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)


- introducing SATaGraT (SAT solver assists Graph Transformation 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, . . .
edge labeled directed graphs without multiple edges and with a finite node set over a set \( \Sigma \) of labels: \( G = (V, E) \) where \( V = \{1, \ldots, n\} = [n] \) 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 \( H \):
\[ g_V : V_G \rightarrow V_H \]
edge labeled directed graphs without multiple edges and with a finite node set over a set $\Sigma$ of labels: $G = (V, E)$ where $V = \{1, \ldots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$

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

\[\text{member}\]
\[\emptyset \text{ member } \emptyset \text{ member } \emptyset\]

\[\begin{array}{c}
1 & \quad 2 \\
\hline
\end{array} \quad \begin{array}{c}
1 & \quad 2 & \quad 3 & \quad 4 \\
\hline
\end{array}\]
Graphs

- edge labeled directed graphs without multiple edges and with a finite node set over a set \( \Sigma \) of labels: \( G = (V, E) \) where \( V = \{1, \ldots, n\} = [n] \) 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 $\Sigma$ of labels: $G = (V, E)$ where $V = \{1, \ldots, n\} = [n]$ and $E \subseteq V \times \Sigma \times V$

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

\[
\begin{align*}
\text{member} & \quad \emptyset \quad \text{member} \quad \emptyset \\
1 \quad 2 & \quad 1 \quad 2 \quad 3 \quad 4
\end{align*}
\]
Graphs

- edge labeled directed graphs without multiple edges and with a finite node set over a set $\Sigma$ of labels: $G = (V, E)$ where $V = \{1, \ldots, n\} = [n]$ 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: \( G \xrightarrow{r,g} H \)

\[
\begin{align*}
\text{remEdge} &= \begin{array}{c}
\text{member} \\
1 \quad 2
\end{array} \quad \rightarrow \quad \begin{array}{c}
\text{member} \\
1 \quad 2
\end{array} \\
\end{align*}
\]

\[
\begin{array}{c}
\emptyset \text{ member} \quad \emptyset \text{ member} \\
1 \quad 2 \quad 3 \quad 4
\end{array}
\]
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: $G \xrightarrow{r,g} H$

\[
\text{member} \quad \text{member} \quad \text{remEdge} = \quad 1 \quad 2 \quad \rightarrow \quad 1 \quad 2
\]

\[
\emptyset \quad \text{member} \quad \emptyset \quad \text{member}
\]

\[
1 \quad 2 \quad 3 \quad 4
\]
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: \( G \xrightarrow{r,g} H \)

\[ \begin{align*}
\text{remEdge} &= \begin{array}{c}
\text{member} \\
\emptyset \text{ member} \\
\end{array} \\
\begin{array}{c}
1 \quad 2 \\
3 \quad 4 \\
\end{array} \\
\rightarrow \\
\begin{array}{c}
\text{member} \\
\emptyset \text{ member} \\
\end{array} \\
\begin{array}{c}
1 \quad 2 \\
\end{array}
\end{align*} \]
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: $G \xrightarrow{r,g} H$

\[
\begin{align*}
\text{remEdge} &= \quad \begin{array}{c}
1 \\
2 \\
3 \\
4 \\
\end{array} \quad \begin{array}{c}
\quad \quad \quad \quad \quad \quad \\
\quad \quad \quad \quad \quad \quad \\
\quad \quad \quad \quad \quad \quad \\
\quad \quad \quad \quad \quad \quad \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\emptyset \text{ member} \quad \emptyset \text{ member} \\
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
\end{array} \quad \begin{array}{c}
\quad \quad \quad \quad \quad \quad \\
\quad \quad \quad \quad \quad \quad \\
\quad \quad \quad \quad \quad \quad \\
\quad \quad \quad \quad \quad \quad \\
\end{array} \\
\end{align*}
\]
Rule application

- \[ r = (L \rightarrow R) \text{ where } V_L = V_R \] (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: \( G \xrightarrow{r,g} H \)

\[
\text{remEdge} = \begin{array}{cc}
\emptyset & \text{member} \\
1 & 2 \\
\end{array} \quad \rightarrow \quad \begin{array}{cc}
\emptyset & \text{member} \\
1 & 2 \\
\end{array}
\]

\[
\emptyset \text{ member} \quad \emptyset \text{ member} \quad \begin{array}{cc}
\emptyset & \text{member} \\
1 & 2 & 3 & 4 \\
\end{array} \quad \rightarrow \quad \begin{array}{cc}
\emptyset & \text{member} \\
1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\text{remEdge,}\{1\rightarrow4,2\rightarrow3\} = \begin{array}{cc}
\emptyset & \text{member} \\
1 & 2 & 3 & 4 \\
\end{array}
\]
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: $G \xrightarrow{r,g} H$

\[
\begin{align*}
\text{remEdge} &= \begin{array}{c}
\shuffle \text{ member} \shuffle \text{ member} \\
1 \leftrightarrow 2 \\
3 \leftrightarrow 4
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
\text{remEdge},\{1 \rightarrow 4, 2 \rightarrow 3\} \Rightarrow
\end{align*}
\]

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

- \( d = G_0 \rightarrow G_1 \rightarrow \cdots \rightarrow G_n \) is called a derivation,
- \( G_0 \xrightarrow{P} G_n \)
Derivation

- $d = G_0 \rightarrow_{r_1,g_1} G_1 \rightarrow_{r_2,g_2} \cdots \rightarrow_{r_n,g_n} G_n$ is called a derivation

- $G_0 \xrightarrow{P}^* G_n$

$\emptyset$ member $\emptyset$ member

1 2 3 4
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.
- $G_0 \xrightarrow{*}{P} G_n$

![Diagram with nodes and edges representing a derivation process.](image)
Derivation

- \( d = G_0 \rightarrow r_1, g_1 \rightarrow G_1 \rightarrow r_2, g_2 \rightarrow \cdots \rightarrow G_n \rightarrow r_n, g_n \) is called a derivation

- \( G_0 \xrightarrow{\star, P} G_n \)

\[ \begin{array}{c}
\varnothing \text{ member } \varnothing \text{ member} \\
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\end{array} \rightarrow \begin{array}{c}
\varnothing \text{ member } \varnothing \text{ member} \\
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\end{array} \xrightarrow{1} \text{ remEdge} \]

\[ \begin{array}{c}
\varnothing \text{ member } \varnothing \text{ member} \\
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\end{array} \rightarrow \begin{array}{c}
\varnothing \text{ member } \varnothing \text{ member} \\
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\end{array} \xrightarrow{3} \text{ remEdge} \]
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 $⇒$ such derivations are called *successful*. 
Graph rewriting for graph problems

VertexCover(k)

Initial: unlabeled & undirected & \( \emptyset \) – loops

Rules:

choose: \( 1 \)  \( \emptyset \) \( \rightarrow \) \( 1 \)  \( \text{member} \)

\( \text{remEdges} \):

\( 1 \)  \( 2 \) \( \rightarrow \) \( 1 \) \( 2 \)

Condition: \( \text{choose}^k \); \( \text{remEdges}^* \)

Terminal: no edges & (member | \( \emptyset \)) – loops
Derivation revisited

∅ ∅ ∅ ∅

1 2 3 4
Derivation revisited

∅  ∅  ∅  ∅ 
1  2  3  4

⇒

choose

∅  ∅  ∅  ∅  
1  2  3  4

⇒

remEdge

∅  ∅  ∅  ∅  
1  2  3  4

⇒

remEdge

∅  ∅  ∅  ∅  
1  2  3  4

⇒

member

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

∅  ∅  ∅  ∅  ⇒  choose

∅  ∅  ∅  member  ∅  ∅  ⇒  remEdge

∅  member  ∅  member  ⇒  choose

∅  member  ∅  member
Derivation revisited

∅ ∅ ∅ ∅

1 2 3 4

⇒ choose

∅ ∅ ∅ member

1 2 3 4

⇒ remEdge

∅ member ∅ member

1 2 3 4

⇒ choose

∅ member ∅ member

1 2 3 4

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

∅  ∅  ∅  ∅ =⇒ choose
1  2  3  4

∅  ∅  ∅  member
1  2  3  4

∅ member  ∅ member
1  2  3  4

=⇒ choose

∅ member  ∅ member
1  2  3  4

=⇒ remEdge

∅ member  ∅ member
1  2  3  4

3 =⇒ remEdge
1  2  3  4
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 \( k \)th derivation step expressed via edges that are in the graph and edges that are not in the graph

\[
\text{graph}(G)(k) = \bigwedge_{(v, a, v') \in E_G} \text{edge}(v, a, v', k) \land \bigwedge_{(v, a, v') \in ([n] \times \Sigma \times [n]) \setminus E_G} \neg \text{edge}(v, a, v', k).
\]
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**:  
  
  $\text{keep}(r, g, k) = \bigwedge_{(v, a, v') \not\in g(E_L \cup E_R)} (\text{edge}(v, a, v', k - 1) \leftrightarrow \text{edge}(v, a, v', k))$

  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:**
  \[
  \text{apply}(r, g, k) = \text{morph}(r, g, k) \land \text{rem}(r, g, k) \land \text{add}(r, g, k) \land \text{keep}(r, g, k)
  \]

**Theorem**

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

- 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:

\[
\text{graph}(G_0)(0) \land \text{apply}(\text{choose}, \{1 \mapsto 4\}, 1) \land \text{apply}(\text{choose}, \{1 \mapsto 2\}, 2) \land \\
\text{apply}(\text{remEdge}, \{1 \mapsto 4, 2 \mapsto 3\}, 3) \land \text{apply}(\text{remEdge}, \{1 \mapsto 2, 2 \mapsto 1\}, 4) \land \\
\text{apply}(\text{remEdge}, \{1 \mapsto 2, 2 \mapsto 3\}, 5) \land \text{apply}(\text{remEdge}, \{1 \mapsto 4, 2 \mapsto 2\}, 6).
\]
\( L(C)(p(n)) \) 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-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
derivation is extracted from the variable assignment

GrGen.NET is used for visualization

additional informations on console
SATaGraT - postprocessing (2)

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

A SAT-Based Graph Rewriting and Verification Tool Implemented in Haskell
Experiments: vertex cover problem

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
<th>k</th>
<th>VC?</th>
<th>SATaGraT 2011</th>
<th>SATaGraT 2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>2</td>
<td>no</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>2</td>
<td>no</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>4</td>
<td>yes</td>
<td>96</td>
<td>34.5</td>
</tr>
<tr>
<td>13</td>
<td>20</td>
<td>3</td>
<td>yes</td>
<td>366</td>
<td>112</td>
</tr>
<tr>
<td>13</td>
<td>18</td>
<td>3</td>
<td>no</td>
<td>357</td>
<td>456</td>
</tr>
<tr>
<td>15</td>
<td>24</td>
<td>3</td>
<td>yes</td>
<td>&gt; 3600</td>
<td>438</td>
</tr>
</tbody>
</table>

- 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?

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

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

Outlook:

- 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