

# Model-Based Software Engineering

## Lecture 08 – Transformation

*Prof. Dr. Joel Greenyer*



June 7, 2016



in the last lecture...

## ***5.1. Introduction to semantics, transformations, execution, analysis***

# Ways to Define Semantics?

in the last lecture...

- How do we define the semantics of a formal language?

- By using natural language
- By giving a formal definition using mathematics
- By implementing a code generator
- By implementing an interpreter (“virtual machine”)
- By specifying a mapping to a semantic domain model
- By implementing a model transformation to a semantic domain model

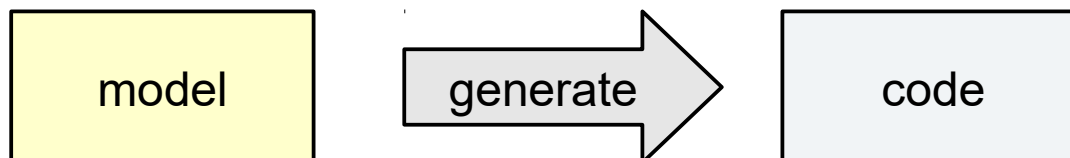
**Purpose:**

human readable

can be  
human  
readable

machine readable:  
*executable,  
automatically  
analyzable*

in the last lecture...

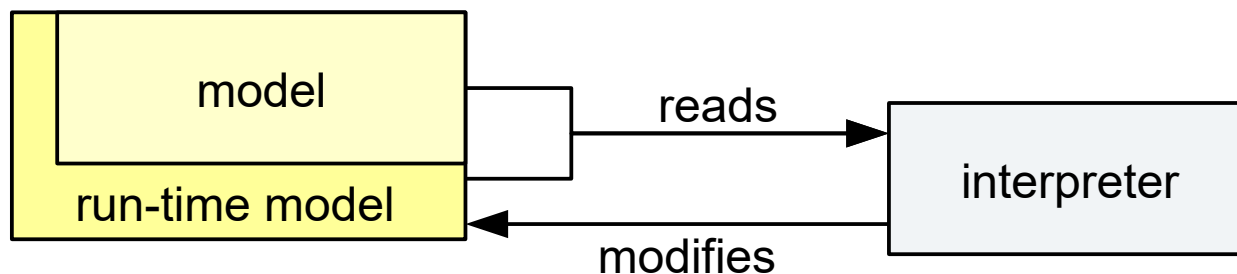


- for example: A state machine to Java generator defines the semantics of state machines **by a mapping to Java**
  - the semantics of Java is precisely specified in a specification  
<https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>
  - the semantics of Java is also precisely defined **through its mapping to Java byte code**,
    - which is again precisely specified in a specification, see  
<https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf>
    - or for which the semantics is defined **in the form of different virtual machine implementations**

# Programming an Interpreter ("Virtual Machine")

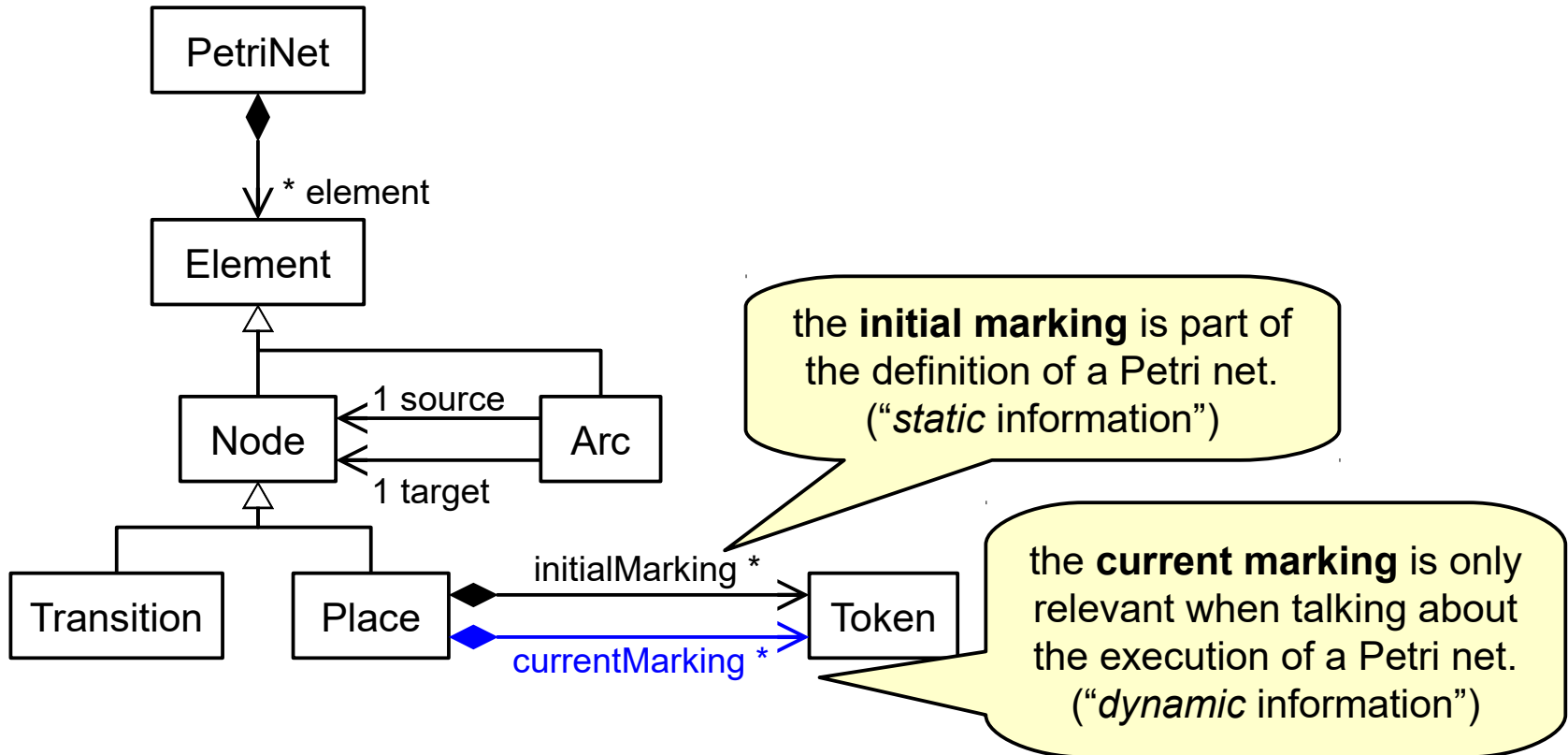
in the last lecture...

- for languages dealing with behavior, we can extend the metamodel by constructs that capture **run-time concepts**
  - for example: model “heap”, “stack”, “variable bindings”, etc. for a programming language
- The interpreter can read the model and its runtime extension
- The runtime extension part captures the “current state” of execution, which the interpreter can modify



in the last lecture...

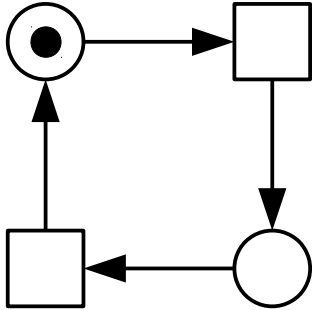
- Example: Petri net runtime extension



in the last lecture...

- Example: Petri net runtime extension

static:

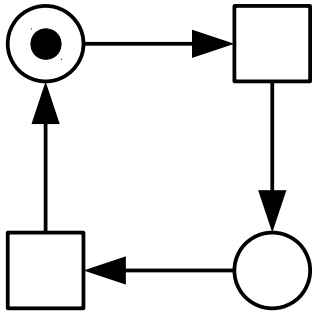


dynamic:

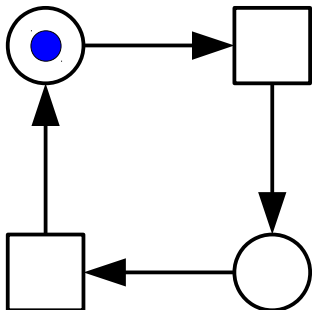
in the last lecture...

- Example: Petri net runtime extension

static:



dynamic:



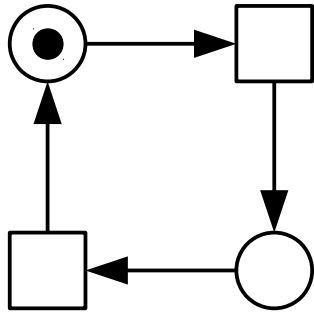


- Example: Petri net runtime extension

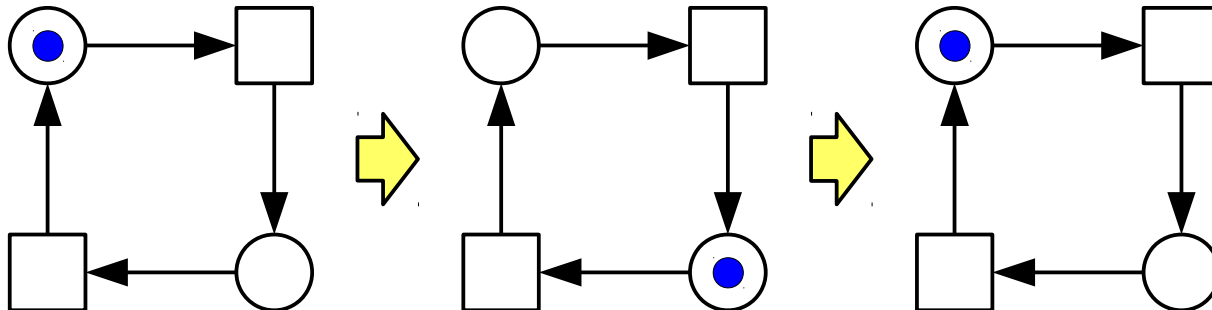
in the last lecture...

- Example: Petri net runtime extension

static:



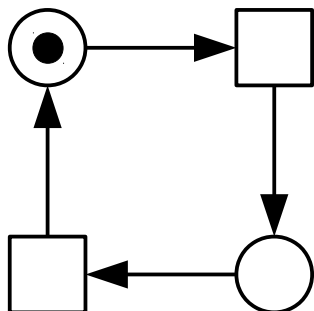
dynamic:



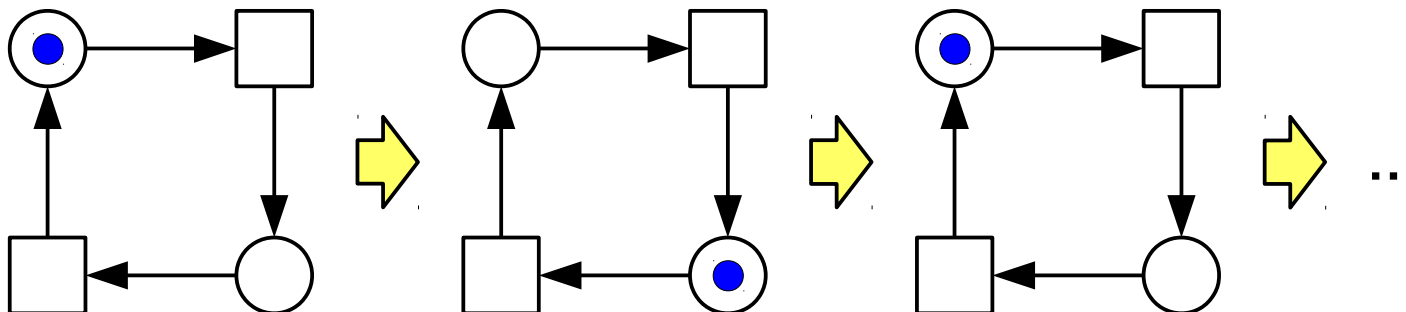
in the last lecture...

- Example: Petri net runtime extension

static:

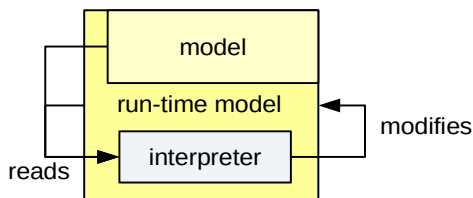


dynamic:

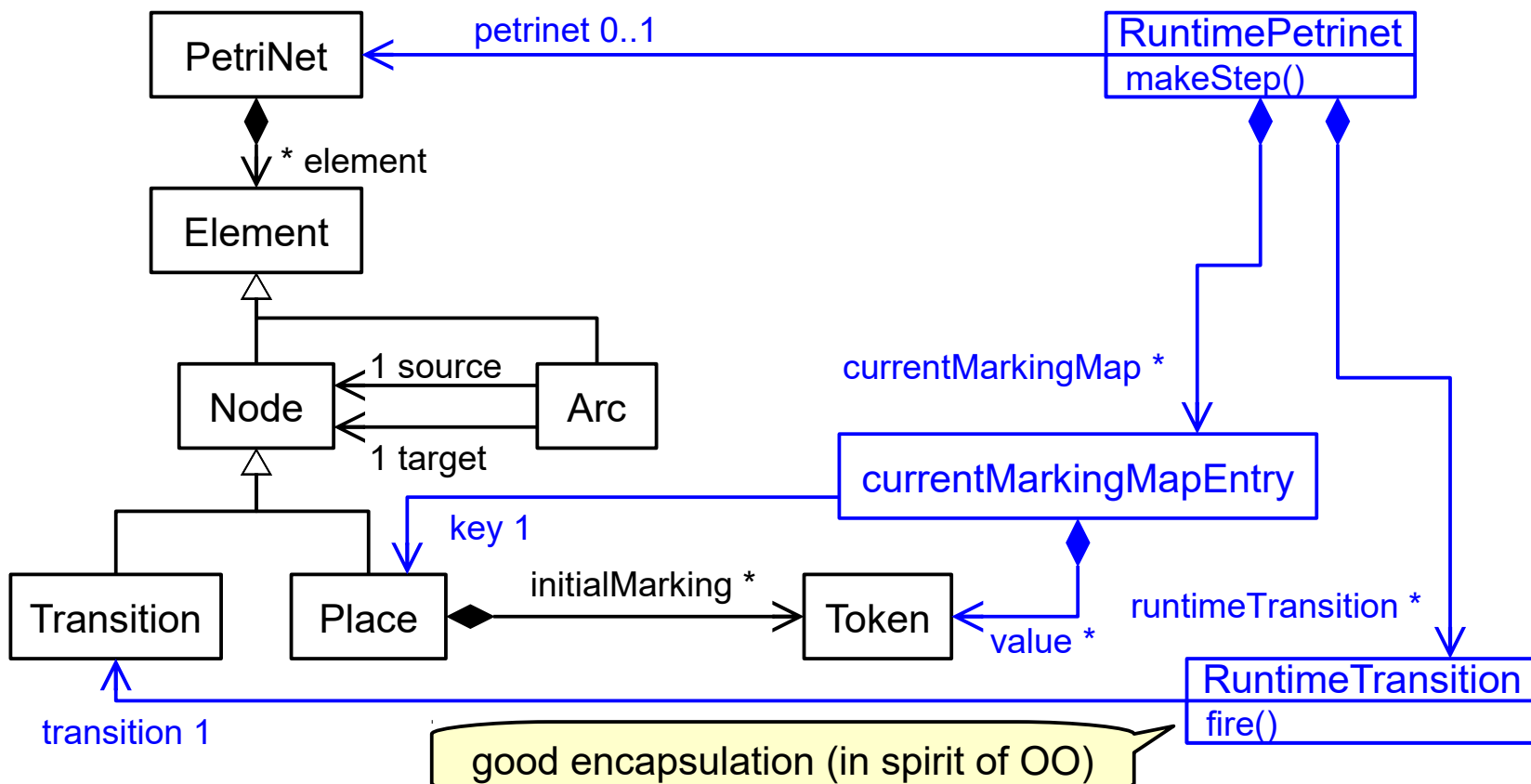


in the last lecture...

- Example: Petri net interpreter part of the runtime model



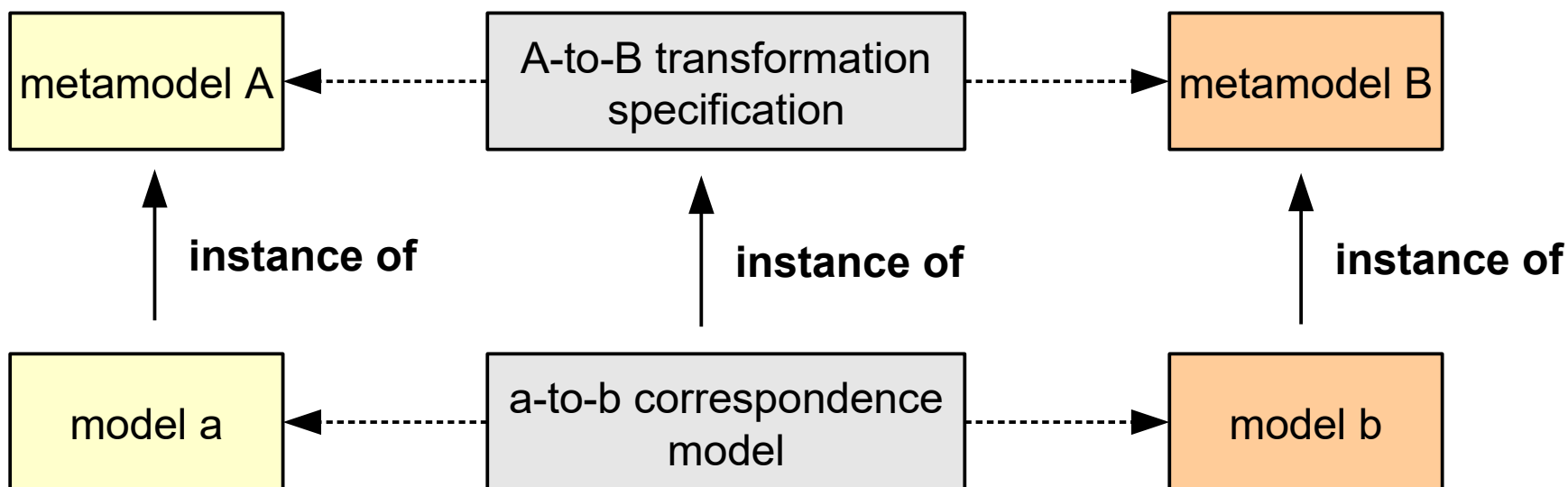
Better separation of concerns: the dynamic logic becomes part of the runtime model extension



# Model-to-Model Transformations

in the last lecture...

- A typical way to view model-to-model transformations
  - transformation from language A to language B
  - the transformation specification refers to metamodels A and B
  - sometimes: the transformation creates a *correspondence model* of how elements of model a and b relate specifically



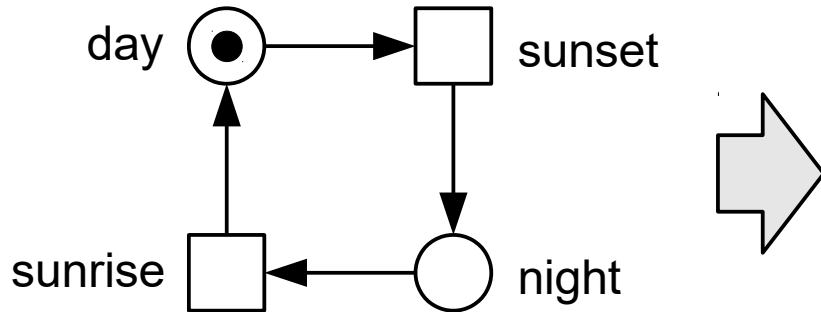
in the last lecture...

## ***5.2. Model-to-text transformation (code generation)***

# Example: Petrinet to Java

in the last lecture...

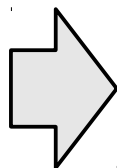
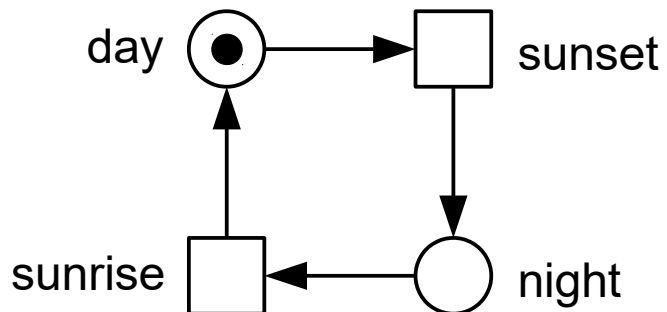
- Example:



# Example: Petrinet to Java

in the last lecture...

- Example:



```

public class DayAndNight {
    // places
    int day=1; int night=0;
    // main makeStep method
    public void makeStep(){
        if (canFireSunset()){
            doFireSunset()
        } else
        if (canFireSunrise()){
            doFireSunrise()
        } else
        { System.out.println("Cannot fire");}
    }
    // transition's canFire and doFire methods
    protected boolean canFireSunset(){
        return (day > 0);
    }
    protected void doFireSunset(){
        day--; night++;
    }
    protected boolean canFireSunrise(){
        return (night > 0);
    }
    protected void doFireSunrise(){
        night--; day++;
    }
}
  
```



# Xtext and Xtend

## in the last lecture...

- We can implement our custom code generator for example as follows:

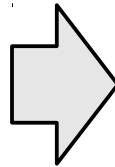
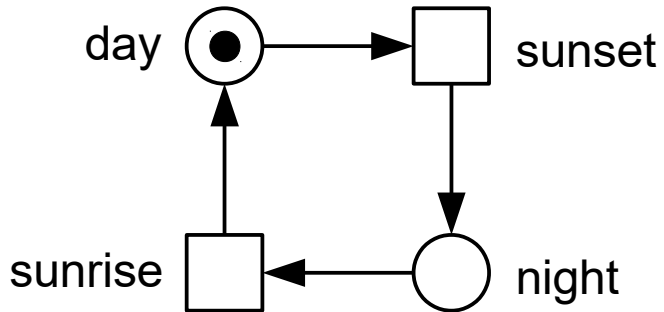
```

22  override void doGenerate(Resource resource,
23      IFileSystemAccess2 fsa,
24      IGeneratorContext context
25  ) {
26      for (pn : resource.allContents.toIterable().filter(Petrinet)) {
27          fsa.generateFile(
28              "petrinets/" + pn.name + ".java",
29              pn.compile
30          )
31      }
32  }
33
34  def compile(Petrinet pn) {
35      ...
36      package petrinets;
37
38      public class «pn.name» {
39          // places
40          «FOR p : pn.element.filter(Place)»
41              «p.compile»
42          «ENDFOR»
43          // main makeStep method
44          public void makeStep() {
45              «FOR t : pn.element.filter(Transition)»
46                  «t.compileForMakeStep»
47              «ENDFOR»
48              { System.out.println("Cannot fire"); }
49          }
50          // transition's canFire and doFire methods
51          «FOR t : pn.element.filter(Transition)»
52              «t.compile»
53          «ENDFOR»
54      }
55      ...
56  }
57
58  }

```

# Xtext and Xtend

- For execution, we also need a main method...



```

public class DayAndNight {

    // main method
    public static void main(String[] args) {
        DayAndNight dayAndNight
            = new DayAndNight();
        // make 100 steps
        for (int i = 0; i < 100; i++) {
            dayAndNight.makeStep();
        }

        public void makeStep(){
            ...
        }

        ...
    }
  
```

# Xtext and Xtend

```
def compile(Petrinet pn) {
    ...
    package petrinets;

    public class «pn.name.toFirstUpper» {
        // main method
        public static void main(String[] args) {
            «pn.name.toFirstUpper» «pn.name.toFirstLower»
                = new «pn.name.toFirstUpper»();
            // make 100 steps
            for (int i = 0; i < 100; i++) {
                «pn.name.toFirstLower».makeStep();
            }
        }

        // places
        «FOR p : pn.element.filter(Place)»
            «p.compile»
        «ENDFOR»

        // main makeStep method
        public void makeStep() {
            «FOR t : pn.element.filter(Transition)»
                «t.compileForMakeStep»
            «ENDFOR»
            { System.out.println("Cannot fire"); }
        }
    }
}
```

# Xtext and Xtend

creates main method  
that calls makeStep()  
100 times.

```
def compile(Petrinet pn) {
    ...
    package petrinets;

    public class «pn.name.toFirstUpper» {

        // main method
        public static void main(String[] args) {
            «pn.name.toFirstUpper» «pn.name.toFirstLower»
                = new «pn.name.toFirstUpper»();
            // make 100 steps
            for (int i = 0; i < 100; i++) {
                «pn.name.toFirstLower».makeStep();
            }
        }

        // places
        «FOR p : pn.element.filter(Place)»
            «p.compile»
        «ENDFOR»

        // main makeStep method
        public void makeStep() {
            «FOR t : pn.element.filter(Transition)»
                «t.compileForMakeStep»
            «ENDFOR»
            { System.out.println("Cannot fire"); }
        }
    }
}
```

# Xtend Template Expressions

---

- Everything enclosed in three single quotes ( `' ' ' ... ' ' '` ) is a **template expression** in Xtend

# Xtend Template Expressions

---

- Everything enclosed in three single quotes ( `' ' ' ... ' ' '` ) is a **template expression** in Xtend
- Use guillemets ( `«, »` “french quotes”) to insert **interpolated expression**

# Xtend Template Expressions

---

- Everything enclosed in three single quotes ( `' ' ' ... ' ' '` ) is a **template expression** in Xtend
- Use guillemets ( `«, »` “french quotes”) to insert **interpolated expression**
  - other Xtext expressions that evaluate to a String

# Xtend Template Expressions

---

- Everything enclosed in three single quotes ( `' ' ' ... ' ' '` ) is a **template expression** in Xtend
- Use guillemets ( `«, »` “french quotes”) to insert **interpolated expression**
  - other Xtext expressions that evaluate to a String
  - their result will be inserted into the template string



# Xtend Template Expressions

- Everything enclosed in three single quotes ( `' ' ' ... ' ' '` ) is a **template expression** in Xtend
- Use guillemets ( `«`, `»` “french quotes”) to insert **interpolated expression**
  - other Xtext expressions that evaluate to a String
  - their result will be inserted into the template string

```
def compile(Petrinet pn){  
    ...  
    package petrinets;  
  
    public class «pn.name.toFirstUpper» {  
        ...  
    }
```

# Xtend Template Expressions

---

- In template expressions, there are special loop constructs
  - using FOR / ENDFOR

# Xtend Template Expressions

- In template expressions, there are special loop constructs
  - using FOR / ENDFOR

template expression  
loop

```
// places
«FOR p : pn.element.filter(Place)»
    int «p.name» = «p.initialMarking»;
«ENDFOR»
```

normal Xtend loop  
(Java-like)

```
for (pn : resource.allContents.toIterable.filter(Petrinet)) {
    fsa.generateFile(
        "petrinets/" + pn.name + ".java",
        pn.compile
    )
}
```

# Xtend Template Expressions

---

- In template expressions, there are special loop constructs
  - using FOR / ENDFOR

# Xtend Template Expressions

---

- In template expressions, there are special loop constructs
  - using FOR / ENDFOR
- The loops also support BEFORE, AFTER, and SEPARATOR expressions
  - for example

# Xtend Template Expressions

- In template expressions, there are special loop constructs
  - using FOR / ENDFOR
- The loops also support BEFORE, AFTER, and SEPARATOR expressions
  - for example

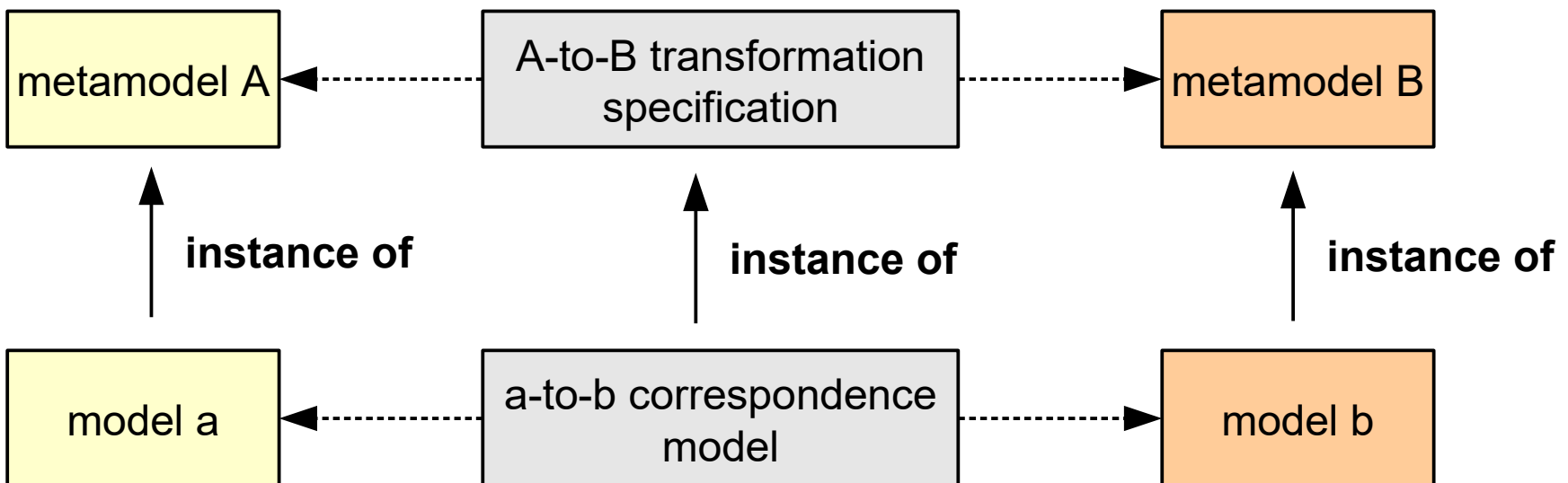
```
«FOR p : paragraphs
  BEFORE '<div>'
  SEPARATOR '</div><div>'
  AFTER '</div>'»
<h1>«p.headline»</h1>
<p>
  «p.text»
</p>
«ENDFOR»
```

### ***5.3. Model-to-model transformation – foundations and classification***

# Model-to-Model Transformations

in the last lecture...

- A typical way to view model-to-model transformations
  - transformation from language A to language B
  - the transformation specification refers to metamodels A and B
  - sometimes: the transformation creates a *correspondence model* of how elements of model a and b relate specifically





# Model-to-Model Transformations

---

- Example Model-to-Model transformation:

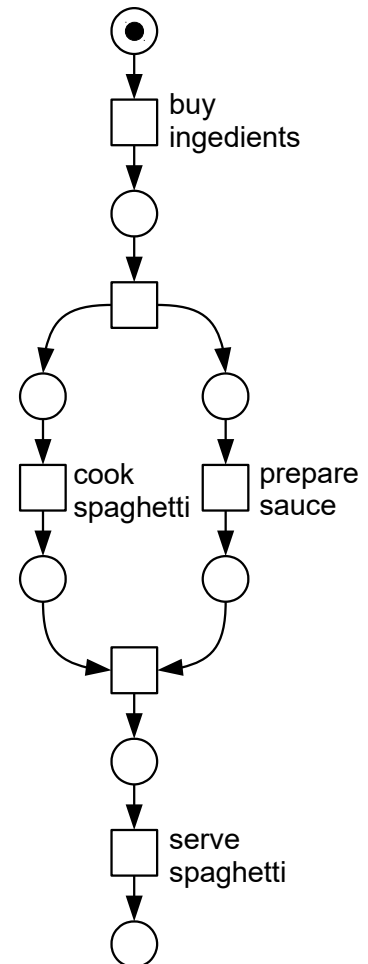
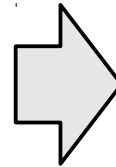
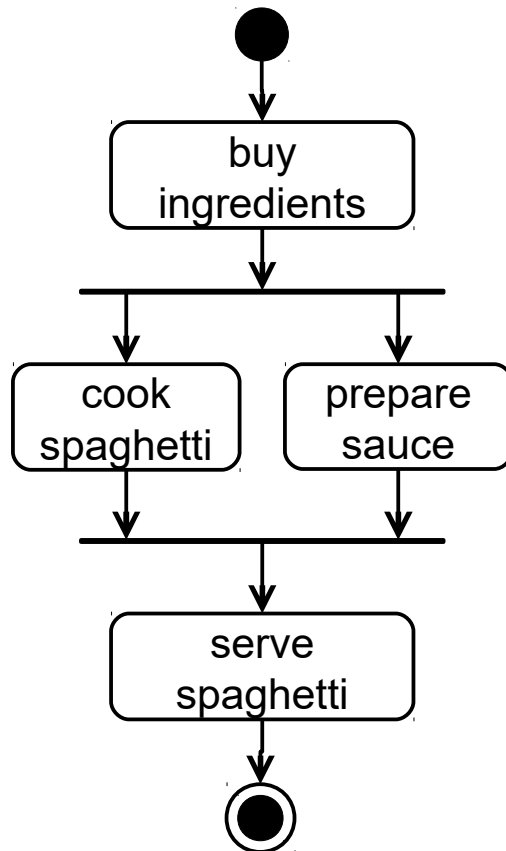
# Model-to-Model Transformations

---

- Example Model-to-Model transformation:
  - Transform UML Activity Diagrams to Petri nets

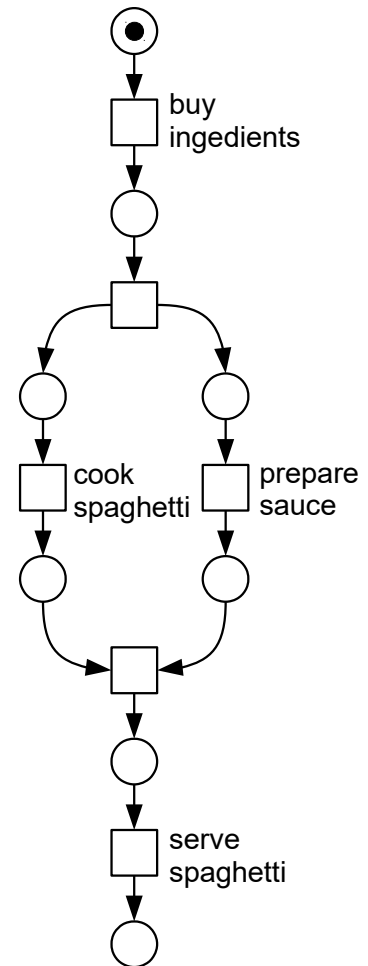
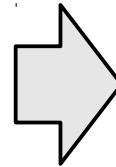
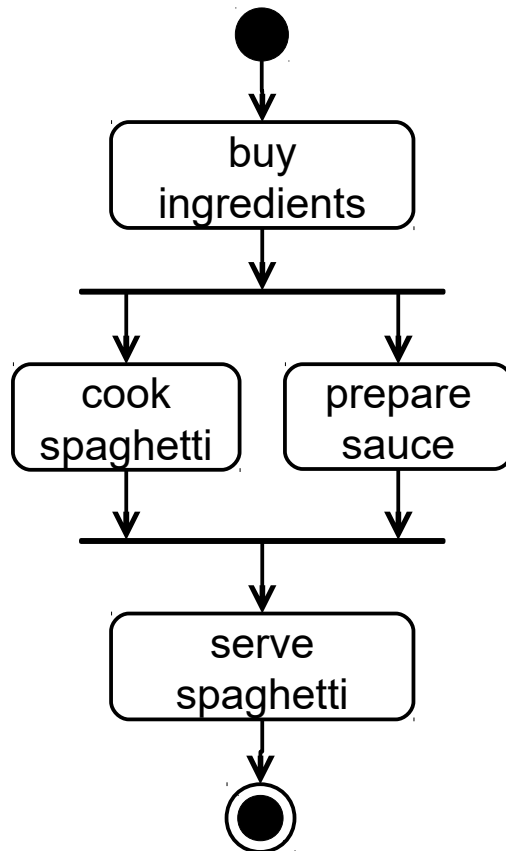
# Model-to-Model Transformations

- Example Model-to-Model transformation:
  - Transform UML Activity Diagrams to Petri nets



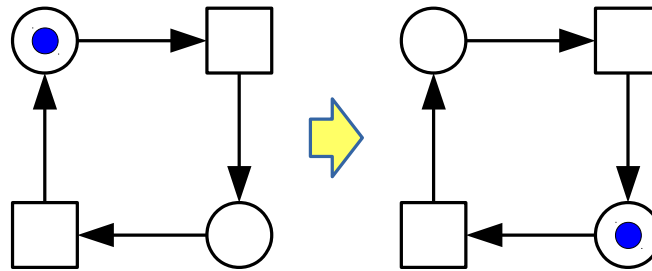
# Model-to-Model Transformations

- Example Model-to-Model transformation:
  - Transform UML Activity Diagrams to Petri nets
  - To support formal analysis and execution



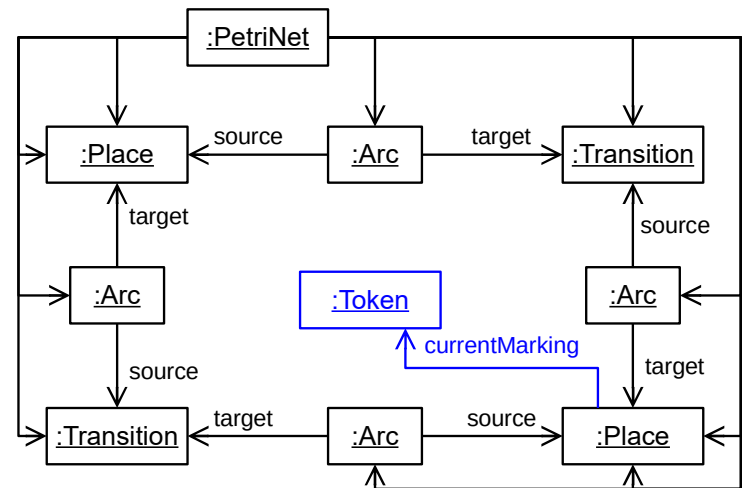
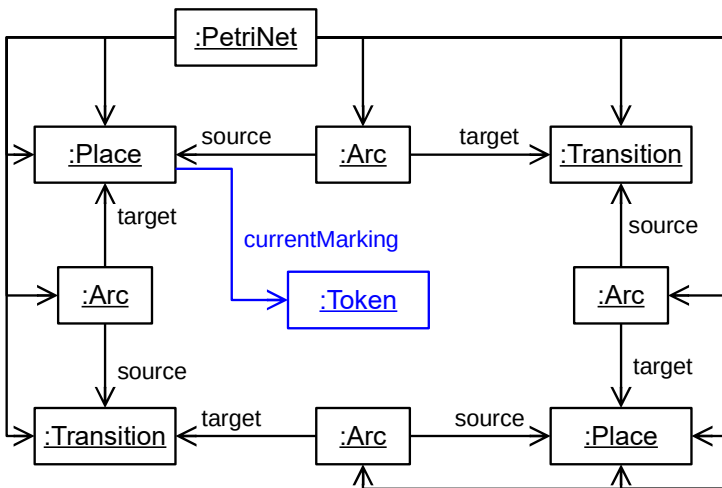
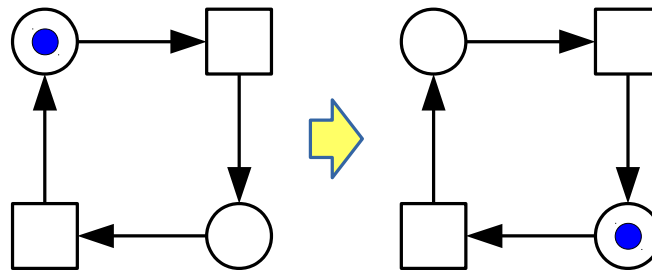
# Model-to-Model Transformations

- Example Model-to-Model transformation:
  - Transform one Petri net into another



# Model-to-Model Transformations

- Example Model-to-Model transformation:
  - Transform one Petri net into another



# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)

# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels



# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels
  - source model is retained

# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels
  - source model is retained
  - example: UML activity diagram to Petri net

# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels
  - source model is retained
  - example: UML activity diagram to Petri net
- Model transformations can also modify a source model (**Endogenous**, “*In-Place*”)

# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels
  - source model is retained
  - example: UML activity diagram to Petri net
- Model transformations can also modify a source model (**Endogenous**, “*In-Place*”)
  - target model is an updated version of the source model

# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels
  - source model is retained
  - example: UML activity diagram to Petri net
- Model transformations can also modify a source model (**Endogenous**, “*In-Place*”)
  - target model is an updated version of the source model
  - original version of the source model is discarded

# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels
  - source model is retained
  - example: UML activity diagram to Petri net
- Model transformations can also modify a source model (**Endogenous**, “*In-Place*”)
  - target model is an updated version of the source model
  - original version of the source model is discarded
  - source and target model has the same metamodel

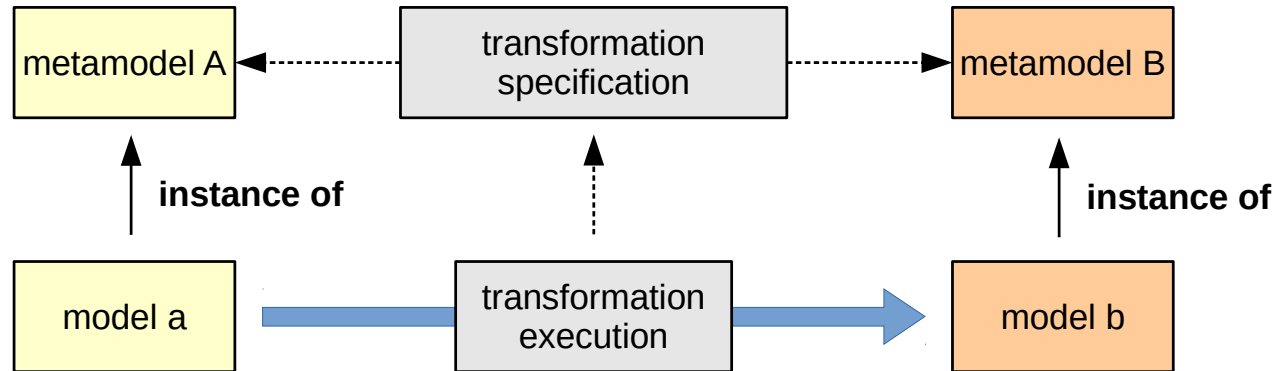
# Exogenous vs. Endogenous Model Transformations

---

- Model transformations can create target models from source models (**Exogenous**, “*Out-place*”)
  - possibly different source and target metamodels
  - source model is retained
  - example: UML activity diagram to Petri net
- Model transformations can also modify a source model (**Endogenous**, “*In-Place*”)
  - target model is an updated version of the source model
  - original version of the source model is discarded
  - source and target model has the same metamodel
  - example: Petri net “move token” transformation

# Exogenous vs. Endogenous Model Transformations

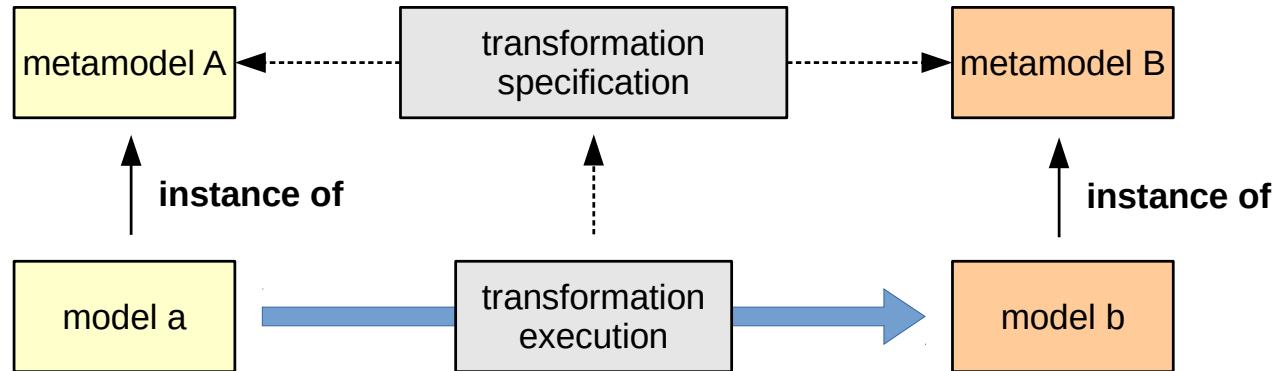
- Exogenous:**





# Exogenous vs. Endogenous Model Transformations

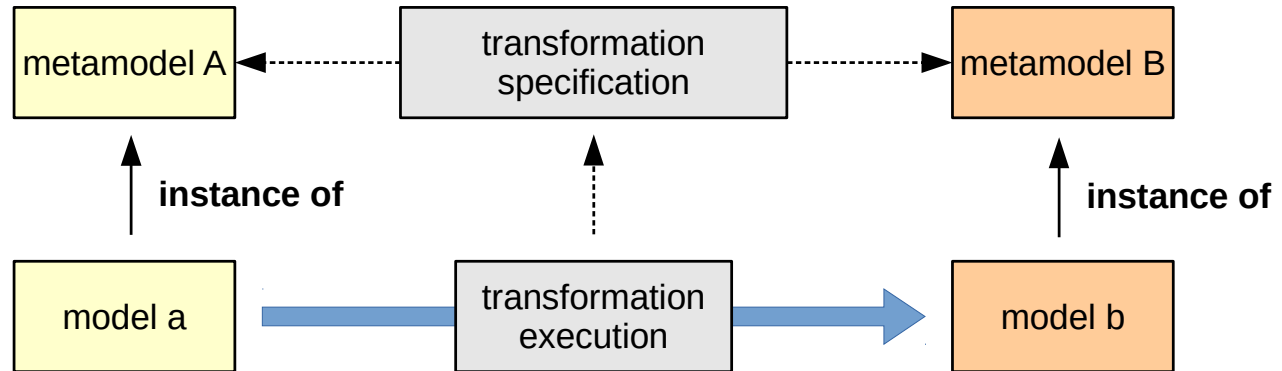
- Exogenous:**



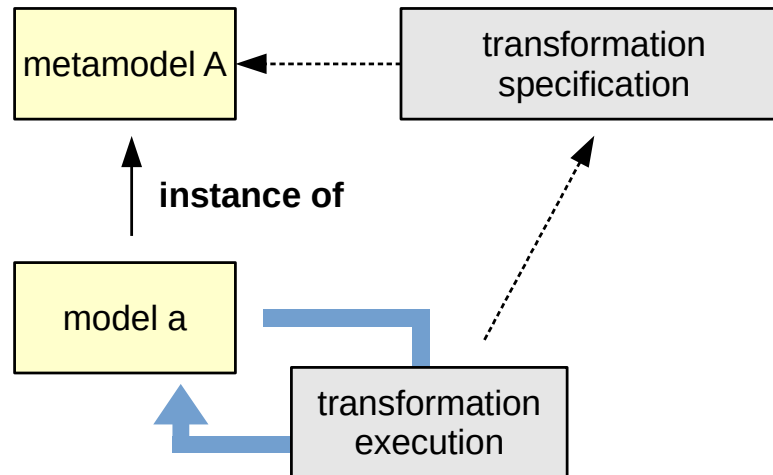
- Endogenous:**

# Exogenous vs. Endogenous Model Transformations

- Exogenous:**



- Endogenous:**



# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (*exogenous*)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (*exogenous*)
  - Execution

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (*exogenous*)
  - Execution (*endogenous*)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (*exogenous*)
  - Execution (*endogenous*)
  - Define the semantics to the modeling language



# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model
    - for different purposes and for different stakeholders

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model (**ex**)
    - for different purposes and for different stakeholders



# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model (**ex**)
    - for different purposes and for different stakeholders
  - Refactoring

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model (**ex**)
    - for different purposes and for different stakeholders
  - Refactoring (**end**)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model (**ex**)
    - for different purposes and for different stakeholders
  - Refactoring (**end**)
  - Model evolution (metamodel changed, instances need to be changed accordingly)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model (**ex**)
    - for different purposes and for different stakeholders
  - Refactoring (**end**)
  - Model evolution (metamodel changed, instances need to be changed accordingly) (**ex**)

# Why Model-to-Model Transformations?

---

- We may need model transformations for various purposes
  - As a step towards code generation (**exogenous**)
  - Execution (**endogenous**)
  - Define the semantics to the modeling language (**end/ex**)
  - Support formal analysis (**end/ex**)
  - Generate Documentation (**ex**)
  - Creating different view models from a base model (**ex**)
    - for different purposes and for different stakeholders
  - Refactoring (**end**)
  - Model evolution (metamodel changed, instances need to be changed accordingly) (**ex**)

# One-to-Many, Many-to-One, Many-to-Many Model Transformations

---

- Transformations can also

# One-to-Many, Many-to-One, Many-to-Many Model Transformations

---

- Transformations can also
  - create multiple target models from one source model (**one-to-many**)

# One-to-Many, Many-to-One, Many-to-Many Model Transformations

---

- Transformations can also
  - create multiple target models from one source model (**one-to-many**)
  - create one target model from multiple source models (**many-to-one**)



# One-to-Many, Many-to-One, Many-to-Many Model Transformations

---

- Transformations can also
  - create multiple target models from one source model (**one-to-many**)
  - create one target model from multiple source models (**many-to-one**)
  - or create many target models from many source models (**many-to-many**)

# One-to-Many, Many-to-One, Many-to-Many Model Transformations

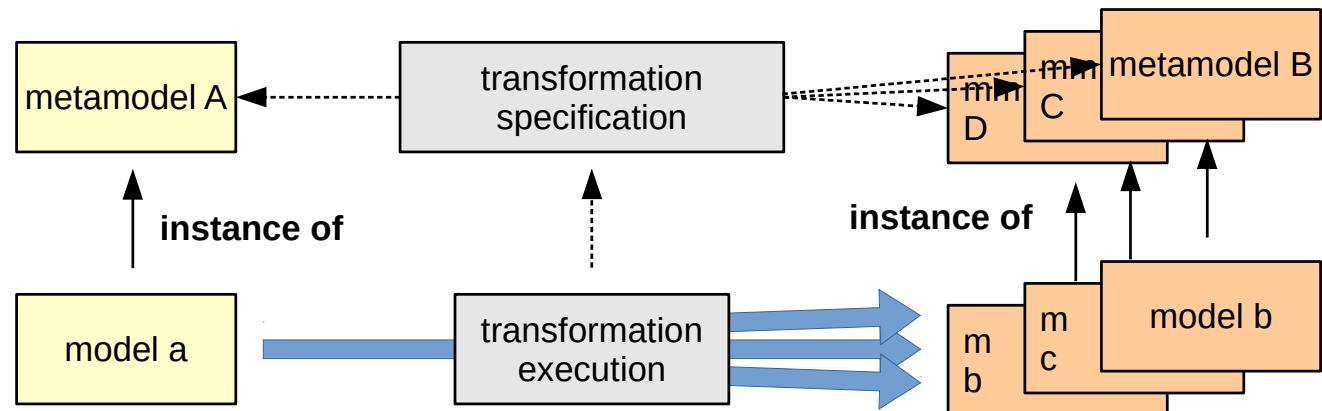
---

- Transformations can also
  - create multiple target models from one source model (**one-to-many**)
  - create one target model from multiple source models (**many-to-one**)
  - or create many target models from many source models (**many-to-many**)
  - Multiple source/target models may have different metamodels

# One-to-Many, Many-to-One, Many-to-Many Model Transformations

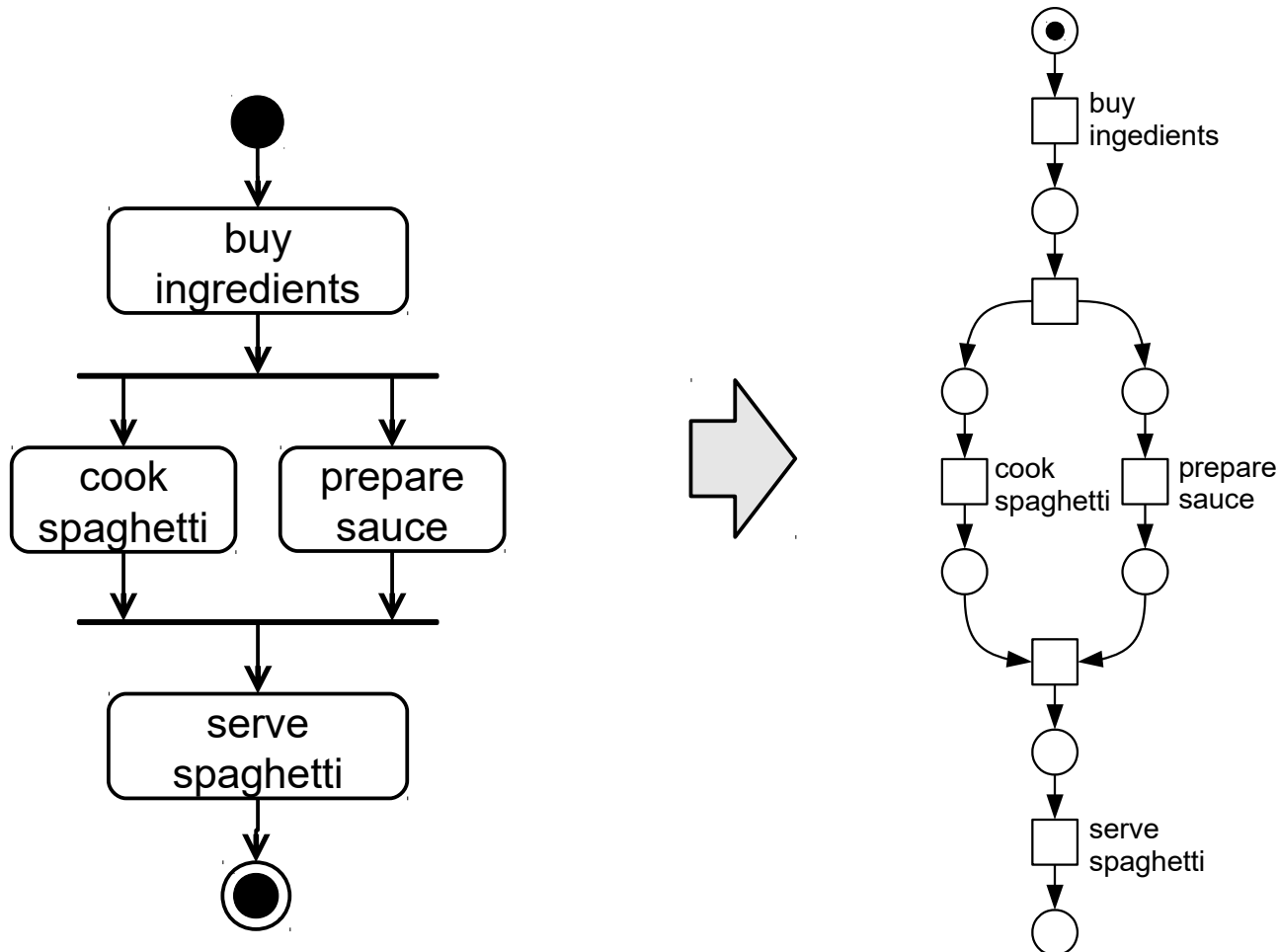
- Transformations can also
  - create multiple target models from one source model (**one-to-many**)
  - create one target model from multiple source models (**many-to-one**)
  - or create many target models from many source models (**many-to-many**)
  - Multiple source/target models may have different metamodels

- illustration:  
one-to-many



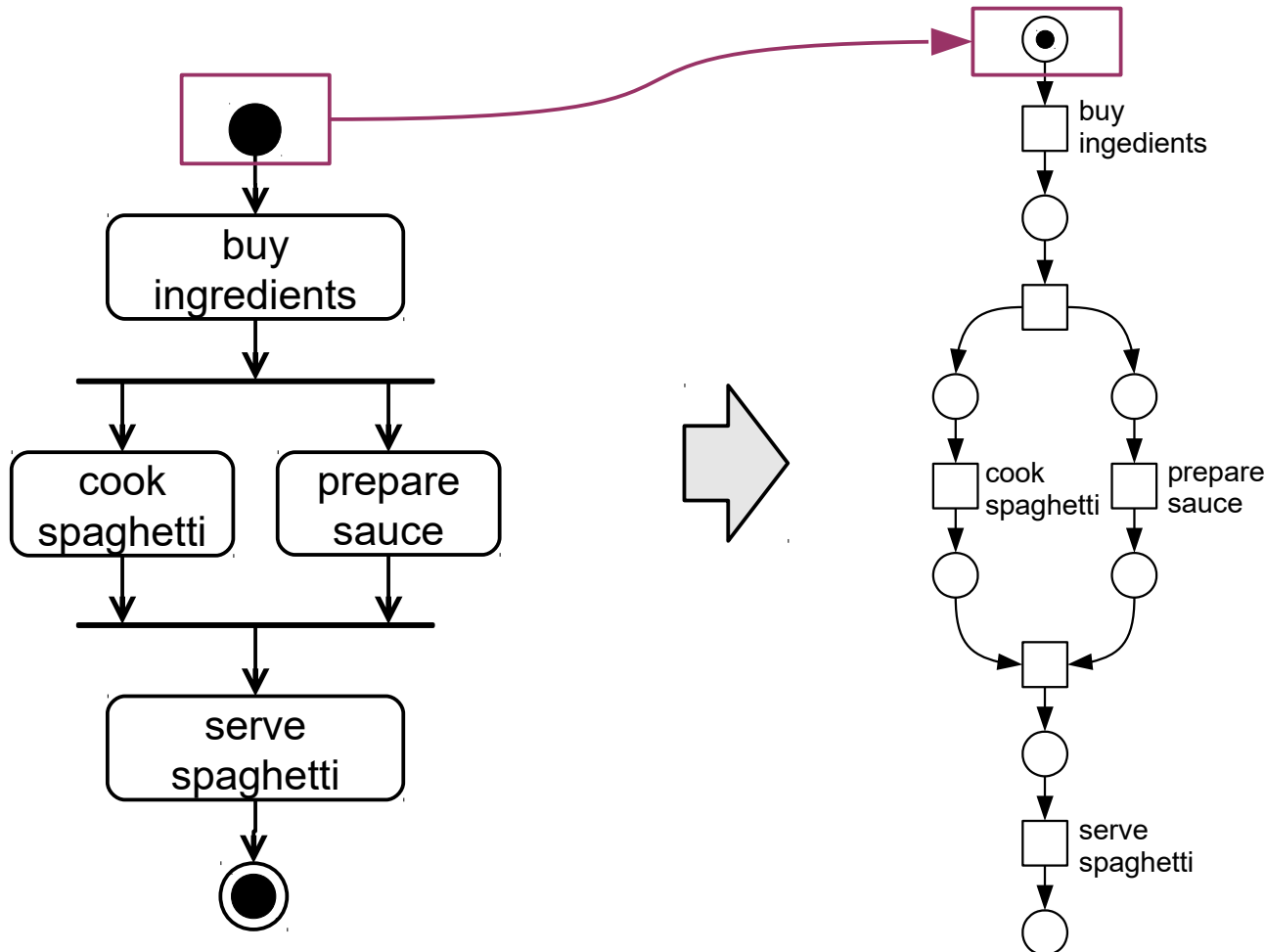
# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



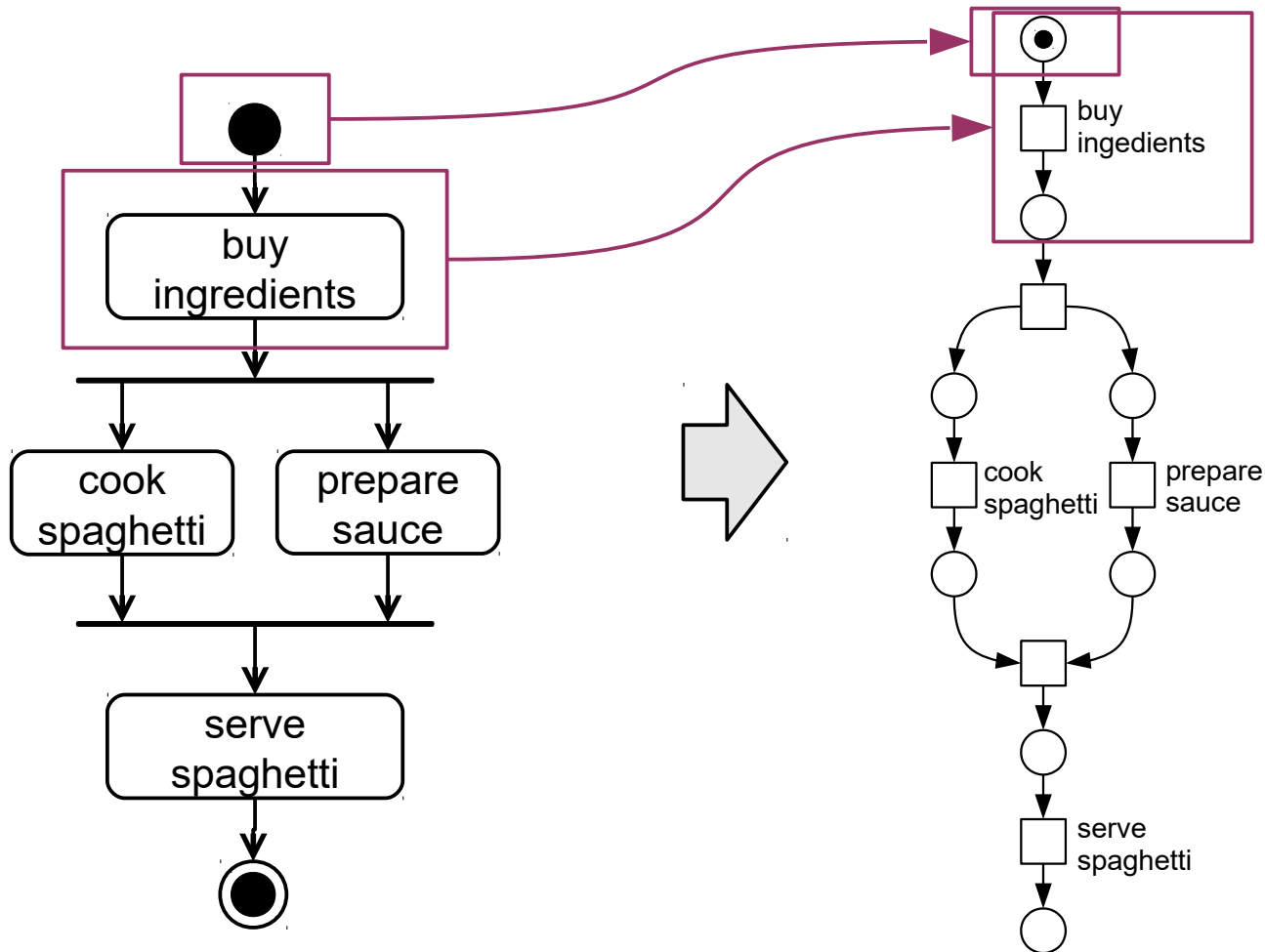
# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



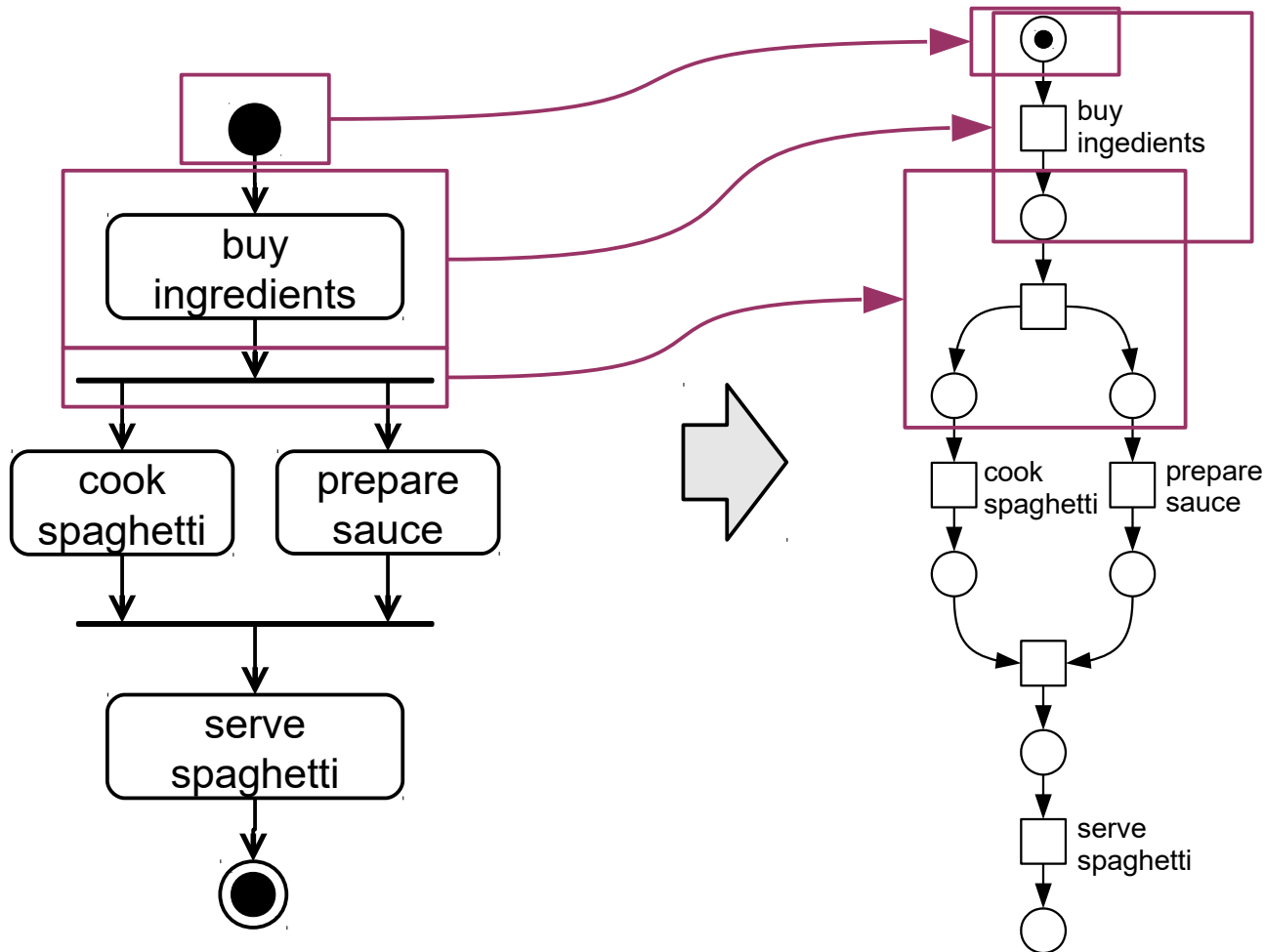
# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



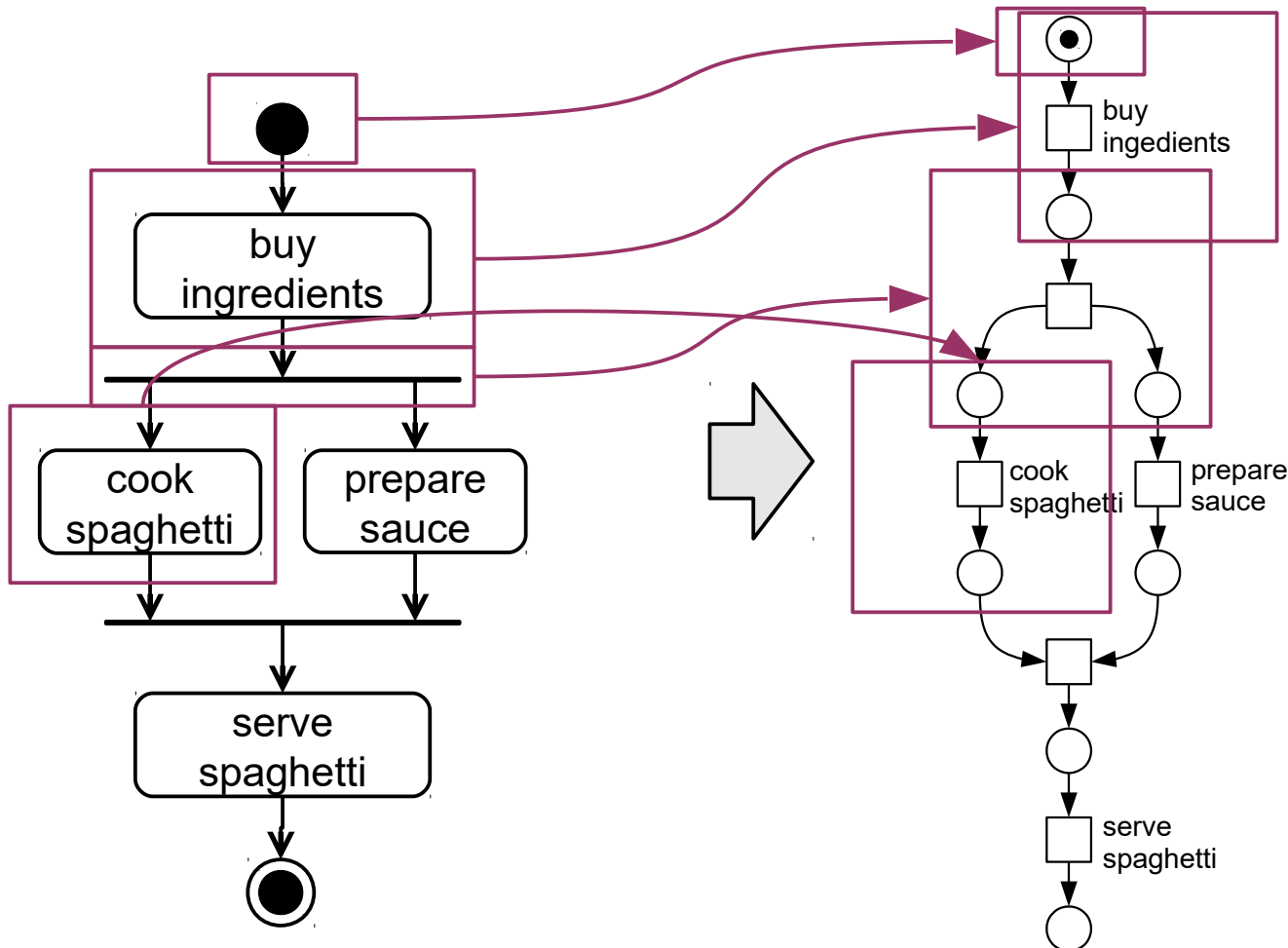
# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



# Model Transformation Rules

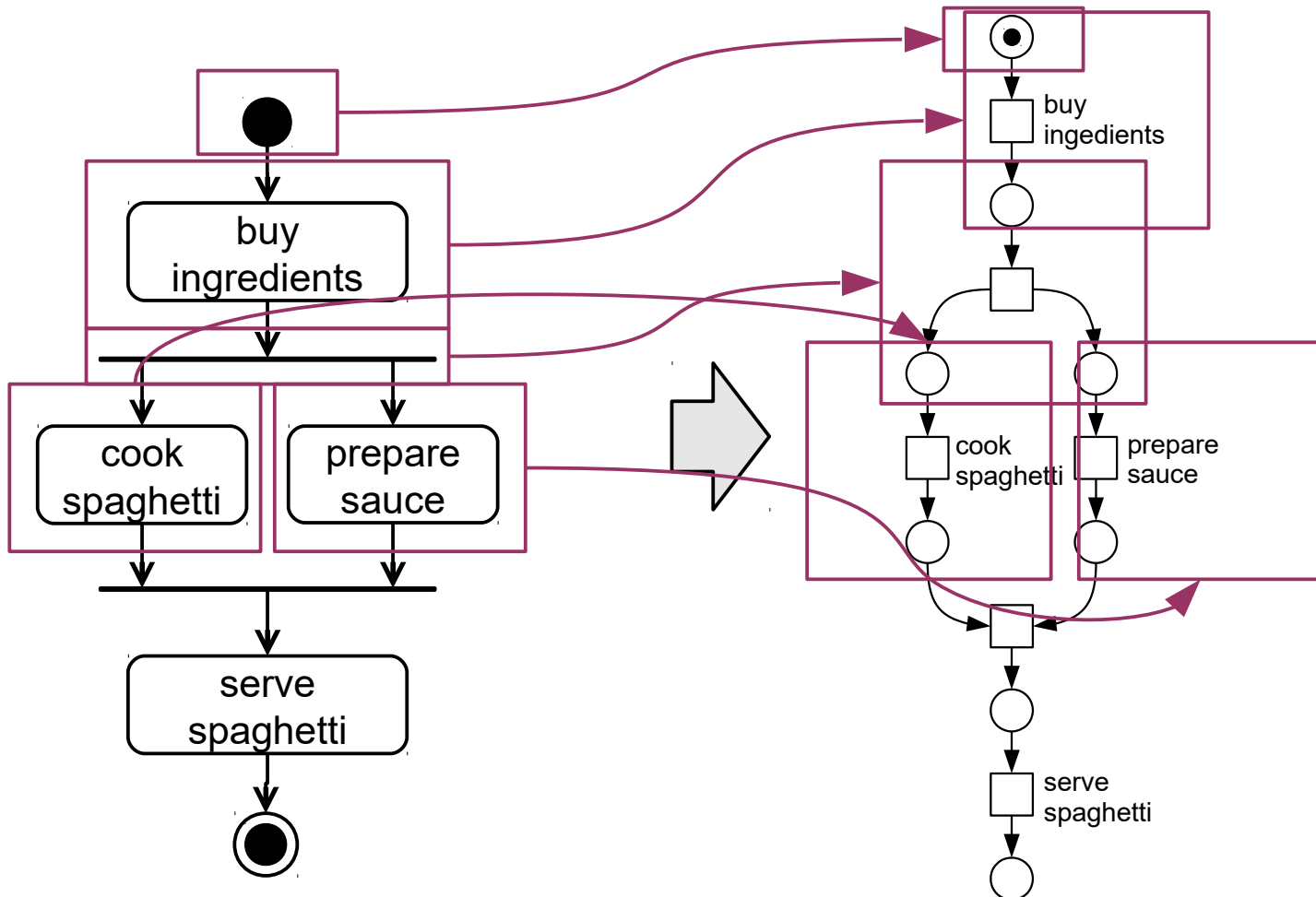
- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases





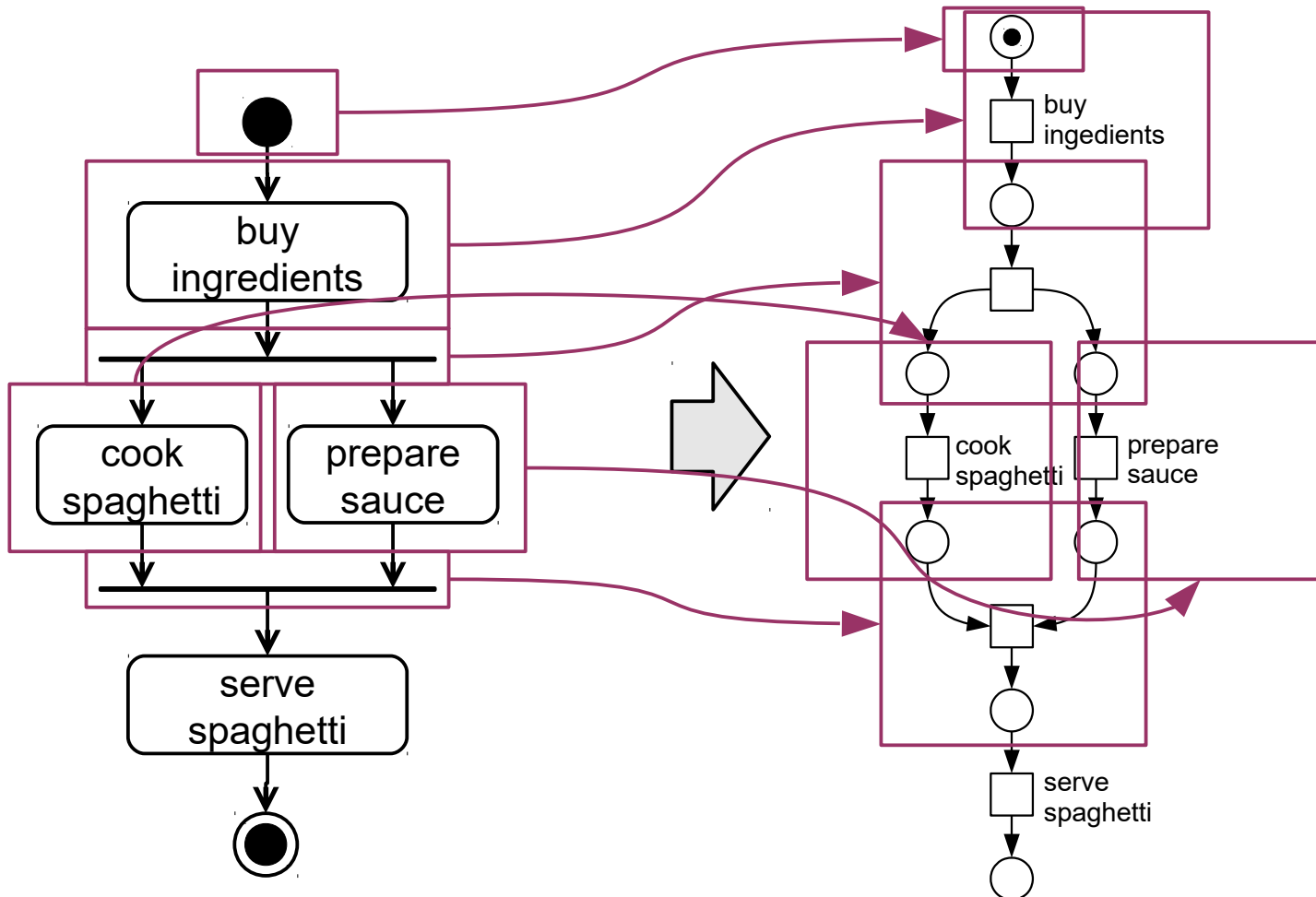
# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



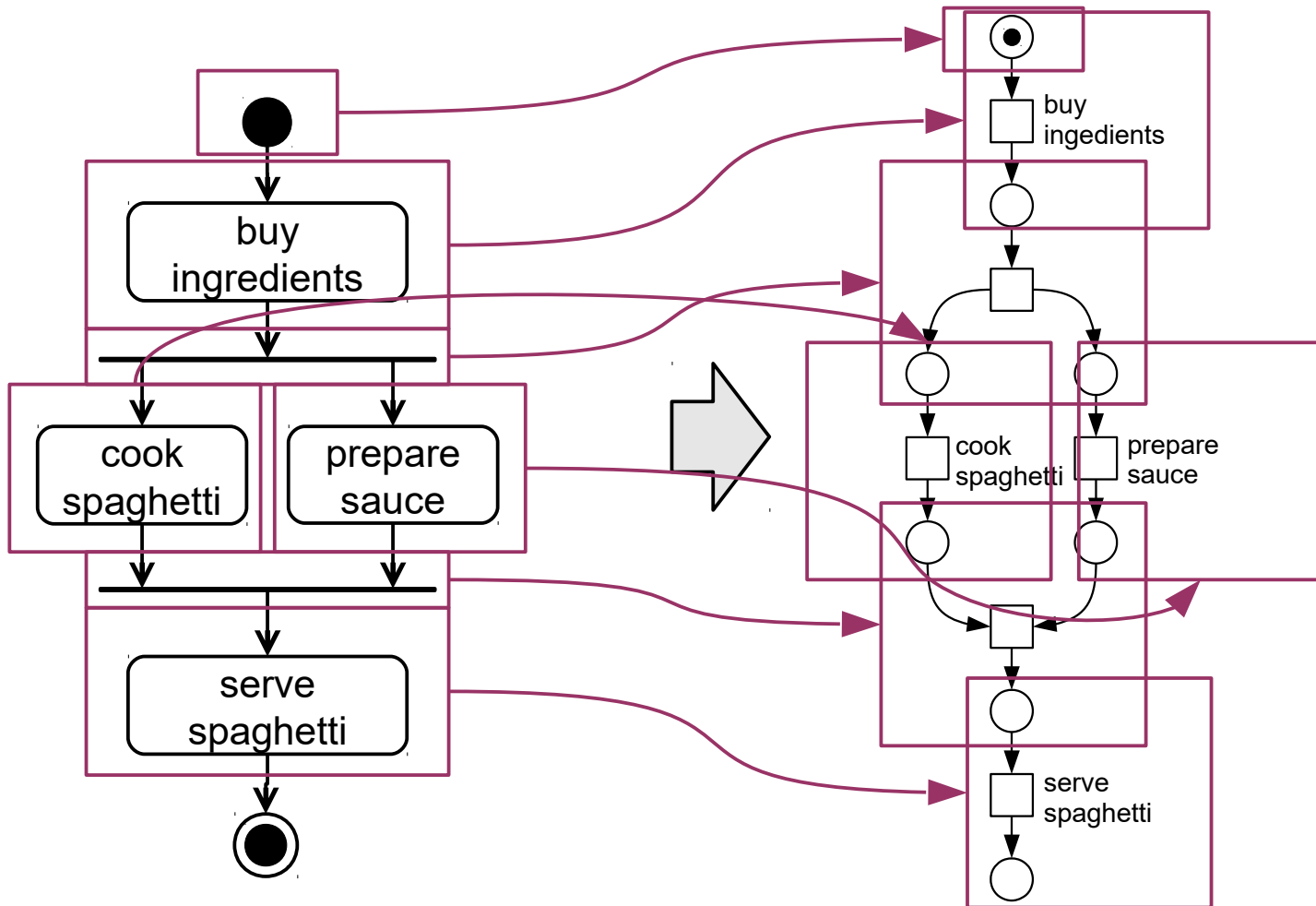
# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



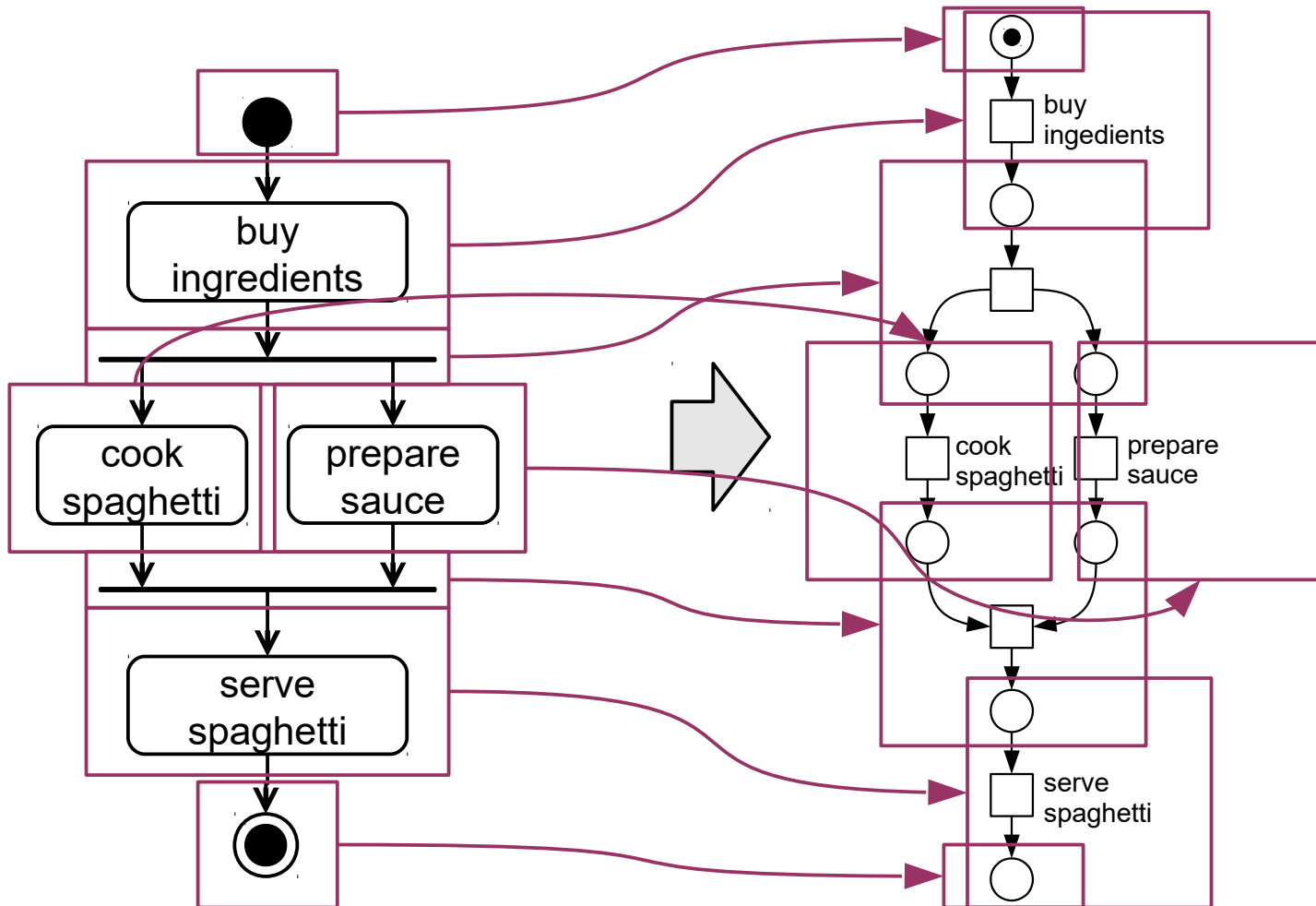
# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



# Model Transformation Rules

- Most model transformation formalisms use some form of **rules** to modularize a transformation for its different cases



# Model Transformation Technology

---

- Many model transformation languages and tools exist

# Model Transformation Technology

---

- Many model transformation languages and tools exist
  - many different ideas and underlying philosophies

# Model Transformation Technology

---

- Many model transformation languages and tools exist
  - many different ideas and underlying philosophies
    - inspired by compiler theory, constraint solving, graph theory, procedural programming, functional programming,

# Model Transformation Technology

---

- Many model transformation languages and tools exist
  - many different ideas and underlying philosophies
    - inspired by compiler theory, constraint solving, graph theory, procedural programming, functional programming,
  - some approaches have perished, some have evolved



# Model Transformation Technology

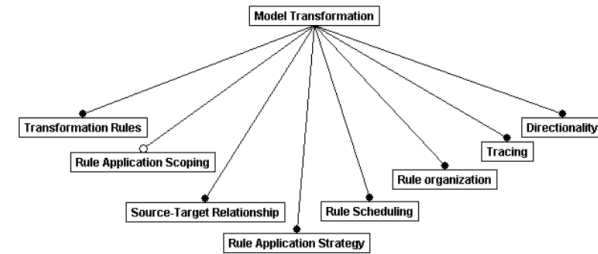
---

- Many model transformation languages and tools exist
  - many different ideas and underlying philosophies
    - inspired by compiler theory, constraint solving, graph theory, procedural programming, functional programming,
  - some approaches have perished, some have evolved
- Examples:
  - Query/View/Transformation-Relations (QVT-R), QVT-Operational (QVT-O), Atlas Transformation language (ATL), Epsilon Transformation language, Story Diagrams, MOFLON, Triple Graph Grammar Interpreter (TGG-Interpreter), VIATRA, UMLX, ATOM, Tefkat, Modgraph, GROOVE, Henshin, ...

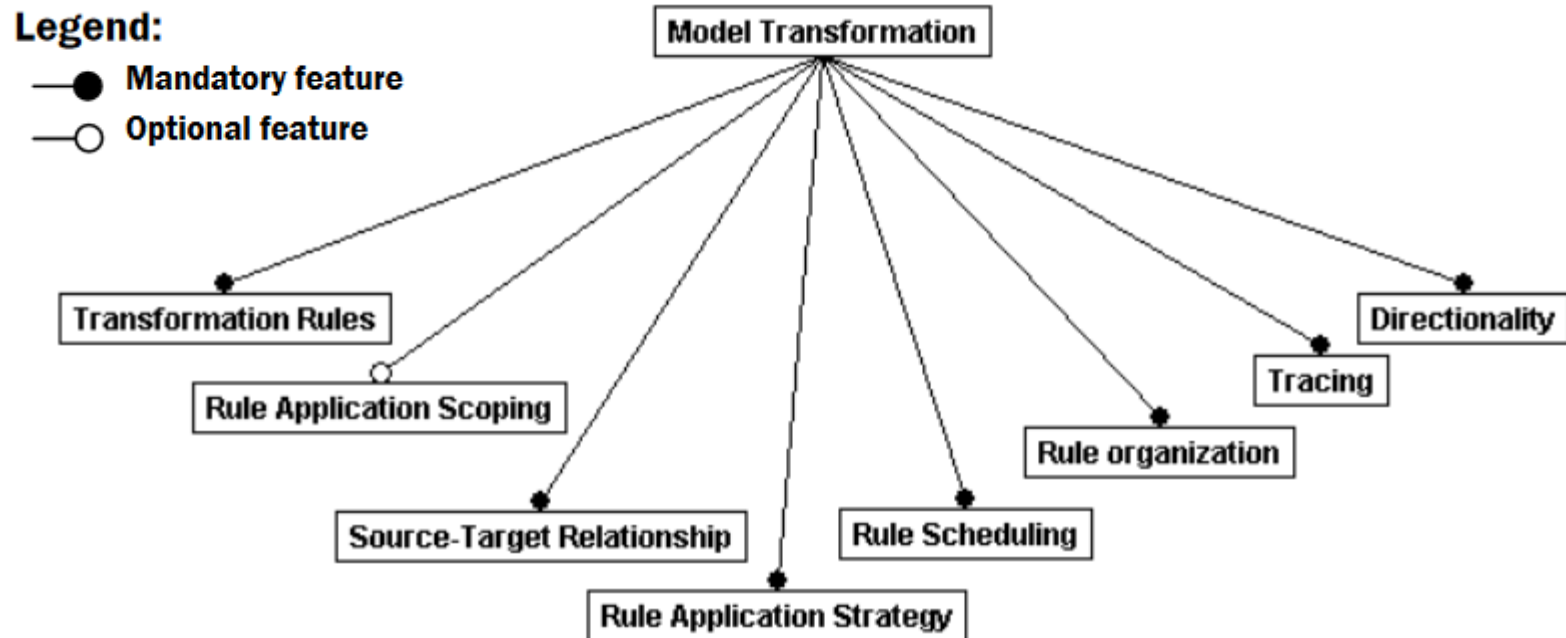
# Model Transformation Taxonomy

- Krzysztof Czarnecki and Simon Helsen, “Classification of Model Transformation Approaches”, Workshop on Generative Techniques in the Context of Model-Driven Approaches, OOPSLA 2003

— <http://www.s23m.com/oopsla2003/czarnecki.pdf>



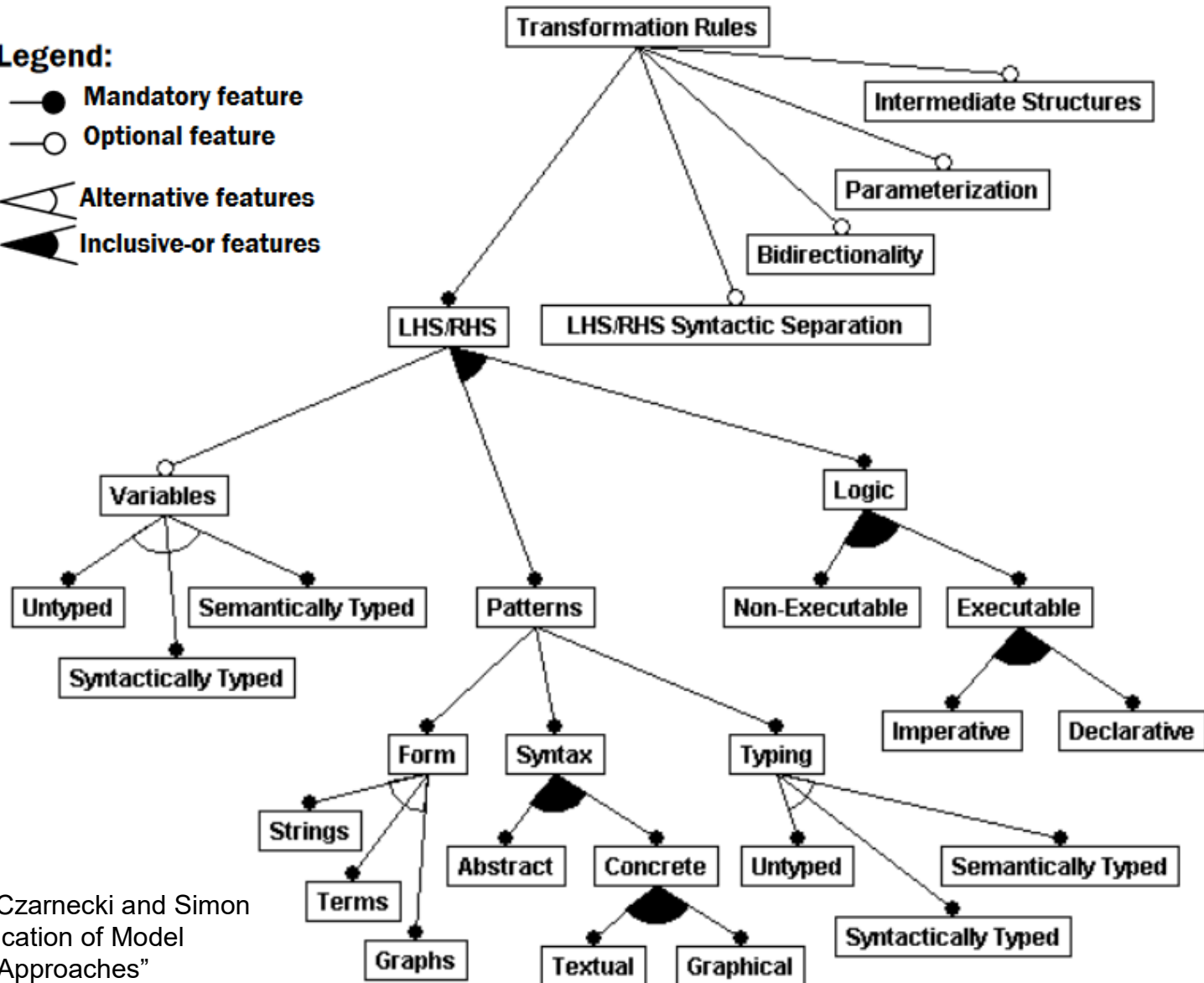
# Model Transformation Taxonomy



# Model Transformation Taxonomy

## Legend:

- Mandatory feature
- Optional feature
- ⌋ Alternative features
- ⌋ Inclusive-or features



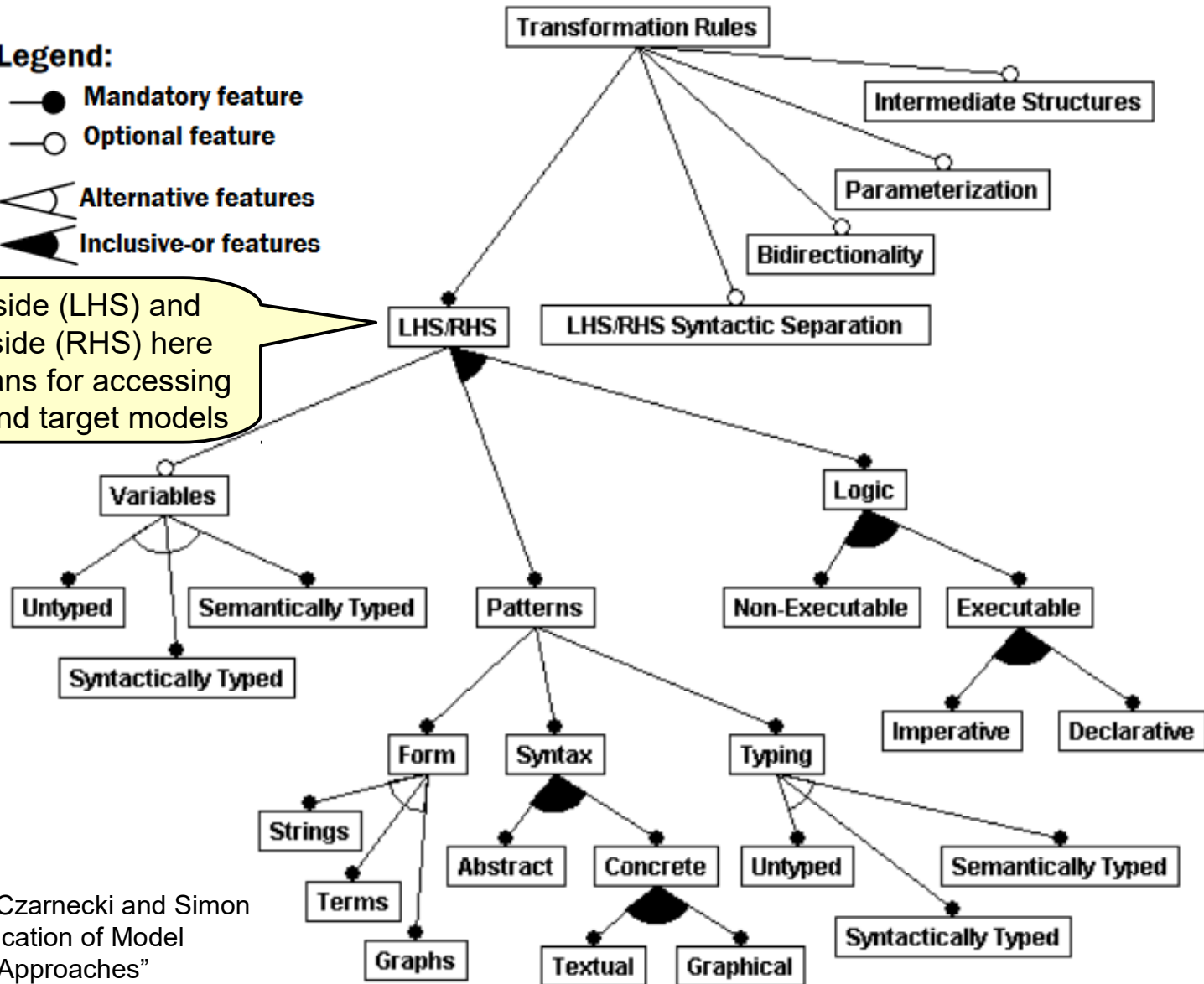
From Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches"

# Model Transformation Taxonomy

## Legend:

- Mandatory feature
- Optional feature
- ⌵ Alternative features
- ⌒ Inclusive-or features

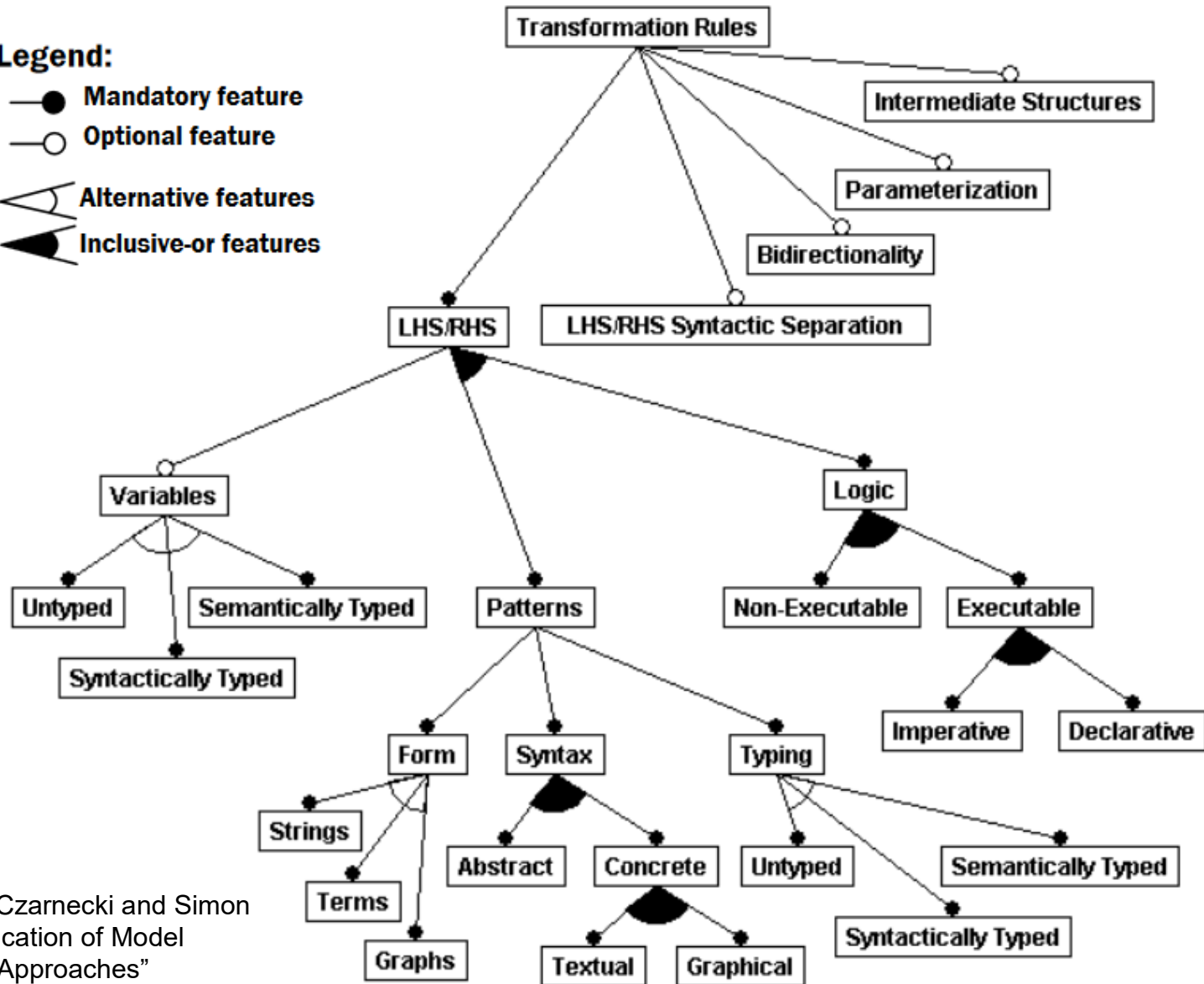
Left-hand side (LHS) and right-hand side (RHS) here refers to means for accessing the source and target models



# Model Transformation Taxonomy

## Legend:

- Mandatory feature
- Optional feature
- ⌵ Alternative features
- ⌵ Inclusive-or features

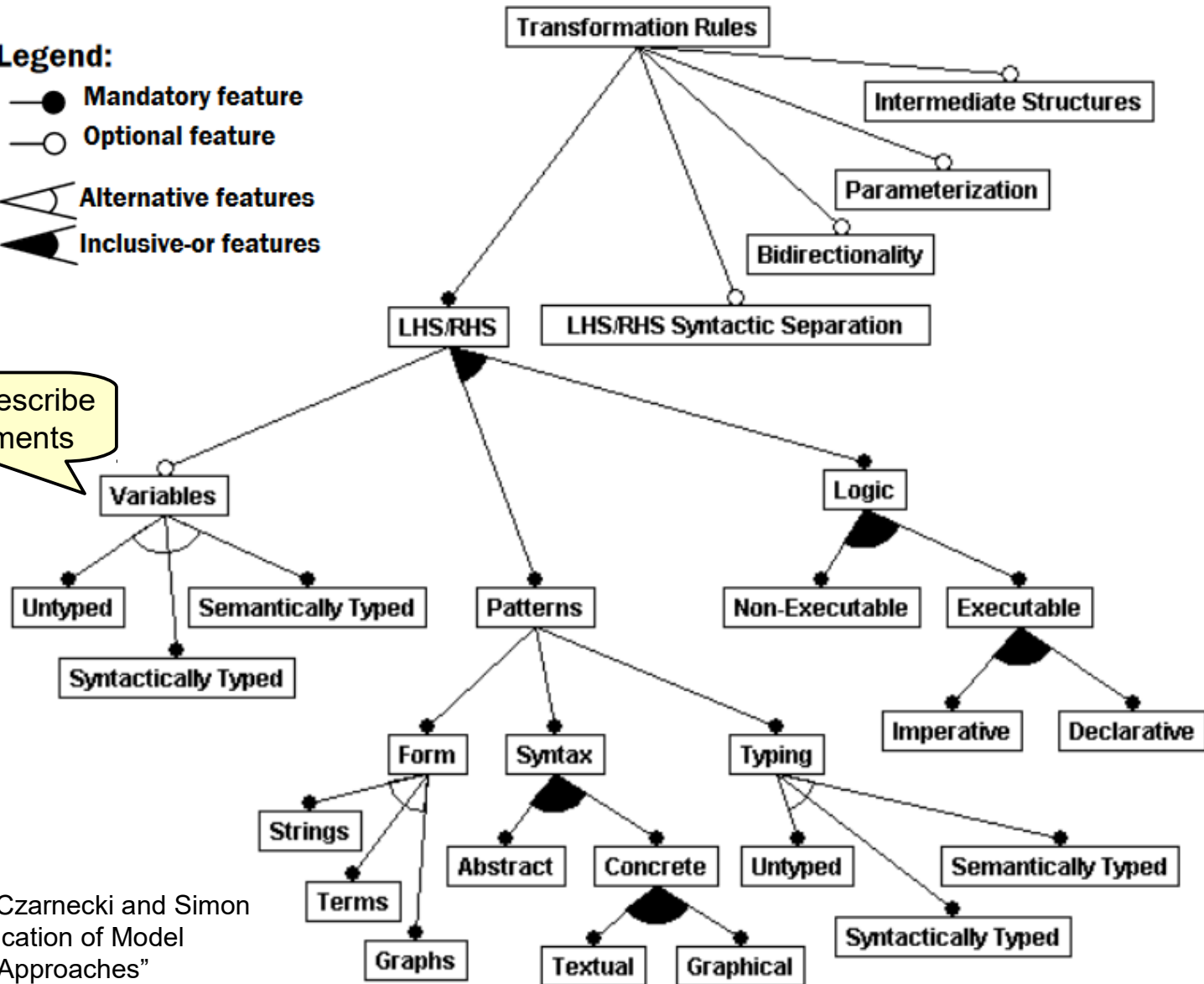


From Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches"

# Model Transformation Taxonomy

## Legend:

- Mandatory feature
- Optional feature
- ⌋ Alternative features
- ⌋ Inclusive-or features

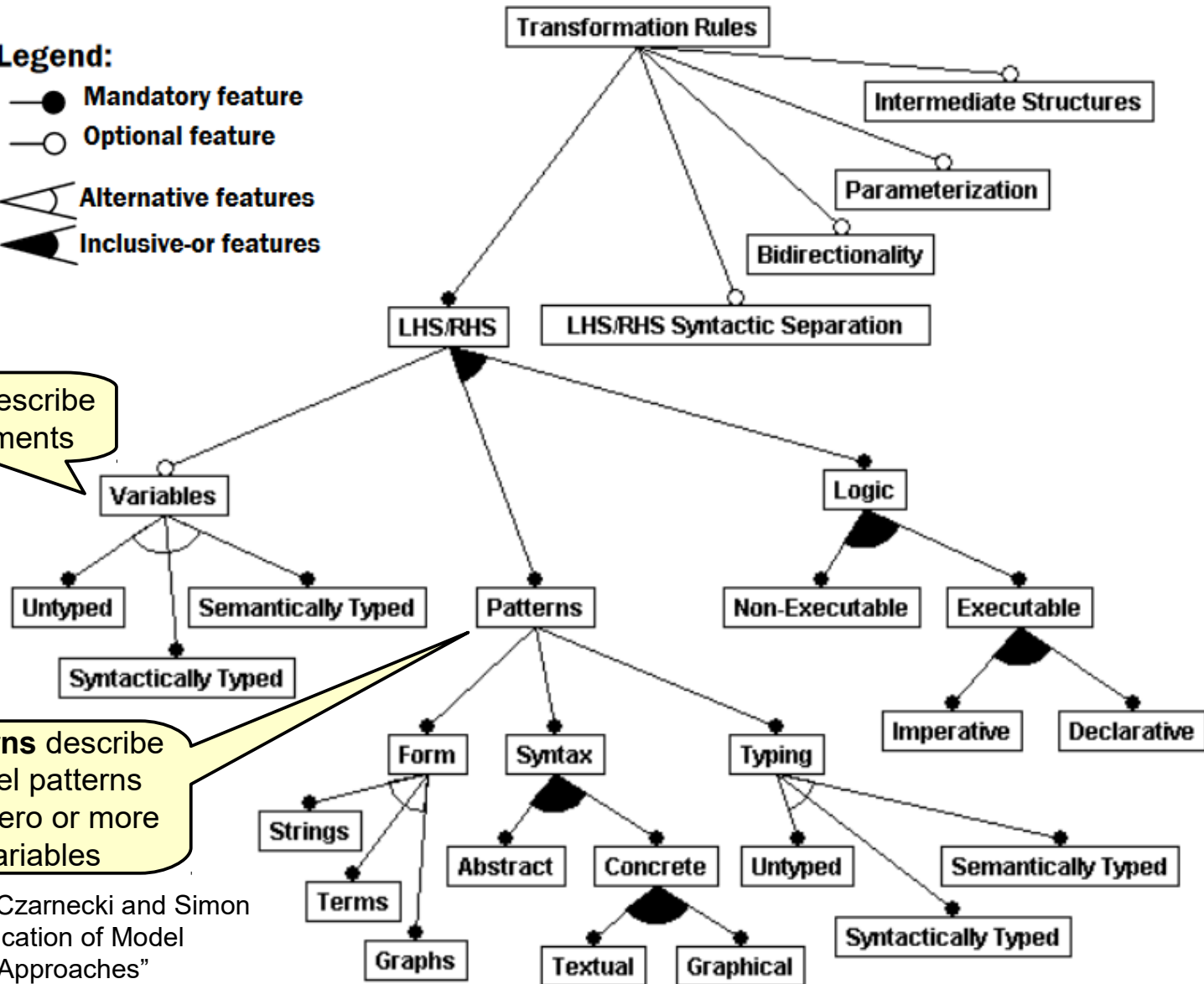


From Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches"

# Model Transformation Taxonomy

## Legend:

- Mandatory feature
- Optional feature
- ⌞ Alternative features
- ⌞ Inclusive-or features



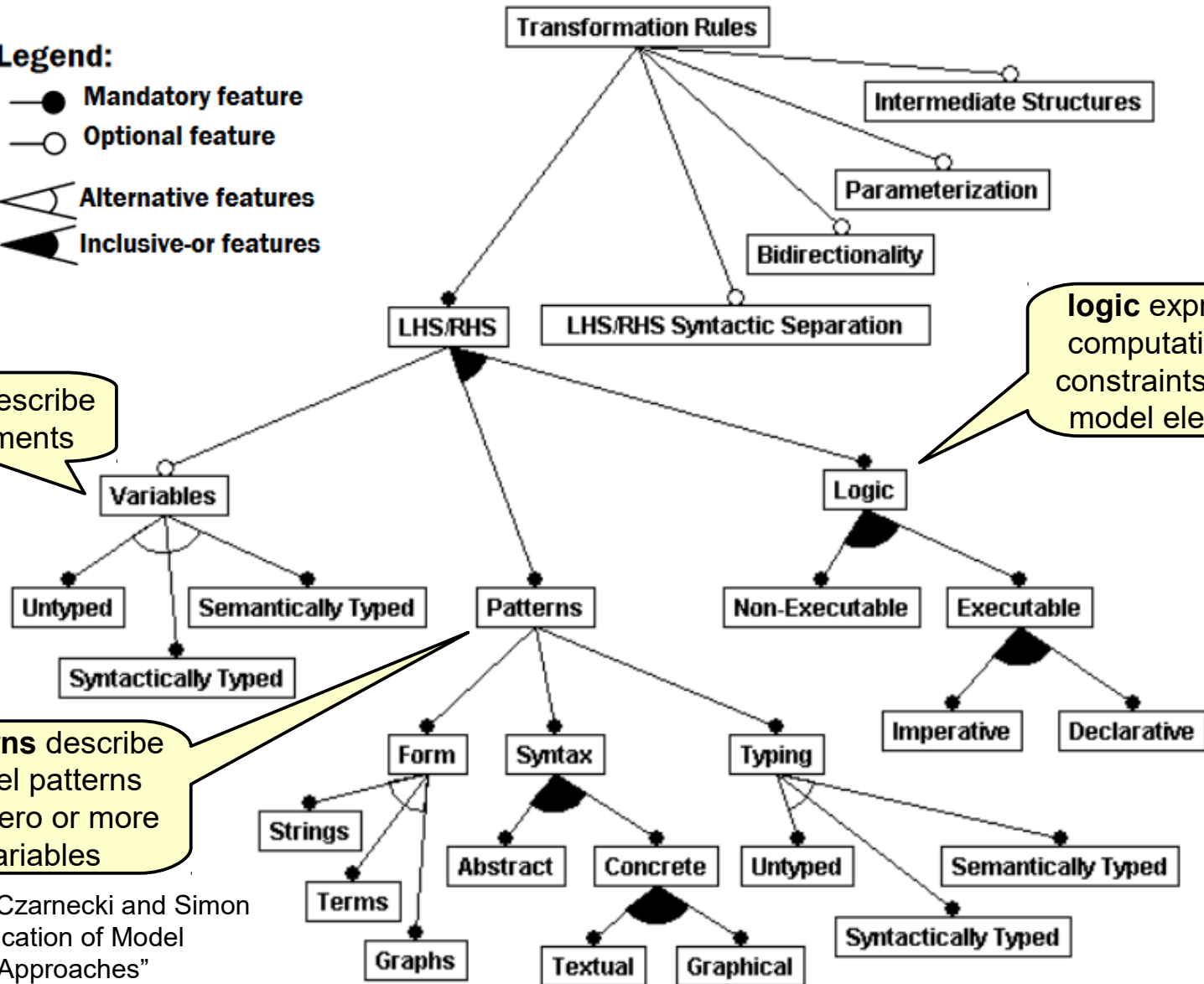
From Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches"



# Model Transformation Taxonomy

## Legend:

- Mandatory feature
- Optional feature
- ⌋ Alternative features
- ⌋ Inclusive-or features



From Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches"

# Model Transformation Taxonomy

## Imperative vs. Declarative

---

- **Imperative** (also called **operational**) Logic:

# Model Transformation Taxonomy

## Imperative vs. Declarative

---

- **Imperative** (also called **operational**) Logic:
  - Detailed instructions are given on how a certain computation must be carried out

# Model Transformation Taxonomy

## Imperative vs. Declarative

---

- **Imperative** (also called **operational**) Logic:
  - Detailed instructions are given on how a certain computation must be carried out
    - by a list of instructions or in form of an algorithm

# Model Transformation Taxonomy

## Imperative vs. Declarative

---

- **Imperative** (also called **operational**) Logic:
  - Detailed instructions are given on how a certain computation must be carried out
    - by a list of instructions or in form of an algorithm
- **Declarative** Logic:

# Model Transformation Taxonomy

## Imperative vs. Declarative

---

- **Imperative** (also called **operational**) Logic:
  - Detailed instructions are given on how a certain computation must be carried out
    - by a list of instructions or in form of an algorithm
- **Declarative** Logic:
  - constraints or conditions that describe a correct outcome of a computation are given

# Model Transformation Taxonomy

## Imperative vs. Declarative

---

- **Imperative** (also called **operational**) Logic:
  - Detailed instructions are given on how a certain computation must be carried out
    - by a list of instructions or in form of an algorithm
- **Declarative** Logic:
  - constraints or conditions that describe a correct outcome of a computation are given
  - declarative logic can be **executable** if some solver or algorithm exists that can find a solution that satisfies the given constraints or conditions

# Model Transformation Taxonomy

## Imperative vs. Declarative

---

- **Imperative** (also called **operational**) Logic:
  - Detailed instructions are given on how a certain computation must be carried out
    - by a list of instructions or in form of an algorithm
- **Declarative** Logic:
  - constraints or conditions that describe a correct outcome of a computation are given
  - declarative logic can be **executable** if some solver or algorithm exists that can find a solution that satisfies the given constraints or conditions
- for declarative logic the processing is *potentially slower* than for imperative logic



# Model Transformation Taxonomy

## Imperative vs. Declarative

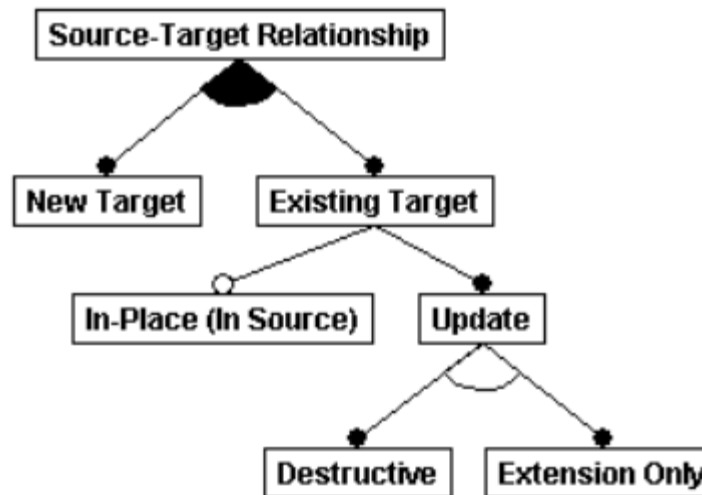
---

- **Imperative** (also called **operational**) Logic:
  - Detailed instructions are given on how a certain computation must be carried out
    - by a list of instructions or in form of an algorithm
- **Declarative** Logic:
  - constraints or conditions that describe a correct outcome of a computation are given
  - declarative logic can be **executable** if some solver or algorithm exists that can find a solution that satisfies the given constraints or conditions
- for declarative logic the processing is *potentially slower* than for imperative logic
- declarative logic is often *easier to understand* by users

# Model Transformation Taxonomy

## Legend:

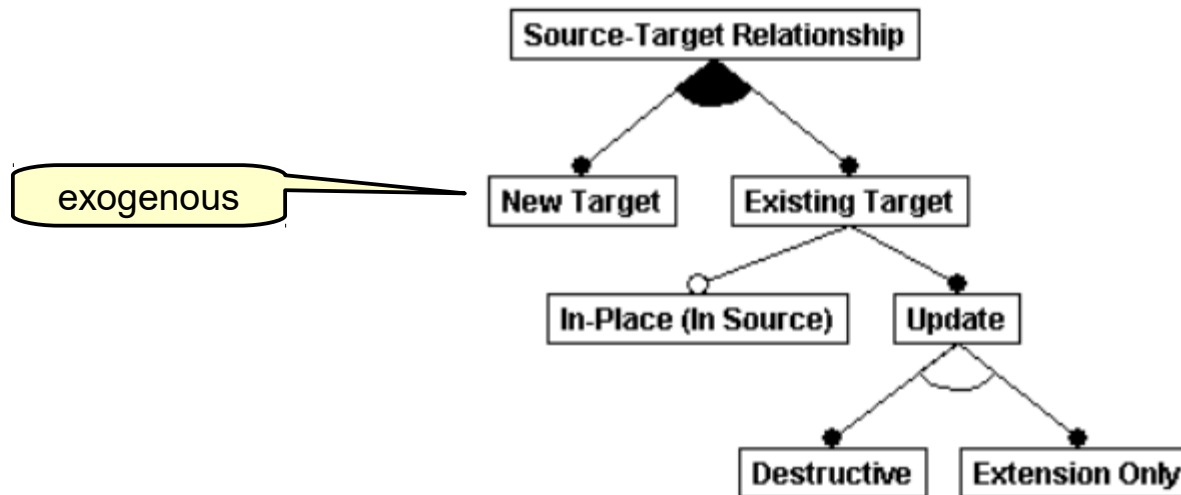
- Mandatory feature
- Optional feature
- ∧ Alternative features
- ▲ Inclusive-or features



# Model Transformation Taxonomy

## Legend:

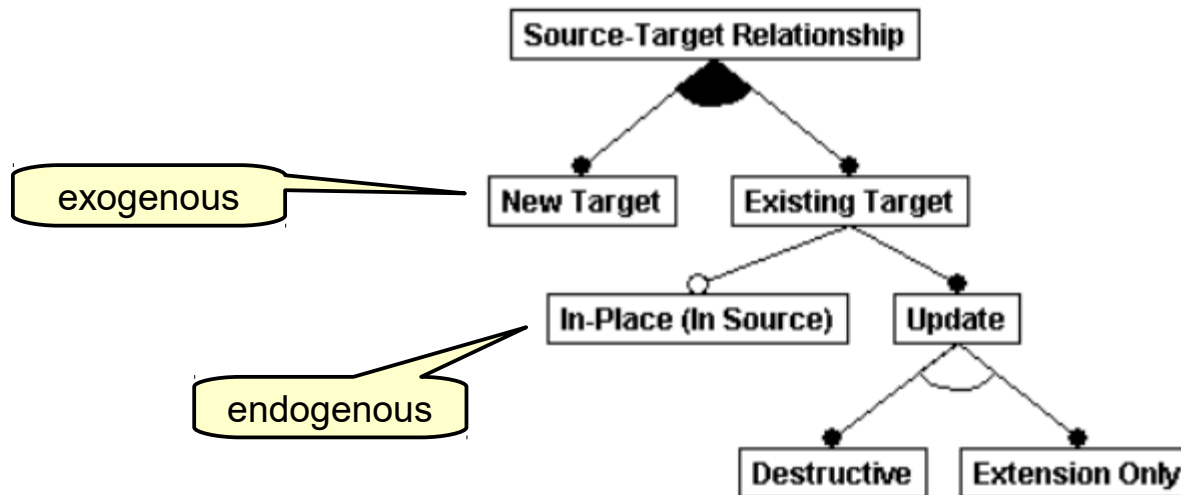
- Mandatory feature
- Optional feature
- ∧ Alternative features
- ▲ Inclusive-or features



# Model Transformation Taxonomy

## Legend:

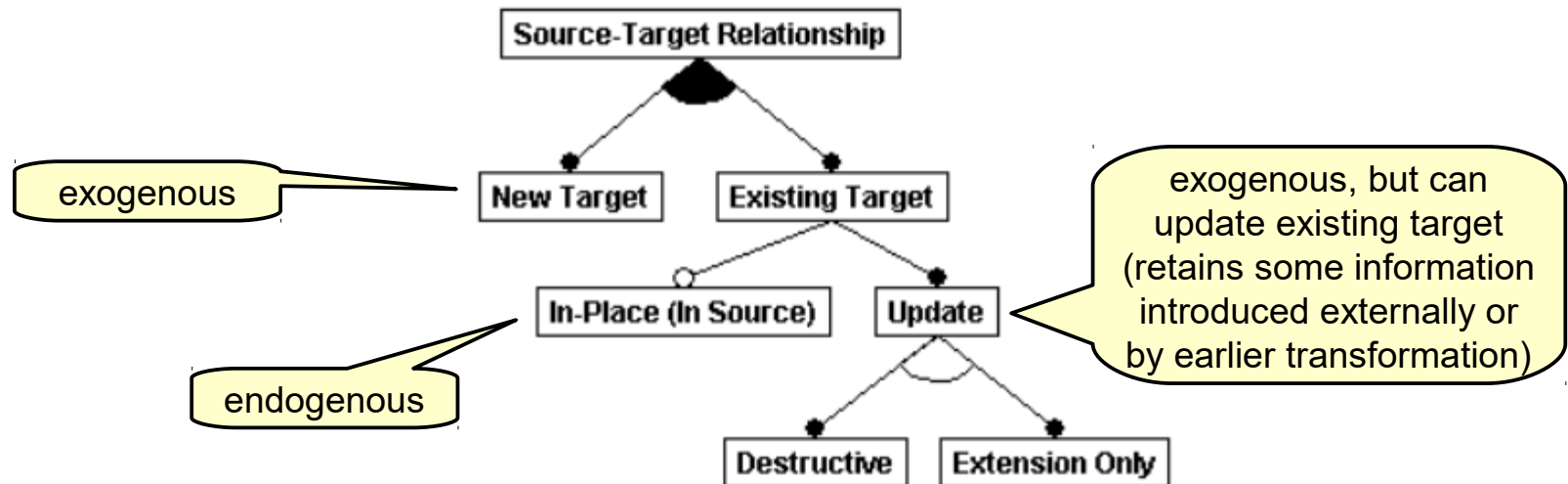
- Mandatory feature
- Optional feature
- ∧ Alternative features
- ▲ Inclusive-or features



# Model Transformation Taxonomy

## Legend:

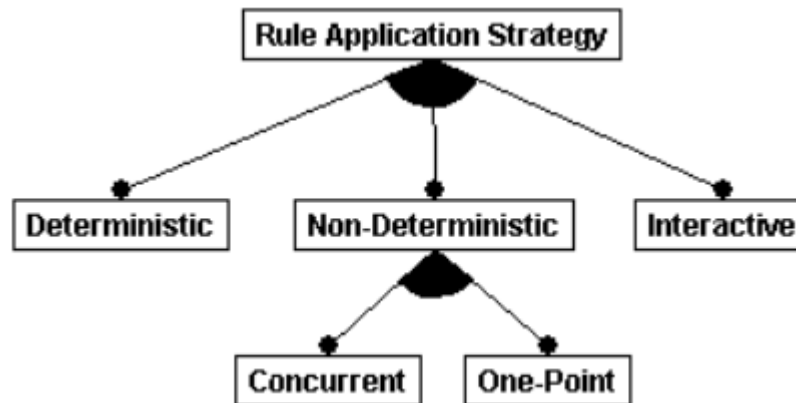
- Mandatory feature
- Optional feature
- ⌵ Alternative features
- ⌴ Inclusive-or features



# Model Transformation Taxonomy

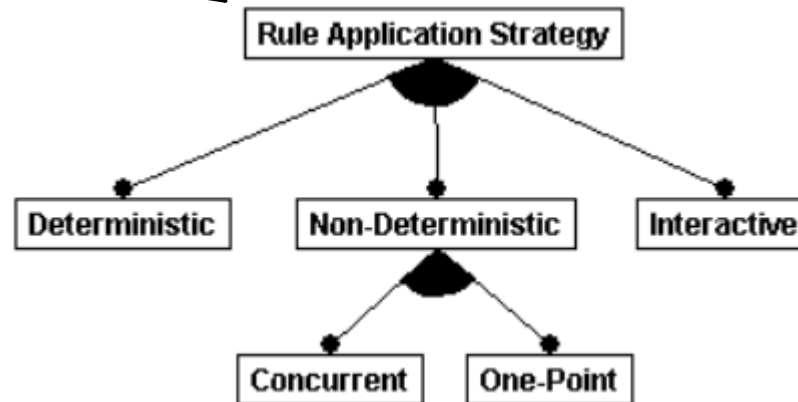
## Legend:

- Mandatory feature
- Optional feature
- ∧ Alternative features
- ▶ Inclusive-or features



# Model Transformation Taxonomy

is strategy for selecting a rule and its application location in the model

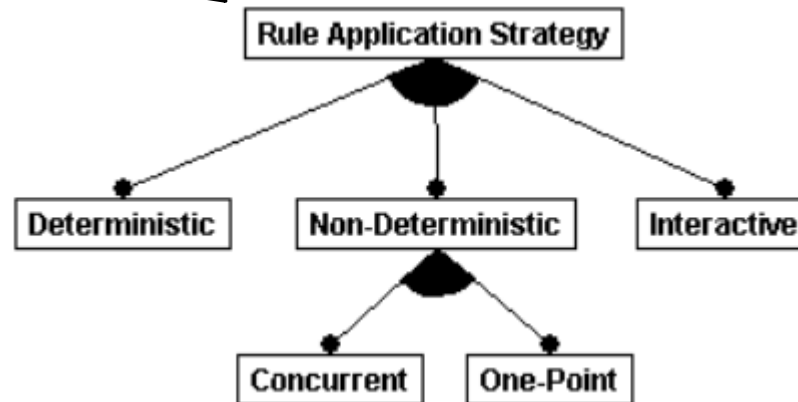


## Legend:

- Mandatory feature
- Optional feature
- ◀ Alternative features
- ▶ Inclusive-or features

# Model Transformation Taxonomy

is strategy for selecting a rule and its application location in the model



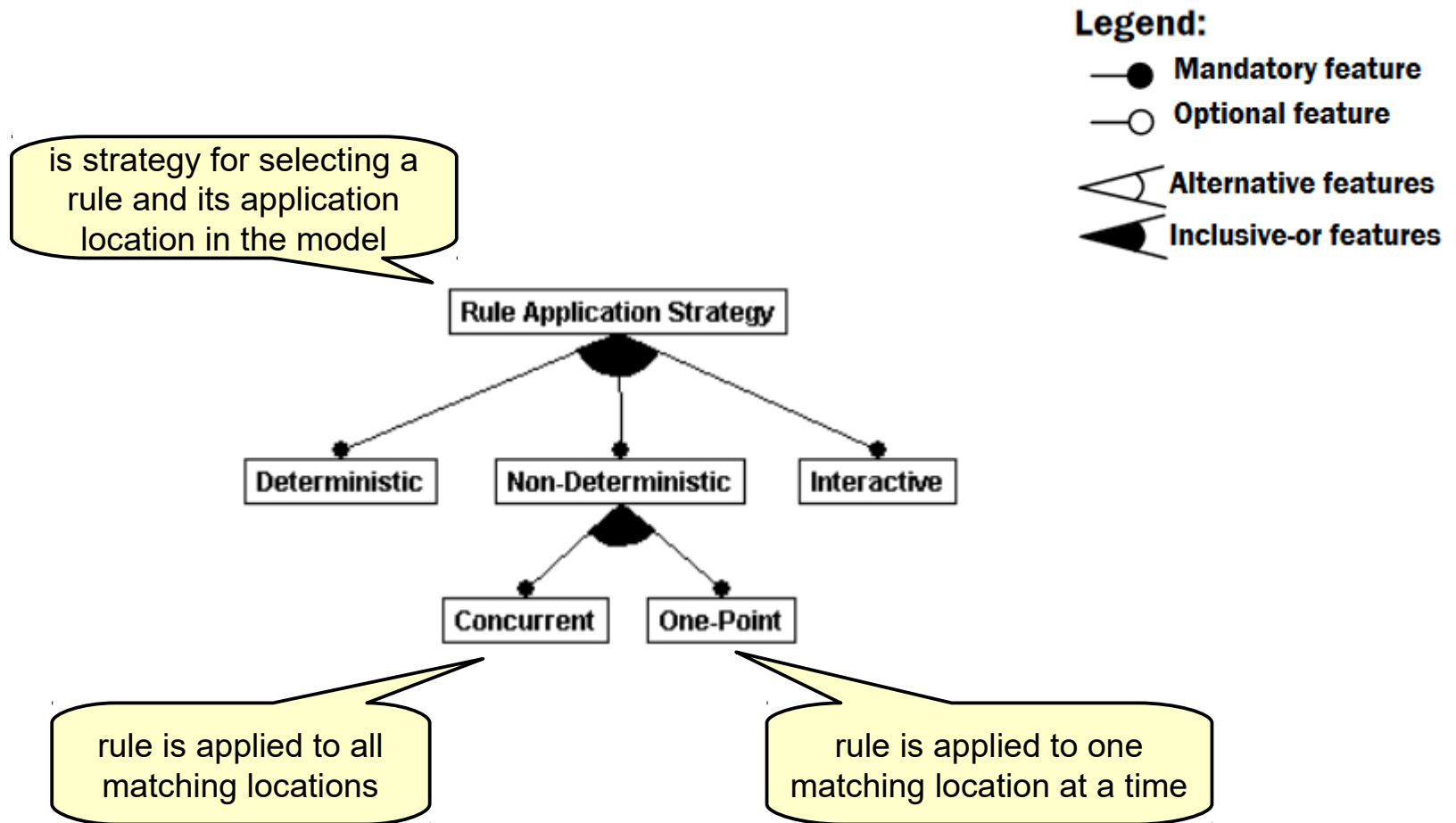
rule is applied to all matching locations

## Legend:

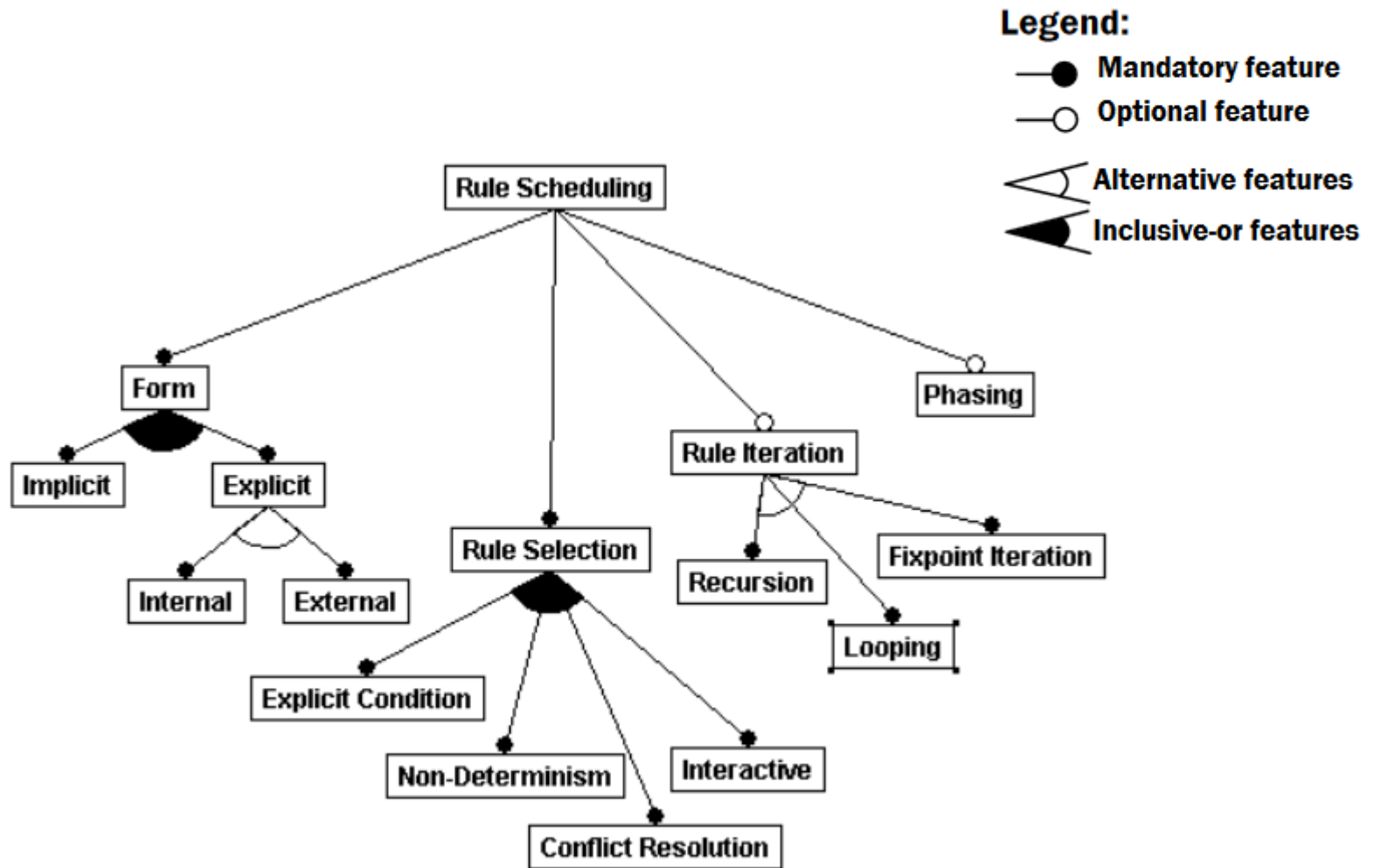
- Mandatory feature
- Optional feature
- ∧ Alternative features
- ▲ Inclusive-or features



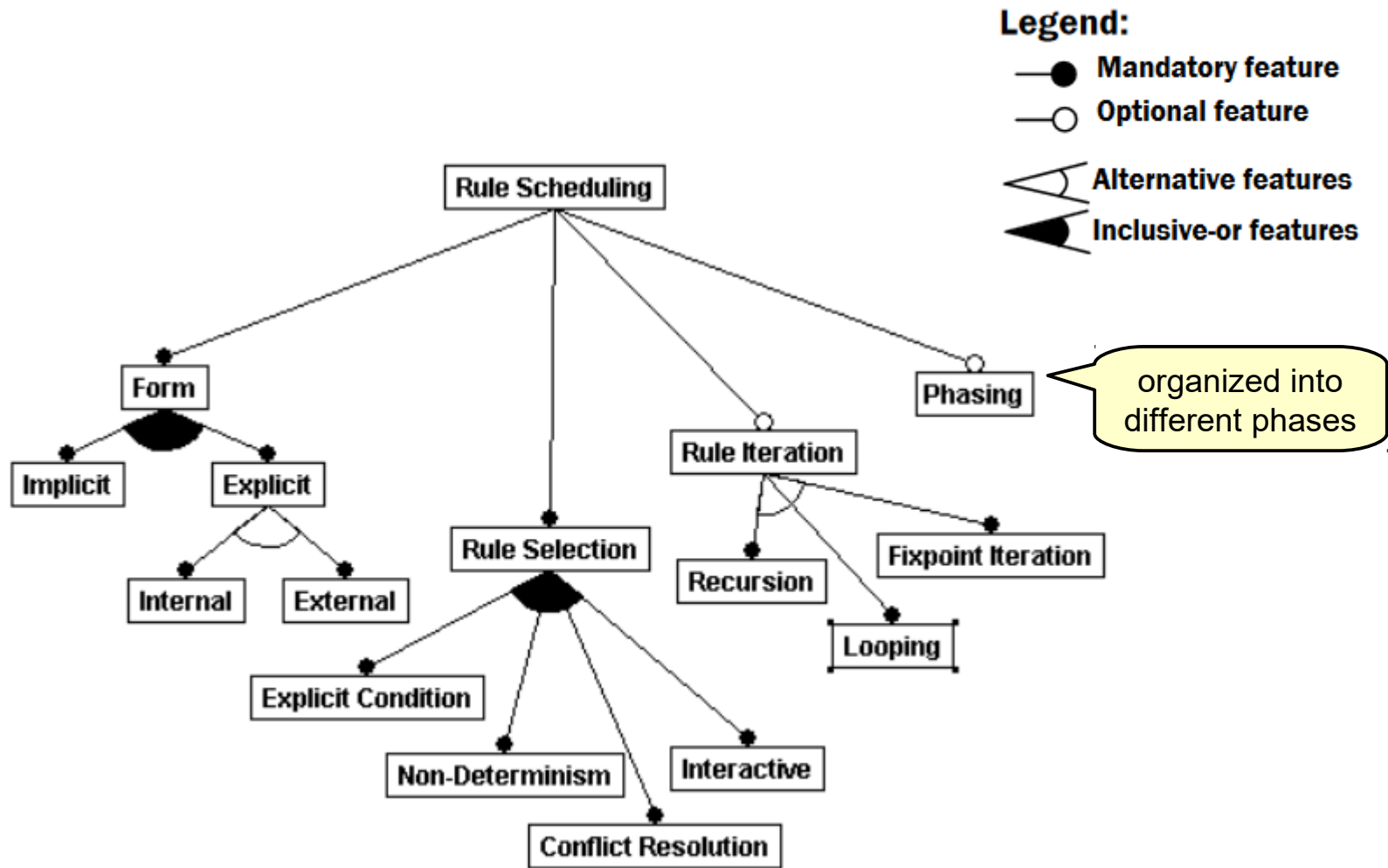
# Model Transformation Taxonomy



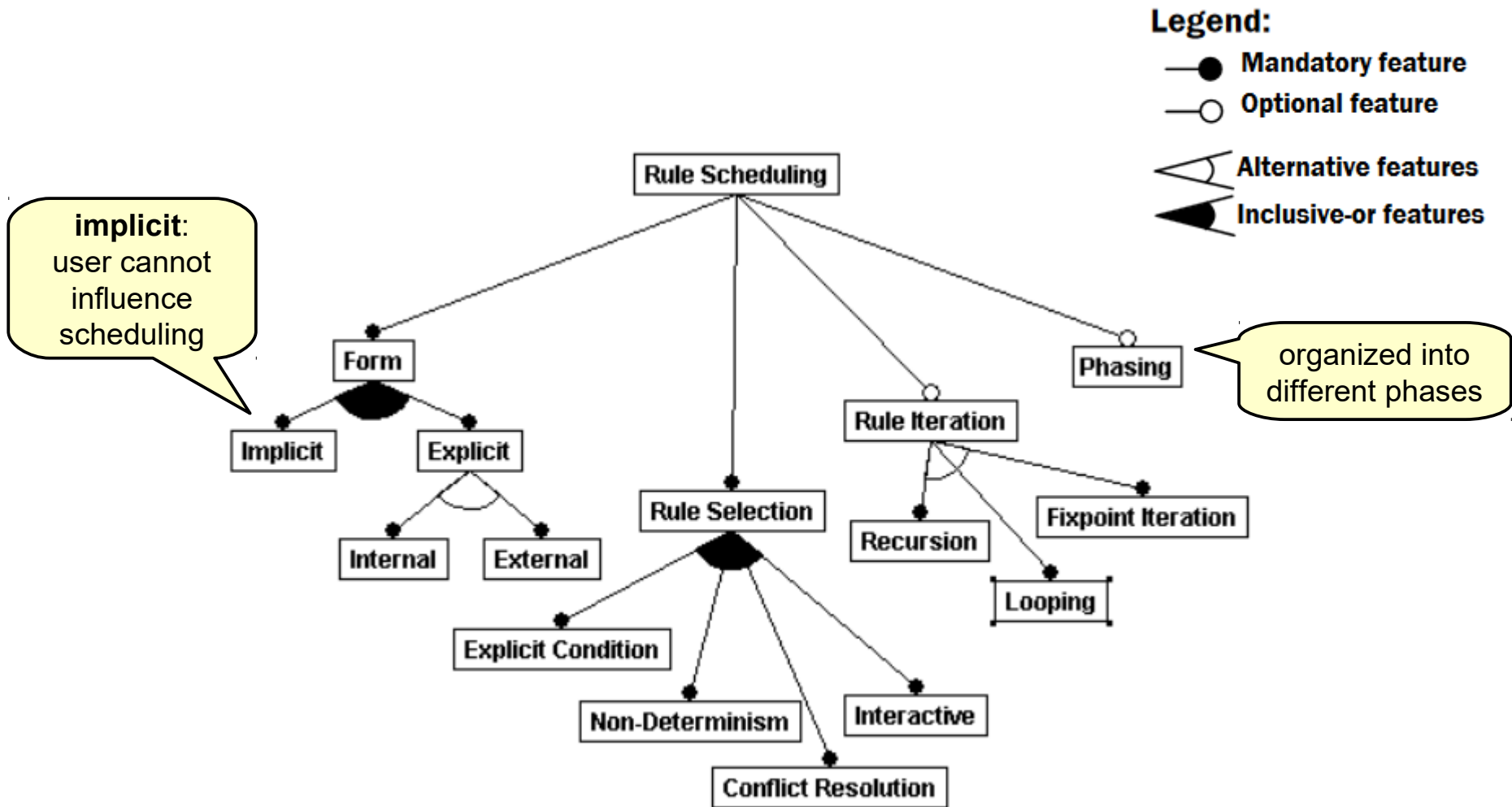
# Model Transformation Taxonomy



# Model Transformation Taxonomy



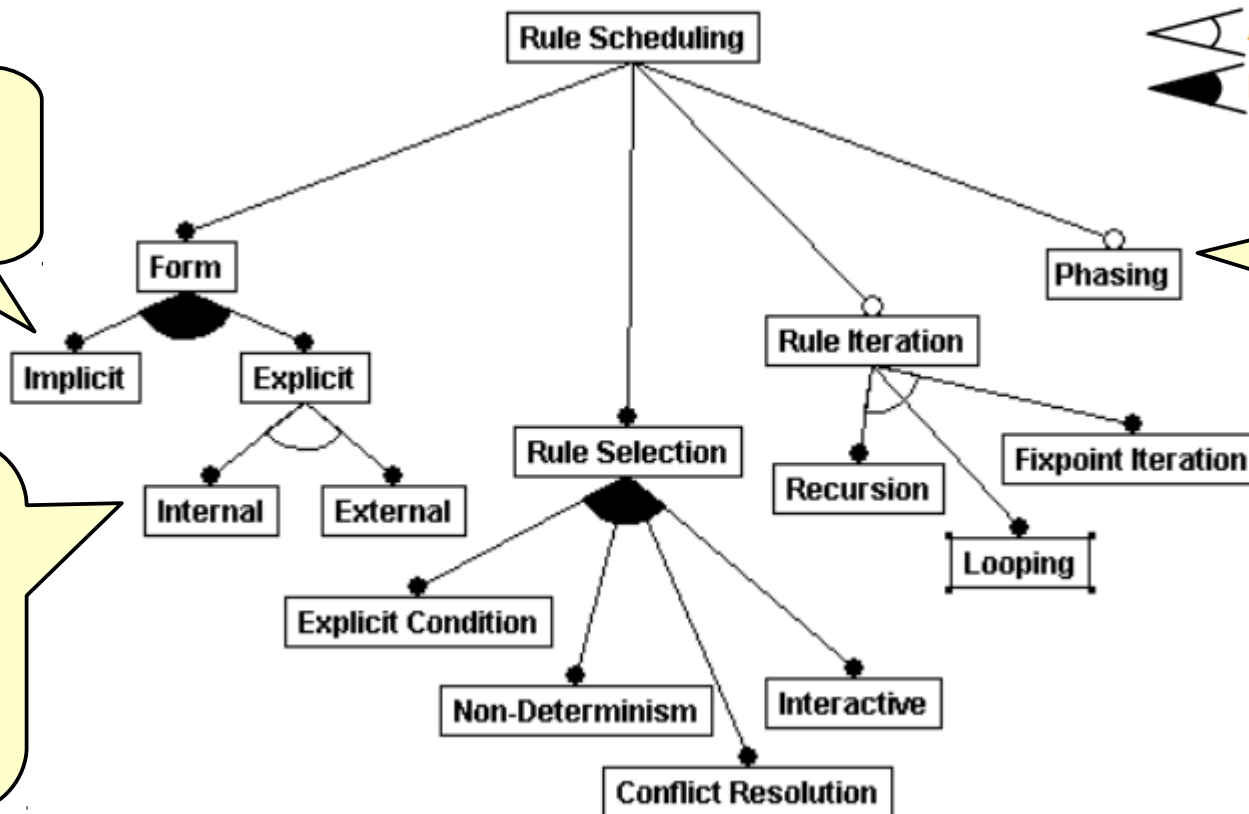
# Model Transformation Taxonomy



# Model Transformation Taxonomy

## Legend:

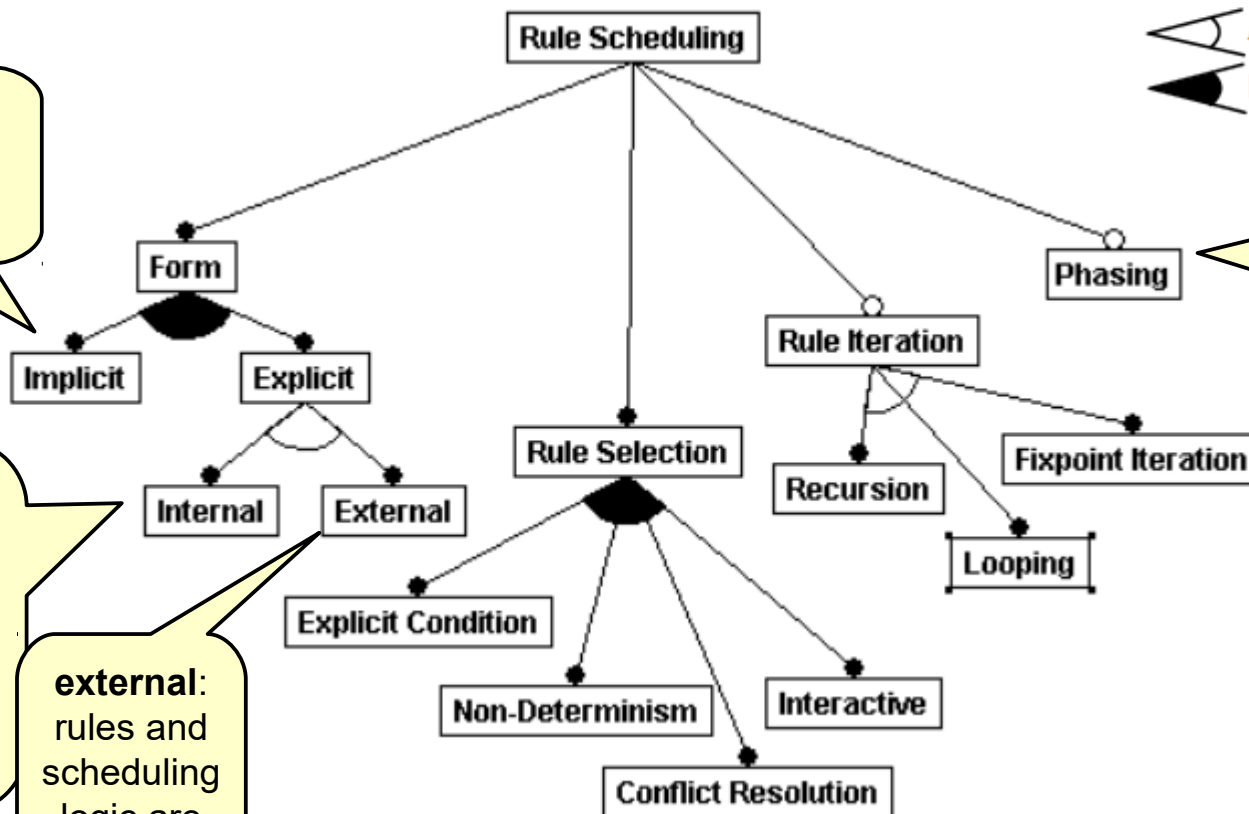
- Mandatory feature
- Optional feature
- ⌵ Alternative features
- ⬤ Inclusive-or features



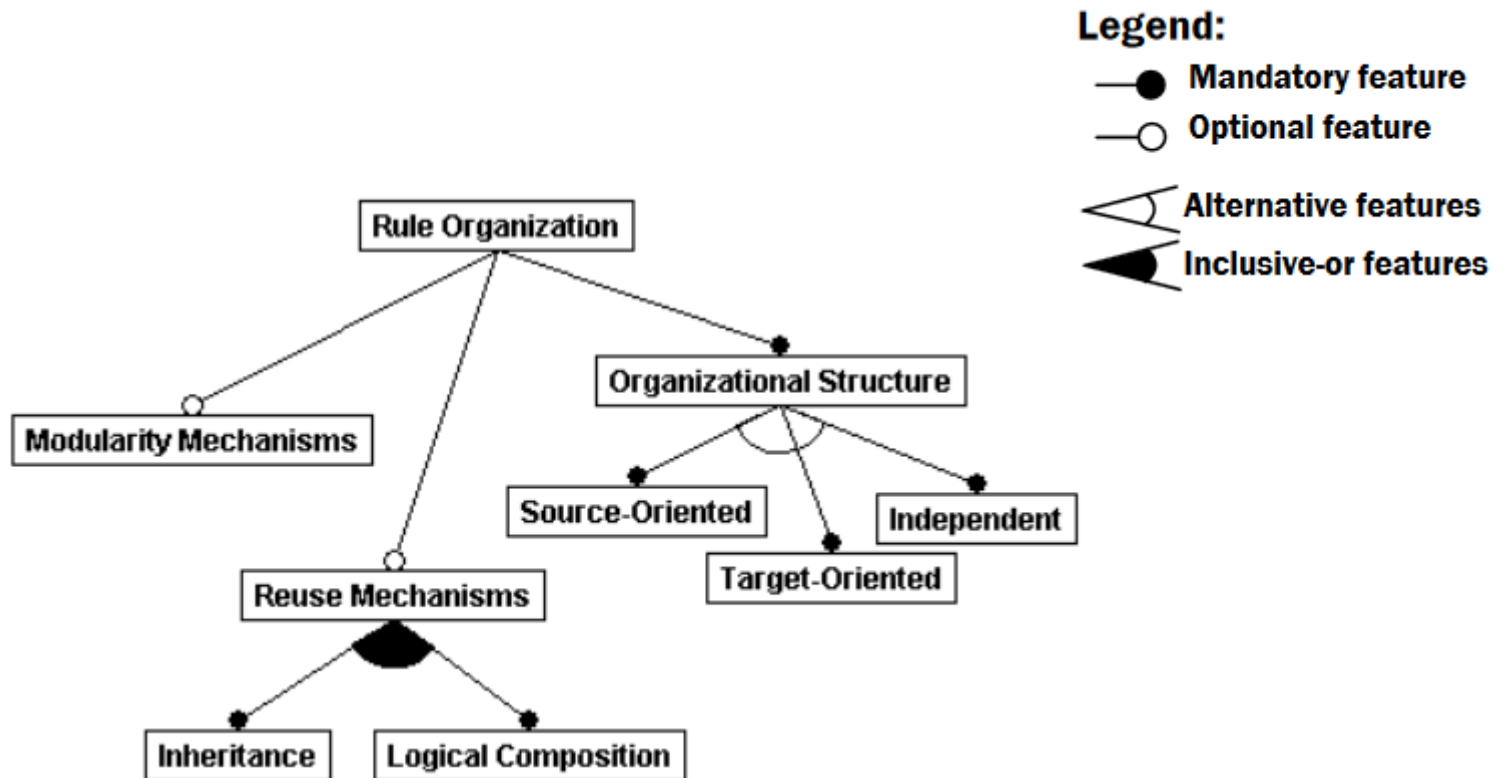
# Model Transformation Taxonomy

## Legend:

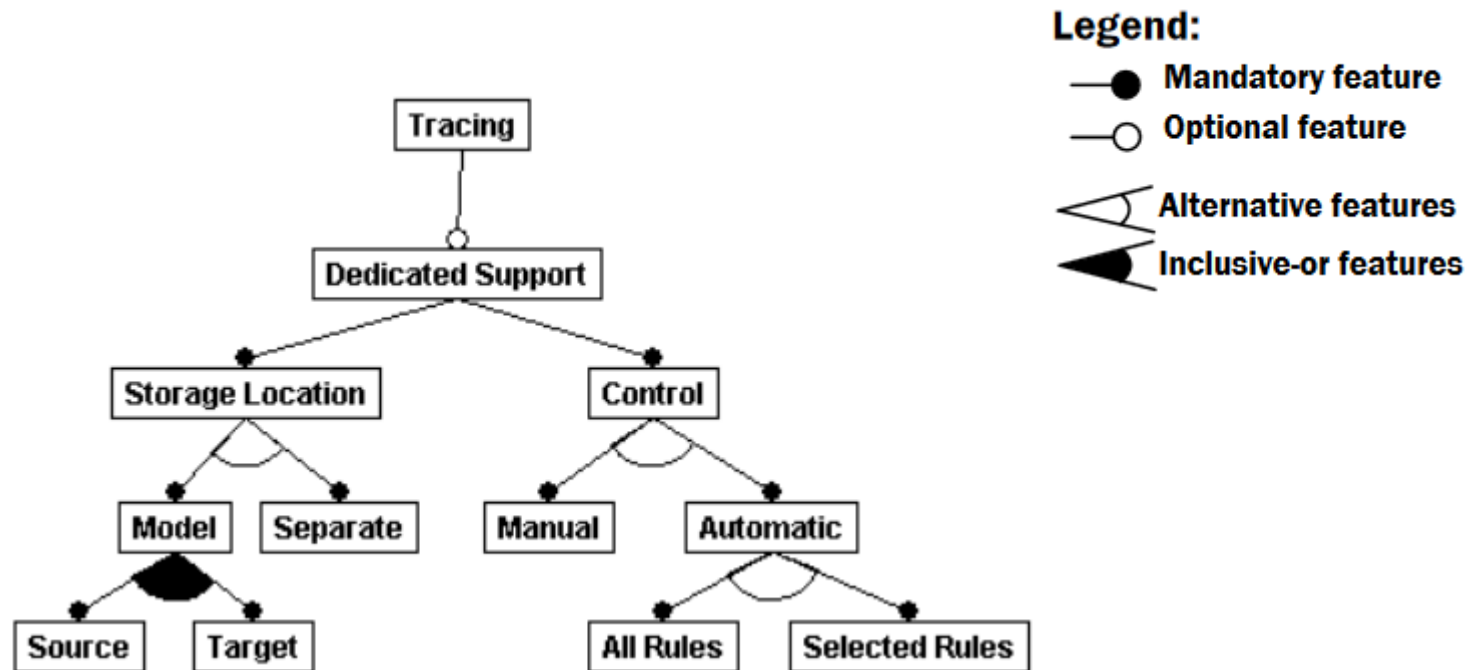
- Mandatory feature
- Optional feature
- ⌵ Alternative features
- ⌵ Inclusive-or features



# Model Transformation Taxonomy

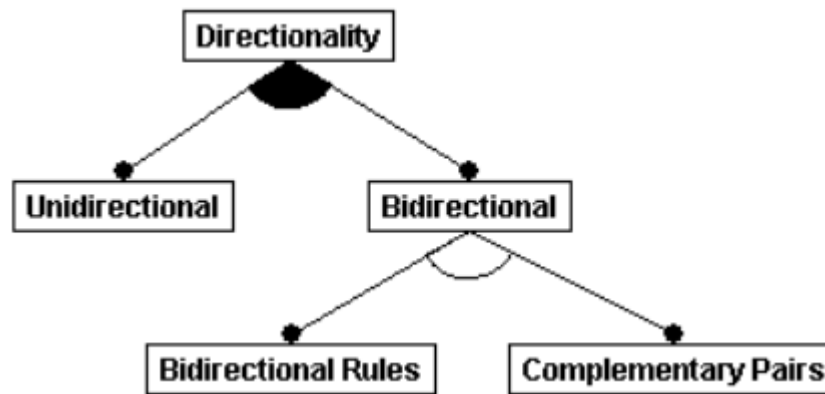


# Model Transformation Taxonomy





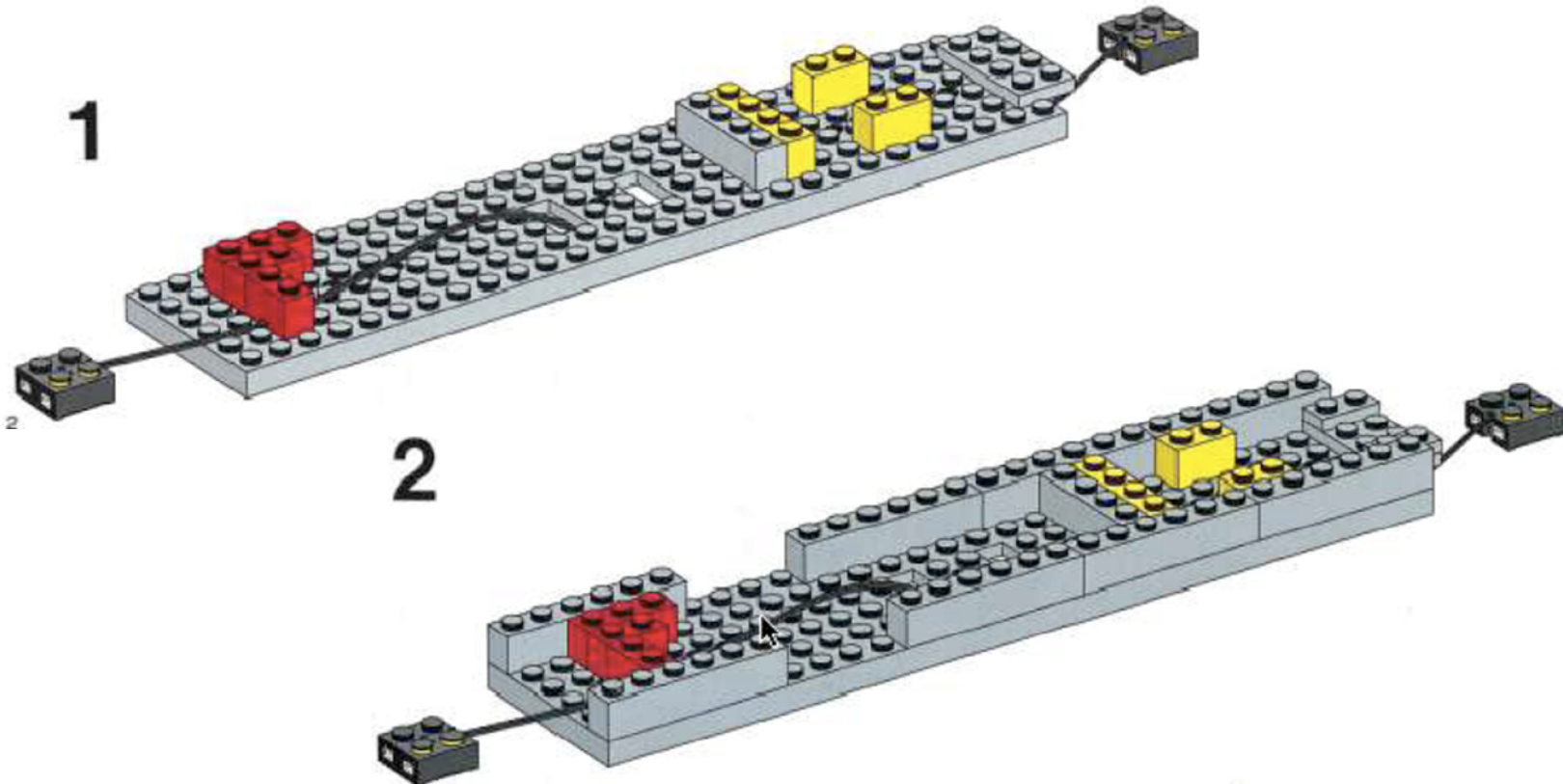
# Model Transformation Taxonomy



### ***5.3. Model-to-model transformation – graph transformations***

# Describe Structural Changes

- Most children understand this way of describing structural changes:



# Graph Transformations

---

- Idea: View the model as a graph
  - objects are nodes
  - links are edges

# Graph Transformations

---

- Idea: View the model as a graph
  - objects are nodes
  - links are edges
- Describe the transformation by rules that describe how and when a particular part of the graph can be modified

# Graph Transformations

---

- Idea: View the model as a graph
  - objects are nodes
  - links are edges
- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
  - (similar to the Lego manual)

# Graph Transformations

---

- Idea: View the model as a graph
  - objects are nodes
  - links are edges
- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
  - (similar to the Lego manual)
  - we use **graph grammars**

# Graph Transformations

---

- Idea: View the model as a graph
  - objects are nodes
  - links are edges
- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
  - (similar to the Lego manual)
  - we use **graph grammars**
  - also called **graph transformation rules**



# View the System as a Graph

---

- Idea: View the model as a graph

# View the System as a Graph

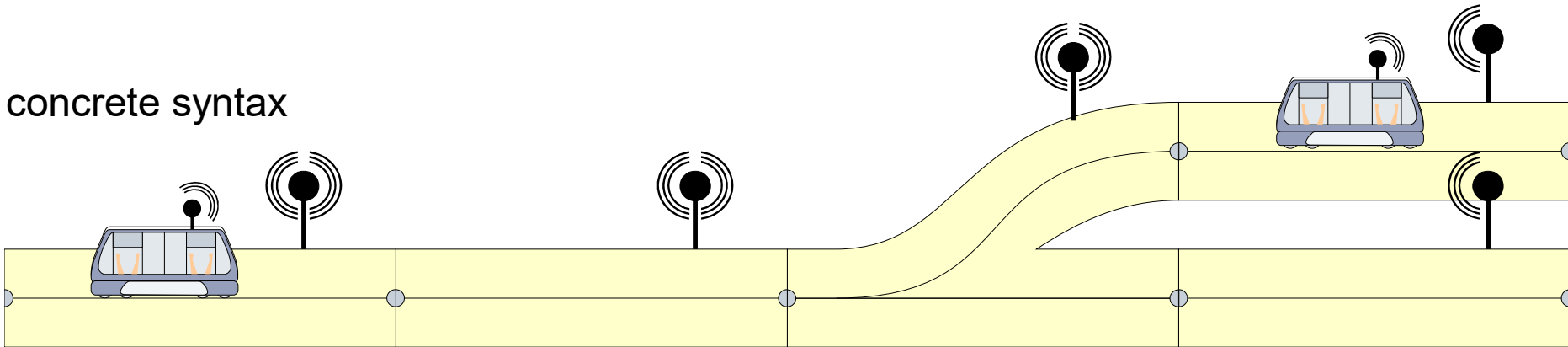
---

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

# View the System as a Graph

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

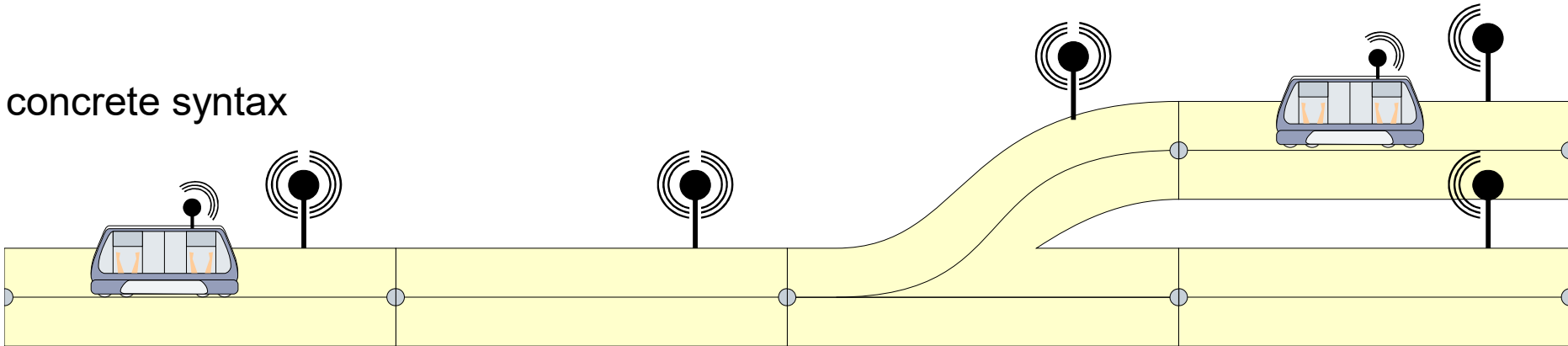
concrete syntax



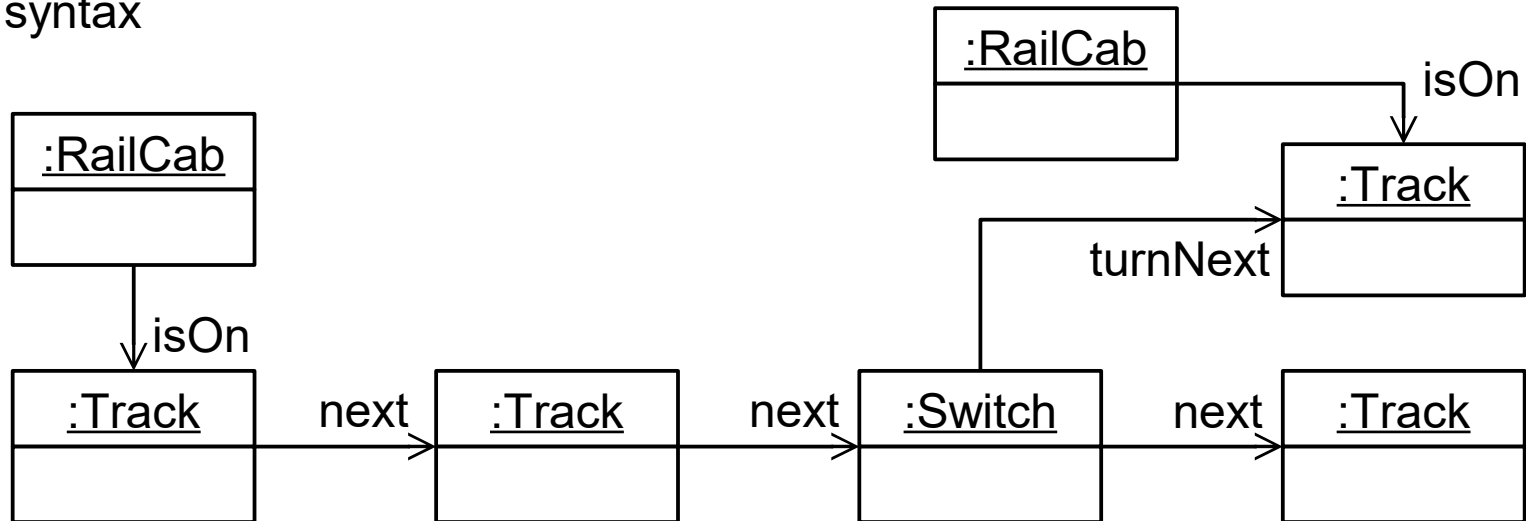
# View the System as a Graph

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

concrete syntax



abstract syntax



# Graph Reconfiguration Behavior

---

- Describe the transformation by rules that describe how and when a particular part of the graph can be modified

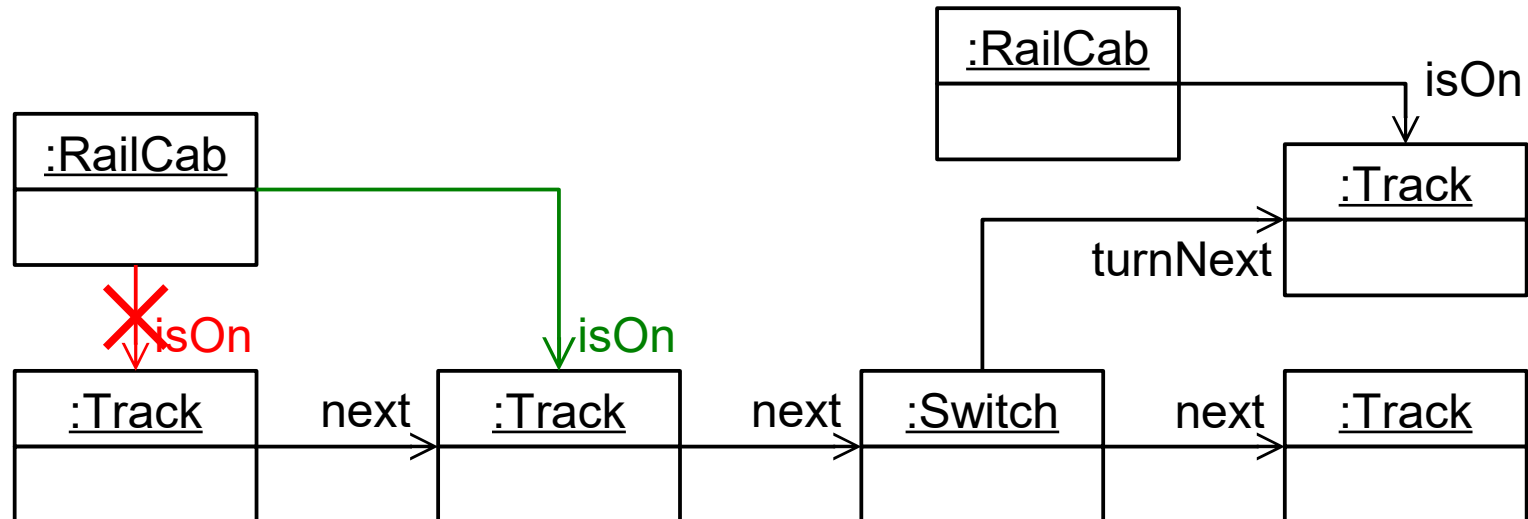
# Graph Reconfiguration Behavior

---

- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
- **Example:** Movement of the RailCab

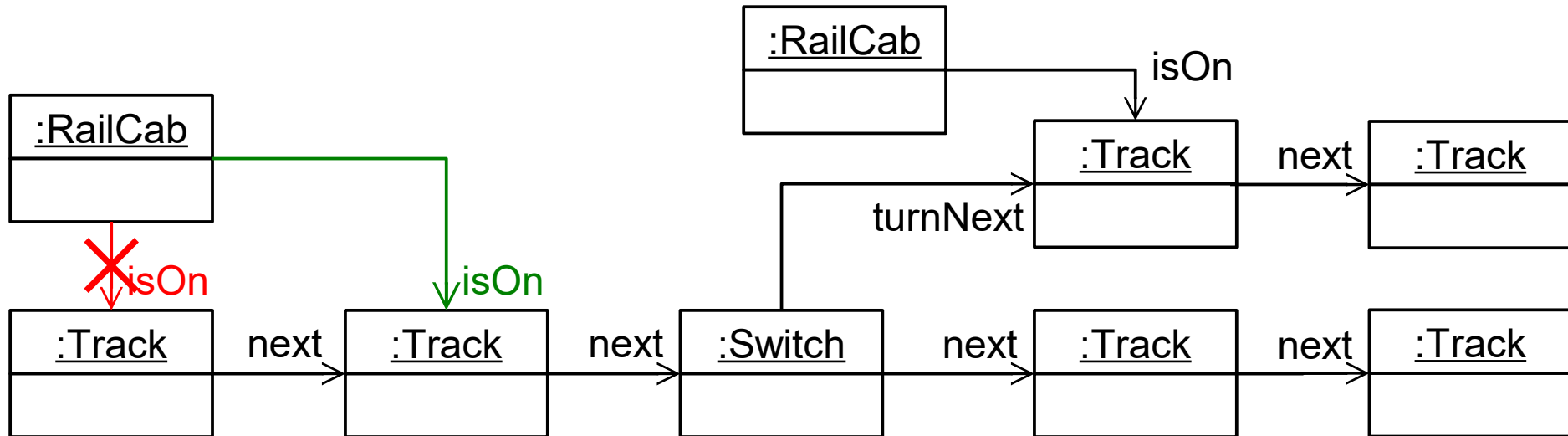
# Graph Reconfiguration Behavior

- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
- Example:** Movement of the RailCab



# Graph Reconfiguration Behavior

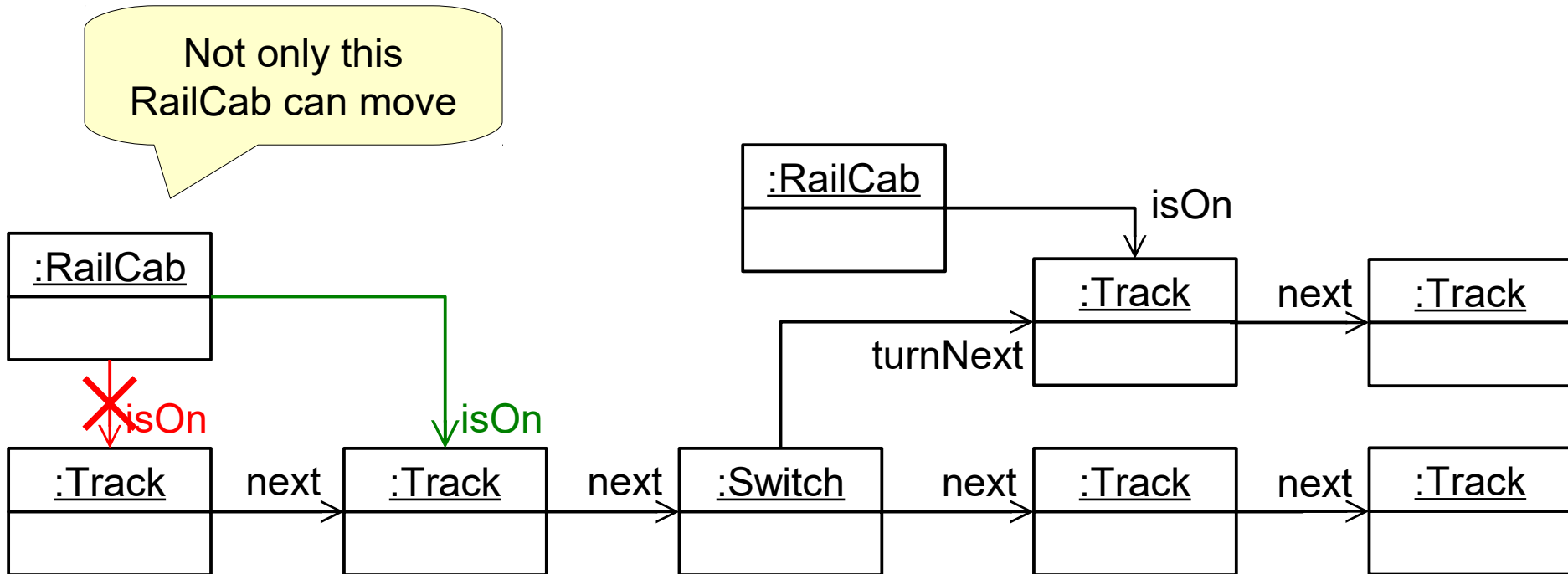
- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
- Example:** Movement of the RailCab





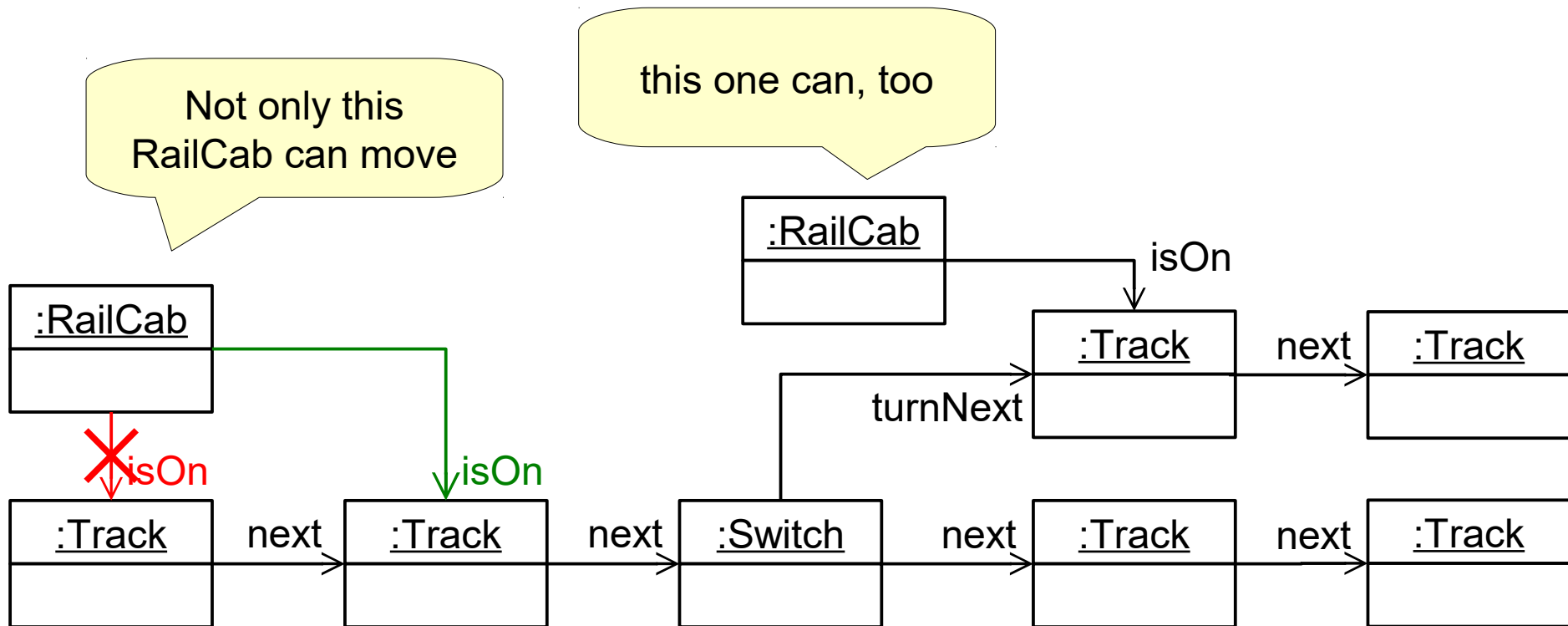
# Graph Reconfiguration Behavior

- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
- Example:** Movement of the RailCab



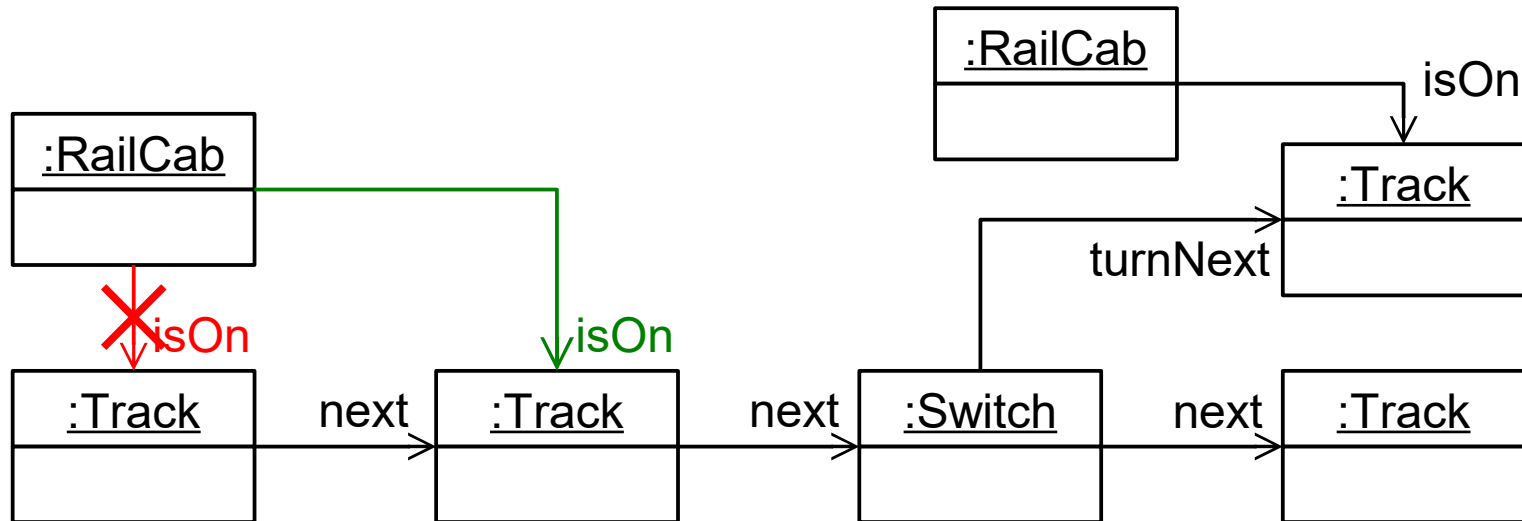
# Graph Reconfiguration Behavior

- Describe the transformation by rules that describe how and when a particular part of the graph can be modified
- Example:** Movement of the RailCab



# Graph Transformation Rule

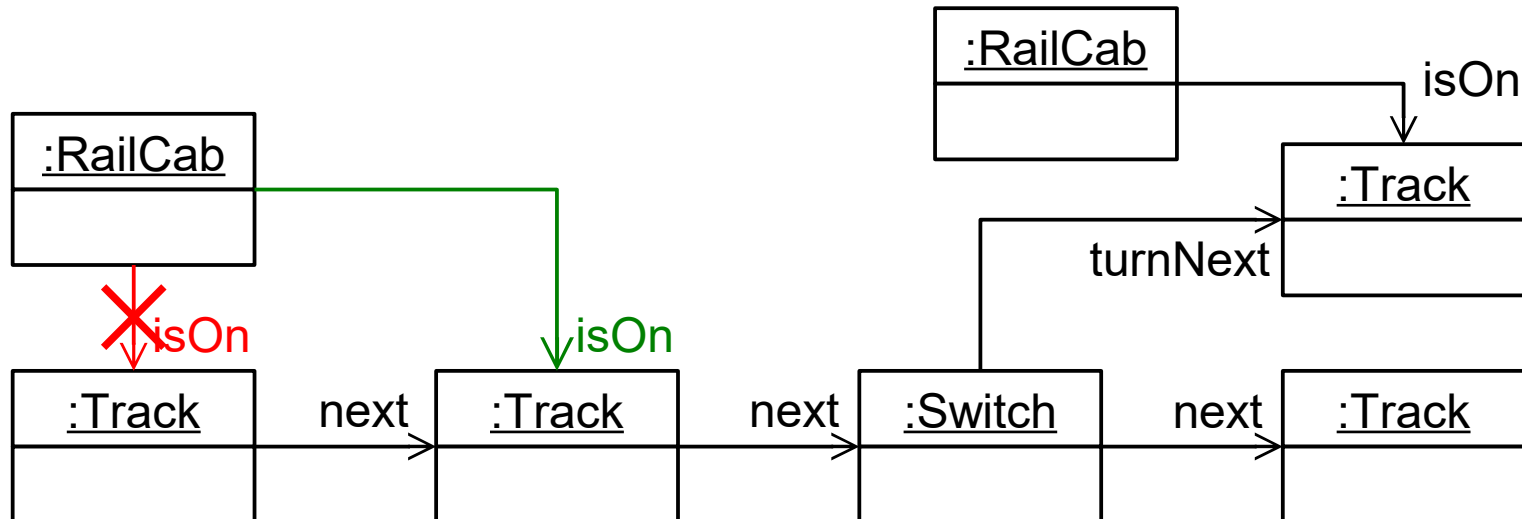
- Describe the necessary **context of the change** and the **change itself** in a **graph transformation rule**



# Graph Transformation Rule

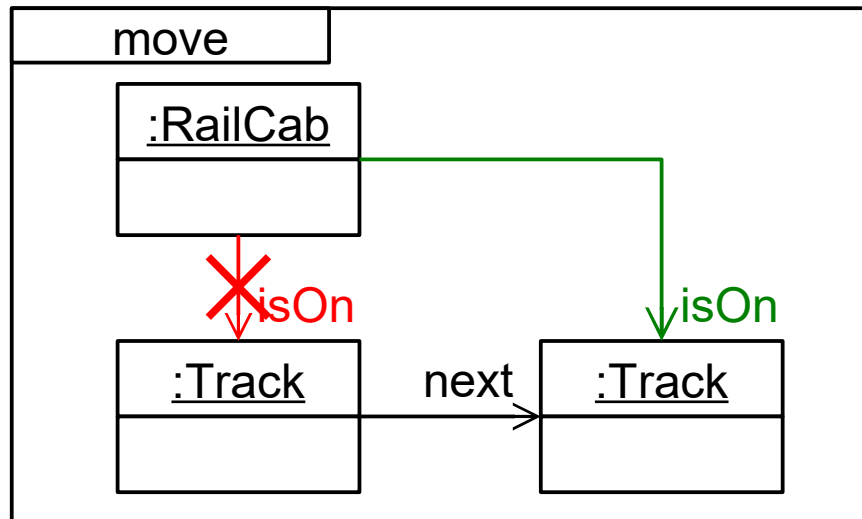
- Describe the necessary **context of the change** and the **change itself** in a **graph transformation rule**

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

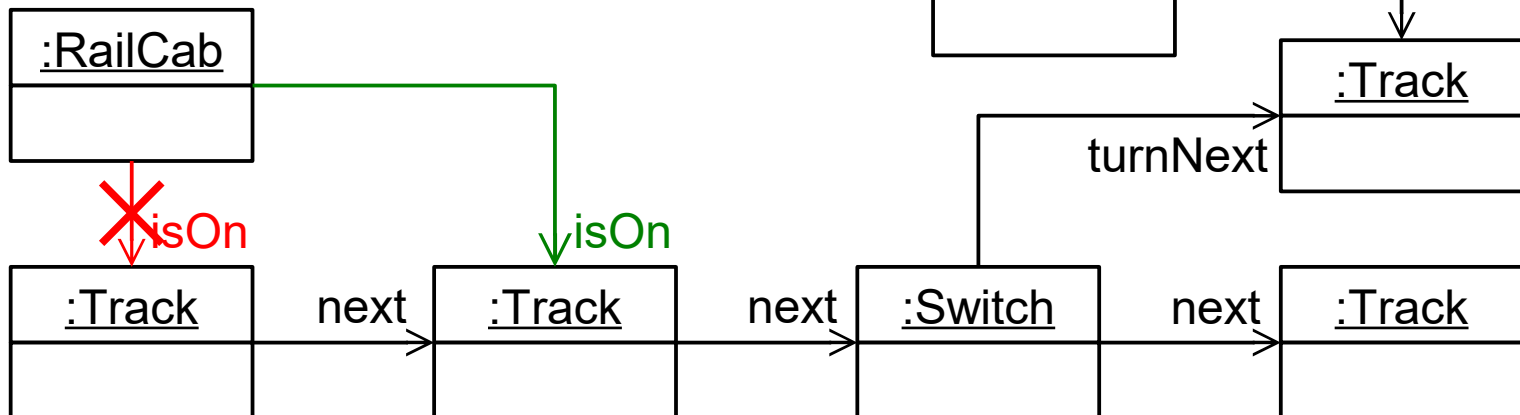


# Graph Transformation Rule

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

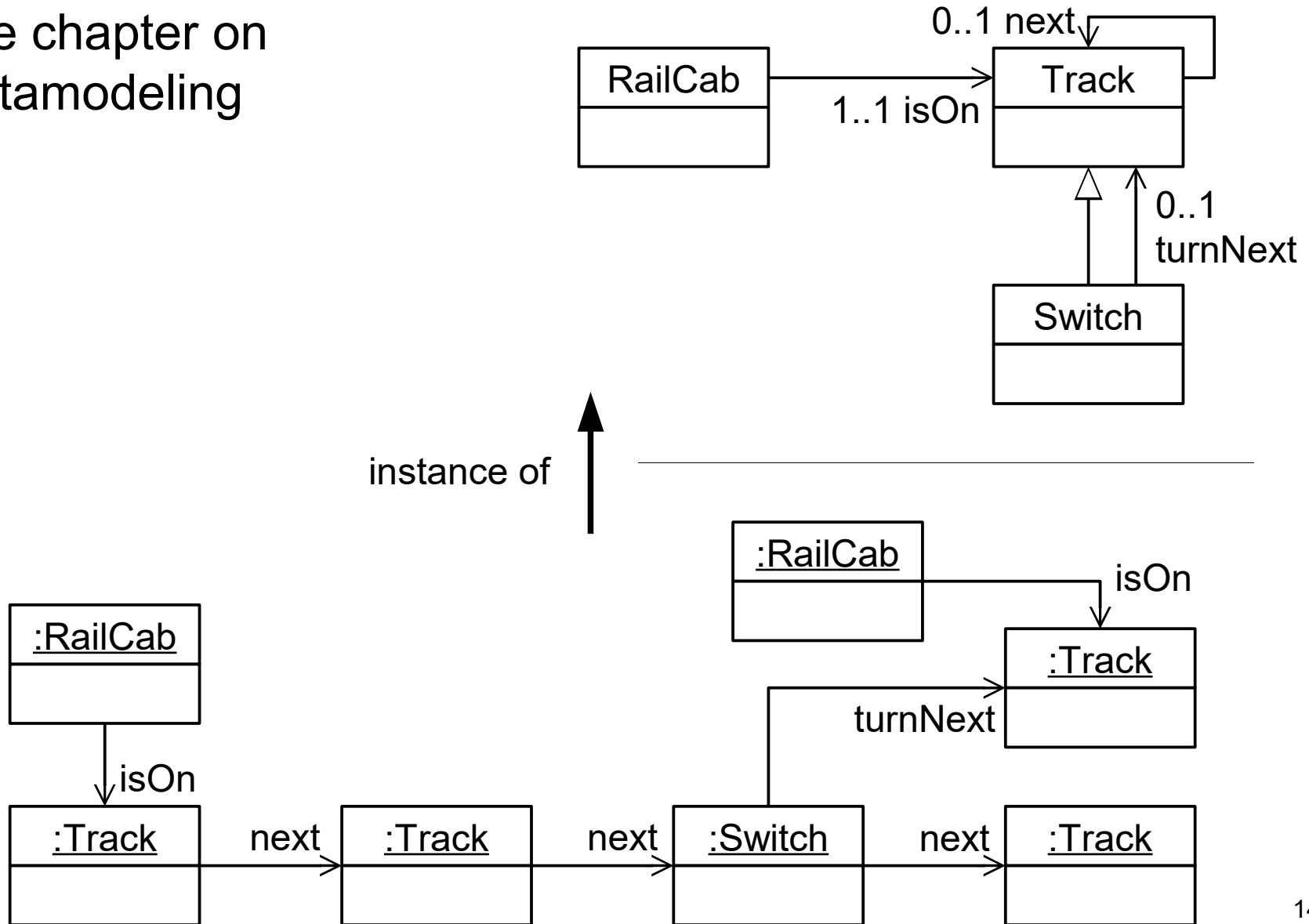


the rule's semantic is clear intuitively, but what does this mean exactly?



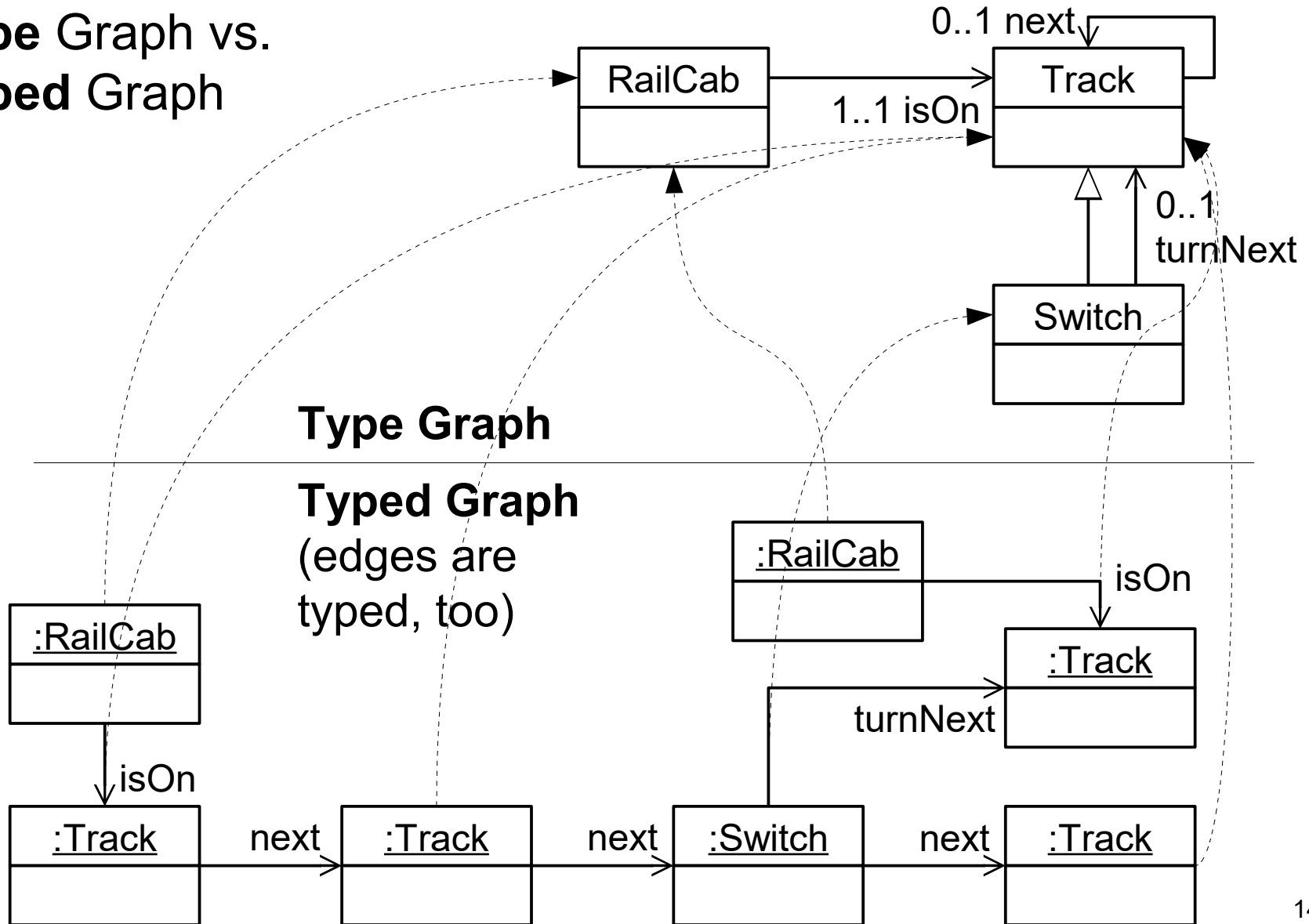
# Model vs. Metamodel

- See chapter on metamodeling



# Models are Typed Graphs

- **Type Graph vs. Typed Graph**



# Graph Grammars

---



# Graph Grammars

---

- A **graph grammar** consists of

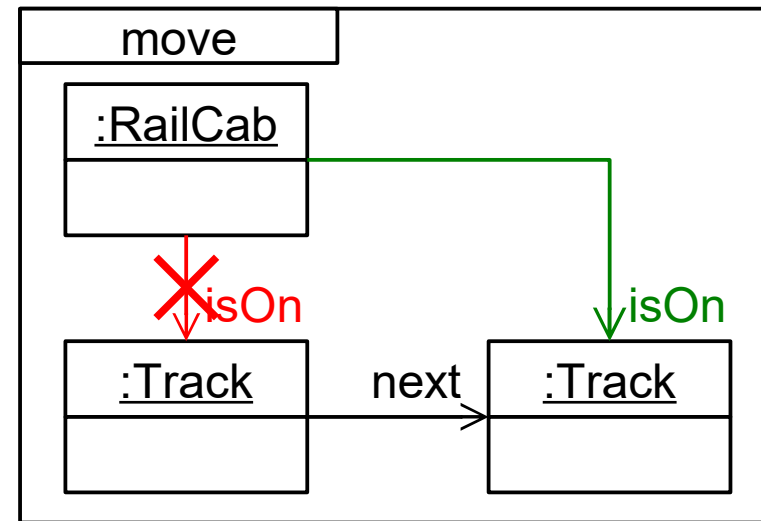
# Graph Grammars

---

- A **graph grammar** consists of
  - a set of **graph grammar rules**

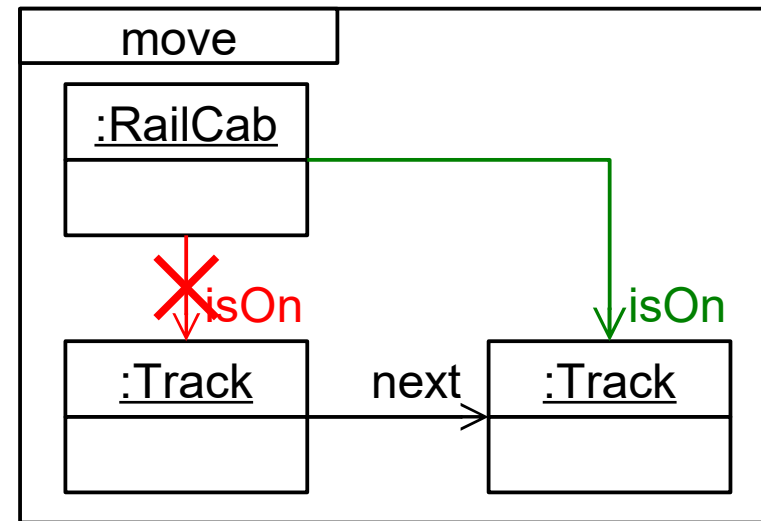
# Graph Grammars

- A **graph grammar** consists of
  - a set of **graph grammar rules**



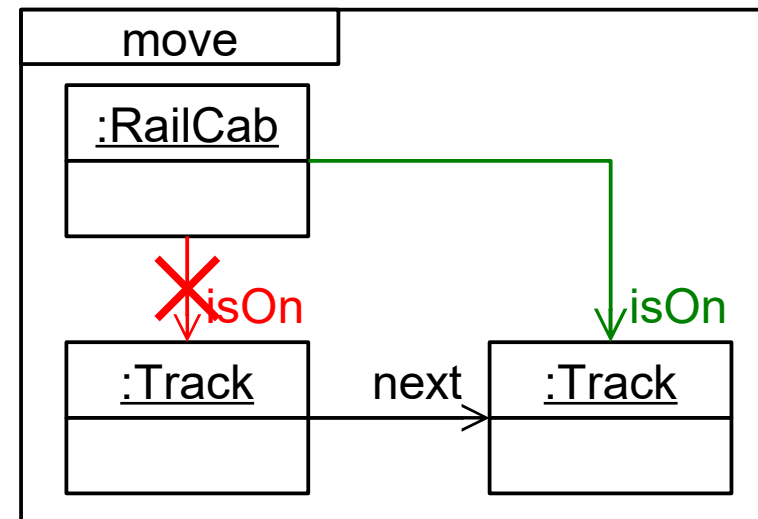
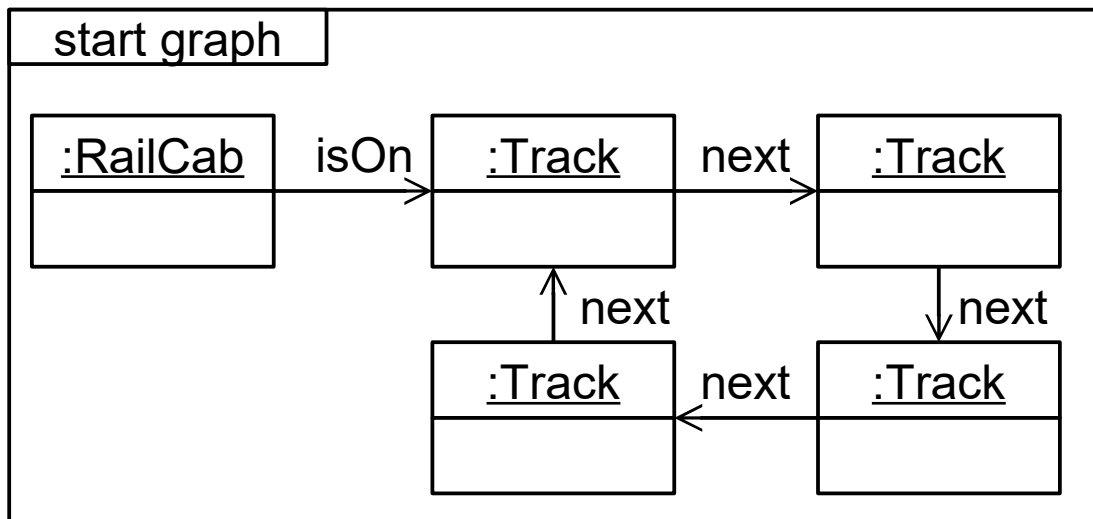
# Graph Grammars

- A **graph grammar** consists of
  - a set of **graph grammar rules**
  - a **start graph** (also called **host graph**)



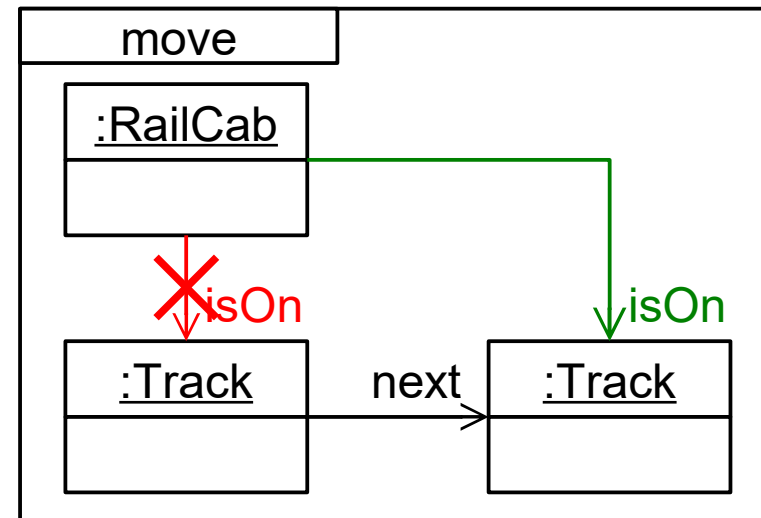
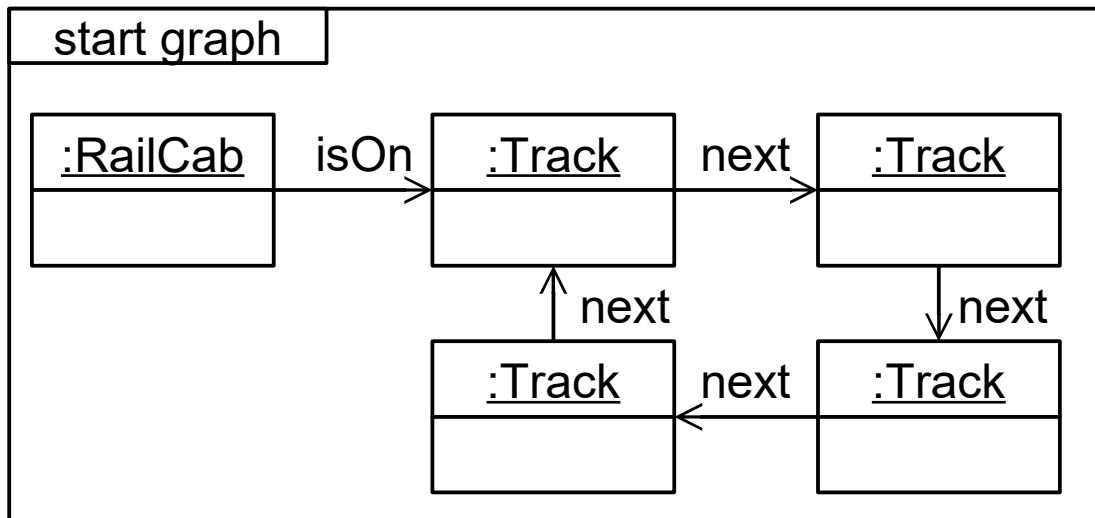
# Graph Grammars

- A **graph grammar** consists of
  - a set of **graph grammar rules**
  - a **start graph** (also called **host graph**)



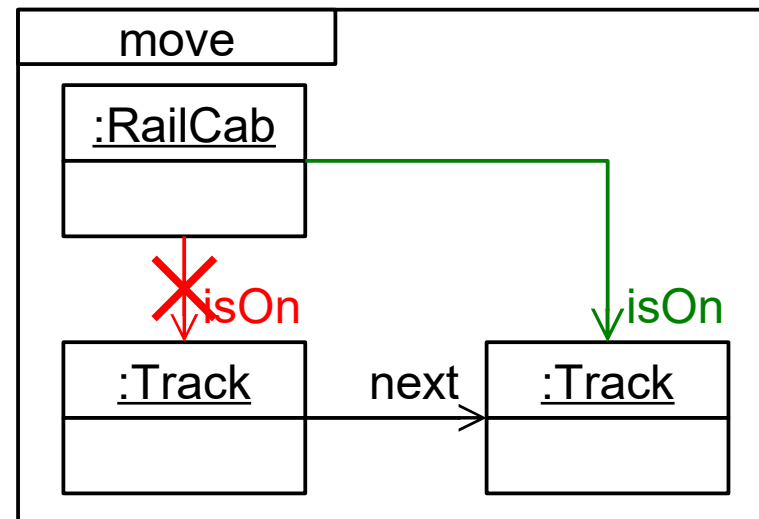
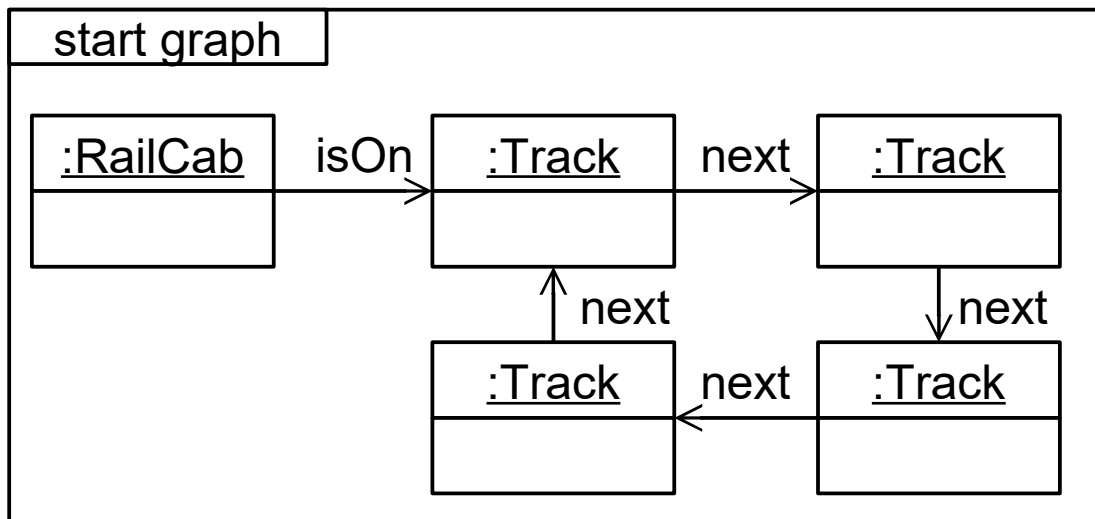
# Graph Grammars

- A **graph grammar** consists of
  - a set of **graph grammar rules**
  - a **start graph** (also called **host graph**)
  - a **type graph**



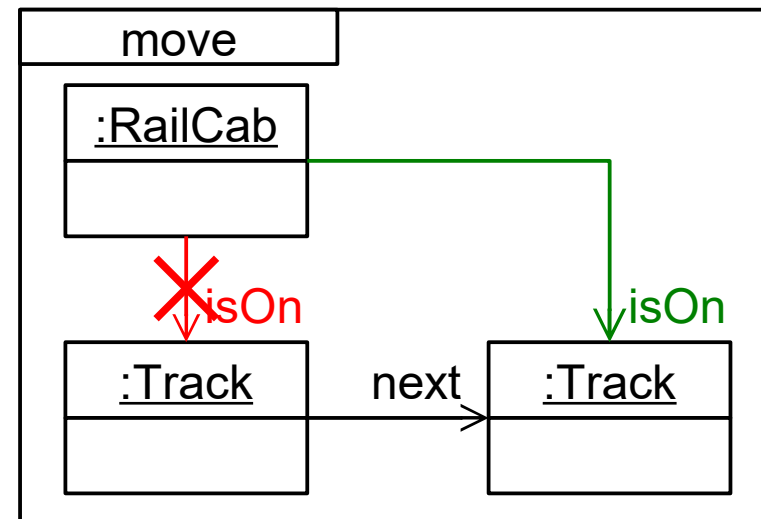
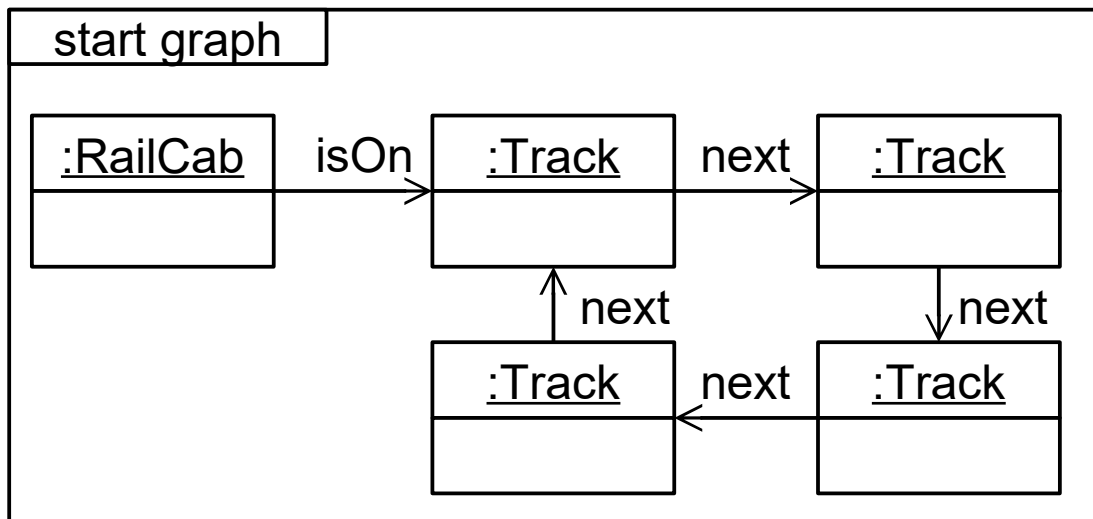
# Graph Grammars

- A **graph grammar** consists of
  - a set of **graph grammar rules**
  - a **start graph** (also called **host graph**)
  - a **type graph**
- A graph grammar describes a (possibly infinite) set of graphs



# Graph Grammars

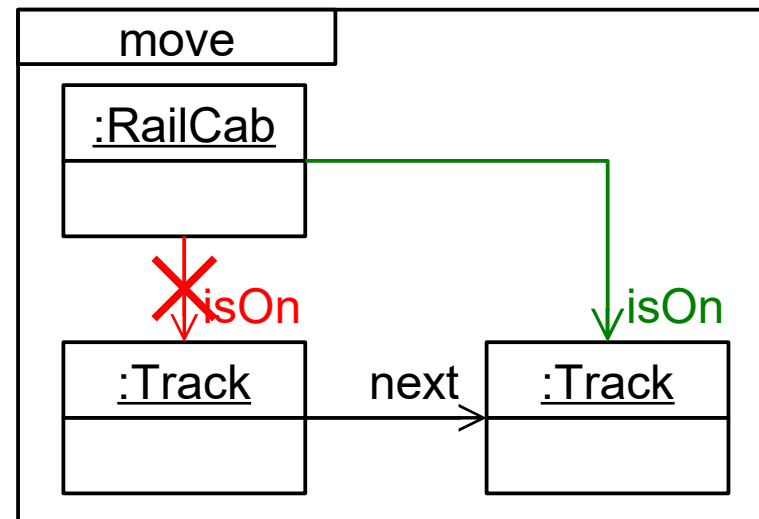
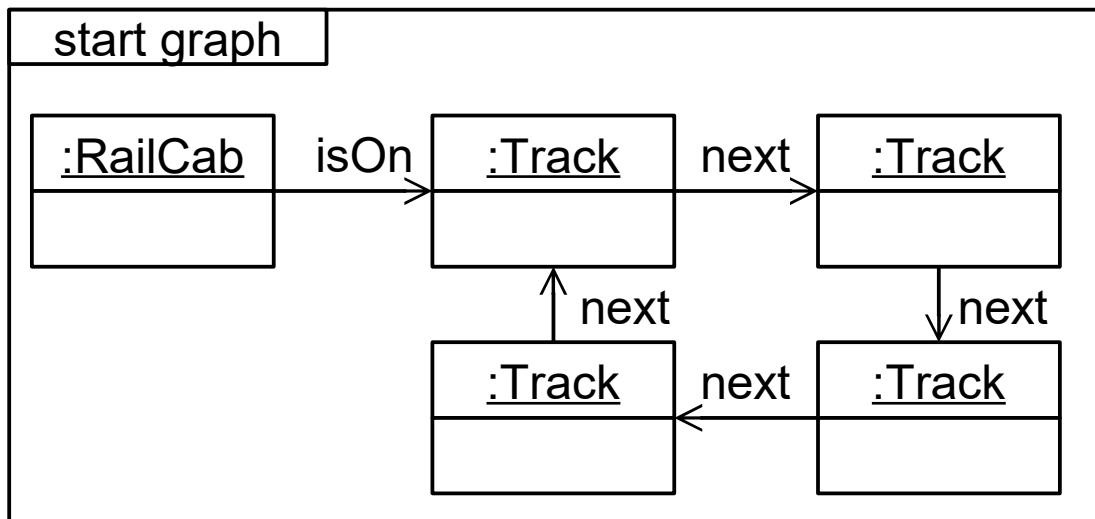
- A **graph grammar** consists of
  - a set of **graph grammar rules**
  - a **start graph** (also called **host graph**)
  - a **type graph**
- A graph grammar describes a (possibly infinite) set of graphs
  - those that can be constructed from the start graph by applying the graph grammar rules in all possible orders





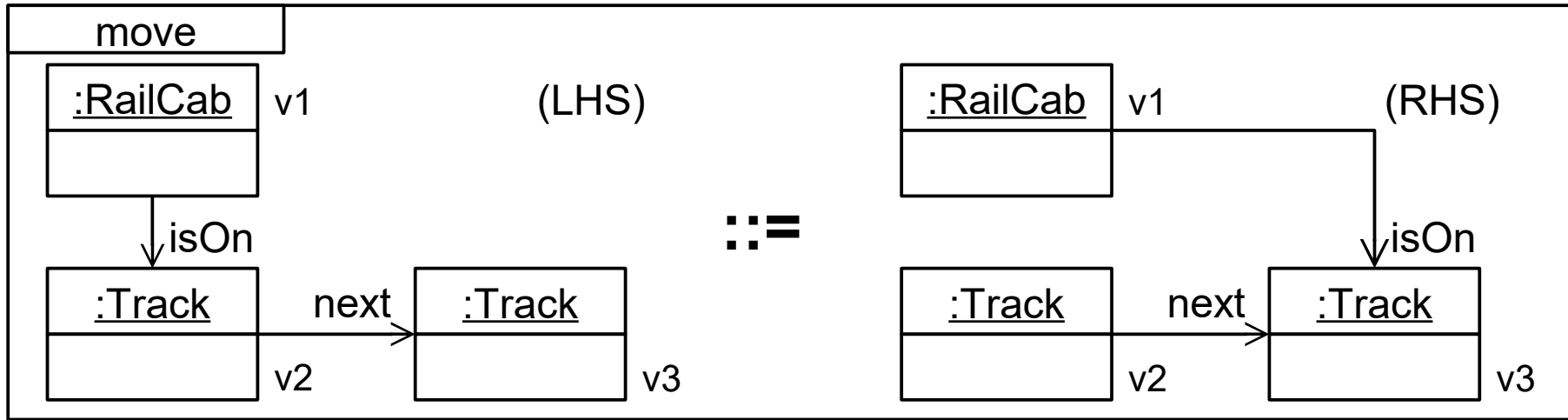
# Graph Grammars

- A **graph grammar** consists of
  - a set of **graph grammar rules**
  - a **start graph** (also called **host graph**)
  - a **type graph**
- A graph grammar describes a (possibly infinite) set of graphs
  - those that can be constructed from the start graph by applying the graph grammar rules in all possible orders
- Graph grammars are also called **Graph Transformation Systems**



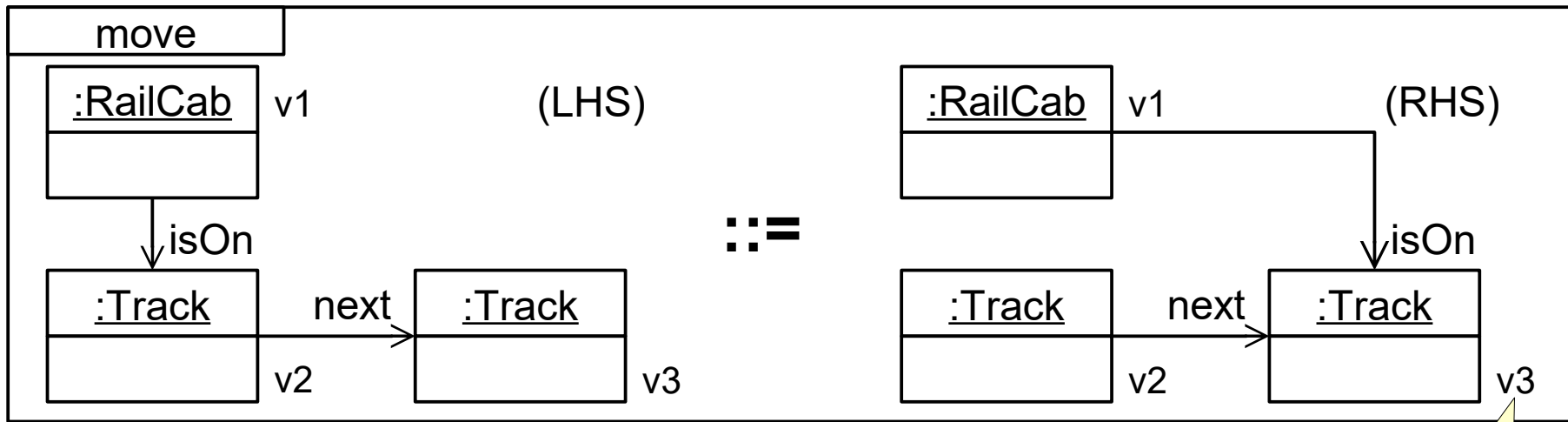
# Graph Grammar Rule

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



# Graph Grammar Rule

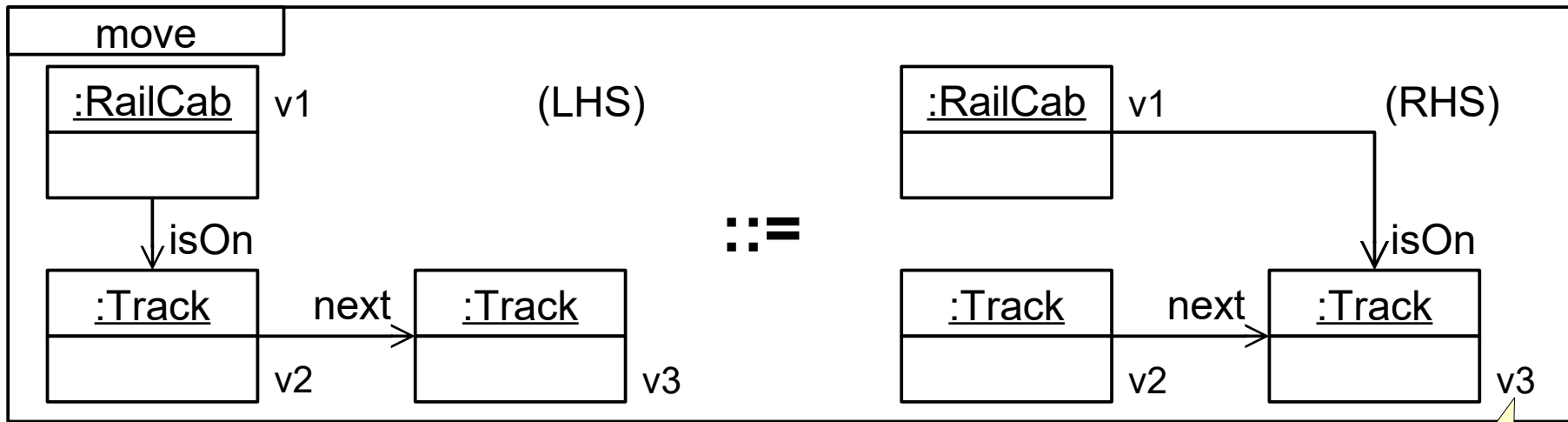
- A graph grammar rule consists of two typed graphs
  - called **left-hand side (LHS)** and **right-hand side (RHS)**



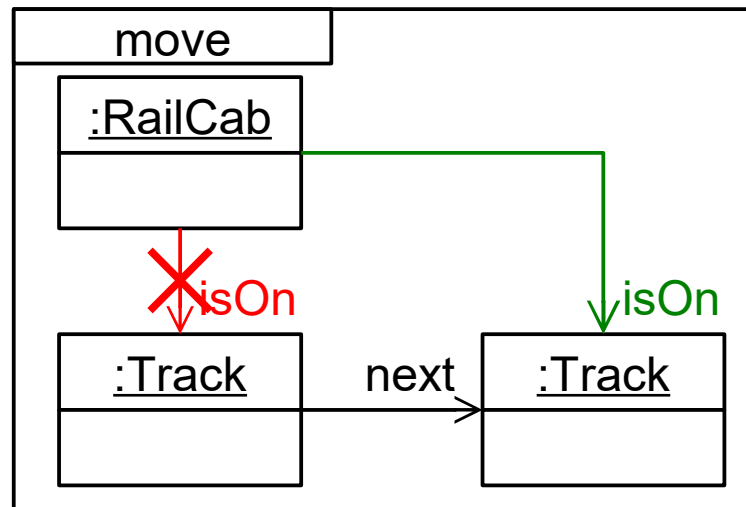
node identities

# Graph Grammar Rule

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



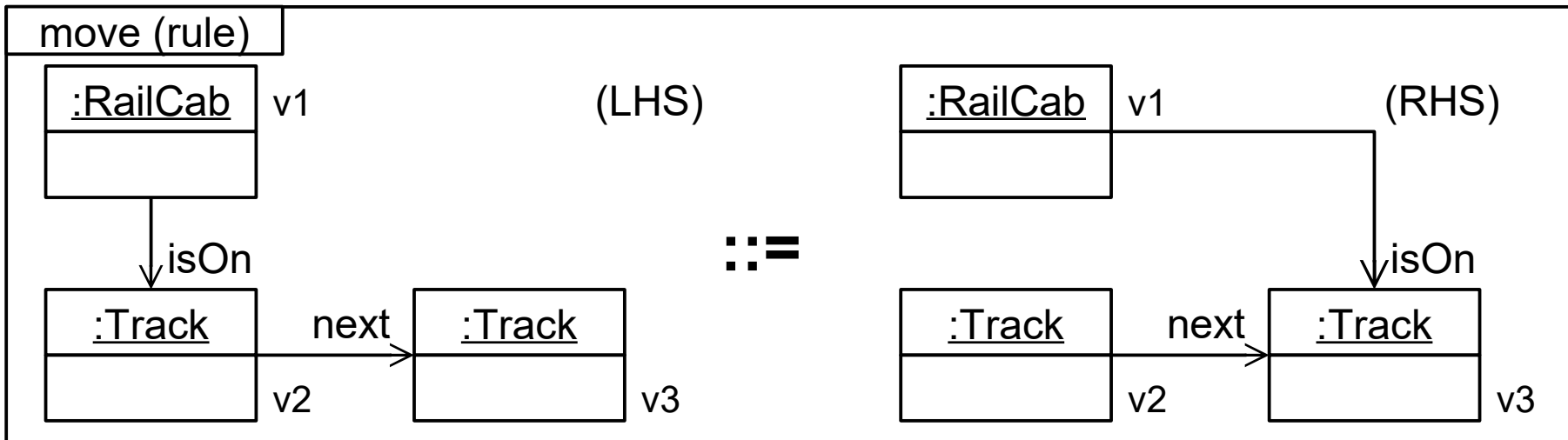
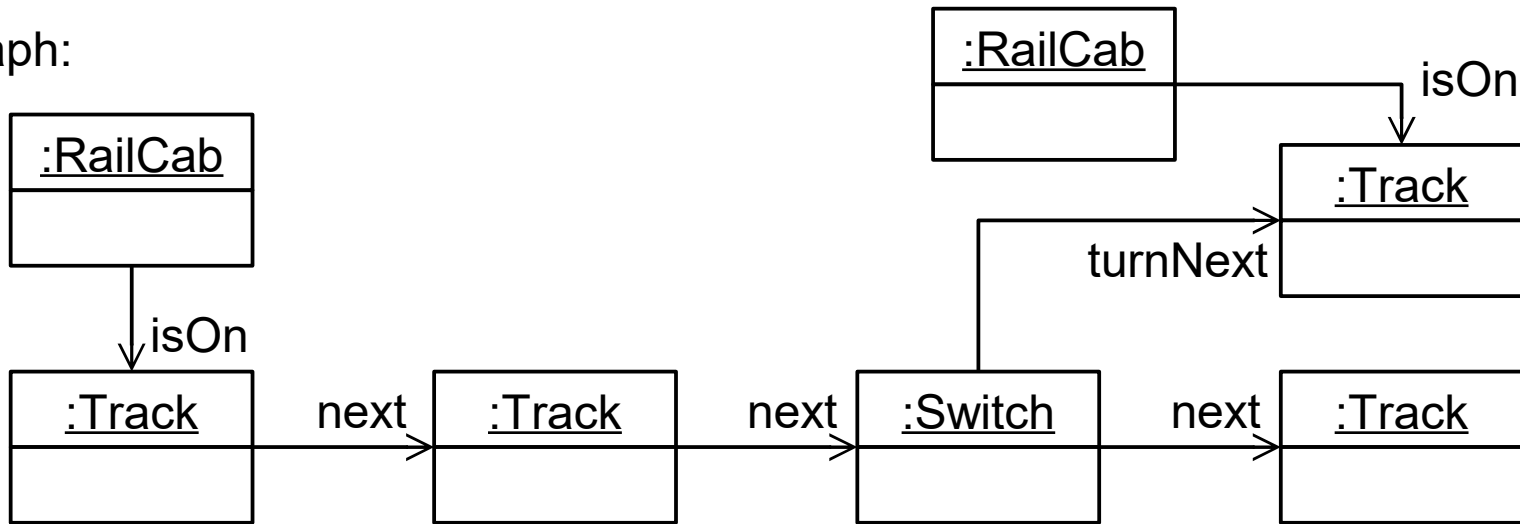
short-hand notation:



node  
identities

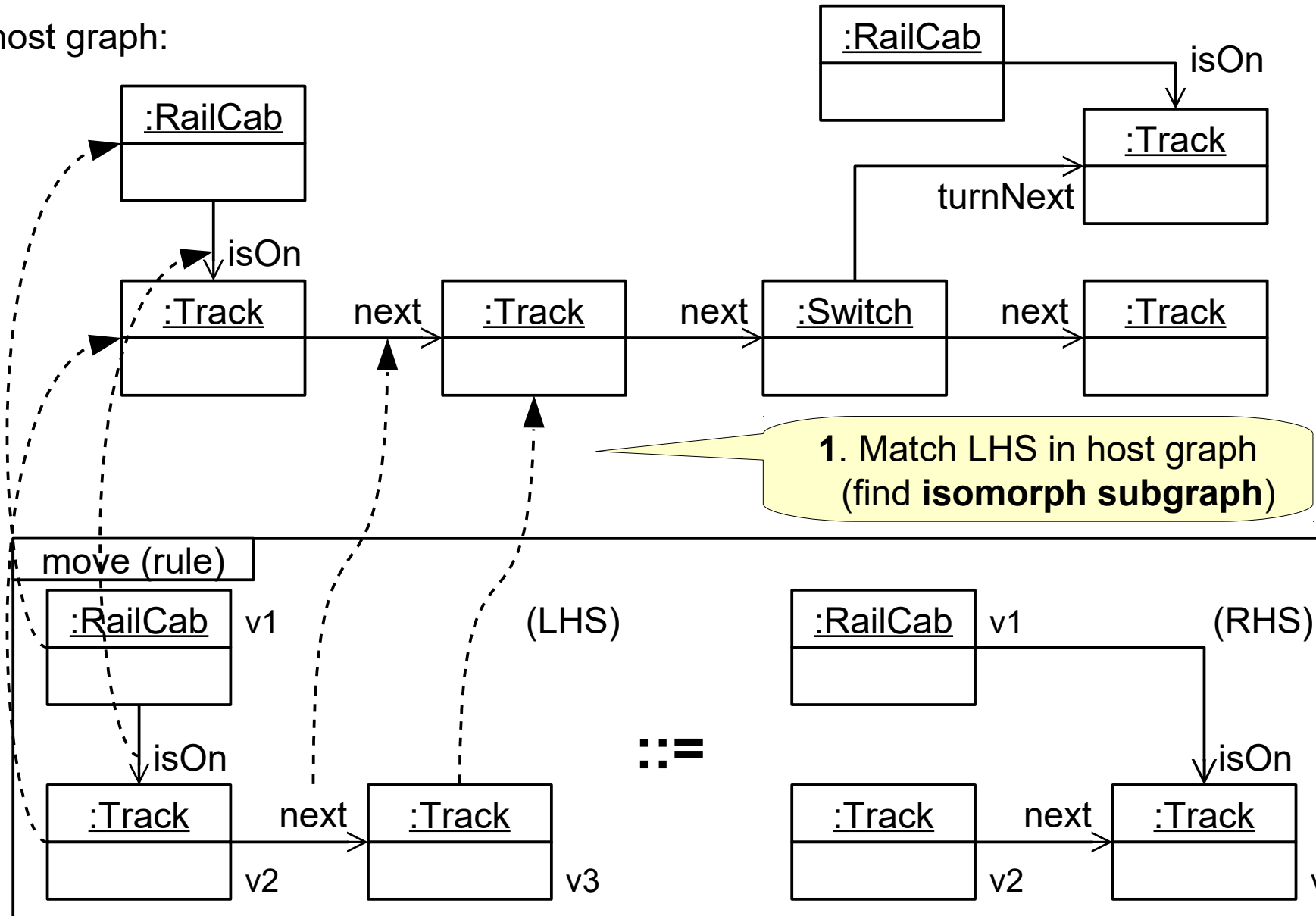
# Graph Grammar Rule Application

host graph:



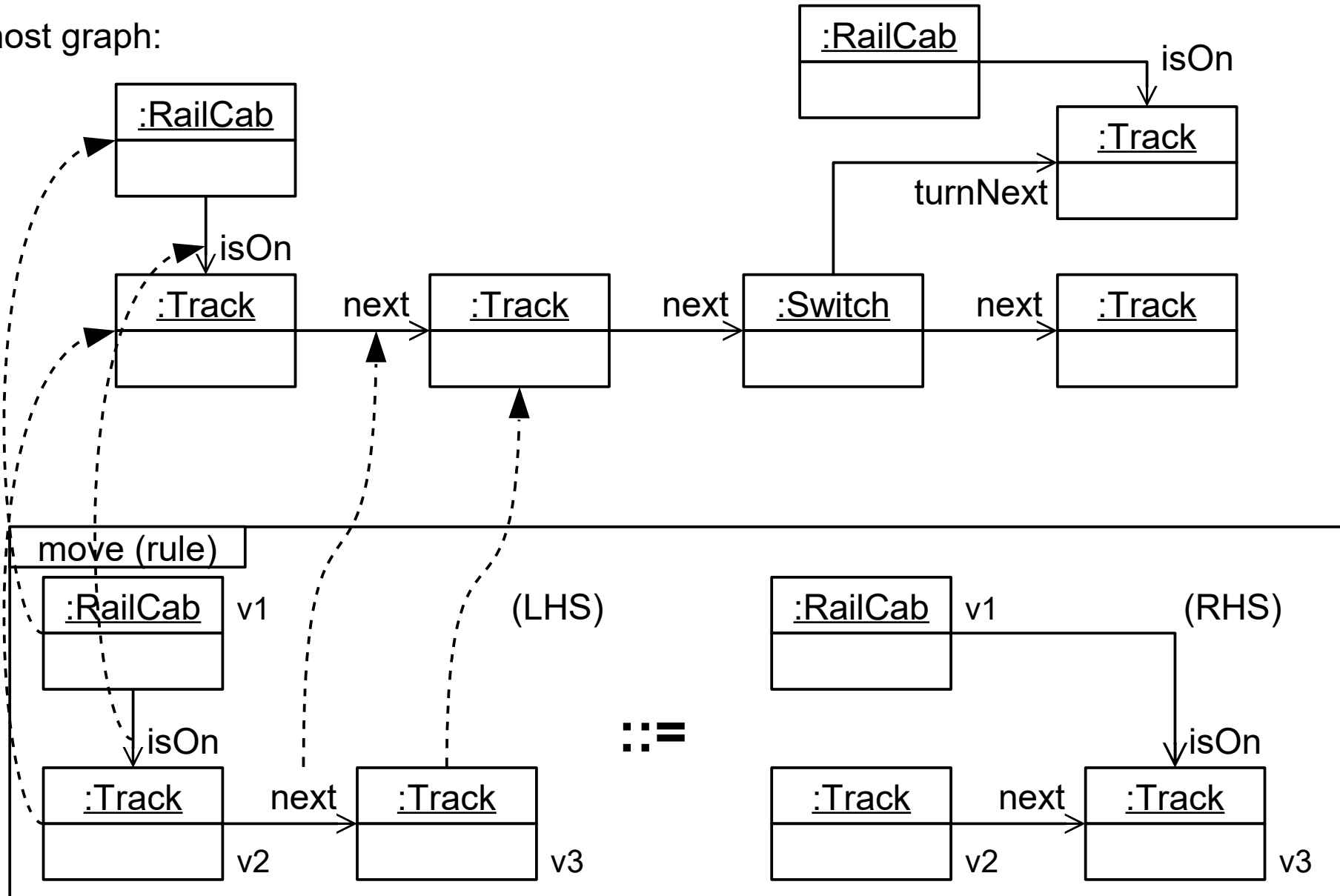
# Graph Grammar Rule Application

host graph:



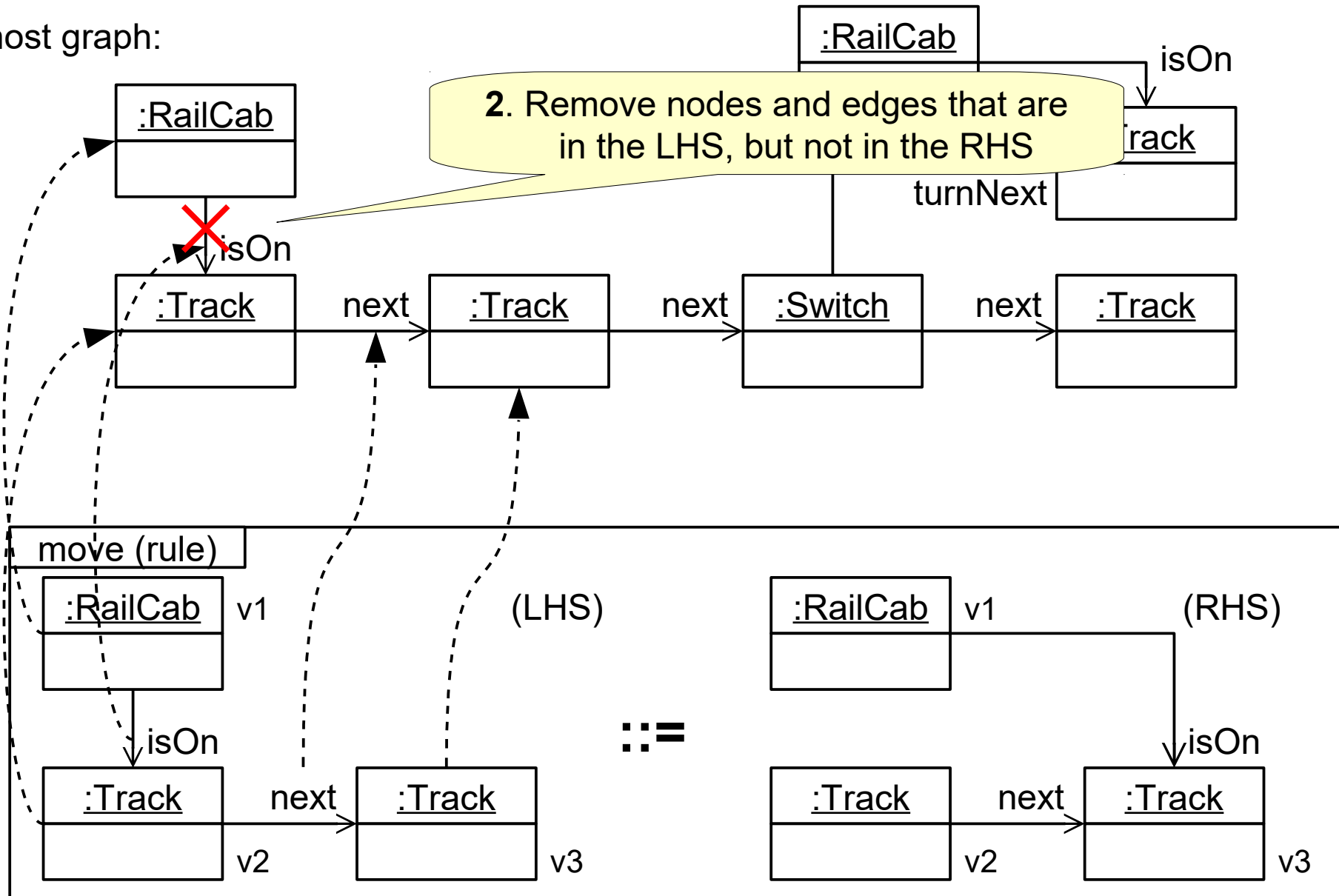
# Graph Grammar Rule Application

host graph:



# Graph Grammar Rule Application

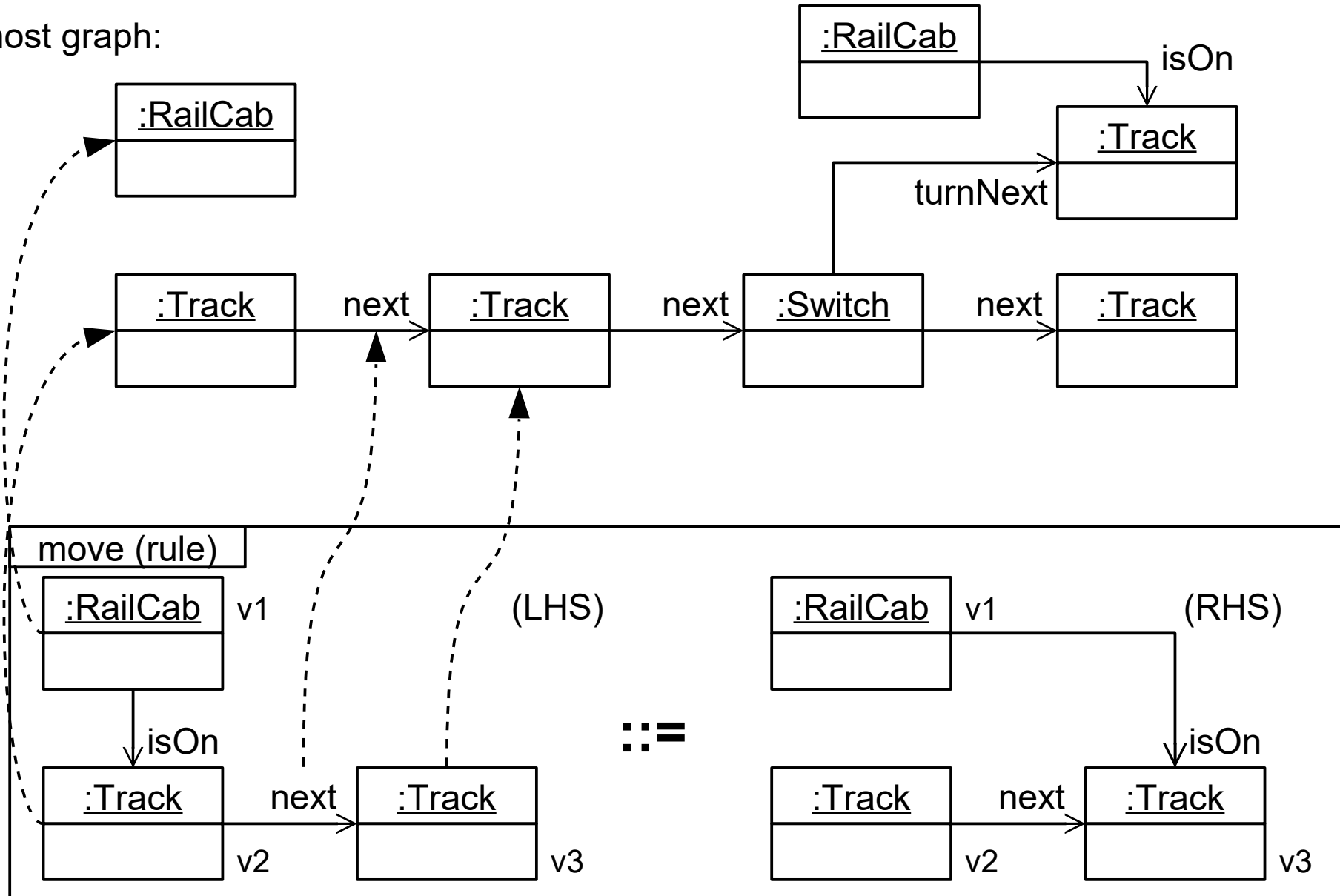
host graph:





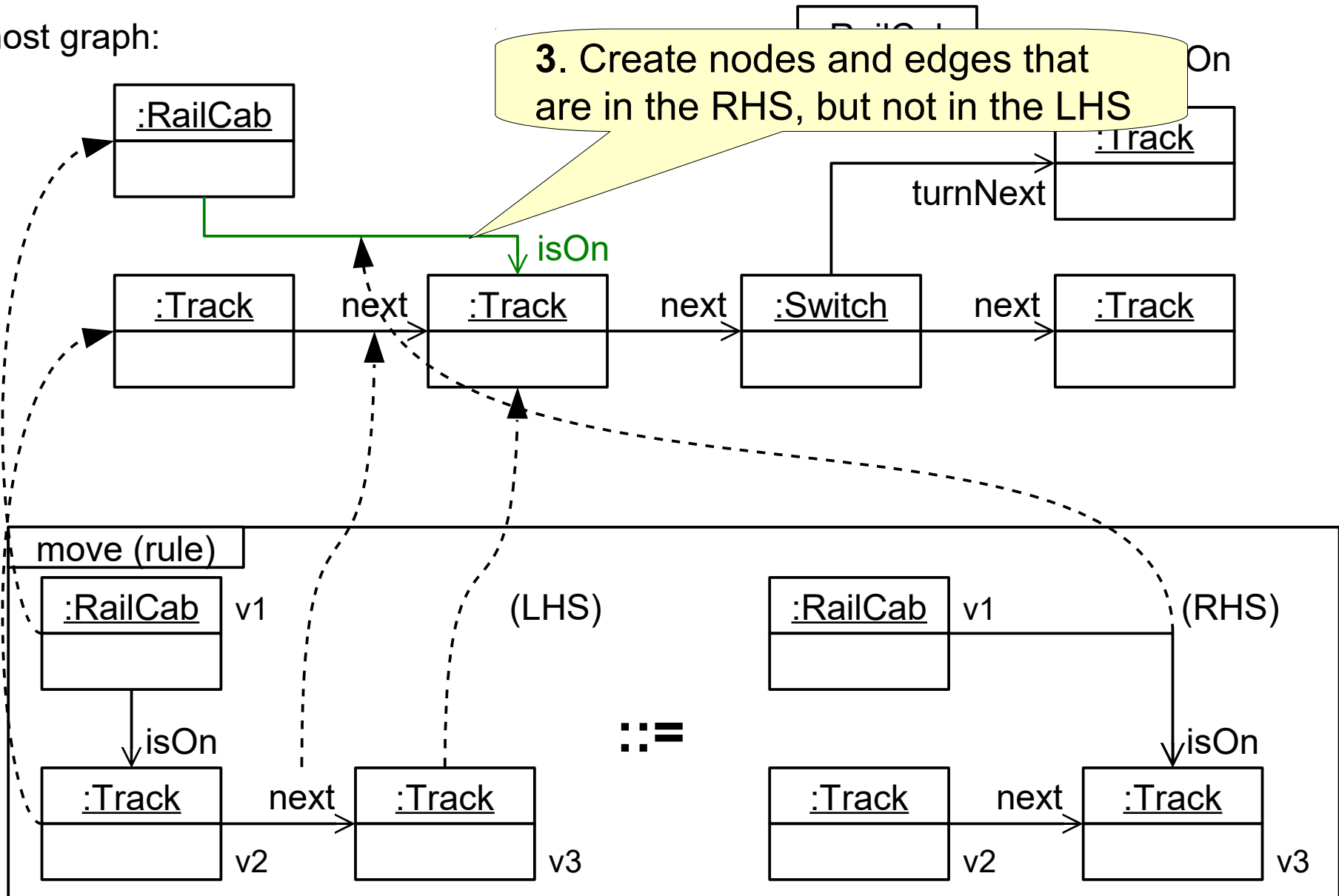
# Graph Grammar Rule Application

host graph:



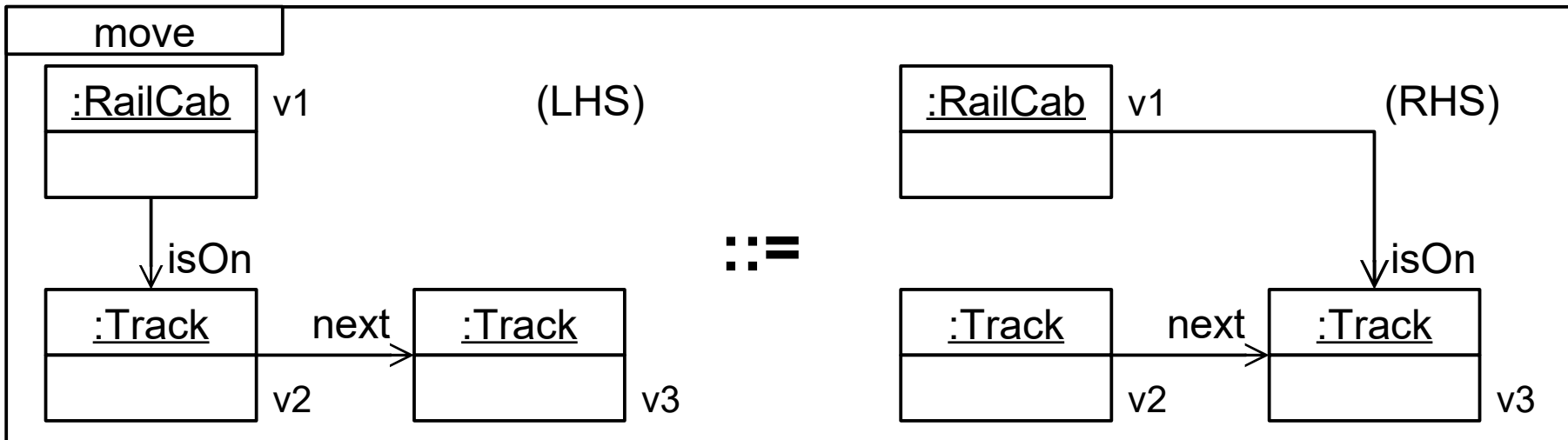
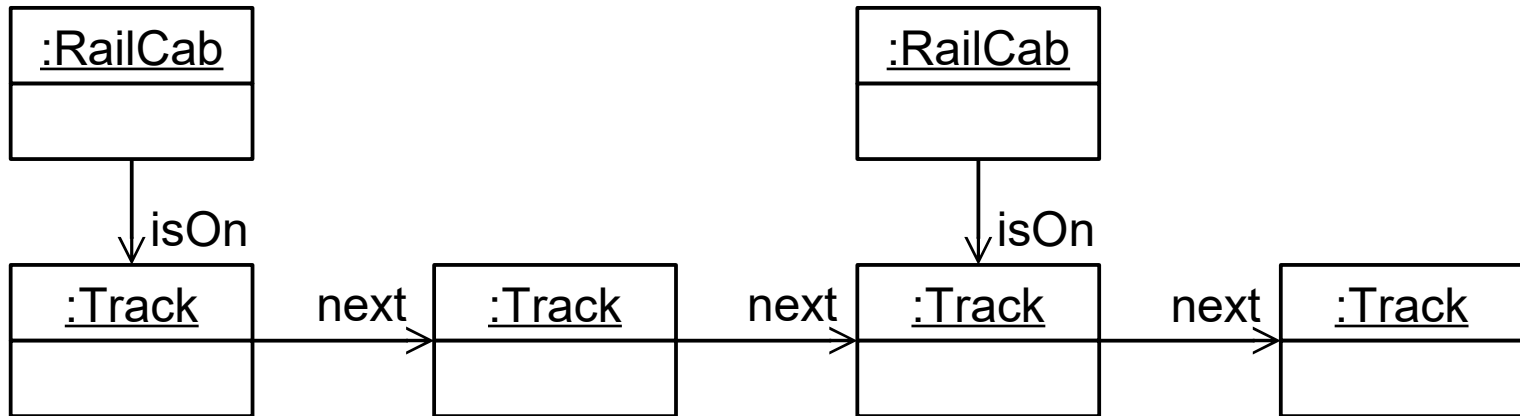
# Graph Grammar Rule Application

host graph:



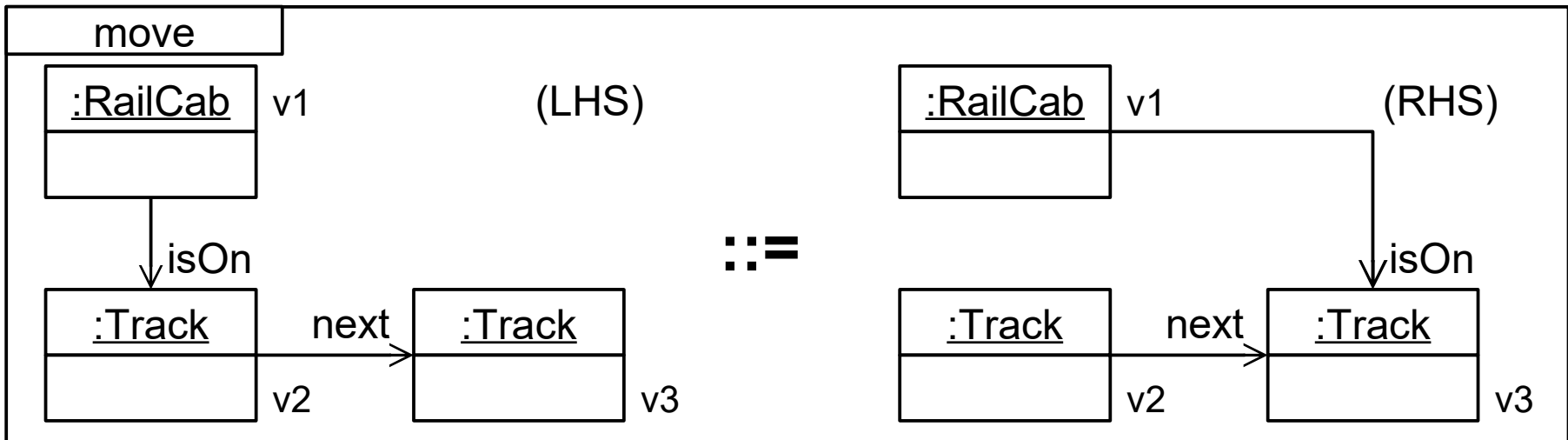
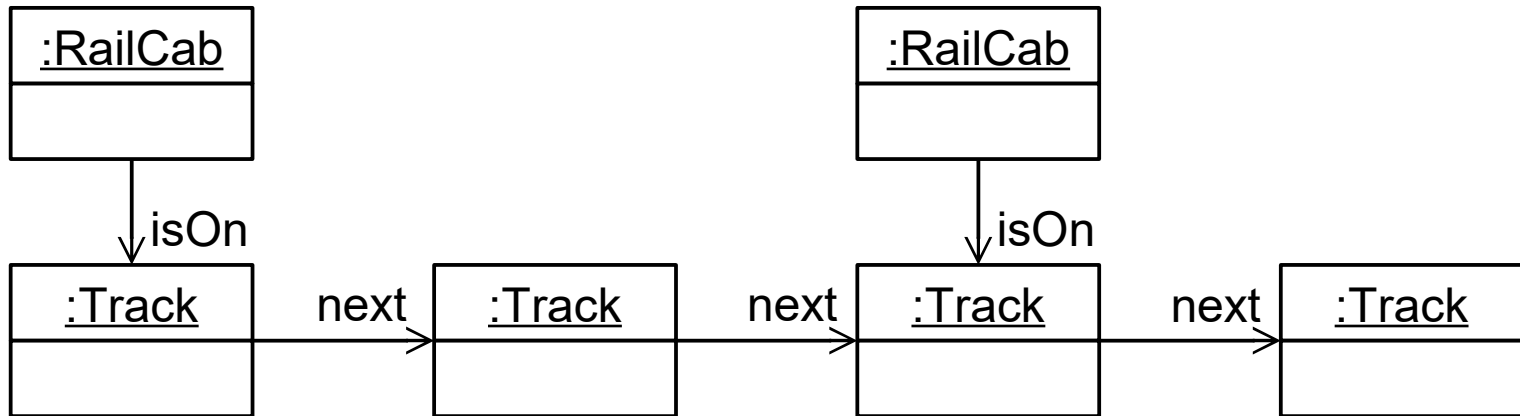
# Graph Grammar Rule Application

- When to move which RailCab?



# Graph Grammar Rule Application

- When to move which RailCab?
  - here we have a non-deterministic choice



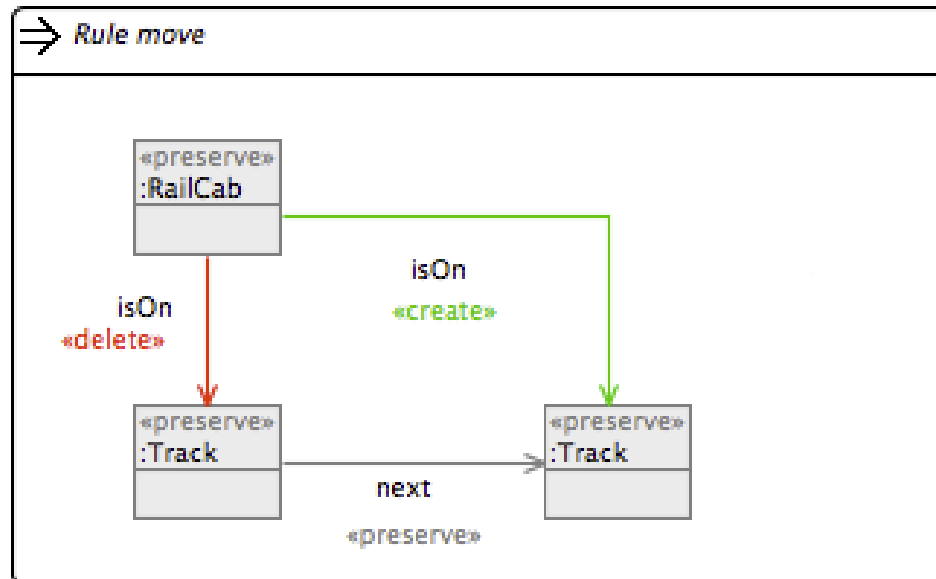
# Eclipse Henshin

---

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

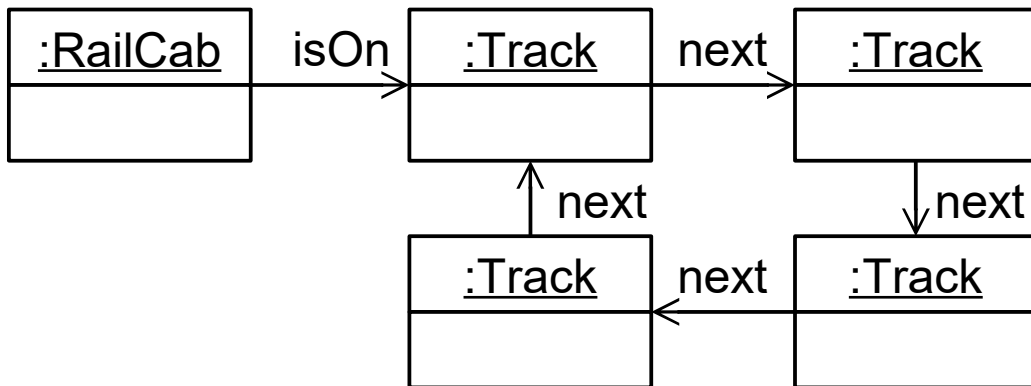


- An Eclipse project that supports the modeling, execution, and analysis of EMF-based graph transformation systems
  - <https://www.eclipse.org/henshin/>

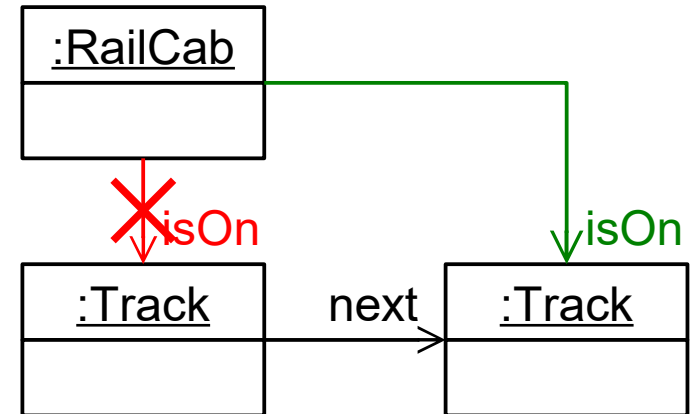


# Exploring the State Space

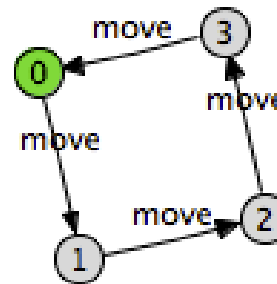
start graph



move

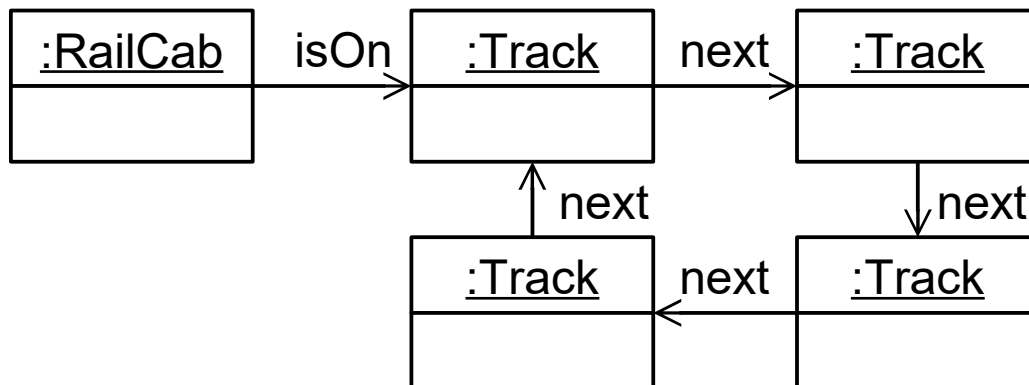


# Exploring the State Space

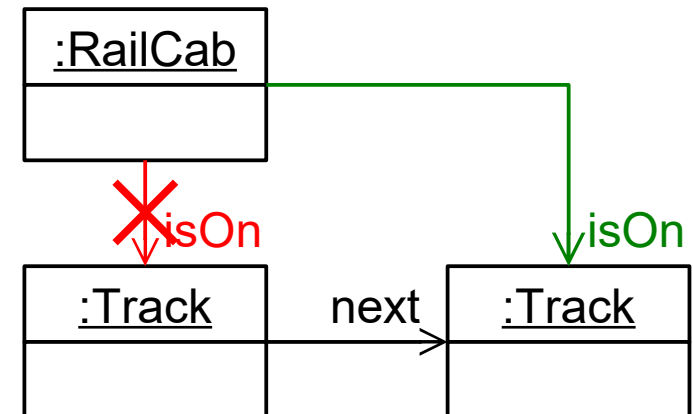


state space explored with Henshin: 4 different graphs; (graph after 4 applications of move rule is isomorphic  $\Rightarrow$  equal to the first)

start graph

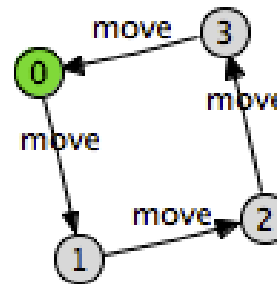


move



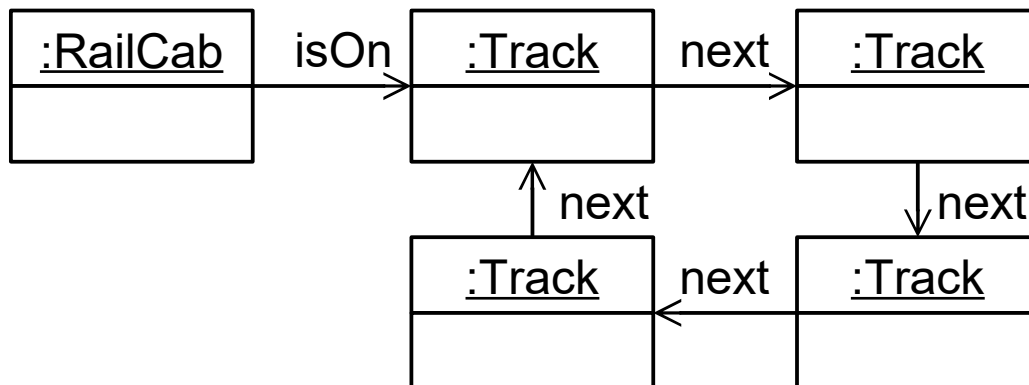


# Exploring the State Space

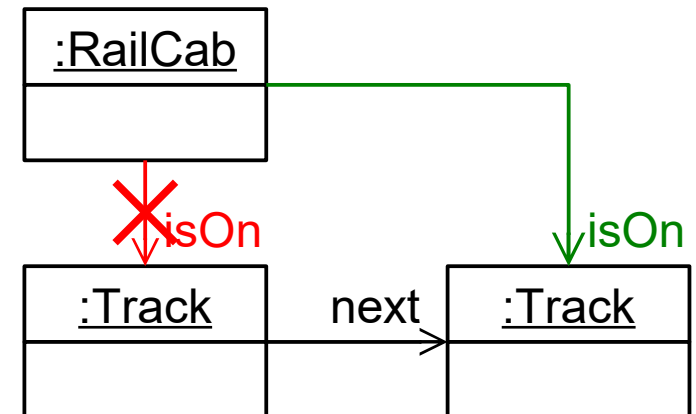


state space explored with Henshin: 4 different graphs; (graph after 4 applications of move rule is isomorphic  $\Rightarrow$  equal to the first)

start graph

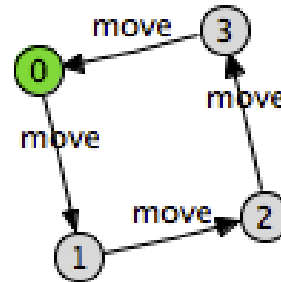


move



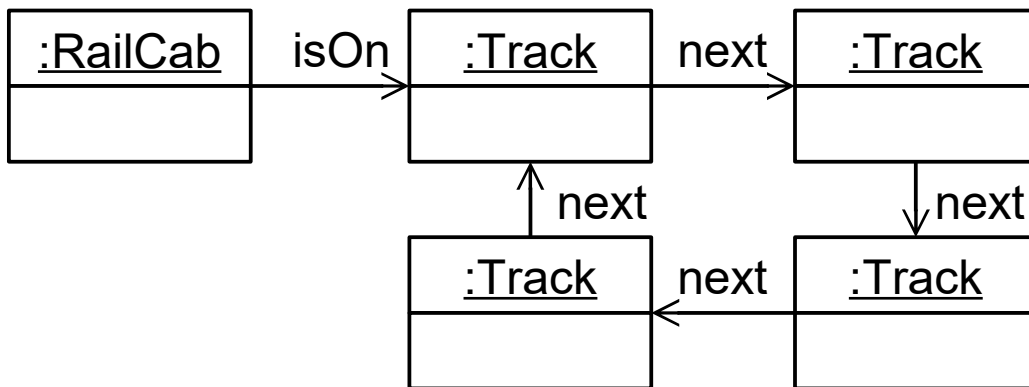
# Exploring the State Space

- A rule application can be considered a transition in a Labeled Transition System

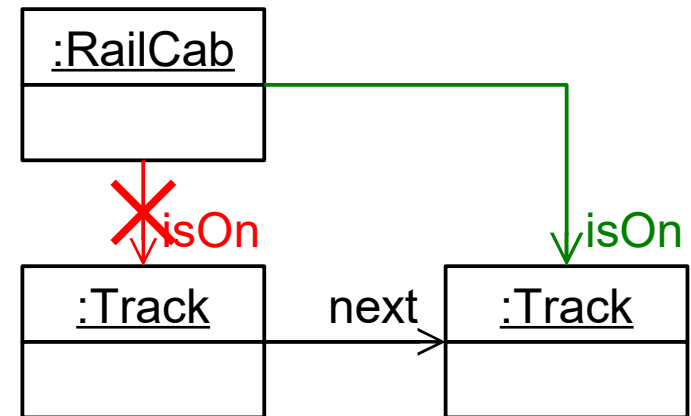


state space explored with Henshin: 4 different graphs; (graph after 4 applications of move rule is isomorphic  $\Rightarrow$  equal to the first)

start graph

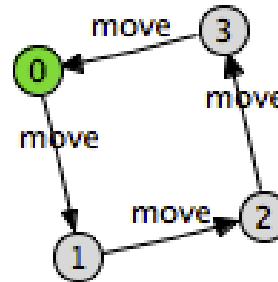


move



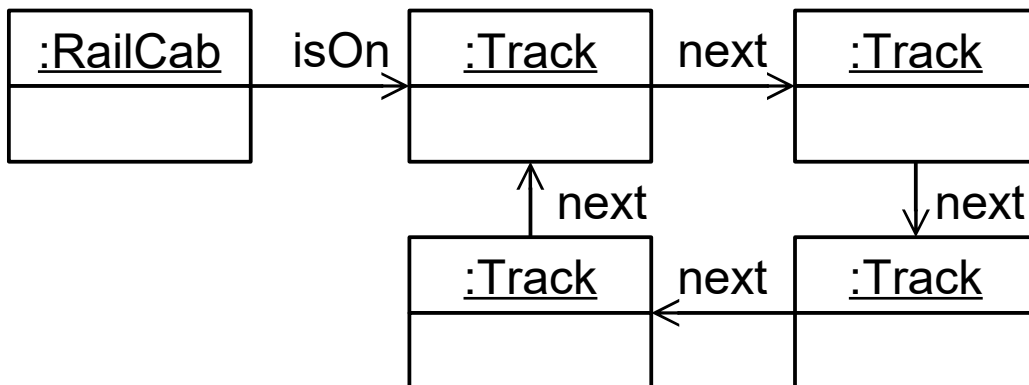
# Exploring the State Space

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

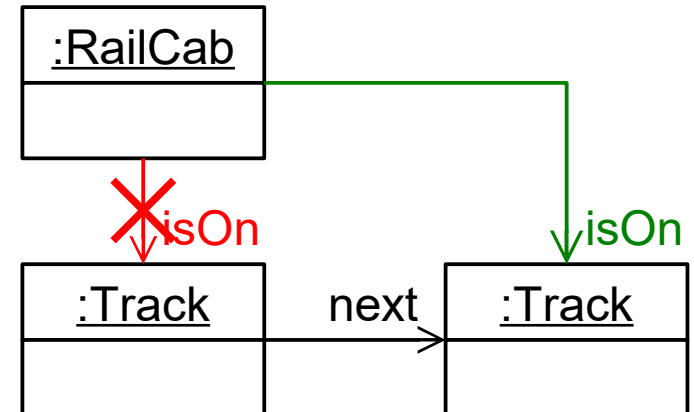


state space explored with Henshin: 4 different graphs; (graph after 4 applications of move rule is isomorphic  $\Rightarrow$  equal to the first)

start graph

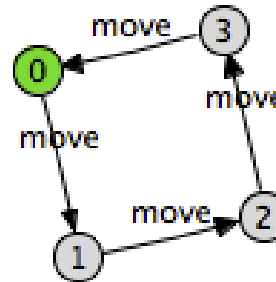


move



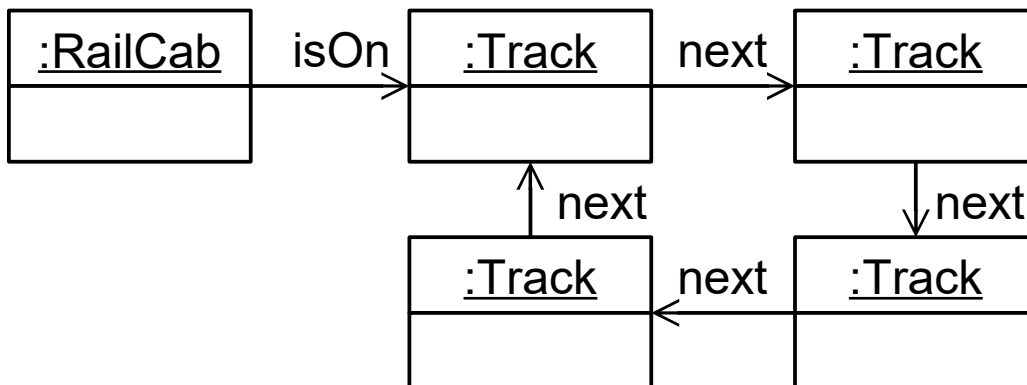
# Exploring the State Space

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

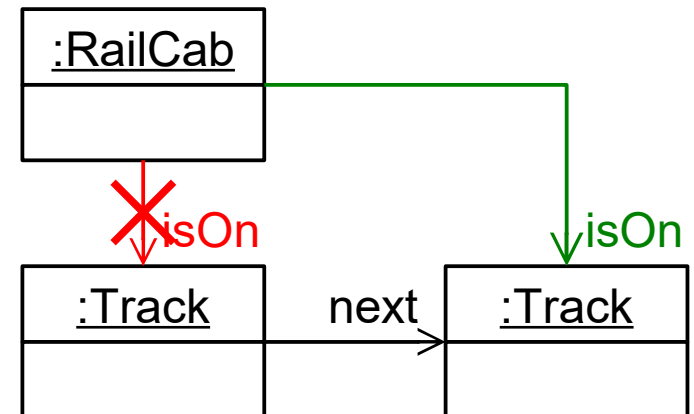


state space explored with Henshin: 4 different graphs; (graph after 4 applications of move rule is isomorphic  $\Rightarrow$  equal to the first)

start graph

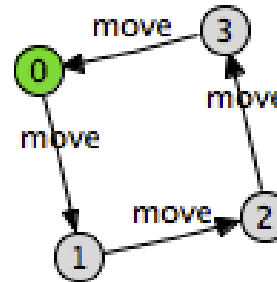


move



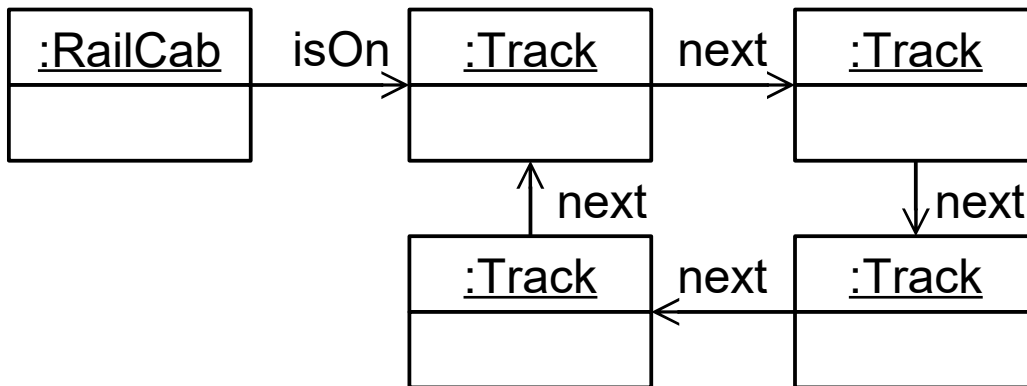
# Exploring the State Space

- 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

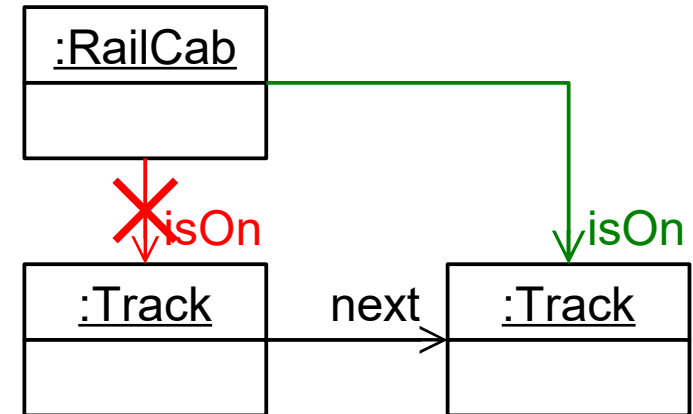


state space explored with Henshin: 4 different graphs; (graph after 4 applications of move rule is isomorphic  $\Rightarrow$  equal to the first)

start graph



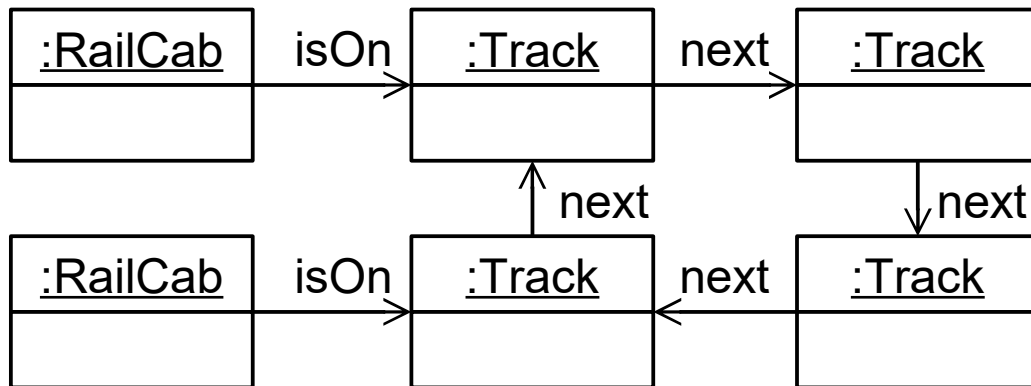
move



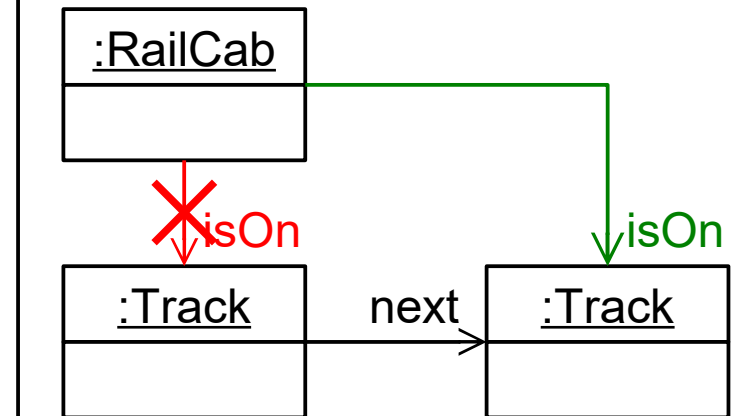
# Exploring the State Space

two RailCabs – how many states now?

start graph

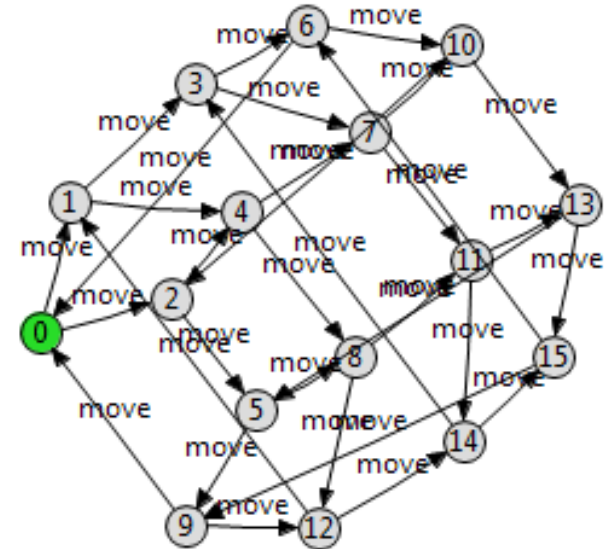


move



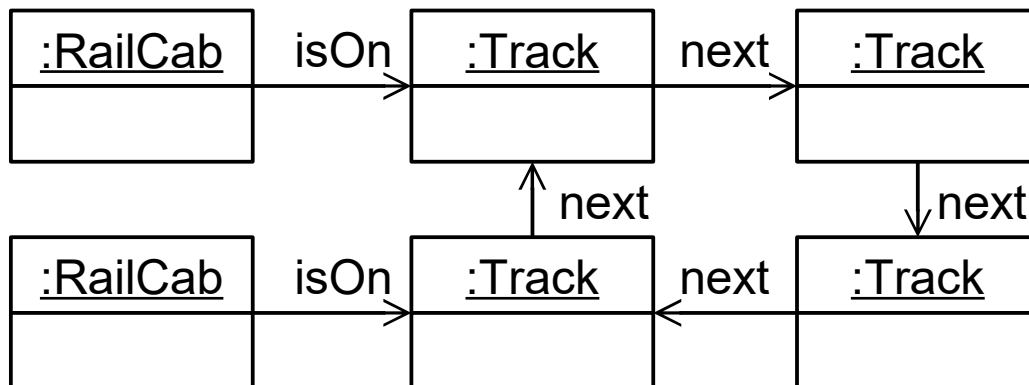
# Exploring the State Space

16 states

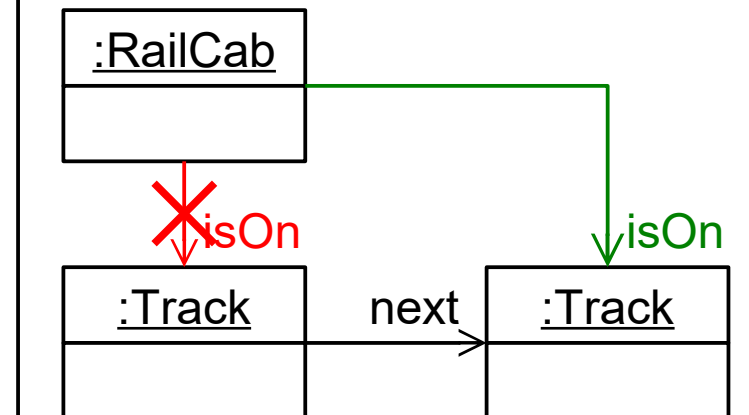


two RailCabs – how many states now?

start graph

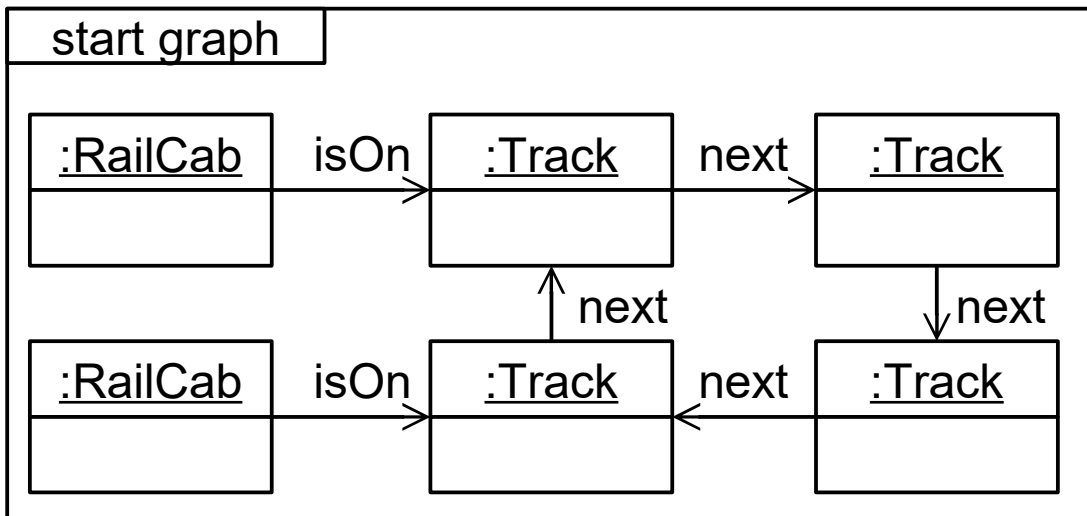
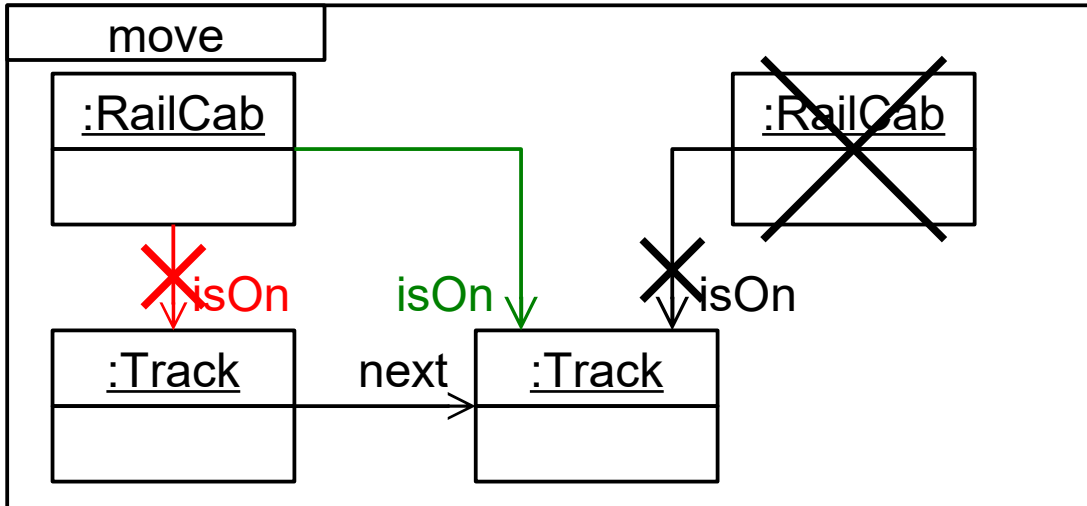


move



# Exploring the State Space

two RailCabs and not a RailCab moving on a track if another RailCab is already on it – how many states?

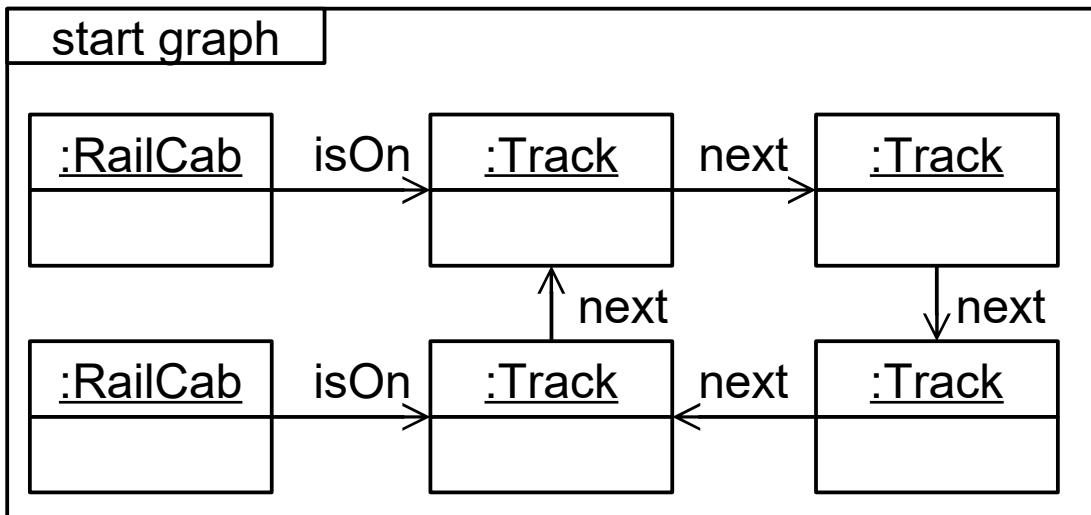
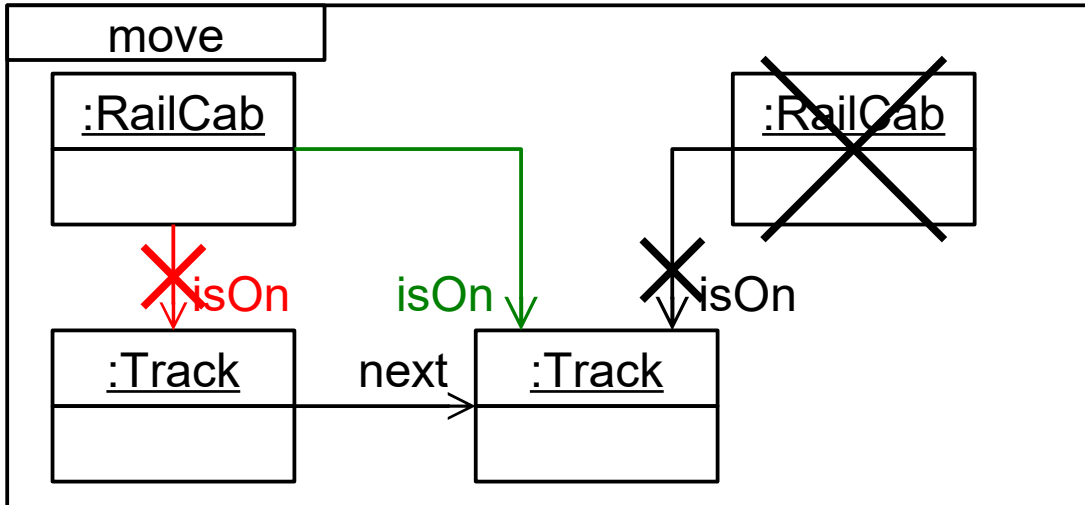
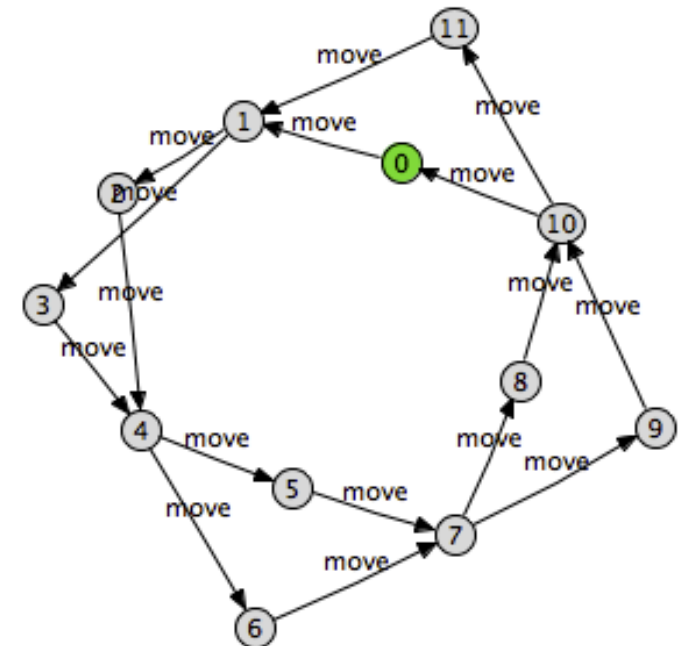




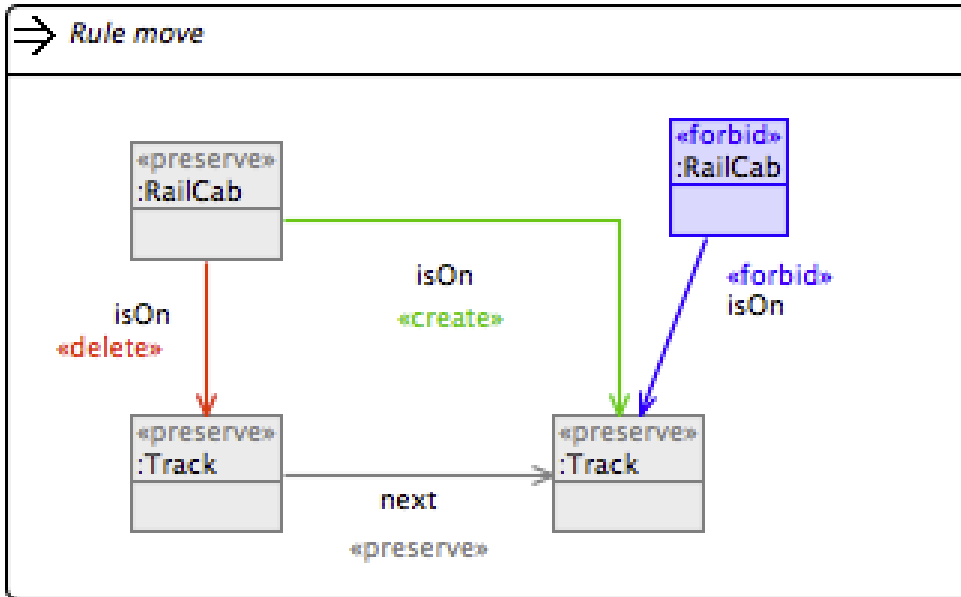
# Exploring the State Space

two RailCabs and not a RailCab moving on a track if another RailCab is already on it – how many states?

16-4 = 12 states



# Exploring the State Space



rule as specified  
in Henshin

start graph

