : $g_V: V_G \rightarrow V_H$



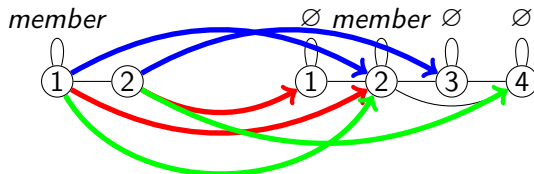
Graphs

- edge labeled directed graphs without multiple edges and with a finite node set over a set Σ of labels: $G = (V, E)$ where $V = \{1, \dots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$
- injective graph morphisms $g: G \rightarrow H$ for matching of subgraphs (structure- and label-preserving)
 \Rightarrow these morphisms are injective mappings between the node sets of G and H : $g_V: V_G \rightarrow V_H$



Graphs

- edge labeled directed graphs without multiple edges and with a finite node set over a set Σ of labels: $G = (V, E)$ where $V = \{1, \dots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$
- injective graph morphisms $g: G \rightarrow H$ for matching of subgraphs (structure- and label-preserving)
 \Rightarrow these morphisms are injective mappings between the node sets of G and H : $g_V: V_G \rightarrow V_H$

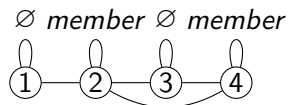
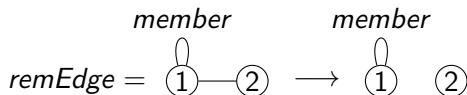


Rule application

- $r = (L \rightarrow R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G : find a match $g(L)$ in G . If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- **rule application**: $G \xRightarrow[r,g]{} H$

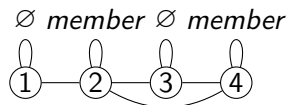
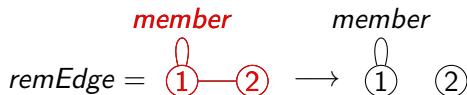
Rule application

- $r = (L \rightarrow R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G : find a match $g(L)$ in G . If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- **rule application:** $G \xRightarrow[r,g]{} H$



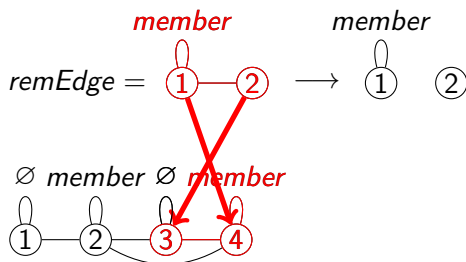
Rule application

- $r = (L \rightarrow R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G : find a match $g(L)$ in G . If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- **rule application:** $G \xrightarrow[r,g]{} H$



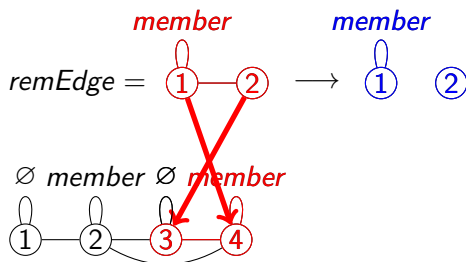
Rule application

- $r = (L \rightarrow R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G : find a match $g(L)$ in G . If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- **rule application:** $G \xrightarrow[r,g]{} H$



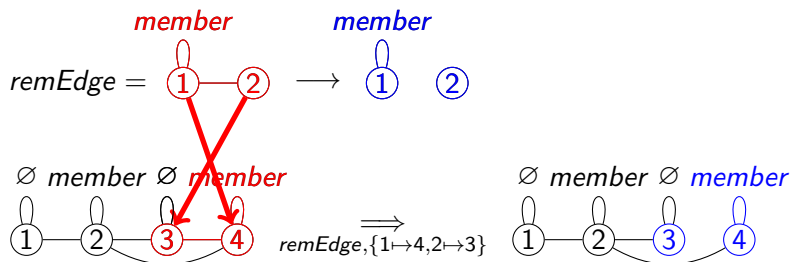
Rule application

- $r = (L \rightarrow R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G : find a match $g(L)$ in G . If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- **rule application:** $G \xRightarrow[r,g]{} H$



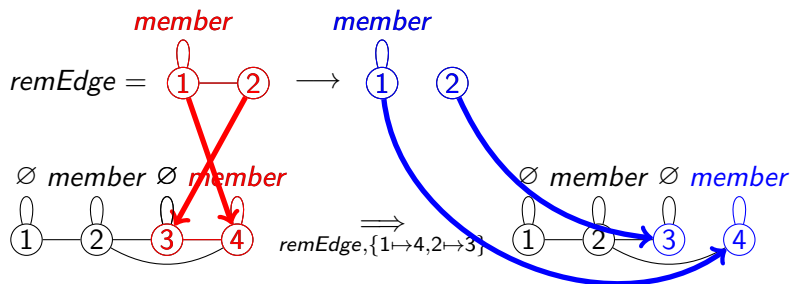
Rule application

- $r = (L \rightarrow R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G : find a match $g(L)$ in G . If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- **rule application:** $G \xRightarrow[r,g]{} H$



Rule application

- $r = (L \rightarrow R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G : find a match $g(L)$ in G . If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- rule application: $G \xRightarrow[r,g]{} H$

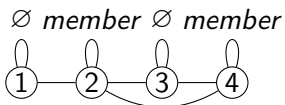


Derivation

- $d = G_0 \xRightarrow[r_1, g_1]{\quad} G_1 \xRightarrow[r_2, g_2]{\quad} \cdots \xRightarrow[r_n, g_n]{\quad} G_n$ is called a **derivation**
- $G_0 \xRightarrow[P]{*} G_n$

Derivation

- $d = G_0 \xRightarrow[r_1, g_1]{\quad} G_1 \xRightarrow[r_2, g_2]{\quad} \cdots \xRightarrow[r_n, g_n]{\quad} G_n$ is called a **derivation**
- $G_0 \xRightarrow[P]{*} G_n$



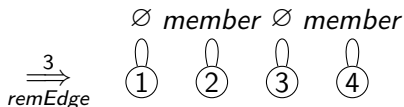
Derivation

- $d = G_0 \xRightarrow[r_1, g_1]{ } G_1 \xRightarrow[r_2, g_2]{ } \cdots \xRightarrow[r_n, g_n]{ } G_n$ is called a **derivation**
- $G_0 \xRightarrow[P]{*} G_n$



Derivation

- $d = G_0 \xRightarrow[r_1, g_1]{} G_1 \xRightarrow[r_2, g_2]{} \cdots \xRightarrow[r_n, g_n]{} G_n$ is called a **derivation**
- $G_0 \xRightarrow[P]{*} G_n$



Graph transformation units

- **graph transformation units**: $gtu = (I, P, C, T)$ where I and T are graph class expressions, R is a set of rules, and C is a control condition
- **graph class expressions**: for example, the class of all undirected graphs, also single graphs allowed
- **control conditions**: guide the rule application, restrict the nondeterminism of units; we use regular expressions
- **Semantics** of $gtu = (I, P, C, T)$: all derivations from initial to terminal graphs that are allowed by the control condition \Rightarrow such derivations are called *successful*

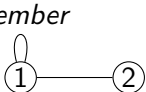
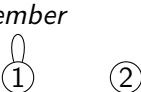
Graph rewriting for graph problems

VertexCover(k)

initial: *unlabeled & undirected & \emptyset – loops*

rules:

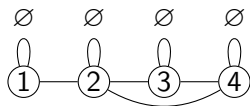
choose :  \rightarrow 

remEdges :  \rightarrow 

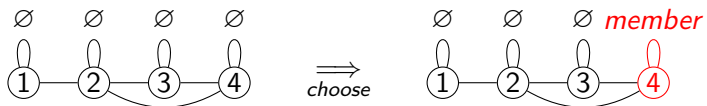
cond.: *choose^k ; remEdges**

terminal: *no_edges & (member | \emptyset) – loops*

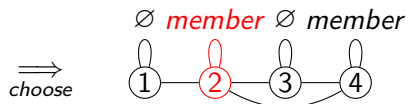
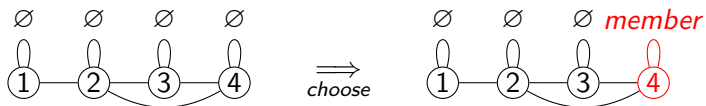
Derivation revisited



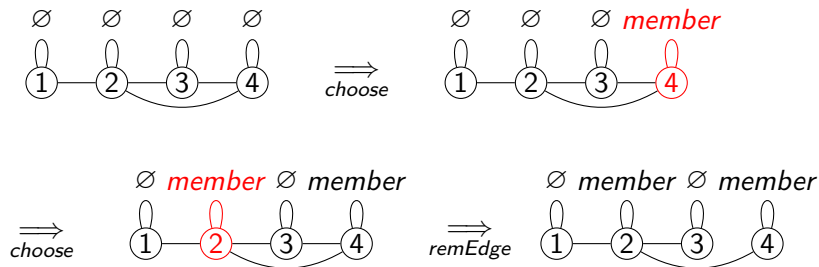
Derivation revisited



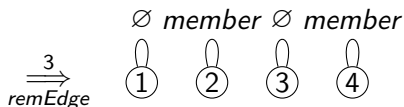
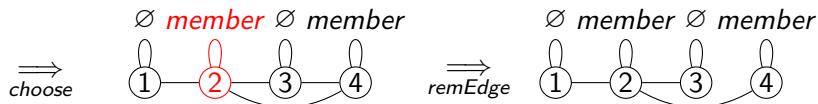
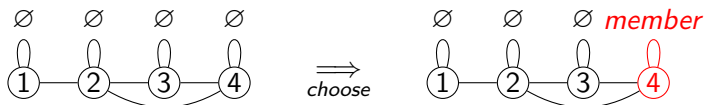
Derivation revisited



Derivation revisited



Derivation revisited



From graphs to SAT

- graphs in derivation steps are represented via variables for their edges: $E(n, m) = \{edge(v, a, v', k) \mid (v, a, v') \in [n] \times \Sigma \times [n], k \in [m]\}$ where n is the graph size and m the maximum derivation step

Theorem

Let p be a formula over $E(n, m)$ and f a satisfying assignment to p . Then $f(p)$ represents a sequence of graphs G_1, \dots, G_m such that G_k contains (v, a, v') if and only if $f(edge(v, a, v', k)) = \text{TRUE}$.

- single graph in the k th derivation step expressed via edges that are **in** the graph and edges that are **not in** the graph

$$\text{graph}(G)(k) = \bigwedge_{(v,a,v') \in E_G} edge(v, a, v', k) \wedge \bigwedge_{(v,a,v') \in ([n] \times \Sigma \times [n]) - E_G} \neg edge(v, a, v', k).$$

From graph rewriting to SAT (1)

The application of a rule r to a graph G_{k-1} with respect to a mapping g is expressed via

■ **matching**: $\text{morph}(r, g, k) = \bigwedge_{(v,a,v') \in E_L} \text{edge}(g(v), a, g(v'), k-1),$

■ **edge deletion**: $\text{rem}(r, g, k) = \bigwedge_{(v,a,v') \in E_L - E_R} \neg \text{edge}(g(v), a, g(v'), k),$

■ **edge addition**: $\text{add}(r, g, k) = \bigwedge_{(v,a,v') \in E_R} \text{edge}(g(v), a, g(v'), k),$

■ **kept edges**:

$$\text{keep}(r, g, k) = \bigwedge_{(v,a,v') \notin g(E_L \cup E_R)} (\text{edge}(v, a, v', k-1) \leftrightarrow \text{edge}(v, a, v', k))$$

where $g(E_L \cup E_R) = \{(g(v), a, g(v')) \mid (v, a, v') \in E_L \cup E_R\}$

\Rightarrow the assignment to variables of kept edges remains unchanged from G_{k-1} to G_k

From graph rewriting to SAT (2)

- whole rule application:

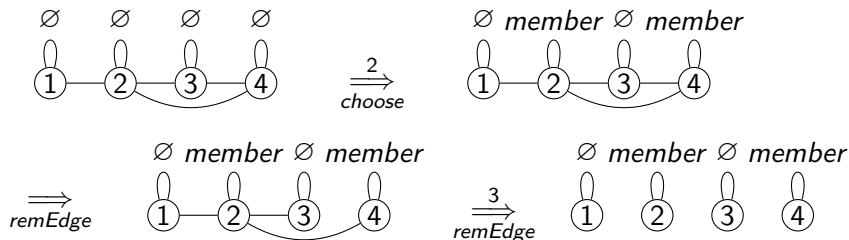
$$\text{apply}(r, g, k) = \text{morph}(r, g, k) \wedge \text{rem}(r, g, k) \wedge \text{add}(r, g, k) \wedge \text{keep}(r, g, k)$$

Theorem

$G_{k-1} \xRightarrow[r, g]{} G_k$ if and only if there is a satisfying assignment to the formula $\text{graph}(G_{k-1})(k-1) \wedge \text{apply}(r, g, k) \wedge \text{graph}(G_k)(k)$.

- further formulas for derivation steps, single derivations, and all derivations up to a certain bound

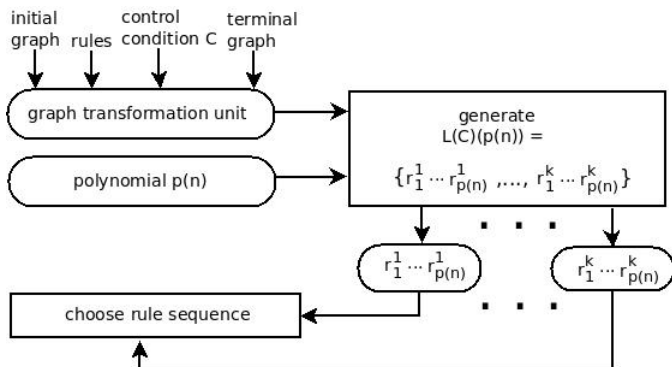
From graph rewriting to SAT (3)



is yielded by a satisfying assignment to:

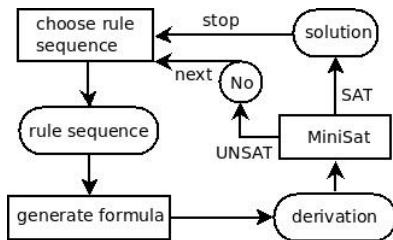
$$\begin{aligned} & \text{graph}(G_0)(0) \wedge \text{apply}(\text{choose}, \{1 \mapsto 4\}, 1) \wedge \text{apply}(\text{choose}, \{1 \mapsto 2\}, 2) \wedge \\ & \text{apply}(\text{remEdge}, \{1 \mapsto 4, 2 \mapsto 3\}, 3) \wedge \text{apply}(\text{remEdge}, \{1 \mapsto 2, 2 \mapsto 1\}, 4) \wedge \\ & \text{apply}(\text{remEdge}, \{1 \mapsto 2, 2 \mapsto 3\}, 5) \wedge \text{apply}(\text{remEdge}, \{1 \mapsto 4, 2 \mapsto 2\}, 6). \end{aligned}$$

SATaGraT - preprocessing



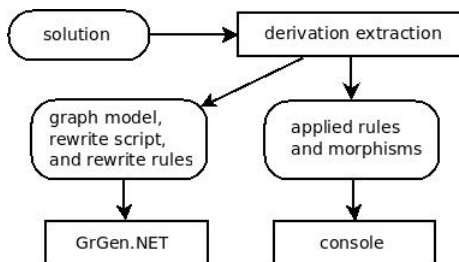
- $L(C)(p(n))$ denotes the language resulting from a control condition C with the restriction to a word length of $p(n)$
- each $r_1 \dots r_n \in L(C)(p(n))$ describes a sequence of rule applications from initial to terminal graphs.

SATaGraT - processing



- generates a formula in CNF
- **MiniSat** is a powerful, competitive, and award-winning SAT solver (<http://www.satcompetition.org/>)
- this process runs as long as no solution is found or all possible rule sequences are processed
- a satisfying assignment states a successful derivation

SATaGraT - postprocessing (1)



- derivation is extracted from the variable assignment
- GrGen.NET is used for visualization
- additional informations on console

SATaGraT - postprocessing (2)

The image shows a screenshot of the yComp Version 1.3.14 GUI. The interface includes a menu bar (File, Edit, View, Navigate, Layout, Help), a toolbar, a search field, and a tree view on the left. The main workspace displays a graph with nodes labeled \$0:N through \$9:N and edges labeled \$10:E through \$24:E. The right panel shows a terminal window with the following content:

```
agtime@agtIME: ~/satagrat
choose; choose; choose; renEdges; renEdges; renEdges; renEdges; renEdges; renEdges;
renEdges; renEdges; renEdges; renEdges; renEdges; renEdges; renEdges; renEdges
Vars: 25103
Clauses: 5151941
Time: 46,21 sec

Graph: {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}, {{(1,*),2}, (1,*),4}, (1,*),5}, (1,*),6}, (2,*),1}, (2,*),3}, (2,*),7}, (2,*),8}, (2,*),9}, (2,*),10}, (3,*),2}, (3,*),11}, (3,*),12}, (3,*),13}, (4,*),1}, (5,*),1}, (6,*),1}, (7,*),2}, (8,*),2}, (9,*),2}, (10,*),2}, (11,*),3}, (12,*),3}, (13,*),3}}. Types = {}

The following rules and graph morphisms are applied:
1. choose {{1,1}}
2. choose {{1,3}}
3. choose {{1,2}}
4. renEdges {{1,1}, (2,4)}
5. renEdges {{1,3}, (2,13)}
6. renEdges {{1,3}, (2,11)}
7. renEdges {{1,2}, (2,9)}
8. renEdges {{1,2}, (2,8)}
9. renEdges {{1,1}, (2,6)}
10. renEdges {{1,1}, (2,5)}
11. renEdges {{1,3}, (2,12)}
12. renEdges {{1,2}, (2,10)}
13. renEdges {{1,2}, (2,1)}

New edge "$10" of type "E" has been created from "$5" to "$0".
New edge "$11" of type "E" has been created from "$6" to "$1".
New edge "$12" of type "E" has been created from "$7" to "$1".
New edge "$20" of type "E" has been created from "$8" to "$1".
New edge "$21" of type "E" has been created from "$9" to "$1".
New edge "$22" of type "E" has been created from "$A" to "$2".
New edge "$23" of type "E" has been created from "$B" to "$2".
New edge "$24" of type "E" has been created from "$C" to "$2".
Executing Graph Rewrite Sequence... (CTRL+C for abort)
l>run
Debug started -- available commands are: (n)ext match, (d)etailed step, (s)tep, step (u)p, step (o)ut of loop, (r)un, toggle (b)reakpoints, toggle (c)hoicepoints, toggle (l)azy choice, show (v)ariables, print stack(t)race, (f)ull state and (a)abort (plus Ctrl+C for forced abort).
```

Experiments: vertex cover problem

$ V $	$ E $	k	VC?	SATaGraT 2011	SATaGraT 2012
7	8	2	no	5	5
9	12	2	no	30	32
11	14	4	yes	96	34.5
13	20	3	yes	366	112
13	18	3	no	357	456
15	24	3	yes	> 3600	438

- SATaGraT 2011 is based on ICGT 2010 and AGTIVE 2011
- SATaGraT 2012 is based on formulas of WFLP 2013

What about verification?

SATaGraT can be used to verify properties like

- Is the graph Eulerian?
- Is there a vertex cover of size k ?
- Is there a feasible schedule with a makespan of at most l for a job-shop instance?
- $\text{length } (xs ++ ys) \stackrel{?}{=} \text{length } xs + \text{length } ys$
- $\text{map } f \text{ } xs ++ \text{map } f \text{ } ys \stackrel{?}{=} \text{map } f \text{ } (xs ++ ys)$

We can verify existentially quantified properties over graphs and existentially or all quantified properties over terms.

Conclusion and Outlook

- SATaGraT is a SAT-based tool for graph rewriting and verification
- verification of existentially quantified graph properties and all quantified properties over terms

Qutlook:

- graphical user interface for input of graph transformation units and the final visualization
- proving all quantified properties over graphs
- termination and non-termination proofs for graph and term rewriting