Towards theorem proving

graph grammars

Leila Ribeiro

joint work with Simone Costa, Fernando Dotti, Fabiane Dillenburg, Alfio Martini

Universidade Federal do Rio Grande do Sul (UFRGS), Brazil

IFIP WG 3 - July 2010 - Etelesen
Motivation

* Theorem proving is a powerful technique for the analysis of computational systems.

* It is possible to prove properties of reachable states for infinite state systems.

* This approach is complementary to other existing analysis techniques for graph grammars, like model checking and analysis based on approximations.

Personal motivation: I kept trying to convince colleagues that using formal methods is nice, they allow to prove many relevant properties, hopefully using some tool, but I myself never used theorem provers before...
But...

* Theorem provers are typically hard to use because
  - the system under analysis must be faithfully encoded in a logical framework (in a rather low level way)
  - many existing tools are hard to install and use
  - proving properties needs a lot of user assistance

* Graphs (with types, attributes, ...) are complex structures, and the properties we wish to prove easily become cryptic logical formulae
The idea...

* Encode GGs in such a way they become closer to the input languages of theorem provers

* Use exiting theorem provers to prove properties of graph grammars
Typed Graphs

Typed Graph: $tG$

Type Graph: $T$

Category of typed graphs and partial graph morphisms: $\text{GrAp}(T)$
(SPO) Rule

- Partial injective graph homomorphism, no vertex deletion
SPO Graph Grammar

- Type Graph $T$
- Initial (start) graph $G$ typed over $T$
- Set of rules typed over $T$

Semantics based on rule applications (POs in $\text{GraP}(T)$)
Relational Graph Grammars

* Definition of graph grammars based on relations and restrictions on these relations

* Faithful encoding of SPO, considering injective rules that do not delete vertices and matches that are injective on edges

* Based on Courcelle’s relational structures: domain+relations, transduction
Relational Graph

\[ V_{GG} : \text{set of vertex ids} \]
\[ E_{GG} : \text{set of edge ids} \]

Graph \( T \)

\[ \text{vert}T \subseteq V_{GG} \]
\[ \text{edge}T \subseteq E_{GG} \]
\[ \text{source}T \in \text{edge}T \rightarrow \text{vert}T \]
\[ \text{target}T \in \text{edge}T \rightarrow \text{vert}T \]

Graph \( G \)

\[ \text{vert}G \subseteq V_{GG} \]
\[ \text{edge}G \subseteq E_{GG} \]
\[ \text{source}G \in \text{edge}G \rightarrow \text{vert}G \]
\[ \text{target}G \in \text{edge}G \rightarrow \text{vert}G \]

\( tG.V \in \text{vert}G \rightarrow \text{vert}T \)
\( tG.E \in \text{edge}G \rightarrow \text{edge}T \)

source/target compat.
Relational Rule

\[ \alpha : L \rightarrow R \]

\[ \alpha_V : \text{vert} L \rightarrow \text{vert} R \]
\[ \alpha_E : \text{edge} L \leftrightarrow \text{edge} R \]

Match is analogous, but total.

\textit{type compat.}

Monday, July 5, 2010
Rule Application

\[
\begin{array}{c}
L \xrightarrow{\alpha} R \\
\downarrow m \\
G \xrightarrow{(PO)} H
\end{array}
\]

\[
\text{vert}H = \text{vert}G \uplus (\text{vert}R - \alpha_V(\text{vert}L))
\]

\[
\text{edge}H = (\text{edge}G - m_E(\text{edge}L)) \uplus E_R
\]

+ source, target, typing functions...

Monday, July 5, 2010
Now, implementing....

* First attempt: Use Isabelle

* Second attempt: Event-B (Rodin platform)
Event-B

* Formalism based on B

* Event-B models are composed by
  - **CONTEXT**: definition of types, operations and facts that may be used in the model
  - **MACHINE**: definition of variables (state), events (transitions) and invariants

**Definition of type graph and rules**

**Definition of start graph and rule application**
CONTEXT  ctx_gg

SETS
vertT  (Type Graph T) Verteces
edgeT  (Type Graph T) Edges

CONSTANTS
sourceT
targetT
Vertex1
Edge1
Vertex2
Edge2

AXIOMS
axm1 : partition(vertT, {Vertex1}, {Vertex2})
axm2 : partition(edgeT, {Edge1}, {Edge2})
axm_src : sourceT ∈ edgeT → vertT
axm_srcD : partition(sourceT, {Edge1 ↔ Vertex1}, {Edge2 ↔ Vertex1})
axm_trg : targetT ∈ edgeT → vertT
axm10 : partition(targetT, {Edge1 ↔ Vertex1}, {Edge2 ↔ Vertex2})
MACHINE mch_gg
SEES ctx_gg
VARIABLES
  vertG
  edgeG
  sourceG
  targetG
  tG_V  Typing vertices, tG_V
  tG_E  Typing edges, tG_E

INVIARANTS
  type_vertG : vertG ∈ P(N)
  type_edgeG : edgeG ∈ P(N)
  type_sourceG : sourceG ∈ edgeG → vertG
  type_targetG : targetG ∈ edgeG → vertG
  type_tG_V : tG_V ∈ vertG → vertT
  type_tG_E : tG_E ∈ edgeG → edgeT

EVENTS
Initialisation
  begin
  act1 : vertG := \{1\}
  actE : edgeG := \{1\}
  acts : sourceG := \{1 \mapsto 1\}
  actt : targetG := \{1 \mapsto 1\}
  act3 : tG_V := \{1 \mapsto Vertex1\}
  act4 : tG_E := \{1 \mapsto Edge1\}
  end
END
**Rule**

**CONTEXT**  
ctx.gg

**SETS**  
- vertL1  
- edgeL1  
- vertR1  
- edgeR1

**CONSTANTS**  
- v1.L1  
- e1.L1  
- v1.R1  
- v2.R1  
- e1.R1  
- tL1.V (LHS1) Typing vertices, tL1.V  
- tL1.E (LHS1) Typing edges, tL1.E  
- tR1.V (RHS1) Typing vertices, tR1.V  
- tR1.E (RHS1) Typing edges, tR1.E  
- alpha1V (Rule 1) Rule alpha1: mapping vertices  
- alpha1E (Rule 1) Rule alpha1: mapping edges

**AXIOMS**

- axm15: partition(vertL1, {v1.L1})  
- axm16: partition(edgeL1, {e1.L1})  
- axm_srcL1: sourceL1 ∈ edgeL1 → vertL1  
- axm_srcDL1: partition(sourceL1, {e1.L1 ↦ v1.L1})  
- axm_tarL1: targetL1 ∈ edgeL1 → vertL1  
- axmtarDL1: partition(targetL1, {e1.L1 ↦ v1.L1})  
- tL1.V ∈ vertL1 → vertT  
- act9: partition((tL1.V, {v1.L1 ↦ Vertex1}))  
- axmtL1E: tL1.E ∈ edgeL1 → edgeT  
- act10: partition((tL1.E, {e1.L1 ↦ Edge1}))  
- axm17: partition(vertR1, {v1.R1, v2.R1})  
- axm_srcR1: sourceR1 ∈ edgeR1 → vertR1  
- axm_srcDR1: partition(sourceR1, {e1.R1 ↦ v1.R1})  
- axmtarR1: targetR1 ∈ edgeR1 → vertR1  
- axmtarDR1: partition(targetR1, {e1.R1 ↦ v2.R1})  
- tR1.V ∈ vertR1 → vertT  
- act13: partition((tR1.V, {v1.R1 ↦ Vertex1}, {v2.R1 ↦ Vertex2}))  
- axm18: partition(edgeR1, {e1.R1})  
- axmtR1: tR1.E ∈ edgeR1 → edgeT  
- act14: partition((tR1.E, {e1.R1 ↦ Edge2}))  
- axmA1V ∈ vertL1 → vertR1  
- axmA1E: alpha1V ∈ vertL1 → vertR1

Monday, July 5, 2010
Rule (Behavior)

EVENTS
Event $\alpha_1 \triangleq$

any
$mV$
$mE$
$newV$
$newE$

where

$\text{grd}_1 : mV \in \text{vert}L1 \rightarrow \text{vert}G$
$\text{grd}_2 : mE \in \text{edge}L1 \rightarrow \text{edge}G$
$\text{grd}_3 : newV \in \mathbb{N} \setminus \text{vert}G$
$\text{grd}_4 : newE \in \mathbb{N} \setminus \text{edge}G$
$\text{grd}_5 : \forall v \cdot v \in \text{vert}L1 \Rightarrow tL1.V(v) = tG.V(mV(v))$
$\text{grd}_6 : \forall e \cdot e \in \text{edge}L1 \Rightarrow tL1.E(e) = tG.E(mE(e))$
$\text{grd}_7 : \forall e \cdot e \in \text{edge}L1 \Rightarrow mV(\text{sourceL1}(e)) = \text{sourceG}(mE(e)) \land$
$mV(\text{targetL1}(e)) = \text{targetG}(mE(e))$

then

$\text{act}_3 : \text{vert}G := \text{vert}G \cup \{\text{newV}\}$
$\text{act}_E : \text{edge}G := (\text{edge}G \setminus \{mE(e1.L1)\}) \cup \{\text{newE}\}$
$\text{acts} : \text{sourceG} := (\{mE(e1.L1)\} \hookrightarrow \text{sourceG}) \cup \{\text{newE} \mapsto (\text{sourceG}(mE(e1.L1)))\}$
$\text{actt} : \text{targetG} := (\{mE(e1.L1)\} \hookleftarrow \text{targetG}) \cup \{\text{newE} \mapsto \text{newV}\}$
$\text{acttV} : tG.V := tG.V \cup \{\text{newV} \mapsto \text{Vertex2}\}$
$\text{act6} : tG.E := (\{mE(e1.L1)\} \hookleftarrow tG.E) \cup \{\text{newE} \mapsto \text{Edge2}\}$

end
Properties

Properties are stated as invariants:

**INVARİANTS**

... : ...

prop1 : \text{finite}(edgeG)

prop2 : \text{card}(edgeG) \leq 2

In Event-B, proof obligations are generated to guarantee that all invariants are well-defined, are valid in any initial state and are preserved by all events.
Another Example: Token Ring
Another Example: Token Ring
86 POs: 57 (A) + 29 (SA)
Final Remarks

- GGs can be faithfully encoded in Event-B
- Rodin offers a set of theorem provers that can be used to verify graph grammars
- However, we need
  - libraries for commonly used operations;
  - theories for graph structures;
  - property patterns (tactics patterns);
  - integration of data types (attributed GTS), NACs, ...
  - more "intelligent" theorem prover: Isabelle???
Thank you!!!