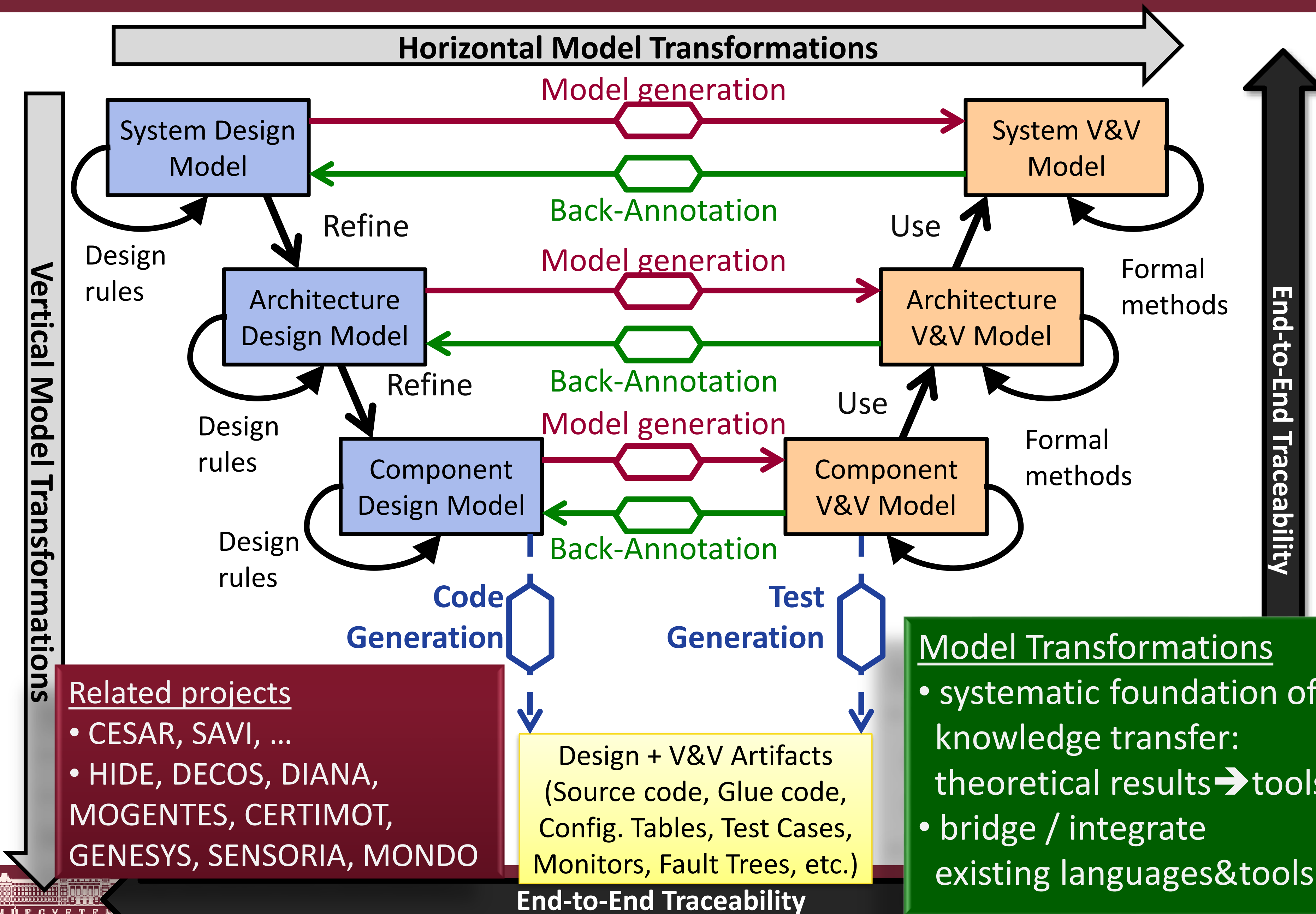
Foundations of Model Transformation

Model Driven Systems Development

Lecture 9-10

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

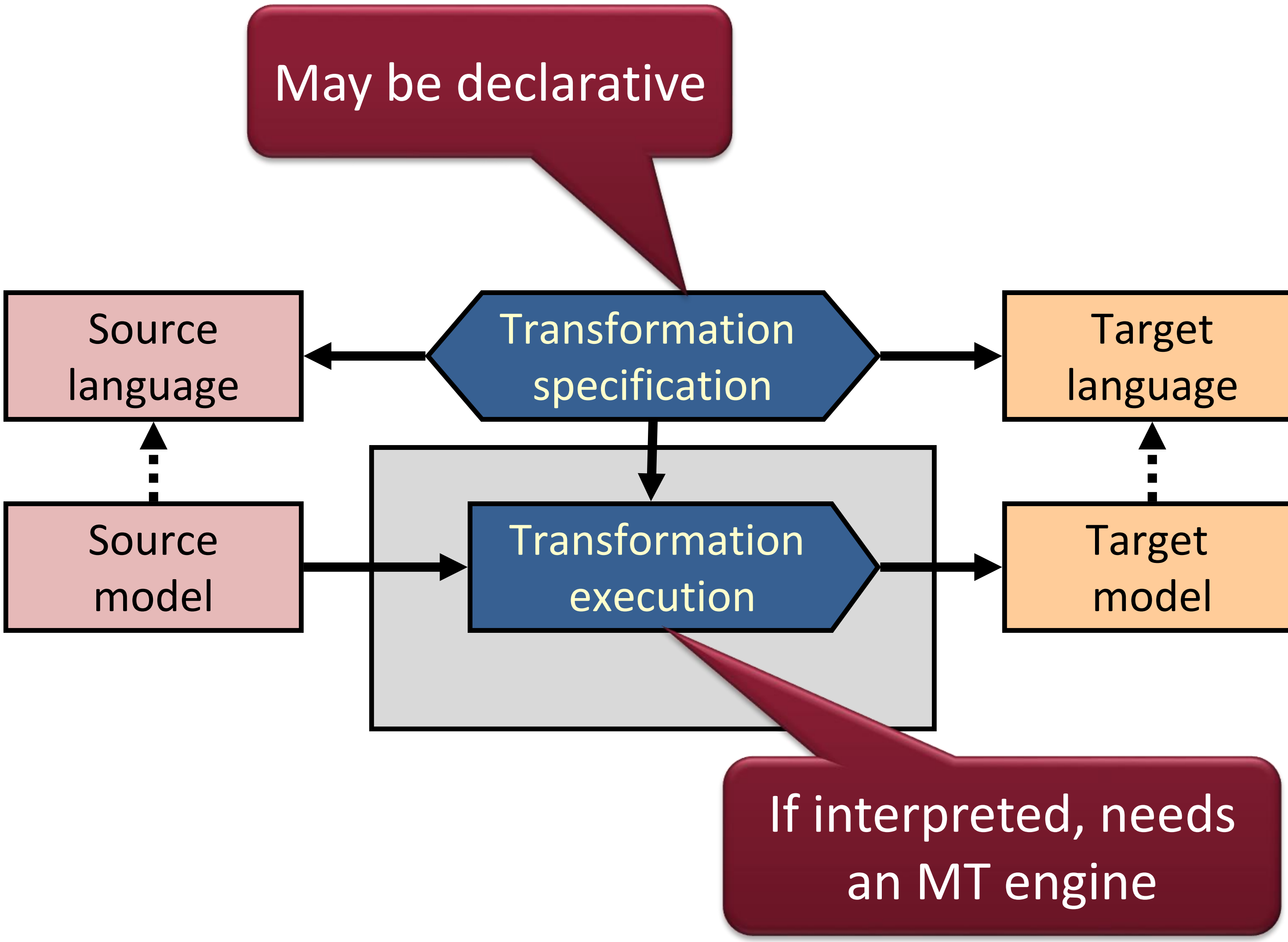
Models and Transformations in Critical Systems



<u>Related projects</u>
- CESAR, SAVI, …
- HIDE, DECOS, DIANA, MOGENTES, CERTIMOT, GENESYS, SENSORIA, MONDO

<u>Model Transformations</u>
- systematic foundation of knowledge transfer: theoretical results→tools
- bridge / integrate existing languages&tools

Definition of Model Transformation

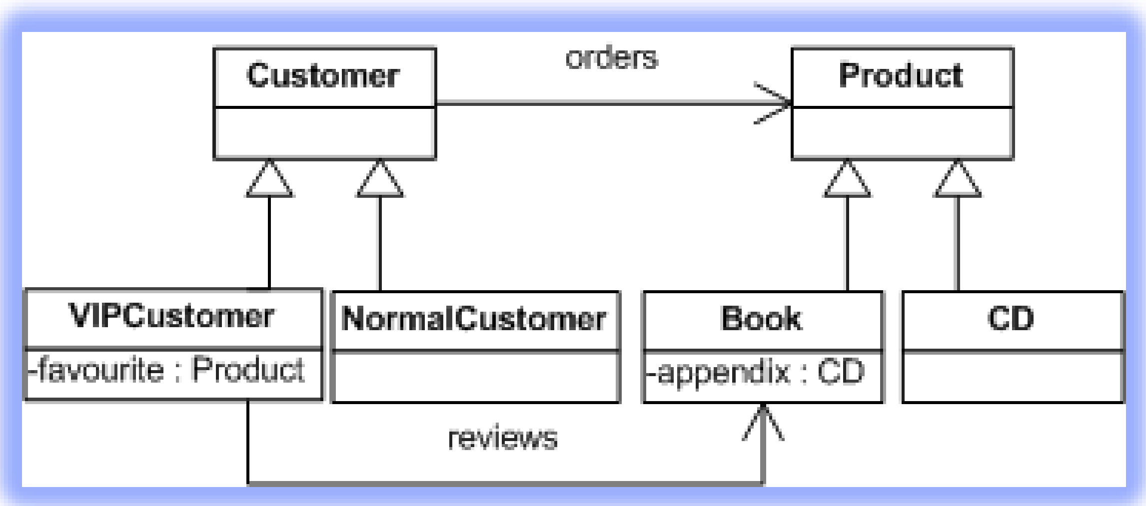


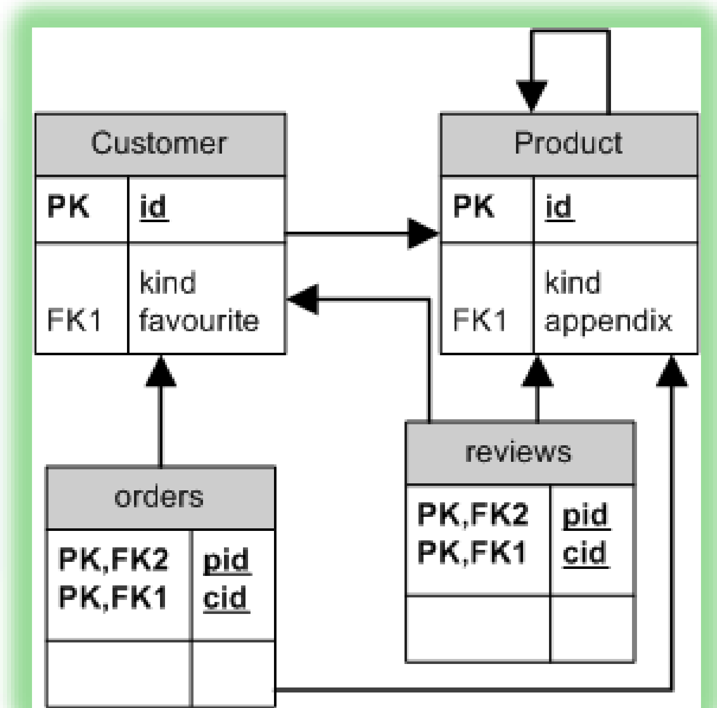
Motivating Example

Object Relational Schema mapping

Example: Object-relational maping

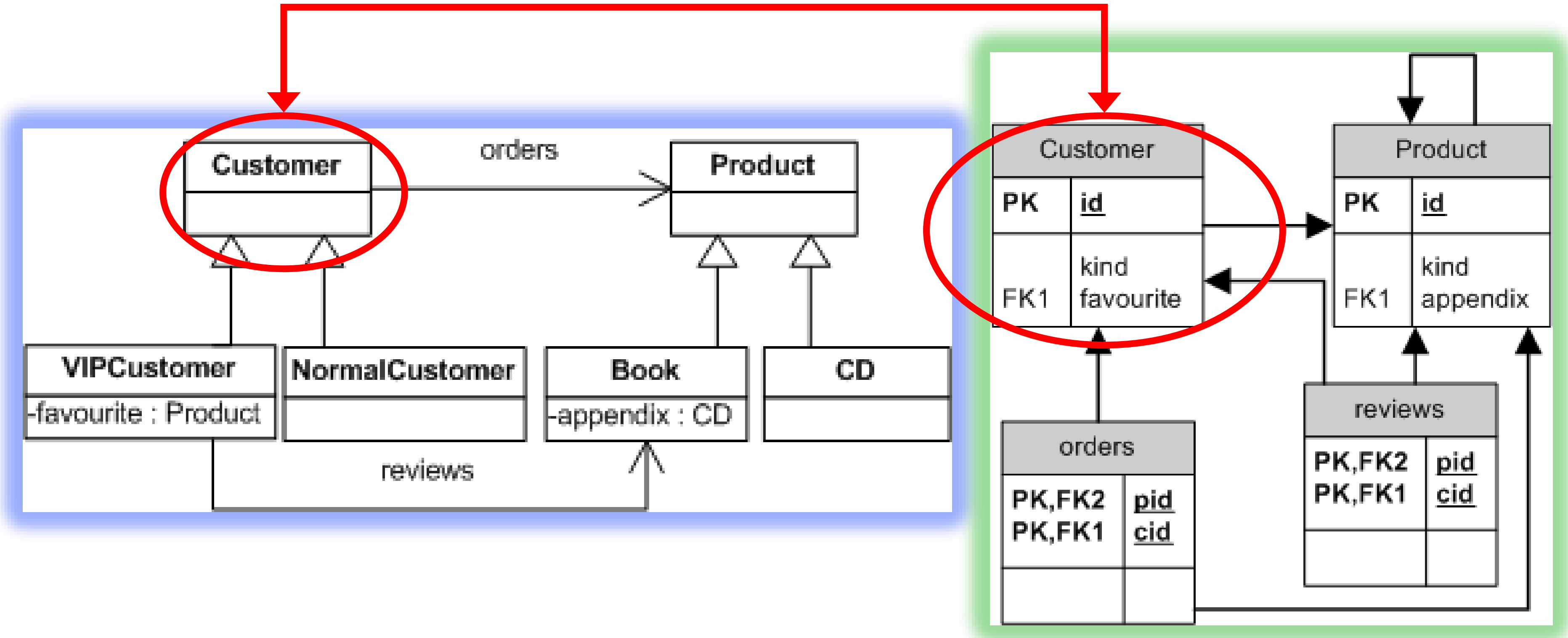
- Important as:
  - Model transformation benchmark
  - Most widely used industrial model transformation (pl. Hibernate, EJB, CDO)
- Objective:
  - **Input**: UML class diagram
  - **Output** Relational database schema





Several alternative ORM strategies, we'll use one

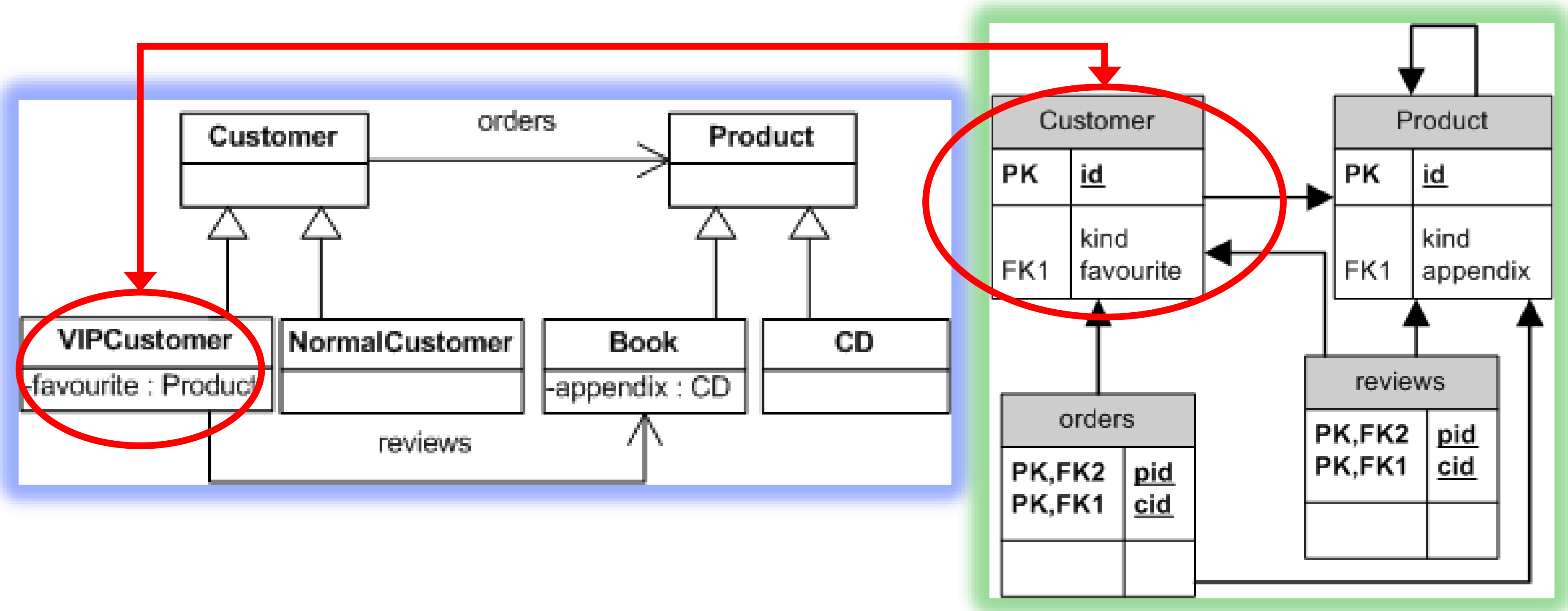
Informal definition of the MT



Topmost (generalization) classes ➔ Database table + 2 column:

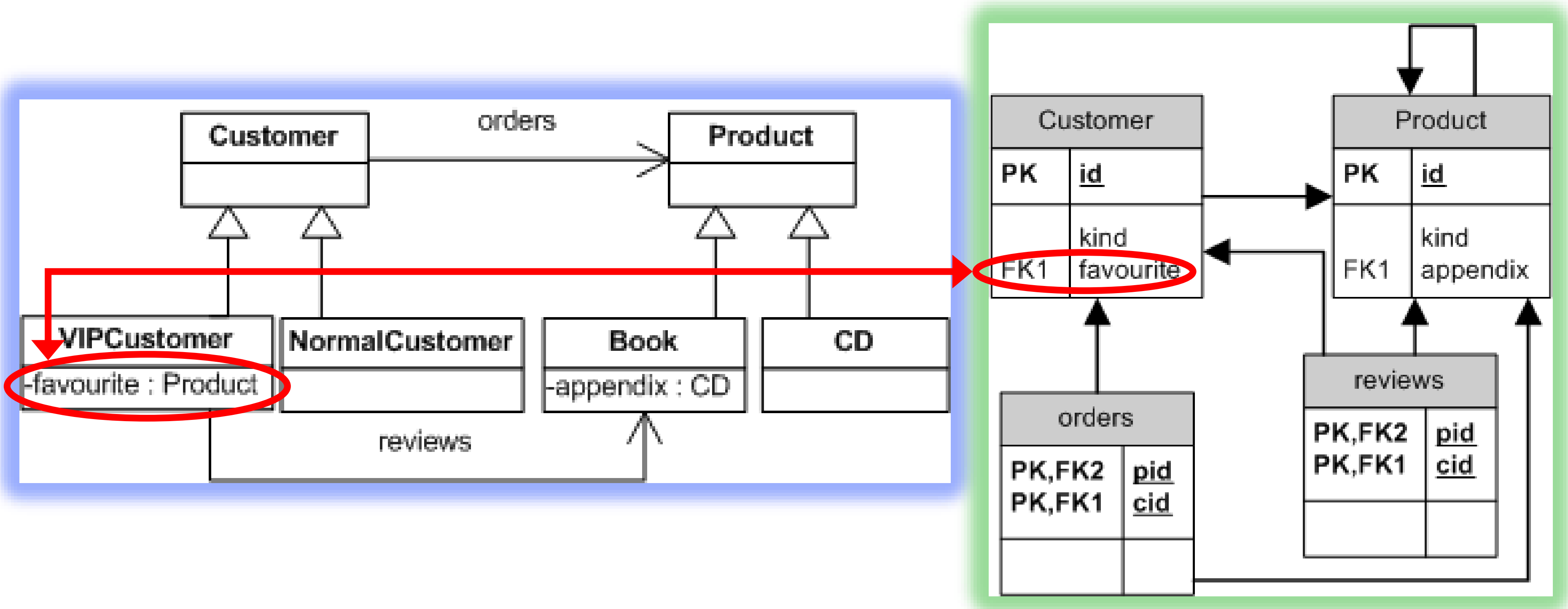
- Unique identifier (primary key),
- type definition

Informal definition of the MT



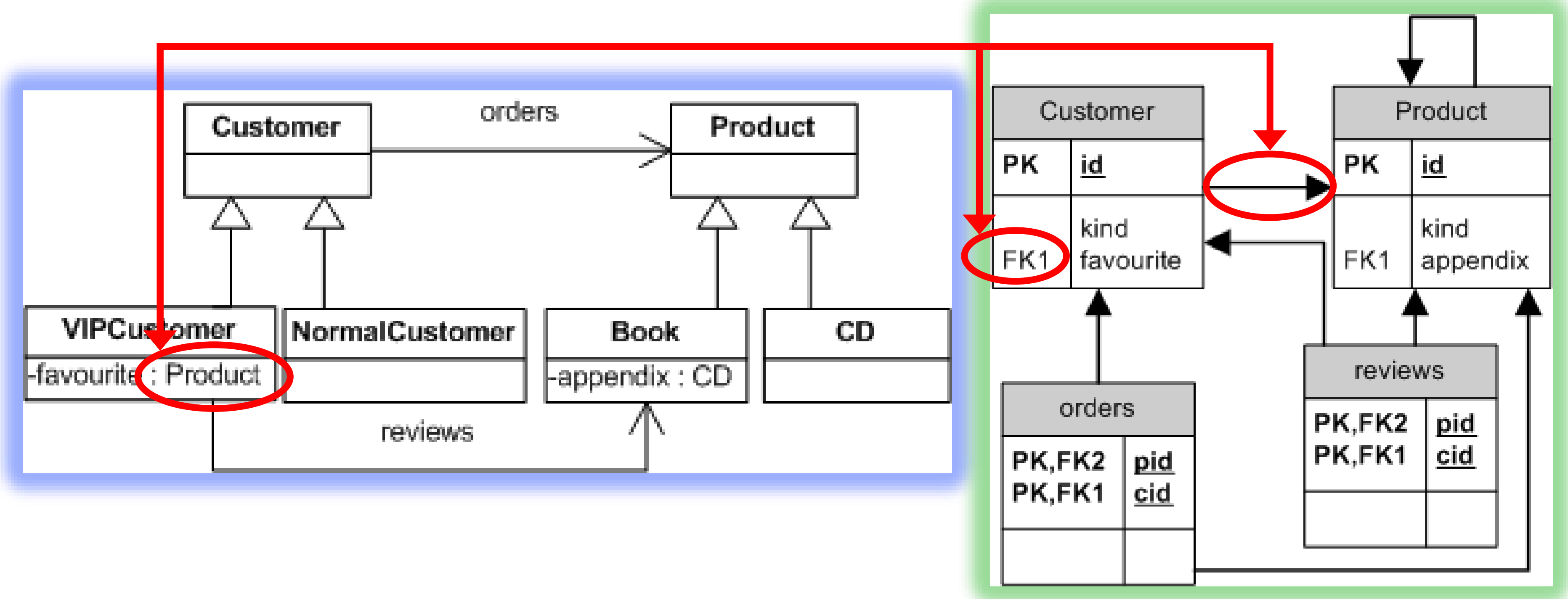
Subclasses ➔ Store instances in the same table as the root class

Informal definition of the MT



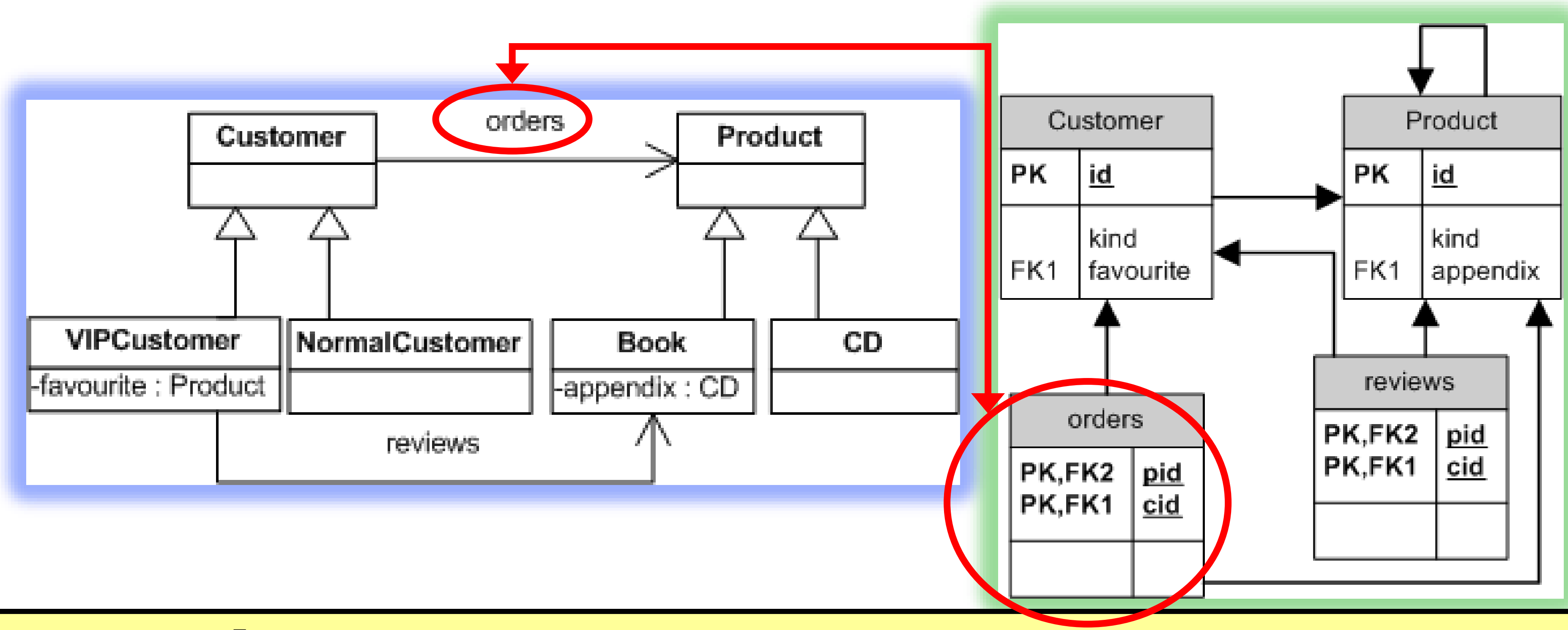
Class attributes ➔ Column of the table

Informal definition of the MT



Type of the attributes ➔ foreign key

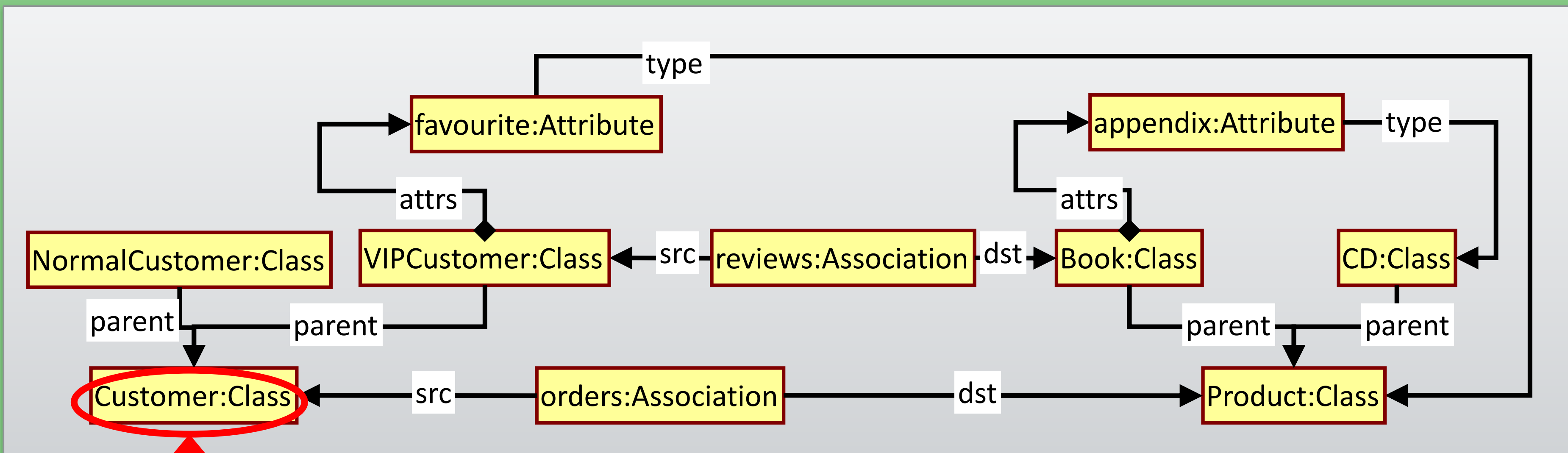
Informal definition of the MT

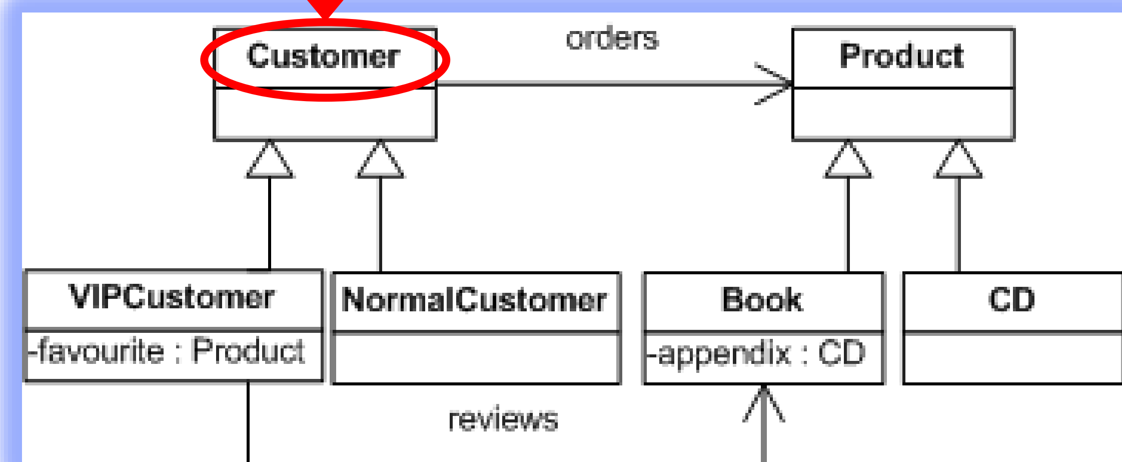


Association ➔ A table with two columns

- source and target identifiers
- foreign keys (for consistency)

Language structure (UML)





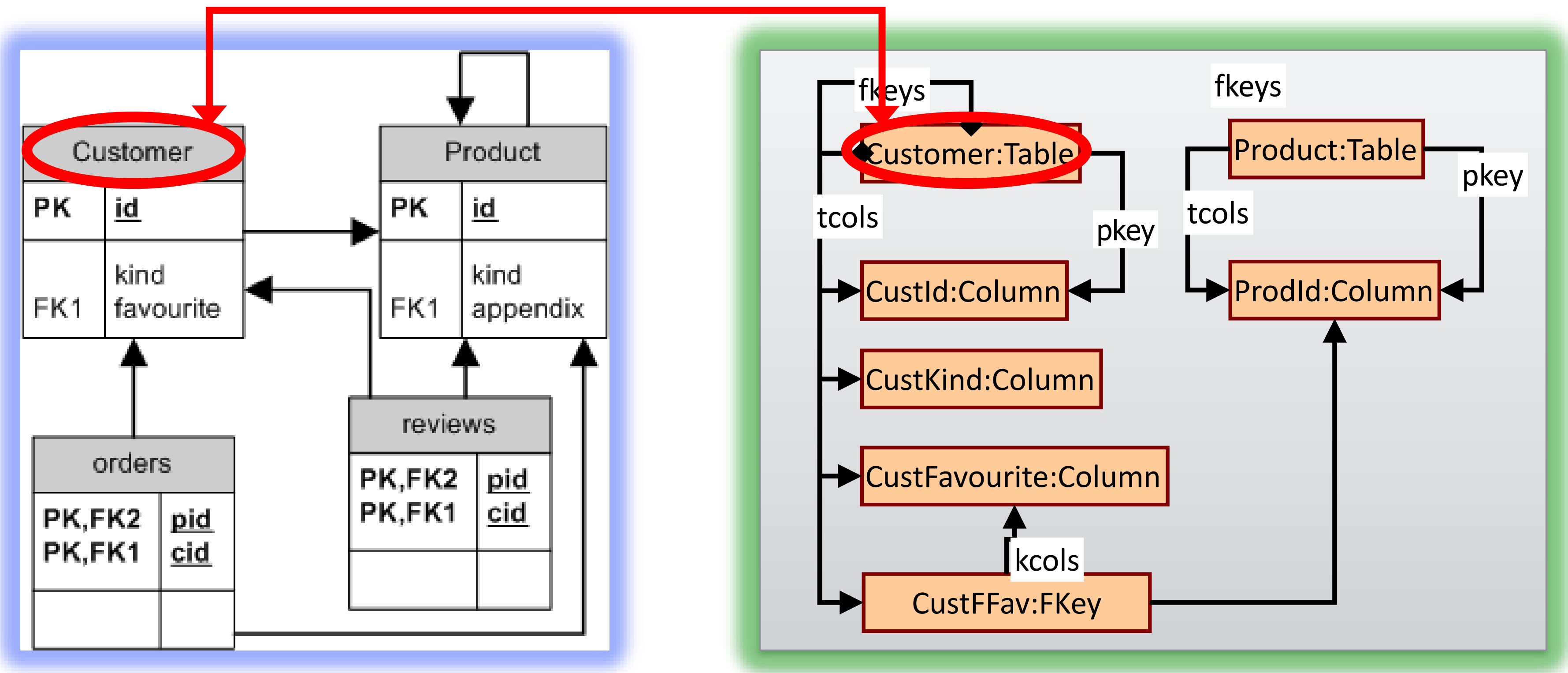
Abstract syntax

- Graph based model representation
- Machine readable

Concrete syntax

- Visual/textual representation
- Human readable

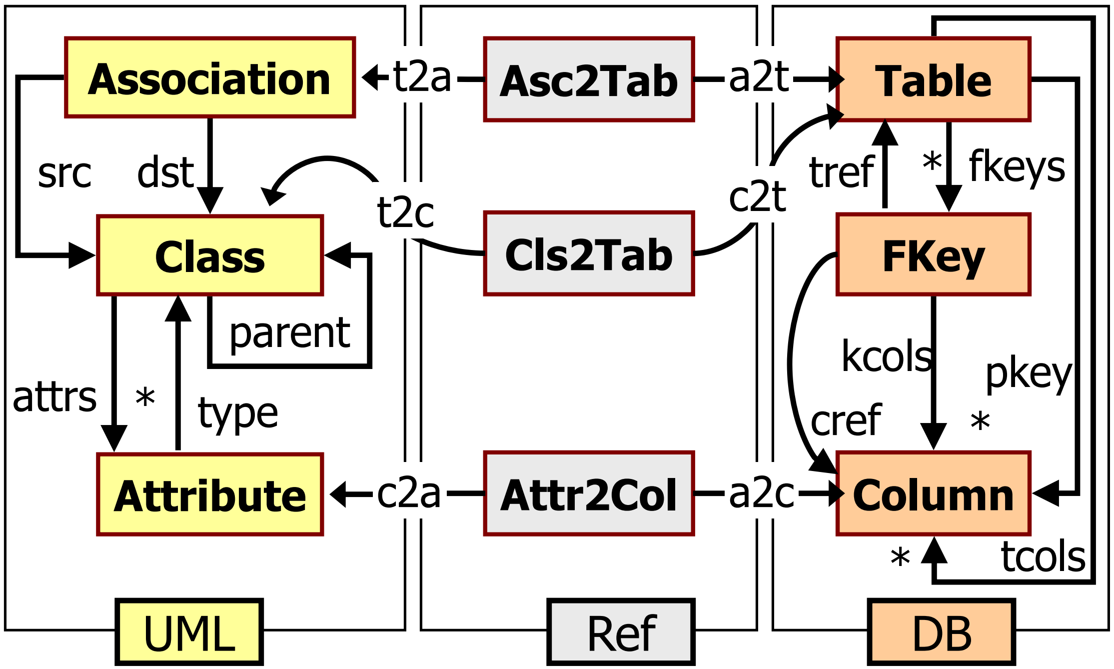
Language structure (RDB Schema)



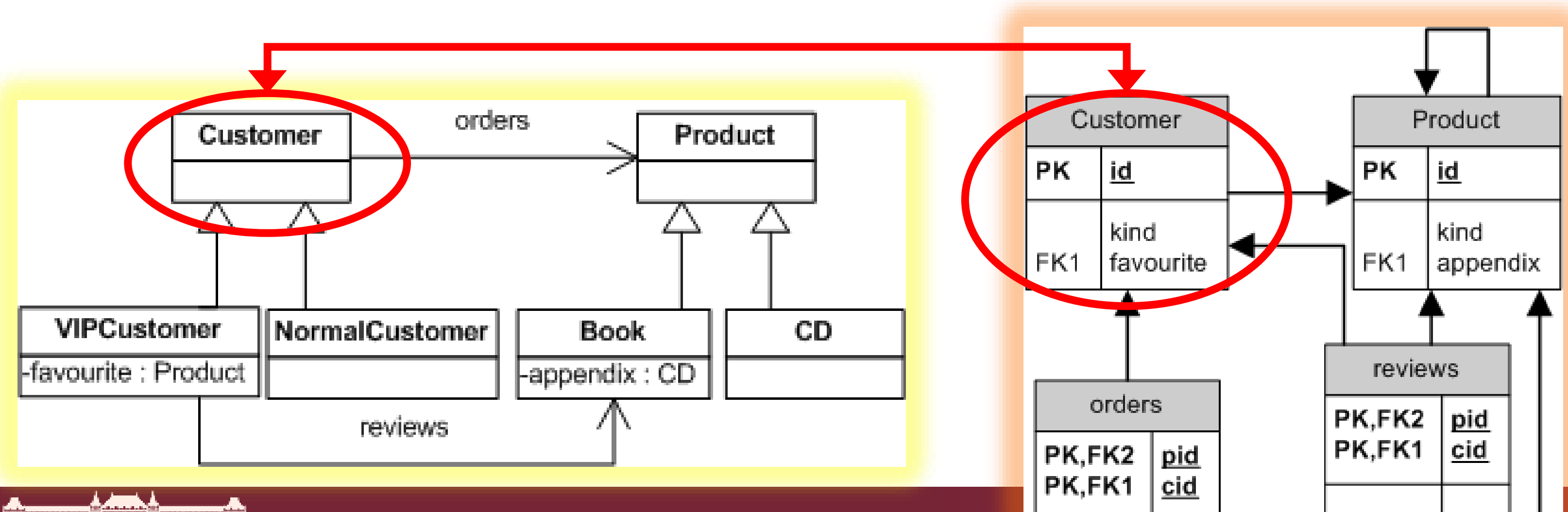
Concrete syntax

Abstract syntax

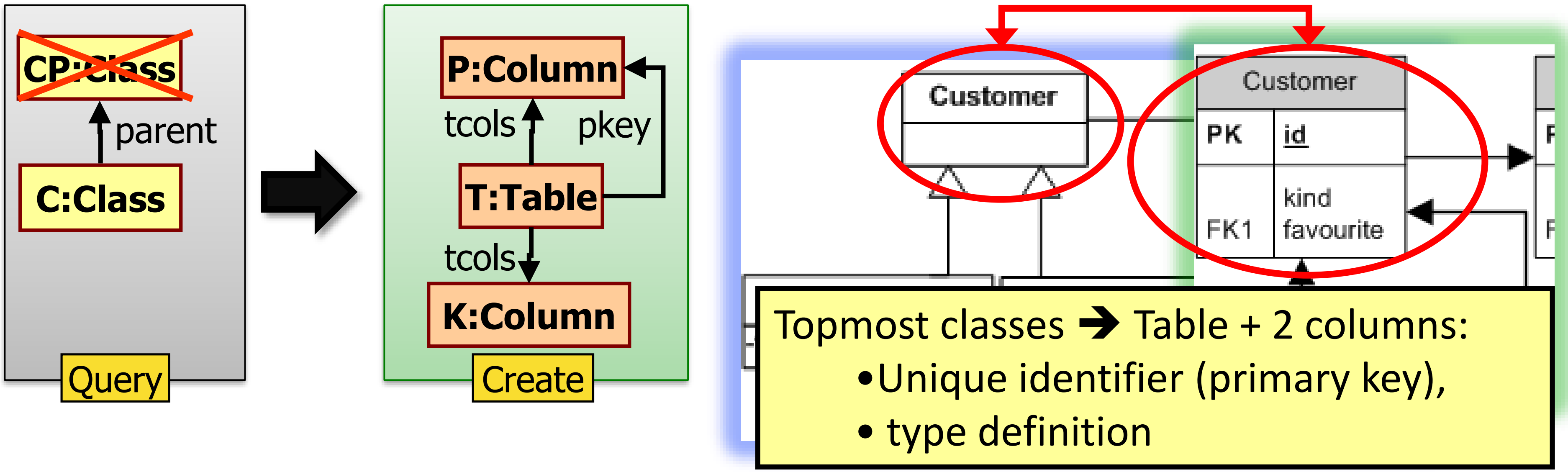
Metamodel of the O-R mapping



- Source, Target metamodels
- **Correspondence / traceability metamodel:**
  - For saving correspondence between source and target
  - Many use cases, see later

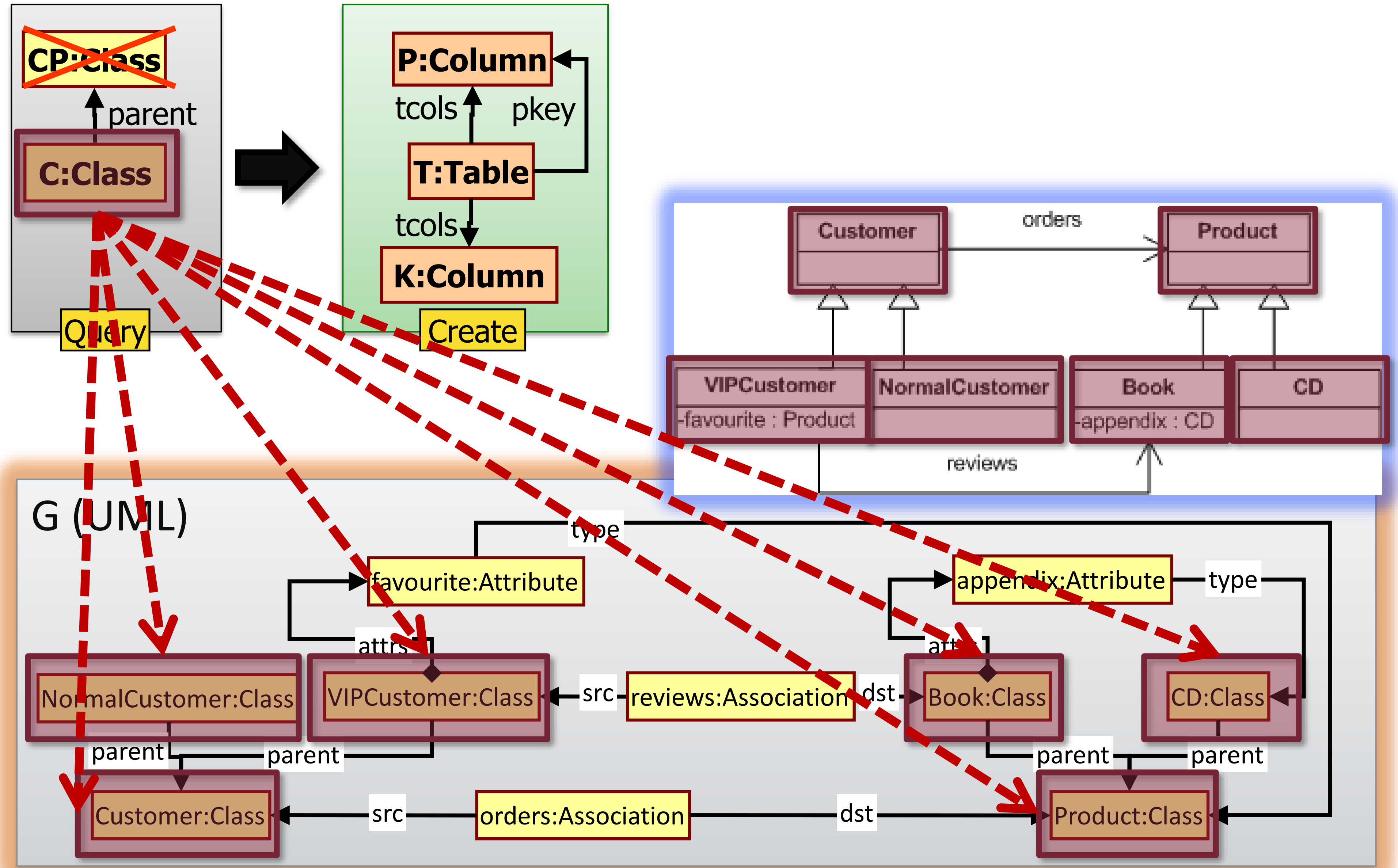


Elaborating the Solution

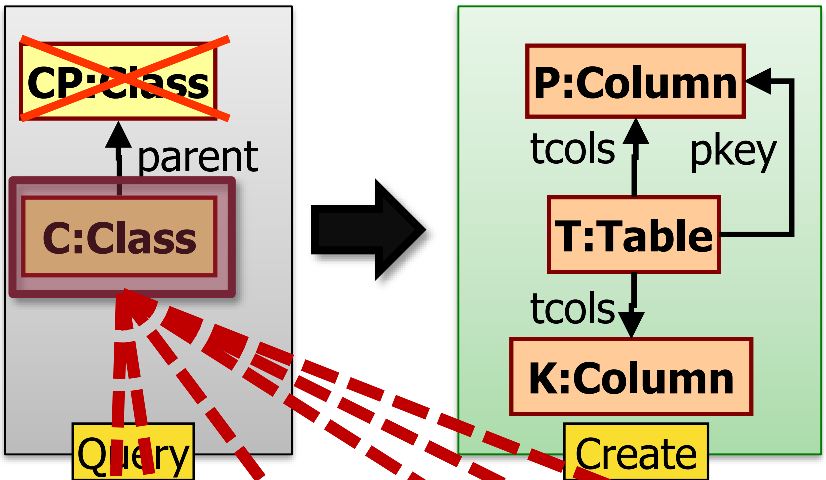


- How to execute?
  1) Evaluate **model query** on source model, find **matches**
     - Classes without superclass

Revision: graph pattern matching

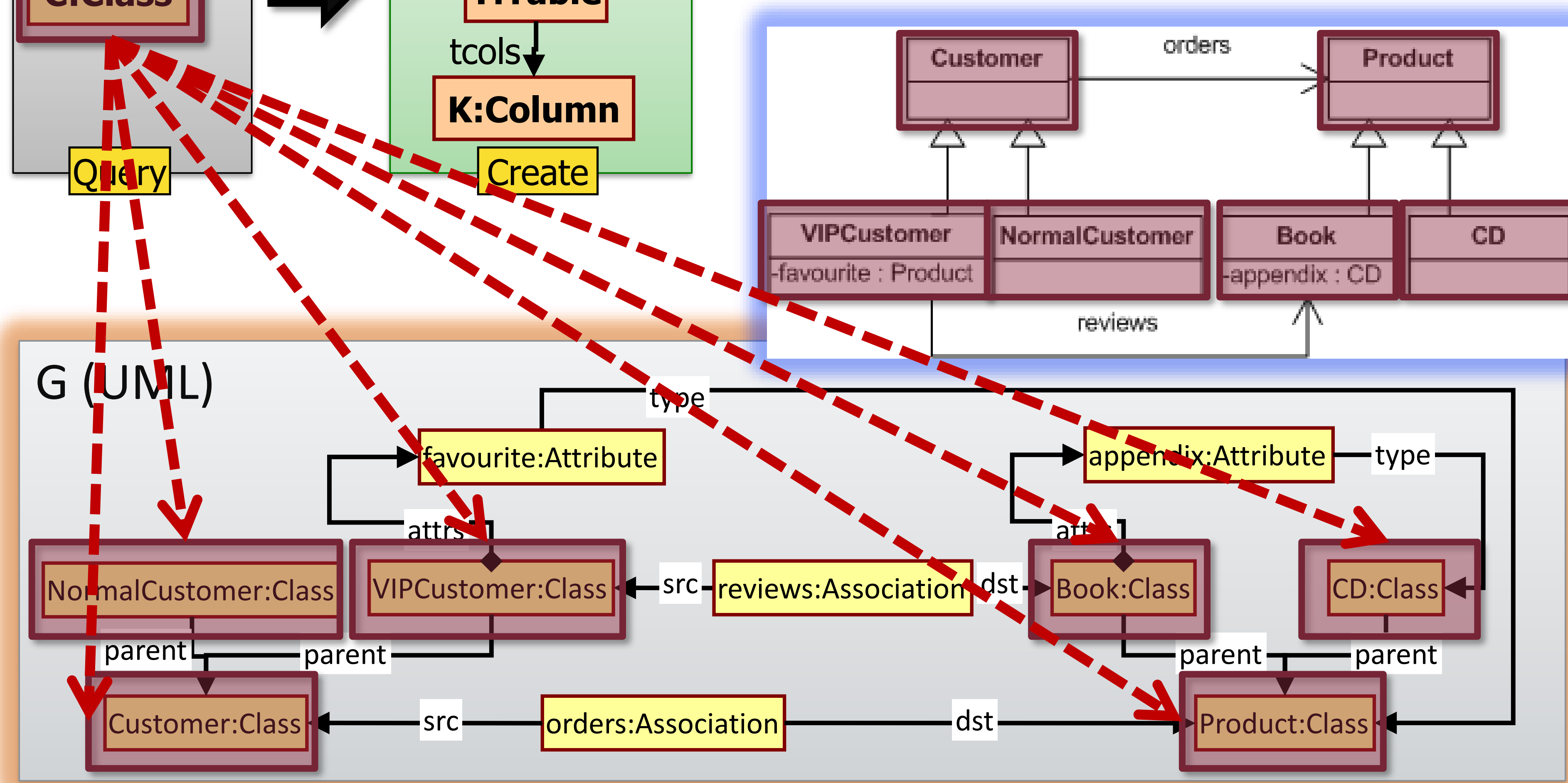


Revision: graph pattern matching

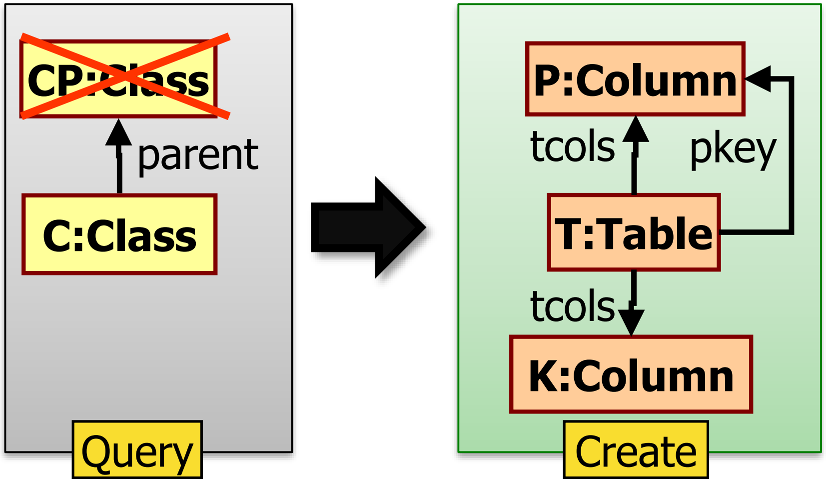


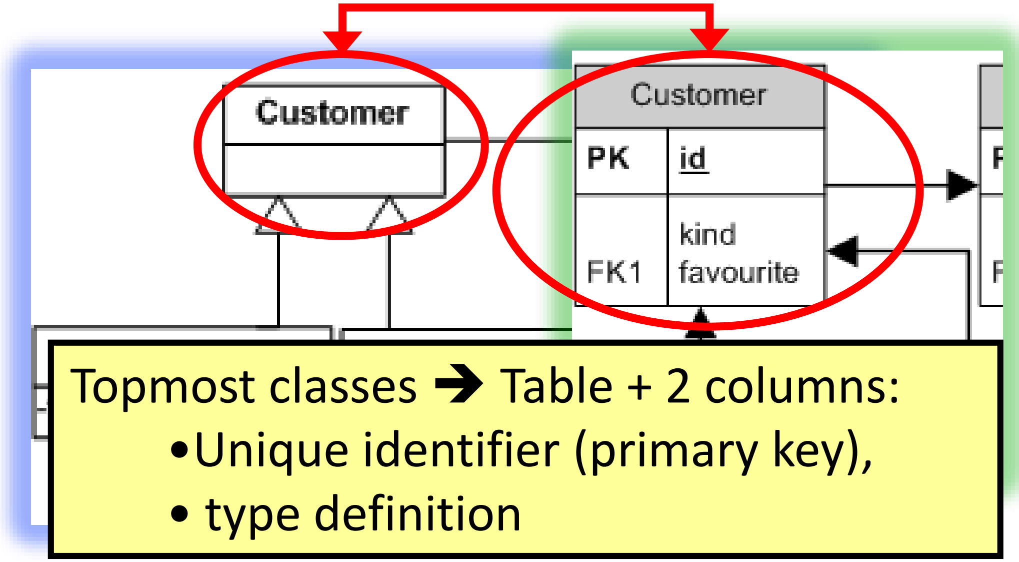
Negated constraint

- Successful match of negative condition ➔ pattern does not match



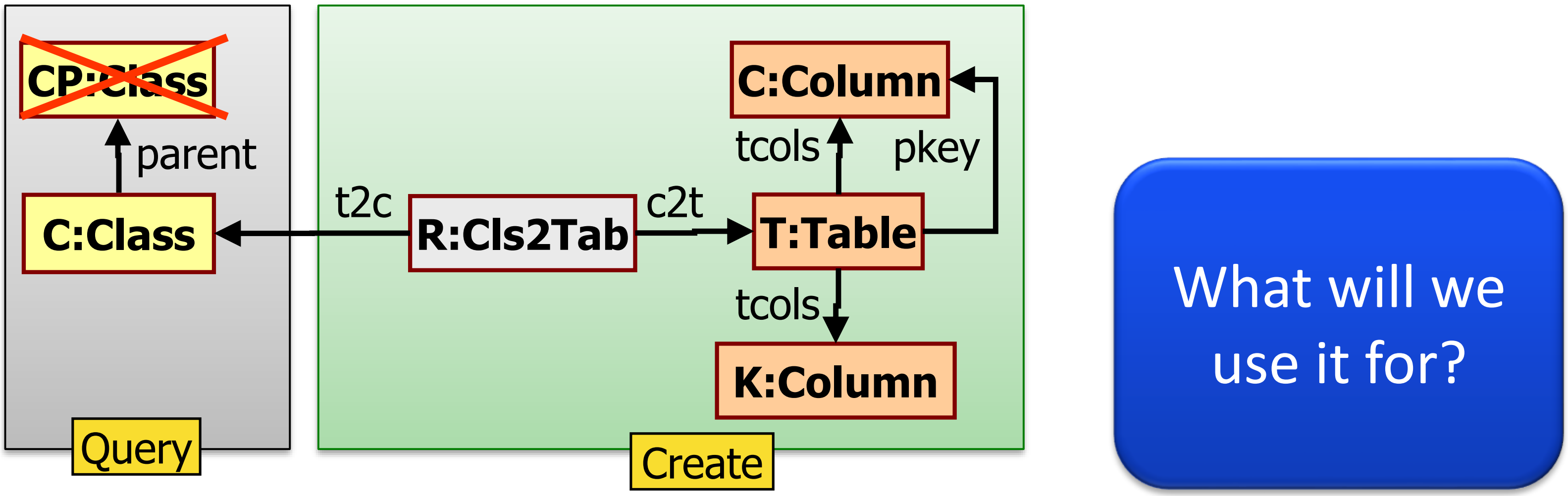
Elaborating the Solution





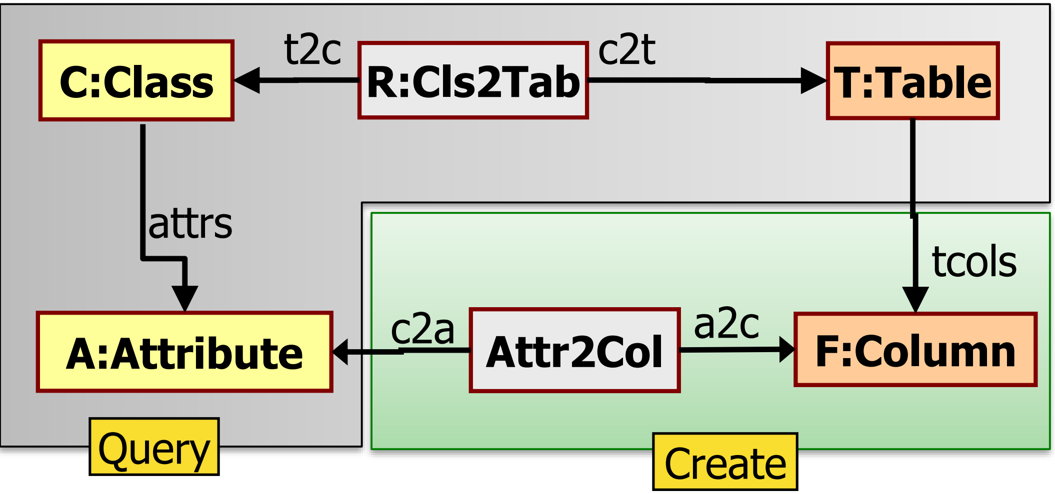
- How to execute?
  1) Evaluate **model query** on source model, find **matches**
     - Classes without superclass
  2) For each match, create new model elements
     - Table with primary key and type columns
     - Something is missing…

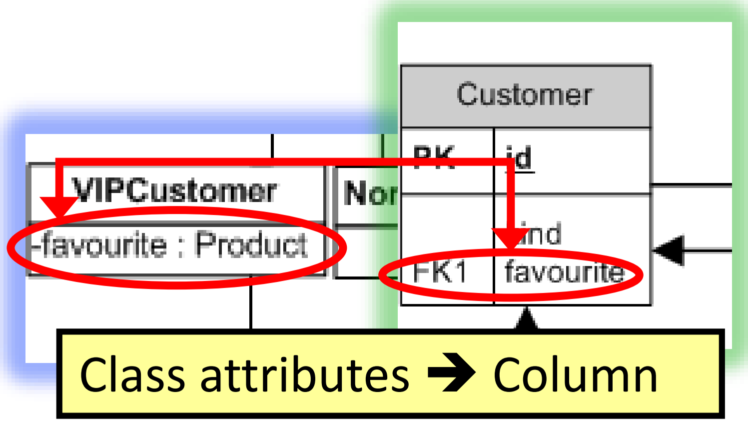
Elaborating the Solution



- How to execute?
  1) Evaluate **model query** on source model, find **matches**
     - Classes without superclass
  2) For each match, create new model elements
     - Table with primary key and type columns
     - **Correspondence** (traceability) between table and class

Elaborating the Solution



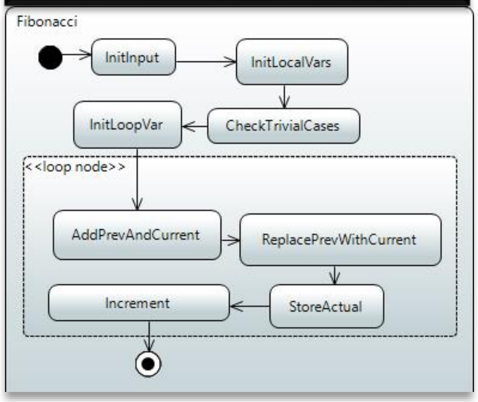


- Which table should the column belong to?
  - Build on previous steps, using correspondence
- Apply the same idea for the rest:
  - Associate subclass to table of parent class
  - Map associations, map types of attributes, etc.

Chaining and Traceability of Model Transformations

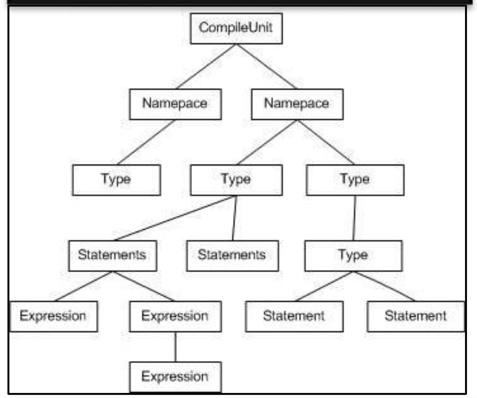
Code Generation by Model Transformations

Source Model



M2M

Code DOM/AST



M2T

Target Code

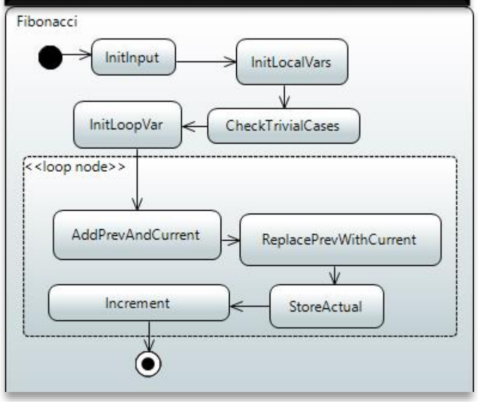
```
#include<iostream>
using namespace std;

int main() {
int number, reverse = 0;
 cout<<"Input a Number to Reverse:  ";
 cin>> number;

   for( ; number!= 0 ; )
   {
      reverse = reverse * 10;
      reverse = reverse + number%10;
      number = number/10;
   }
   cout<<"New Reversed Number is:  "<<reverse;

   return 0;
}
```

Model-to-Model (M2M) Transformation
- SRC: In-memory model (objects)
- TRG: In-memory model (objects)

Model-to-Text (M2T) Transformation
- SRC: In-memory model (objects)
- TRG: Textual code (string)

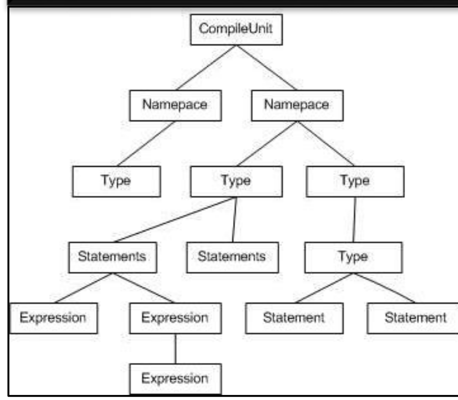
Chaining of Model Transformations

Source Model



M2M

Code DOM/AST



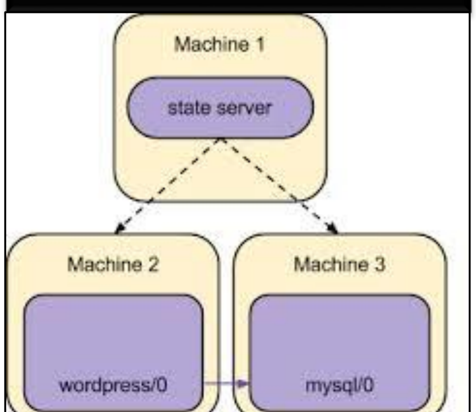
M2T

Target Code

```
#include<iostream>
using namespace std;

int main() {
int number, reverse = 0;
 cout<<"Input a Number to Reverse:  ";
 cin>> number;

   for( ; number!= 0 ; )
   {
      reverse = reverse * 10;
      reverse = reverse + number%10;
      number = number/10;
   }
   cout<<"New Reversed Number is:  "<<reverse;

   return 0;
}
```

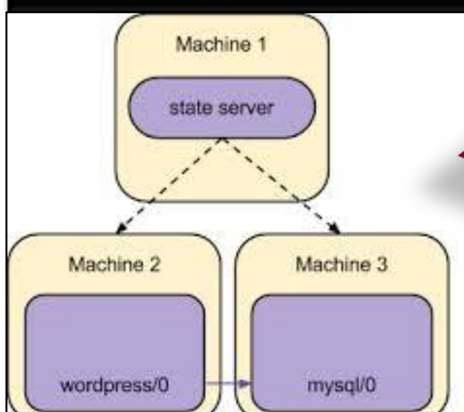
M2M

Inter Model 1



M2M

Inter Model 2

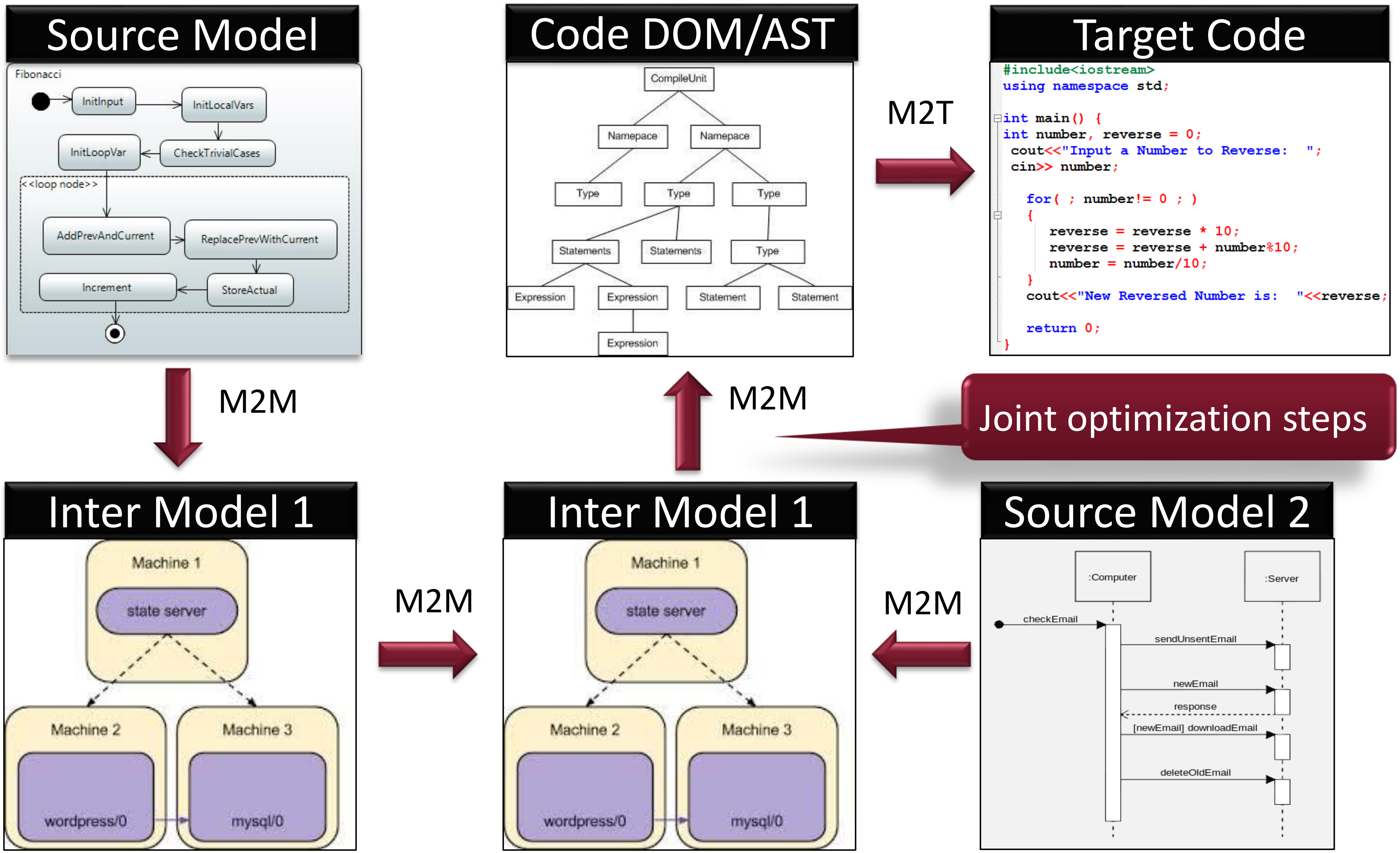


M2M

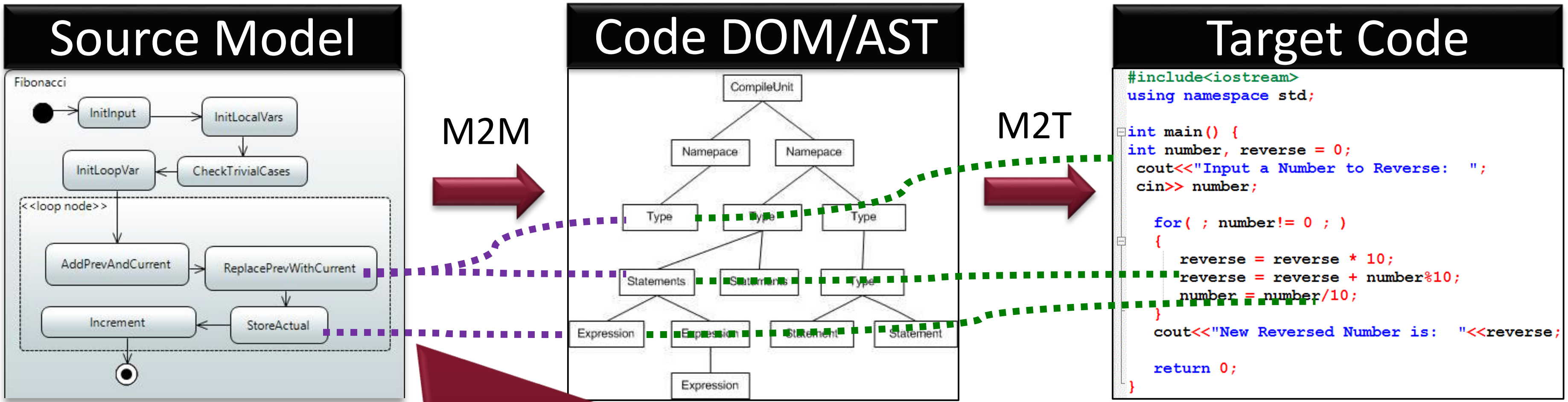
Goal:

- Reduce abstraction gap by „divide and conquer”
- Intermediate models
- Chain of model transformations

Model Transformation Flows / Chains



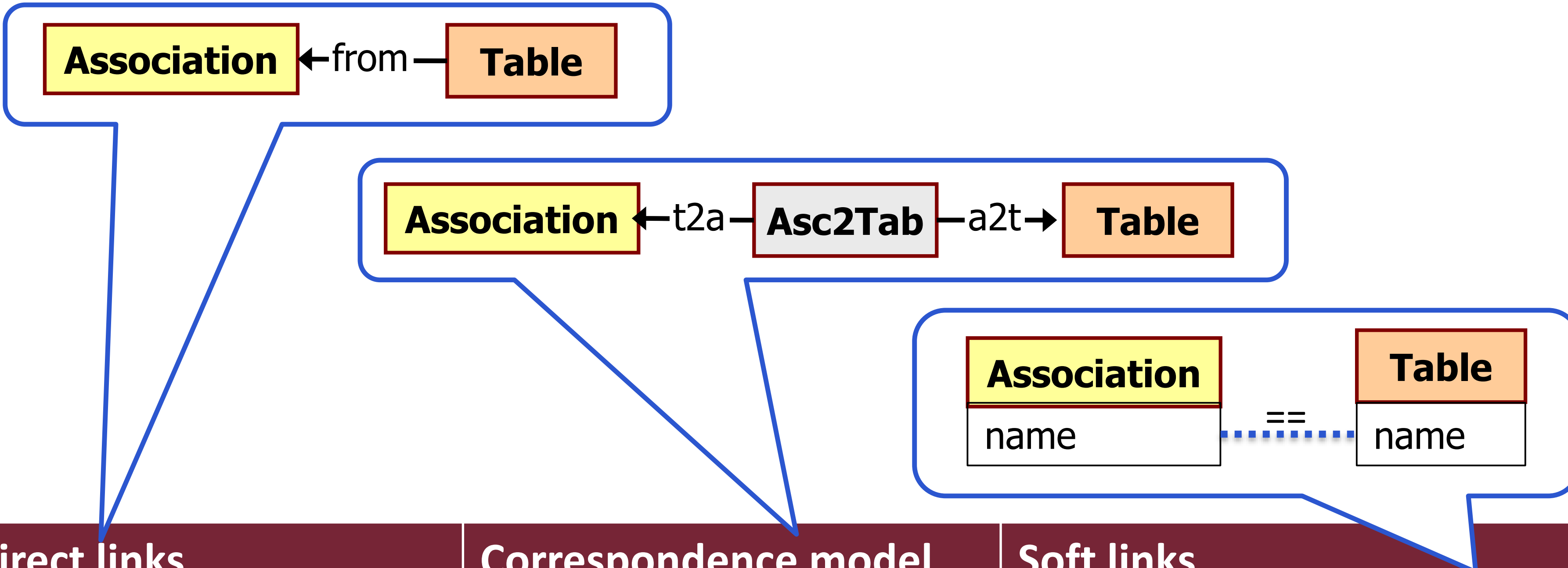
Traceability in Model Transformations



Traceability / correspondence links:
- Connect SRC and TRG models

Objectives:
- Make transformation specification easier
- Support end-to-end traceability
- Improve incrementality (see later)

Forms of Traceability



Direct links	Correspondence model	Soft links
Cross-reference between SRC↔TRG	Stored in separate metamodel & model	Match by id / qualified name using model query / index
Intrusive: must extend meta & instance models	Complex, large overhead	Requires unique identifier; limited expressiveness

Rule-based Transformations

Model Transformation Specification

- Imperative with direct model manipulation
  - Quick&easy for simple batch transformations
  - But what if we need…
    - Incrementality?
    - Bidirectionality?
- **Rule-based declarative**
  - *Graph Transformation* based
  - Hybrid: query + imperative action (VIATRA etc.)
  - „Relational” (QVT-R, TGG, ATL, etc.)
  - „Explicit”

Rule-based MT core idea

- Unit: **MT rule**

For each occurrence of…	**…transform it like this**
Root class in inheritance hierarchy	Create entity table with default columns
Attribute of class	Add columns to table of class
Association between classes	Create switch with foreign key columns
PRECONDITION **Declarative Model Query**	**ACTION** **May be imperative**

- This is just the core idea, many variants
  - We'll discuss two formalisms later (VIATRA, GT)

Inversion of Control (**IoC**)

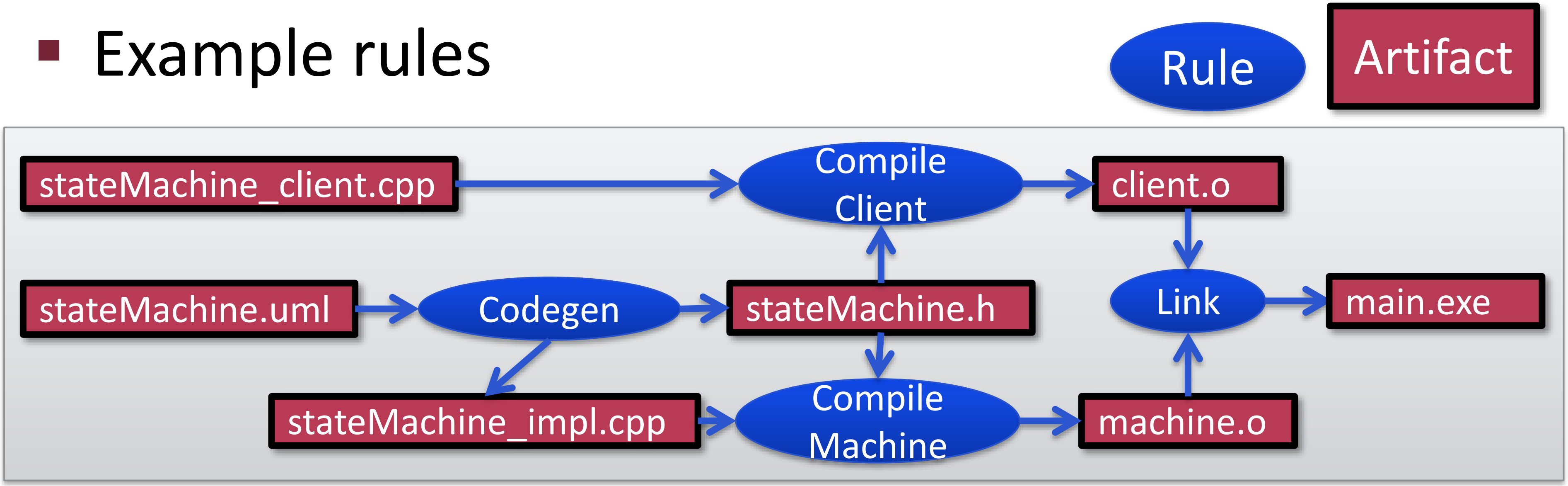
- Declarative rule execution
  - **Transformation engine** interprets preconditions
  - Rules are **fired** by engine when&where enabled
- Several variants
  - „As long as possible" / „fire why possible" semantics
    - Iterate while there are **rule activations**
    - Select one activation (**conflict resolution**), fire it
  - „Fire all current" semantics
    - Select all *current* activations, fire them all, stop
  - Arbbitrary control flow

Rule-based Systems

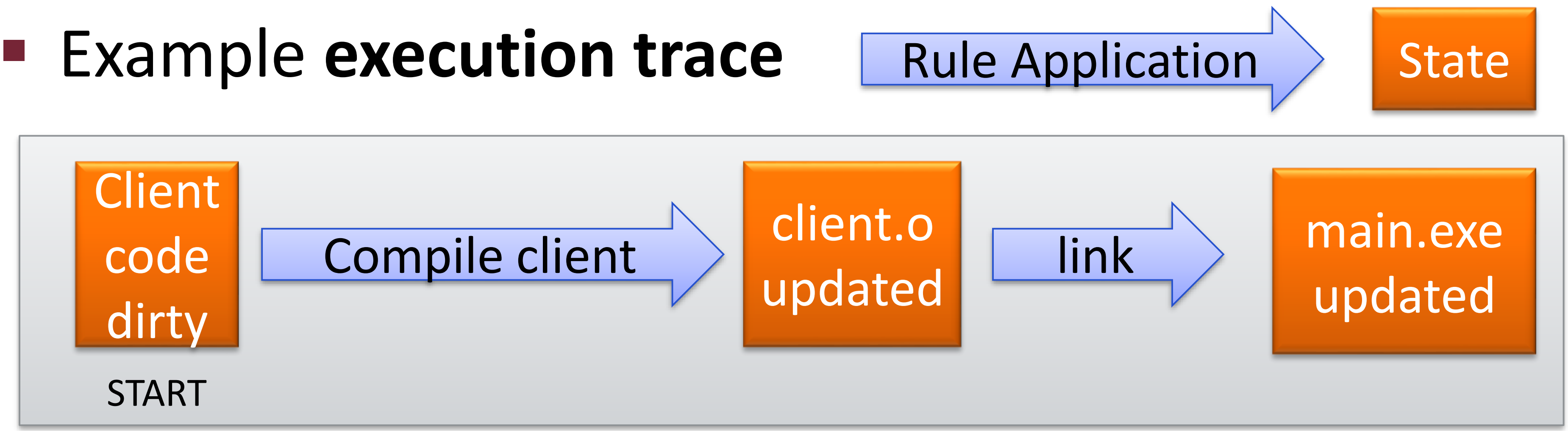
- Where have I seen rule-based systems?
  - **Model transformations** — We are interested in this
  - Build scripts (MAKEFILE, Maven, etc.) — Easy example
    - Rule: build this artifact *like this* (action) when those others are ready and dirty (precondition)
  - Business rule & expert systems (Jboss Drools, etc.)
  - Context-free grammars (see Textual Syntax Lecture)
  - CSS
  - …
- There are some vague commonalities

Build Script Example

- Example rules

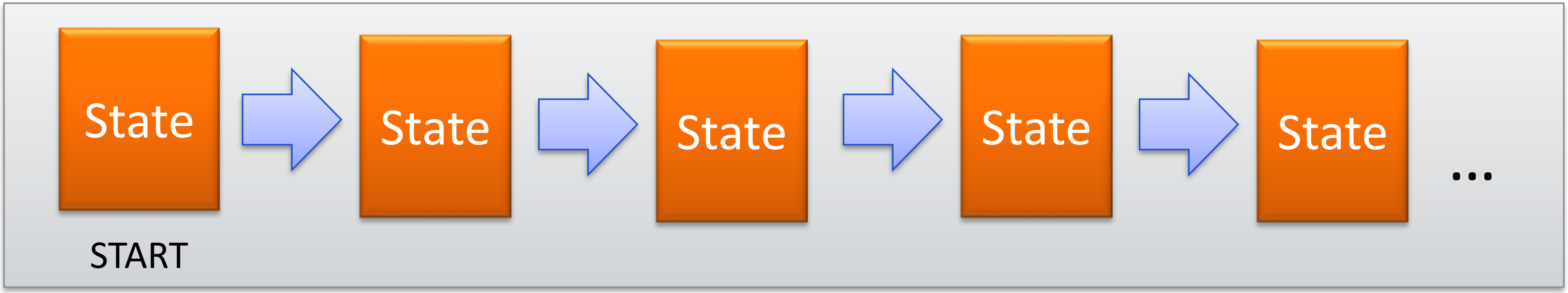


- Example **execution trace**



Common Rule-based Problems

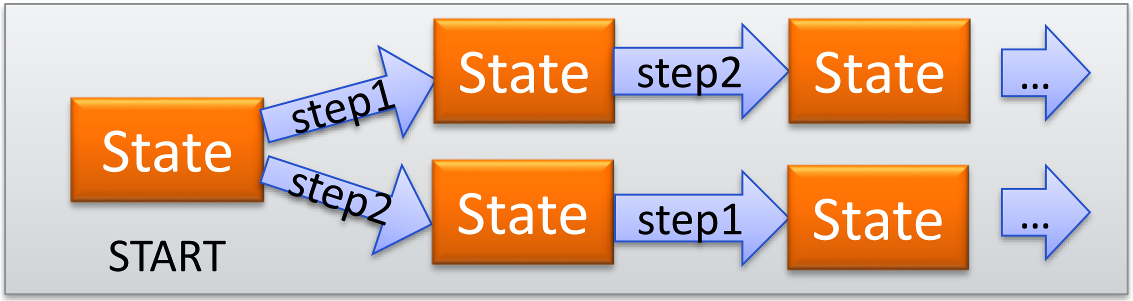
- Problem 1: **Termination**



  - Vital to ensure!
  - Non-terminating examples
    - Makefile: a build step overwrites (re-dirties) one of its inputs
    - MT rule creates new object, has to be xformed same rule
    - MT Rule1 creates element, Rule2 deletes it, Rule 1 again, …
  - No systematic way to guarantee, requires thought

Common Rule-based Problems

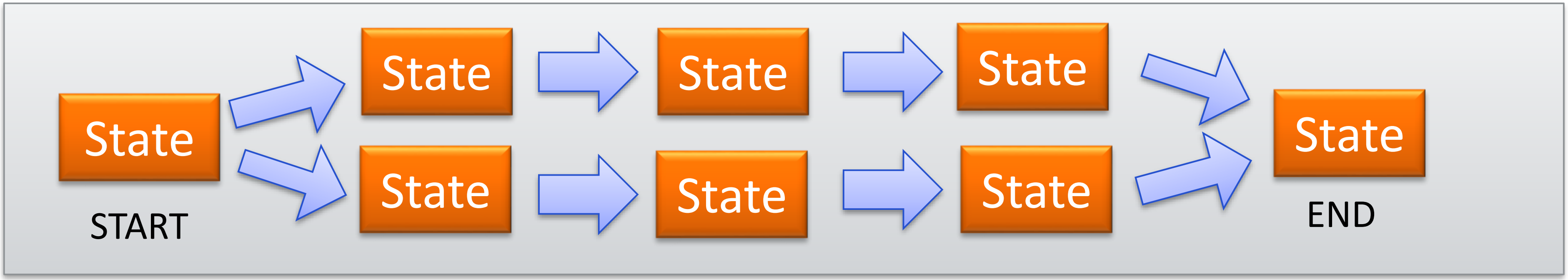
- Problem 2: **Ordering of steps** (rule applications)



  - May be required for correctness
    - MT example: transform attribute only after relevant class
  - In other cases, only performance is impacted
    - Makefile: if client is built before dirty .uml, must rebuild
  - How to manage?
    - Smart engine (limited applicability, works for Makefile)
    - Express in precondition (attribute rule requires class)
    - **Rule priorities** (execute class rules before attribute rules)

Common Rule-based Problems

- Problem 3: **Confluence**

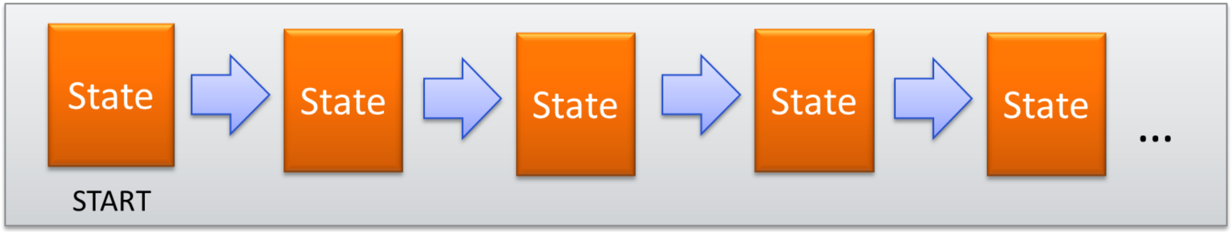


  - Final state must be determined by start state
    - No matter the internal choices (which rule to apply now?)
    - Confluence is important; full determinism is optional
  - Examples
    - ORM: Which root class to transform first? Doesn’t matter.
    - Makefile: Which dirty file to recompile first? Doesn’t matter.
  - No systematic way to guarantee, requires thought

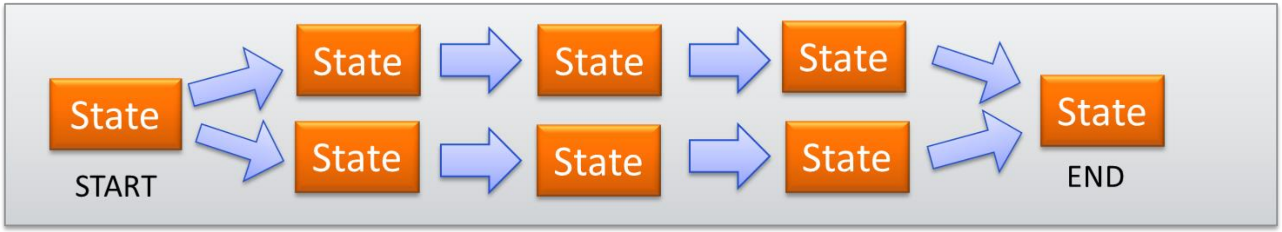
Graph Transformation (GT) Rules

The Motivation for GT

- Writing correct rule-based MTs may be hard
  - Termination

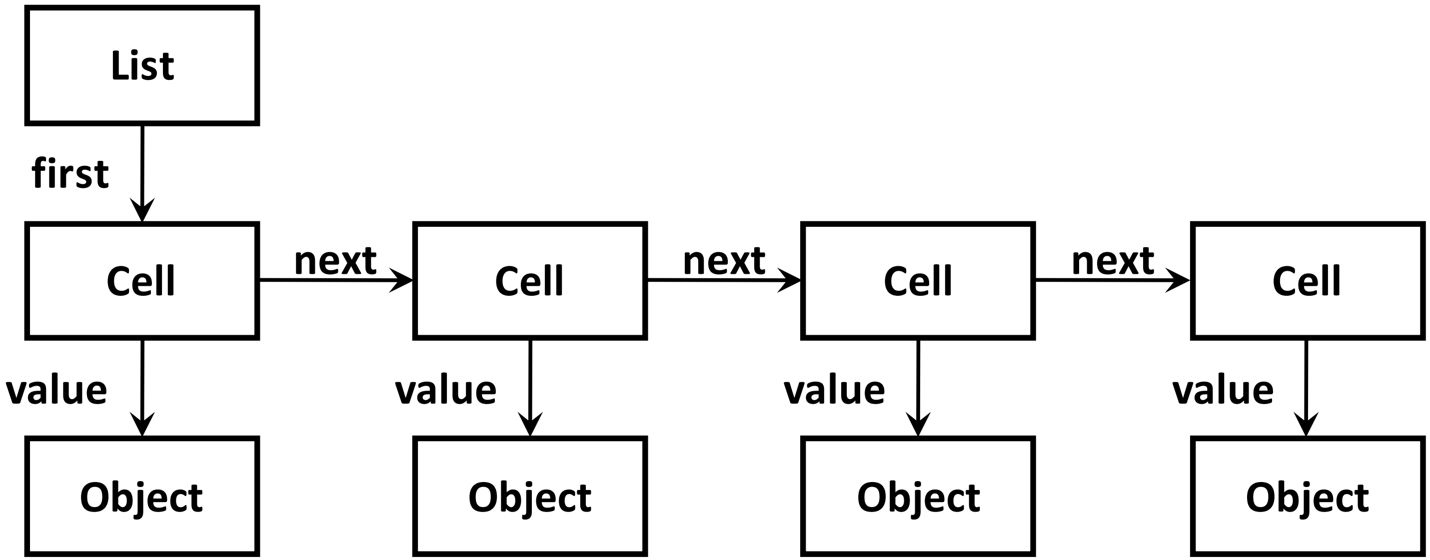


  - Confluence



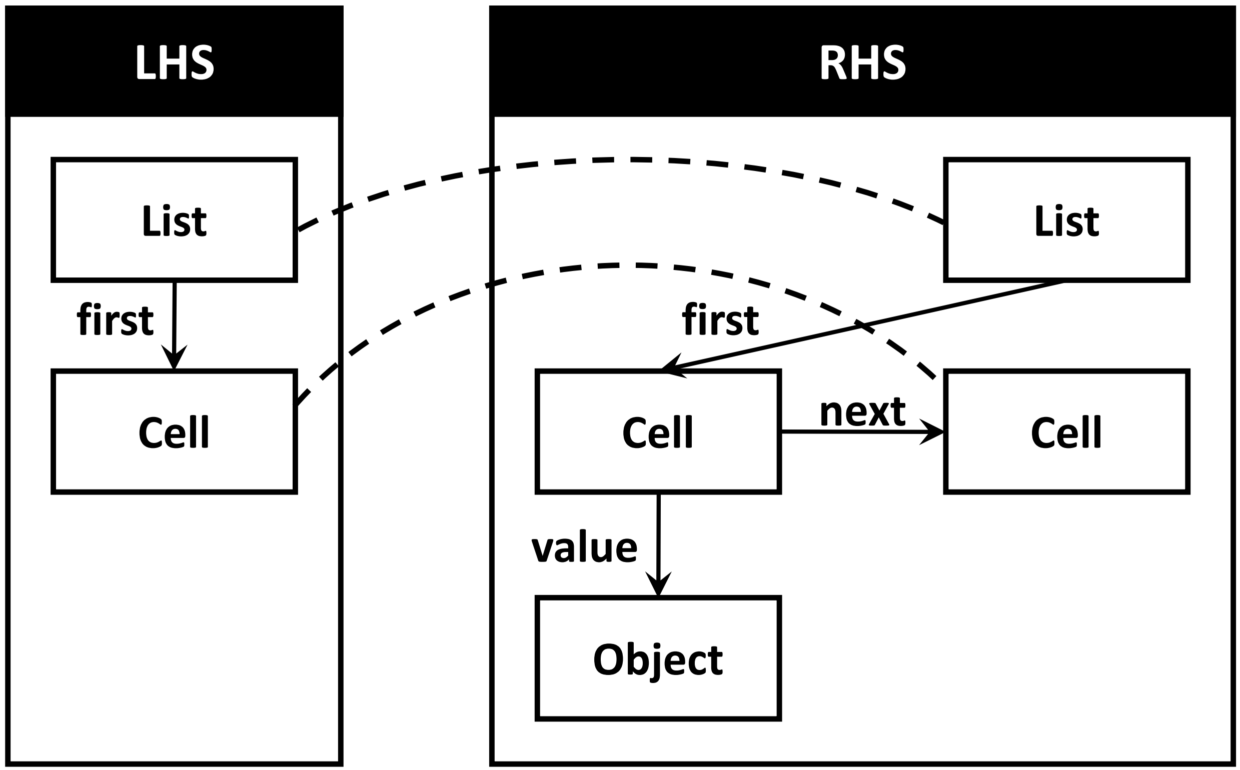
  - …
- Graph Transformation (GT)
  - Formal mathematical model…
  - …to represent MT rules…
  - …and reason about them

Model = Labelled Graph

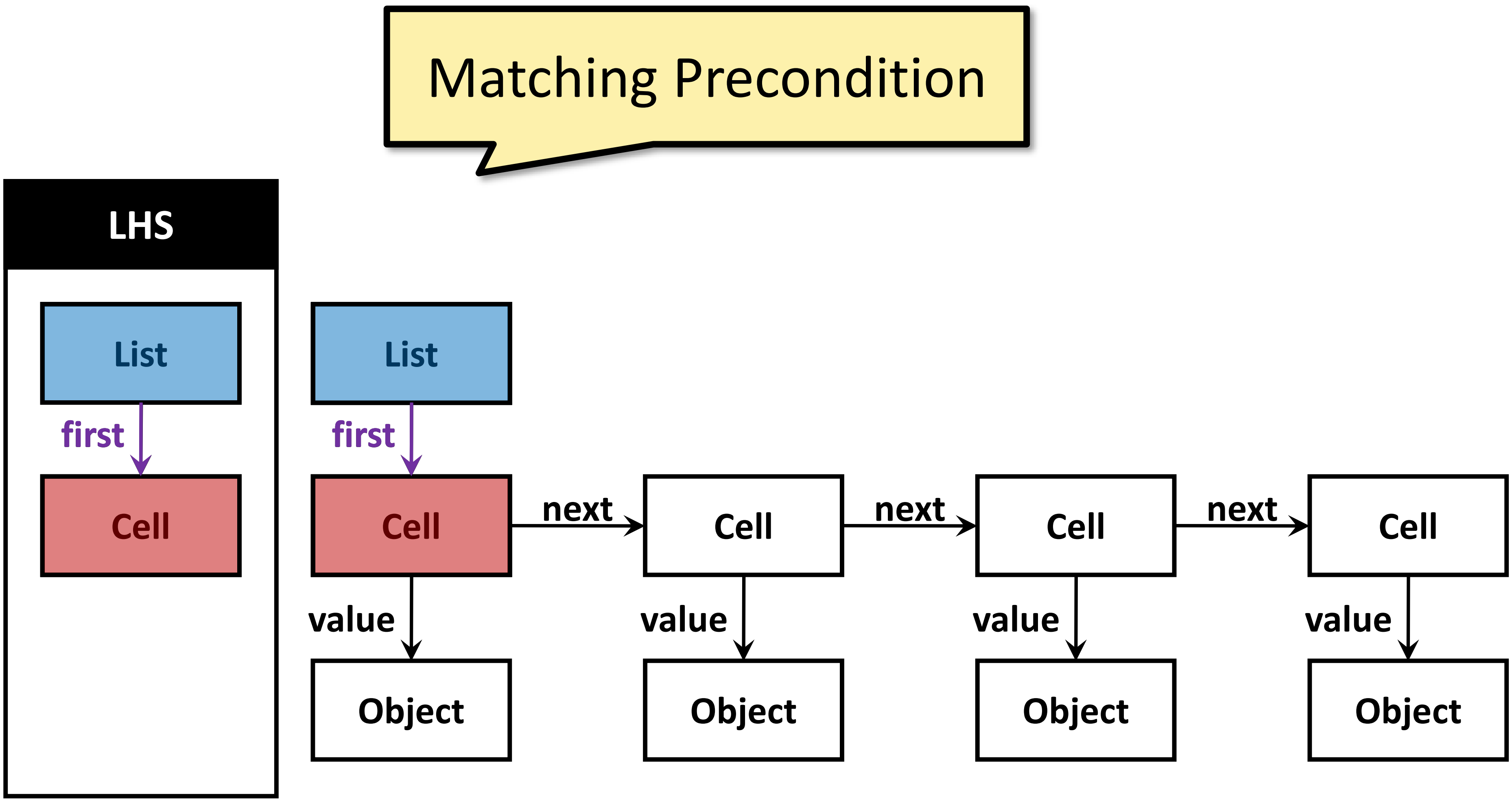


Operation = Graph Transformation

- Graph transformation as graph rewriting rules
- Left Hand Side: Precondition Right Hand Side: Postcondition

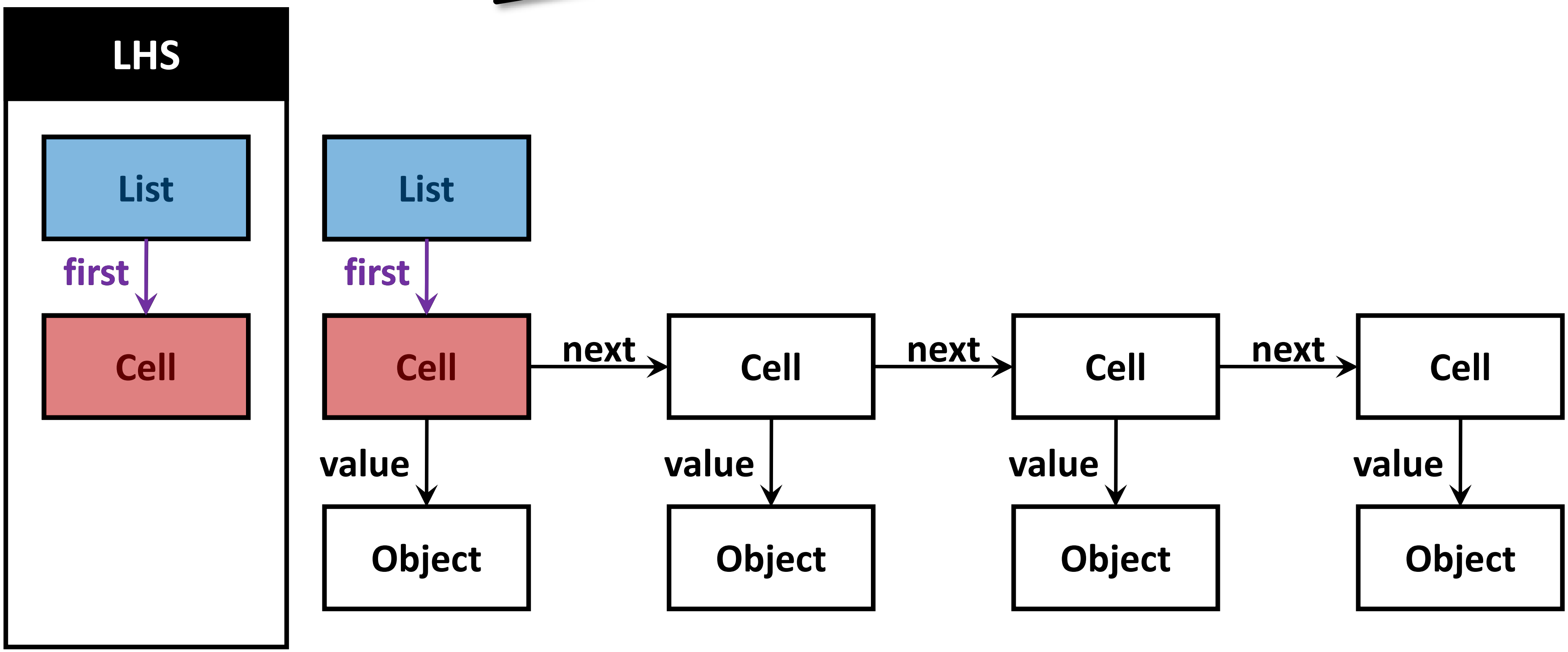


Execution of Graph Transformation Rules



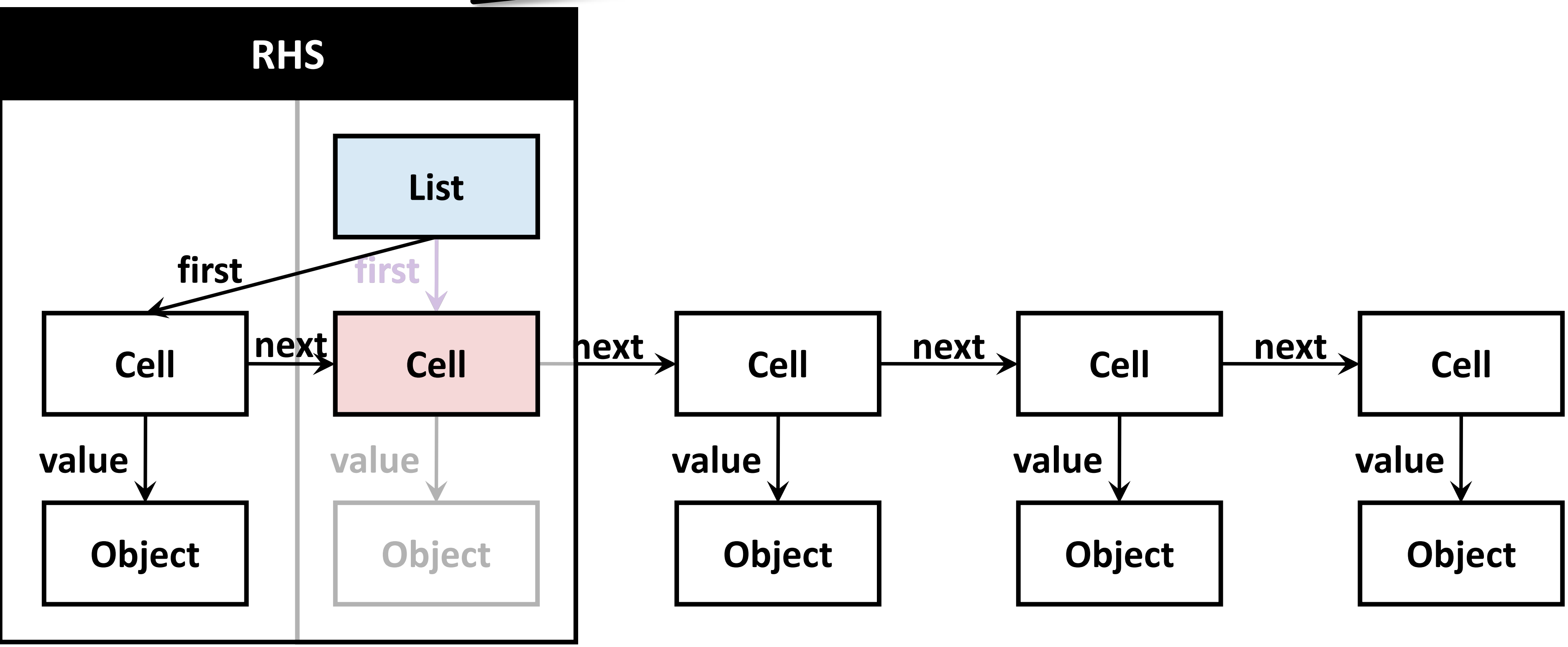
Execution of Graph Transformation Rules

Matching precondition



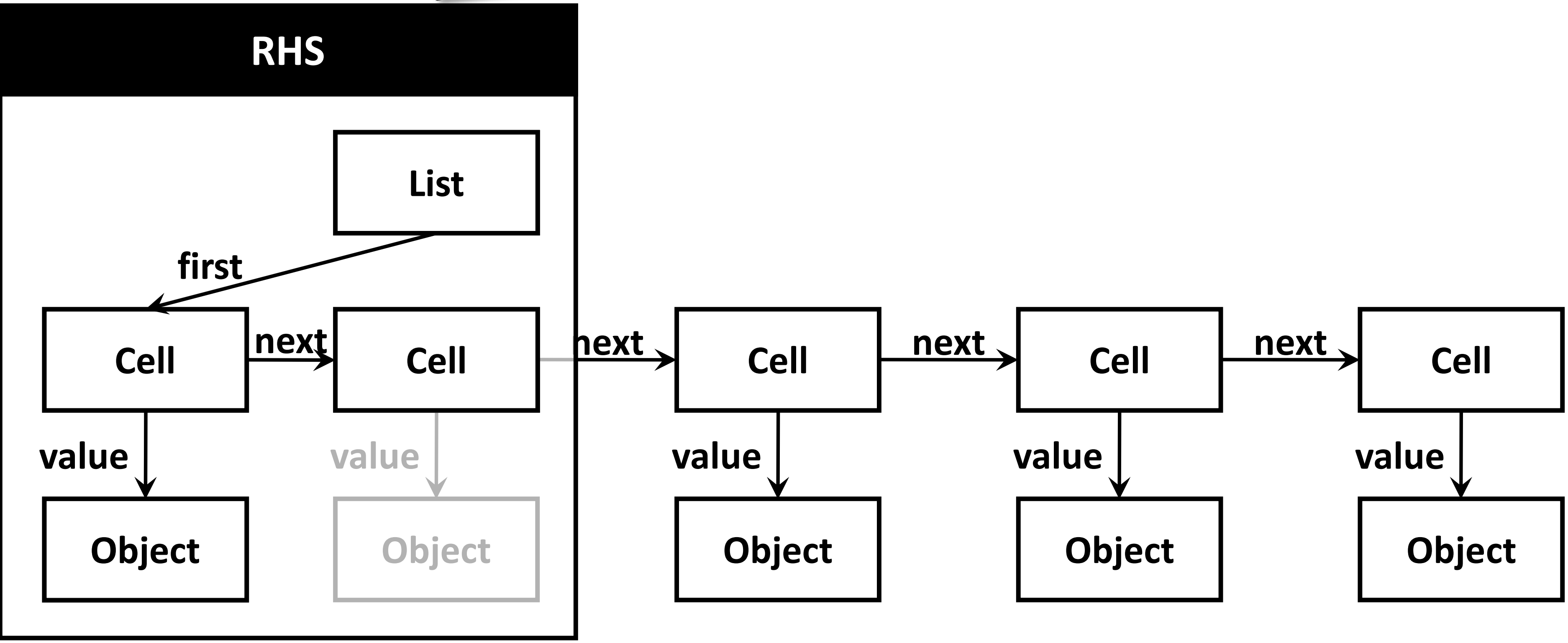
Execution of Graph Transformation Rules

Rewriting the graph by the match

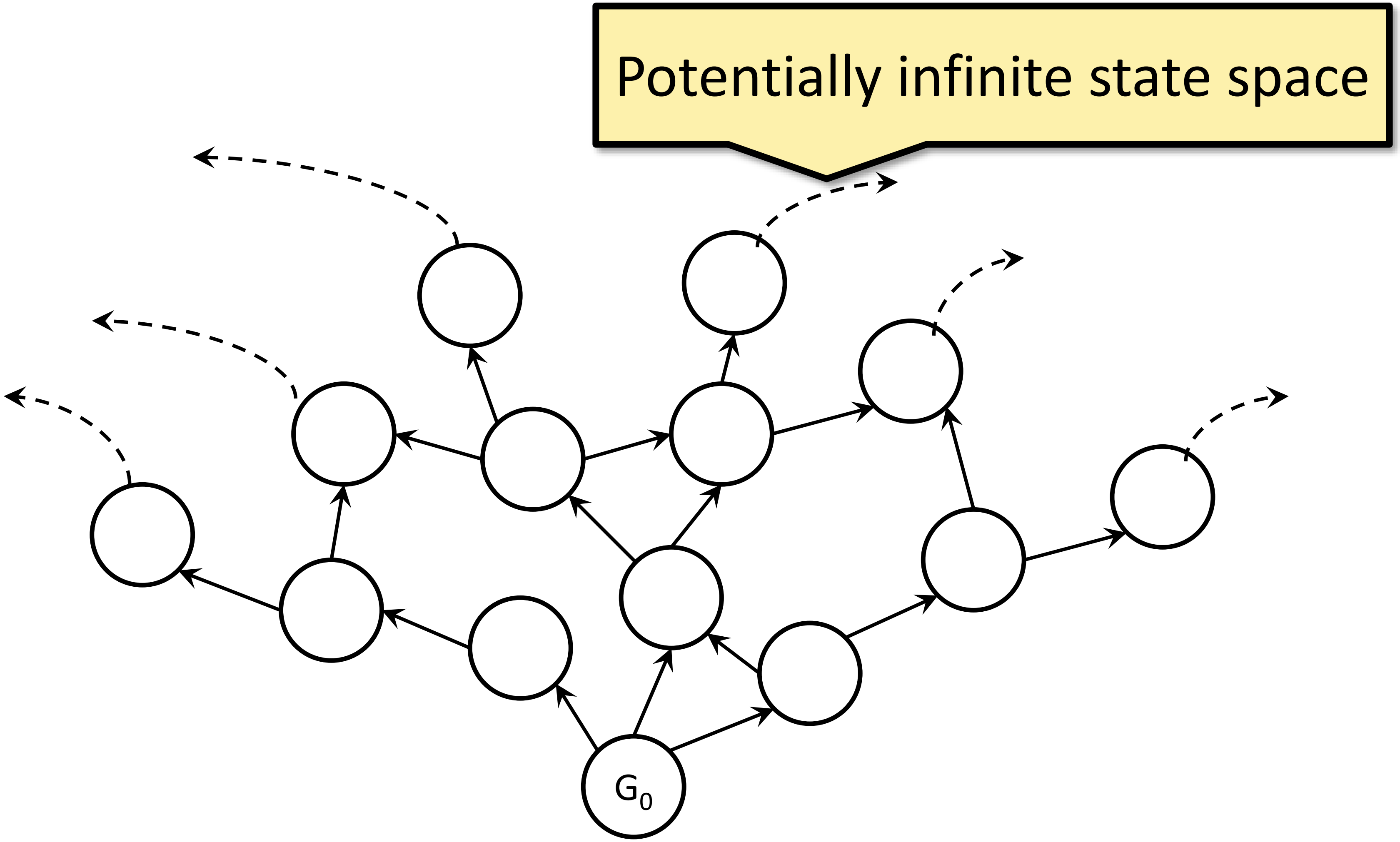


Execution of Graph Transformation Rules

We get a new graph

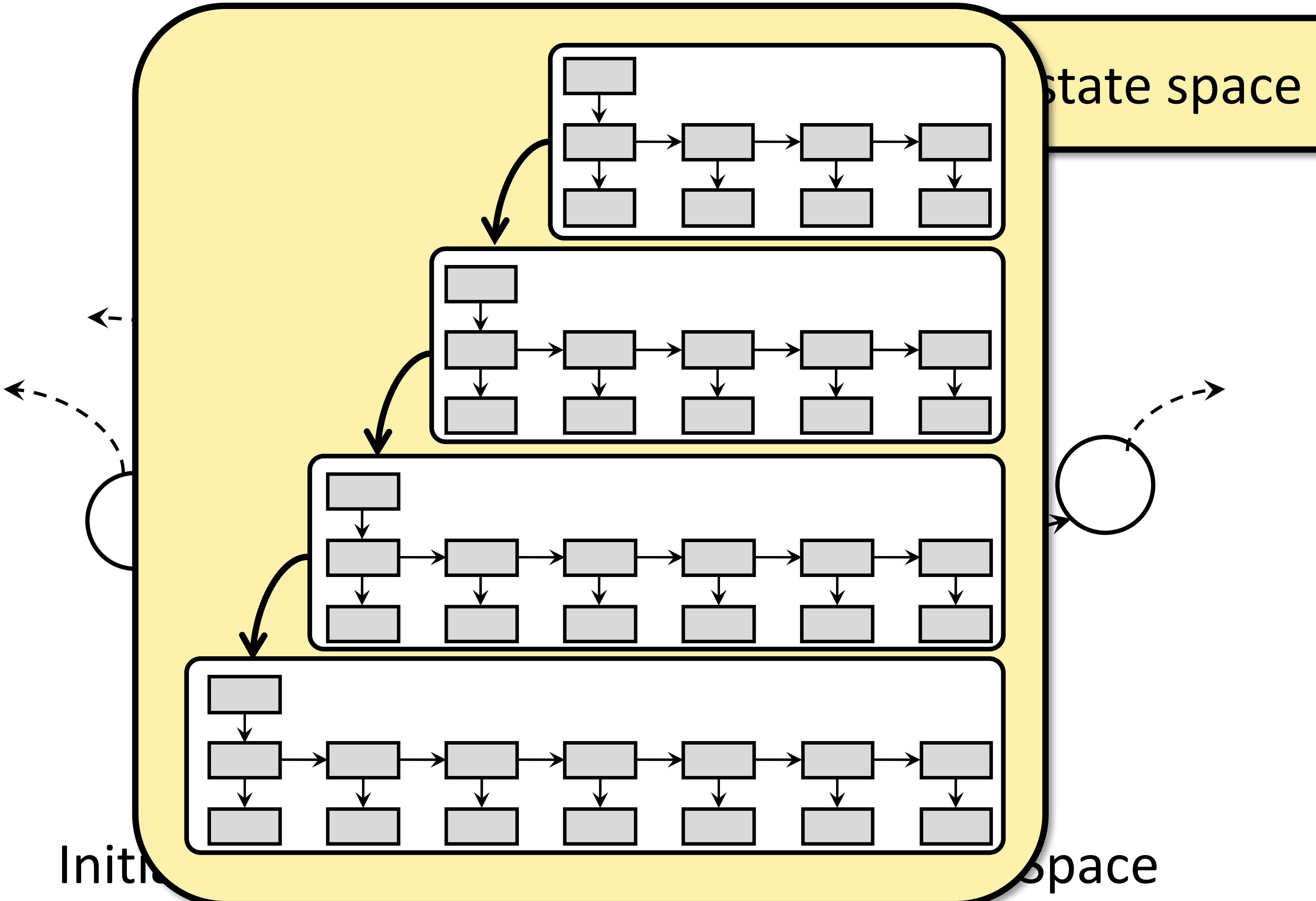


State Space



Initial Graph + Transformations → State Space

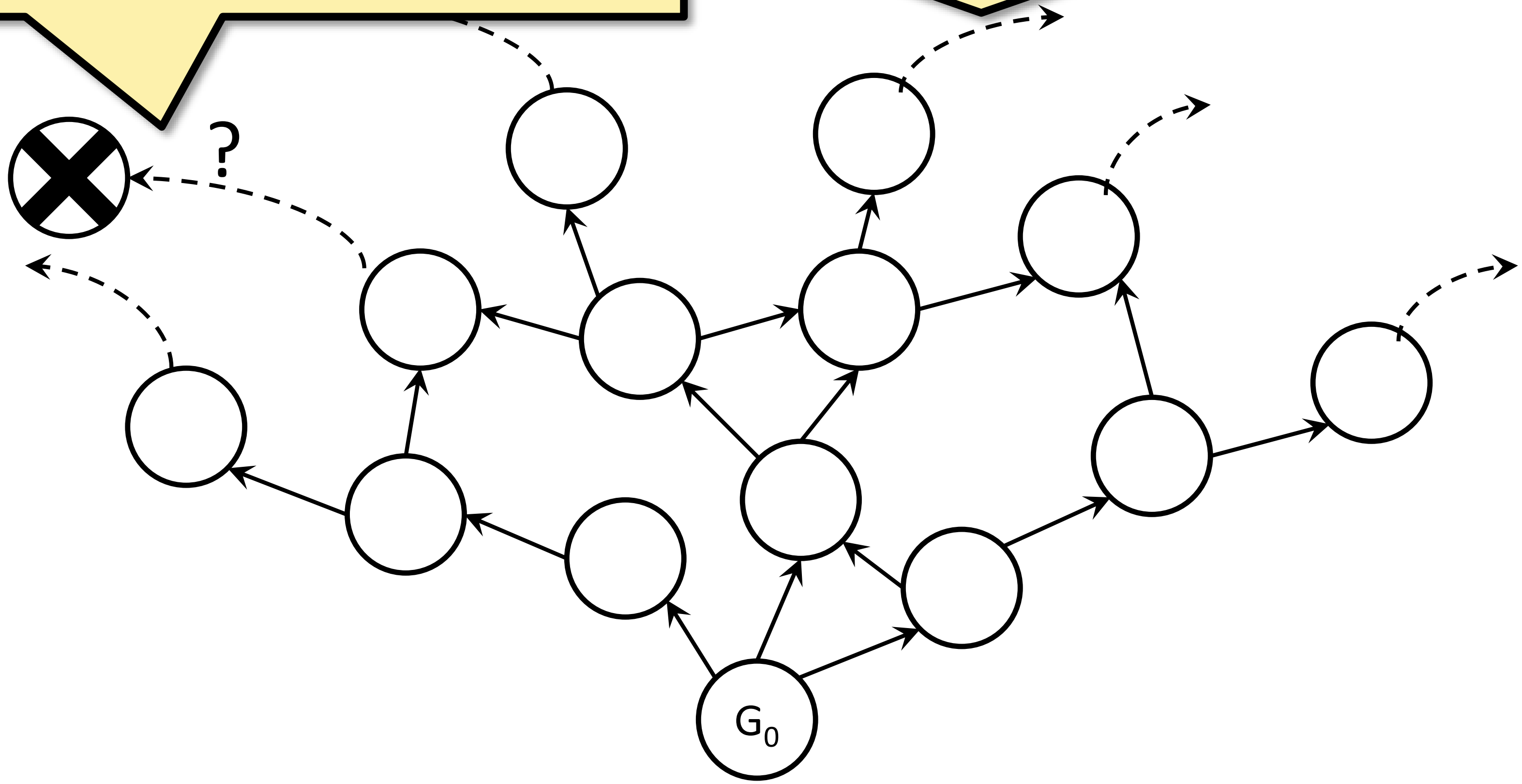
State Space



State Space

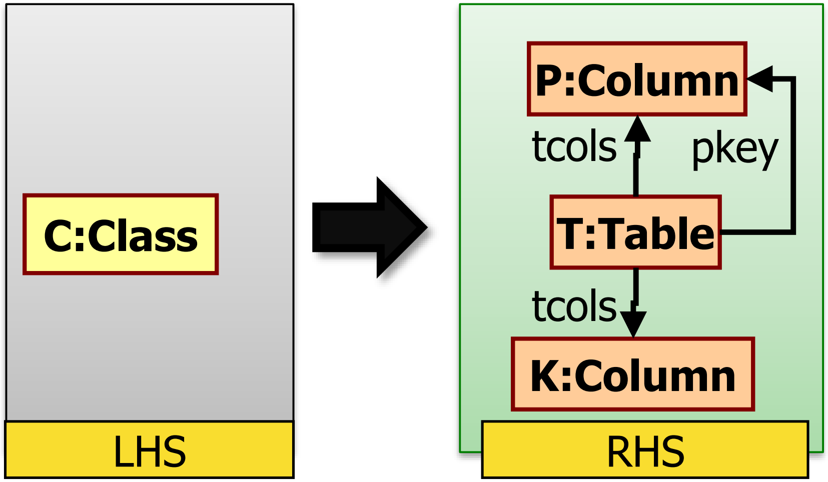
Solutions are in the state space

Potentially infinite state space



Initial Graph + Transformations → State Space

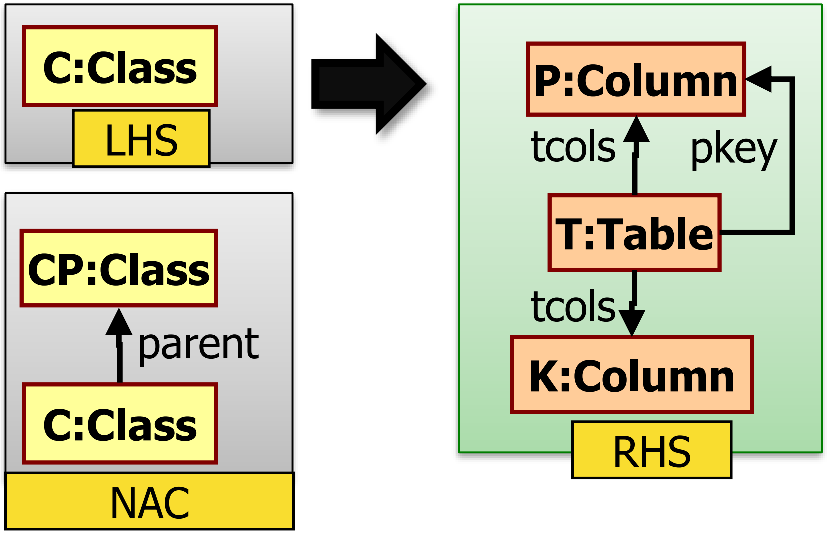
Structure of a GT rule



- **Graph Transformation Rules**
  - **Left hand side** - LHS
    - Graph pattern
    - Precondition for the rule application
  - **Right hand side** - RHS:
    - Graph pattern + LHS mapping
    - Declarative definition of the rule application
      - What we get (and not how we get it)

- **Graph Transformation (GT)**:
  - Declarative and formal paradigm
  - Rule base transformation
  - Match of the LHS ➔ match of the RHS
  - Generalization of Chomsky grammars (hierarchy) (text ➔ graph)

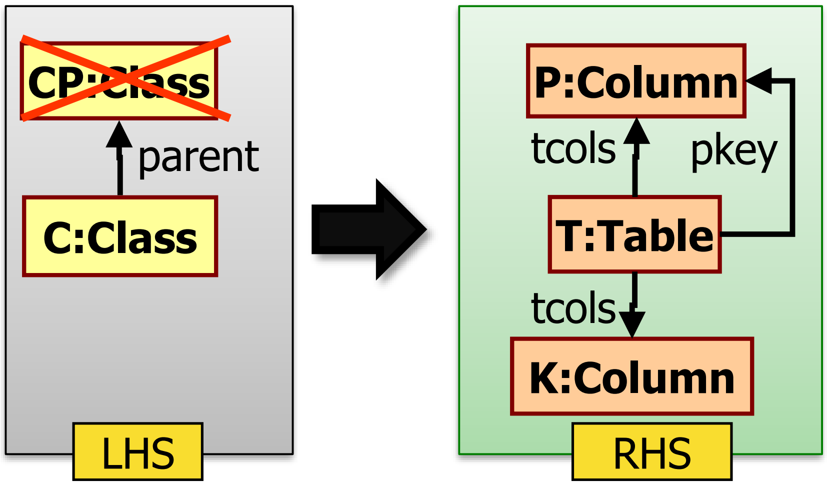
Structure of a GT rule



- **Graph Transformation Rules**
  - **Left hand side** - LHS
    - Graph pattern
    - Precondition for the rule application
  - **Right hand side** - RHS:
    - Graph pattern + LHS mapping
    - Declarative definition of the rule application
      - What we get (and not how we get it)
  - **Negative Application Condition**(NAC):
    - Graph pattern + LHS mapping
    - Negative precondition of the rule application
    - If it can be made true➔ the rule cannot be applied
    - Multiple NACs ➔ only one is true ➔ rule cannot be applied

- **Graph Transformation (GT)**:
  - Declarative and formal paradigm
  - Rule base transformation
  - Match of the LHS➔ Image of the RHS
  - Generalization of Chomsky grammars (hierarchy) (text ➔ graph)

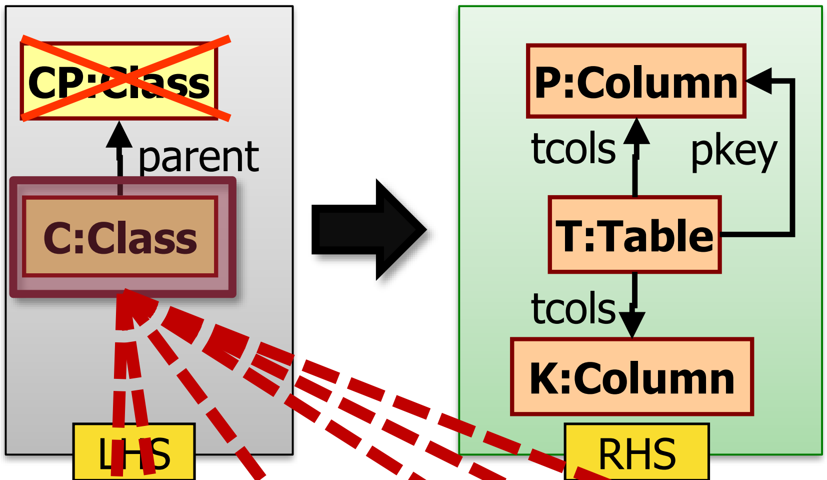
Structure of a GT rule



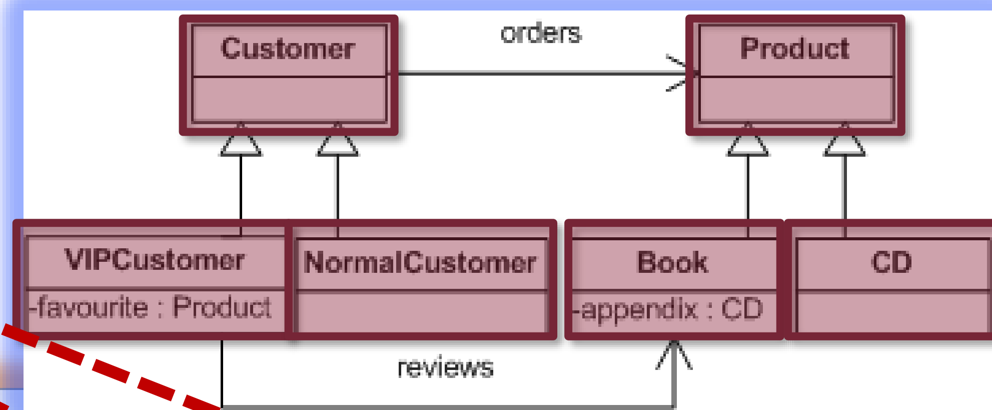
- **Graph Transformation Rules**
  - **Left hand side** - LHS
    - Graph pattern
    - Precondition for the rule application
  - **Right hand side** - RHS:
    - Graph pattern + LHS mapping
    - Declarative definition of the rule application
      - What we get (and not how we get it)
  - **Negative Application Condition**(NAC):
    - Graph pattern + LHS mapping
    - Negative precondition of the rule application
    - If it can be made true➔ the rule cannot be applied
    - Multiple NACs ➔ only one is true ➔ rule cannot be applied

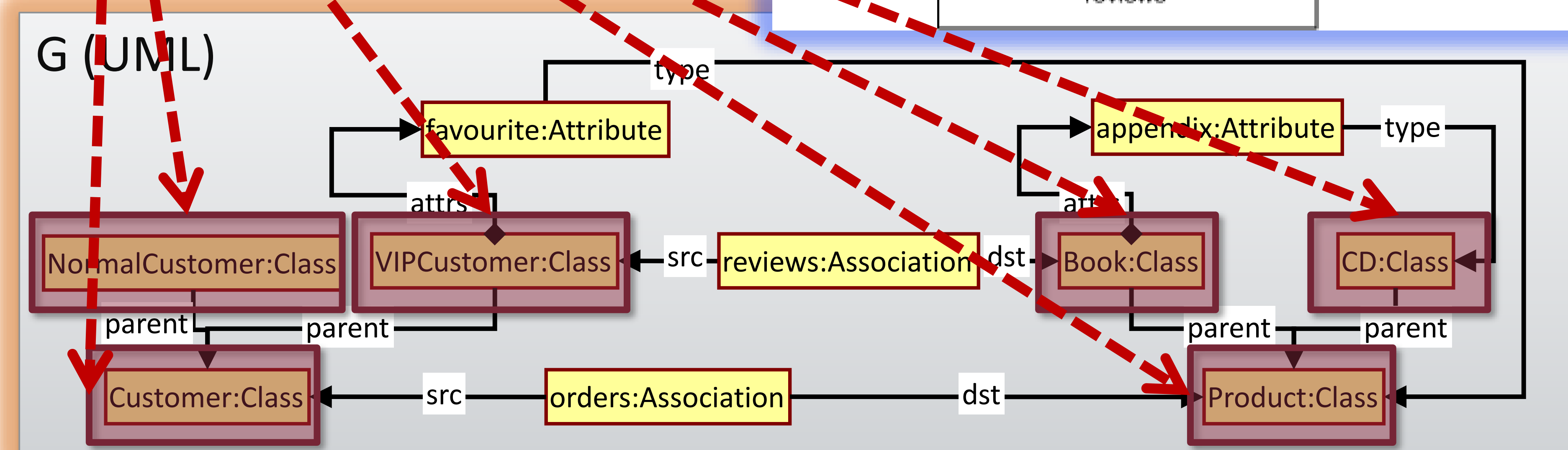
- **Graph Transformation (GT)**:
  - Declarative and formal paradigm
  - Rule base transformation
  - Match of the LHS➔ Image of the RHS
  - Generalization of Chomsky grammars (hierarchy) (text ➔ graph)

Application of GT rules



1. **Graph pattern matching**
   - Match of the LHS pattern in the underlying model
   - match ***m***: LHS ➔ G mapping

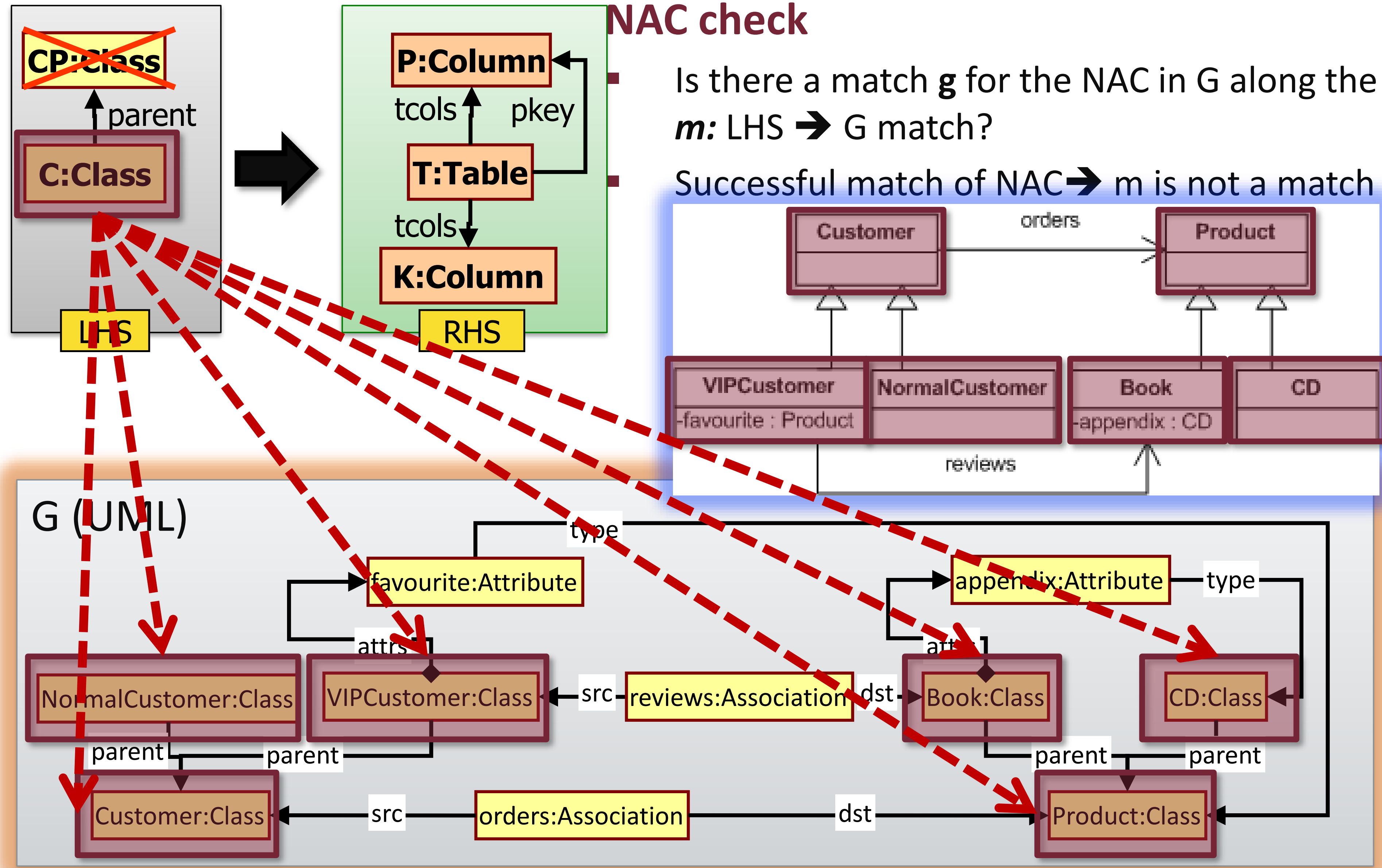




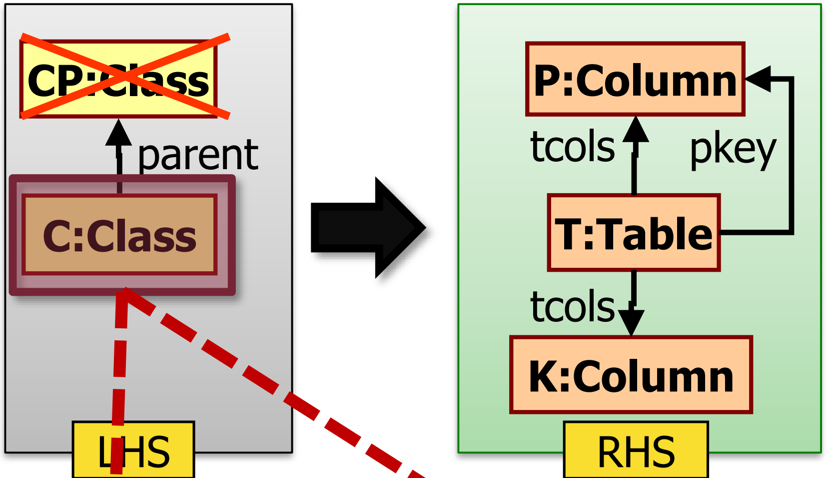
Application of GT rules

NAC check

- Is there a match **g** for the NAC in G along the ***m:*** LHS ➔ G match?
- Successful match of NAC➔ m is not a match

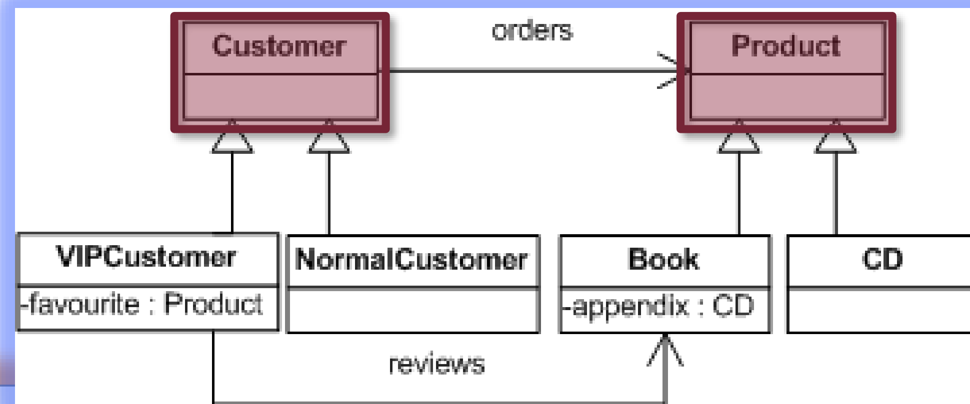


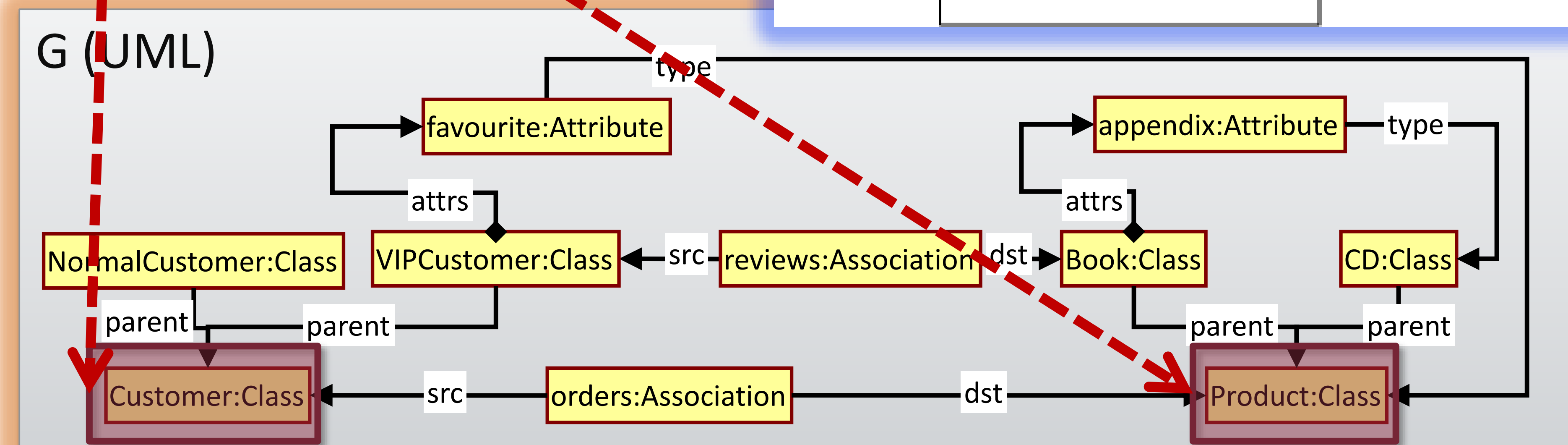
Application of GT rules



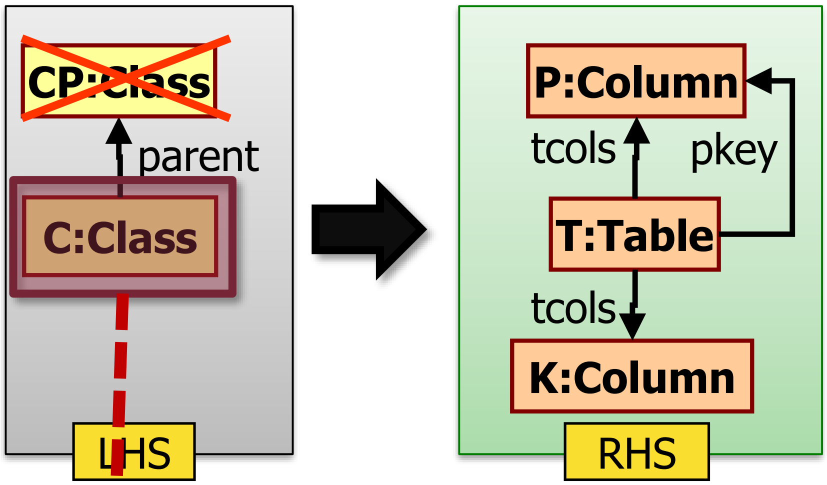
3. Non-deteministic selection

- Random selection of a match (if more than one)
- No match ➔ rule fails



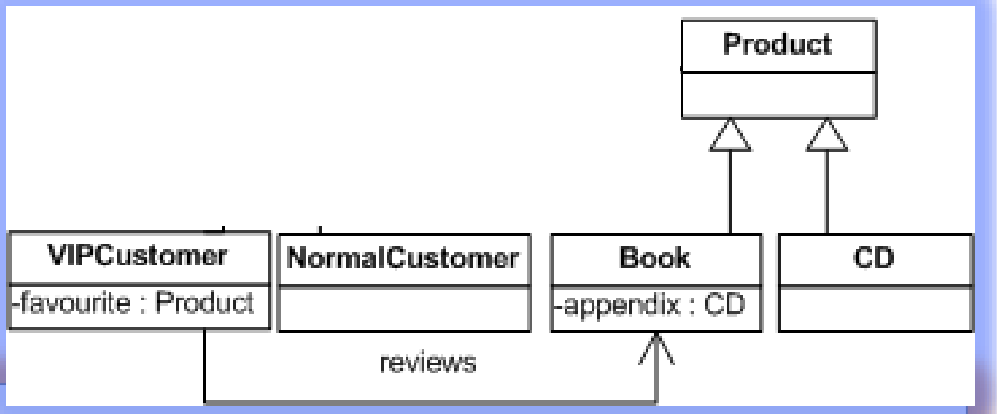


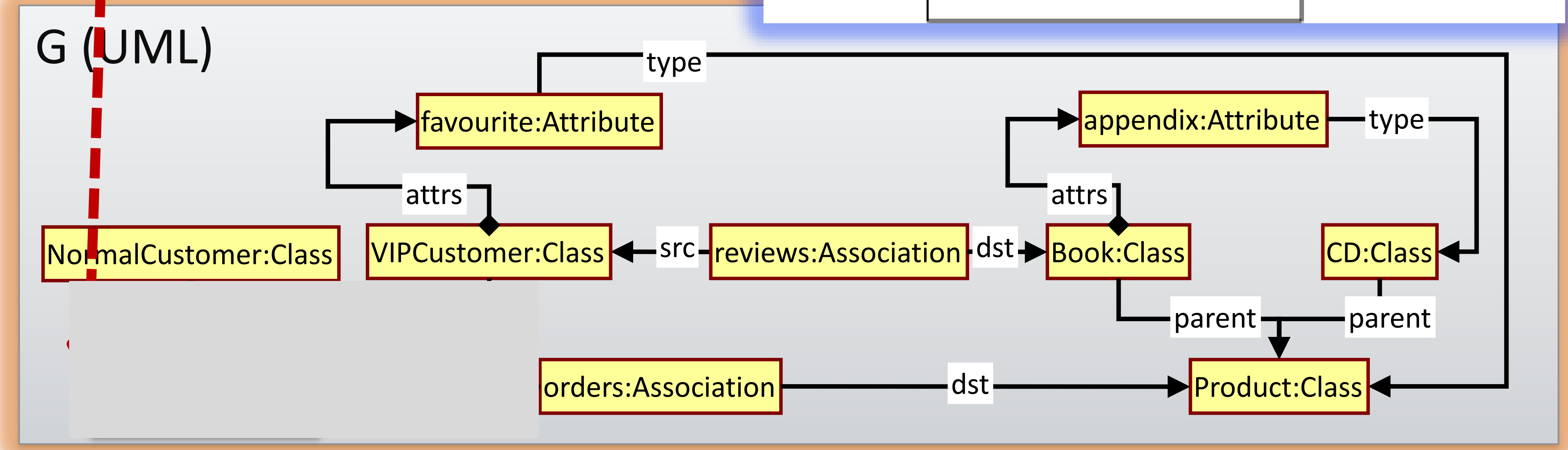
Application of GT rules



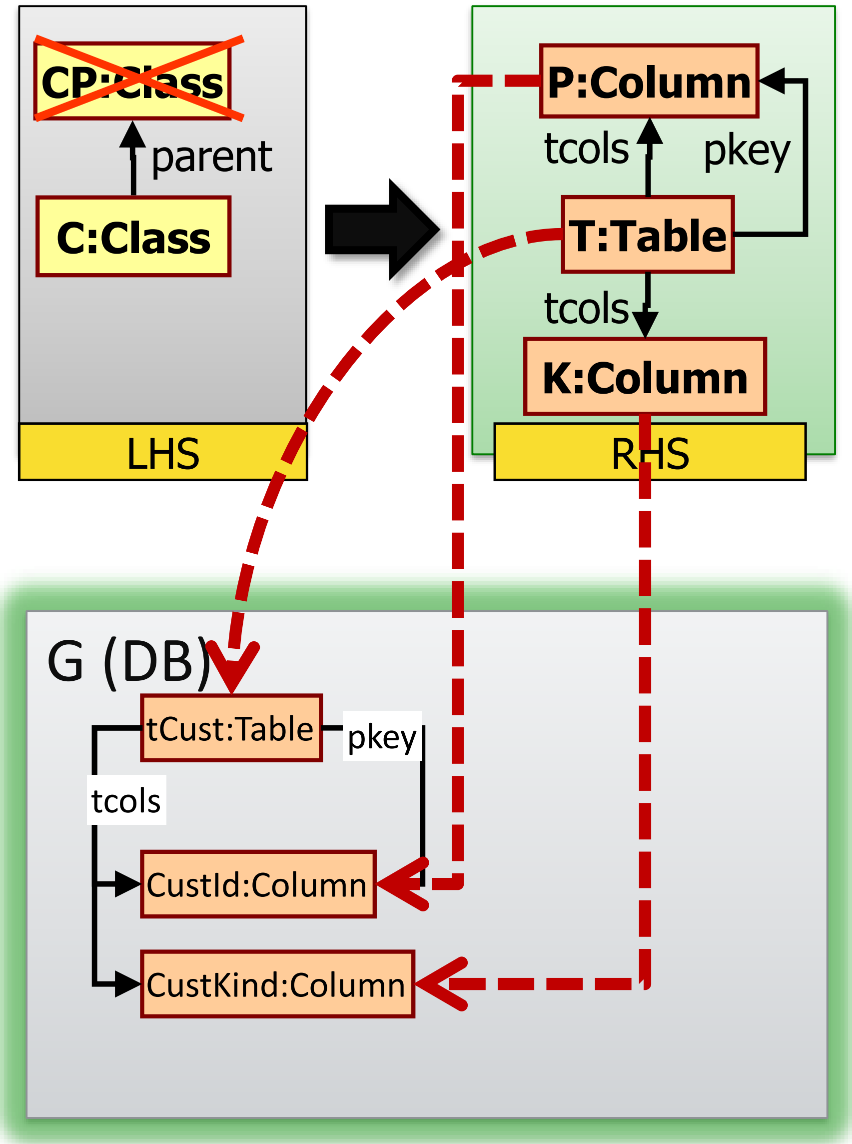
4. Deletion

- Deletion of LHS \ RHS from G
- In LHS yes, in RHS no



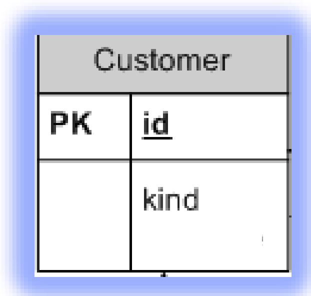


Application of GT rules



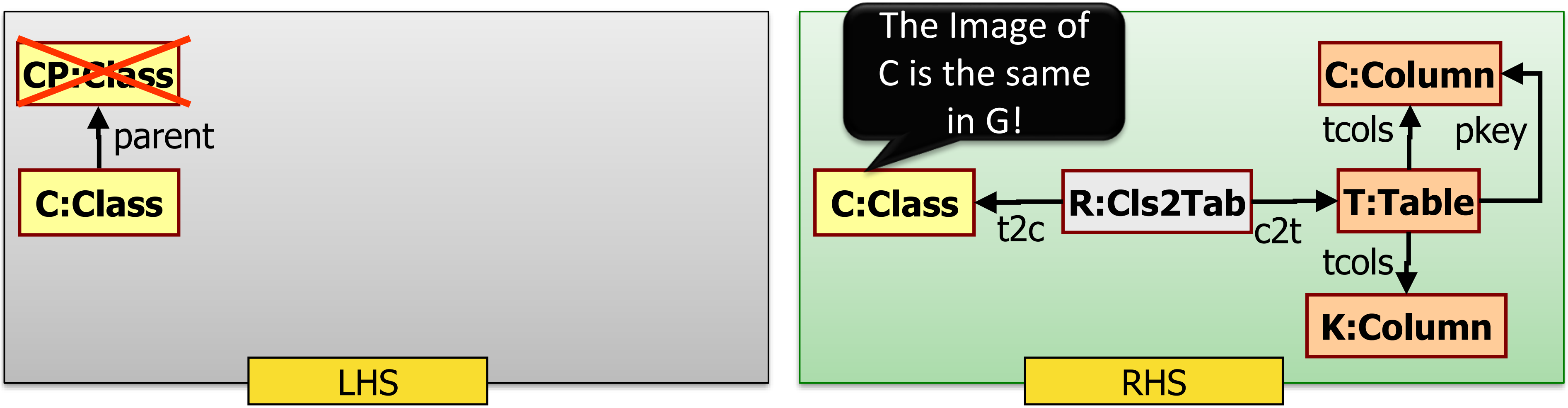
5. Creation (and binding)

- Creation of RHS \ LHS in G with their corresponding relations
- Output:
  a „match" of RHS in G

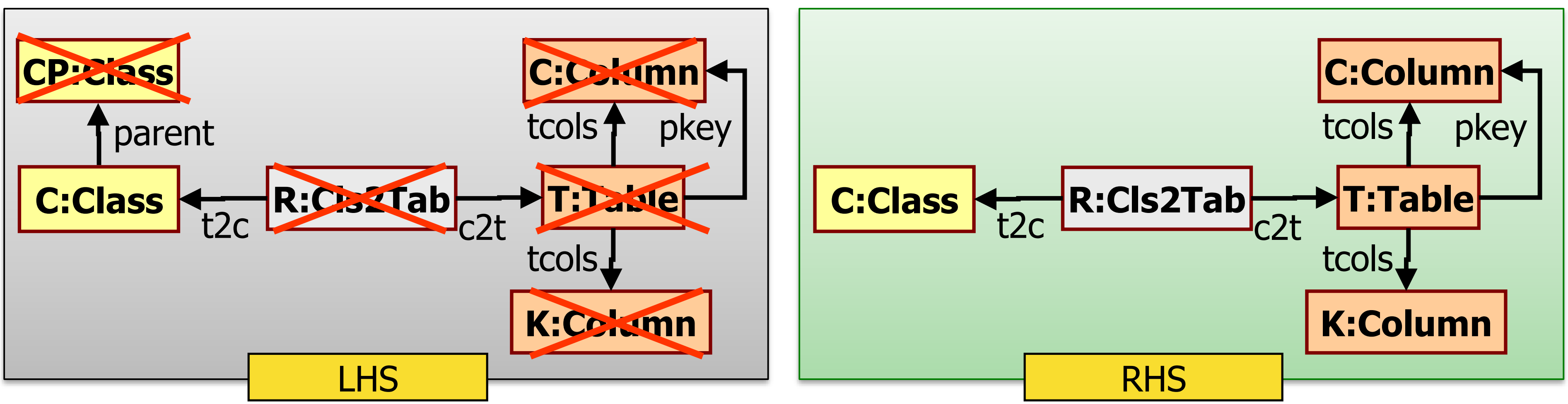


Typical problems...

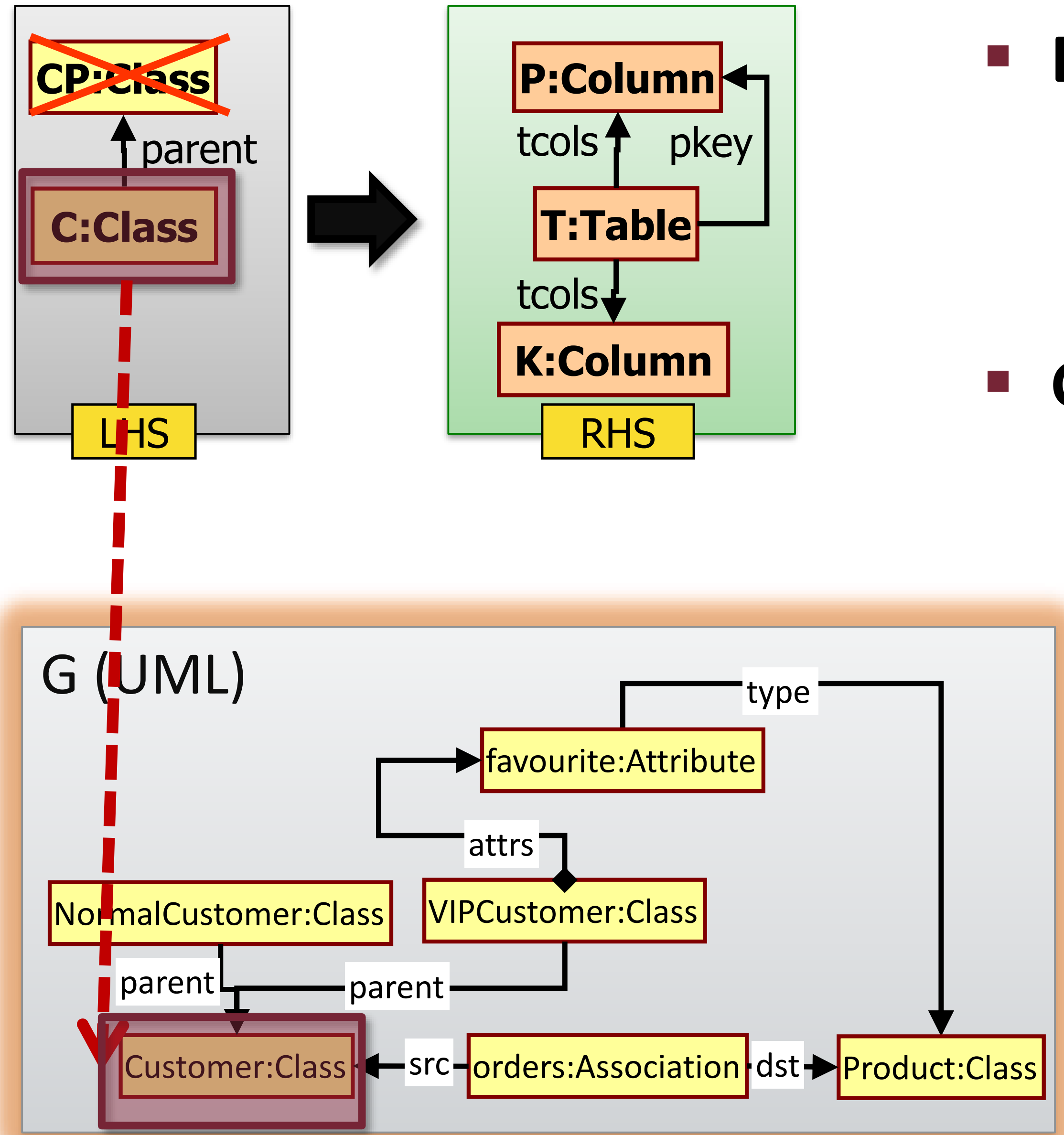
1) Saving the source model, traceability



2) Application of the same rule along the same match

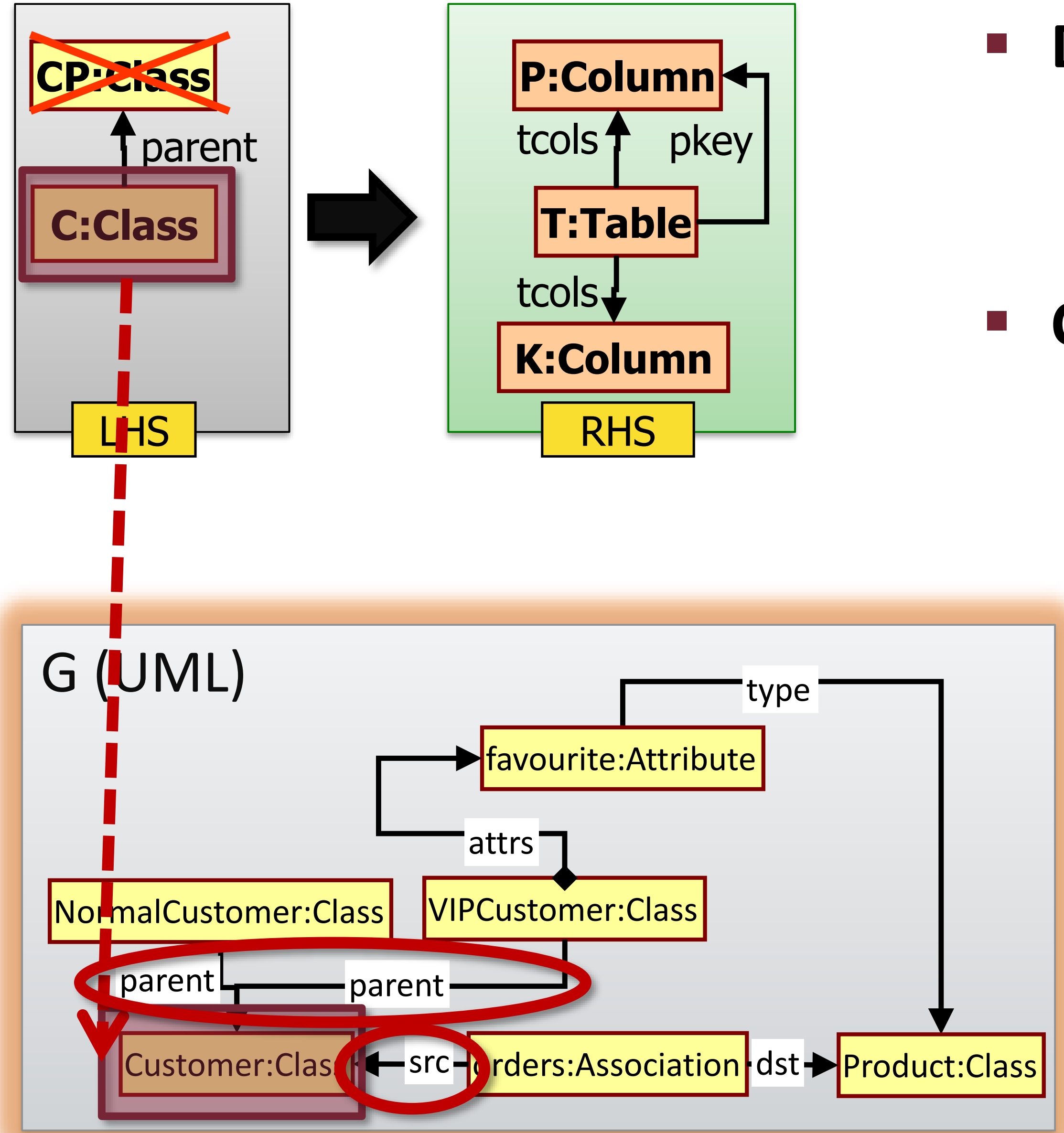


Semantics : Handling of Dangling edges



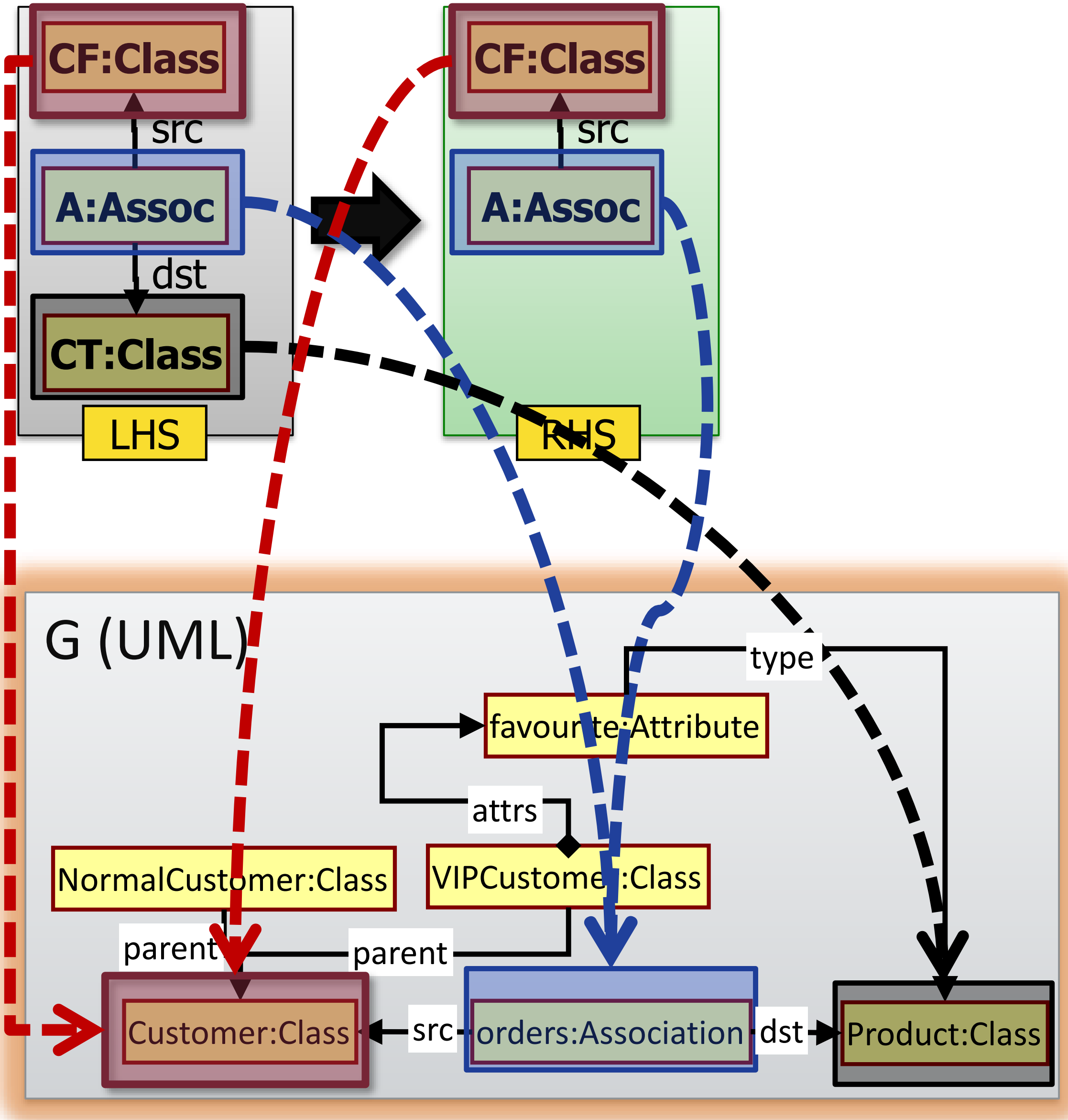
- **Dangling edges**:
  - Delete a node
    - What to do with the dangling edges?
- **Greedy approach**
  - Delete all dangling edges
  - **Pro**:
    - Intuitive for engineers
    - Easy to implement
  - **Con**:
    - Verification is hard (side effect of rules)

Semantics : Handling of Dangling edges



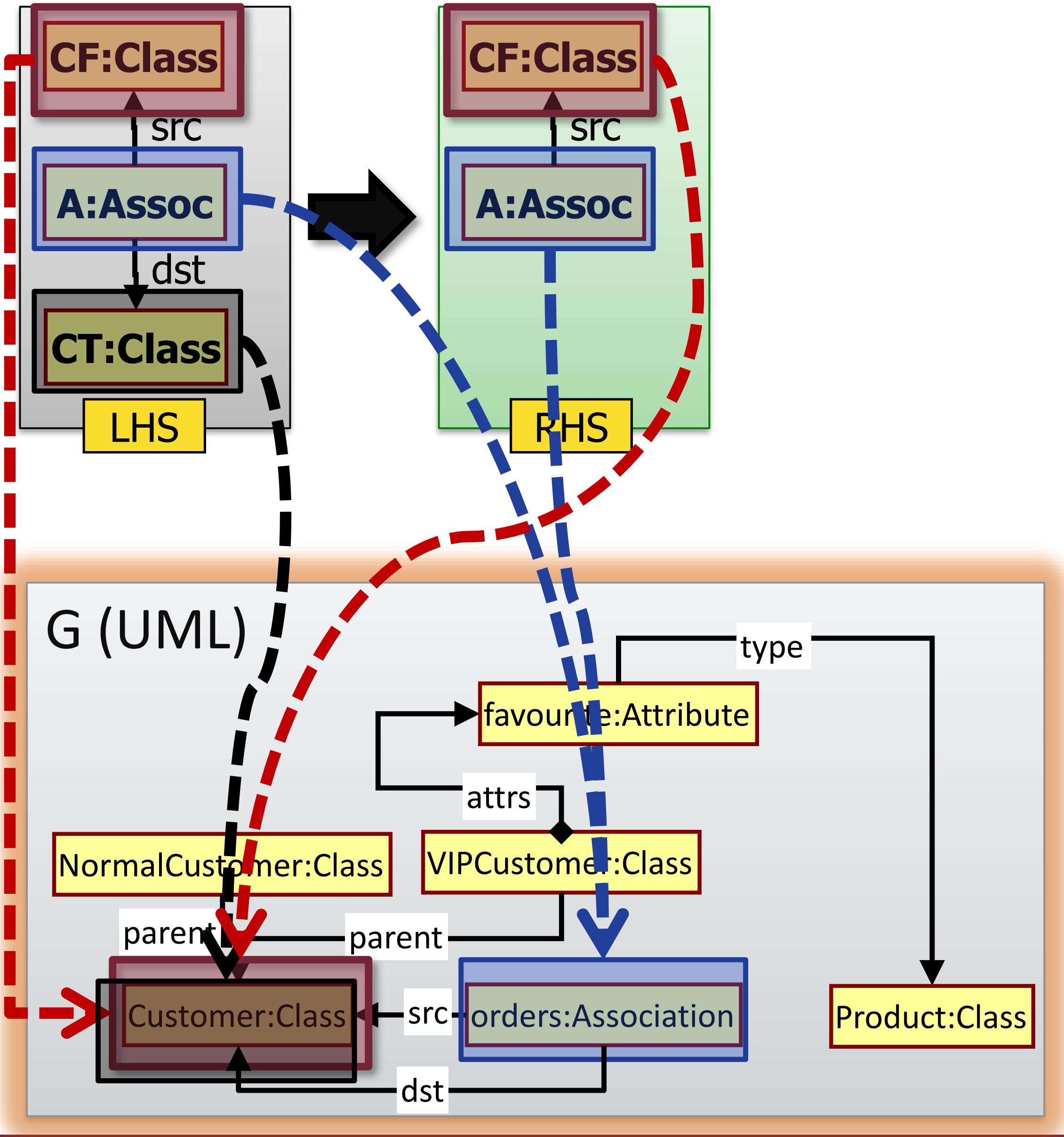
- **Dangling edges**:
  - Delete a node
    - What to do with the dangling edges?
- **Conservative approach**
  - The rule cannot be applied if it would produce a dangling edge
  - **Pro**:
    - Side effect free rules
    - Helps verification
  - **Con**:
    - Harder to implement
    - What is its meaning for engineers (not mathematicans)

Semantics: Injective matching



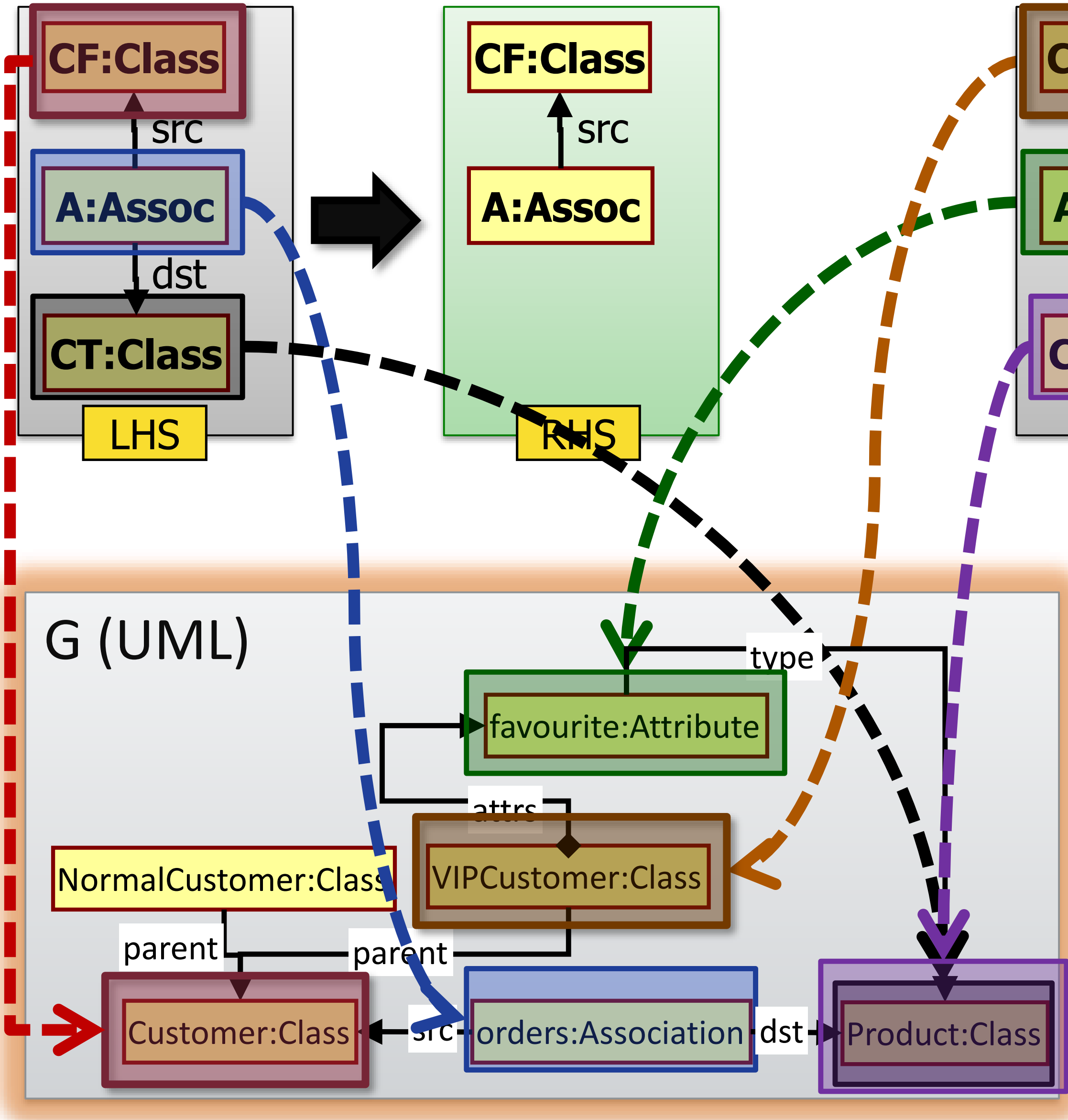
- **Injective matching („kisajátító”)**
  - For all nodes in the LHS➔ separate nodes are matched in G
- **Pro**:
  - Intuitive for engineers
- **Con**:
  - Verbose specification of rules (many alternate subrules)

Semantics: Non-injective matching



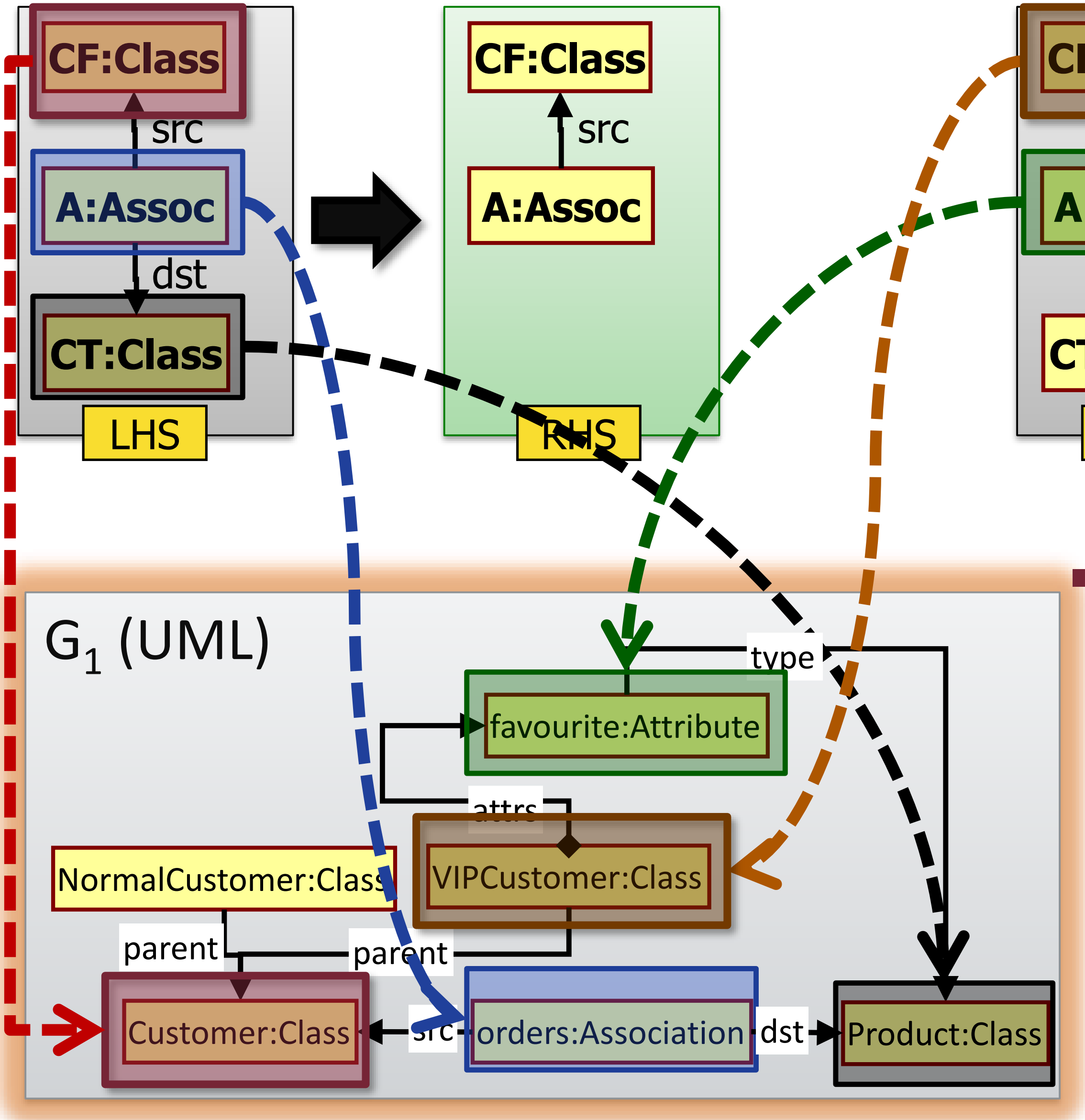
- **Non-Injective matching („közösködő”)**
  - For multiple nodes in the LHS ➔ the same node can be matched in G
- **Con**:
  - Contradictionary specification for a node
    - For **CF** : keep it
    - For **CT** : delete
- **Solution**:
  - Nodes to be deleted in LHS are matched with injective semantics

Conflict / Parallel independence



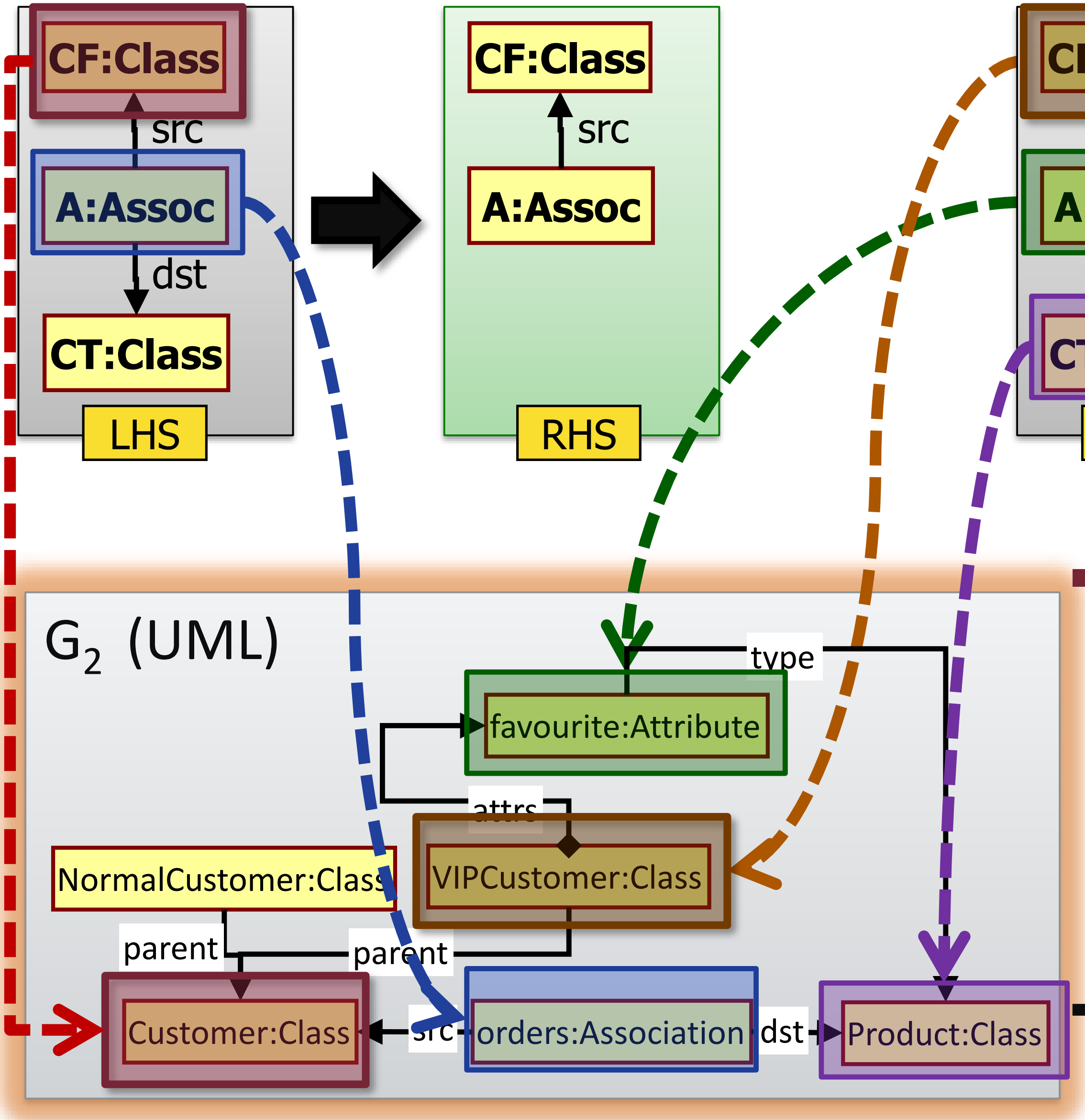
- **Parallel independence** (between two rule applications)
  - Neither prevents the application of the other
- **Conflict** (between two rule apps)
  - If they are not parallel independent
- **Parallel independence** (between two rules)
  - Any two of their rule application are parallel independent

Sequential independence



- **Sequential independence** (two following rule applications)
  - Their order can be swapped without any effect on their final result

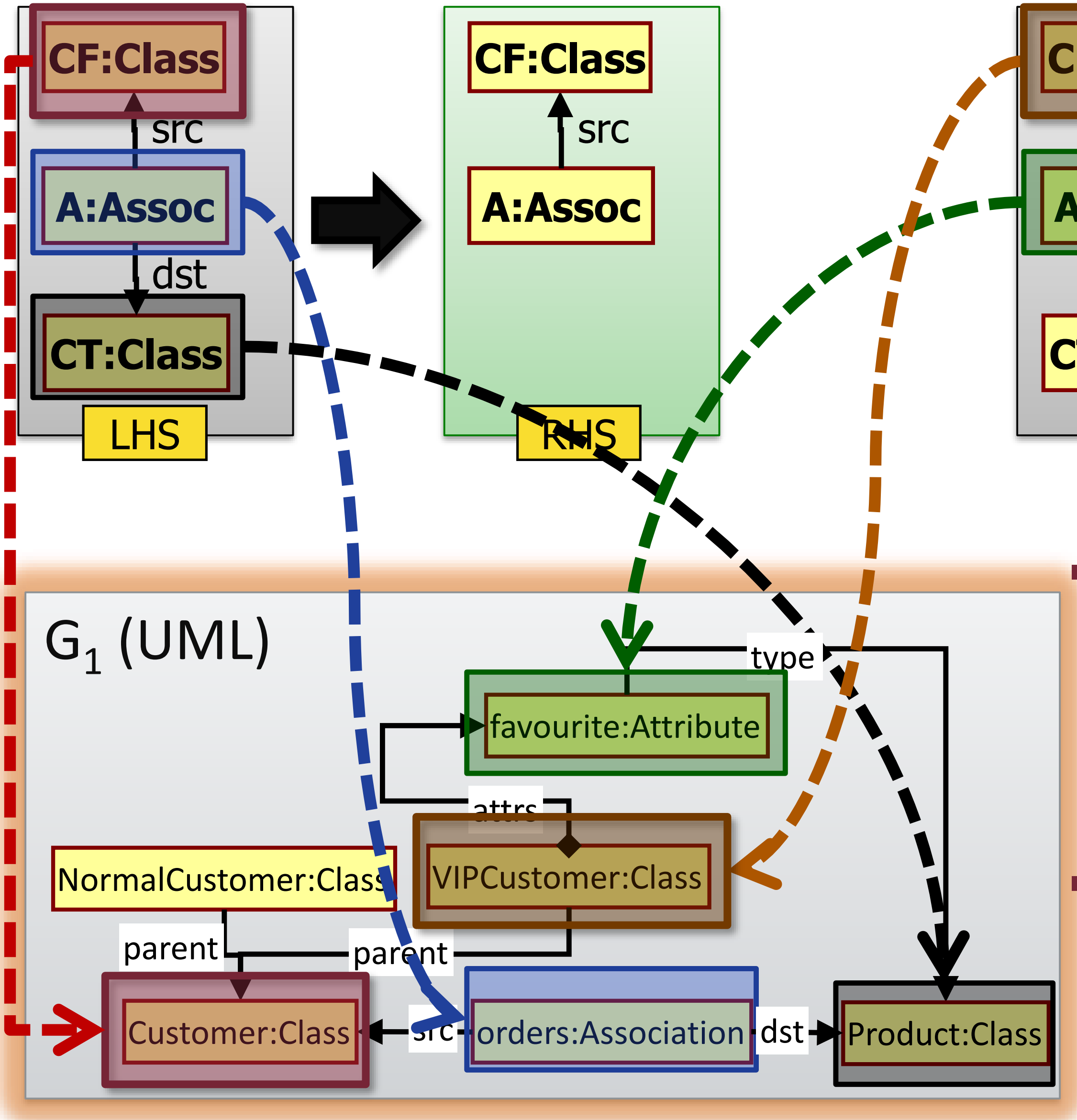
Sequential independence



- **Sequential independence** (two following rule applications)
  - Their order can be swapped without any effect on their final result

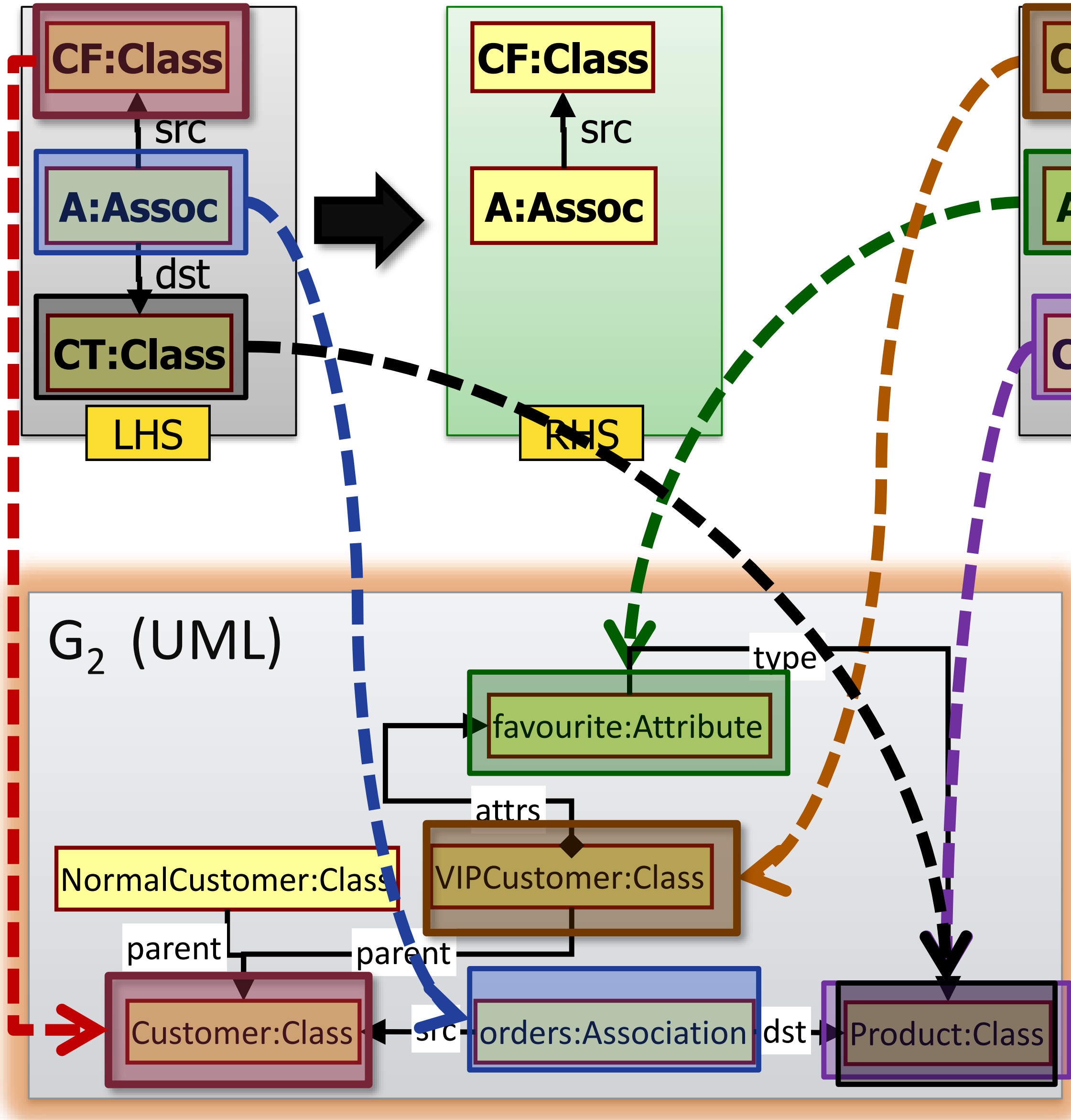
→ Example

Causal dependence I.



- **Sequential independence** (two following rule applications)
  - Their order can be swapped without any effect on their final result
- **Causally dependent** (two following rule applications)
  - If they are not serial independent

Causally dependence II.

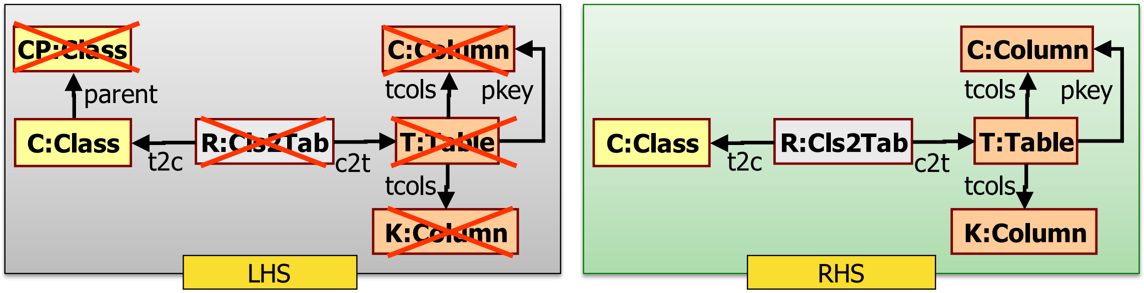


- **Serial independence** (two following rule applications)
  - Their order can be swapped without any effect on their final result
- **Causally dependent** (two following rule applications)
  - If they are not serial independent

➔ Example

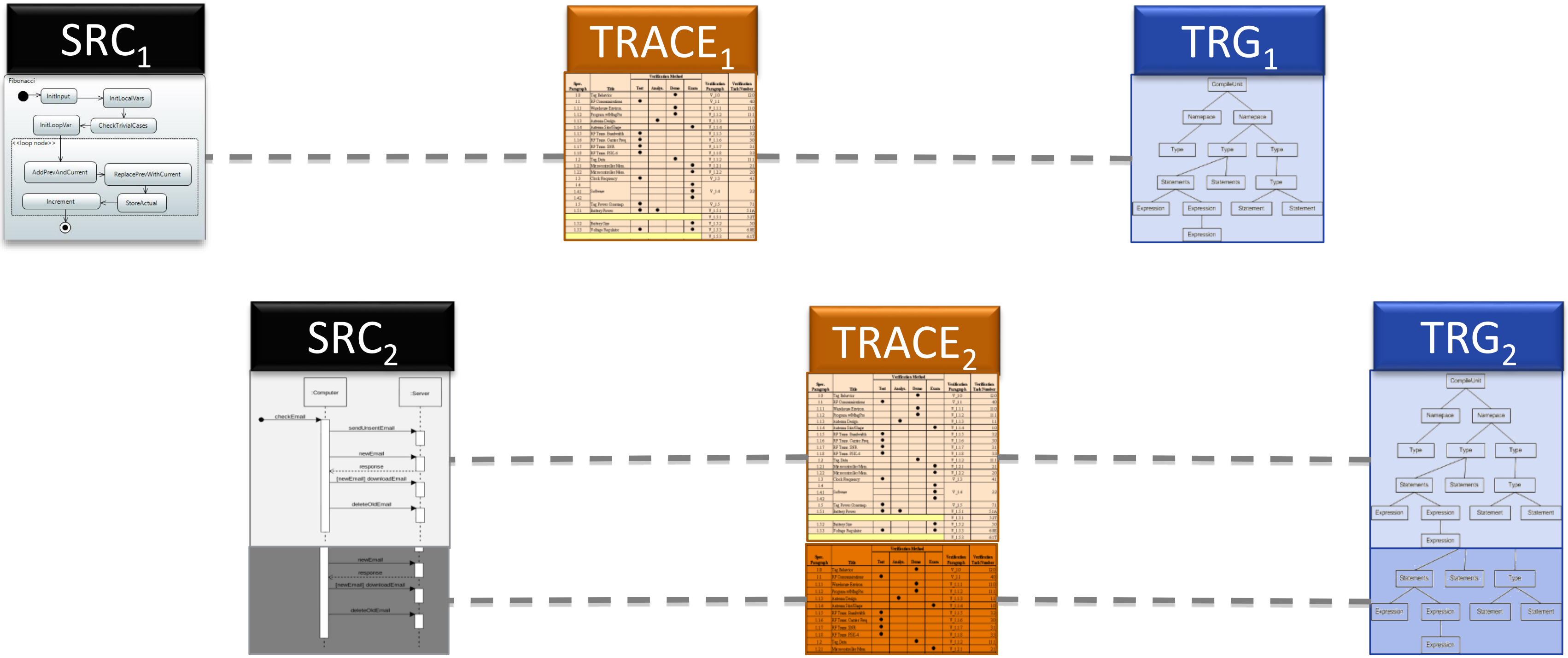
Summary

- **Graphtransformation,**
  as a modeltransformation paradigm
  - Rule and pattern based formal specification
  - Querying and manipulating graph based models
  - Intuitive graph based specification
- **Structure**
  - LHS graph pattern: precondition
  - RHS graph pattern: postcondition
  - NAC: negative condition
- **Rule application**
  - Graph pattern matching
  - Deletition + Creation
  - Dangling edges and injectivity
  - Affect of multiple rule application (conflicts and causality)



Incrementality in model transformations

No Incrementality: Batch Transformations

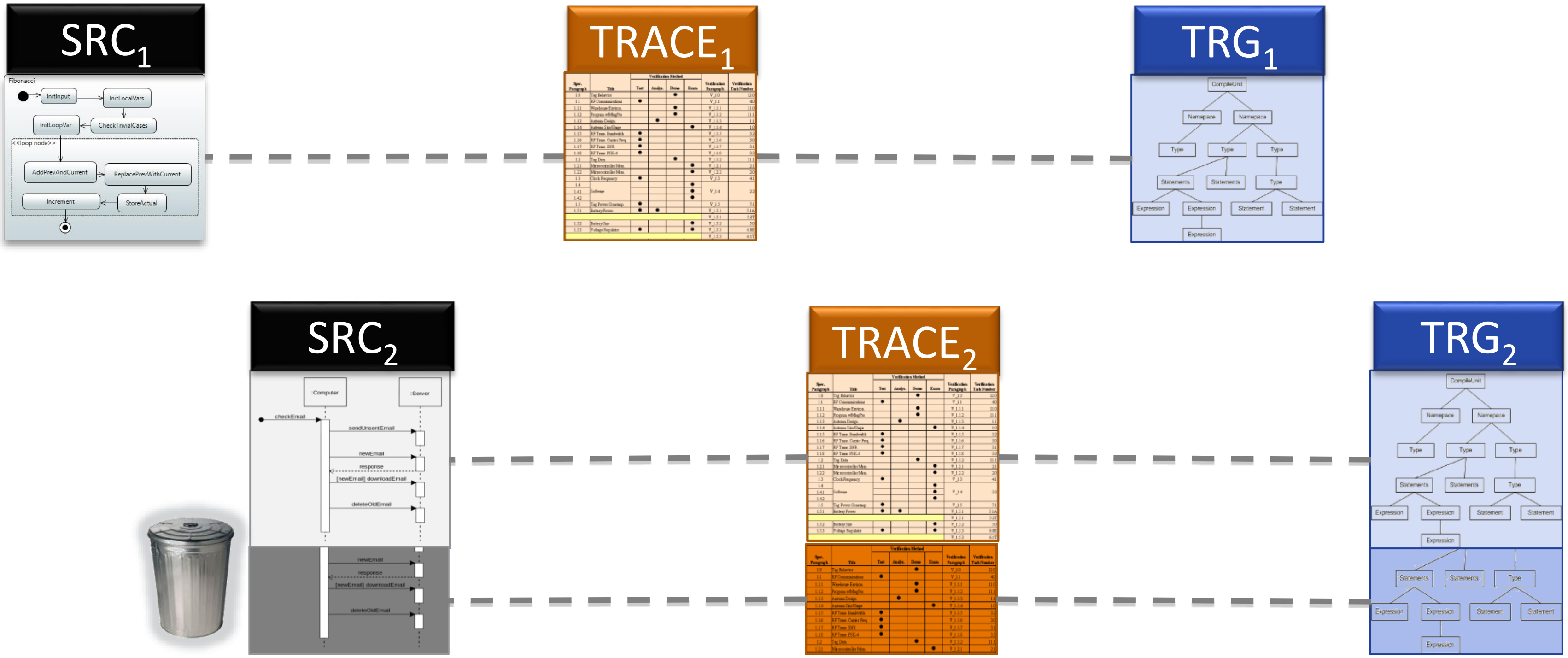


1. First transformation

2. Source model changes

3. Re-execute from scratch for all source models

Dirty Incrementality



Pros:

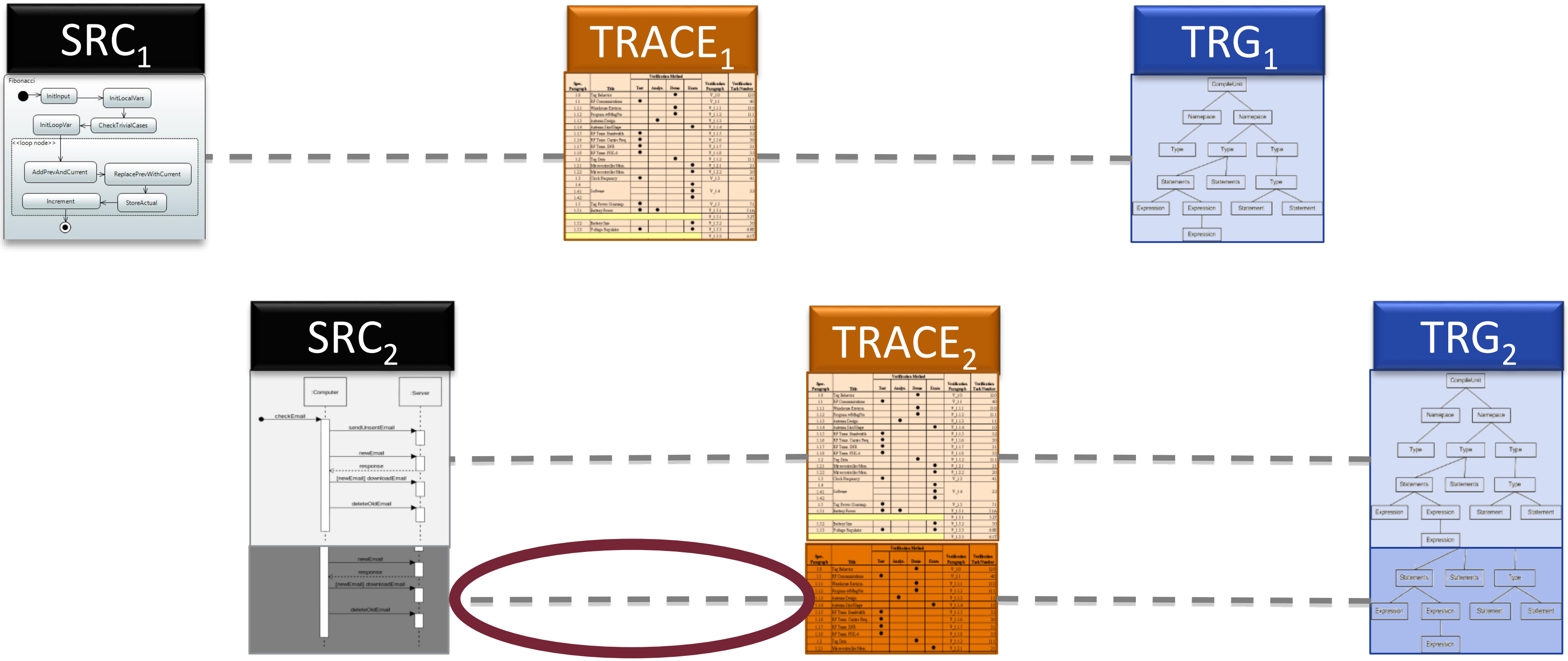
- Large-step incrementality
- Avoids continuous exec.

Cons:

- Complex MT can be slow
- Cleanup (after an error)?
- Chaining?

1. First transformation
2. Source model changes
3. Re-execute from scratch only for changed models

Incrementality by Traceability



Pros:

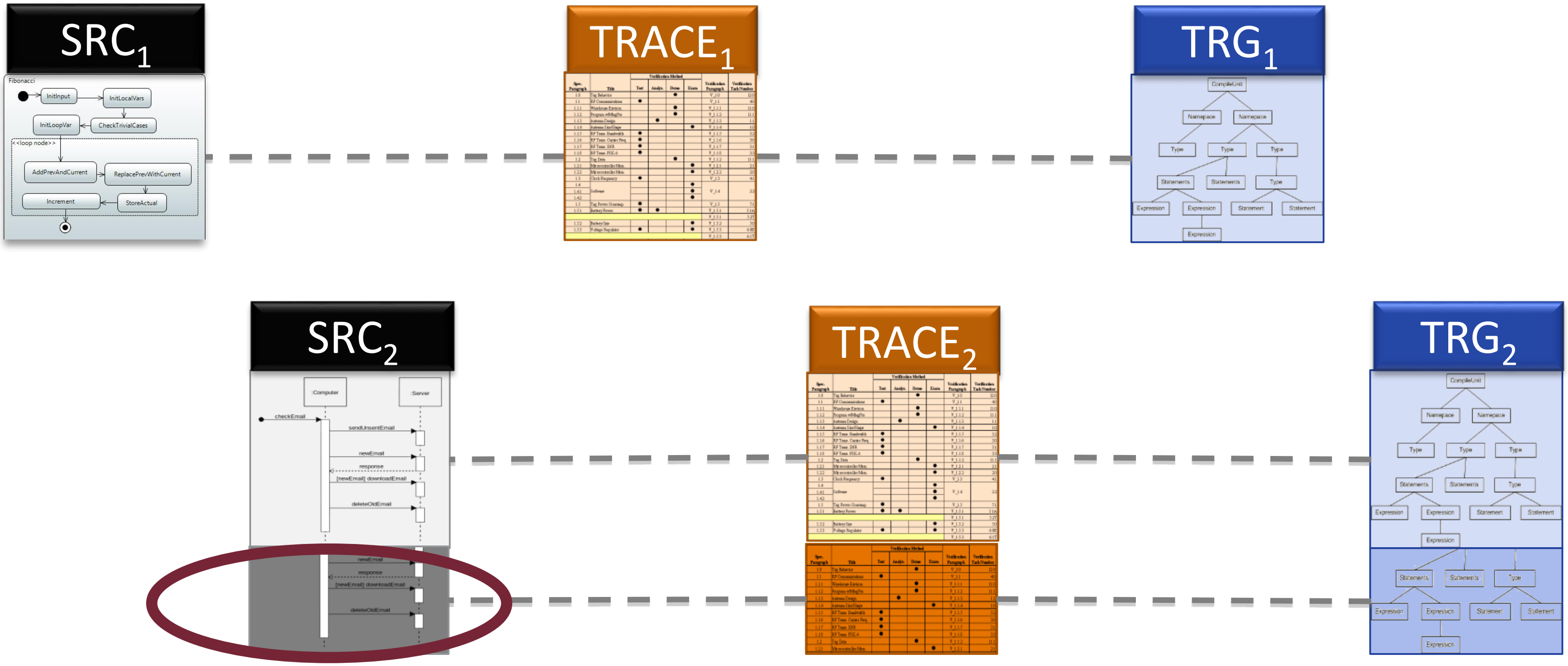
- Small-step incrementality
- Better performance

Cons:

- Highly depends on traceability links
- Smart matcher needed

1. First transformation
2. Source model changes
3. Detect missing trace links
4. Re-execute MT only for untraceable elements

Event Driven Transformations



Pros:

- Refined context: driven by changes of query result set
- Chaining
- Avoids continuous comp.

Cons:

- Language-level restrictions

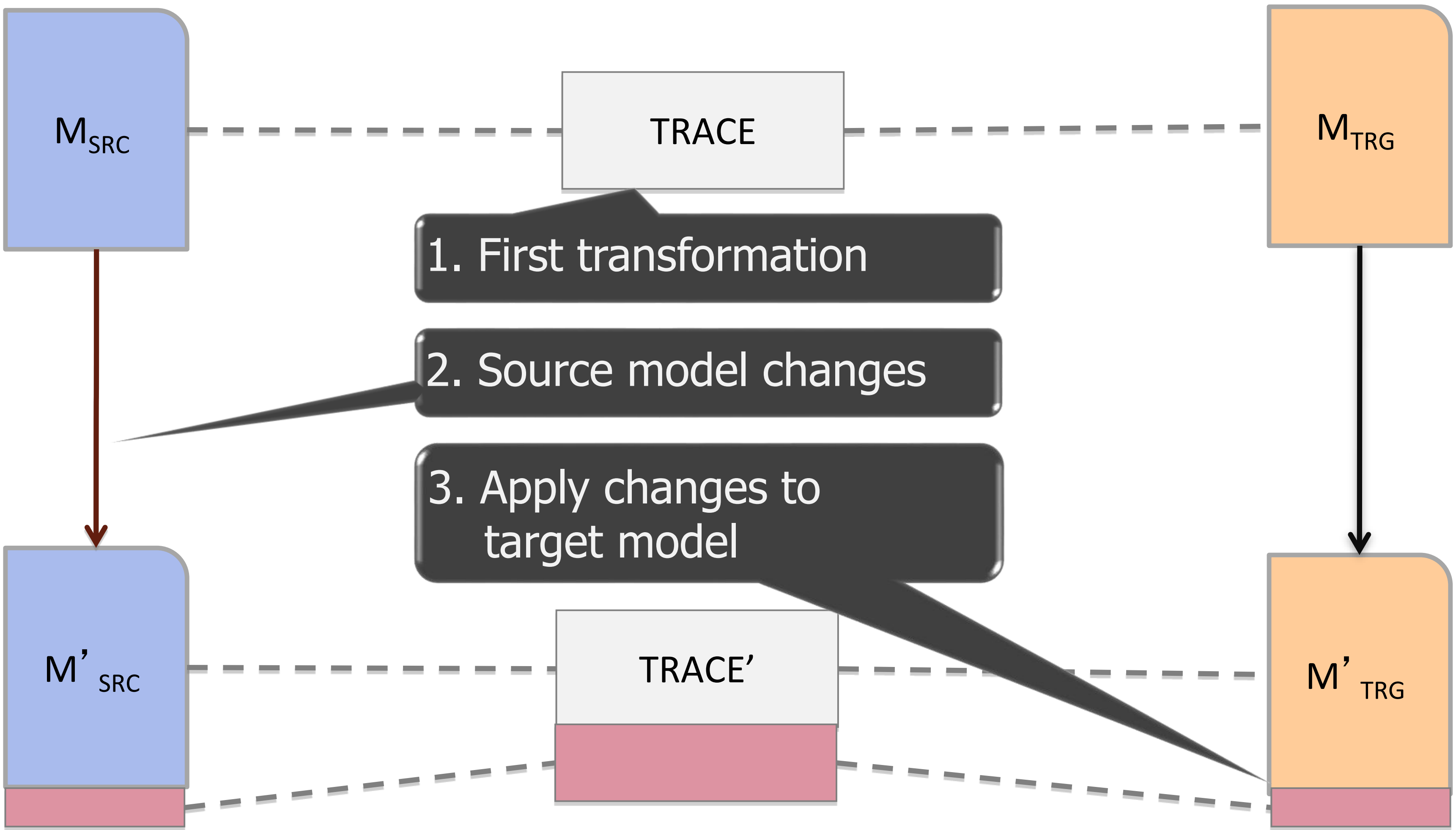
1. First transformation
2. Source model changes
3. Process change notification
4. Propagate change

VIATRA

Aspects of Incrementality

- Goals: to save work by…
  - **Target Incrementality**
    - …reusing unchanged parts of the target
    - Further benefits
      - Existing links to unchanged parts preserved
      - Existing analysis on unchanged parts preserved
      - Does not propagate along transformation chains
  - **Source Incrementality**
    - …ignoring unchanged parts of the source
    - Use incremental model query!

Incremental Forward Transformation

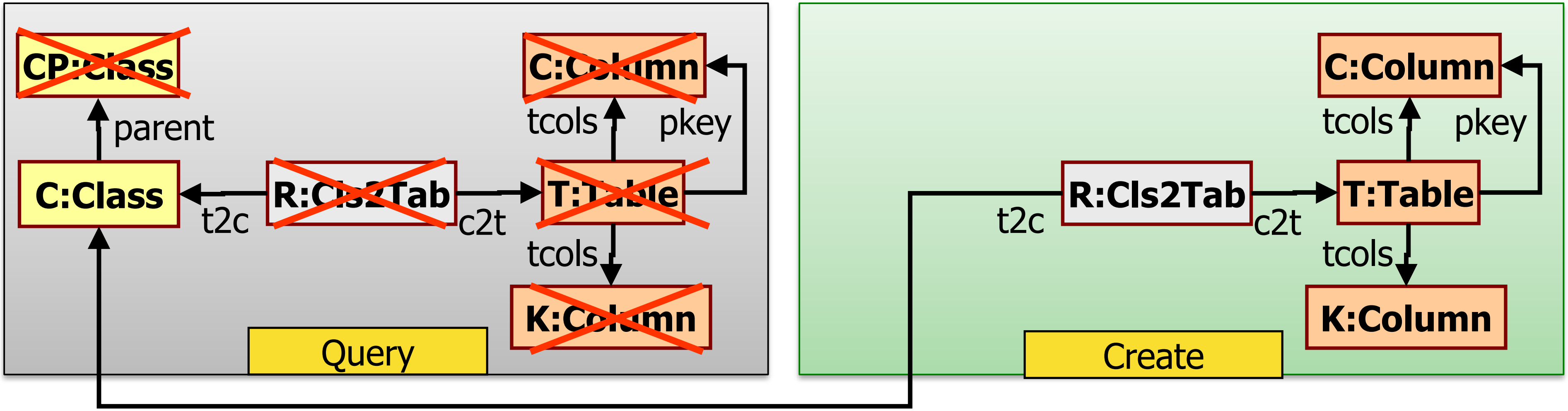


Practical application scenarios:

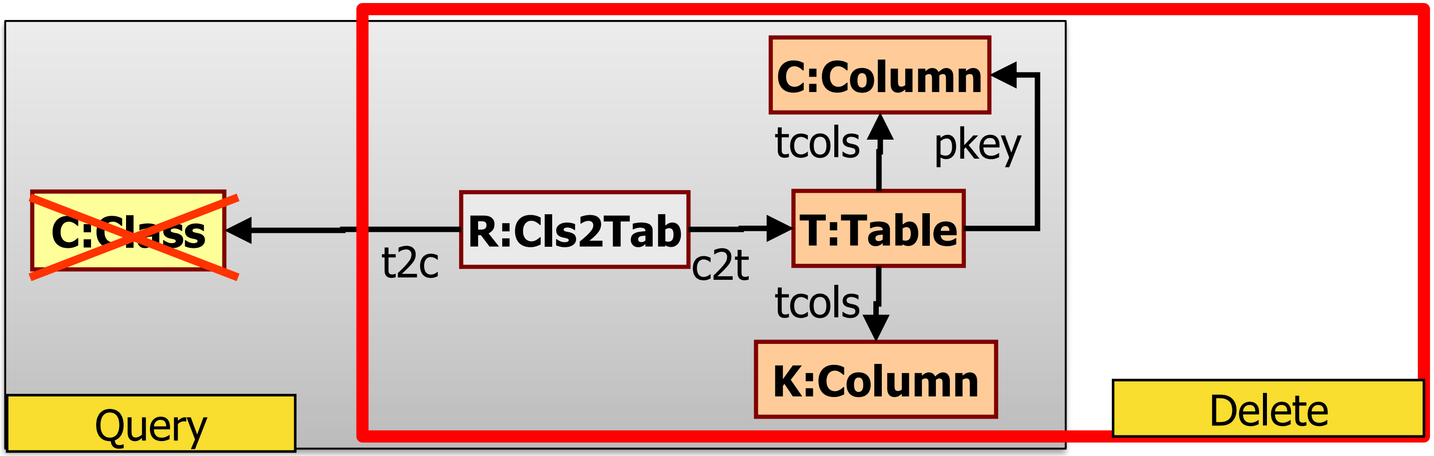
- Incremental model synchronization
- Tool integration

Revisit Motivating Example

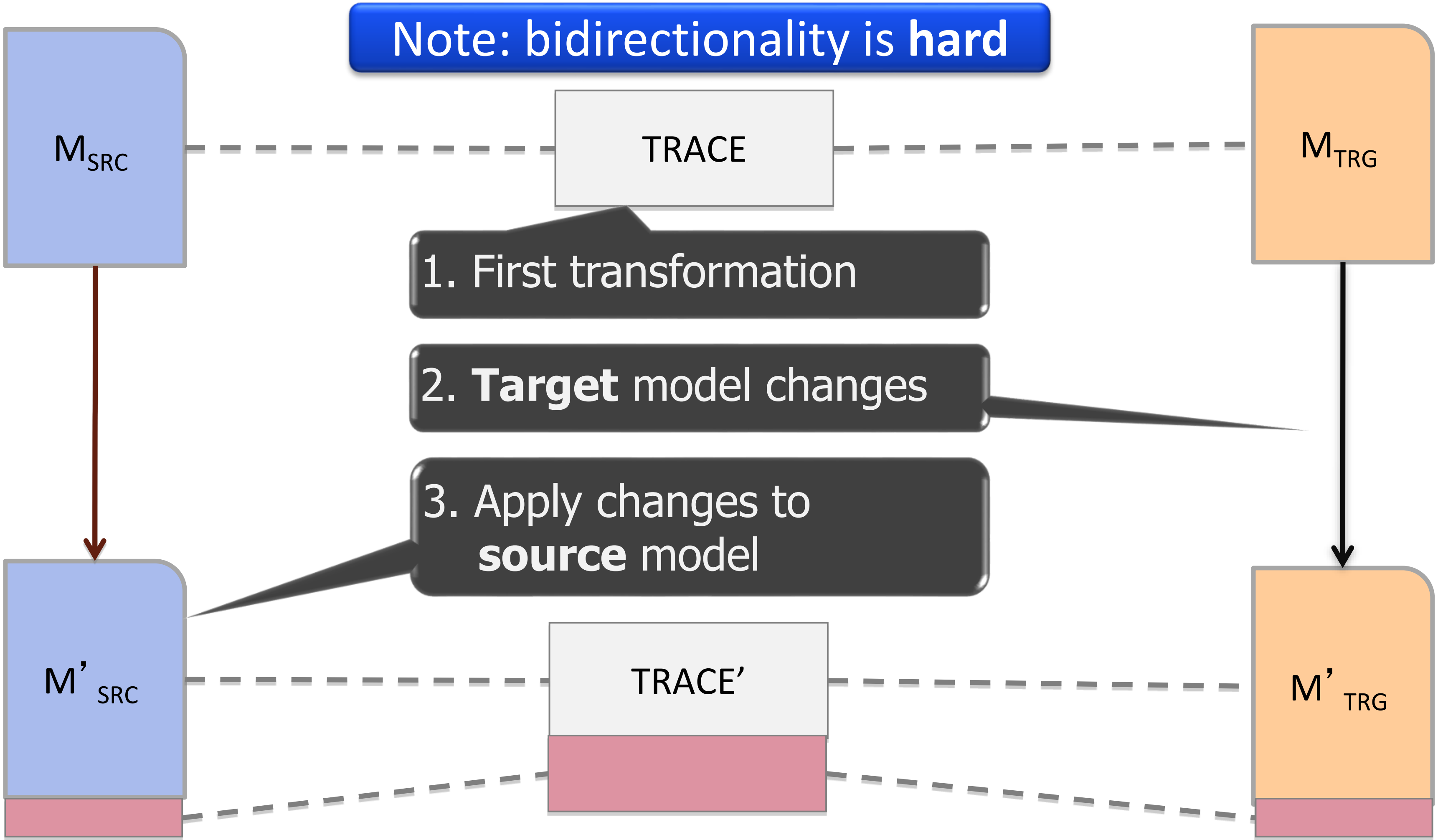
- Map new, unmapped root classes to tables



- Remove old tables no longer having a source class



Incremental Backward Transformation



Extra challenge if not hard enough:

SRC→TRG specified

TRG→SRC inferred

Recent Approaches:

A. Schürr, P. Stevens, N. Foster, T. Hettel, Cicchetti&Pierantonio, Czarnecki&Diskin

VIATRA: A Reactive Incremental Transformation Platform

Reactive Event Driven Transformations

What has changed?

Observed events

Controlled events

VIATRA: Reactive Transformation Engine

Actions

When to react?

Perform in consistent state

Reactive Event Driven Transformations

- Model modified
- Match appeared
- Event sequence identified

Observed events

Controlled events

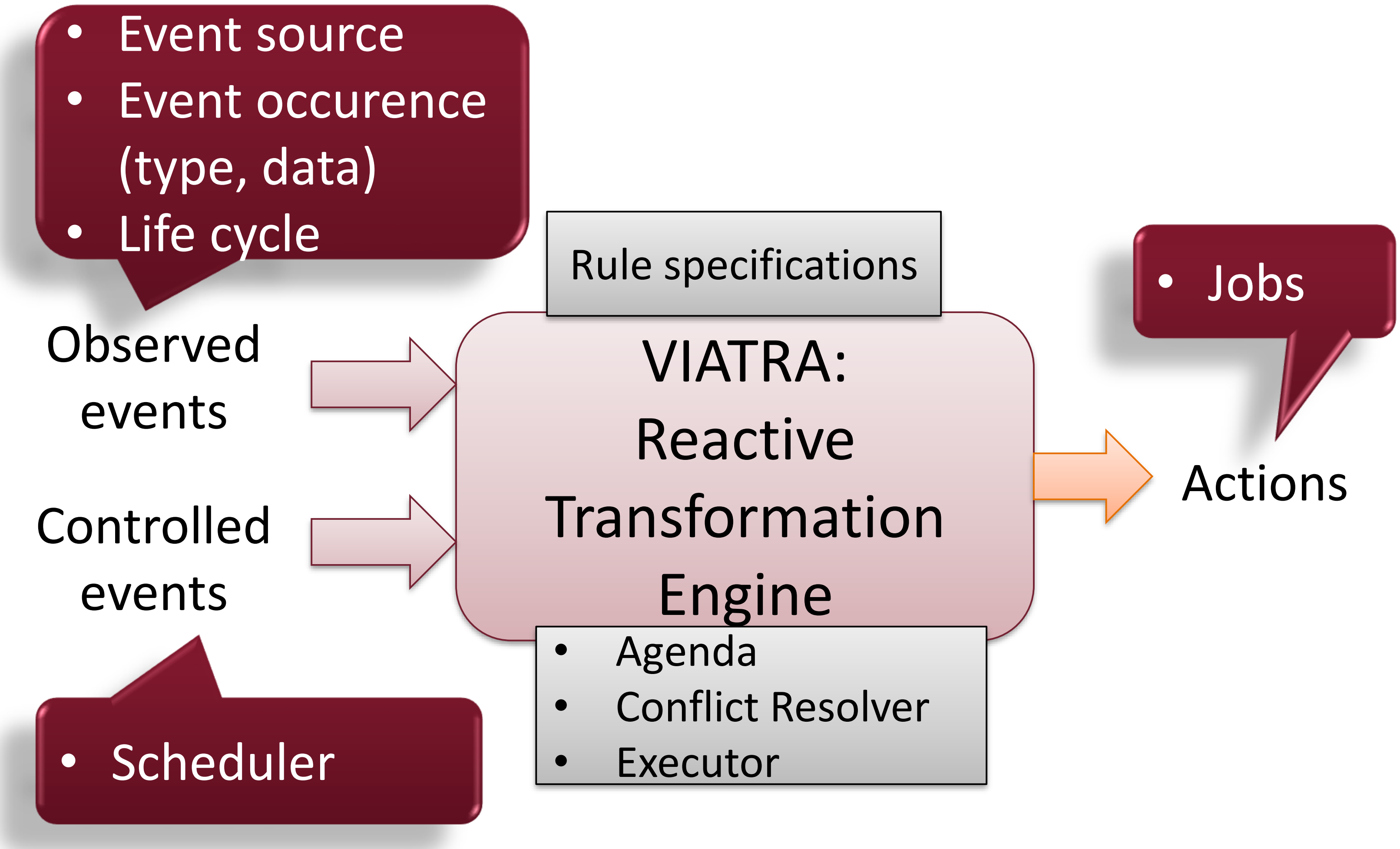
- „Run” button pushed
- Consistent state reached after editing
- Transaction committed

VIATRA: Reactive Transformation Engine

Actions

- Modify model
- Add error marker
- Update view
- Send e-mail

Reactive Event Driven Transformations



Language Example

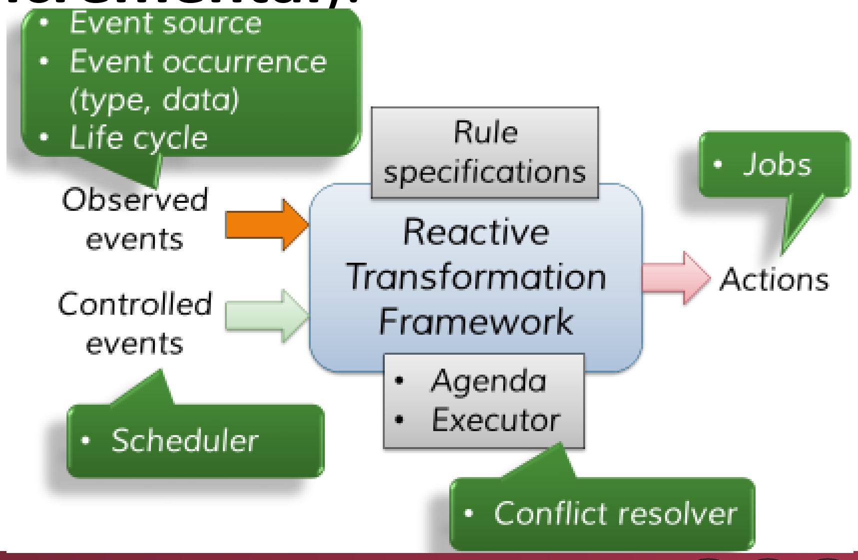
pattern someCondition(param1, param2) {...}

Query language

Xtend (Java)

Event data

```
val rule = createRule().precondit… on).
  action[ match |  // perform acti… ].build
val incrRule = createRule().precondition(someCondition).
  lifecycle(ActivationLifecycles.incremental).
  action(::Appeared)[
   match | // perform action].
  action(::Disappeared)[
   match | // perform action].
  build
```

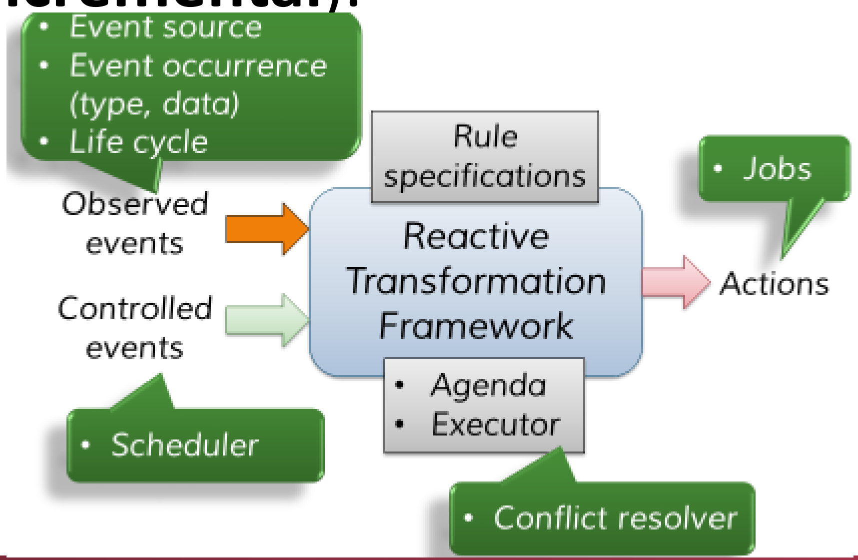


Language Example

```
pattern someCondition( param1, param
- - - - - - - - - - - - - - - - - - - - - -
val rule = createRule().precondition(someCondition).
  action[ match |  // perform action ].build
val incrRule = createRule().precondition(someCondition).
  lifecycle(ActivationLifecycles.incremental).
  action(::Appeared)[
   match | // perform action].
  action(::Disappeared)[
   match | // perform action].
  build
```

Rule specification

Xtend (Java)



Language Example

```
pattern someCondition( param1, param2 ) {...}
```

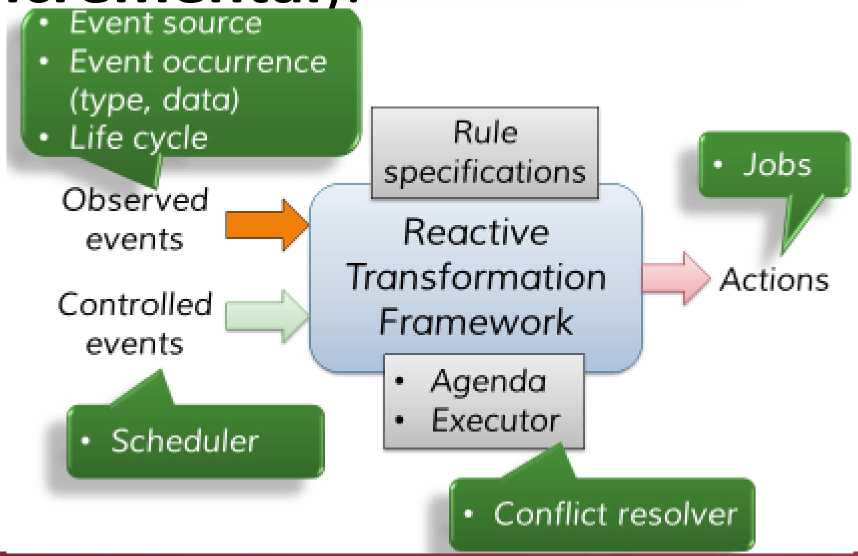
Query language

- -

Xtend (Java)

```
val rule = createRule().precondition(someCondition).
  action[ match |  // perform action ].build
val incrRule = createRule().preconditi
  lifecycle(ActivationLifecycles.incremental).
  action(::Appeared)[
   match | // perform action].
  action(::Disappeared)[
   match | // perform action].
  build
```

Observed events



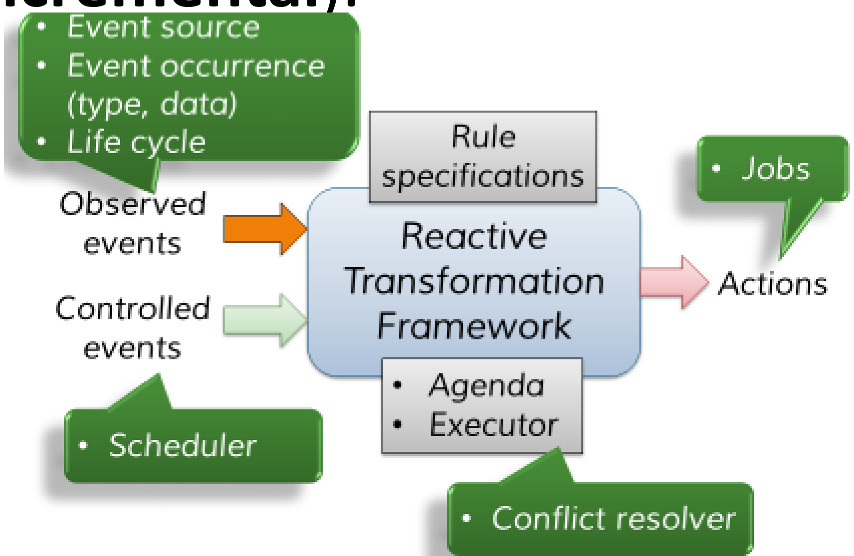
Language Example

pattern someCondition(param1, param2) {...} Query language

Xtend (Java)

val rule = createRule().precondition(someCondition).
action[match | // perform action].build
val incrRule = createRule().precondition(someCondition).
lifecycle(**ationLifecycles.incremental**).
actio
match | // perform action].
action(**::Disappeared**)[
match | // perform action].
build

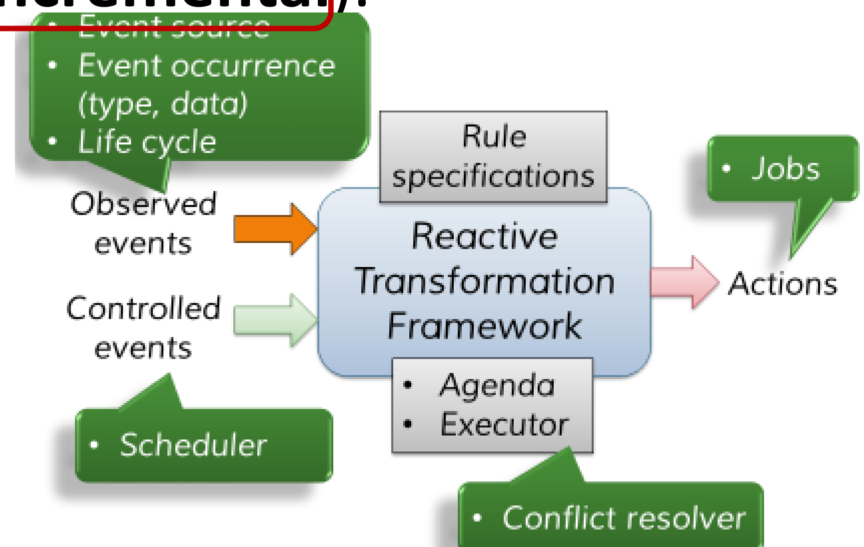
Job specification



Language Example

```
pattern someCondition( param1, param2 ) {…}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
val rule = createRule().precondition(some
  action[ match |  // perform action ].build
val incrRule = createRule().precondition(someCondition).
  lifecycle(ActivationLifecycles.incremental).
  action(::Appeared)[
   match | // perform action].
  action(::Disappeared)[
   match | // perform action].
  build
```

Activation state-event transitions



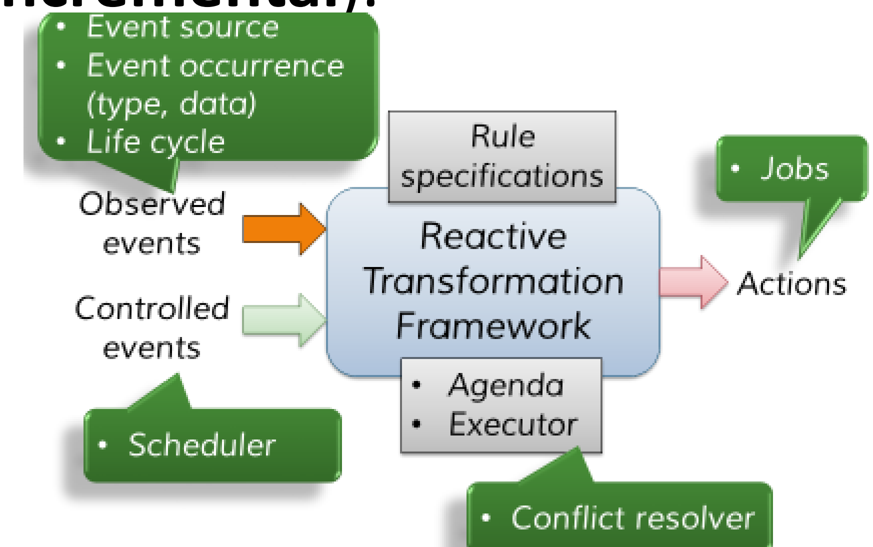
Language Example

pattern someCondition(param1, param2) {...} Query language

Xtend (Java)

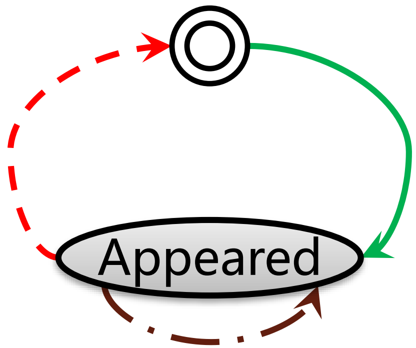
Jobs associated with event types

```
Rule().precondition(someCondition).
 // perform action ].build
val rRule = createRule().precondition(someCondition).
  lifecy le(ActivationLifecycles.incremental).
  action(::Appeared)[
   match | // perform action].
  action(::Disappeared)[
   match | // perform action].
  build
```

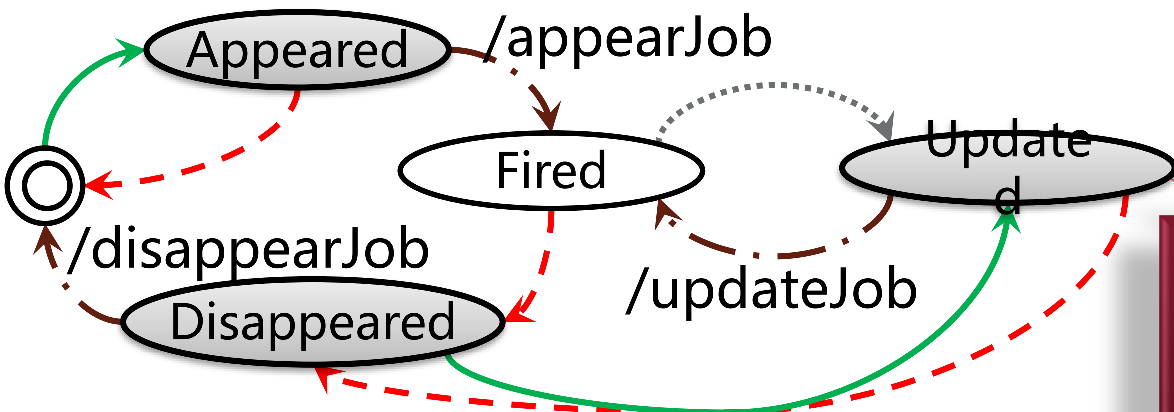


Activation Lifecycles

- Batch transformation



- Event-driven transformation

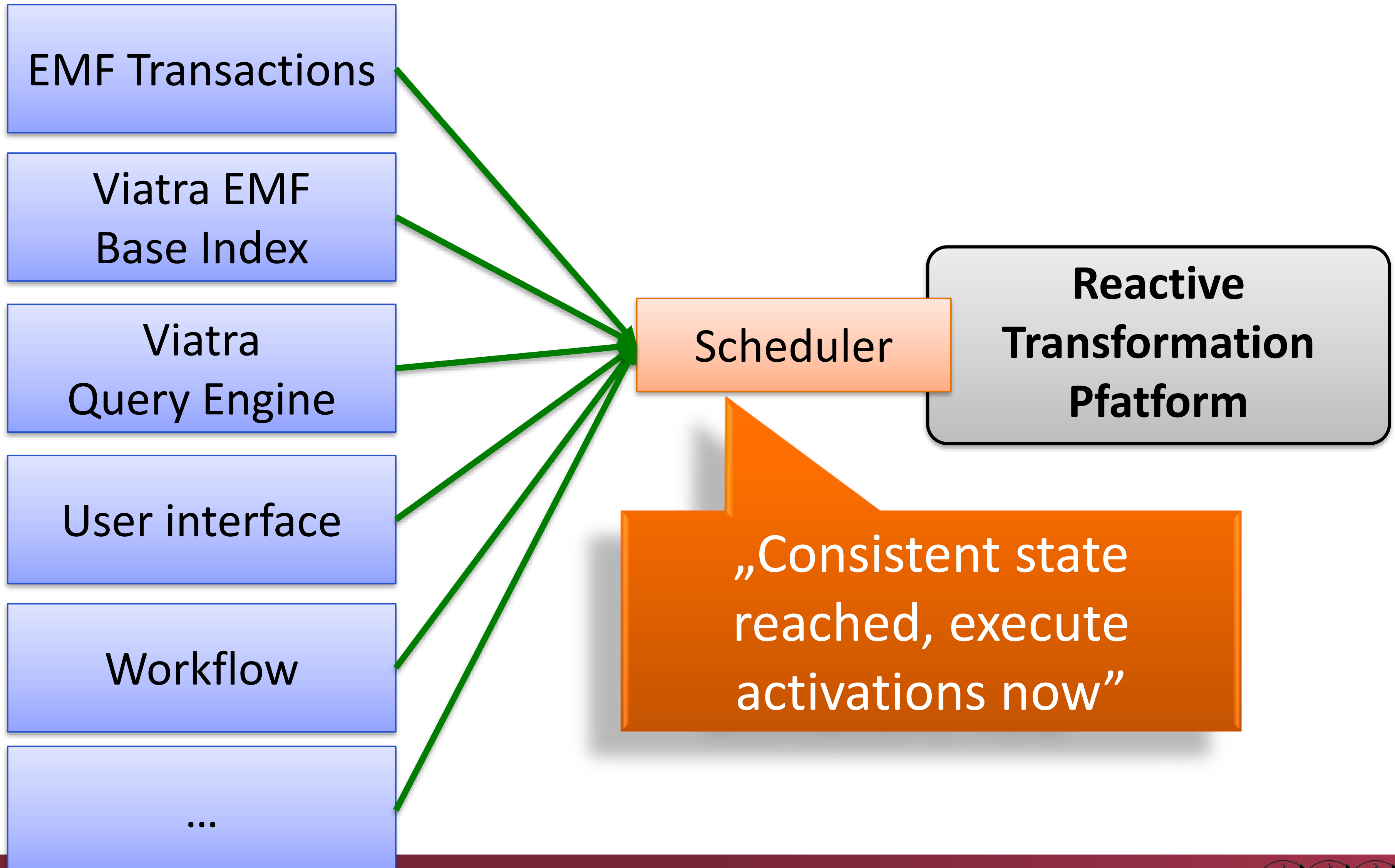


Phases
- Enabled
- Disabled
- Initial

Transitions
- Appear
- Update
- Disappear
- Fire

Only feature of event data object has changed

Scheduling


EMF Transactions
Viatra EMF Base Index
Viatra Query Engine
User interface
Workflow
…
Scheduler
Reactive Transformation Pfatform
„Consistent state reached, execute activations now”

Conflict Resolution

- Multiple actions available
  - Activations of different rules
  - Different activations in the same rule
    - Different matches of the precondition pattern
- Which activation to execute next?
- Conflict resolver can be selected
  - Global conflict set: deals with all rules
  - Scoped conflict set: selected rules
  - Customizable resolution strategy: e.g. priority-based

VIATRA: Overview of Features

- **Reactive MT Platform**
  - **MT Language**:
    - Internal DSL over Xtend
    - Transformation API
  - **MT Engine**:
    - Event-driven virtual machine
    - Batch + Incremental MTs
    - Control flow library
    - Compiles to Java
    - Debugger
    - High performance
  - Integrations:
    - EMF, Viatra Query, Xtend, EMF-UML, …

Design Space Exploration

- Explore design model candidates
- Satisfying multiple criteria
- Rule based exploration
- Optimization

Complex Event Processing

- Detect complex event sequences
- Rule based reaction
- Xtext based language

Model Obfuscator

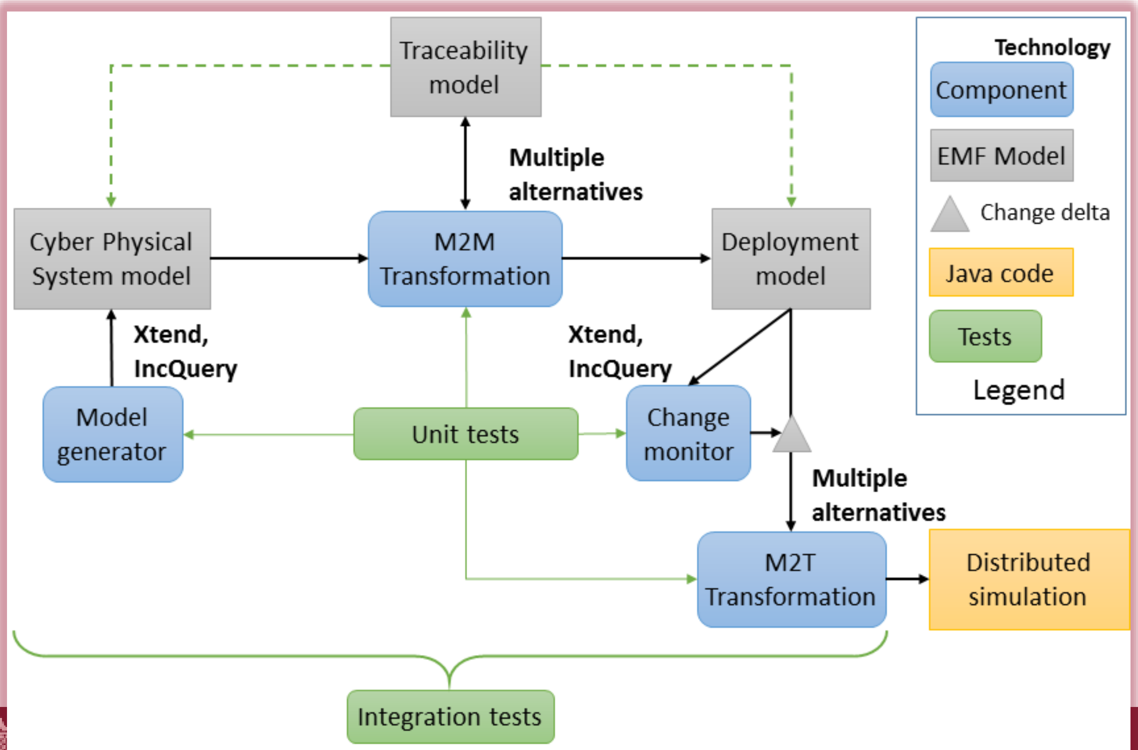
- Remove sensitive information from confidential models
- Original model ➔ Obfuscated model

Performance benchmarks

https://github.com/viatra/viatra-cps-benchmark

CPS Reallocation Benchmark

- Benchmark setup
  - Rule-based redeployment for cloud-based CPS
    - Model generator + Unit tests
    - M2M + M2T transformations
- Different target architecture / platform
  - Industrial (Low-Synch)
  - Client-Server
  - Publish-Subscribe



Test Scenario

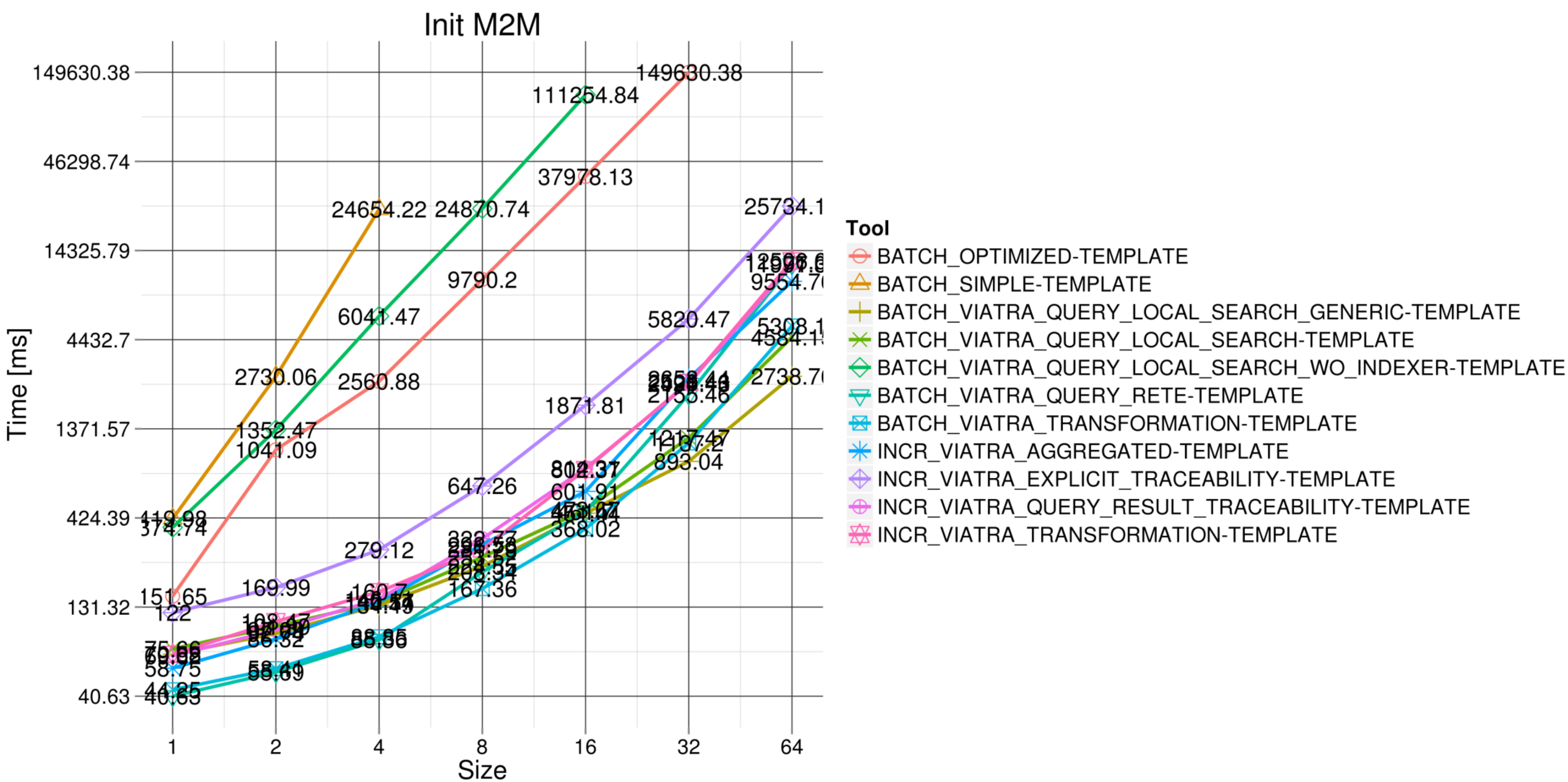
- Different transformation variants
  - Batch
    - Xtend (2 versions)
    - IncQuery+Xtend
  - Incremental
    - Dirty (2 approaches)
    - Explicit traceability
    - Query-driven
    - Change-driven (VIATRA-EVM)
- Executions
  - First transformation execution
  - Small modification + (re)execution
- Environment
  - New machine with 16 GB RAM
- Parameters
  - 10 GB Heap
  - Maximum 10 minutes execution times for complete chain

Scale	SRC Objects	SRC References	TRG Objects	TRG References	Trace Objects	Trace References	SUM Objects	SUM References
1	395	772	366	736	354	720	**1 115**	**2 228**
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]5	762	1 535	**2 384**	**4 891**
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]2	1 522	3 056	**4 750**	**10 725**
8	3 604	17 111	[illegible]	6 108	3 254	6 520	**10 124**	**29 739**
16	7 820	89 193	7 124	12 395	7 112	14 236	**22 056**	**115 824**
32	17 714	594 181	16 308	24 837	16 297	32 605	**50 319**	**651 623**
64	43 795	4 424 529	40 960	50 028	40 948	81 908	**125 703**	**4 556 465**

Trace model's size similar to target model

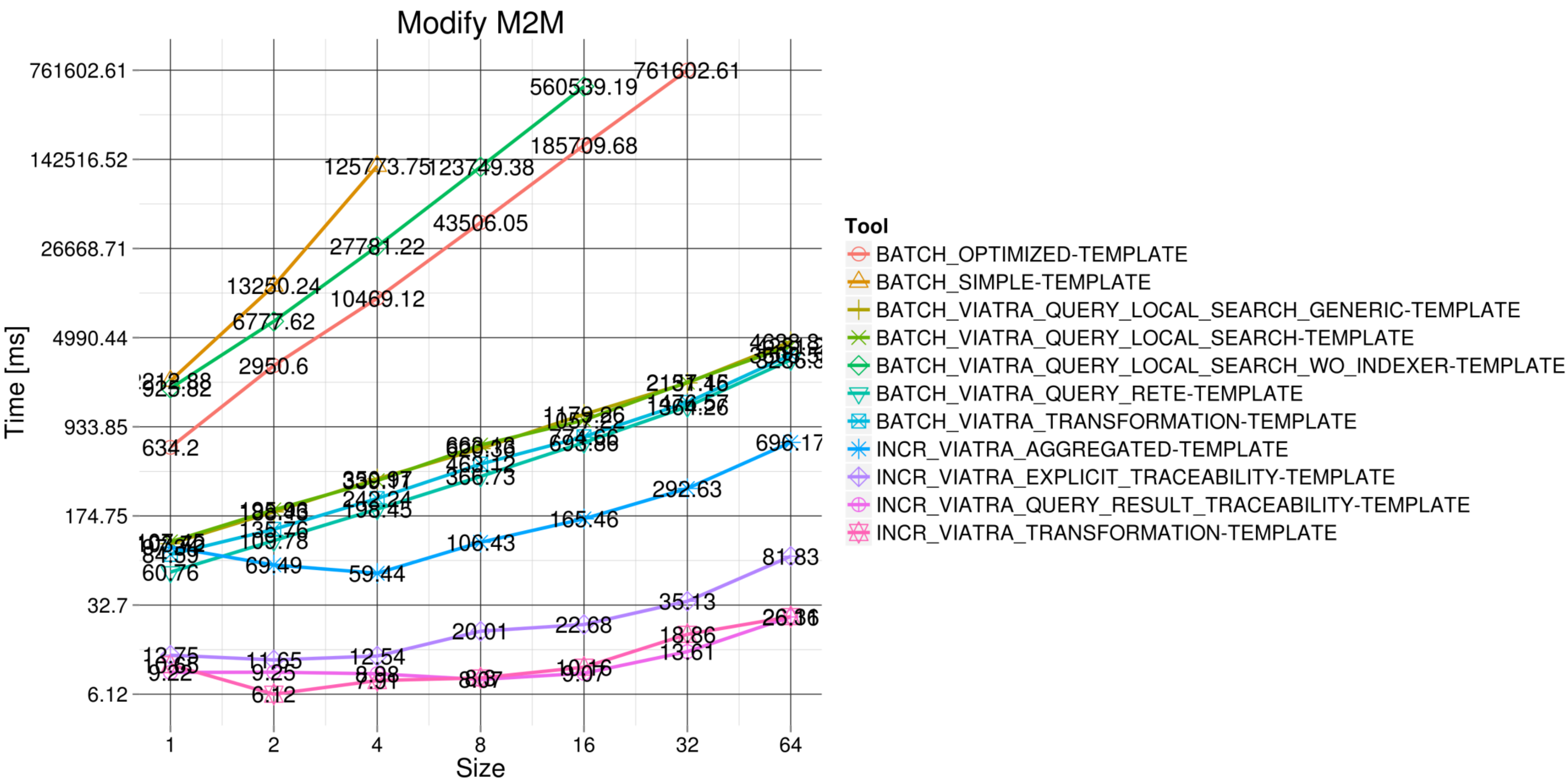
Benchmark results

- **Runtime of initialization and first M2M phase**



Benchmark results

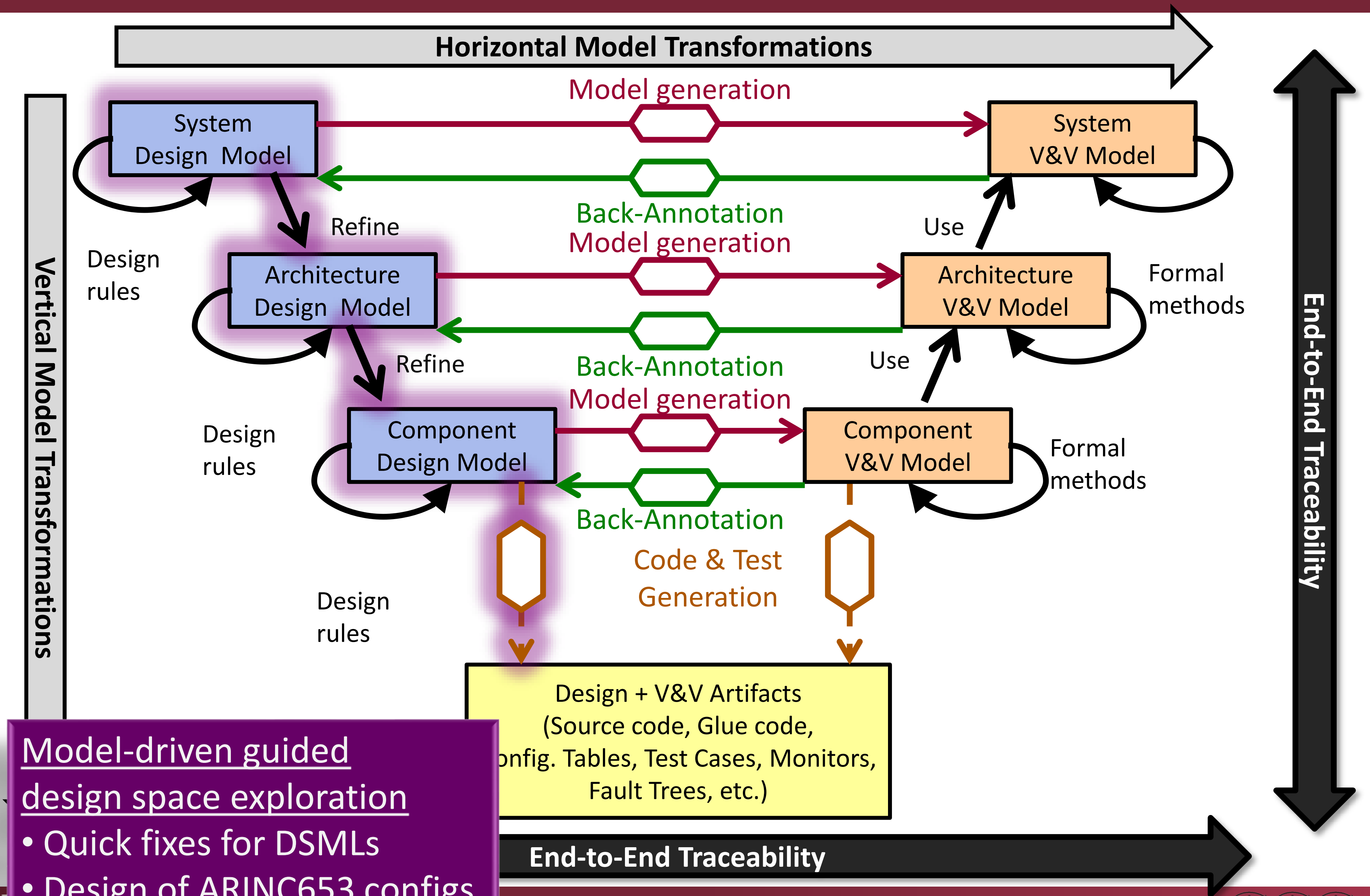
- **Runtime of model modification and M2M phase**



Design Space Exploration

Á. Hegedüs, Á. Horváth, D. Varró:
A model-driven framework for guided design space exploration.
Automated Software Engineering (August 2014)
DOI: 10.1007/s10515-014-0163-1

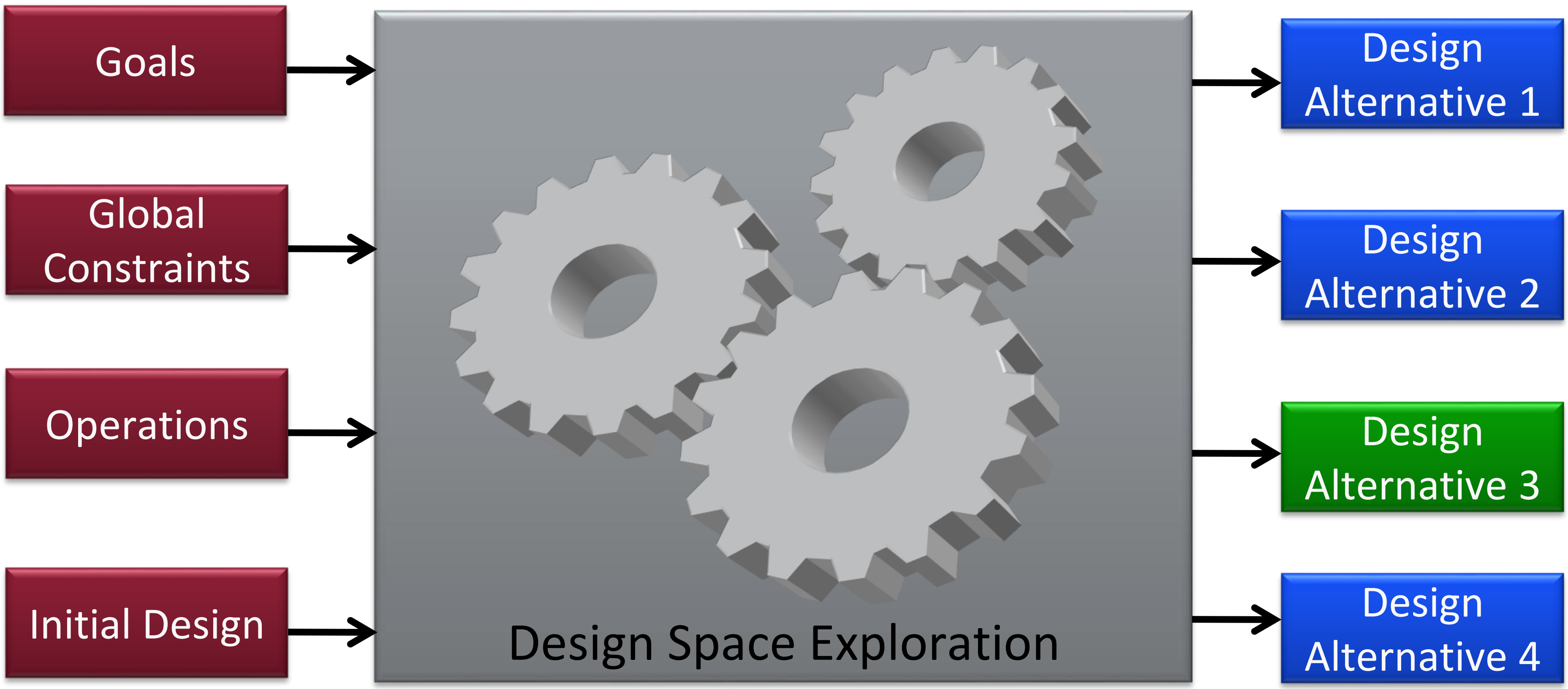
Model-Driven Guided Design Space Exploration



<u>Model-driven guided design space exploration</u>

- Quick fixes for DSMLs
- Design of ARINC653 configs

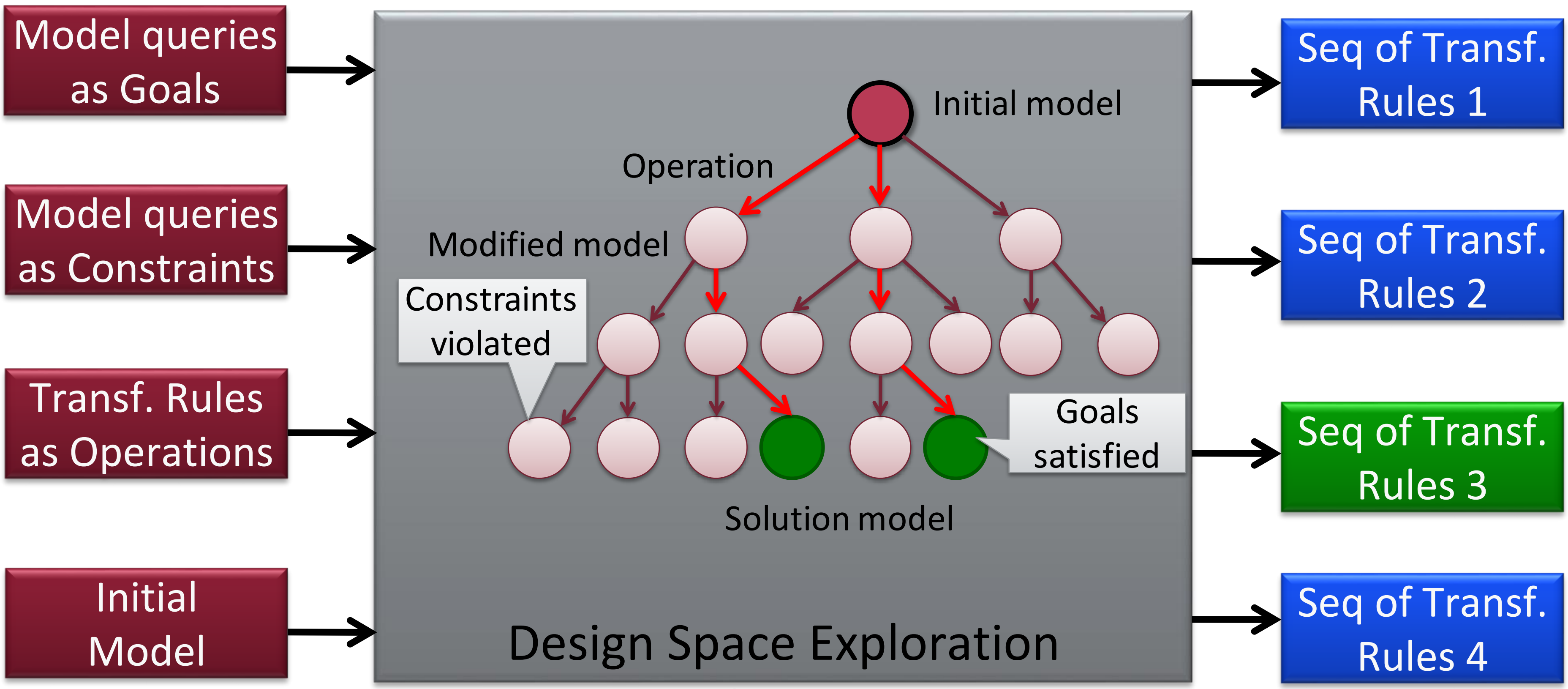
Design Space Exploration



Special state space exploration

- potentially infinite state space
- „dense” solution space

Model Driven Guided Design Space Exploration

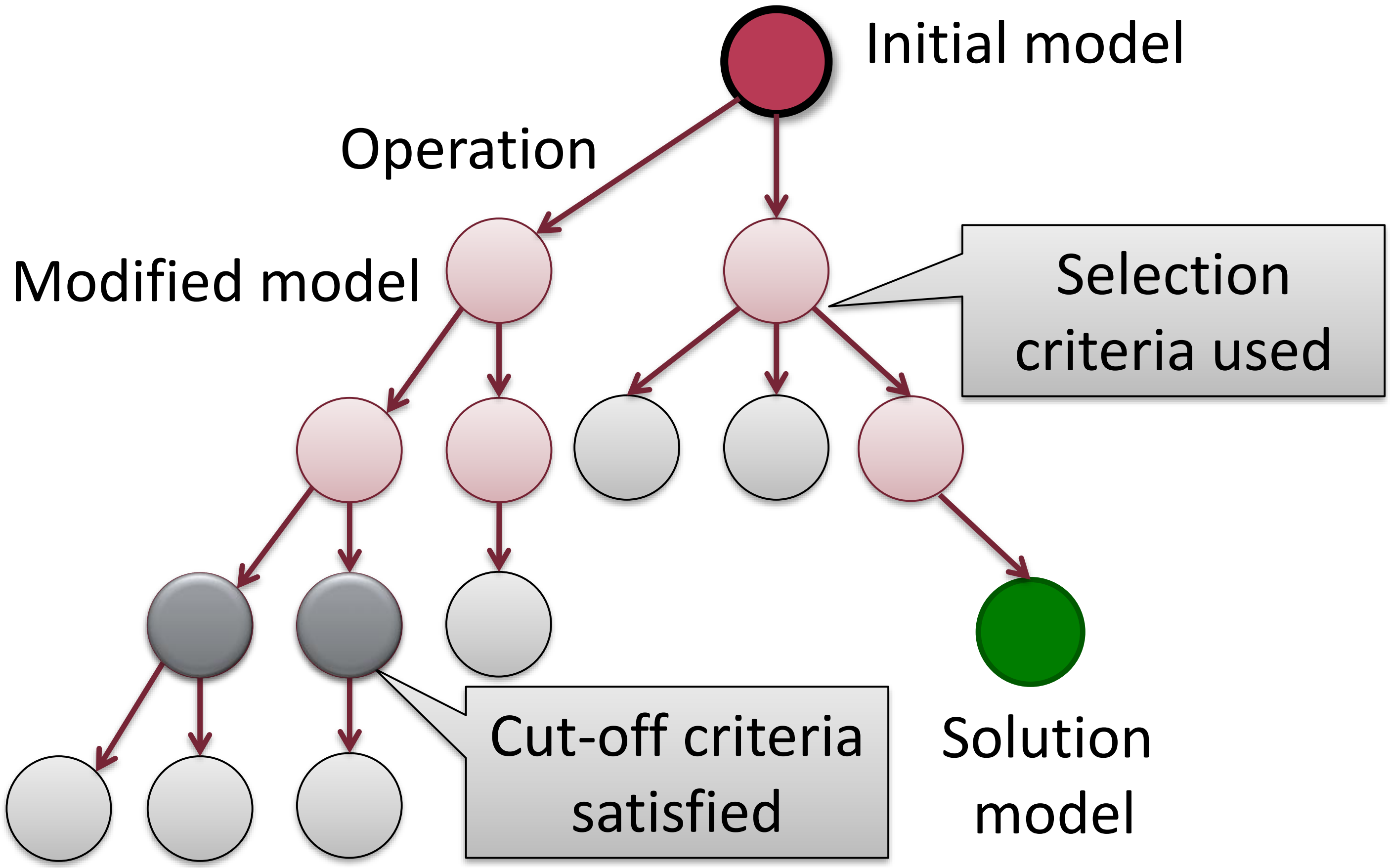


Guidance for exploration: Hints

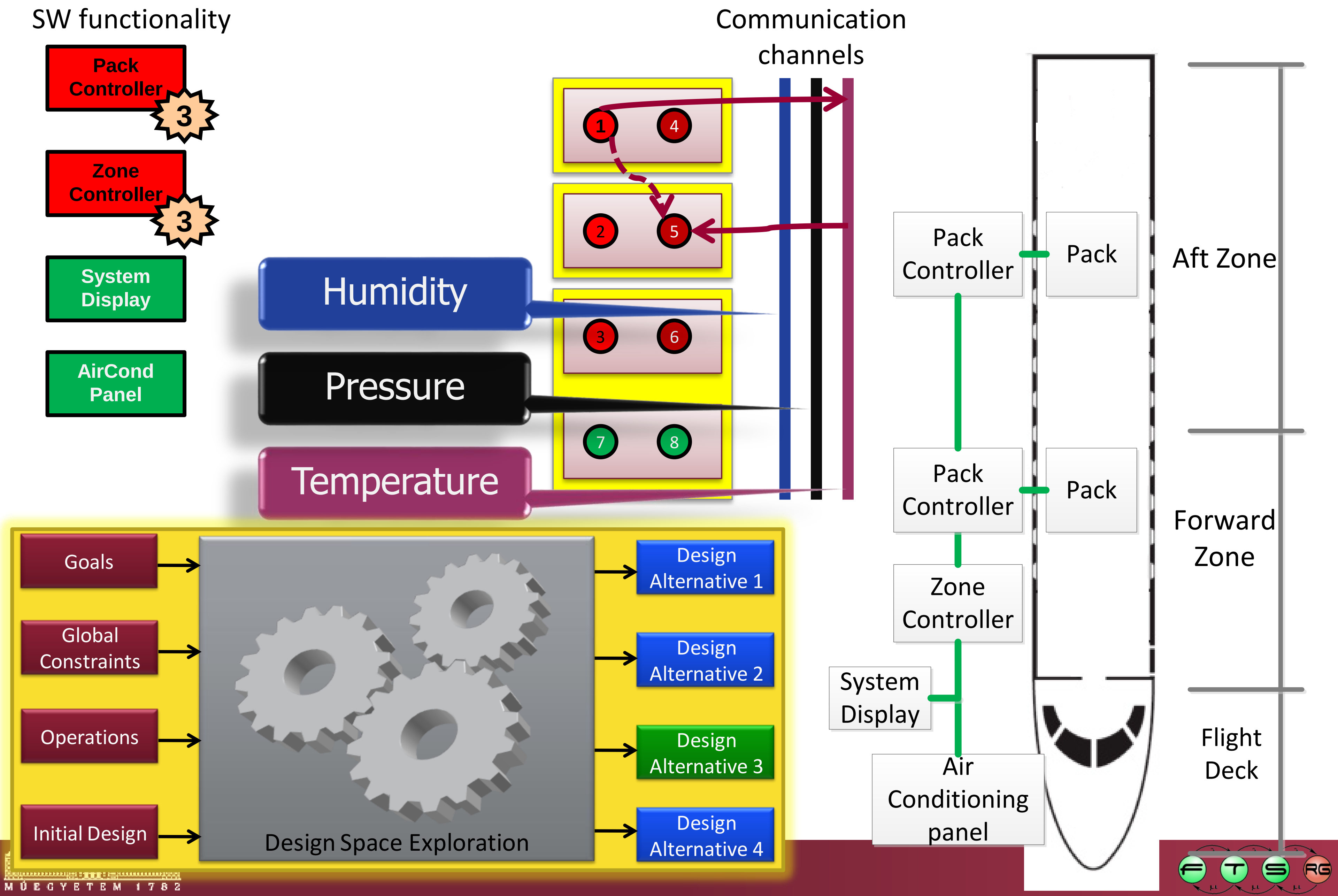
- designer / end user
- formal analysis

Guided Design Space Exploration

- High-level overview

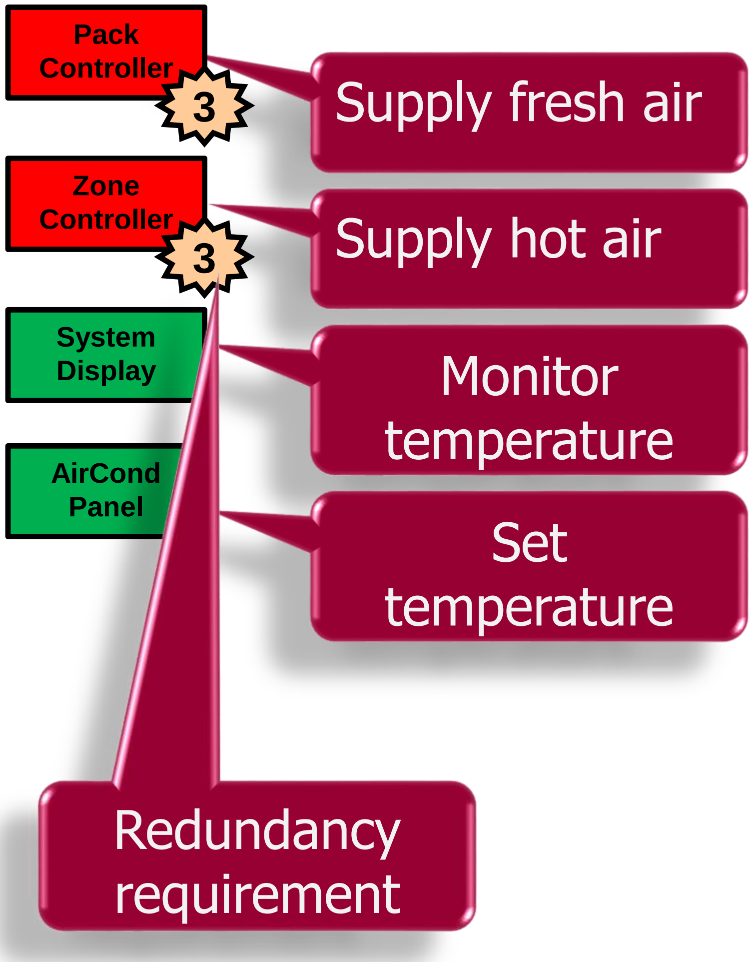


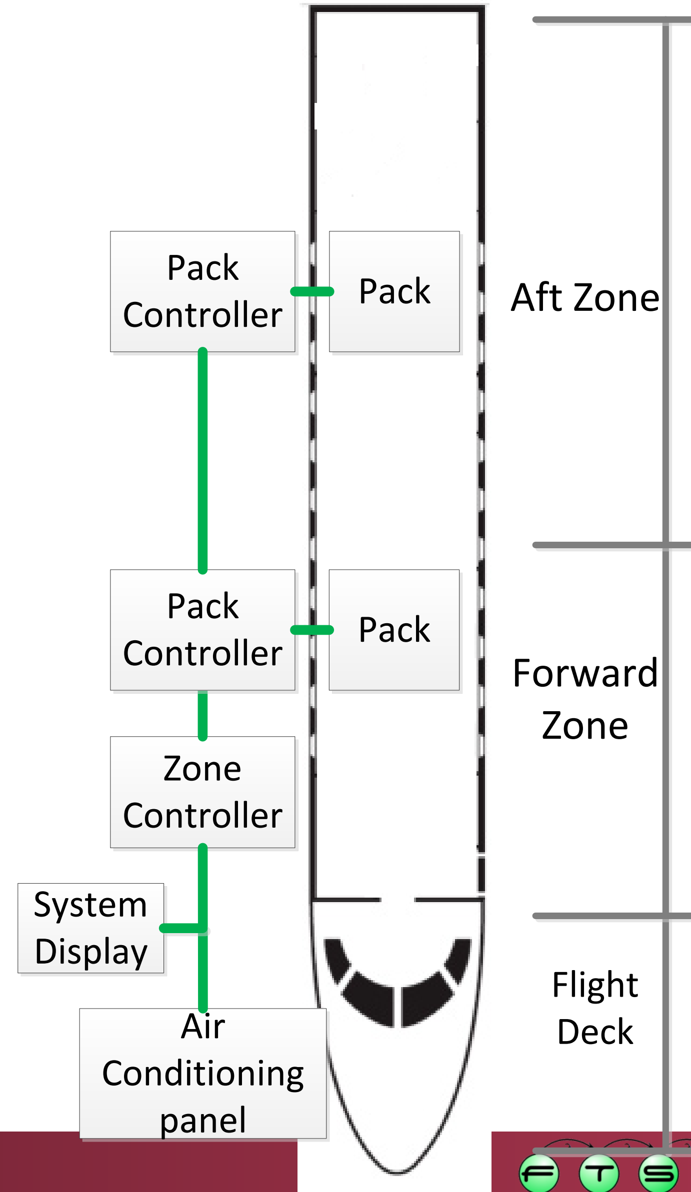
Design Space Exploration for IMA Config. Design



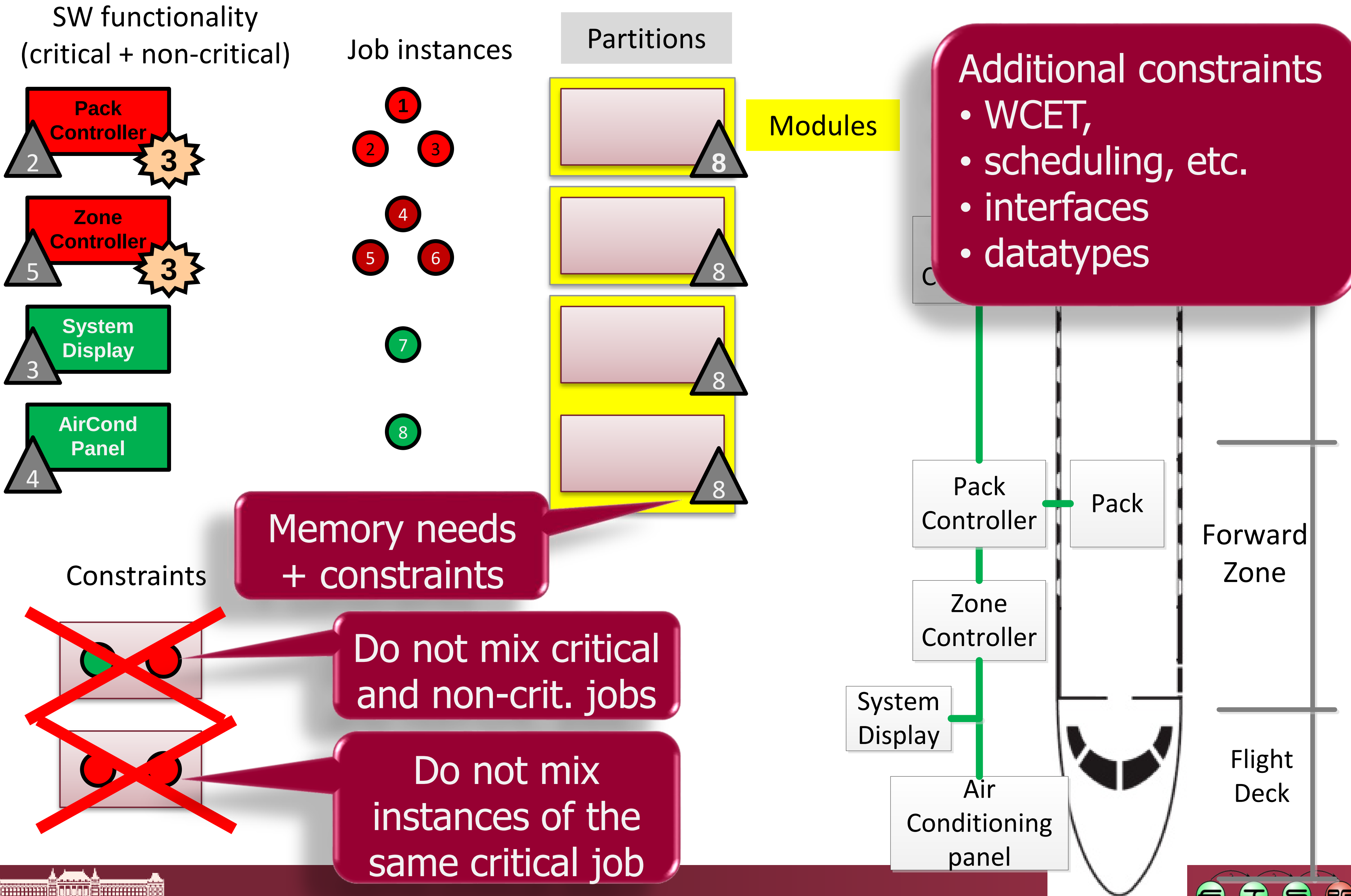
Designing ARINC653 configurations

SW functionality
(critical + non-critical)

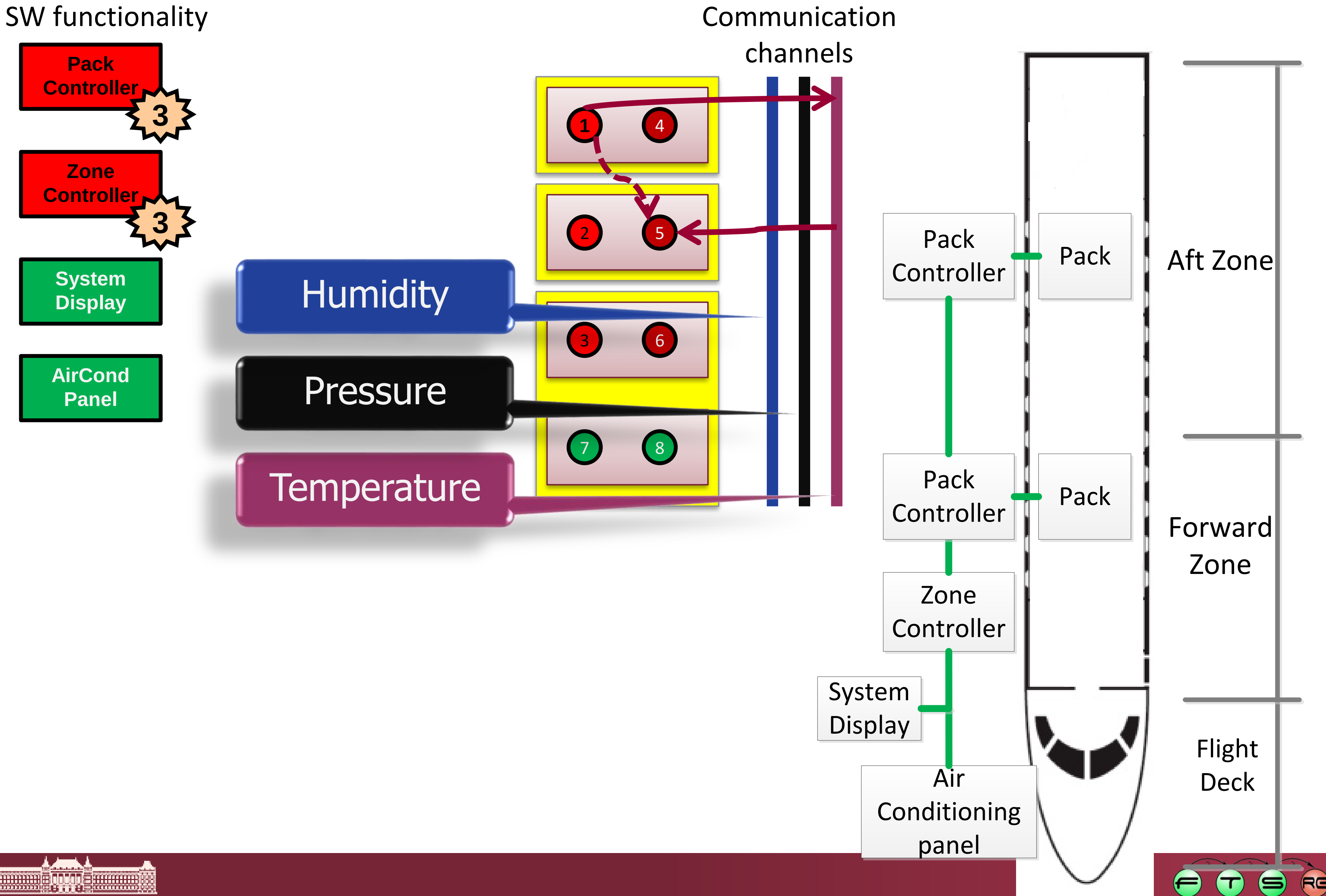




Job instances, Partitions, Modules



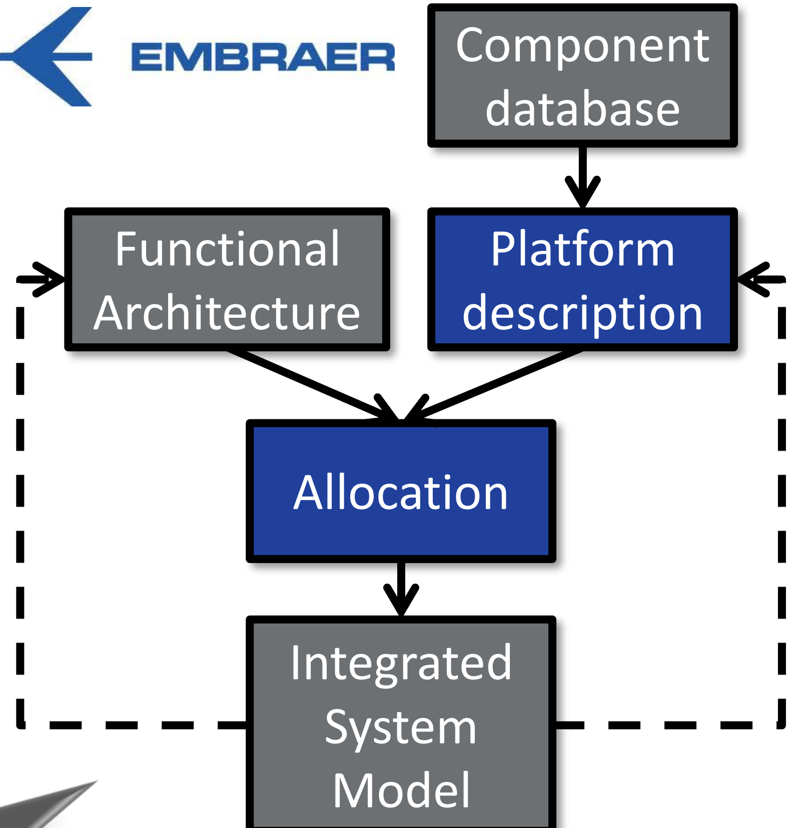
Allocating communication channels



Model Driven Development of IMA Configs

Inputs:
- Platform Independent Model (PIM) (functional + nonfunc. reqs; Simulink)
- Platform Description Model (PDM) for ARINC 653 (DSML)

EMBRAER



Output:
- Integrated system model
- Ready for simulation
- End-to-end traceability

Model Driven Development of IMA Configs

Model transformation chains:
- Designer-guided manual steps
- Automated steps
  - design space exploration
  - optimization
  - code generators
- Continuous validation of design rules