2.4 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:**

![Concrete syntax diagram]

**Abstract syntax:**

![Abstract syntax diagram]
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:

Type Graph

Typed Graph (edges are typed, too)
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

```
start graph

:Shuttle <<<create>>>
  isOn
   :Track <<<delete>>>
     next
      :Track
    next
     :Track
next
next
next

rule: move

:Shuttle
  isOn
  <<<delete>>>
  next
  :Track
  isOn
  next
  :Track
```
Graph Grammar Rule

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

Short hand form:
Graph Grammar Rule

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

\[
\begin{align*}
\text{lhs:} & & \text{rhs:} \\
v_{2}: & \text{Shuttle} & v_{2}: & \text{Shuttle} \\
\quad & \text{isOn} & \quad & \text{isOn} \\
v_{1}: & \text{Track} & \quad & \text{Track} \\
\quad & \text{next} & \quad & \text{next} \\
v_{3}: & \text{Track} & v_{3}: & \text{Track} \\
\end{align*}
\]
Graph Grammar Rule

Application

move

lhs: :Shuttle
    v2
        isOn
            :Track
                next
                    v1
                        :Track
                            next
                                v3
                                    :Track
                                        ::=
                                            :Shuttle
                                                isOn
                                                    :Track
                                                        turnNext
                                                            :Switch
                                                                next
                                                                    :Track
                                                                        v2
                                                                            :Shuttle
                                                                                isOn
                                                                                    v3
                                                                                        :Track
                                                                                            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>
<tr>
<td>v1</td>
<td>v3</td>
</tr>
</tbody>
</table>

Graph Grammar Rule
Application

move

<table>
<thead>
<tr>
<th>lhs:</th>
<th>rhs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>v2</td>
</tr>
<tr>
<td>v1</td>
<td>v3</td>
</tr>
</tbody>
</table>
Graph Grammar Rule Application

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

move

<table>
<thead>
<tr>
<th>lhs</th>
<th>rhs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v1</code> &lt;nil&gt;</td>
<td><code>v2</code> &lt;nil&gt;</td>
</tr>
<tr>
<td><code>v1</code> :Shuttle :Track</td>
<td><code>v2</code> :Shuttle :Track</td>
</tr>
<tr>
<td><code>v2</code> :Shuttle</td>
<td>:Shuttle</td>
</tr>
</tbody>
</table>

::=`

Next steps:

- Remove nodes and edges which are in lhs, but not in rhs.
3. Create nodes and edges which are in rhs, but not in lhs
Non-Determinism

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

lhs: \[ \text{:Shuttle} \]
\[ \text{isOn} \]
\[ \text{:Track} \]
\[ \text{next} \]

rhs: \[ \text{:Shuttle} \]
\[ \text{isOn} \]
\[ \text{:Track} \]
\[ \text{next} \]

\[ \text{move} \]
\[ \text{::=} \]
\[ \text{v1} \]
\[ \text{v2} \]
\[ \text{v3} \]
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
Attribute Condition

- Example 1: A broken shuttle cannot move

Graph:

- Shuttle
  - failure: boolean
- Track
  - isOn
- Switch
  - turnNext
- move
  - Shuttle
    - failure=false
  - Track
    - isOn
  - Track
    - next

<<create>>
<<delete>>
Attribute Condition

- Example 2: May follow faster shuttles on Track

```
Example 2: May follow faster shuttles on Track

Shuttle
  broken: boolean
  speed: integer

Track
  isOn

Switch
  turnNext

move

s1:Shuttle
  speed ≤ s2.speed

<<create>>

s2:Shuttle

<<delete>>
```

Graph Grammars
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:

\( V = \{v_1, v_2, v_3\} \)
\( E = \{e_1, e_2, e_3\} \)
\( s(e_1) = v_1 \)
\( t(e_1) = v_2 \)
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:

\( V = \{v_1, v_2\} \)  \( L_V(v_1) = \text{"shuttle"} \)
\( E = \{e_1\} \)  \( L_V(v_2) = \text{"track"} \)
\( s(e_1) = v_1 \)  \( L_E(e_1) = \text{"isOn"} \)
\( t(e_1) = v_2 \)
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 $f: G_1 \rightarrow G_2$, $f = (f_V, f_E)$ 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 $f_V \circ t_1 = t_2 \circ f_E$

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 $\text{type}: G \rightarrow G_{\text{Type}}$

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

\[
\text{Lhs: } v_1 \quad \text{:Track} \quad \text{next} \quad \text{:Track} \quad \text{next} \quad \text{:Track} \\
\text{subgraph} \\
\text{isOn} \quad \text{isOn} \\
\text{v2} \quad \text{v1} \quad \text{v3} \\
\text{isomorphism} \\
\text{:= } \ldots
\]
Definition: Subgraph

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

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

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

A graph morphism $f: G_1 \rightarrow G_2$, $f = (f_V, f_E)$ 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 Graph $G_{\text{Type}}$:
  - Let $\text{type} = (\text{type}_V, \text{type}_E)$ be a graph morphism and
  - $f = (f_V, f_E)$ is an isomorphism mapping a graph to a host graph, 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 $L \in G_{Type}$ the lhs graph of $r$, $R \in G_{Type}$ the rhs graph of $r$, $G_{Type}$ the set of all graphs typed over $G_{Type}$ such that there is a maximum common subgraph $TL$ of $L$ and $R$, with:
  - $\exists TL \in G_{Type} : TL \leq L$ and $TL \leq R$ and
  - $TL$ not empty
**Definition:**

**Graph Grammar Rule**

$L$ (lhs graph)

- **LHSH**
  - :Shuttle
  - :Track
  - :Track

- next
- isOn

$R$ (rhs graph)

- **RHS**
  - :Shuttle
  - :Track
  - :Track

- next
- isOn

$T_L$ (common subgraph)

- **LHS**
  - :Shuttle
  - :Track

- next

Partial morphism $r : L \rightarrow R$
Rule Application - Formally

- The application of a rule $r: L \rightarrow R$ is defined by a relation $app_r = \{ (G, G'') \in (G_{Type} \times G_{Type}) \mid G''$ 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 $G_{Type}$
Rule Application – Step 1

- **Search:**
  Given a host graph $G$, find a subgraph isomorphism

  $$\text{match}: L \rightarrow G_{\text{Sub}}, \ G_{\text{Sub}} \leq G$$

![Diagram of subgraph isomorphism](image)
Rule Application – Step 2 (1/3)

2. **Delete:**

Construct a graph $G'$ from $G$ by deleting all nodes and edges in $G_{Sub}$ which are in $match(L) = G_{Sub}$ but not in $G'_{Sub} = match(TL) (\leq G_{Sub})$.

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

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

```
<table>
<thead>
<tr>
<th>L (lhs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Request</td>
</tr>
<tr>
<td>:Broker</td>
</tr>
<tr>
<td>:Shuttle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TL (common subgraph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Request</td>
</tr>
<tr>
<td>:Assignment</td>
</tr>
<tr>
<td>:Broker</td>
</tr>
<tr>
<td>:Shuttle</td>
</tr>
</tbody>
</table>

```

CreateAssignment

```plaintext
createAssignment:
:Customer :Broker :Shuttle :Shuttle :Shuttle :Request created queuedRequest
```
Rule Application – Step 2 (2/3)

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

![Diagram showing graph transformation](image)

- Delete:
  - Remove all edges leading to deleted nodes

- Create Assignment:
  - Create new assignment

L (lhs)

TL (common subgraph)
Rule Application – Step 2 (3/3)

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

```
createAssignment

:Request
   queuedRequest
   respFor
   :Broker

L
(lhs)

:Customer
 :Broker
 respFor
 :Shuttle
 respFor
 :Shuttle
 respFor
 :Shuttle

:Broker
 respFor
 :Shuttle

TL
(common subgraph)

:Assignment
    assignment
    assigned
    <<create>>
    :Broker
    respFor
    :Shuttle
```
Create:
Add nodes and edges to $G'$, constructing a graph $G''$, such that there is an isomorphism $\text{match}_R : R \rightarrow G''\text{Sub}$ with $G''\text{Sub} \leq G''$ such that

$G''\text{Sub} \leq G''$ such that $(G'_{\text{Sub}} =) \text{match}(TL) \leq \text{match}_R(R) (= G''_{\text{Sub}})$
Rule Application – Step 3 (2/3)

- **Create:**
  \[ G'_{Sub} = \text{match}(TL) \leq \text{match}_R(R) = G''_{Sub} \]
Rule Application – Step 3 (3/3)

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

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

  or \( G = G'' \}

- A concrete rule application is represented by a tuple \( (G, G'') \in app_r \) where \( G \) is the host graph before applying the rule and \( G'' \) 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)

```
move

:Shuttle

<<create>>

isOn

<<delete>>

next

:Track

:Track

isOn
```

Shuttle

Track

Track

next