2.5 Graph Grammars
Graph Grammars

- Example
  - Describing program states as graphs
  - Describing program behavior through graph transformations
- Models and graphs
- Graph grammars
- Story diagrams
Example: Graph Grammars in every day life

- Most children know graph grammars
- Every Lego build instruction is a graph grammar
Example

- Example: simulation of a track-based transportation system

Concrete syntax:

Abstract syntax:
Object Graph

- The object graph represents the state of a program
- The behavior of a program is a change of the state
  - A change of the object graph
Object Graph

- The *object graph* represents the state of a program
- The behavior of a program is a change of the state
  - A change of the object graph

Example: Moving a shuttle
Object Graph

- The program behavior can be described by *rules for transforming the object graph*.
Motivation

- Graph transformation rules allow us:
  - To describe the behavior of systems in an abstract way (e.g. RailCab)
    - To model and analyze them
  - To describe the semantics of pointer programs (OO)
    - To validate or verify them
    - To model the program behavior visually (Executable code can be generated from them)
Graph Grammars

- Example
  - Describing program states as graphs
  - Describing program behavior through graph transformations
- Models and graphs
- Graph grammars
- Story diagrams
Instance Model and Class Model

Class Diagram:

Object graph:
Instance Model and Class Model

Class Diagram:

Object graph:

Typed Graph (edges are typed, too)

Typed Graph

Type Graph
Graph Grammars

- Example
  - Describing program states as graphs
  - Describing program behavior through graph transformations
- Models and graphs
- Graph grammars
- Story diagrams
Graph Grammars

- A graph grammar consists of
  - A set of graph grammar rules (production rules)
  - A start graph
  - A type graph
- A graph grammar describes how to produce valid sets of graphs (sets may be infinite)
- Synonyms:
  - Graph Rewriting System or Graph Transformation System

<table>
<thead>
<tr>
<th>start graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Shuttle</td>
</tr>
<tr>
<td>:Track</td>
</tr>
<tr>
<td>:Track</td>
</tr>
<tr>
<td>:Track</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rule: move</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Shuttle</td>
</tr>
<tr>
<td>:Track</td>
</tr>
<tr>
<td>:Track</td>
</tr>
<tr>
<td>:Track</td>
</tr>
</tbody>
</table>

- move: "<<create>>"
Graph Grammar Rule

- A **graph grammar rule** consists of
  - A left-hand side and right-hand side typed graph

**Graph Grammar Rule**

Short hand form:

```
move

lhs:

v2

:Shuttle

isOn

:Track

next

v1

::=

rhs:

v2

:Shuttle

isOn

:Track

next

v1

\<<create>>

v3

<<delete>>

identifier
```

```
move

:Shuttle

isOn

:Track

next

<<create>>

:Track

isOn

<<delete>>

:Track
```
A graph grammar rule consists of
- A left-hand side and right-hand side typed graph

\[ \text{lhs:} \quad \text{rhs:} \]

\[
\begin{align*}
\text{Shuttle} & \quad \text{Track} \\
\text{Track} & \quad \text{Shuttle} \\
\text{Switch} & \\
\end{align*}
\]

(edges are typed, too)
Graph Grammar Rule Application

move

lhs:  

\[ \text{v2} \rightarrow \text{Shuttle} \]

\[ \text{v1} \rightarrow \text{Track} \]

\[ \text{isOn} \rightarrow \text{Track} \]

\[ \text{next} \rightarrow \text{next} \]

rhs:  

\[ \text{v2} \rightarrow \text{Shuttle} \]

\[ \text{v1} \rightarrow \text{Track} \]

\[ \text{isOn} \rightarrow \text{Track} \]

\[ \text{next} \rightarrow \text{next} \]
Graph Grammar Rule 
Application

1. Match lhs in host graph

move

<table>
<thead>
<tr>
<th>lhs</th>
<th>rhs</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>v2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v1</th>
<th>v3</th>
</tr>
</thead>
</table>

:Shuttle :Shuttle
:Track   :Track
:Track   :Track

isOn     isOn

next      next

1. Match lhs in host graph
Graph Grammar Rule Application

2. Remove nodes and edges which are in lhs, but not in rhs

<table>
<thead>
<tr>
<th>move</th>
<th>v1</th>
<th>:Track</th>
<th>next</th>
<th>v2</th>
<th>:Shuttle</th>
<th>.isOn</th>
</tr>
</thead>
</table>

lhs: v2 :Shuttle
 v1 :Track
 :Track

rhs: v2 :Shuttle
 :Switch
 v1 :Track
 :Track
 :Track

::=
Graph Grammar Rule Application

3. Create nodes and edges which are in rhs, but not in lhs

move

lhs:

\[
\begin{align*}
&v_1: \text{:Track} \\
&\quad \downarrow \text{next} \\
&v_2: \text{:Shuttle} \\
&\quad \downarrow \text{isOn} \\
&v_3: \text{:Track}
\end{align*}
\]

rhs:

\[
\begin{align*}
&v_1: \text{:Track} \\
&\quad \downarrow \text{next} \\
&v_2: \text{:Shuttle} \\
&\quad \downarrow \text{isOn}
\end{align*}
\]

::=
Non-Determinism

- Example 1: When to move which shuttle?
  - Not determined!
Non-Determinism

- Example 2: Go straight or turn?
  - Not determined!

(remember: a switch is also a track)
Negative Application Conditions

- Example: Do not crash with other shuttle

```
:Shuttle
  isOn
  next
:Track
:Shuttle
  isOn
  next
:Track
:Shuttle
:Track

move

:Shuttle
  isOn
  isOn
  <<create>>
  <delete>
:Track
  next
:Shuttle
  isOn
  isOn
:Track
```
Attribute Condition

- Example 1: A broken shuttle cannot move

```
move

:Shuttle
  failure=false

:Track
  isOn

<<create>>

:Track
  isOn

<<delete>>
```
Attribute Condition

- Example 2: May follow faster shuttles on Track

```
<table>
<thead>
<tr>
<th>Shuttle</th>
<th>Track</th>
</tr>
</thead>
<tbody>
<tr>
<td>broken: boolean</td>
<td>next 1</td>
</tr>
<tr>
<td>speed: integer</td>
<td>turnNext 1</td>
</tr>
</tbody>
</table>

move

s1:Shuttle

speed ≤ s2.speed

<<create>>

isOn <<delete>>

:Track

s2:Shuttle

isOn

:Track

next
```
Graph Grammars: Overview

- Non-deterministic rule application
- Negative application conditions
- Conditions on attribute values
- Inheritance in the type graph
- There may be other extensions depending on the problem domain...
  - Ordered lists
  - Qualified associations
  - „maybe“-conditions
- Now a bit more formally…
Definition: Directed Graph

- $G = (V, E, s, t)$
- $V$ – finite set of nodes (vertices)
- $E$ – finite set of edges
- $s: E \rightarrow V$ – source function
- $t: E \rightarrow V$ – target function

Example:
$V = \{v1, v2, v3\}$
$E = \{e1, e2, e3\}$
$s(e1) = v1$
$t(e1) = v2$
Definition: Labeled Graph

- $G_L = (G, L_V, L_E)$
- $G$ – Directed Graph
- $L_V: V \rightarrow \Sigma$ – labeling function for vertices
- $L_E: E \rightarrow \Sigma$ – labeling function for edges
- $\Sigma$ – set of labels

Example:
- $V = \{v1, v2\}$  \[ L_V(v1) = \text{“shuttle”} \]
- $E = \{e1\}$ \[ L_V(v2) = \text{“track”} \]
- $s(e1) = v1$ \[ L_E(e1) = \text{“isOn”} \]
- $t(e1) = v2$
Definition: Morphism

- Given two graphs $G_i = (V_i, E_i, s_i, t_i), i \in \{1, 2\}$
- A graph morphism $f: G_1 \rightarrow G_2, f = (f_V, f_E)$ consists of two functions
  - $f_V: V_1 \rightarrow V_2$
  - $f_E: E_1 \rightarrow E_2$

preserving the source and target functions:
  - $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$

Given two functions $f$ and $g$ then $f \circ g$ is a function that maps a value $x$ to $f(g(x))$
Definition: Morphism

- **Given two graphs** $G_i=(V_i, E_i, s_i, t_i), i \in \{1, 2\}$
- **A graph morphism**
  
  $f: G_1 \to G_2, f = (f_V, f_E)$

  consists of two functions
  
  - $f_V: V_1 \to V_2$
  - $f_E: E_1 \to E_2$

  preserving the source and target functions:
  
  - $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$

Example:

$$f_V(s_1(e_{11})) = s_2(f_E(e_{11}))$$

$$f_V(t_1(e_{11})) = t_2(f_E(e_{11}))$$
Definition: Typed Graph

- A **Typed Graph** $G_{\text{Typed}} = (G, \text{type})$ consists of
  - a graph $G$ and
  - a graph morphism to a type graph $\text{type}: G \to G_{\text{Type}}$

Note:
A morphism is a **total function** ($f_V, f_E$ are total), i.e. every element in the domain (typed graph) has to be related to exactly one element of the co-domain (type graph)
Matching a Left-hand Side

- Matching a graph inside another "host" graph
  - Find a *typed isomorphism* to a *subgraph* of the host graph
Definition: Subgraph

- A subgraph \( G_{\text{Sub}} = (V_{\text{Sub}}, E_{\text{Sub}}, s_{\text{Sub}}, t_{\text{Sub}}) \) of \( G = (V, E, s, t) \) is a graph with
  - \( V_{\text{Sub}} \subseteq V \)
  - \( E_{\text{Sub}} \subseteq E \)
  - \( s_{\text{Sub}} = s \big|_{E_{\text{Sub}}} \)
  - \( t_{\text{Sub}} = t \big|_{E_{\text{Sub}}} \)

\( E_{\text{Sub}} \big|_{E_{\text{Sub}}} \)

Means that the domain of the source and target function for the subgraph is reduced to the edges which are in it.

- If \( G_{\text{Sub}} \) is a subgraph of \( G \), we may also write \( G_{\text{Sub}} \leq G \).
Definition: Isomorphism

- A graph morphism \( f: G_1 \to G_2, f = (f_V, f_E) \) with \( f_V, f_E \) bijective is called a graph isomorphism.
**Definition: Typed Isomorphism**

- Both the graph and host graph are typed over the same type $\text{Graph}_{\text{Type}}$:
  - Let $\text{type} = (\text{type}_V, \text{type}_E)$ be a graph morphism and
  - $f = (f_V, f_E)$ is an isomorphism, then $f$ is a *typed isomorphism* when
    - $\text{type}_V(v) = \text{type}_V(f_V(v))$
    - $\text{type}_E(e) = \text{type}_E(f_E(e))$
Definition: Graph Grammar Rule

- A graph grammar rule \( r \) is defined as a partial morphism \( r : L \rightarrow R \) with \( L \in G_{\text{Type}} \) the lhs graph of \( r \), \( R \in G_{\text{Type}} \) the rhs graph of \( r \), \( G_{\text{Type}} \) the set of all graphs typed over \( G_{\text{Type}} \) such that there is a maximum common subgraph \( TL \) of \( L \) and \( R \), with:
  - \( \exists TL \in G_{\text{Type}} : TL \leq L \) and \( TL \leq R \) and
  - \( TL \) not empty
Definition: Graph Grammar Rule

Let $L$ be the left-hand side (lhs) graph and $R$ be the right-hand side (rhs) graph.

$L$ is a partial morphism $r : L \rightarrow R$ that maps:
- $v_1$ from $L$ to $v_1$ in $R$,
- $v_2$ from $L$ to $v_2$ in $R$,
- $v_3$ from $L$ to $v_3$ in $R$.

The $T_L$ (common subgraph) is:
- $v_1$ from $L$ to $v_1$ in $R$,
- $v_2$ from $L$ to $v_2$ in $R$.

The graph grammar rule involves:
- $:Shuttle$ from $L$ to $R$,
- $:Track$ from $L$ to $R$,
- $:Track$ from $L$ to $R$.
Rule Application - Formally

- The application of a rule \( r: L \rightarrow R \) is defined by a relation \( app_r = \{(G, G'') \in (\mathcal{G}_{\text{Type}} \times \mathcal{G}_{\text{Type}}) \mid G'' \text{ is defined by the three steps}
\]
1. Search (on host graph G)
2. Delete (result is G')
3. Create (result is G'')
\}
or \( G = G'' \)

- \( app_r \) represents all possible applications of \( r \)

An element is a possible (host) graph typed over \( \mathcal{G}_{\text{Type}} \)
Rule Application – Step 1

- **Search:**
  Given a host graph $G$, find a subgraph isomorphism
  
  $match: L \rightarrow G_{Sub}, G_{Sub} \leq G$

![Diagram showing graph structure and isomorphism](imagedescr)
Rule Application – Step 2 (1/3)

2. Delete:
Form a new graph \( G' \) from \( G \) by deleting all nodes and edges in \( G_{\text{Sub}} \) which are in \( \text{match}(L) = G_{\text{Sub}} \) but not in \( G'_{\text{Sub}} = \text{match}(TL) \) (\( \leq G_{\text{Sub}} \))

(Only the edge labeled “isOn” is removed)
**Rule Application – Step 2 (2/3)**

- **Delete:**
  Remove all edges leading to deleted nodes

---

- **Create Assignment**
  - **:Request**
  - **:Assignment**
  
  
  ![Diagram]

- **:Broker**
  - **:Shuttle**
  - **:Request**
  - **:Broker**
  - **:Shuttle**

- **L (lhs)**
  - **:Request**
  - **:Broker**
  - **:Shuttle**

- **TL (common subgraph)**
  - **:Broker**
  - **:Shuttle**

- **Queue Request**
  - **:Request**
  - **:Broker**
  - **:Shuttle**

- **Resp For**
  - **:Request**
  - **:Broker**
  - **:Shuttle**

- **Delete**
  - **:Request**
  - **:Broker**
  - **:Shuttle**

- **Create**
  - **:Request**
  - **:Broker**
  - **:Shuttle**
Rule Application – Step 2 (2/3)

- **Delete:**
  Remove all edges leading to deleted nodes

---

![Diagram showing graph grammar rule application step 2]
Rule Application – Step 2 (3/3)

- **Delete:**
  Remove all edges leading to deleted nodes

```plaintext
createAssignment

:Customer

:Broker

:Shuttle

:Broker

:Shuttle

:Shuttle

L (lhs)

:Request

:Broker

:Request

:Broker

:Shuttle

:Broker

:Shuttle

:Shuttle

TL (common subgraph)

queuedRequest

respFor

respFor

respFor

respFor

respFor

respFor

assigned

<<delete>>

<<create>>

<<create>>
```

Graph Grammars
Rule Application – Step 3 (1/3)

- **Create:**
  Add nodes and edges to $G'$, forming an extended graph $G''$, such that there is an isomorphism $\text{match}_R: R \rightarrow G''_{\text{Sub}}$ from $R$ to a new subgraph $G''_{\text{Sub}} \subseteq G''$ such that $(G'_{\text{Sub}} =) \text{match}(TL) \leq \text{match}_R(R)$ (= $G''_{\text{Sub}}$)
Rule Application – Step 3 (2/3)

- **Create:**
  \[
  G_{\text{Sub}}' = \text{match}(TL) \leq \text{match}_R(R) = G_{\text{Sub}}''
  \]

---

**Graph Grammars**

© Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn

HEIZ NIXDORF INSTITUT
Universität Paderborn
Softwaretechnik
Prof. Dr. Wilhelm Schäfer
Rule Application – Step 3 (3/3)

- Create:

\[ G'_{\text{Sub}} = \text{match}(TL) \leq \text{match}_R(R) = G''_{\text{Sub}} \]
Rule Application - Formally

- The application of a rule $r: L \rightarrow R$ is defined by a relation $app_r = \{(G, G^\prime\prime) \in (G_{Type} \times G_{Type}) \mid G^\prime\prime \text{ is defined by the three steps}\$
  1. Search (on host graph $G$)
  2. Delete (result is $G'$)
  3. Create (result is $G''$)
  or $G = G'' \}$

- A concrete rule application is represented by a tuple $(G, G^\prime\prime) \in app_r$ where $G$ is the host graph before applying the rule and $G^\prime\prime$ is the host graph after rule application.
What about Inheritance?

- $G_{Type} = (G_L, I)$
- $G_L$ – Directed labeled Graph
- $I \subseteq V \times V$ – inheritance relation (cycle free)

Example:
- $V = \{v1, v2, v3\}$
- $E = \{e1, e2, e3\}$
- $s(e1) = v1$
- $t(e1) = v2$
- $I = \{(v2, v3)\}$

But now the formalization of the matching becomes more complicated…
Summary

- Graph transformation rules allow us:
  - To describe the behavior of systems (e.g. RailCab) in an abstract, formal way
    - To model and analyze them
  - To describe the semantics of OO-programs
    - To validate or verify them
  - To model the program behavior visually (Executable code can be generated from them)