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
   $\Rightarrow$ chip design, term rewriting, UML/OCL models
2. Java, C++, Python, Haskell
   $\Rightarrow$ 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 **a**ssists **Gra**ph **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 2013

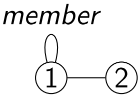
  plus first steps for translations into CSP and SMT
- Solvers: SAT solvers MiniSat, Limboole, and Funsat; CSP solver Sugar; SMT solver Yices
- Verification: existentially quantified graph properties and all quantified properties over terms
- Examples: Hamiltonian path problem, job-shop scheduling, ...

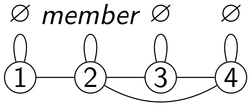
Graphs

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

Graphs

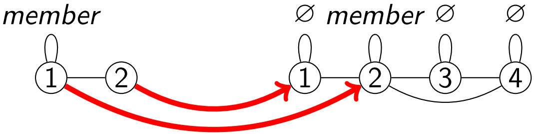
- edge labeled directed graphs without multiple edges and with a finite node set over a set Σ of labels: $G = (V, E)$ where $V = \{1, \ldots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$
- injective graph morphisms $g \colon G \to H$ for matching of subgraphs (structure- and label-preserving)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





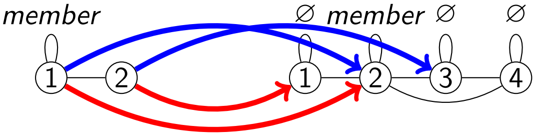
Graphs

- 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 $g\colon G \to H$ for matching of subgraphs (structure- and label-preserving)
  $\Rightarrow$ these morphisms are injective mappings between the node sets of G and H: $g_V\colon V_G \to V_H$



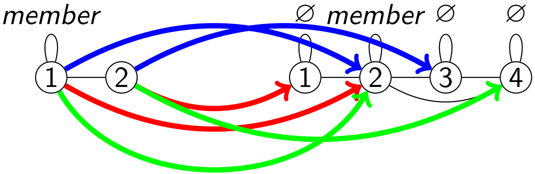
Graphs

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



Graphs

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

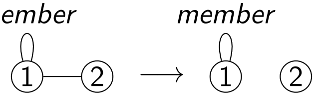


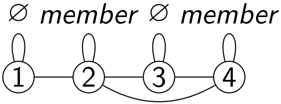
Rule application

- $r = (L \to R)$ 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: $G \underset{r,g}{\Longrightarrow} H$

Rule application

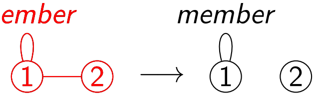
- $r = (L \to R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G: find a match $g(L)$ in G. If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- rule application: $G \underset{r,g}{\Longrightarrow} H$

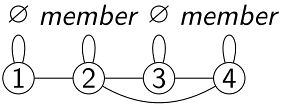
$remEdge =$ 



Rule application

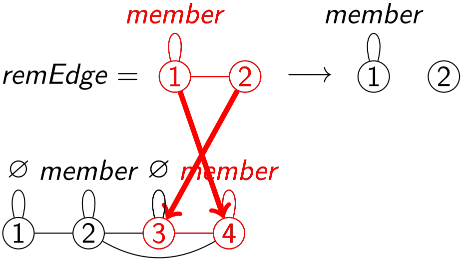
- $r = (L \to R)$ where $V_L = V_R$ (no node addition or deletion)
- rule application to a graph G: find a match $g(L)$ in G. If $g(L)$ is found, delete the edges of $g(L)$ and add the edges of $g(R)$.
- rule application: $G \underset{r,g}{\Longrightarrow} H$

remEdge =




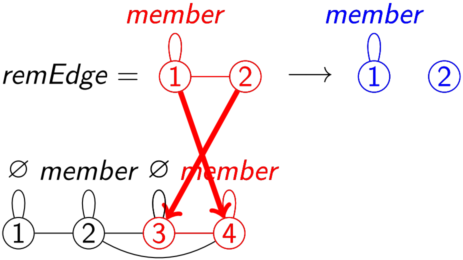
Rule application

- $r = (L \to R)$ 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: $G \underset{r,g}{\Longrightarrow} H$



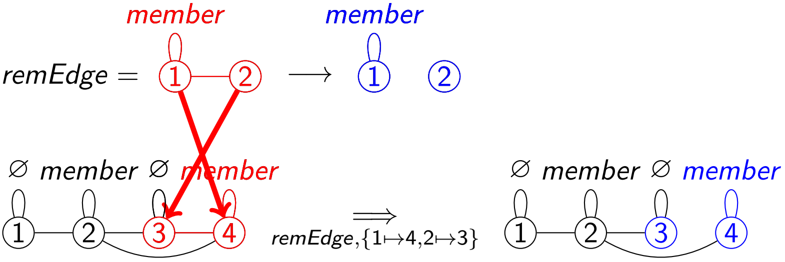
Rule application

- $r = (L \to R)$ 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: $G \underset{r,g}{\Longrightarrow} H$



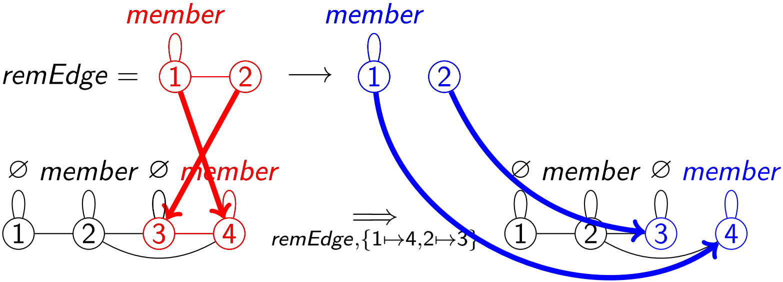
Rule application

- $r = (L \to R)$ 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: $G \underset{r,g}{\Longrightarrow} H$



Rule application

- $r = (L \to R)$ 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: $G \underset{r,g}{\Longrightarrow} H$

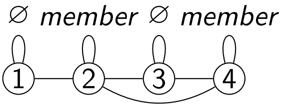


Derivation

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

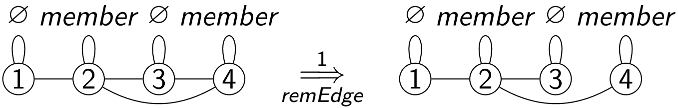
Derivation

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



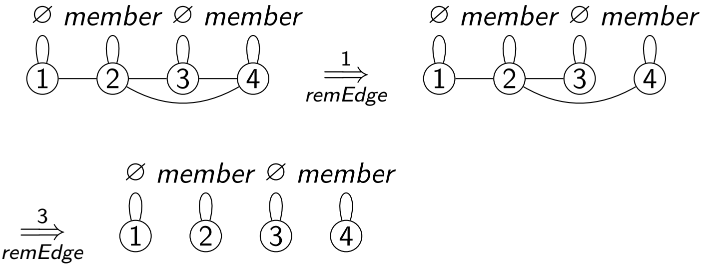
Derivation

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



Derivation

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



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

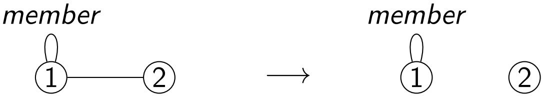
VertexCover(k)

initial: $unlabeled \& undirected \& \varnothing - loops$

rules:

choose :

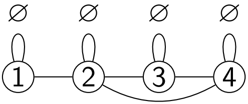
remEdges :



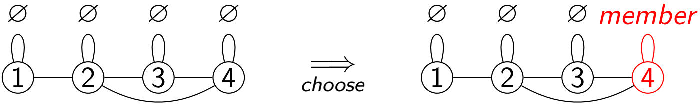
cond.: $choose^k \ ; \ remEdges^*$

terminal: $no\_edges \& (member \mid \varnothing) - loops$

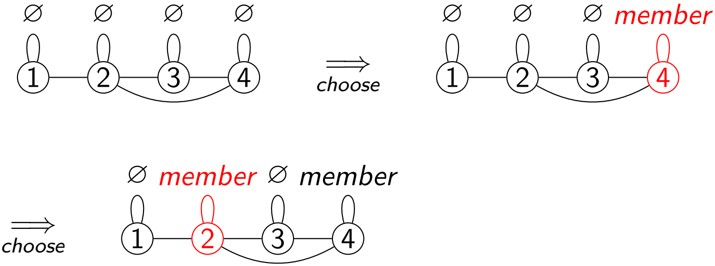
Derivation revisited



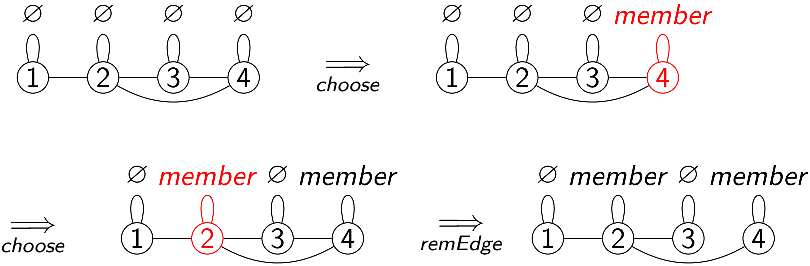
Derivation revisited



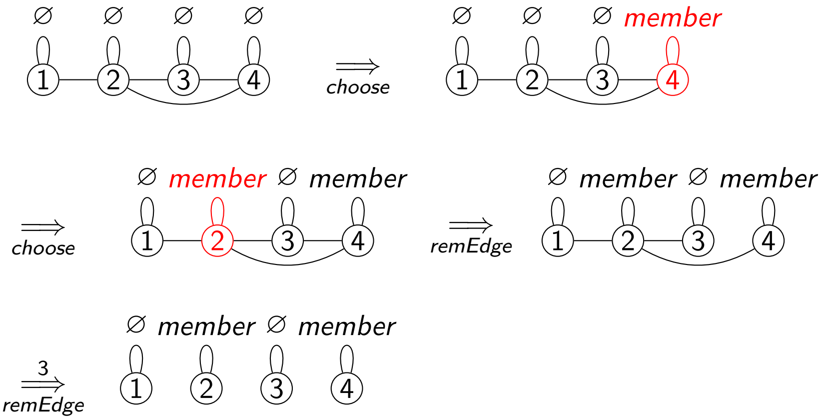
Derivation revisited



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 $G_1, \ldots, G_m$ such that G_k contains (v, a, v') if and only if $f(edge(v, a, v', k)) = TRUE$.

- single graph in the kth 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: $\mathrm{morph(r, g, k)} = \bigwedge_{(v,a,v') \in E_L} edge(g(v), a, g(v'), k-1),$
- edge deletion: $\mathrm{rem(r, g, k)} = \bigwedge_{(v,a,v') \in E_L - E_R} \neg edge(g(v), a, g(v'), k),$
- edge addition: $\mathrm{add(r, g, k)} = \bigwedge_{(v,a,v') \in E_R} edge(g(v), a, g(v'), k),$
- kept edges:
  $\mathrm{keep(r, g, k)} = \bigwedge_{(v,a,v') \notin g(E_L \cup E_R)} (edge(v, a, v', k-1) \leftrightarrow edge(v, a, v', k))$
  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)

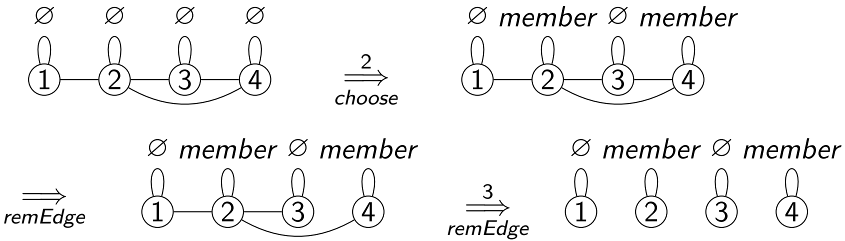
- whole rule application:
  $\mathrm{apply}(r, g, k) = \mathrm{morph}(r, g, k) \wedge \mathrm{rem}(r, g, k) \wedge \mathrm{add}(r, g, k) \wedge \mathrm{keep}(r, g, k)$

Theorem

$G_{k-1} \underset{r,g}{\Longrightarrow} G_k$ if and only if there is a satisfying assignment to the formula $\mathrm{graph}(\mathrm{G}_{\mathrm{k}-1})(\mathrm{k}-1) \wedge \mathrm{apply}(\mathrm{r}, \mathrm{g}, \mathrm{k}) \wedge \mathrm{graph}(\mathrm{G}_{\mathrm{k}})(\mathrm{k})$.

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

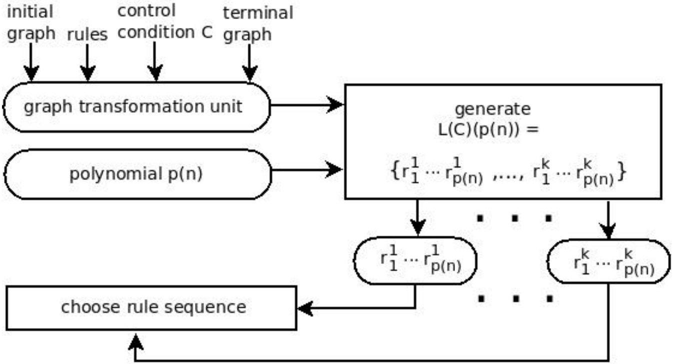
From graph rewriting to SAT (3)



is yielded by a satisfying assignment to:

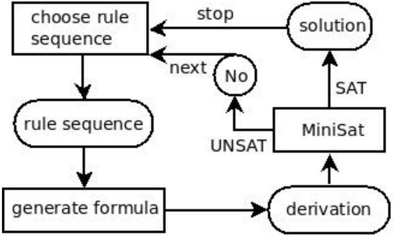
$\mathsf{graph}(G_0)(0) \wedge \mathsf{apply}(\mathsf{choose}, \{1 \mapsto 4\}, 1) \wedge \mathsf{apply}(\mathsf{choose}, \{1 \mapsto 2\}, 2) \wedge$
$\mathsf{apply}(\mathsf{remEdge}, \{1 \mapsto 4, 2 \mapsto 3\}, 3) \wedge \mathsf{apply}(\mathsf{remEdge}, \{1 \mapsto 2, 2 \mapsto 1\}, 4) \wedge$
$\mathsf{apply}(\mathsf{remEdge}, \{1 \mapsto 2, 2 \mapsto 3\}, 5) \wedge \mathsf{apply}(\mathsf{remEdge}, \{1 \mapsto 4, 2 \mapsto 2\}, 6).$

SATaGraT - preprocessing



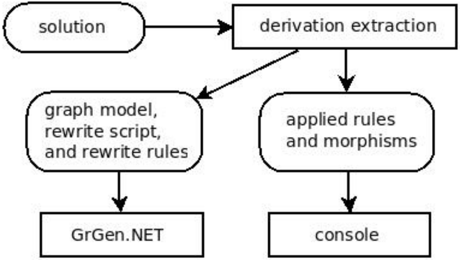
- $L(C)(p(n))$ denotes the language resulting from a control condition C with the restriction to a word length of $p(n)$
- each $r_1 \cdots r_n \in L(C)(p(n))$ describes a sequence of rule applications from initial to terminal graphs.

SATaGraT - processing



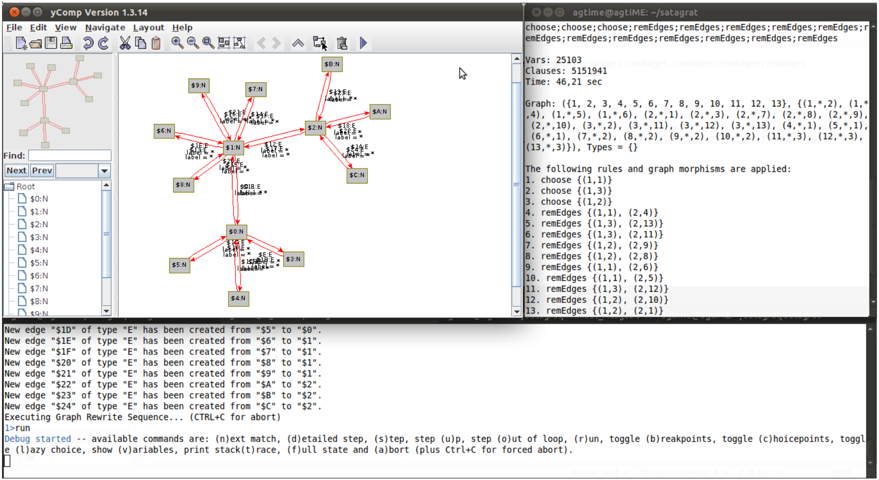
- generates a formula in CNF
- MiniSat is a powerful, competitive, and award-wining SAT solver (http://www.satcompetition.org/)
- this process runs as long as no solution is found or alle possible rule sequenes are processed
- a satisfying assignment states a successful derivation

SATaGraT - postprocessing (1)



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

SATaGraT - postprocessing (2)



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?
- `length (xs ++ ys)` $\stackrel{?}{=}$ `length xs + length ys`
- `map f xs ++ map f ys` $\stackrel{?}{=}$ `map f (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