Model-Based Software Engineering

Lecture 09 – Transformation

Prof. Dr. Joel Greenyer

June 21, 2016

Leibniz Universität Hannover

in the last lecture...

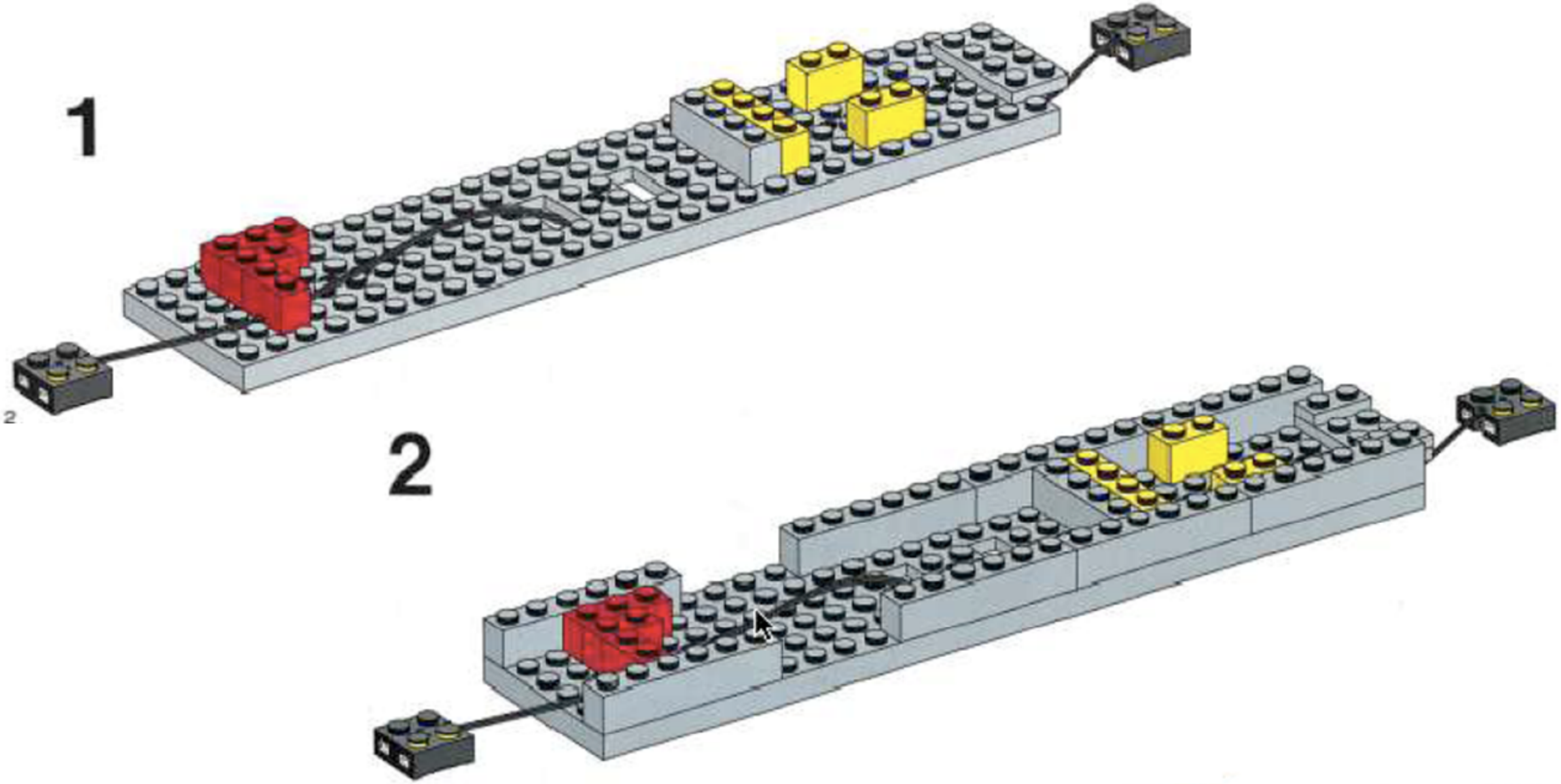
5.3. Model-to-model transformation – graph transformations

Leibniz Universität Hannover

Describe Structural Changes

in the last lecture...

- Most children understand this way of describing structural changes:

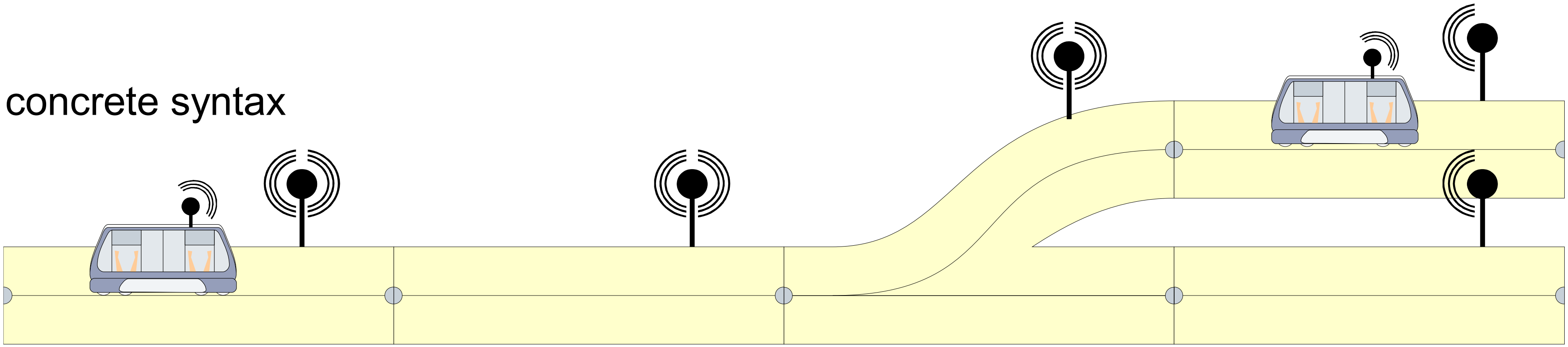


View the System as a Graph

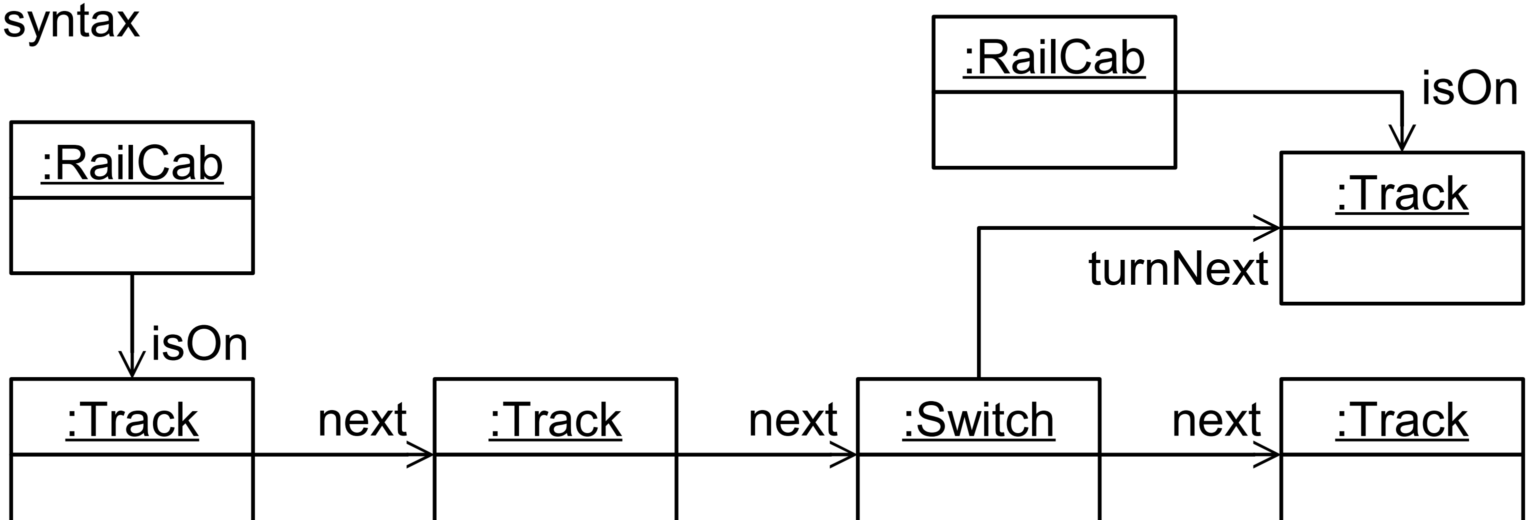
in the last lecture...

- Idea: View the model as a graph
- **Example**: train system “RailCab”

concrete syntax



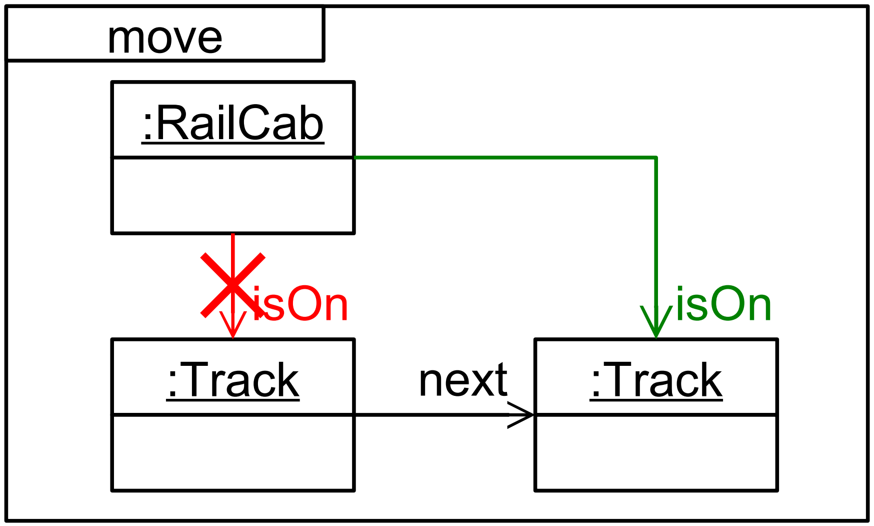
abstract syntax



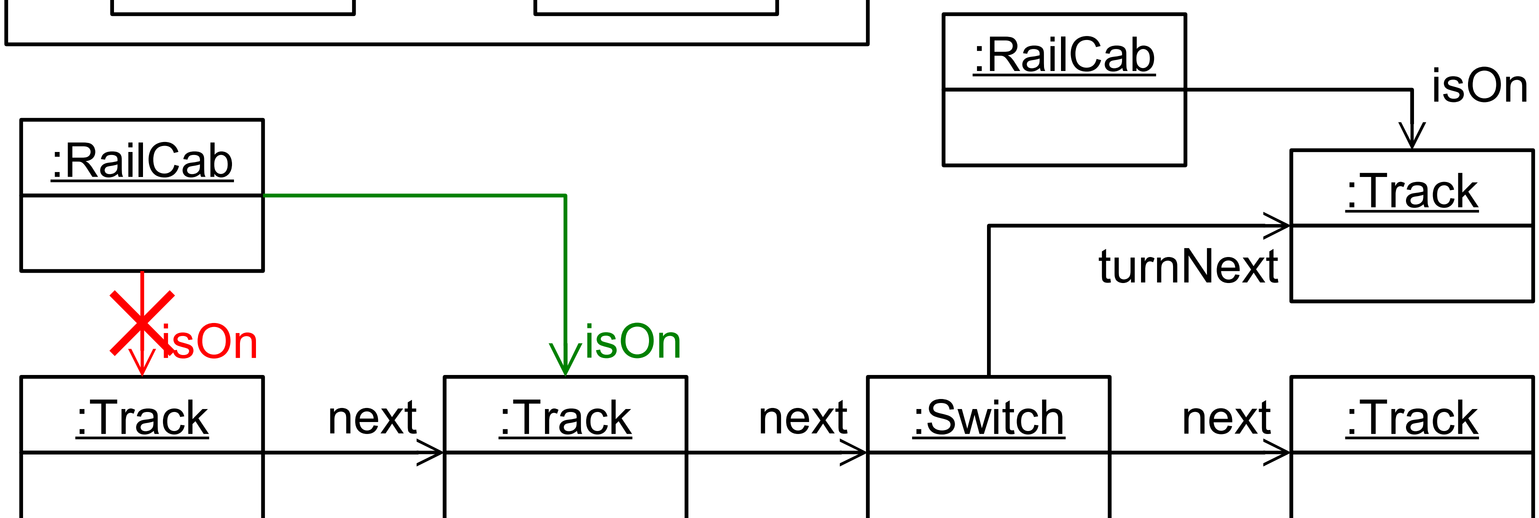
Graph Transformation Rule

in the last lecture...

- Describe the necessary **context of the change** and the **change itself** in a **graph transformation rule**



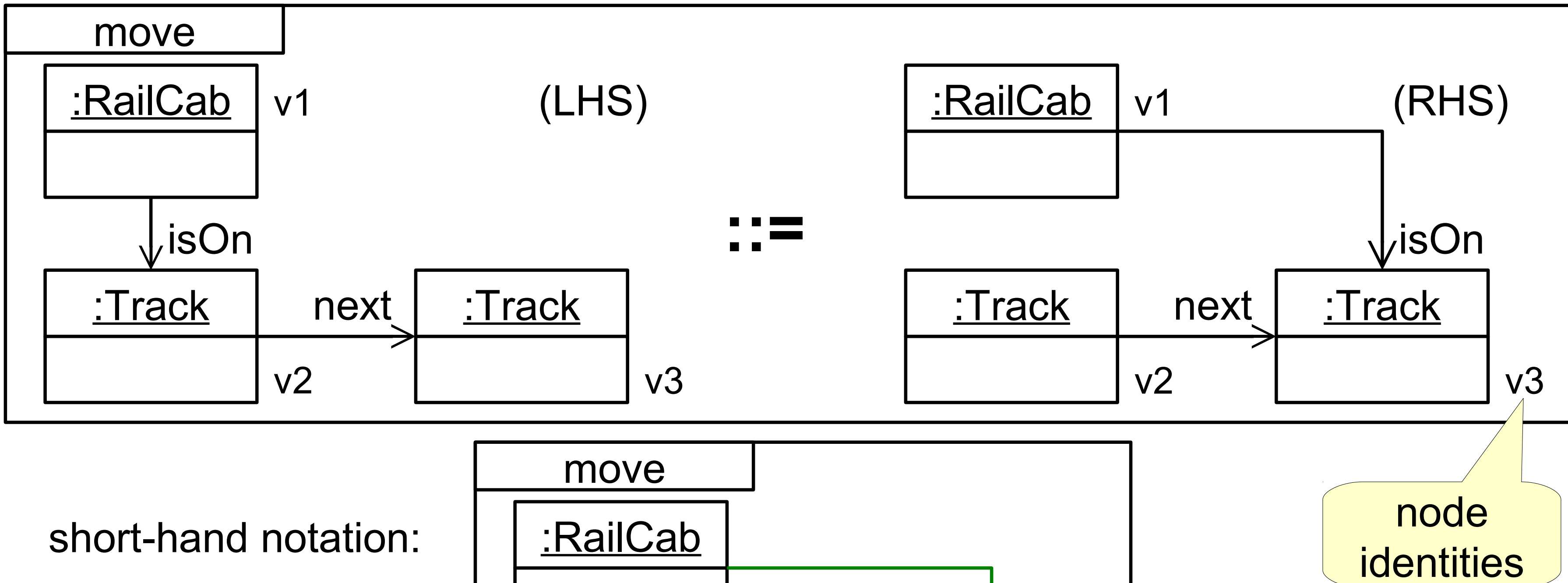
the rule's semantic is clear intuitively, but what does this mean exactly?



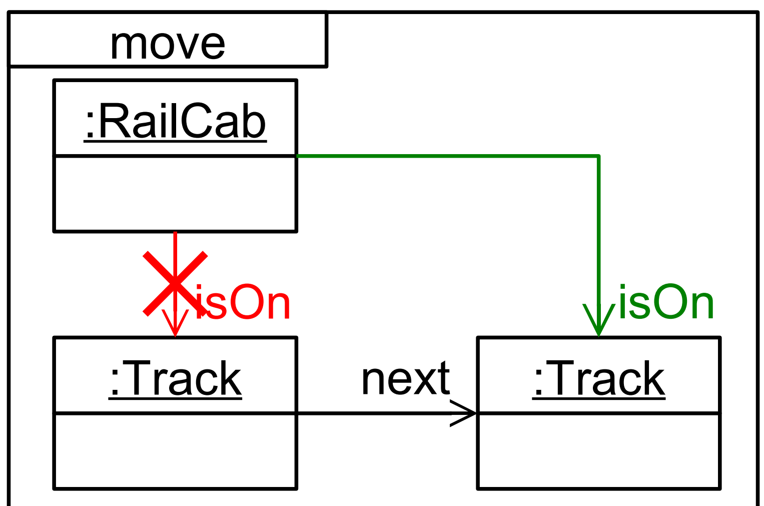
Graph Grammar Rule

in the last lecture...

- A graph grammar rule consists of two typed graphs
  - called **left-hand side** (LHS) and **right-hand side** (RHS)



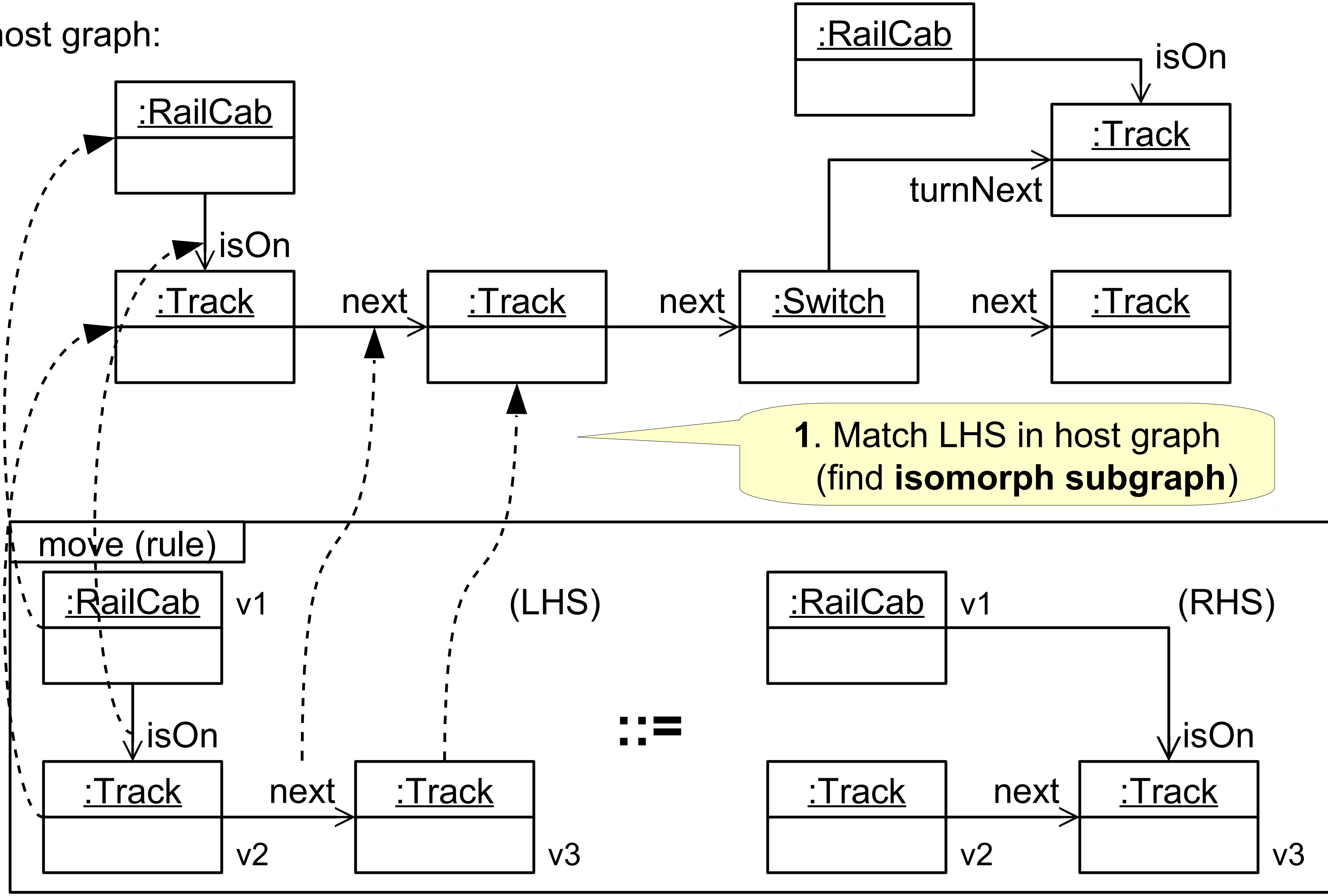
short-hand notation:



Graph Grammar Rule Application

in the last lecture...

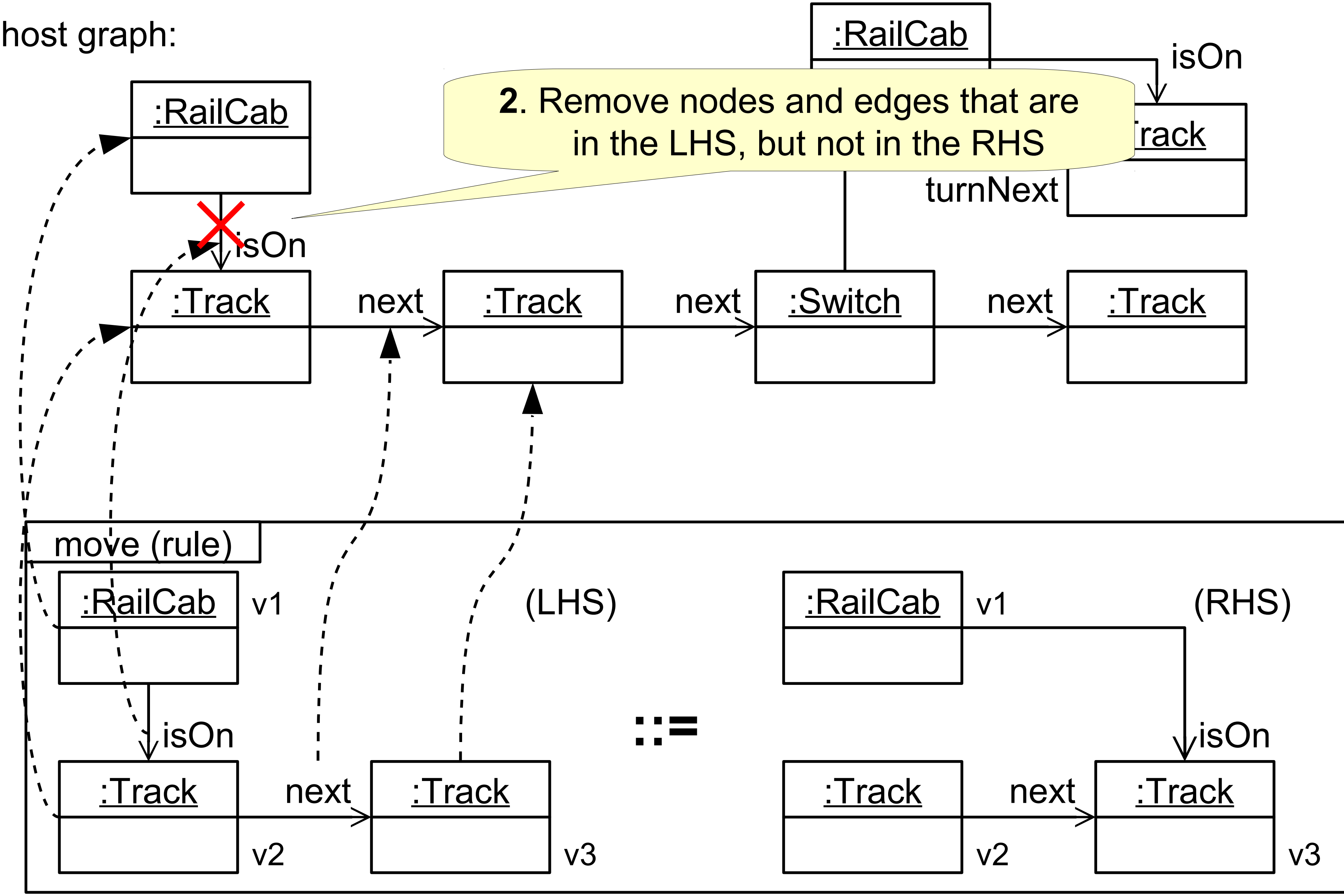
host graph:



Graph Grammar Rule Application

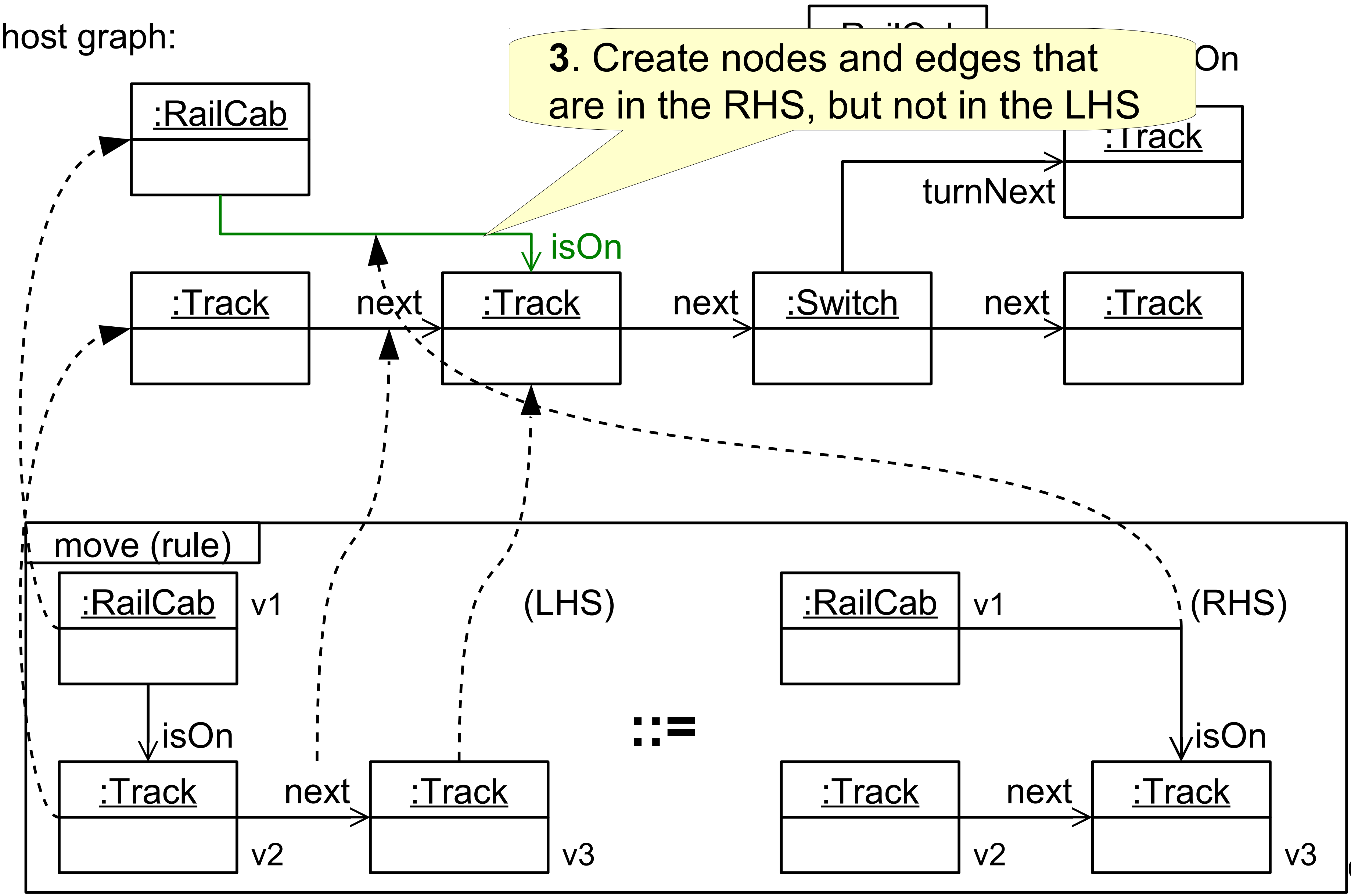
in the last lecture...

host graph:



Graph Grammar Rule Application

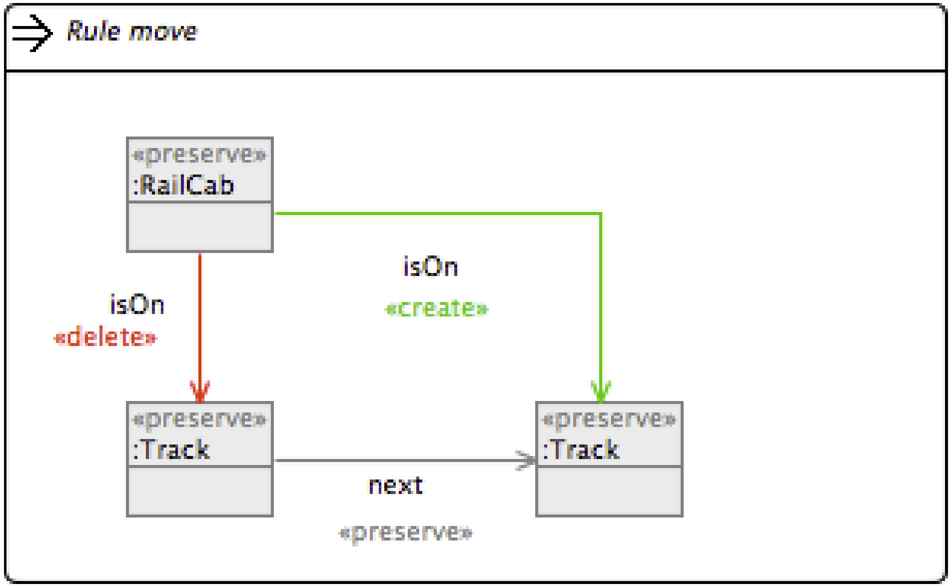
host graph:



Eclipse Henshin

in the last lecture...

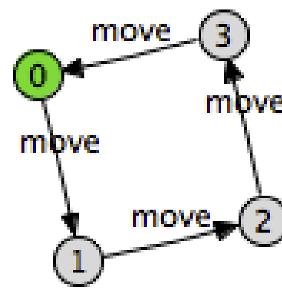
- An Eclipse project that supports the modeling, execution, and analysis of EMF-based graph transformation systems
  - https://www.eclipse.org/henshin/



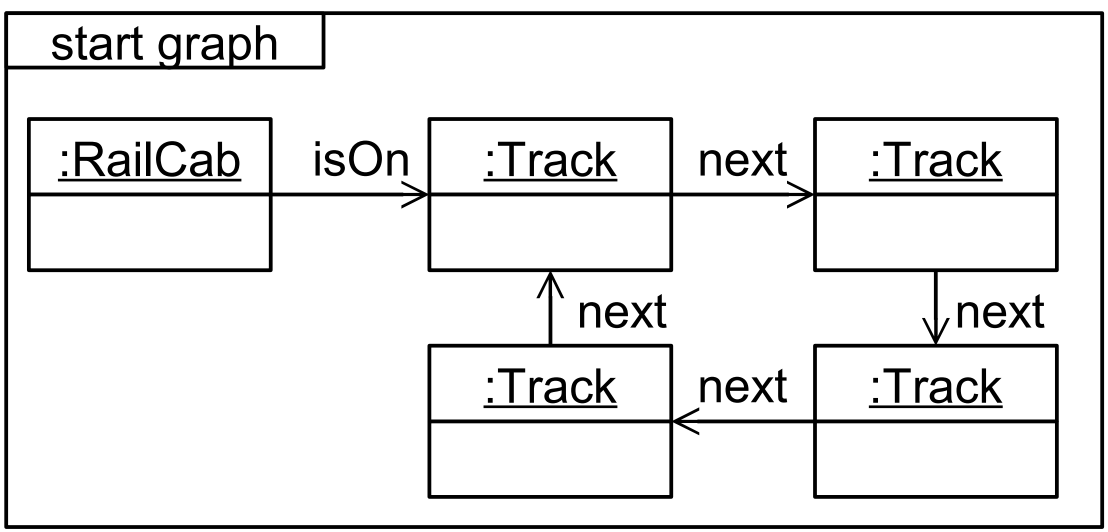
Exploring the State Space

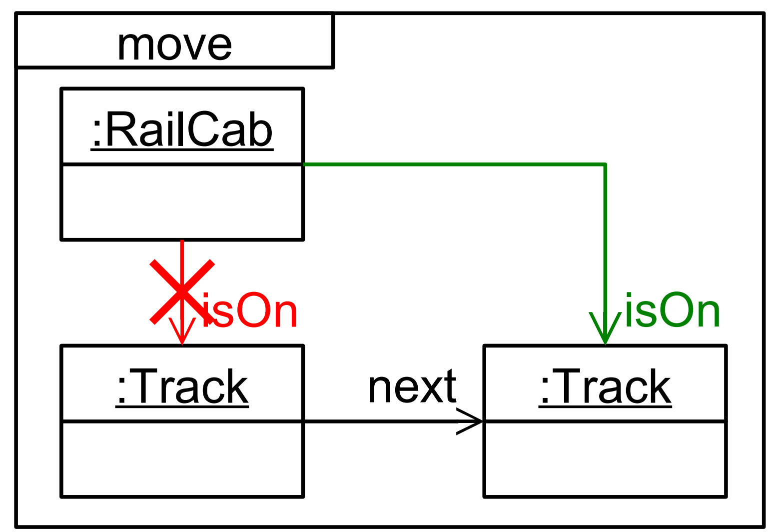
in the last lecture...

- A rule application can be considered a transition in a Labeled Transition System
  - source state: host graph before the rule application
  - transition: rule application
  - target state: host graph after the rule application



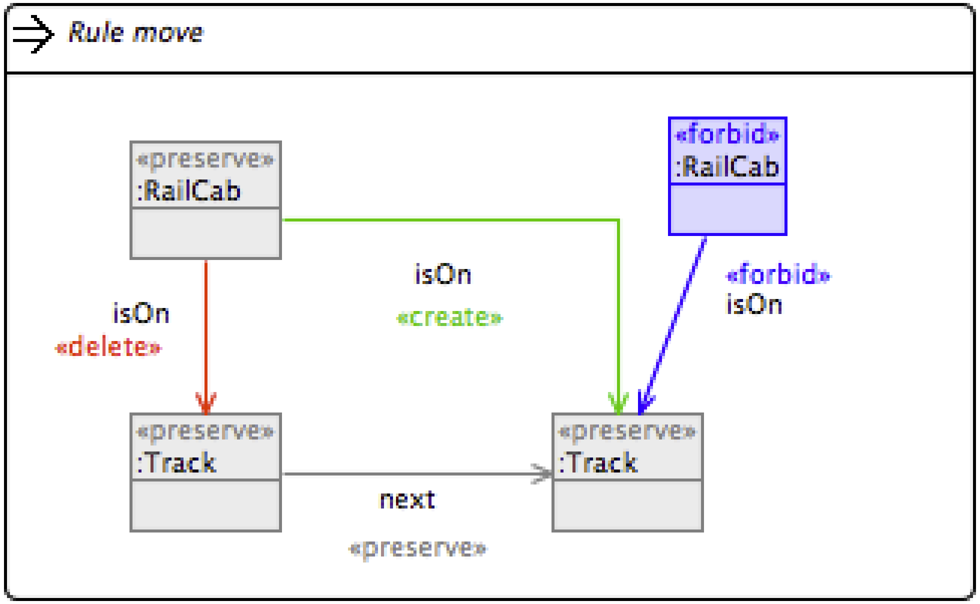
state space explored with Henshin: 4 different graphs; (graph after 4 applications of move rule is isomorphic⇒equal to the first)



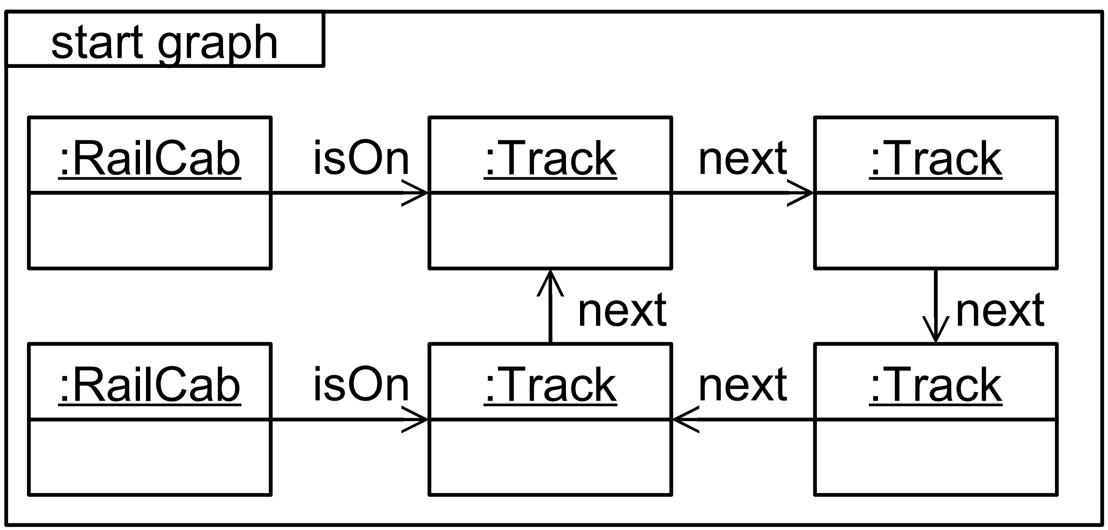


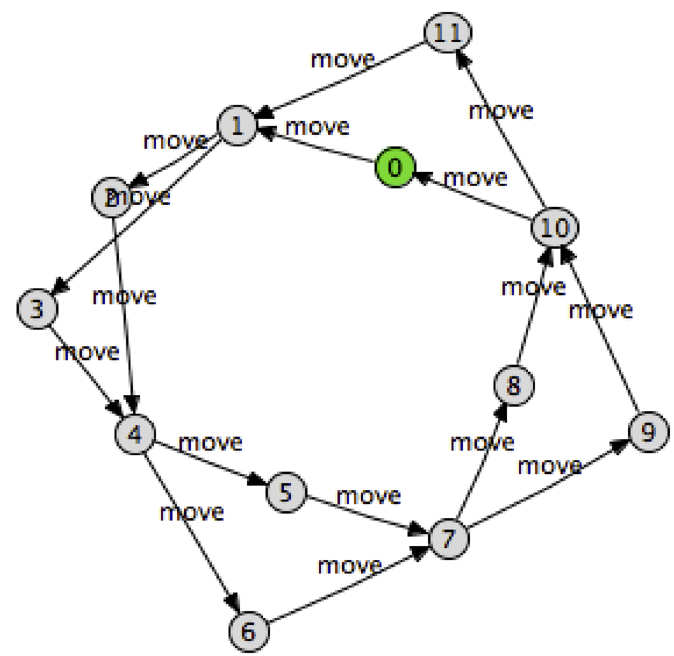
Exploring the State Space

in the last lecture...



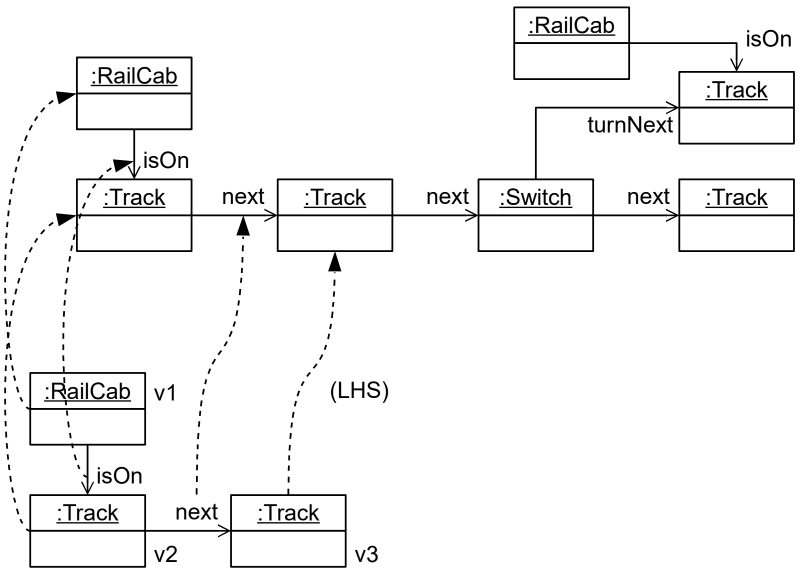
rule as specified
in Henshin





Graph Transformations More Formally

- A **match** of a rule graph in a host graph is a **typed graph isomorphism** between the rule graph and a subgraph of the host graph
  - What is a morphism?
  - What is a graph morphism?
  - What is a graph isomorphism?
  - What is a typed graph isomorphism?



Morphisms (Background)

- In mathematics, a **morphism** is a **structure-preserving mapping** from one mathematical structure to another

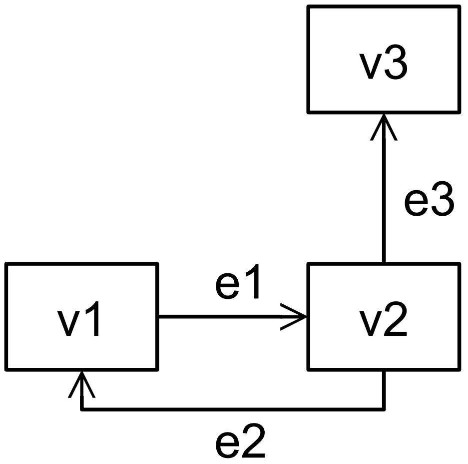
- Example: A **group (homo)morphism** is a function that maps one group to another in an **operation-preserving** way
  - **group**: a **set** of elements and an **operation** that maps any two elements from that set to a third element from that set
    - **example**: natural numbers with addition $(\mathbb{N}, +)$
  - given two groups $(G_1, *)$ and $(G_2, \#)$, a **homomorphism** $h\colon G_1 \rightarrow G_2$ is an operation-preserving function, i.e., for $a, b \in G_1$ it holds that $h(a * b) = h(a) \# h(b)$
    - **example**: given (STRING, ·) and $(\mathbb{N}_{\geq 0}, +)$, then $length\colon STRING \rightarrow \mathbb{N}$ is a homomorphism, since for two strings a and b, it holds that $length(a \cdot b) = length(a) + length(b)$ *(“ · ” means the concatenation of two strings)*

Definition: Graph

- We define a graph G as a tuple $G = (V, E, s, t)$ where
  - V is a finite set of nodes (vertices)
  - E is a finite set of edges
  - $s: E \rightarrow V$ is the source function, defining edges' source nodes
  - $t: E \rightarrow V$ is the target function, defining edges' target nodes

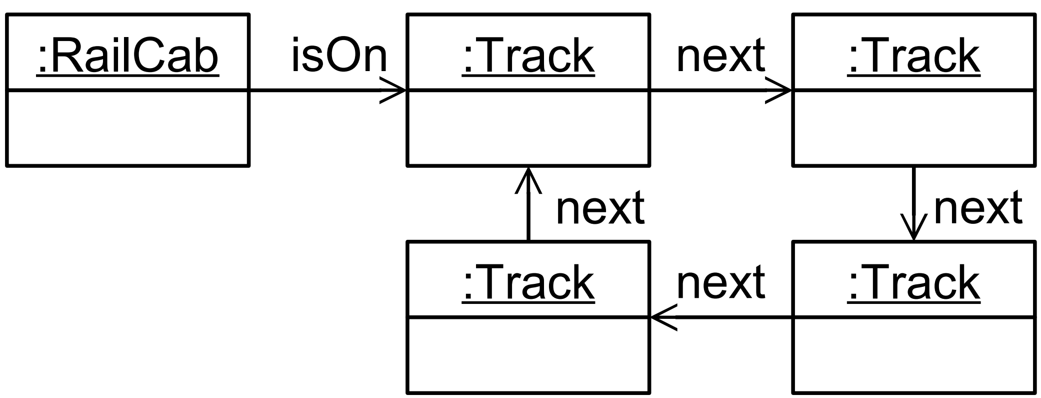
- Example: How to interpret this graph mathematically?
  - $V = \{v1, v2, v3\}$
  - $E = \{e1, e2, e3\}$
  - $s = \{(e1, v1), (e2, v2), (e3, v2)\}$
  - $t = \{(e1, v2), (e2, v1), (e3, v3)\}$

  We also write for example $s(e1) = v2$



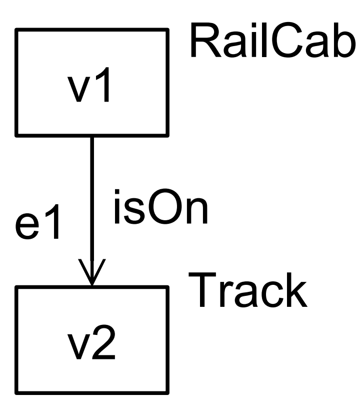
Labeled Graph

- Problem: How to formalize the following graph?
  - multiple nodes called “:Track”
  - multiple edges called “isOn”
- Element names are the same, but identities are not
- We cannot have a set with the same element occurring multiple times, e.g. $\cancel{E = \{isOn, next, next, next, next\}}$
- Solution: Model labels explicitly



Labeled Graph

- **Labels**: Giving names to nodes and edges
- A graph G can be extended by labeling functions L_V and L_E
  - $L_V: V \to \Sigma$ is a node labeling function
  - $L_E: V \to \Sigma$ is an edge labeling function
  - Σ: is a set of labels
- Example: How to interpret the given graph mathematically?
  - $V = \{v1, v2\}$
  - $E = \{e1\}$
  - $s = \{(e1, v1)\}$
  - $t = \{(e1, v2)\}$
  - $L_V = \{(v1, RailCab), (v2, Track)\}$
  - $L_E = \{(e1, isOn)\}$



Graph Morphism

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

  that preserve the source and target functions, i.e.,
  - $f_V \circ s_1 = s_2 \circ f_E$ and
  - $f_V \circ t_1 = t_2 \circ f_E$
- Example: for these two graphs a graph morphism would be
  - $f_V = \{(v_{11}, v_{21}), (v_{12}, v_{22})\}$
  - $f_E = \{(e_{11}, e_{21})\}$

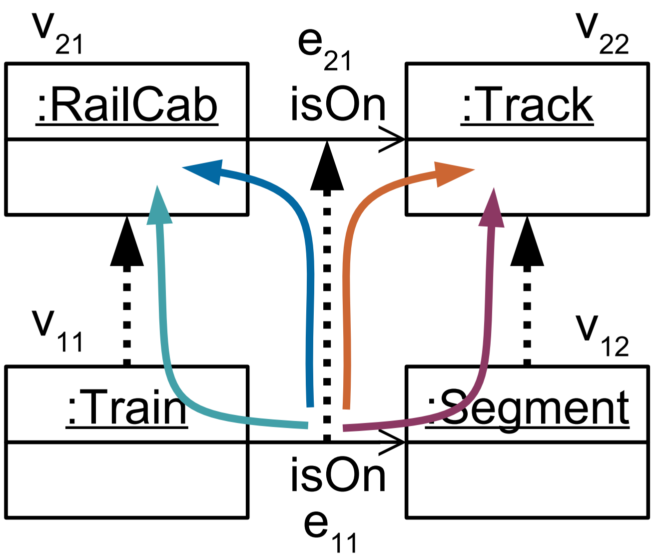
Example:

$f_V(s_1(e11))$

$= s_2(f_E(e11))$

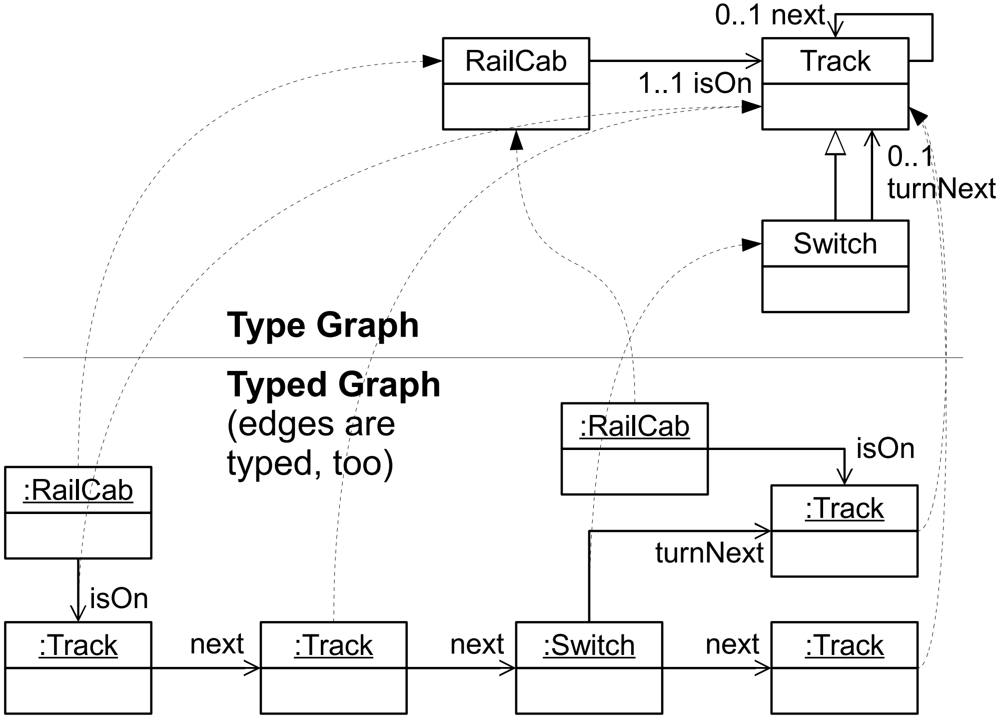
$f_V(t_1(e11))$

$= t_2(f_E(e11))$



Typed Graph

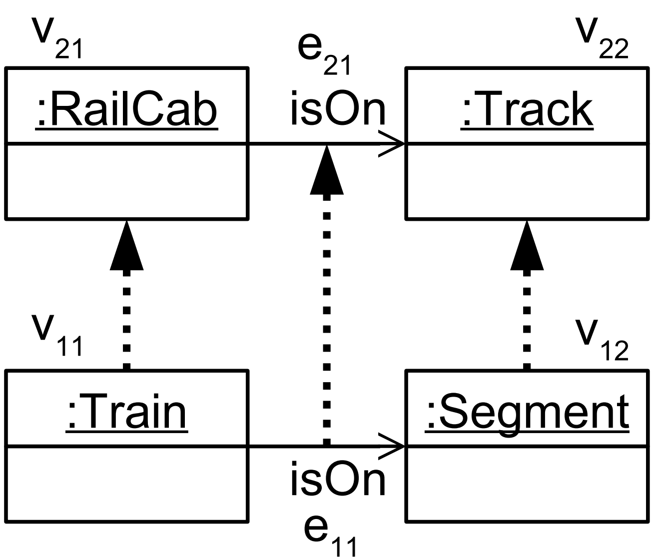
- A graph G can be **typed** by giving a **graph morphism** $type: G \rightarrow G_{Type}$
  - G_{Type} is the **type graph,** the tuple $(G, type)$ is the **typed graph**
- A graph morphism (also **graph homomorphism**) is a **total function**
  - f_V and f_E are total
  - every element in the domain (typed graph) has to be related to exactly one element of the co-domain (type graph)



Graph Isomorphism

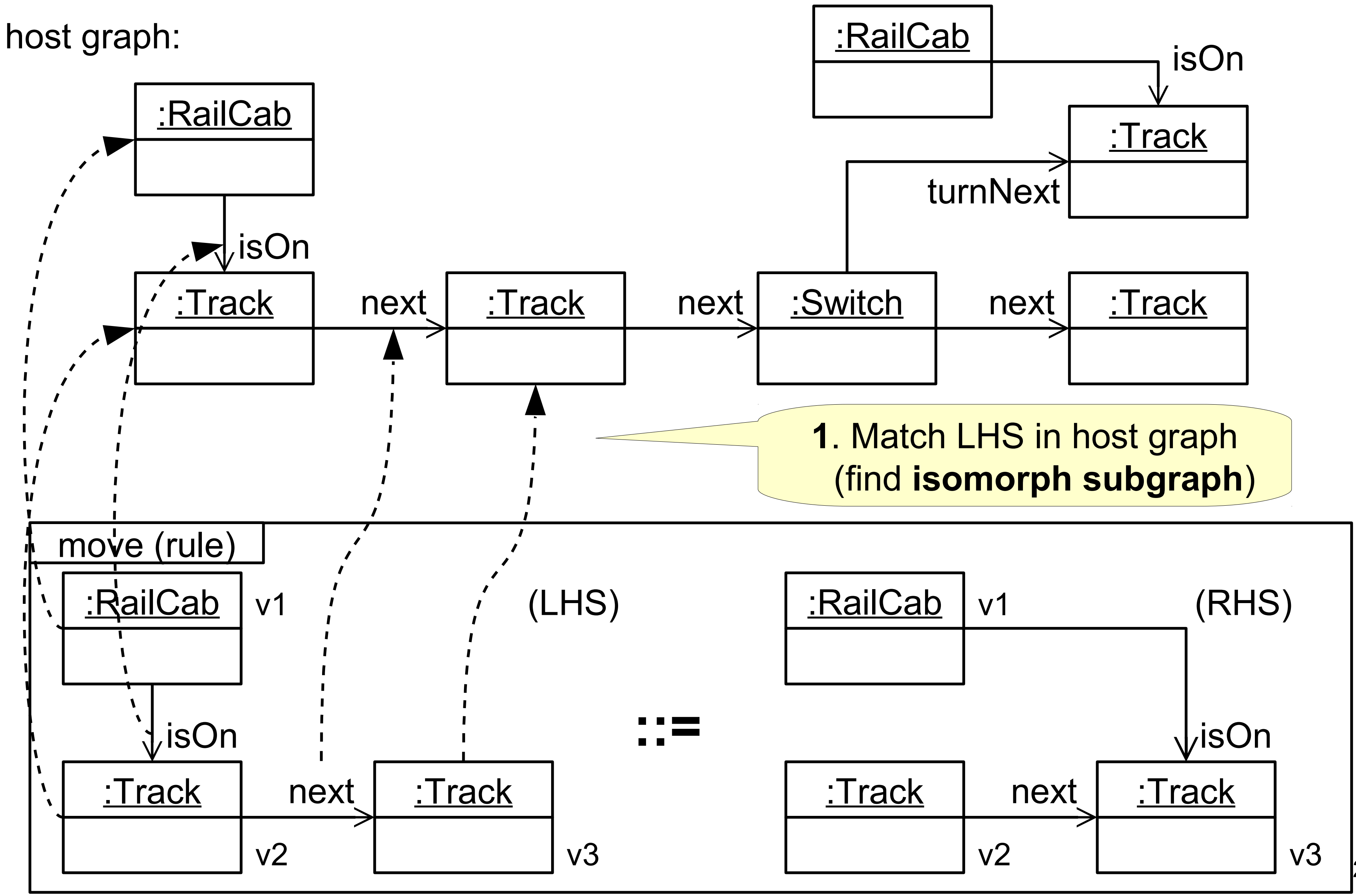
- A graph morphism $f = (f_V, f_E)$ is called a **graph isomorphism** if f_V and f_E are **bijective**
  - each element in the domain corresponds to exactly one element of the co-domain
  - the graph morphism is reversible

- Example:
  (from before)



Graph Grammar Rule Application

in the last lecture...



Subgraph

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

It means that the source and target functions for the subgraph are reduced to the edges which are in it.

- If G_{Sub} is a subgraph of G, we also write $G_{Sub} \leq G$

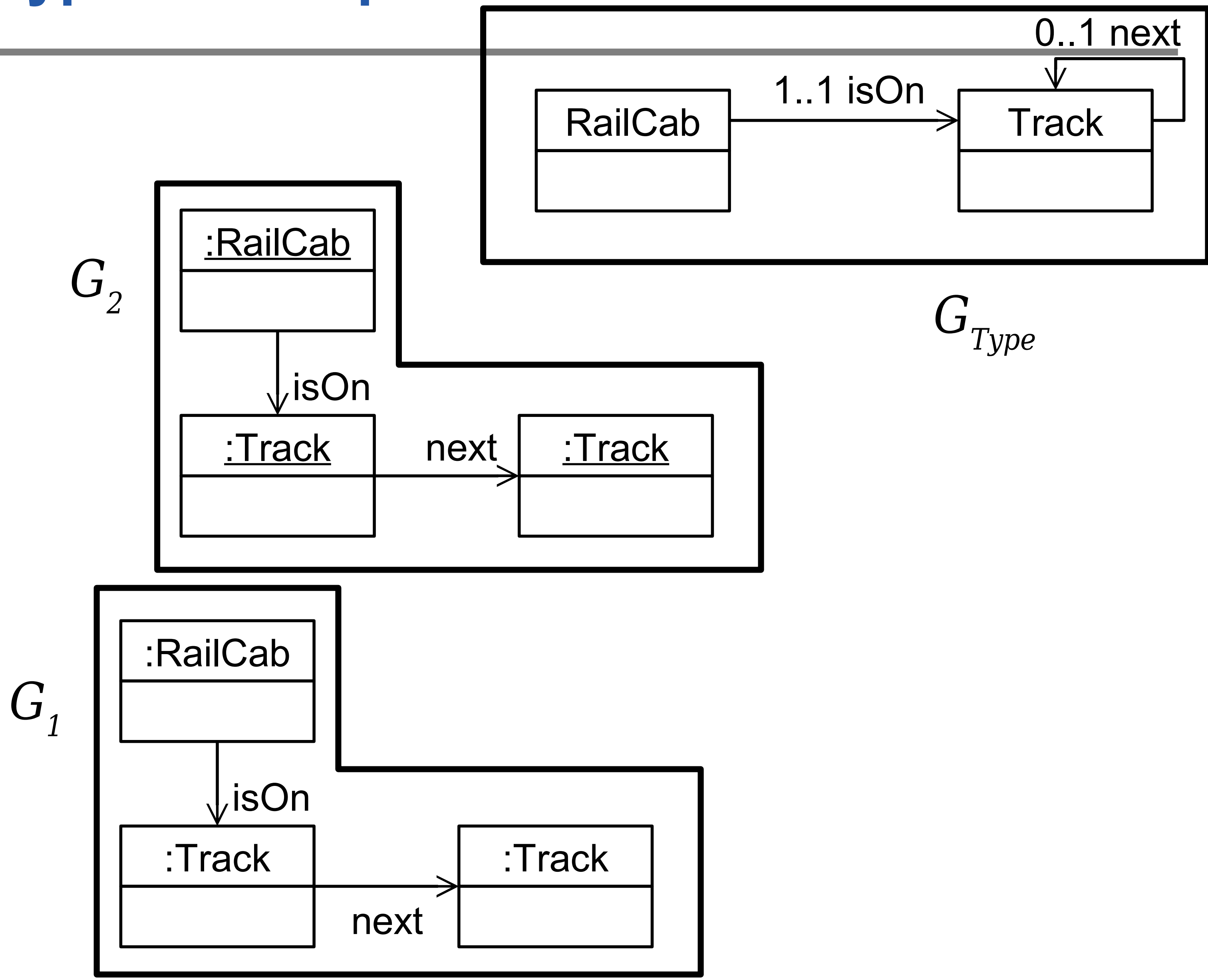
Typed Isomophism

- When matching a rule graph to the host graph:
  - rule graph and host graph have the **same type graph**
  - the match must **respect the typing of the graphs**
  - there must be a ***typed subgraph isomorphism***

- Let $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$ be two graphs
  - typed by graph morphisms
    $type_1 = (type_{1V}, type_{1E}): G_1 \rightarrow G_{Type}$ and
    $type_2 = (type_{2V}, type_{2E}): G_2 \rightarrow G_{Type}$

- An **(iso)morphism** $f = G_1 \longrightarrow G_2, f = (f_V, f_E)$ is **typed** when
  - $type_1 = type_2 \circ f$, i.e.,
    - for all $v \in V_1$ it holds that $type_{1V}(v) = type_{2V}(f_V(v))$ and
    - for all $e \in E_1$ it holds that $type_{1E}(e) = type_{2E}(f_E(e))$

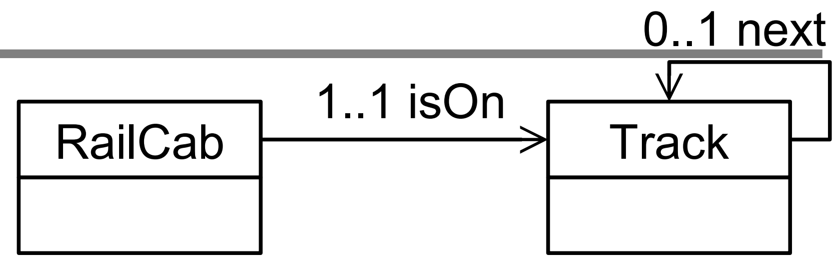
Typed Isomophism



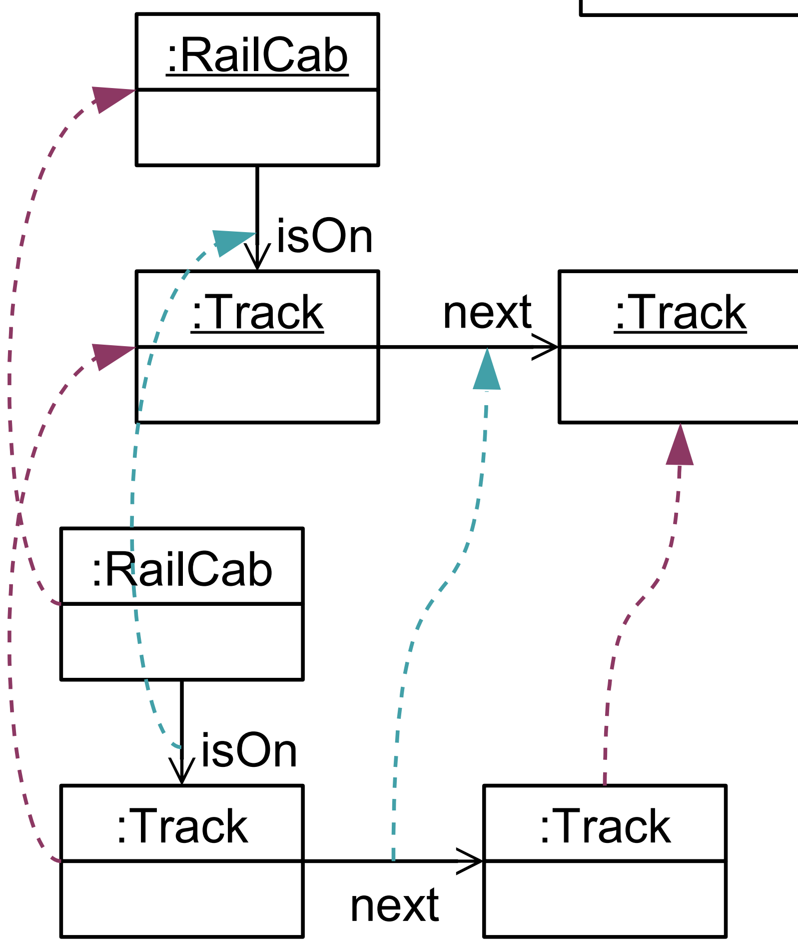
Typed Isomophism

for all $v \in V_1$ $type_{1V}(v) = type_{2V}(f_V(v))$

for all $e \in E_1$ $type_{1E}(e) = type_{2E}(f_E(e))$



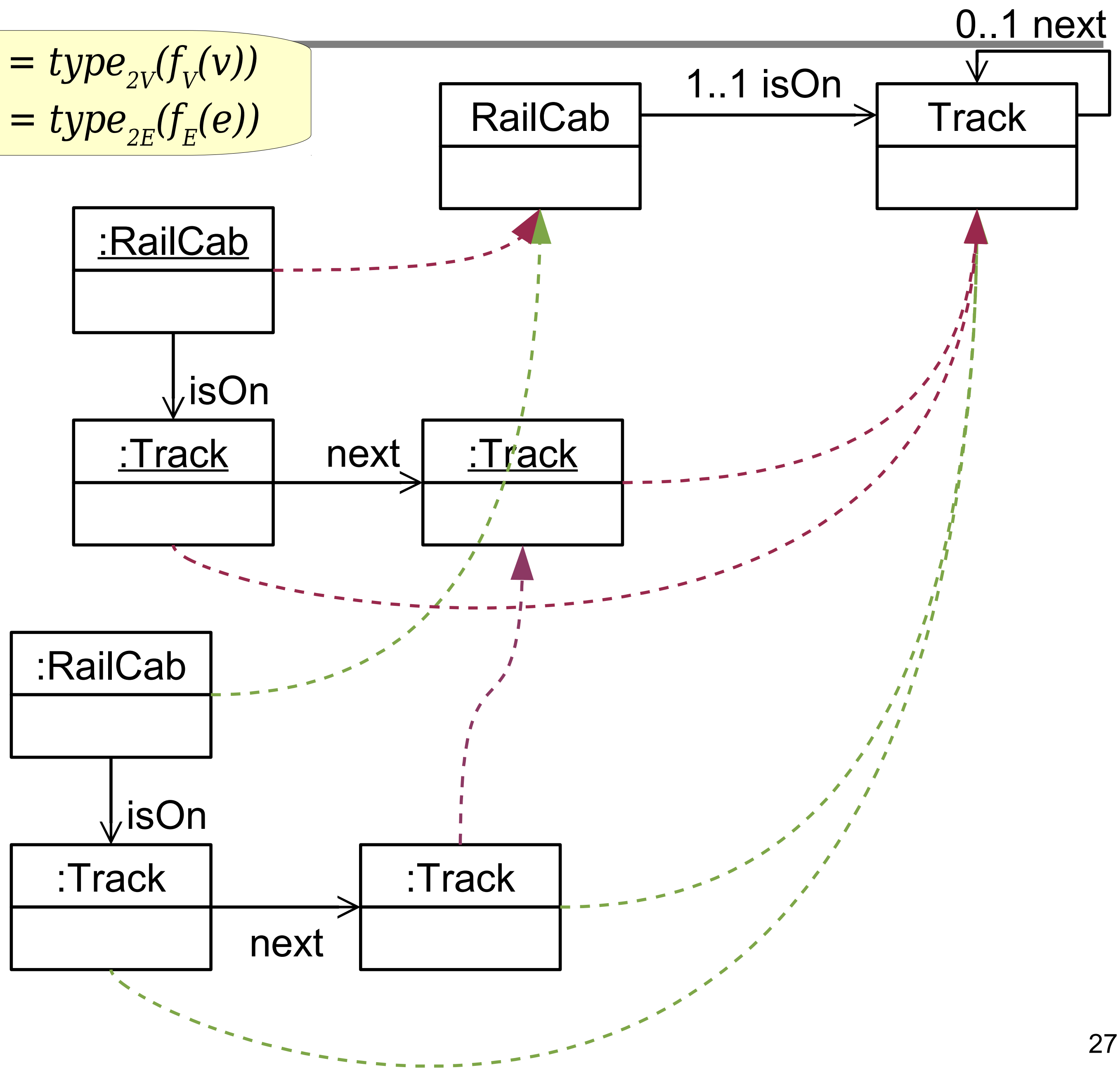
f_V and f_E



Typed Isomophism

for all $v \in V_1$ $type_{1V}(v) = type_{2V}(f_V(v))$

for all $e \in E_1$ $type_{1E}(e) = type_{2E}(f_E(e))$



$type_{1V}$ and

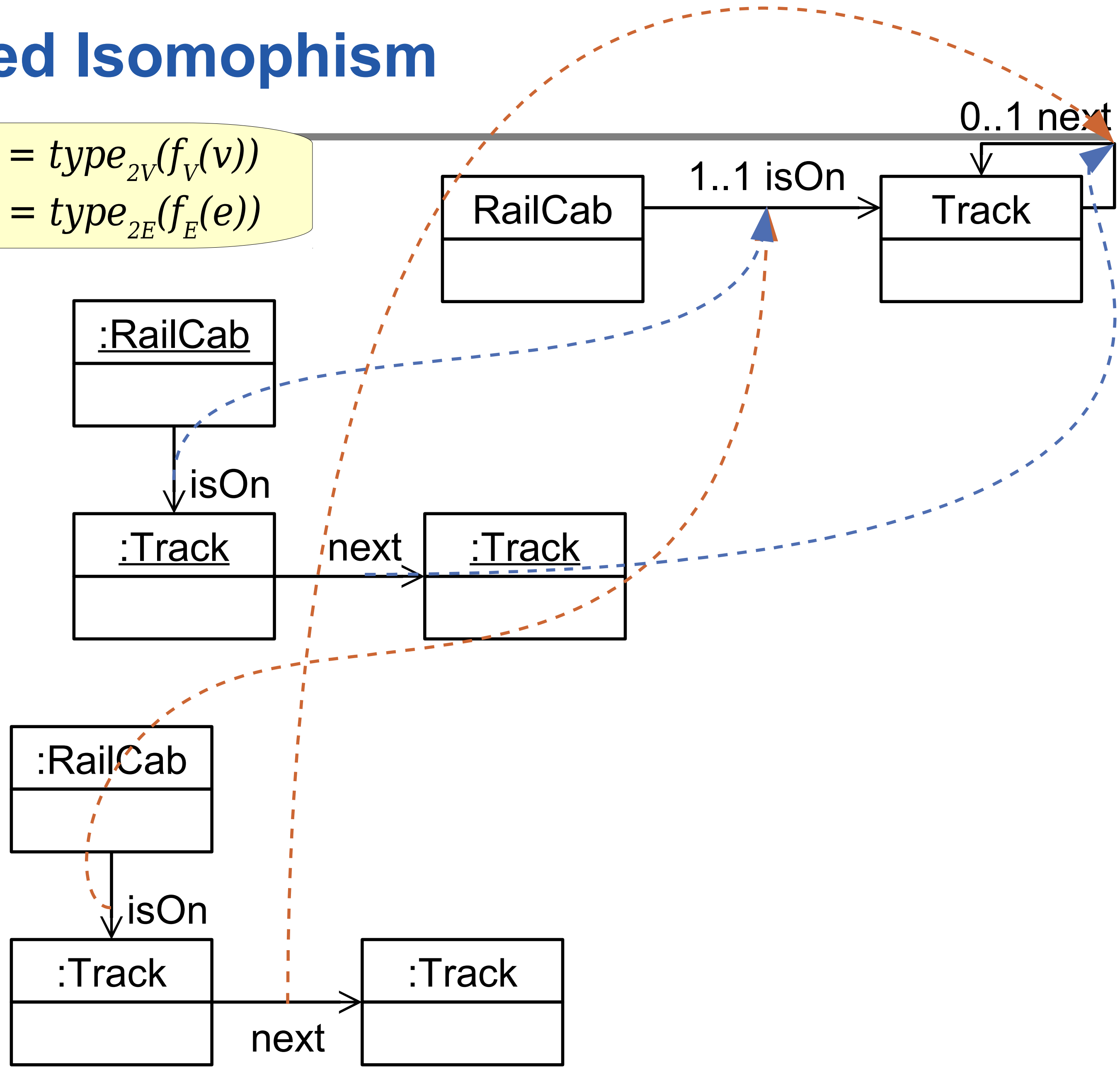
$type_{2V}$

Typed Isomophism

for all $v \in V_1$ $type_{1V}(v) = type_{2V}(f_V(v))$

for all $e \in E_1$ $type_{1E}(e) = type_{2E}(f_E(e))$

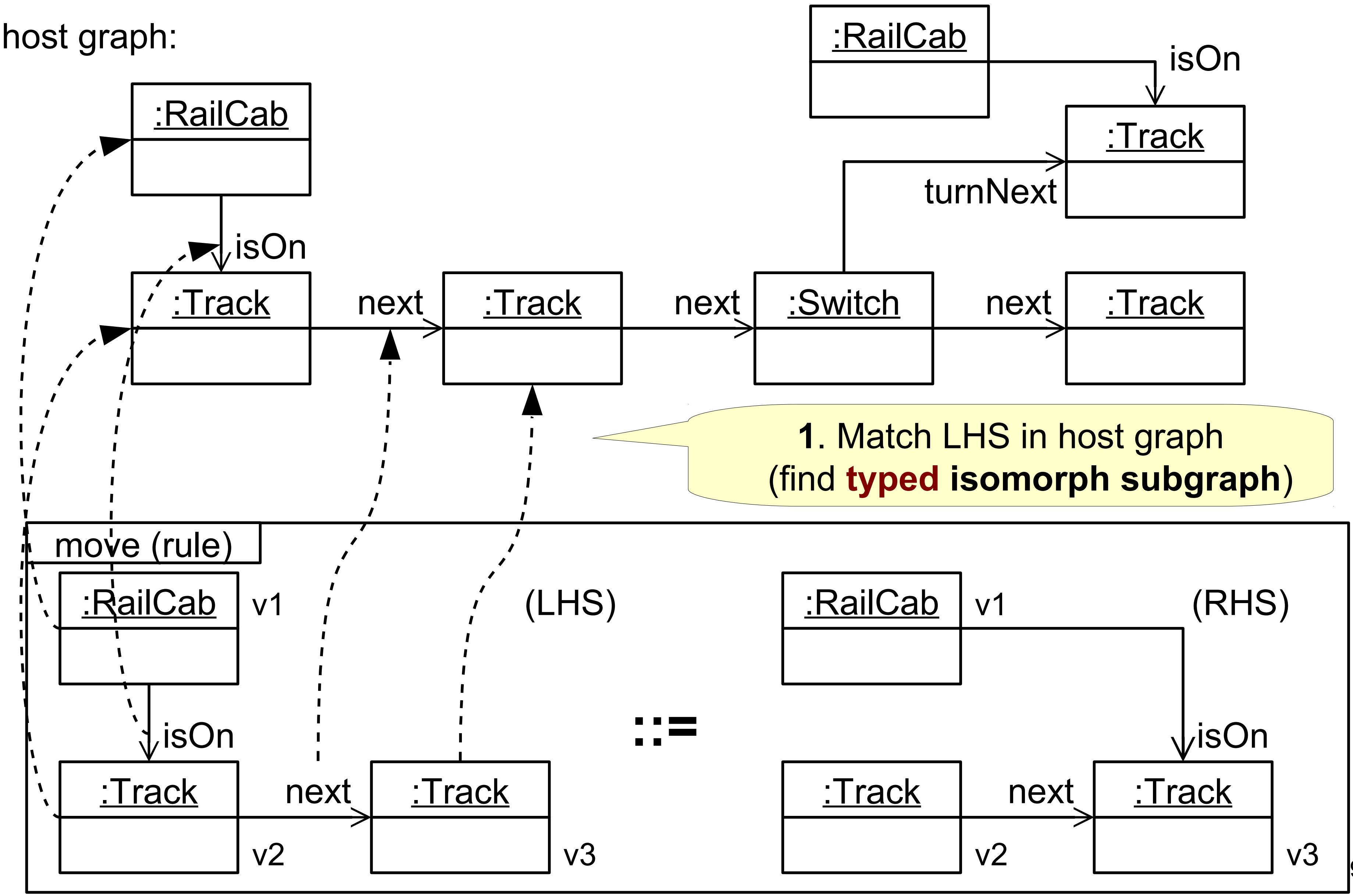
$type_{1E}$ and $type_{2E}$



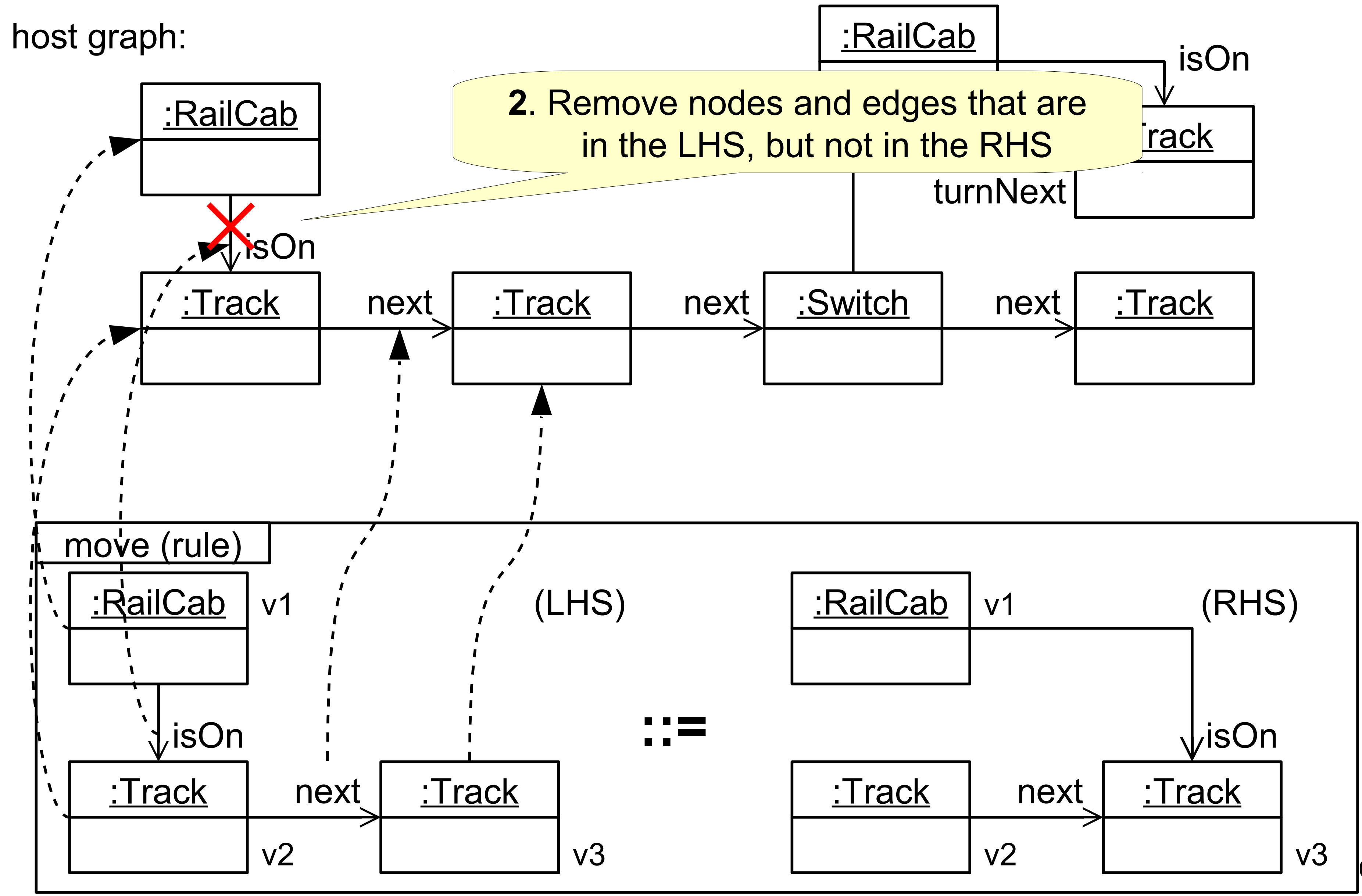
Graph Grammar Rule Application

in the last lecture...

host graph:

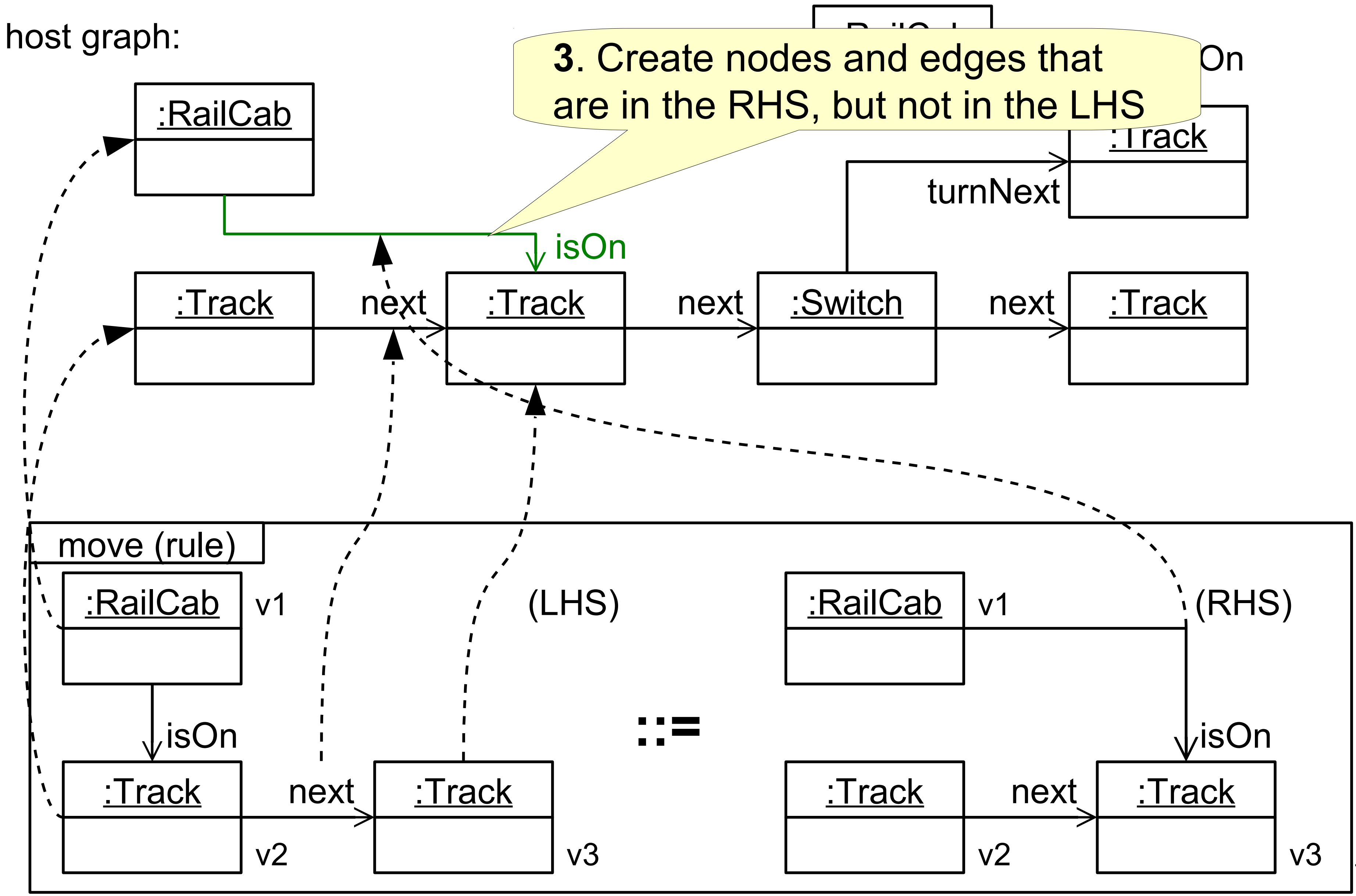


Graph Grammar Rule Application



Graph Grammar Rule Application

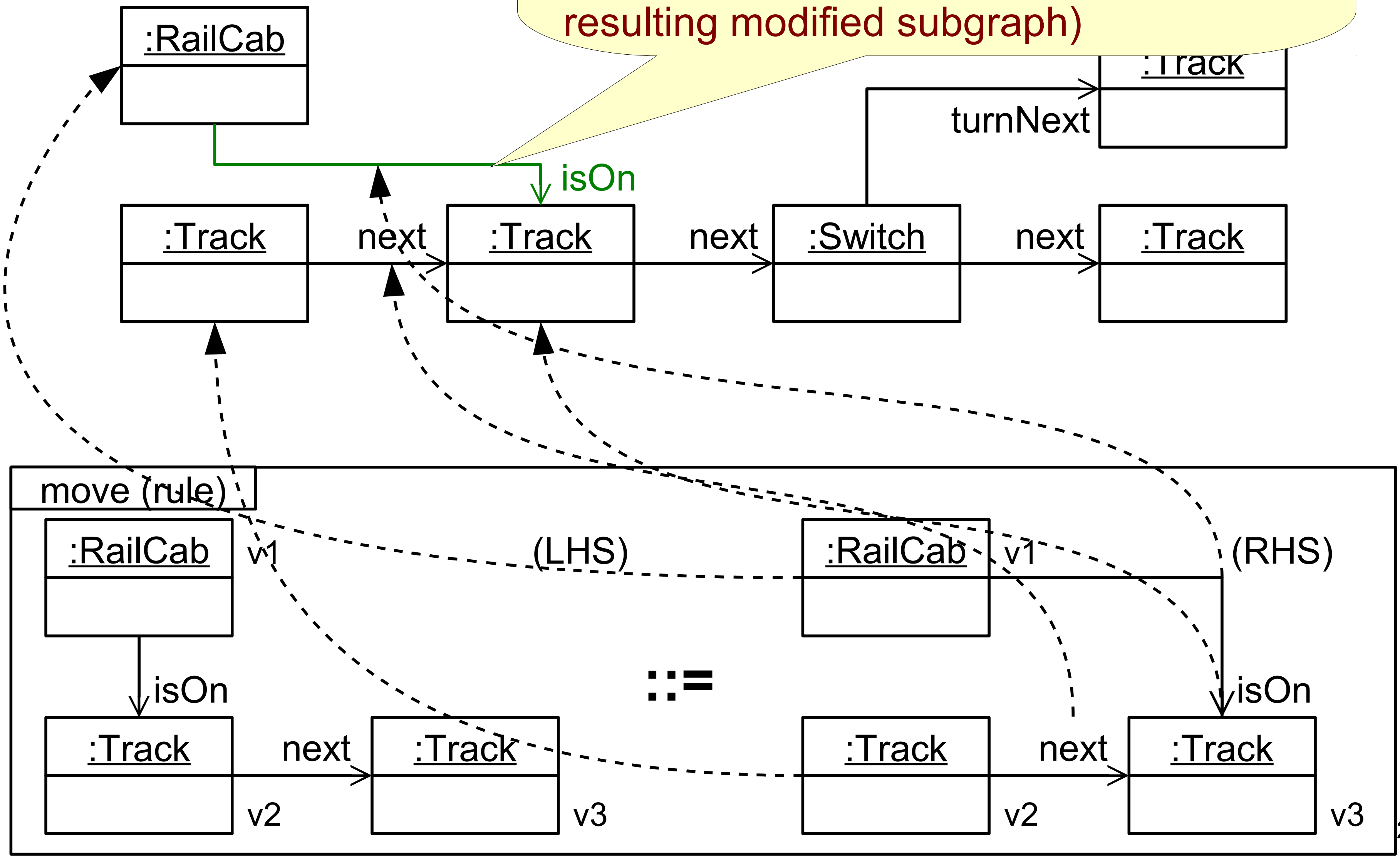
in the last lecture...



Graph G

3. Create nodes and edges that are in the RHS, but not in the LHS
(such that there is **typed isomorphism** between the rule's RHS graph and the resulting modified subgraph)

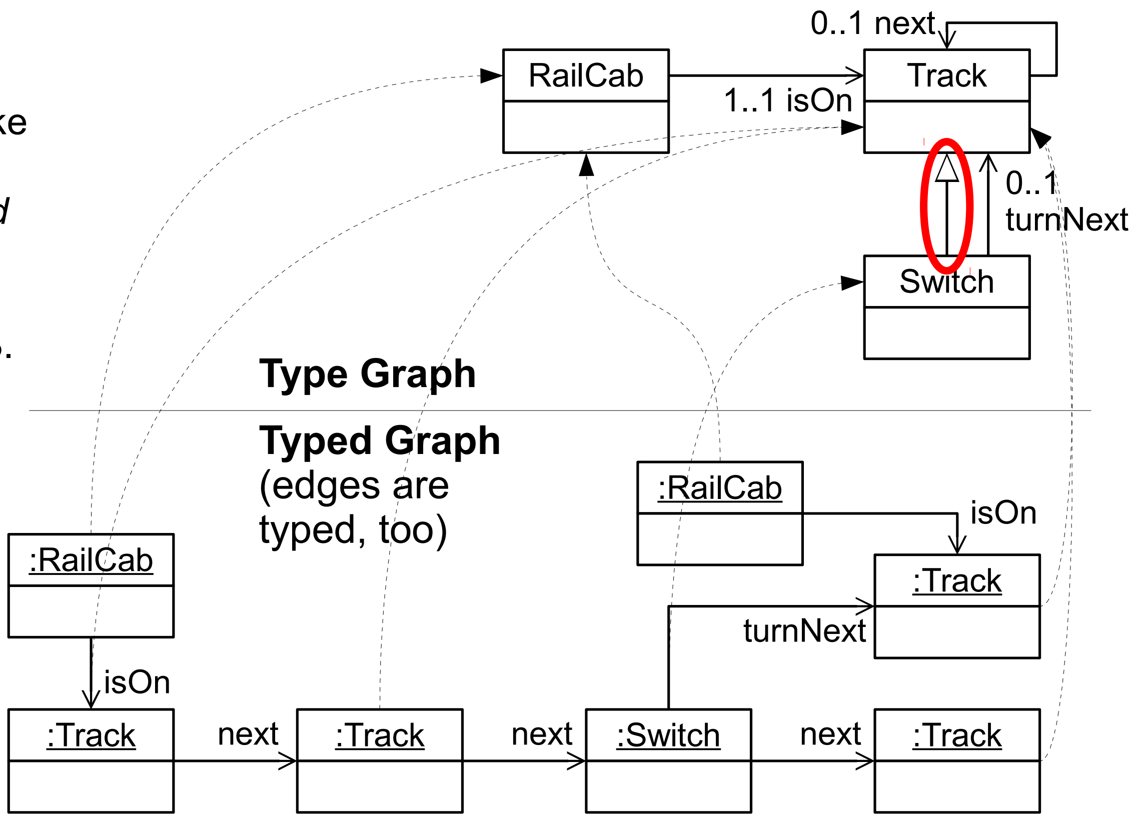
host graph:



Graph Transformations More Formally

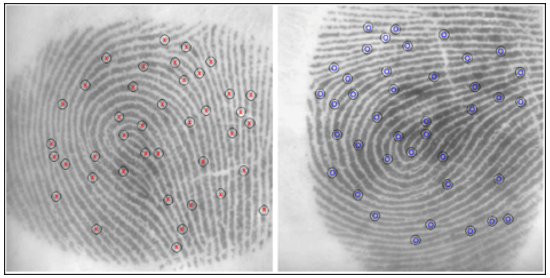
- How do we treat the concept of generalization (inheritance)?
  - this makes matters a bit more complicated...
- How do we treat attribute values?

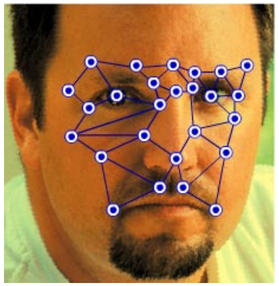
see for example: Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer, Fundamental Aspects of Software Engineering, *Attributed graph transformation with node type inheritance*, Theoretical Computer Science, Volume 376, Issue 3, 2007, Pages 139-163.



Graph Matching Algorithm

- Finding an isomorphic subgraph is an NP-complete problem
  - exponential in the size of the involved graphs
- In the MBSE context, the graphs are usually typed and often strongly structured
  - so matching graph transformation rule patterns can happen in practically acceptable time
- In some applications, graphs are not that structured, but then also heuristics can be employed to find close matches





Graph Transformations – Intermediate Summary

- Graph transformation rules allow us
  - to describe the behavior of systems (e.g. RailCab) formally
  - to describe the behavior of object-oriented programs formally

- The behavior can be analyzed formally

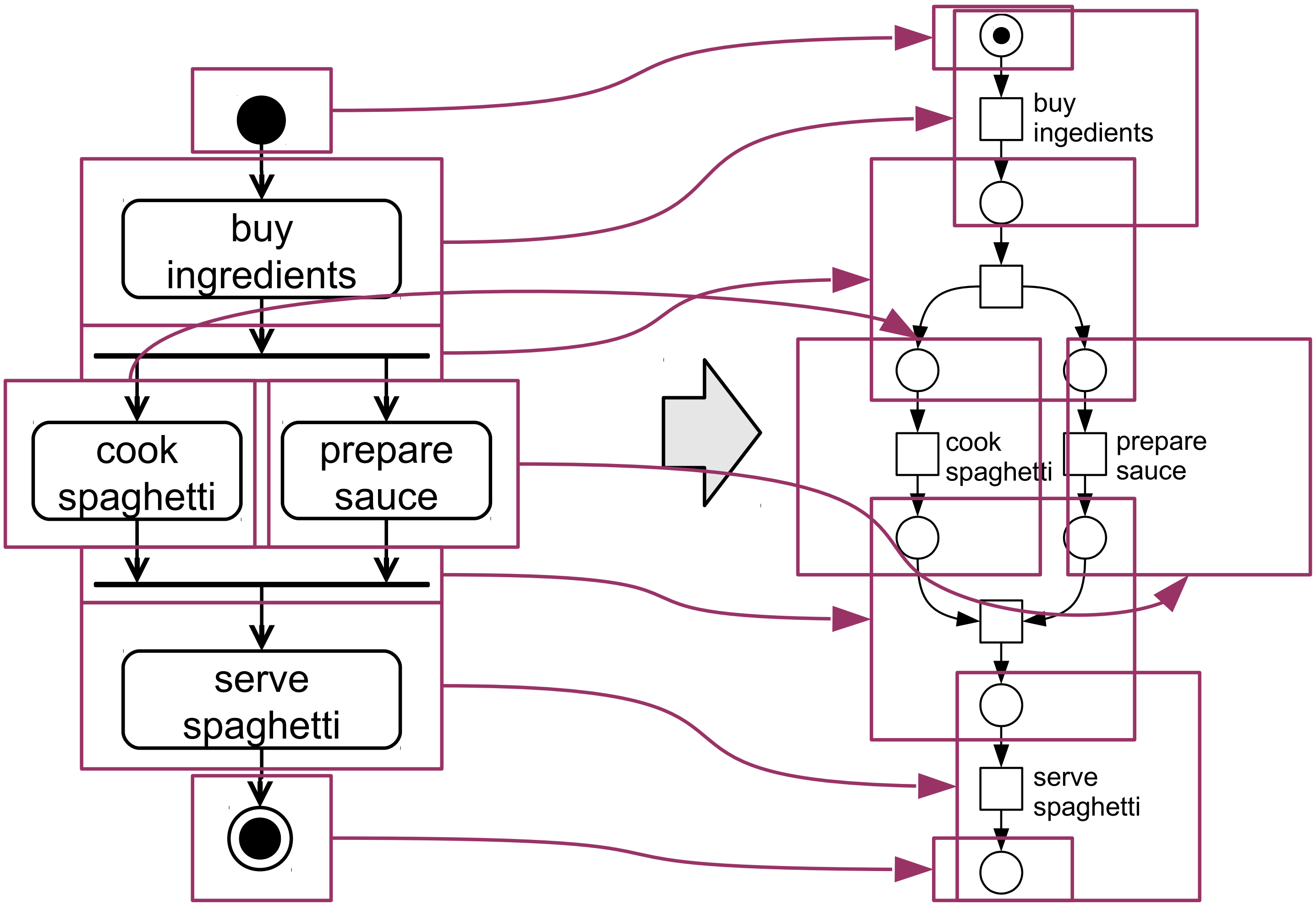
- And the behavior can be execute
  - by and interpreter (like Henshin)
  - or by code generation (like SDMTools
    https://www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/story-diagram-tools/)

- So far, we have mainly considered **endogenous** model transformations, how about **exogenous** ones?

5.4. Model-to-model transformation – Triple Graph Grammars

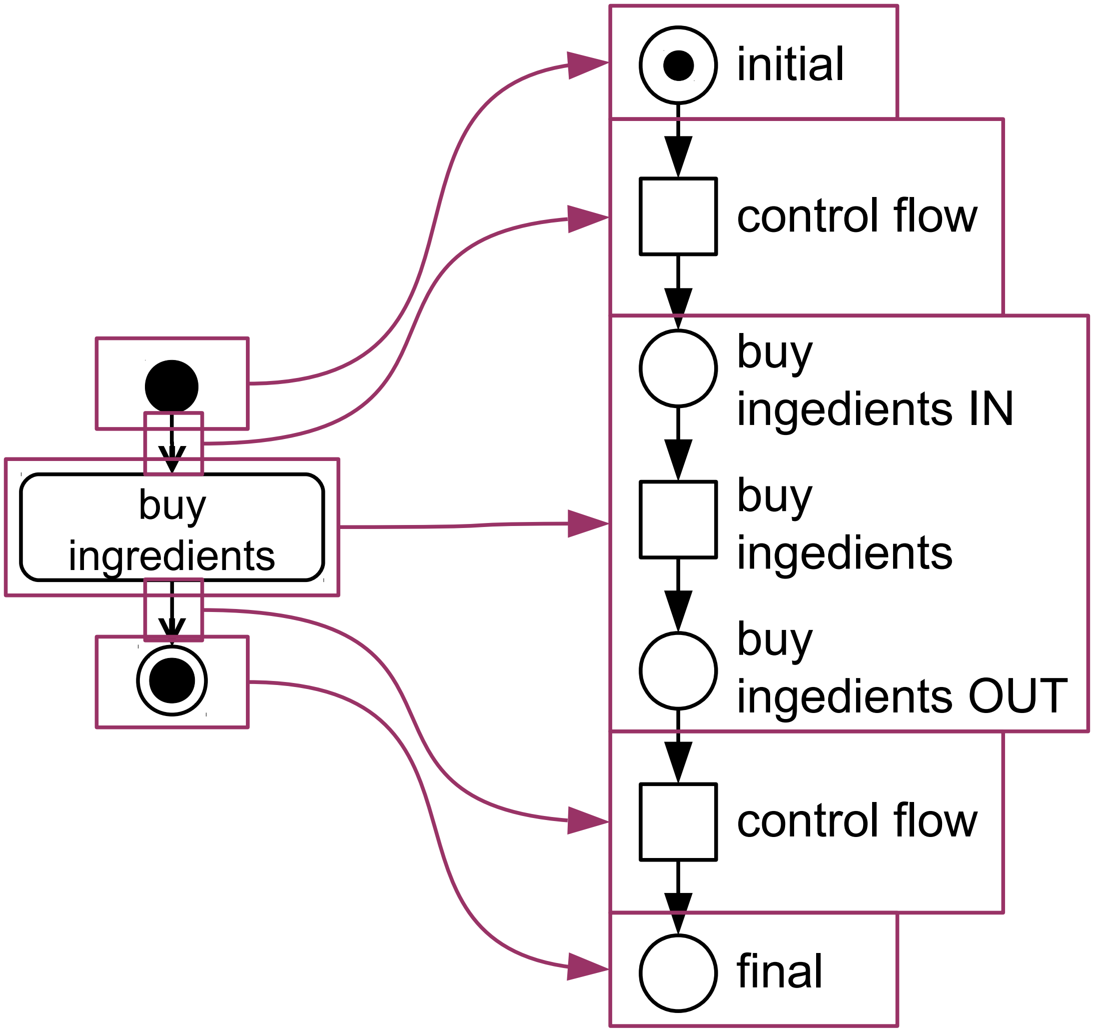
Exogenous Model Transformations

- Example: transform Activity Diagrams to Petri nets



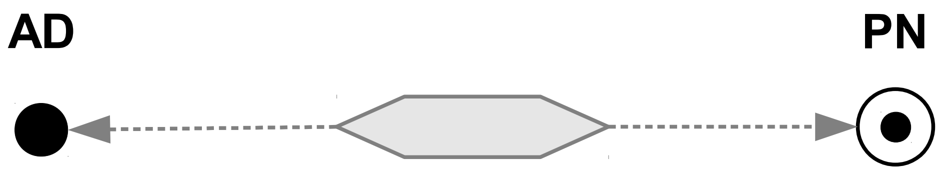
Example:Transform Activity Diagrams to Petri nets

- Let's start simple: How to transform
  - Initial nodes?
  - Final nodes?
  - Action nodes?
  - Control flow edges?

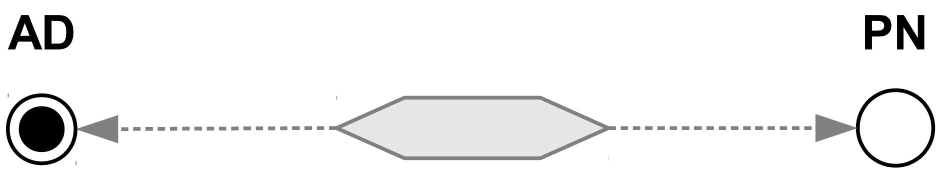


Relations Between Model Patterns Example: Activity to Petri net

- Intial node ↔ Place with intial marking

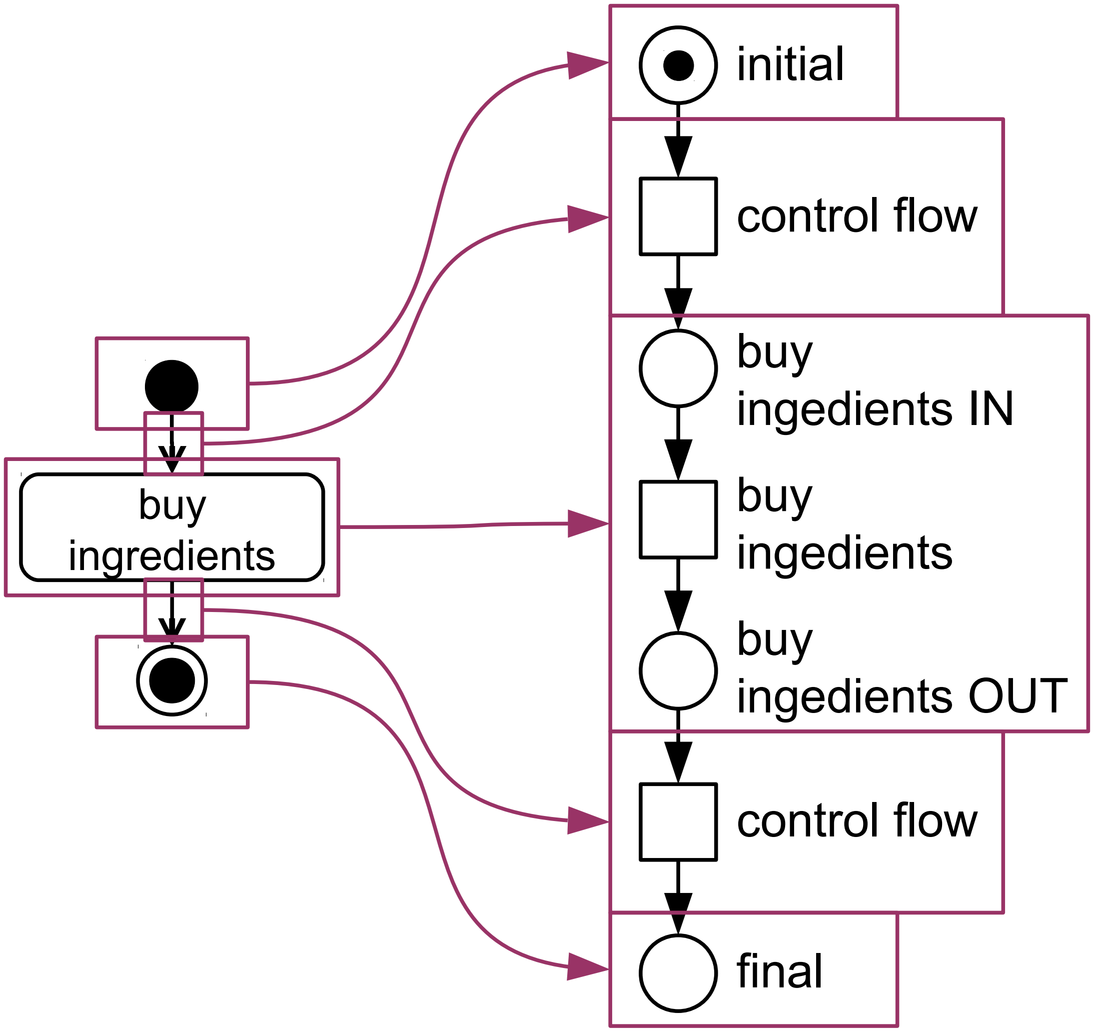


- Final node ↔ Empty Place



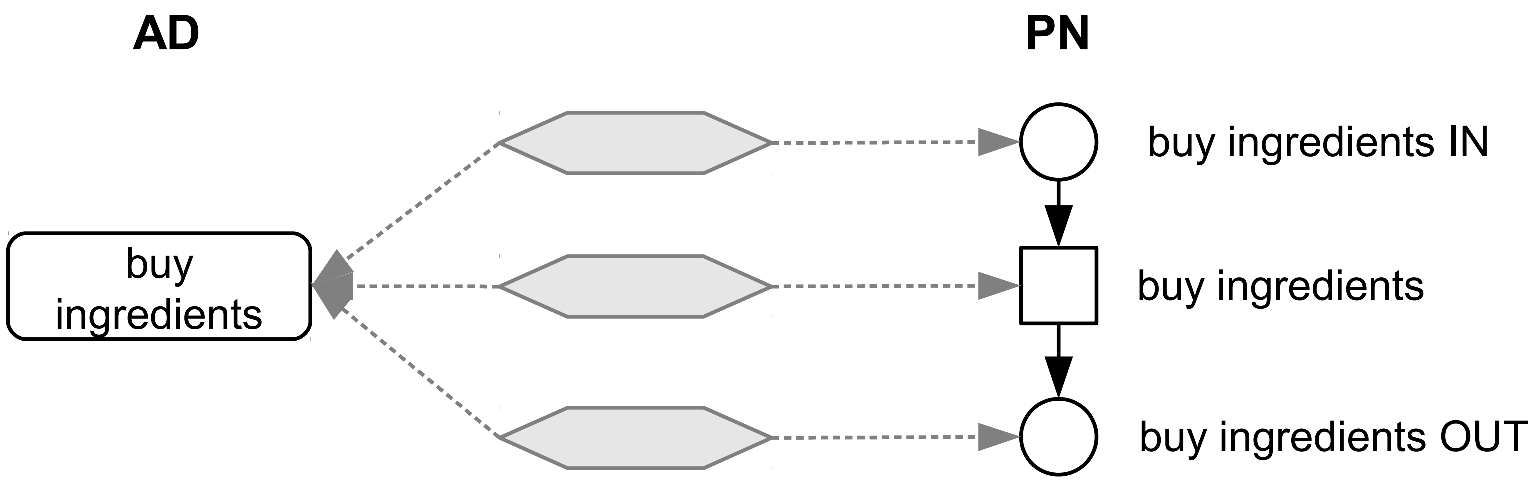
Example:Transform Activity Diagrams to Petri nets

- Let's start simple: How to transform
  - Initial nodes?
  - Final nodes?
  - Action nodes?
  - Control flow edges?



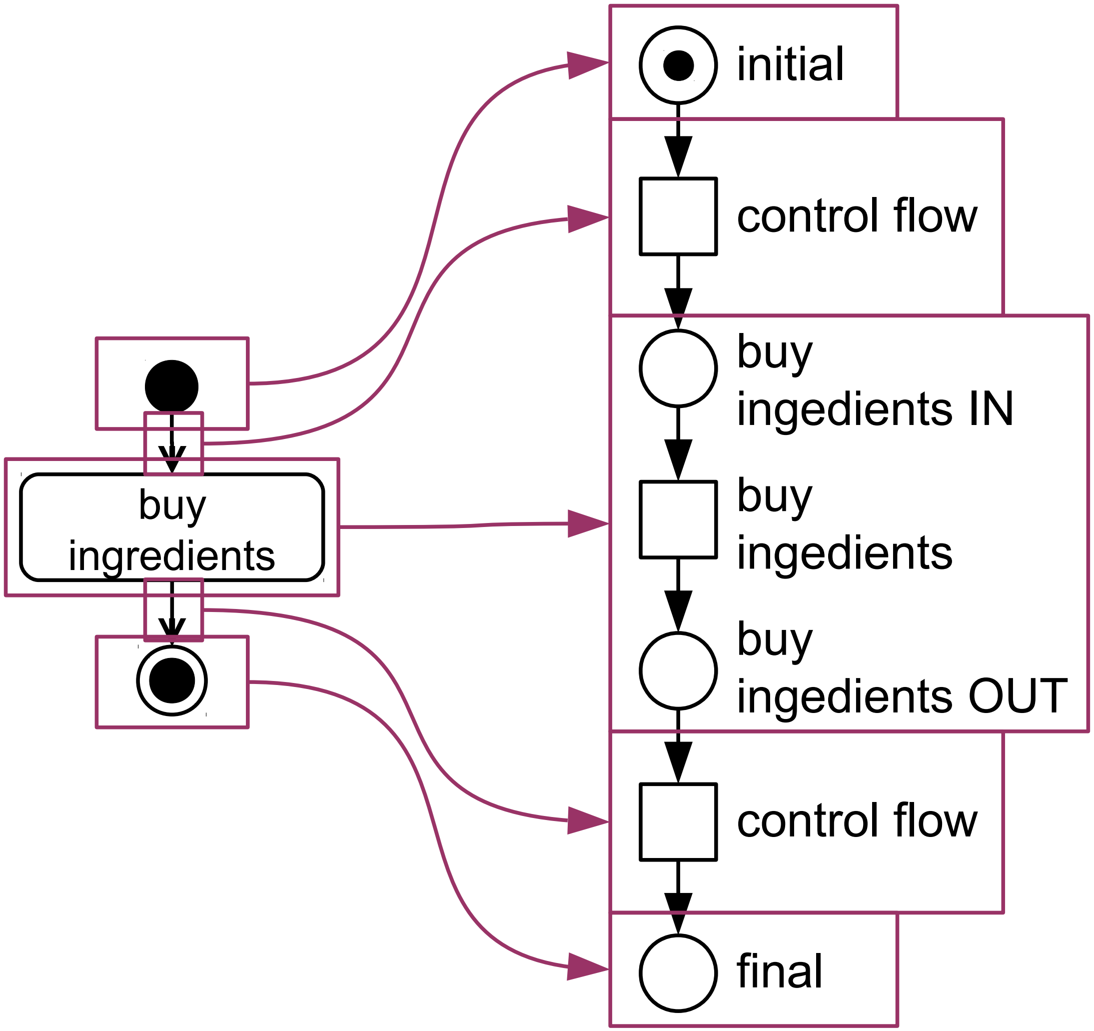
Relations Between Model Patterns Example: Activity to Petri net

- Action node ↔ Transition with input and output place



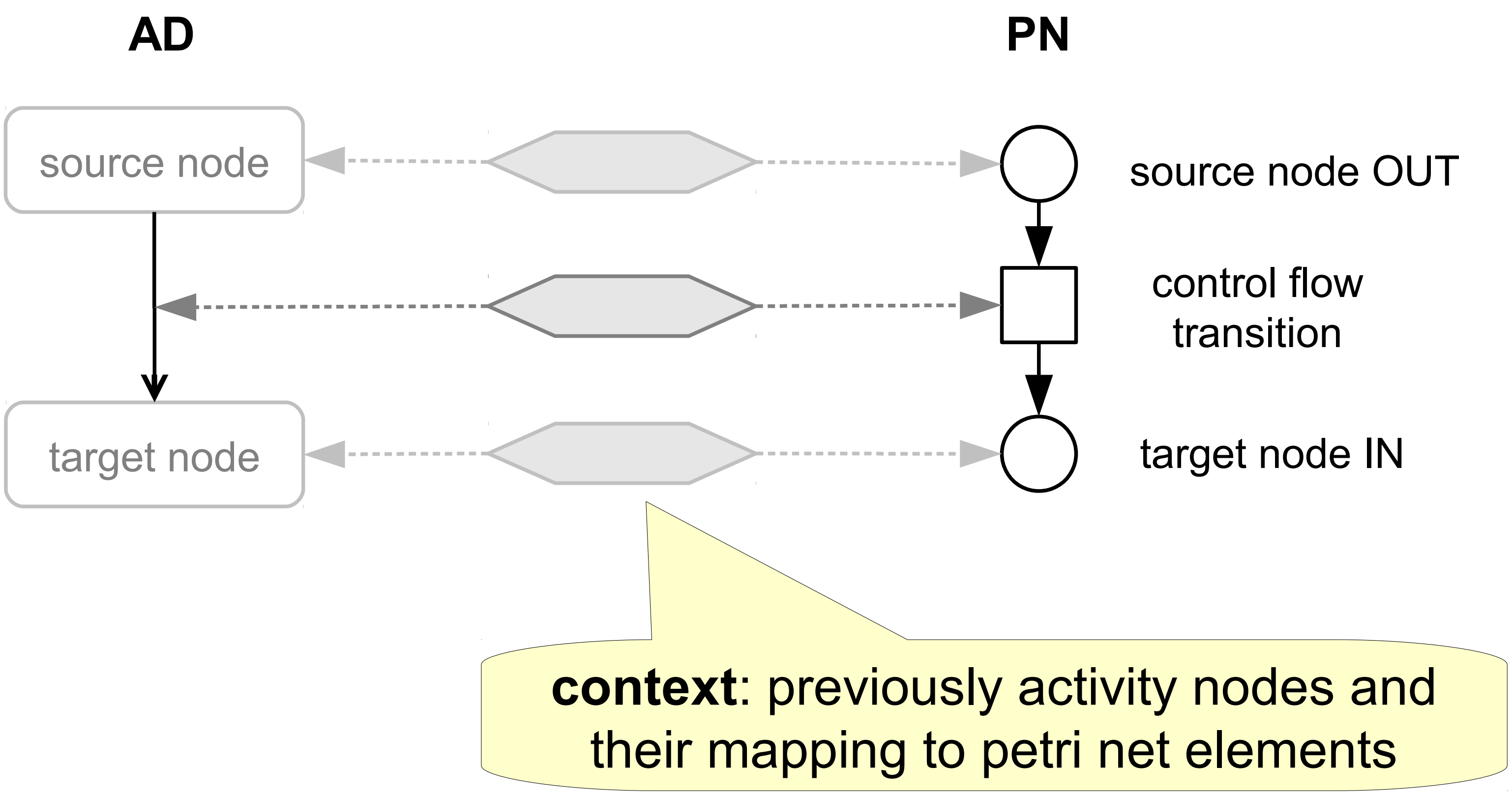
Example:Transform Activity Diagrams to Petri nets

- Let's start simple: How to transform
  - Initial nodes?
  - Final nodes?
  - Action nodes?
  - Control flow edges?



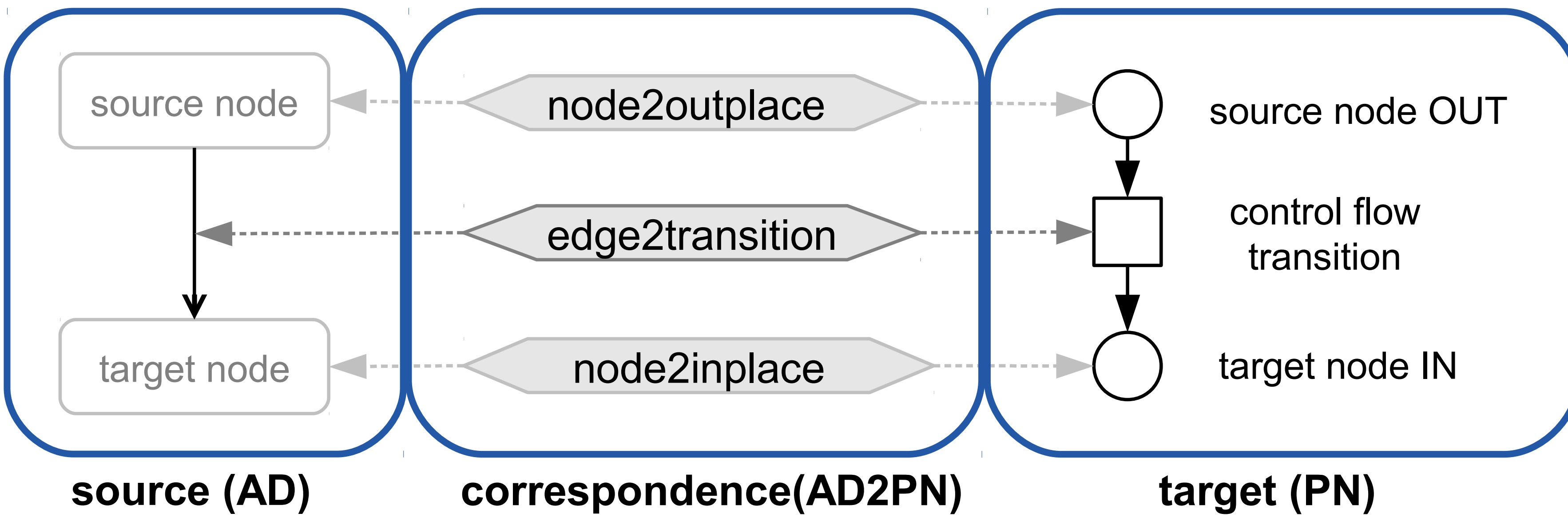
Relations Between Model Patterns Example: Activity to Petri net

- Control flow edge ↔ Transition connecting in- and out places that correspond to the edge's source and target nodes



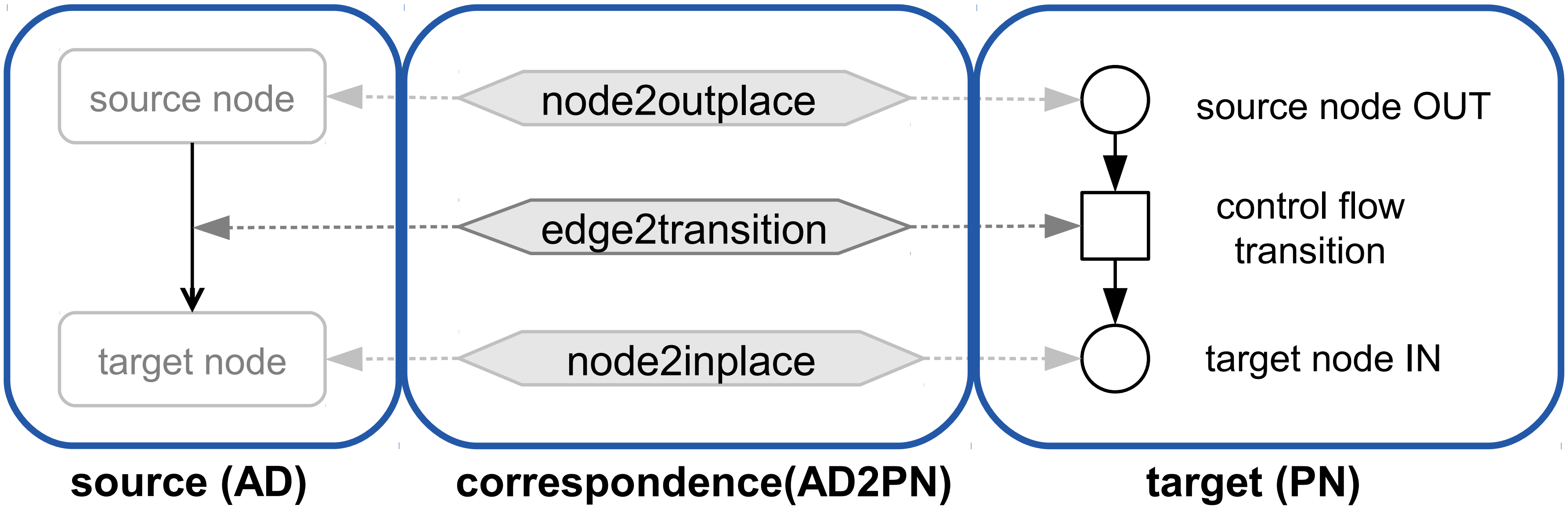
Triple Graphs

- **Idea 1**: describe the mapping of models as a **triple graph**
- What does a **triple graph** consist of?
  - **source** graph (model)
  - **target** graph (model)
  - **correspondence** graph (model) that connects the source and target graphs (models)



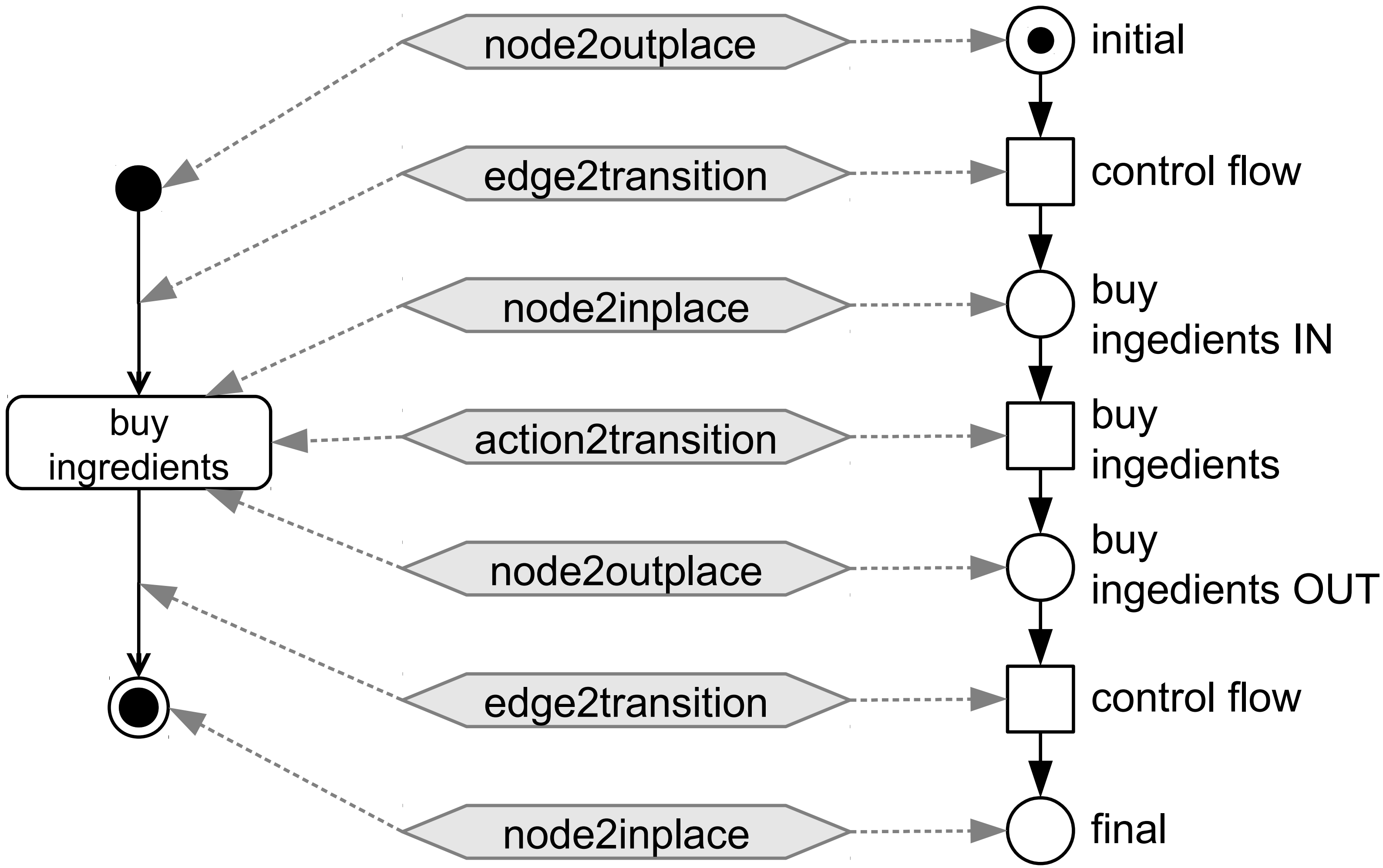
Triple Graphs

- The three different graphs (source, target, correspondence) are typed over (usually different) type graphs (metamodels)
- Also called source-, target-, and correspondence- **domain**
  - **source domain:** Activity Diagrams
  - **target domain:** Petri net
  - **correspondence domain:** AD2PN



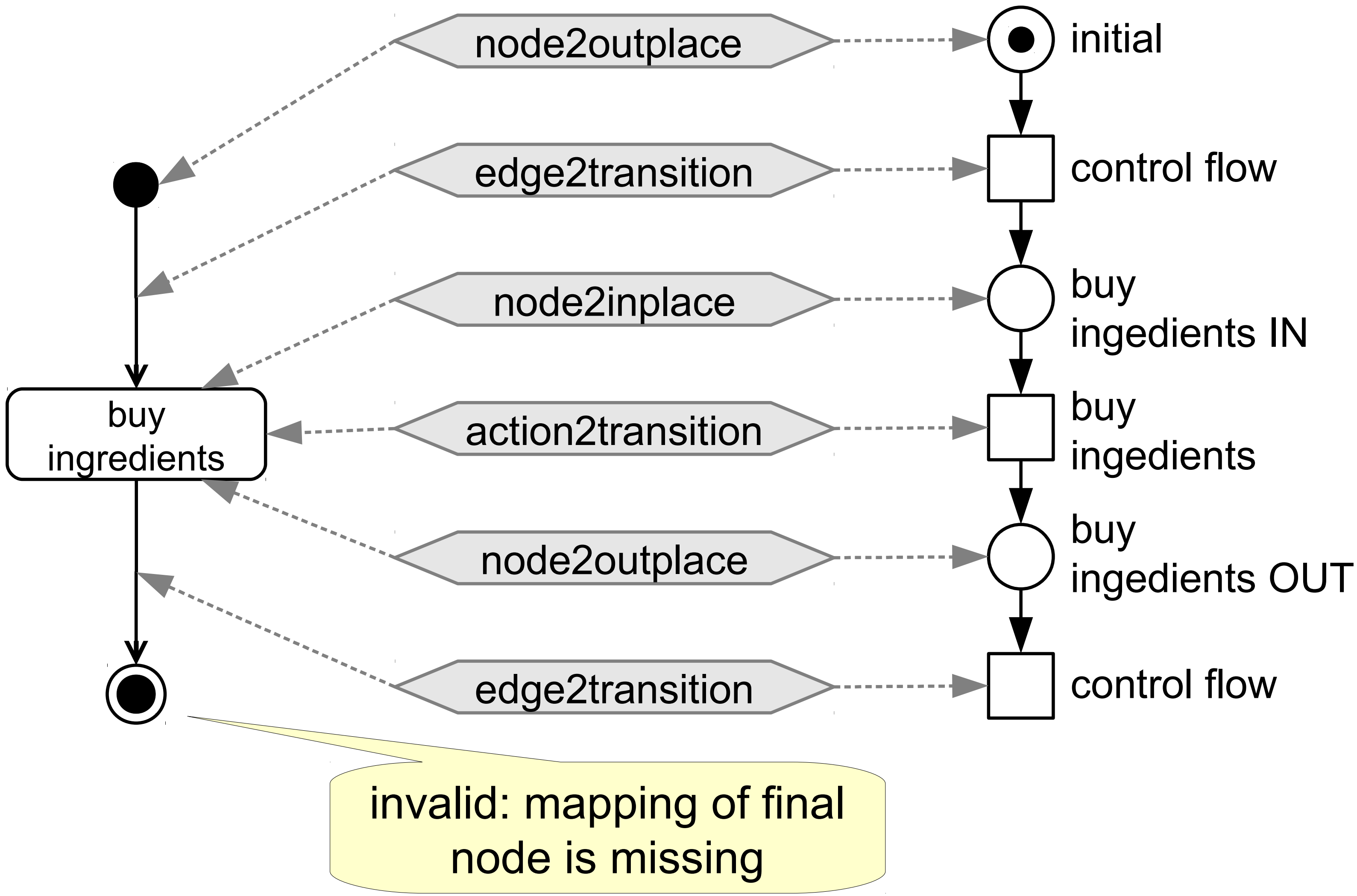
Triple Graphs

- Example of a bigger triple graph



Triple Graphs

- An “invalid” triple graph



Triple Graph Grammar (TGG)

- How to describe which triple graphs are valid in which ones are not?
  - i.e., express which mappings are valid and which ones are not

- **Idea 2**: Use a **graph grammar** that describes the production of valid triple graphs
  - → Triple Graph Grammar (TGG)