Model-based software development

Lecture X.

Graph pattern matching,
Graph transformation

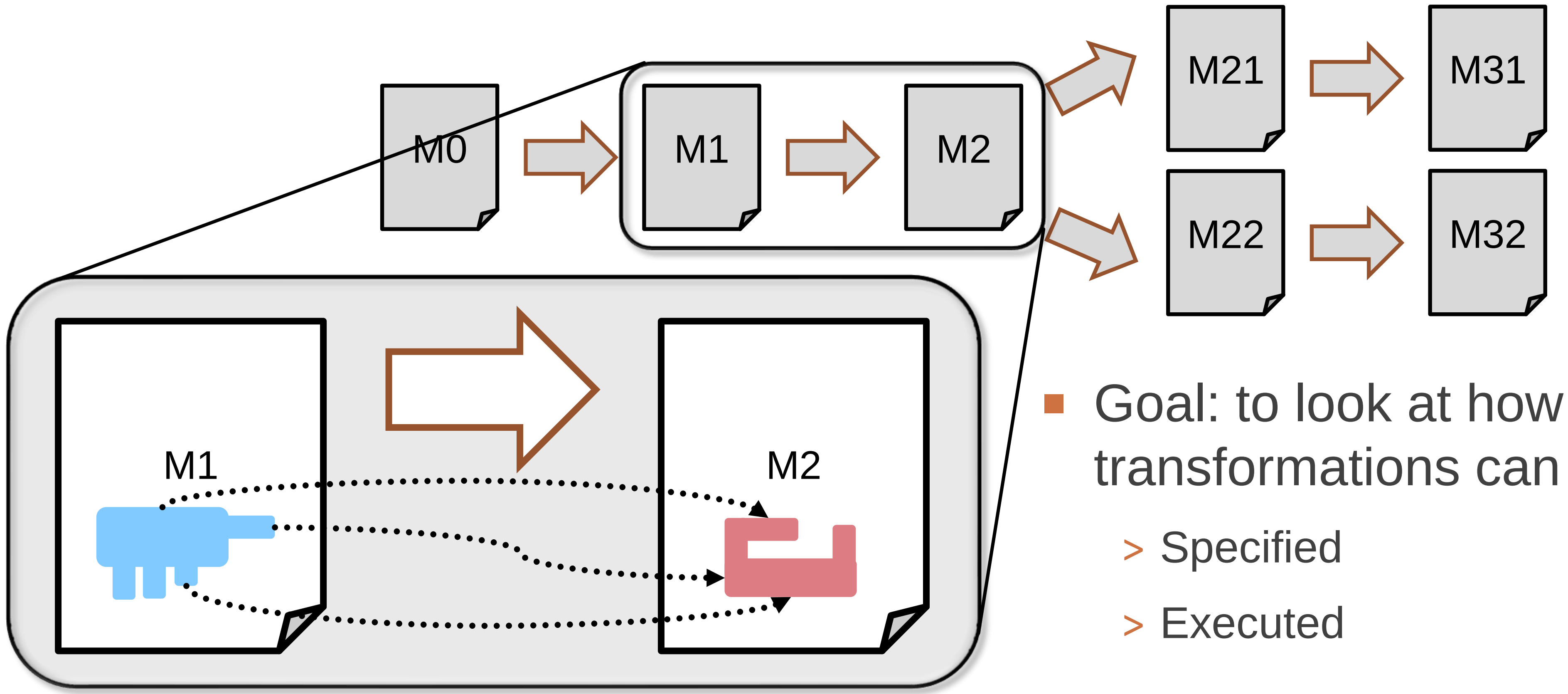
Dr. Semeráth Oszkár

Graph pattern matching, Graph transformation

- **Definitions**
- **Graph pattern matching**
- **Model transformations**
- **Incremental transformations**
- **Design space exploration**

Motivation: Transformation of models

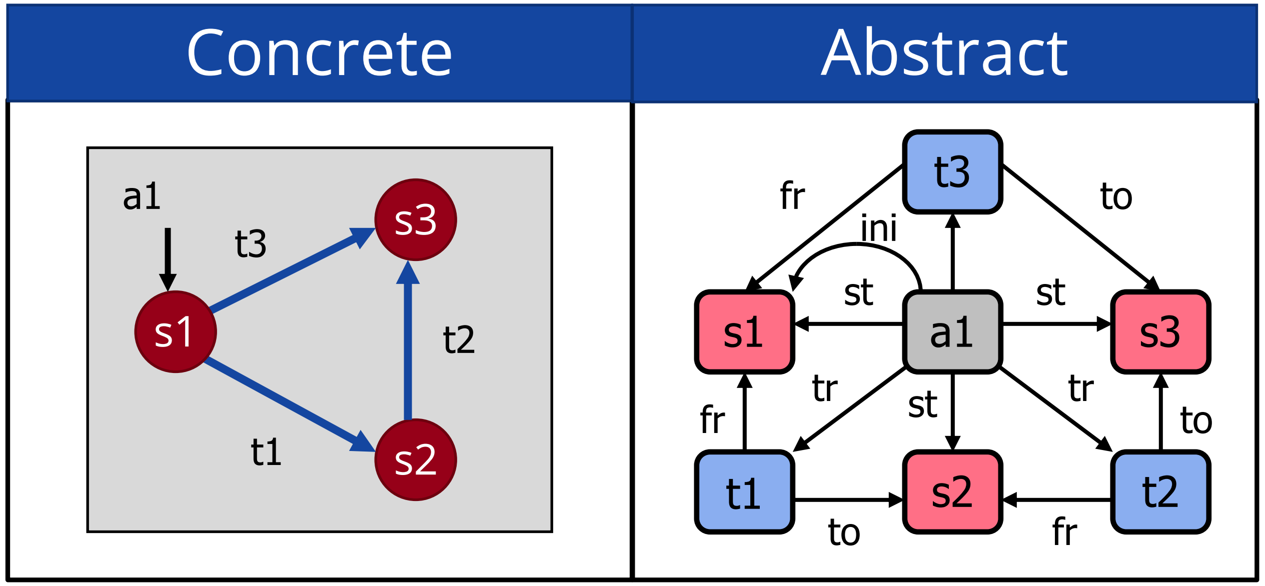
- **Model-based development:** Models as primary documents
- Developing models, automating model processing



- Goal: to look at how model transformations can be
  - Specified
  - Executed

Abstract syntax

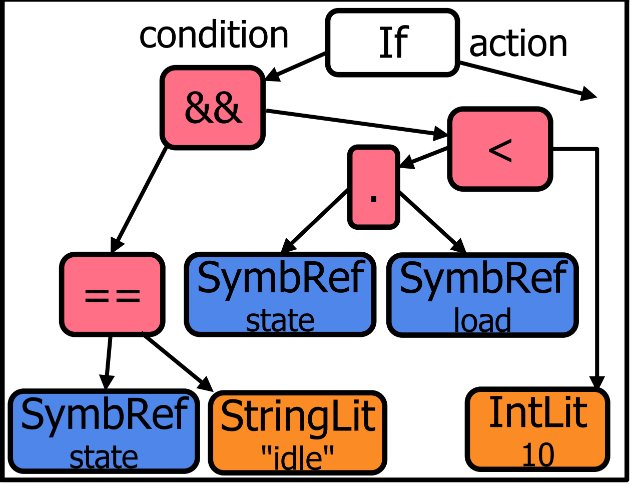
- **How to modify the models?**
- **Idea:** modify the representation of models directly → **Abstract syntax**



Concrete

```
if (
   state ==
   "idle" &&
   this.load<10)
   …
```

Abstract

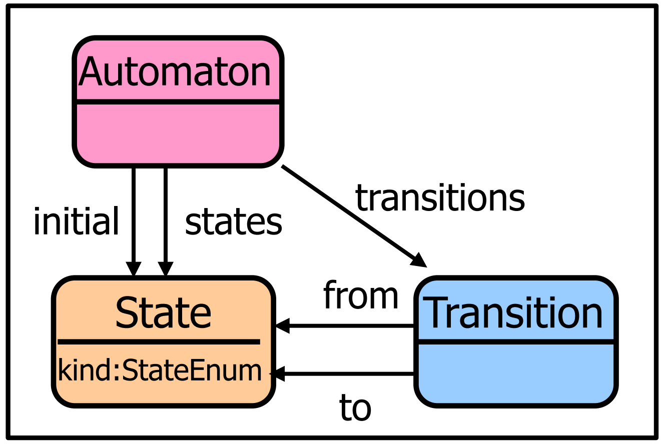


- **Task:** method to modify graphs!

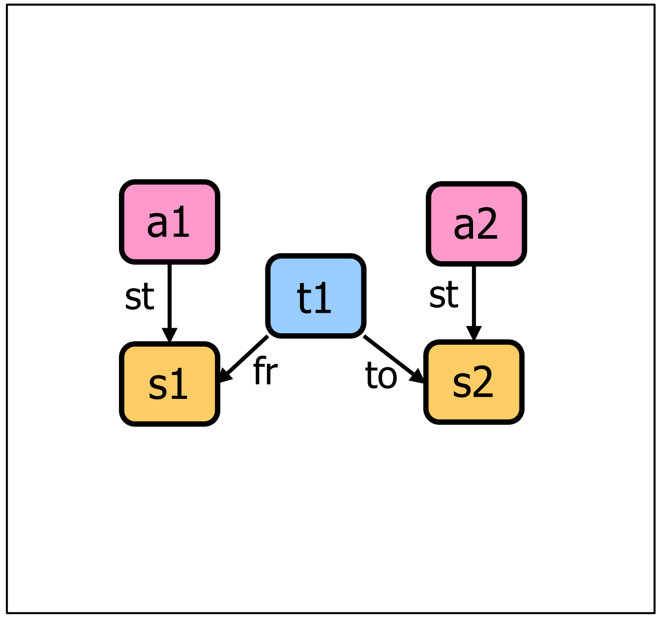
Graph pattern matching, Graph transformation

A simple example

Metamodel



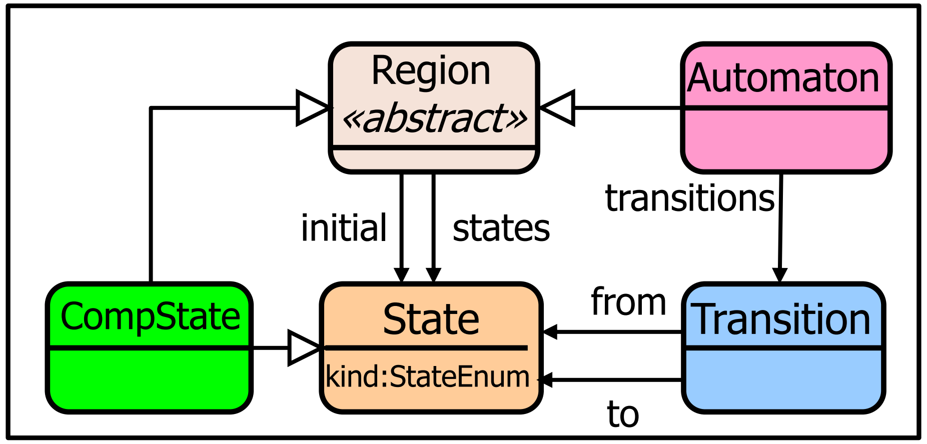
Violation example



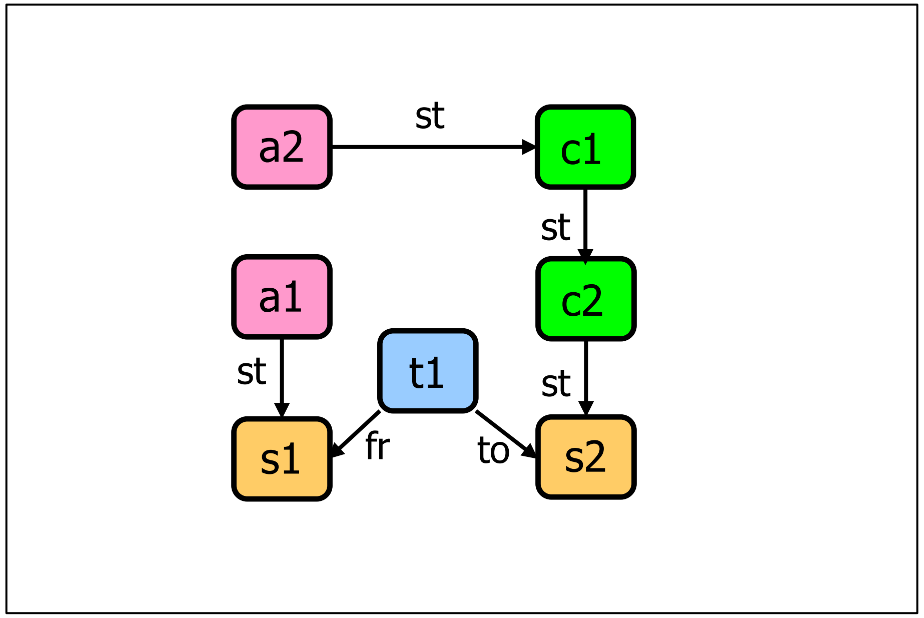
- Well-formedness constraint:
  - Transition source & target states must be owned by same automaton
- Goal: to find violations...
  - A violation is a *Transition*, whose „*from*” link points to a *State s1*, and „*to*” link points to a *State s2*, where the automaton of *s1* is not the automaton of *s2*

A more complex example

Metamodel



Violation example



- Well-formedness constraint:
  - Transition source & target states must be owned by same automaton
- Goal: to find violations...
  - A violation is a *Transition*, whose „*from*" link points to a *State s1*, and „*to*" link points to a *State s2*, where the automaton of *s1* is not the automaton of *s2*

Programmatic traversal vs. queries

- **Goal:** find constraint violations in model

Traverse model in general-purpose language

```
for (Automaton automaton : automatons) {
  for (Transition transition : automaton.getTransitions()) {
    State sourceState = transition.from;
    // which automaton defines this state?
    Automaton sourceAutomaton = null;
    for (Automaton candidate : automatons) {
      if (candidate.getStates().contains(sourceState)) {
        sourceAutomaton = candidate;
        break;
      }
    }
    // ... do the same for targetState, then
    if (sourceAutomaton != targetAutomaton)
      // report violation
  }
}
```

„simple example"

Programmatic traversal vs. queries

- Goal: find constraint violations in model
  - Traverse model in general-purpose language
  - Use a **Query DSL**
    - More concise
    - **Declarative** functional specification of the query
    - Freely interpreted by **query engine** (e.g. optimization)
    - Can be platform-independent
- Validation is just one use cases for **model queries**
  - Derived features
  - M2M/M2T Transformation, Simulation
  - …

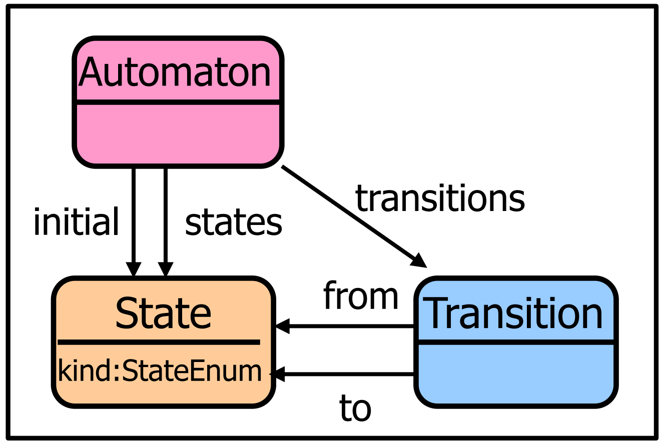
Query Language Styles

- SQL-like (relational algebra)
  - Example: EMF Query
  - ☺ Good for attribute restrictions
  - ☹ Not very concise for relationships (many joins)
- Functional style
  - Example: OCL
  - Somewhat declarative

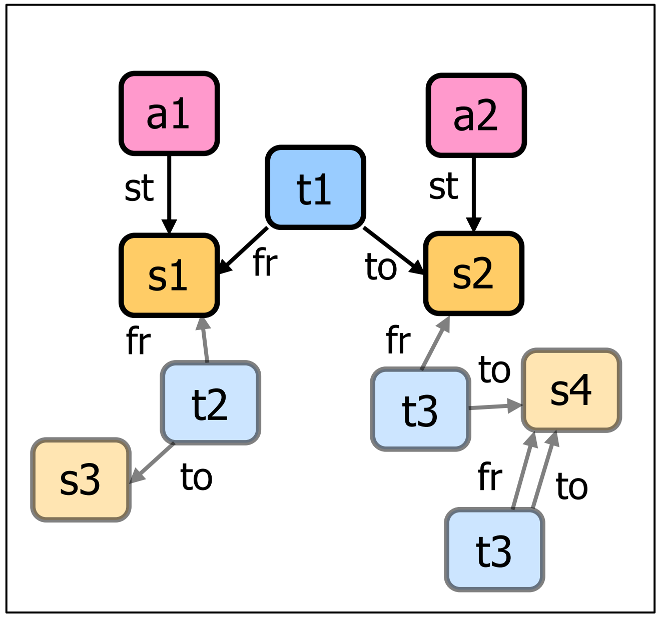
```
context Transition inv:
  Automaton.allInstances()->forAll(a |
    a.states->includes(self.from) =
    a.states->includes(self.to)
  );
```

- Logic style
  - Domain relational calculus / graph patterns / Datalog
  - Even more declarative

Metamodel



Violation example



A violation is a *Transition*,
whose „*from*” link points to a *State s1*
and „*to*” link points to a *State s2*,
where the automaton of *s1*
is not the automaton of *s2*

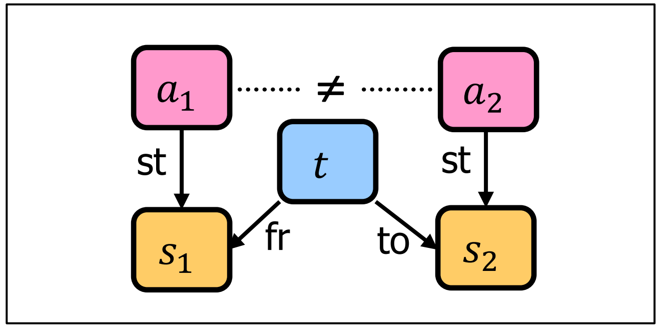
in formal logic (Domain Relational Calculus)

$$\{t \mid \exists s_1, s_2, a_1, a_2 : Transition(t) \wedge from(t, s_1) \wedge to(t, s_2) \wedge states(a_1, s_1) \wedge states(a_2, s_2) \wedge a_1 \neq a_2\}$$

```
violates(t):-
Transition(t),from(t,s1),to(t,s2),
states(a1,s1),states(a2,s2),a1=\=a2
```

Datalog-like query languages

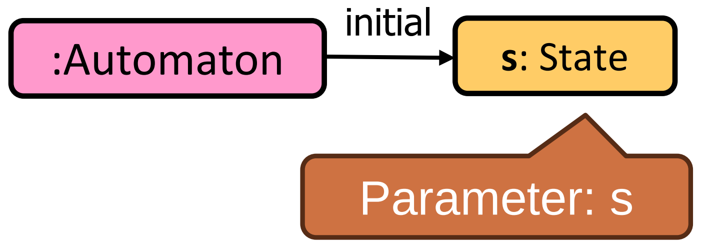
Query engine



Pattern

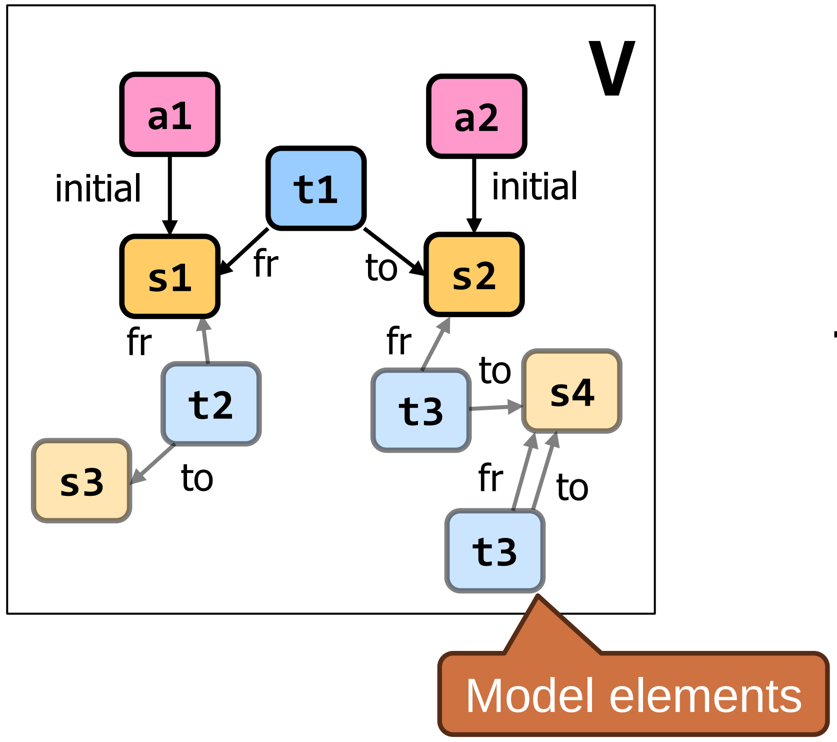
Pattern matching

Pattern



+

Example model



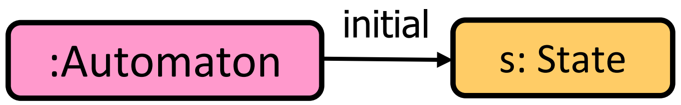
→

Matches

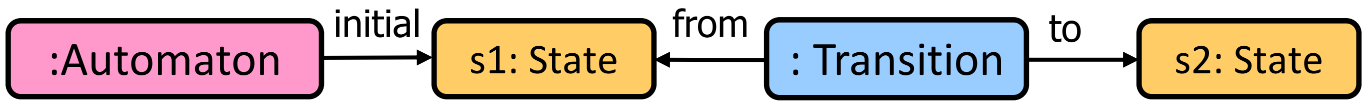
s
s1
s2

Examples

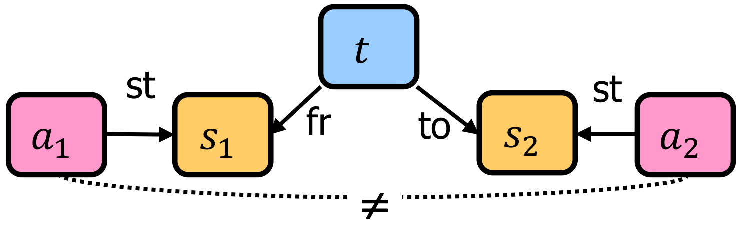
- **Simple example:** Initial states in the model



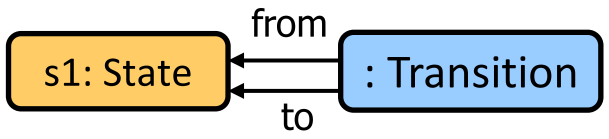
- **Chain (∧):** Second states in the model

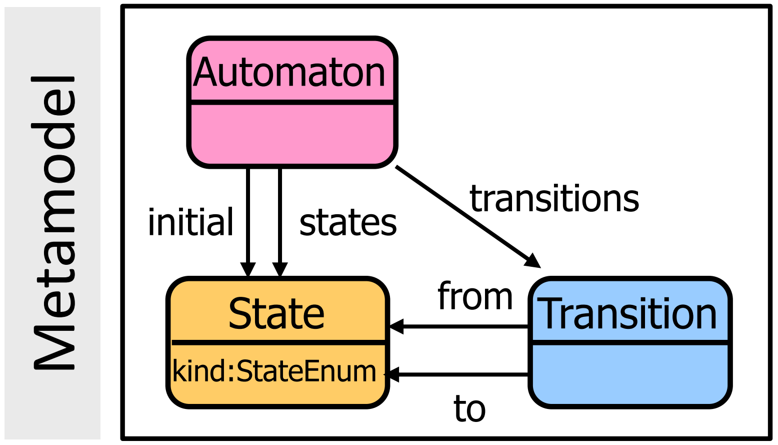


- ≠ **:** Transition across automata



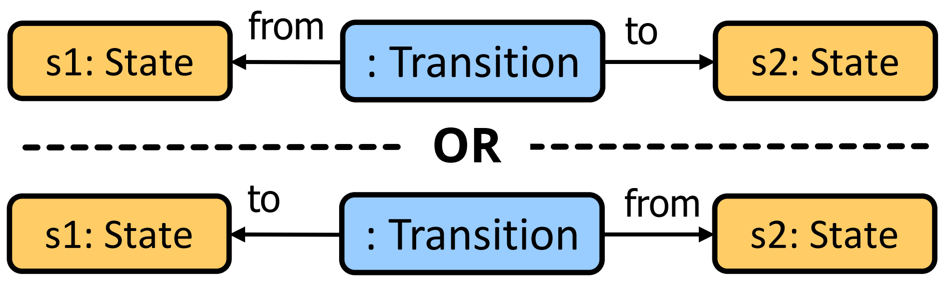
- =: Loop edge

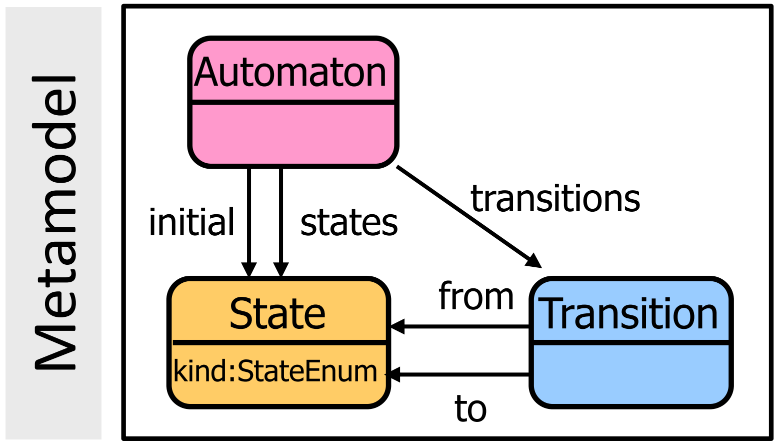




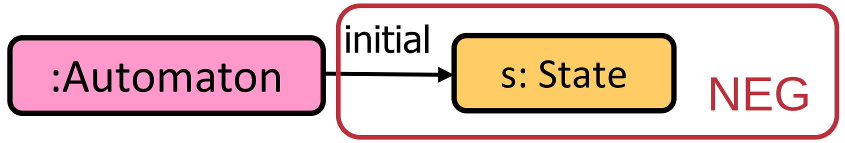
Examples 2

- (∨): Two states are connected

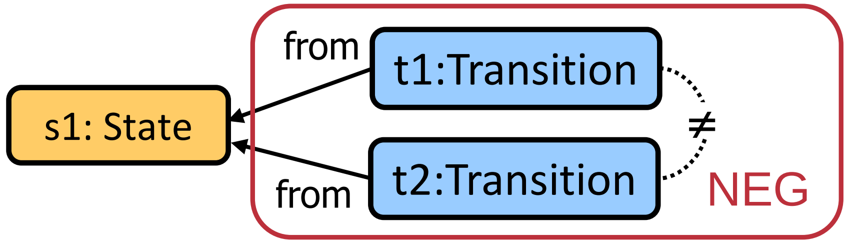




- (¬, Negative Application Condition):
  - automaton without initial state



  - a state from whose initial state no two transactions leave (deterministic)

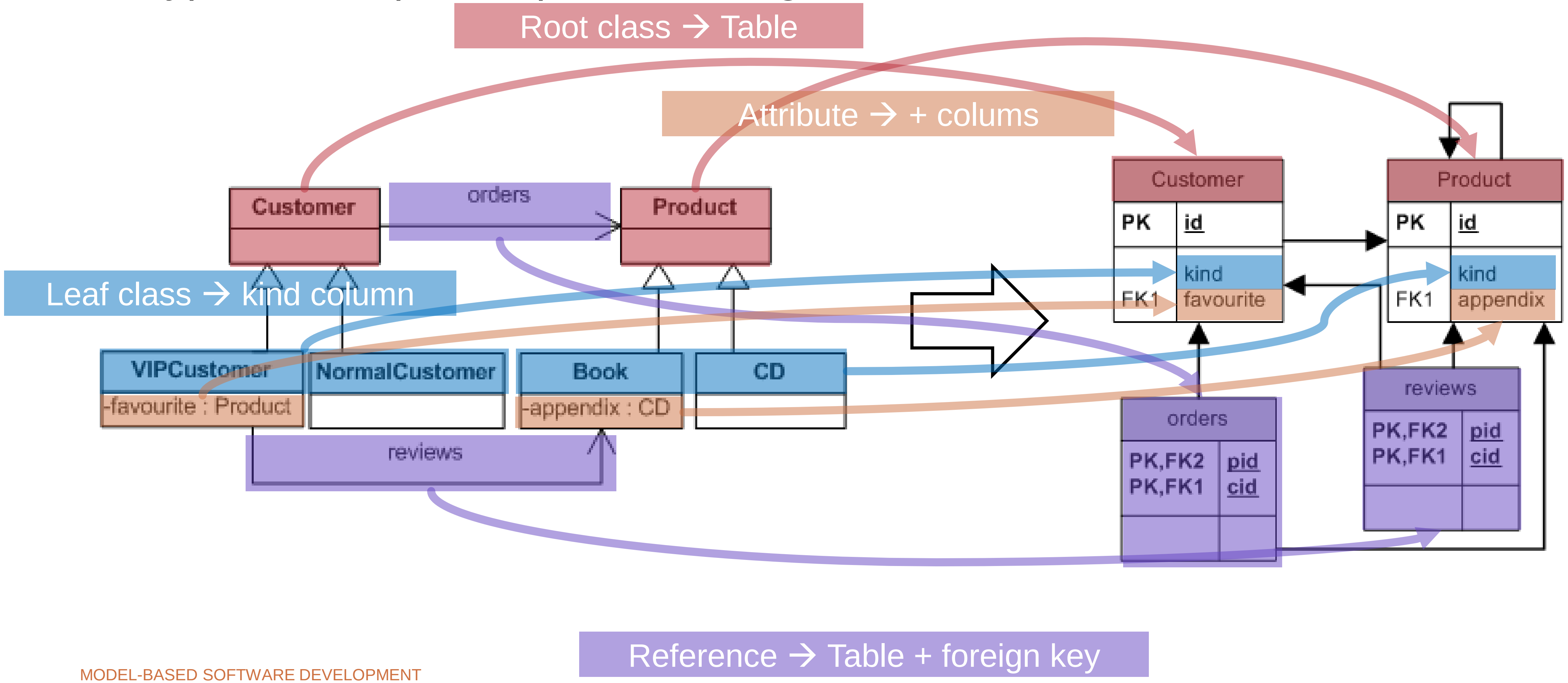


Graph pattern matching, Graph transformation

- Definitions
- Graph pattern matching
- **Model transformations**
- Incremental transformations
- Design space exploration

Example Transformation

- Typical example: map a class diagram to database tables!



Example Transformation

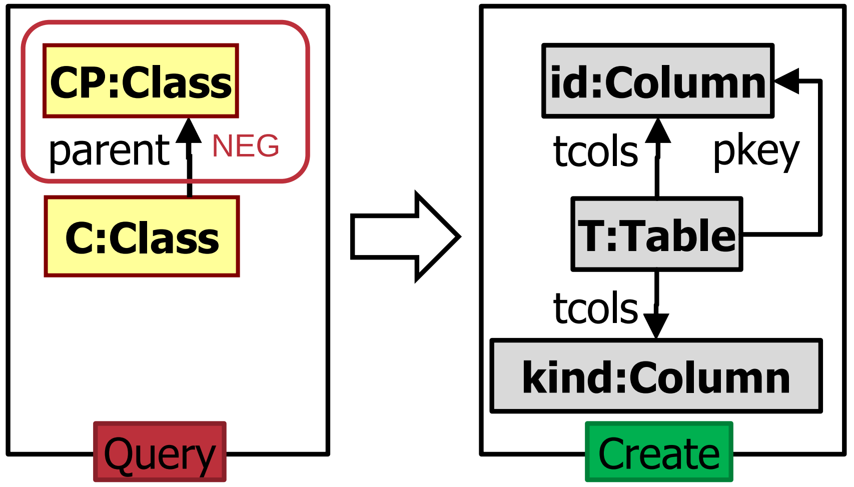
- How would we solve the problem of creating tables representing root classes?

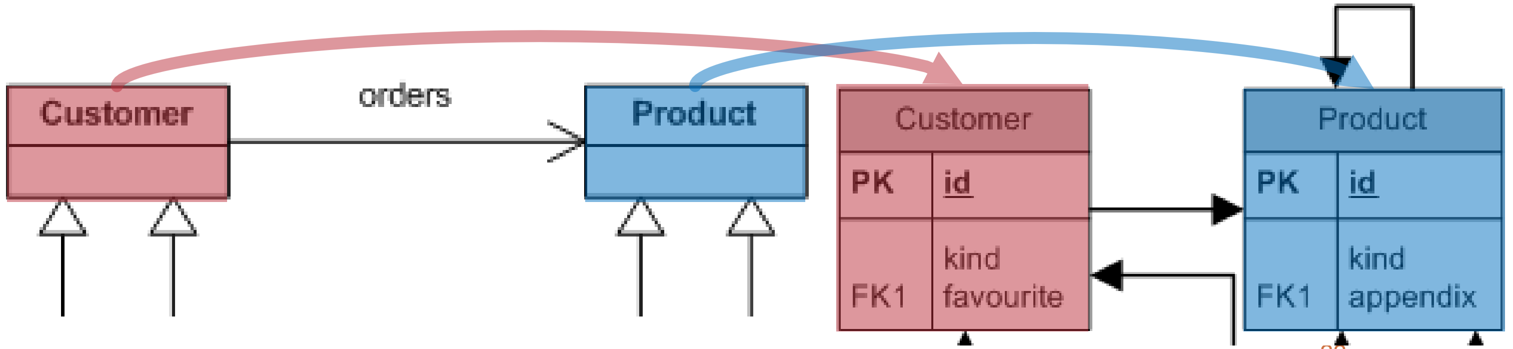
1. Query the root classes
(class that has no ancestor)

2. Create the tables and with them the necessary columns

3. Repeat as long as we can

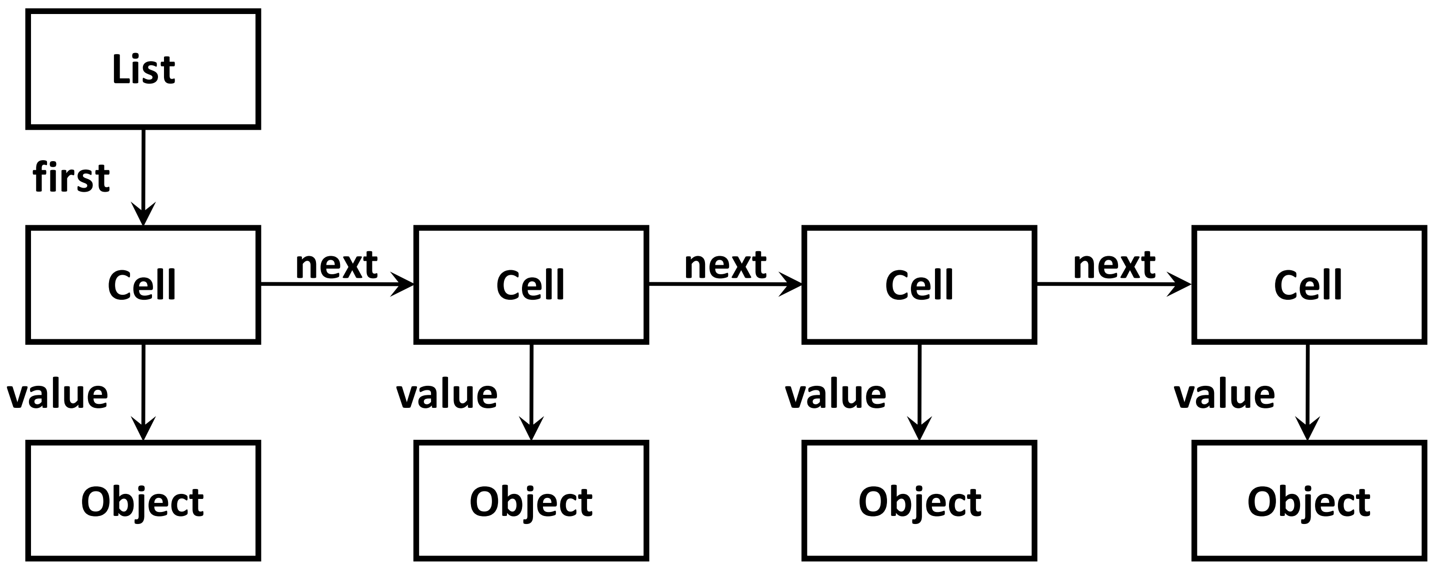
- Goal: To formulate the whole transformation with similar rules





Graph transformation

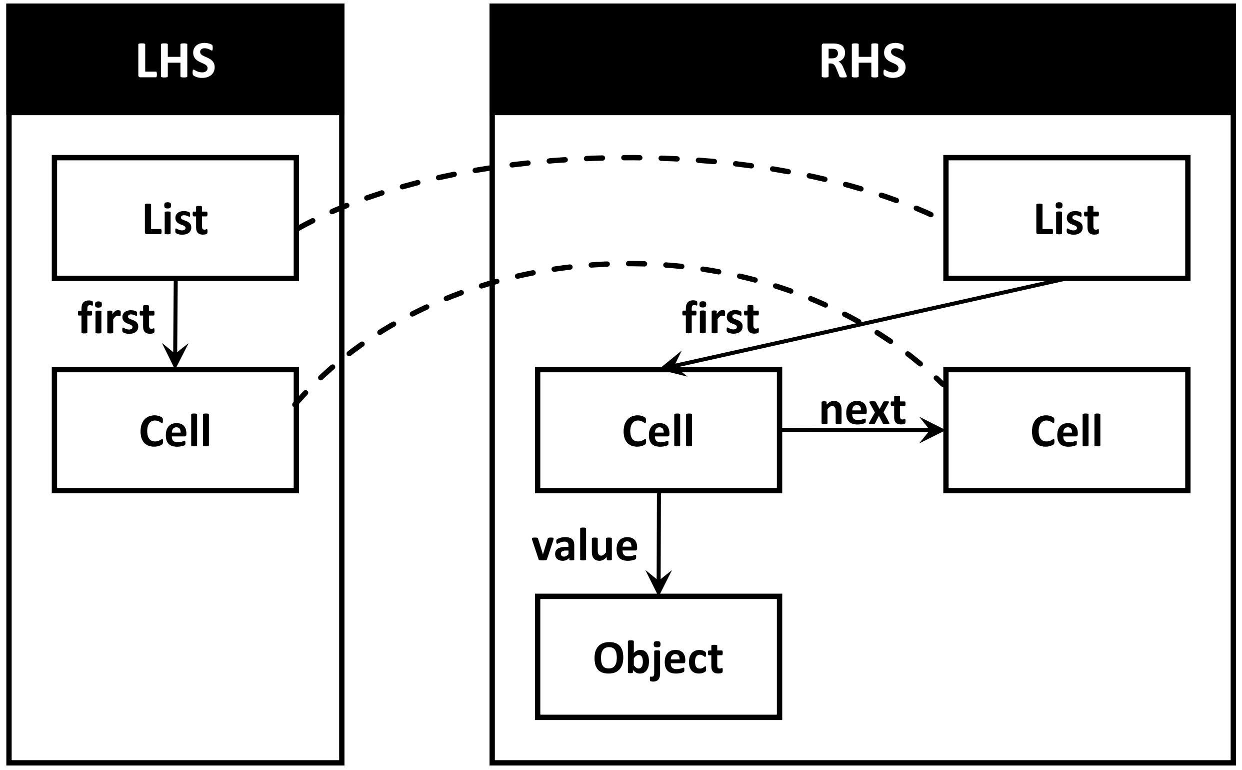
- Model = Labelled Graph



Graph Transformation rule

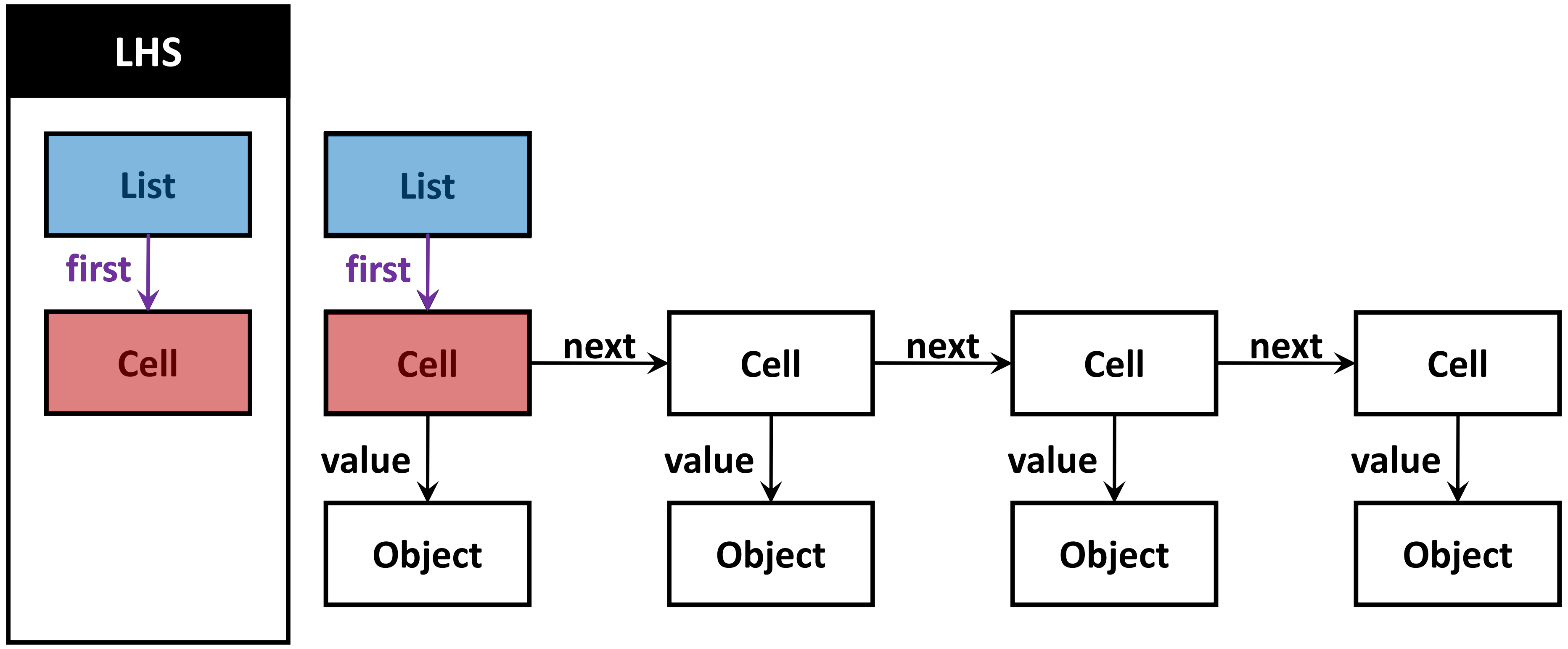
- Graph rewriting rule, defined with two graphs

Left Hand Side Right Hand Side



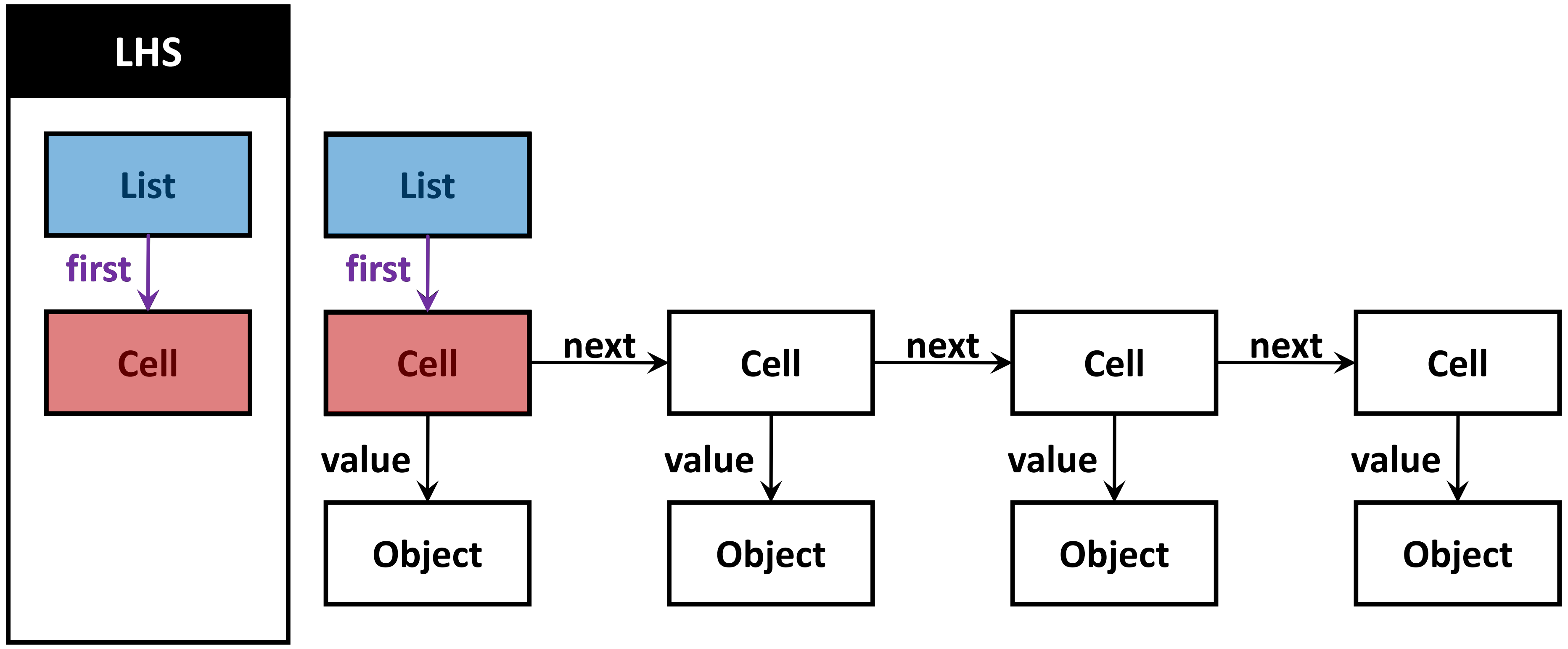
Graph Transformation: Pattern matching

- **Matching:** find the subgraphs containing LHS in the source graph



Graph Transformation: Pattern matching

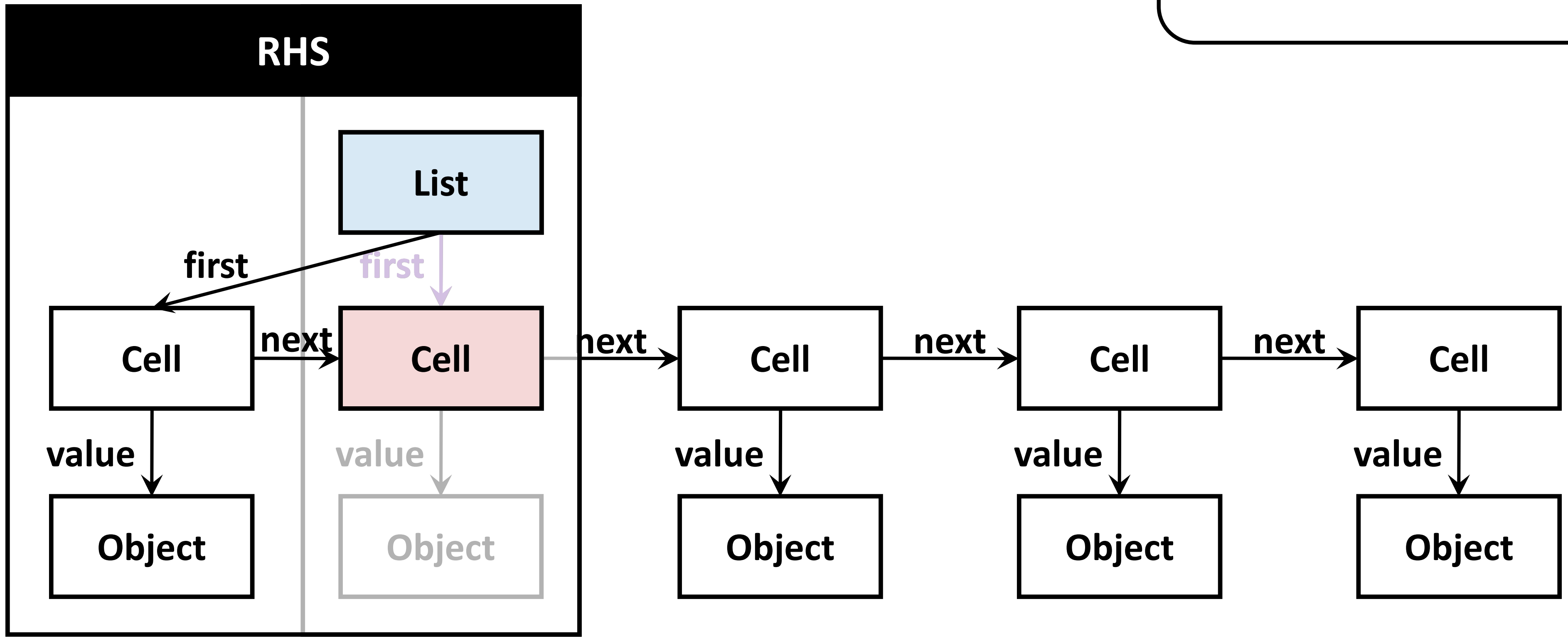
- **Matching:** find the subgraphs containing LHS in the source graph



Graph Transformation: Execution of rewriting

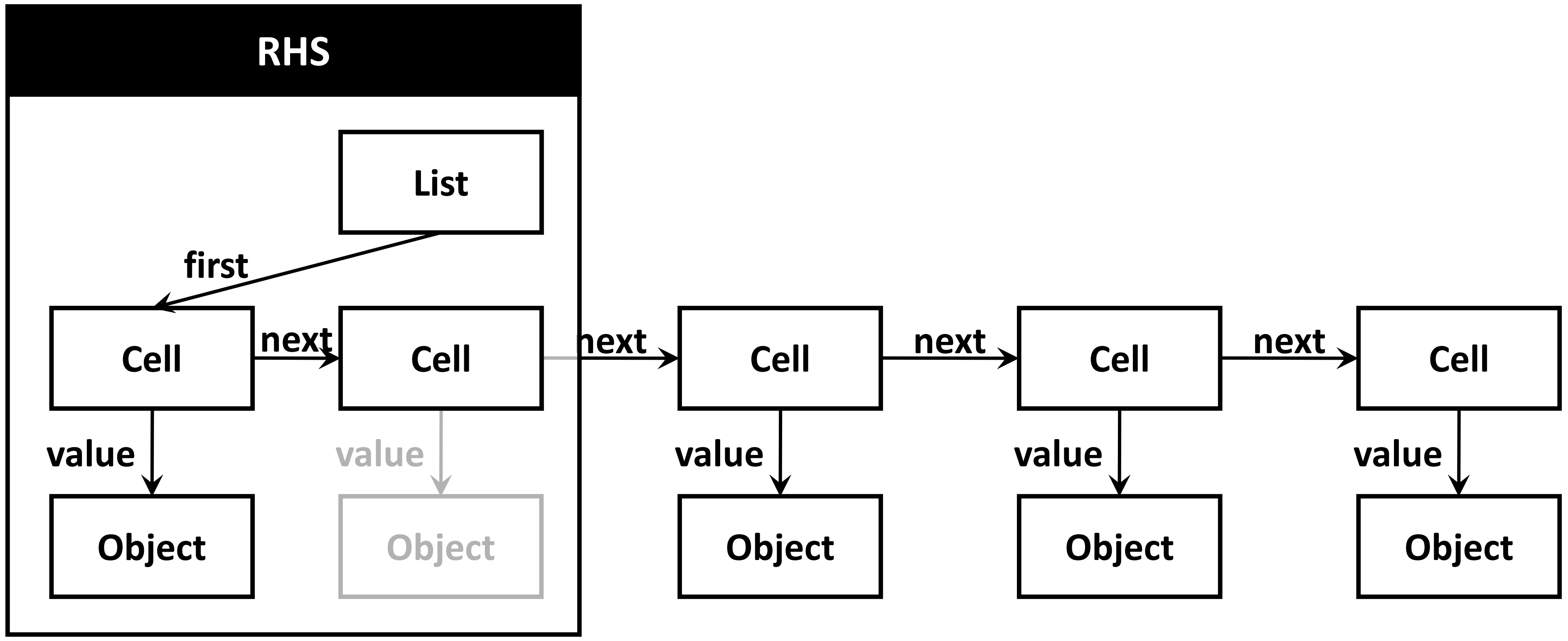
- Rewriting the graph by the match: replace LHS with RHS.

LHS\RHS → Delete
RHS\LHS → Insert
RHS∩LHS → Leave it



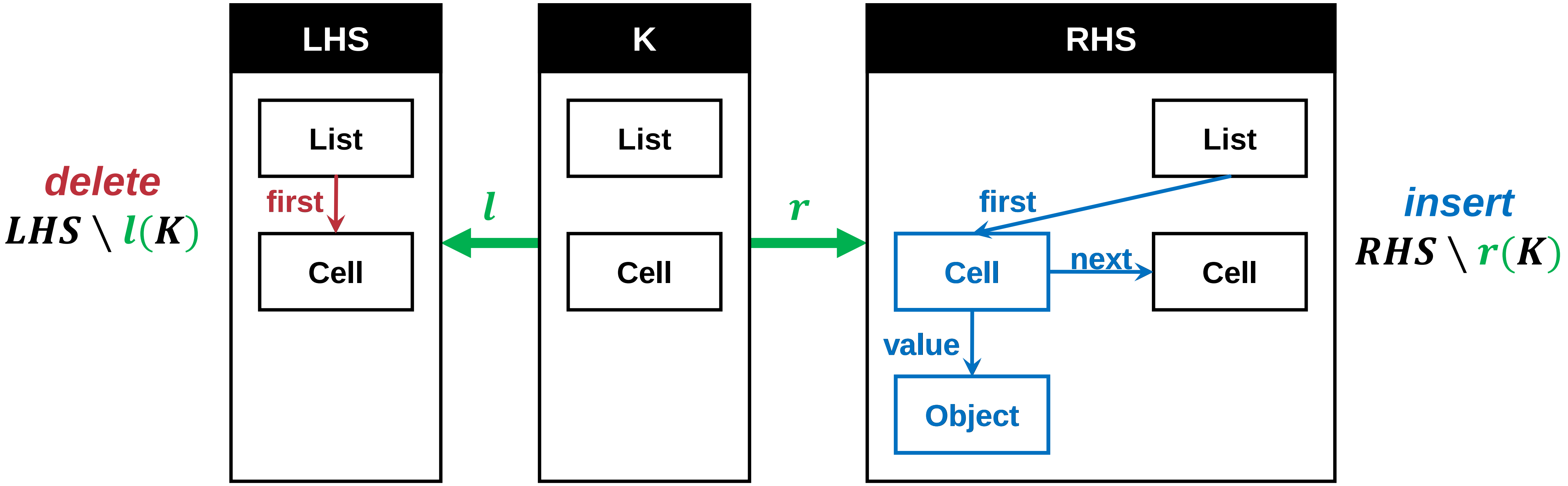
Graph Transformation: Execution of rewriting

- We get a new graph

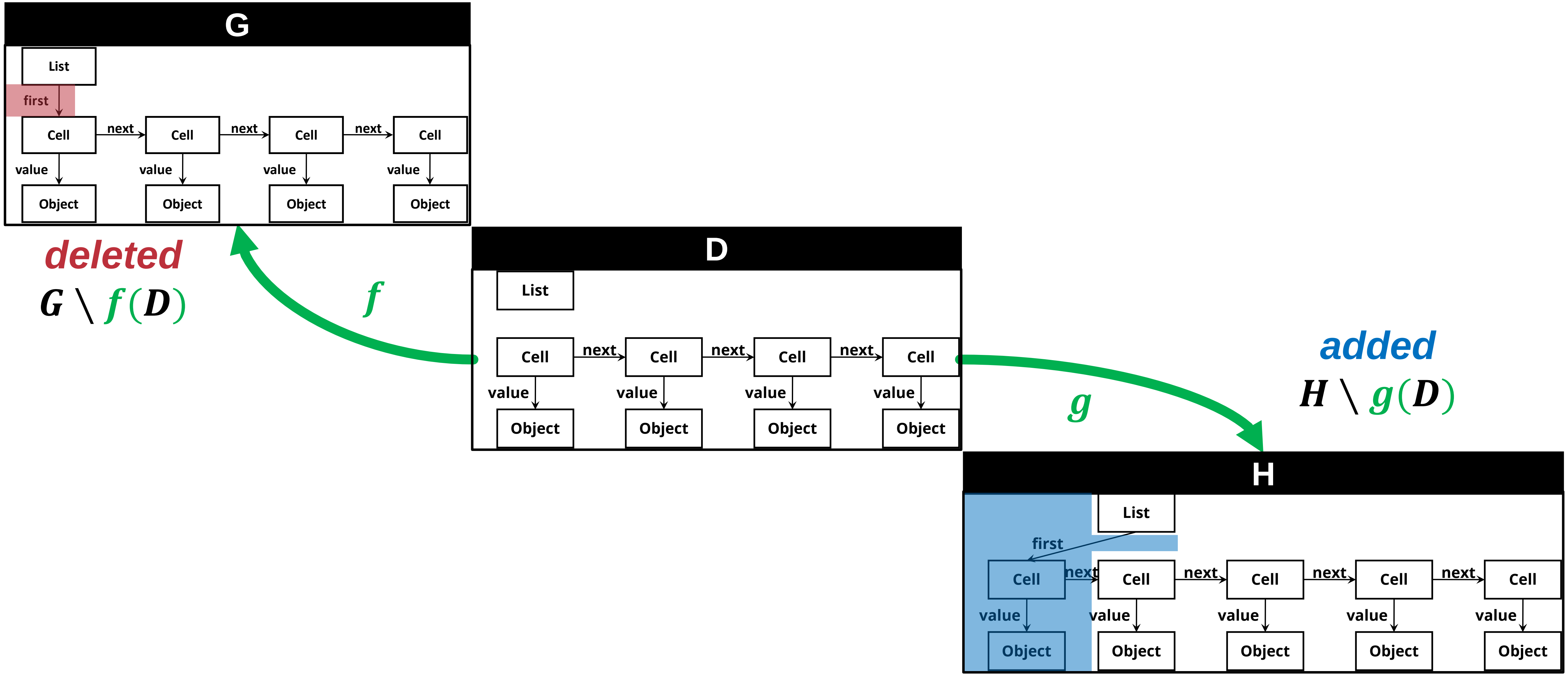


Anatomy of graph transformation

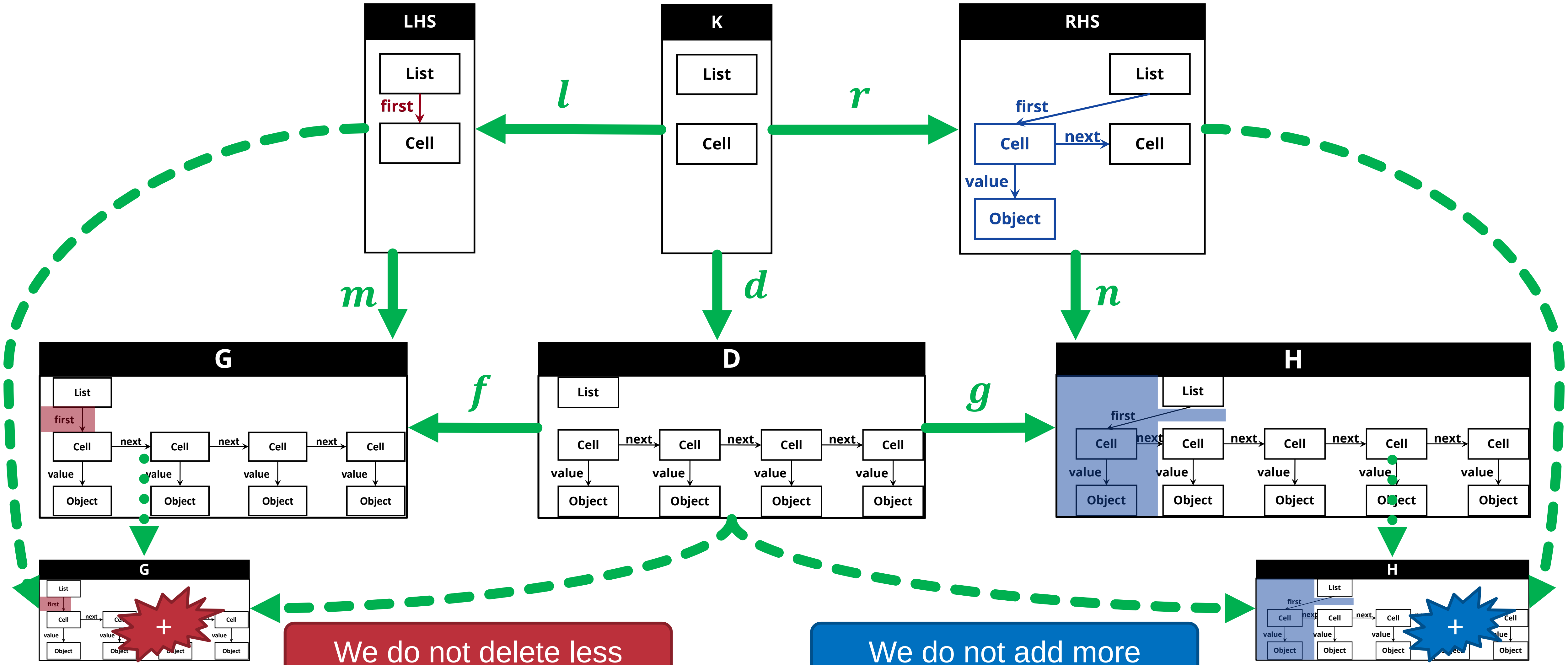
- Let us examine which graph can be matched to which graph during the transformation.



Anatomy of transformed models



Complete anatomy



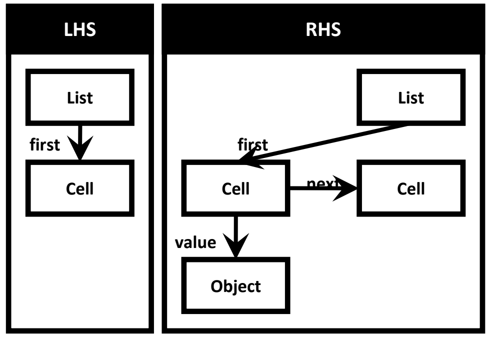
We do not delete less

We do not add more

Graph Transformation

- Formulating rules for rewriting models
- Extending grammar rules

List → List,Cell **vs**

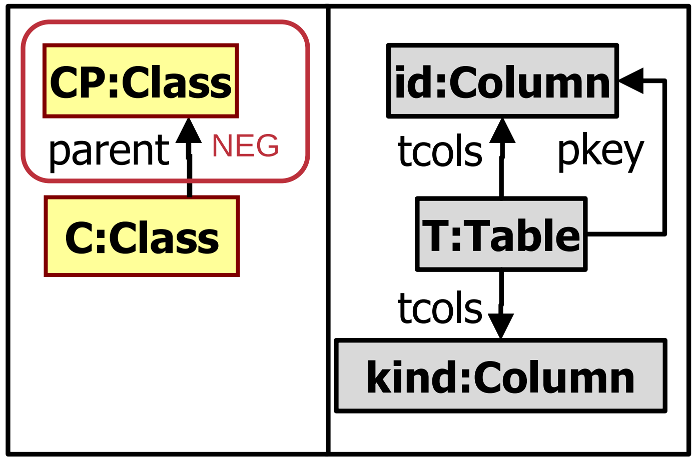


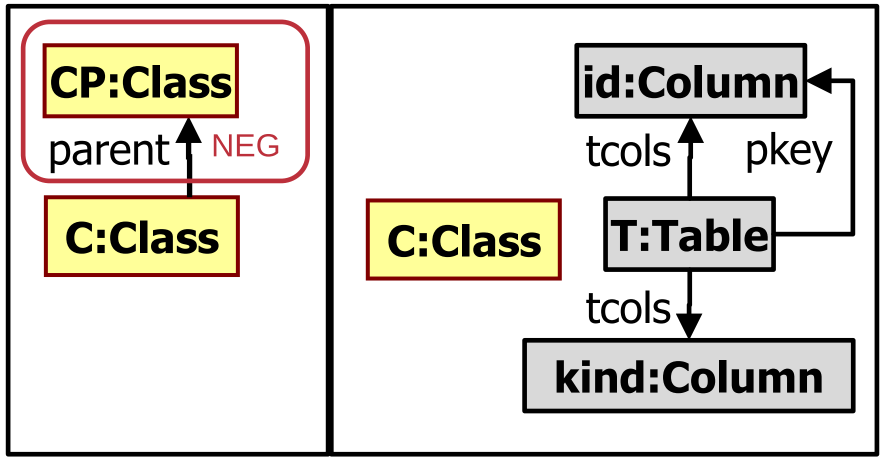
- Clear but mathematically precise formalism

  (Termination, Ordering, Confluence, …)
- Tool support (see previous practice)

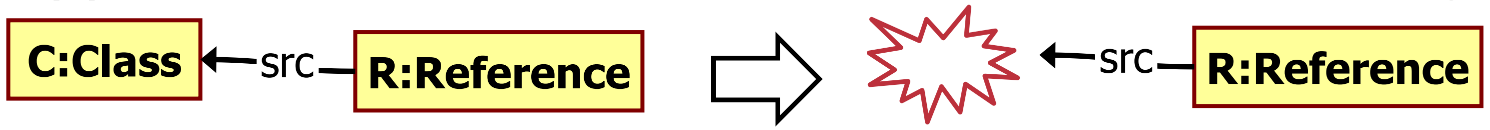
“Dangling edge” problem

- Mapping of ancestor classes (with and without deletion)





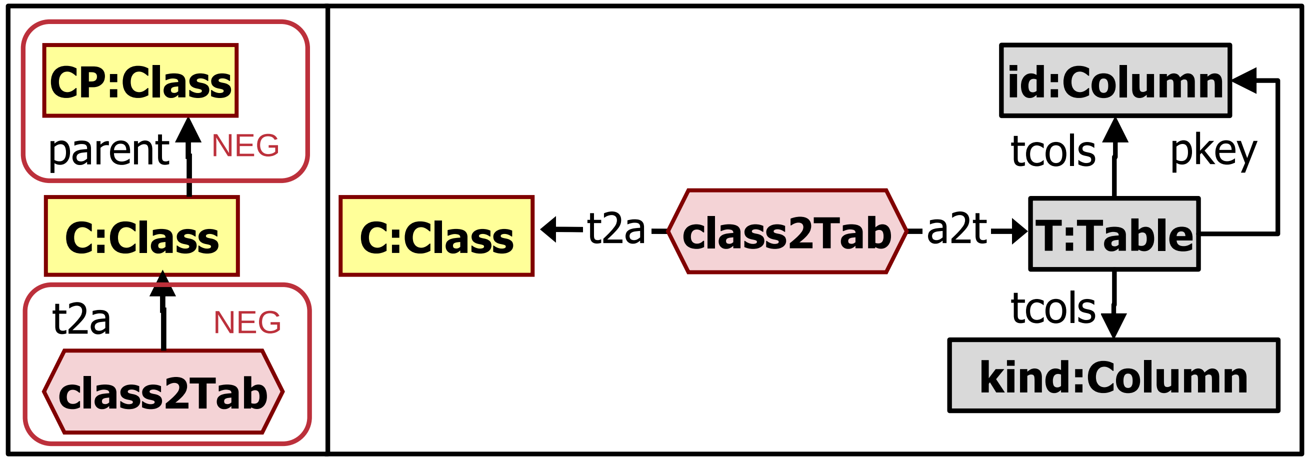
- What happens if we delete an element that still has an edge pointing to it?



- Resolving „dangling edges”: Delete edges / Undo transformation

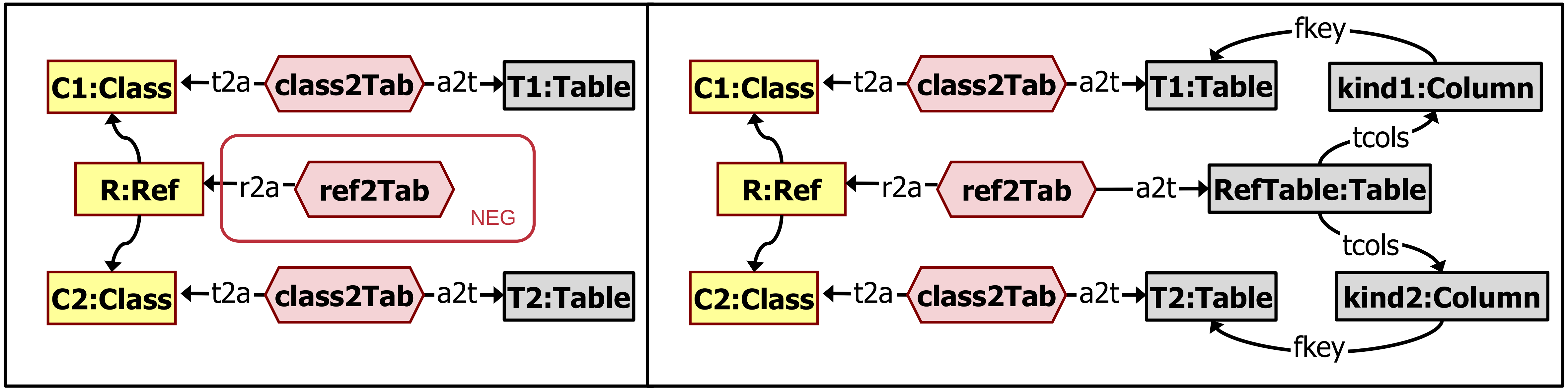
Examples

- Mapping of ancestor classes with traceability :
  - > *Find an ancestor class*
  - > *that has not yet been mapped,*
  - > *then map it.*



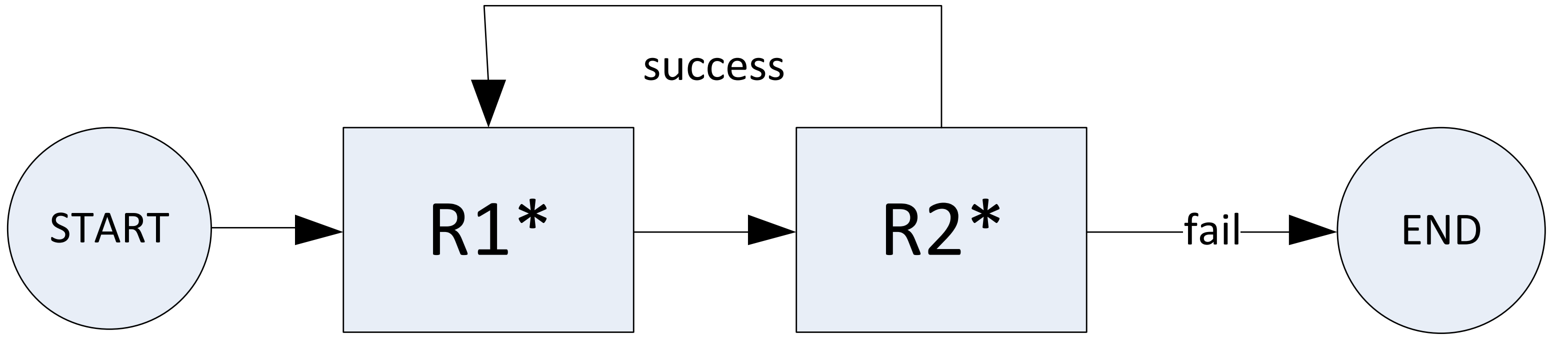
Examples

- Mapping references

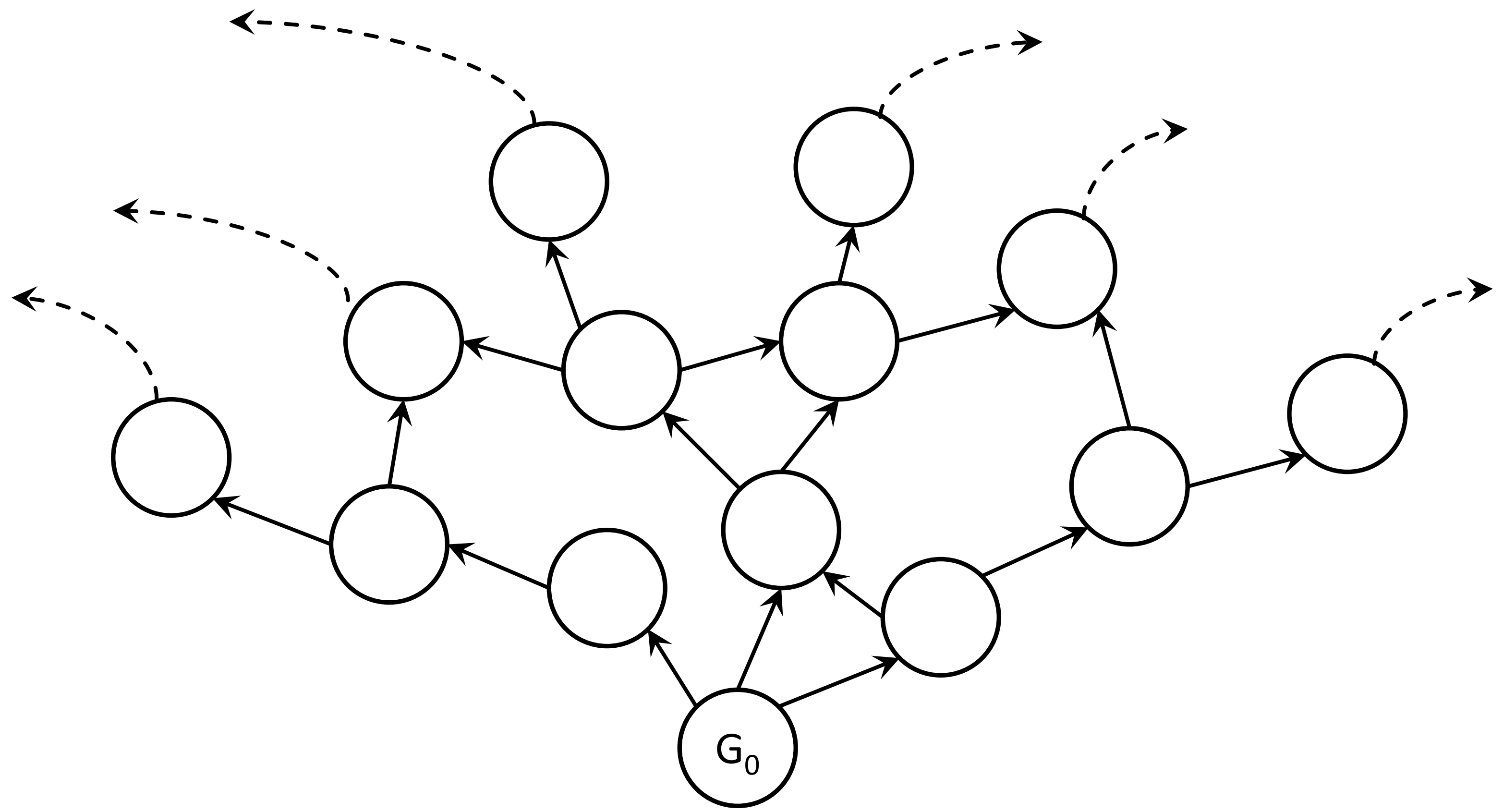


Control mechanisms

- In what order should the rules be executed?
- Multiple options, see previous lecture.
- But for example:
  - Fire arbitrary transformations as long as possible (~ default)
  - Fire all possible transformations once
  - control graph (explicit control)

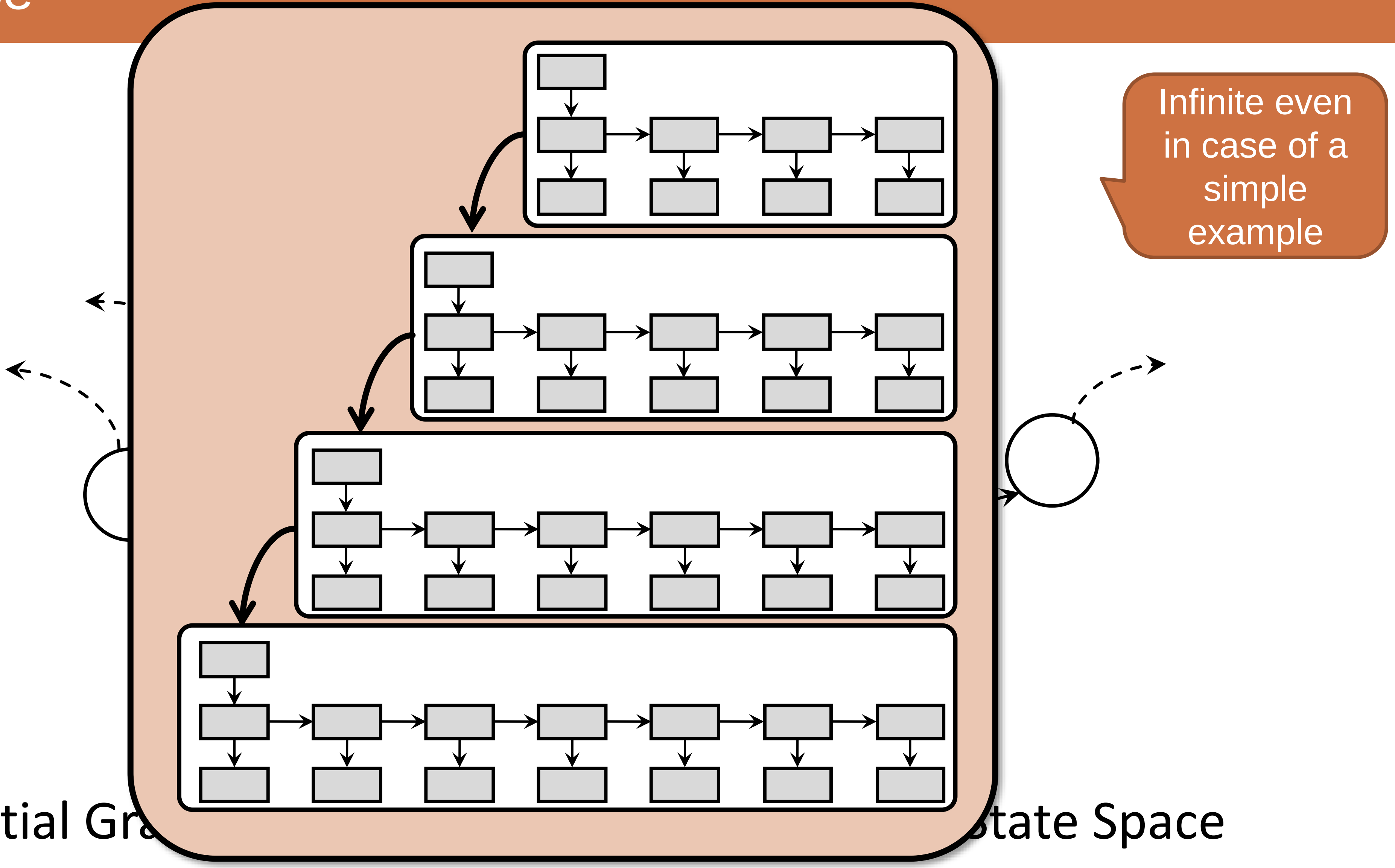


State Space



Initial Graph + GT rules → **(Tipically infinite)** State Space

State Space


Infinite even in case of a simple example
Initial Gra
tate Space

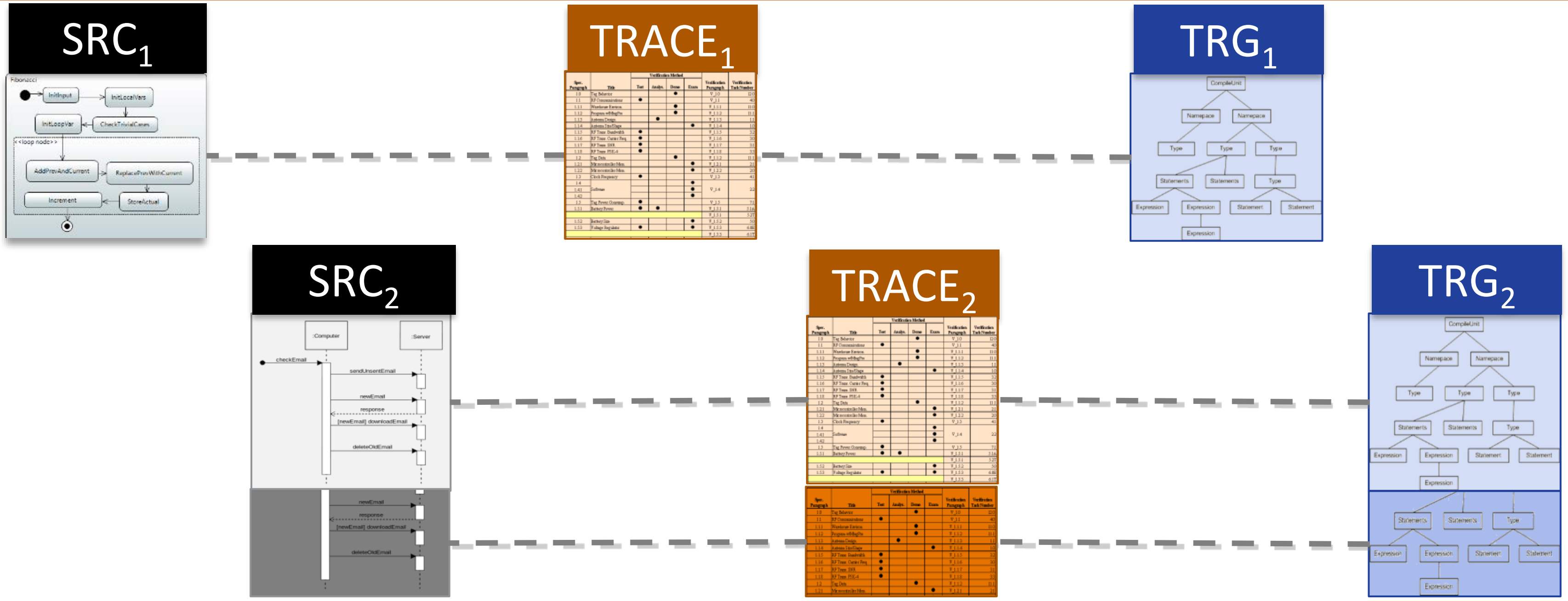
Types of model transformations

- By number of inputs and outputs
  (In-place vs out-place)
- By the language
  (Endogenous vs exogenous)
- By the direction
  (unidirectional vs bidirectional)

Graph pattern matching, Graph transformation

- Definitions
- Graph pattern matching
- Model transformations
- **Incremental transzformations**
- Design space exploration

No Incrementality: Batch Transformations

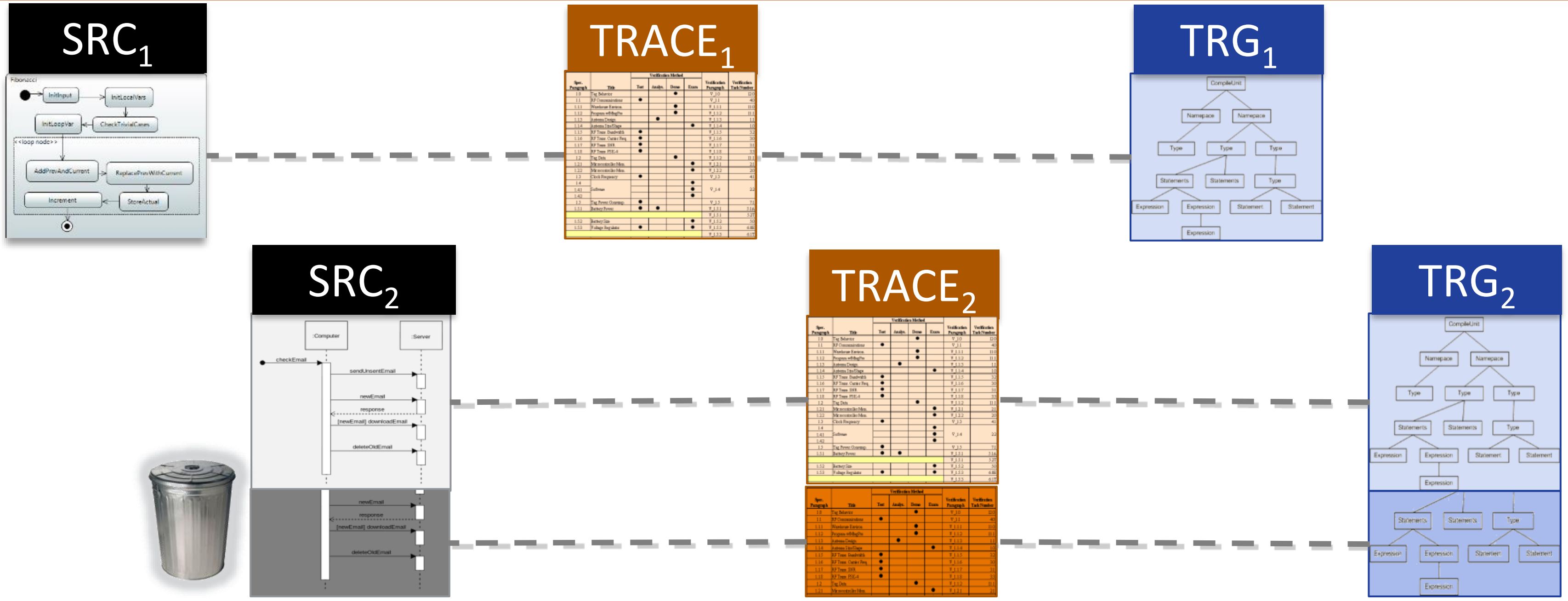


1. First transformation

2. Source model changes

3. Re-execute from scratch for all source models

Dirty Incrementality



Pros:

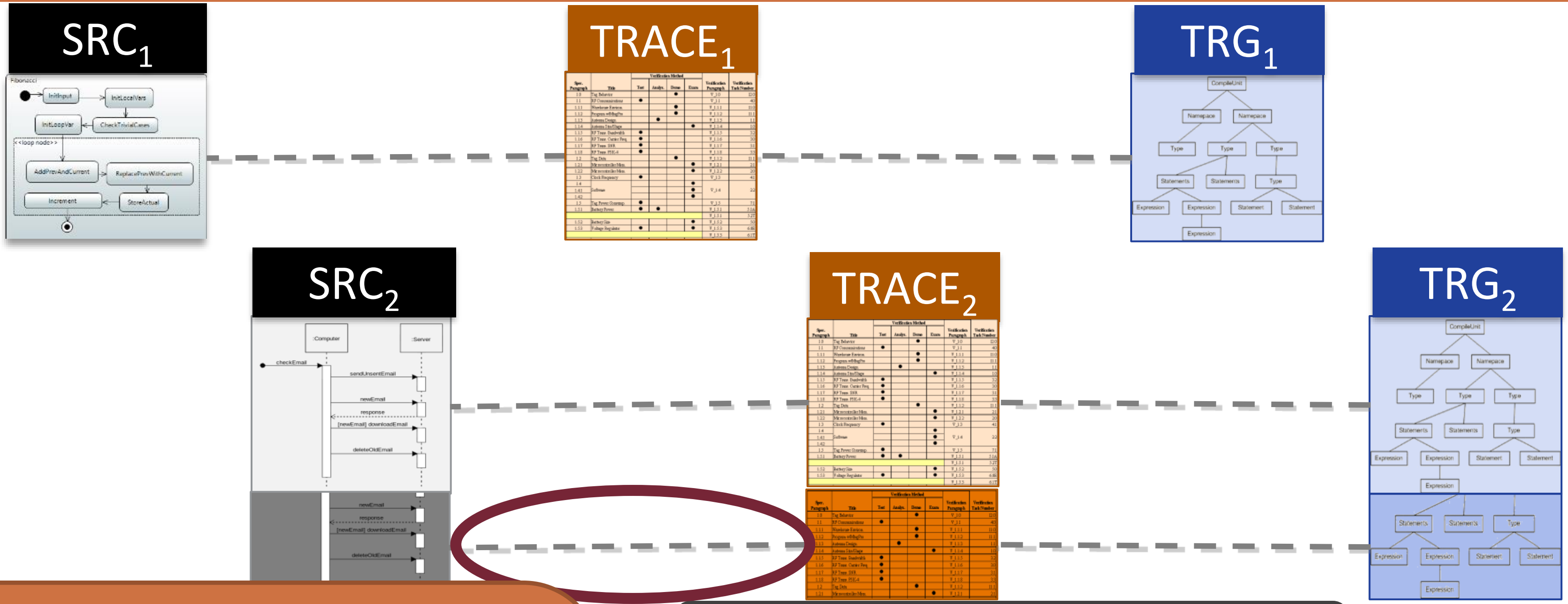
- Large-step incrementality
- Avoids continuous execution

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:

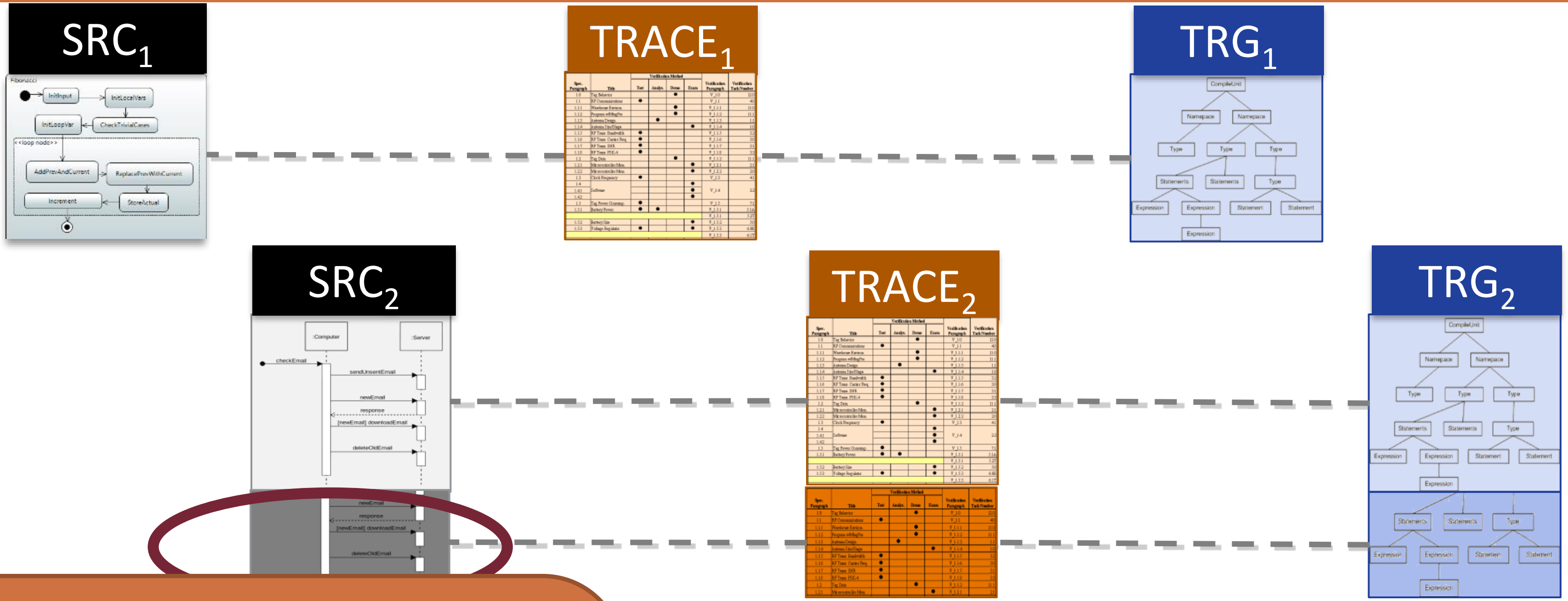
- Small-step incrementality
- Better performance

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.

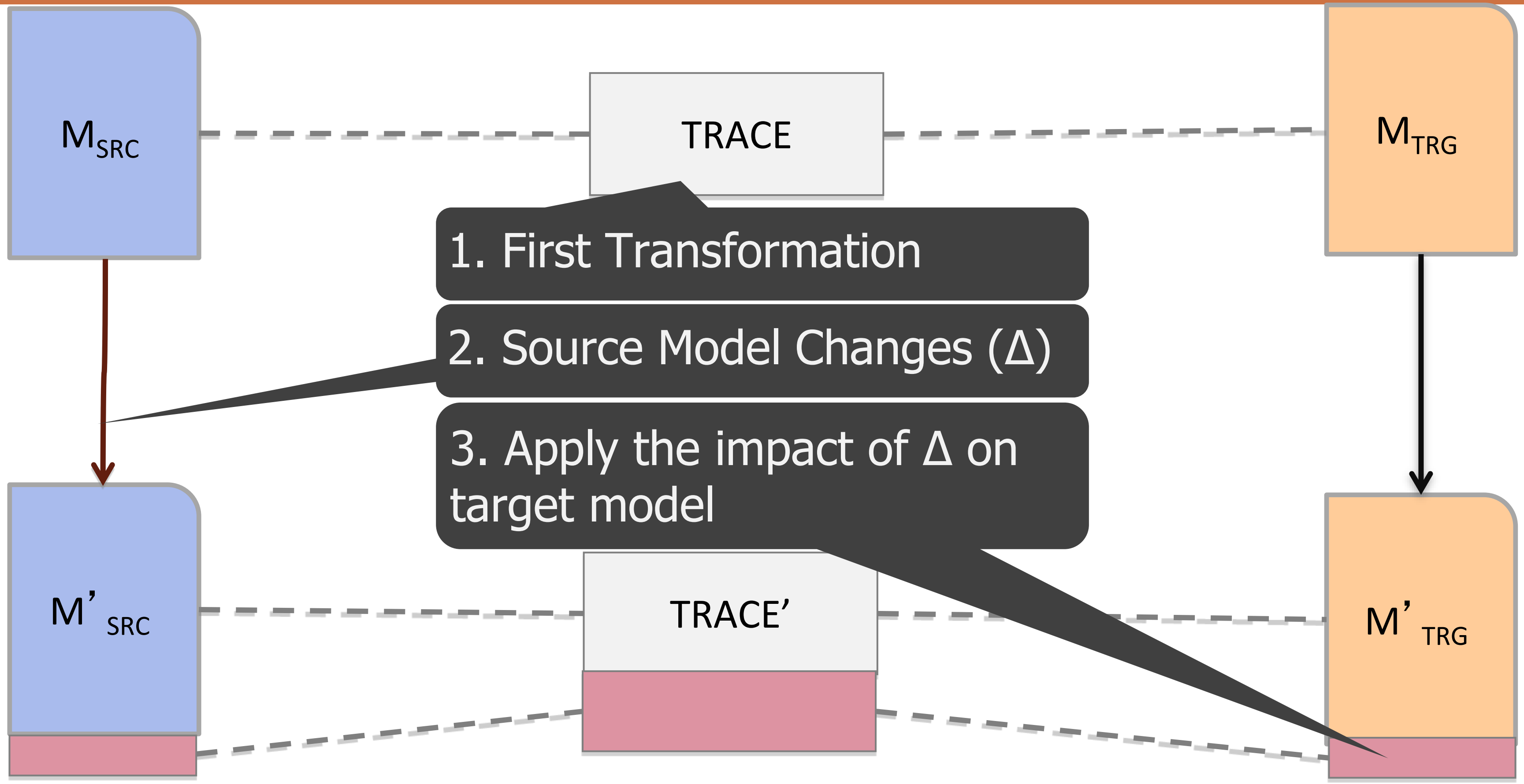
Cons:
- Language-level restrictions
- Must “listen” live

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

Incremental Forward Transformation

- Goals: reuse computations
  - **Target Incrementality**
    - …by reusing the unchanged parts of the target model
    - Unchanged model elements does not need to be modified
    - Anything derived from the unchanged part does not need to be regenerated (e.g., code)
    - Change does not propagate further
  - **Source Incrementality**
    - … by ignoring the unchanged parts of the source model
    - Incremental pattern matchers.

Incremental Forward Transformation Setup

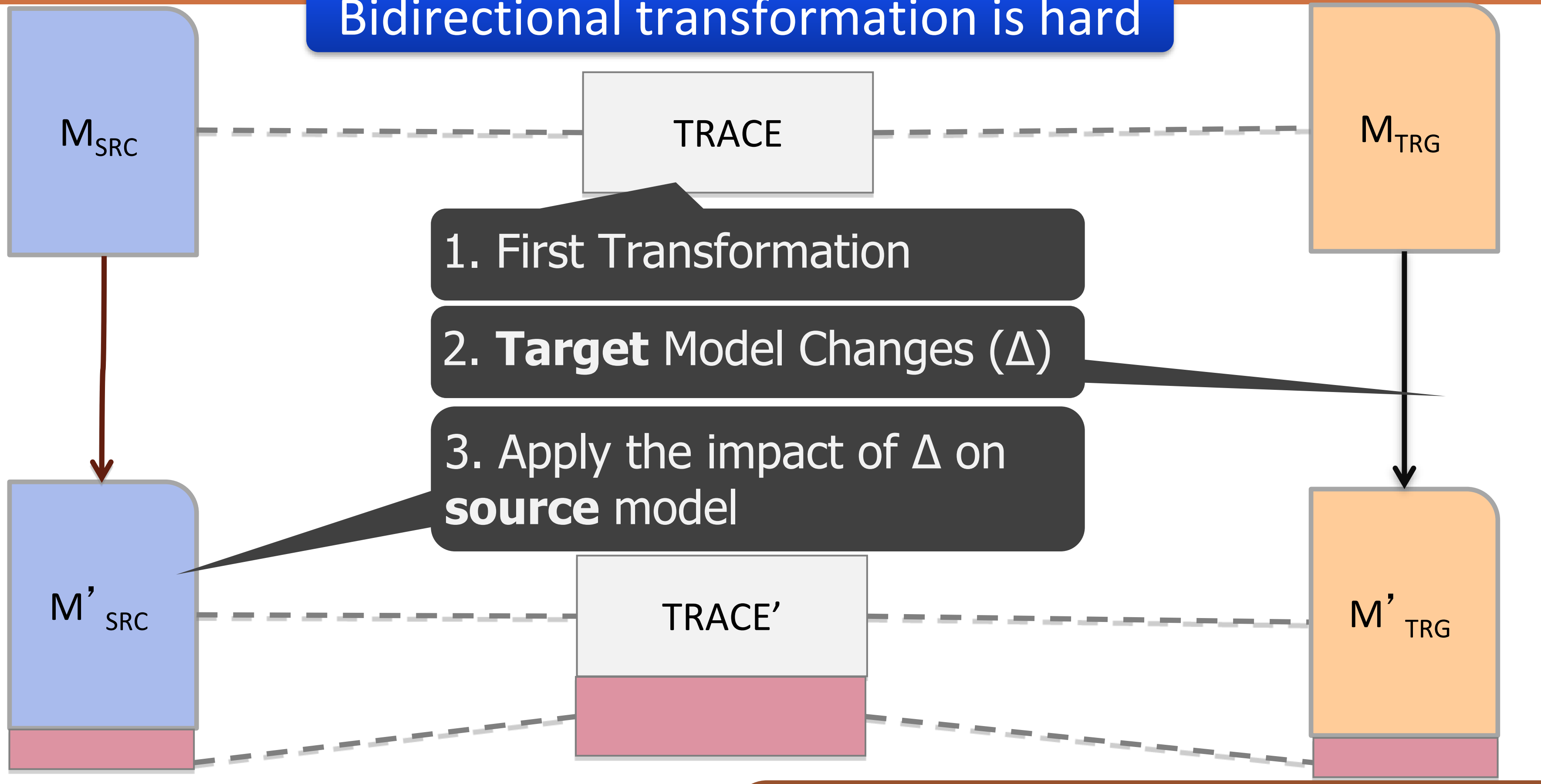


1. First Transformation
2. Source Model Changes (Δ)
3. Apply the impact of Δ on target model

Typical Scenarios:
- Incremental Transformation for
- Tool integration

Incremental backward transformation?

Bidirectional transformation is hard



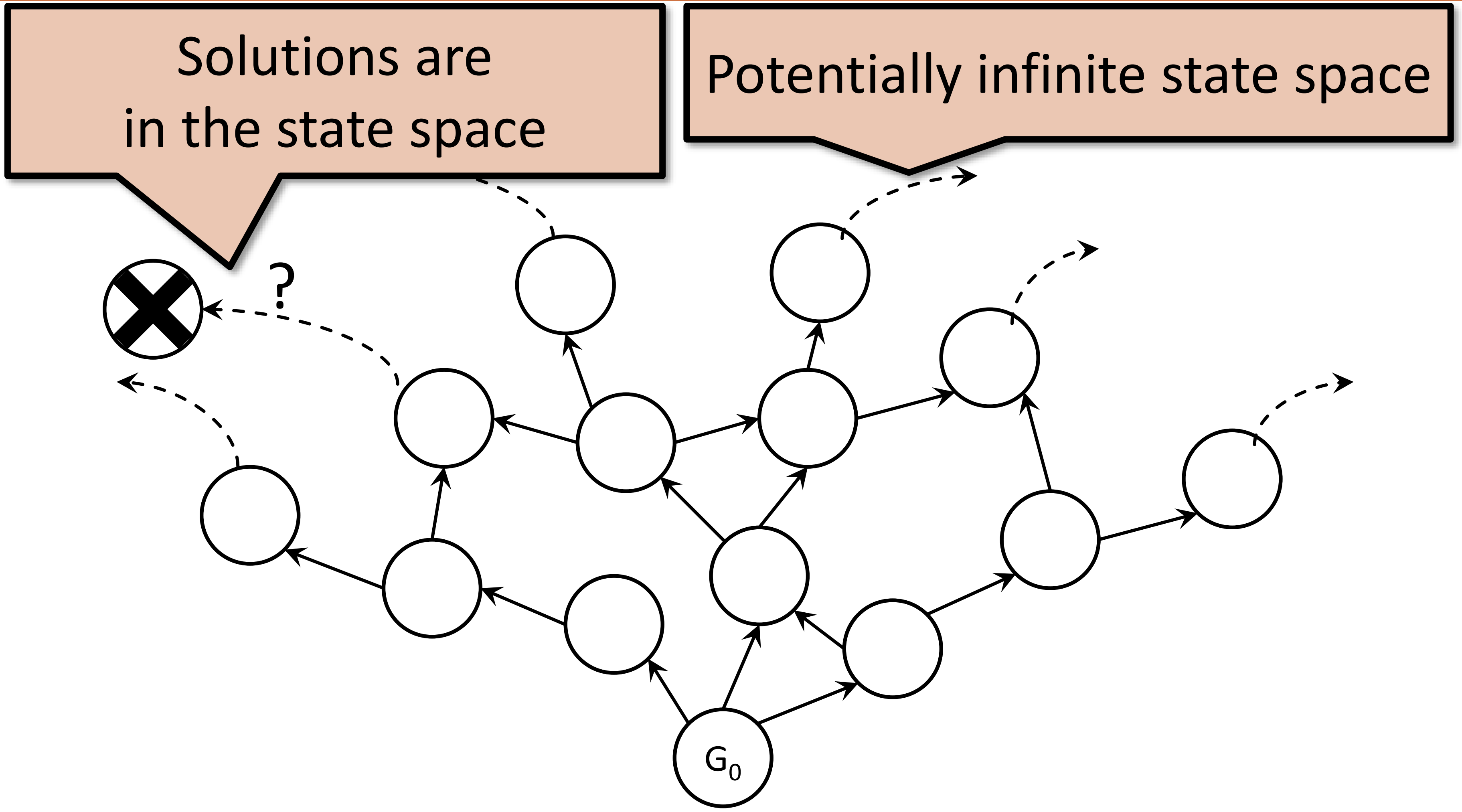
1. First Transformation
2. **Target** Model Changes (Δ)
3. Apply the impact of Δ on **source** model

Some related work:
A. Schürr, P. Stevens, N. Foster, T. Hettel, Cicchetti&Pierantonio, Czarnecki&Diskin

Graph pattern matching, Graph transformation

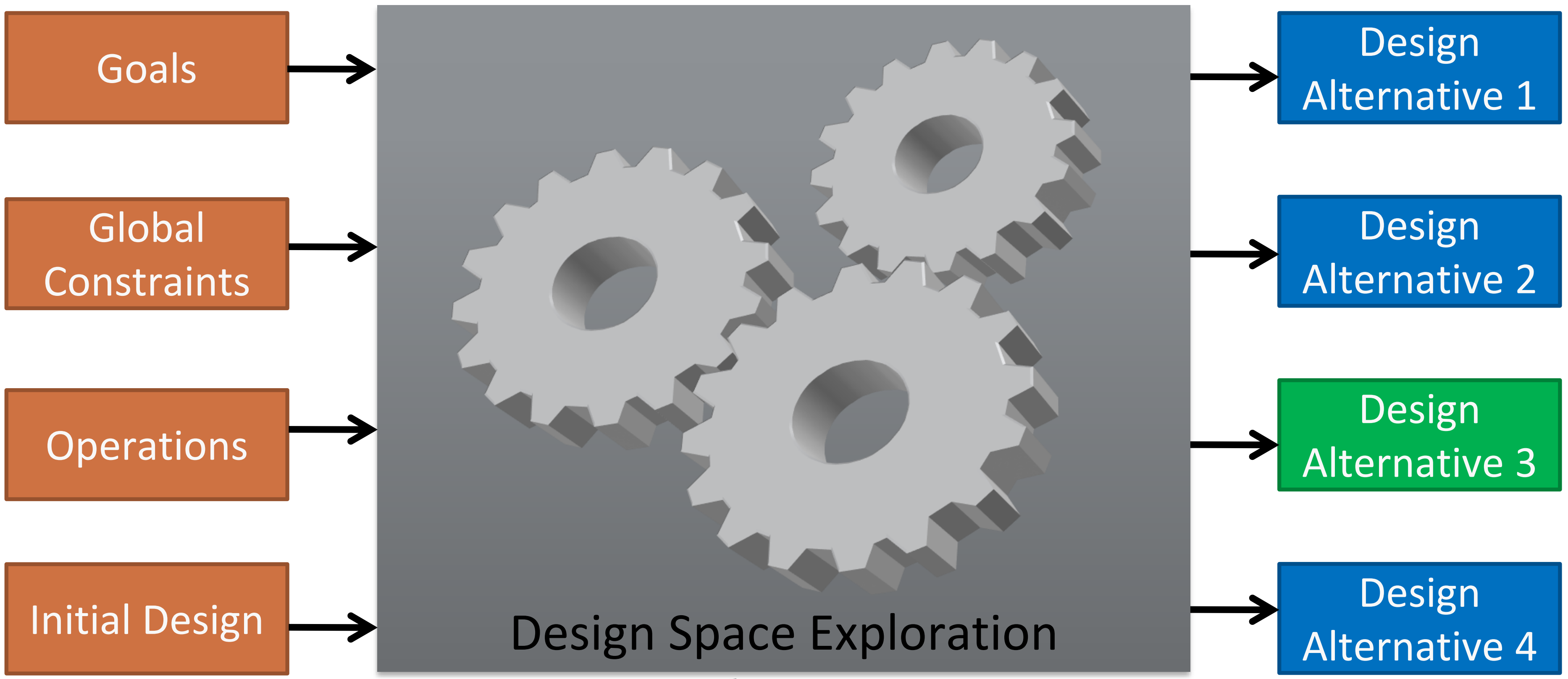
- Definitions
- Graph pattern matching
- Model transformations
- Incremental transzformations
- **Design space exploration**

Revisit: state space of GT system


Solutions are in the state space
Potentially infinite state space
?
G_0

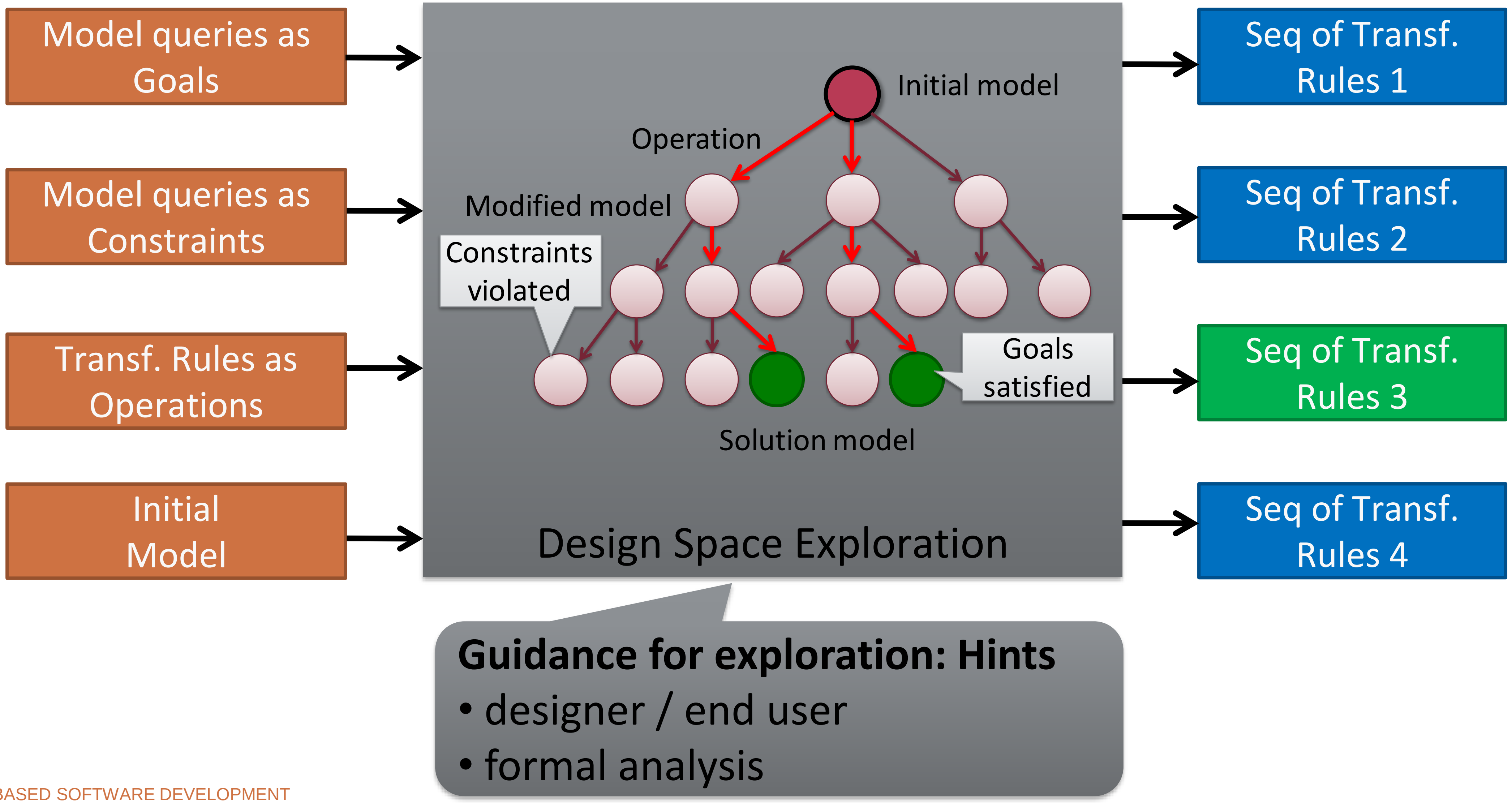
Initial Graph + GT rules → State Space

Design Space Exploration



Special state space exploration
- potentially infinite state space
- „dense” solution space

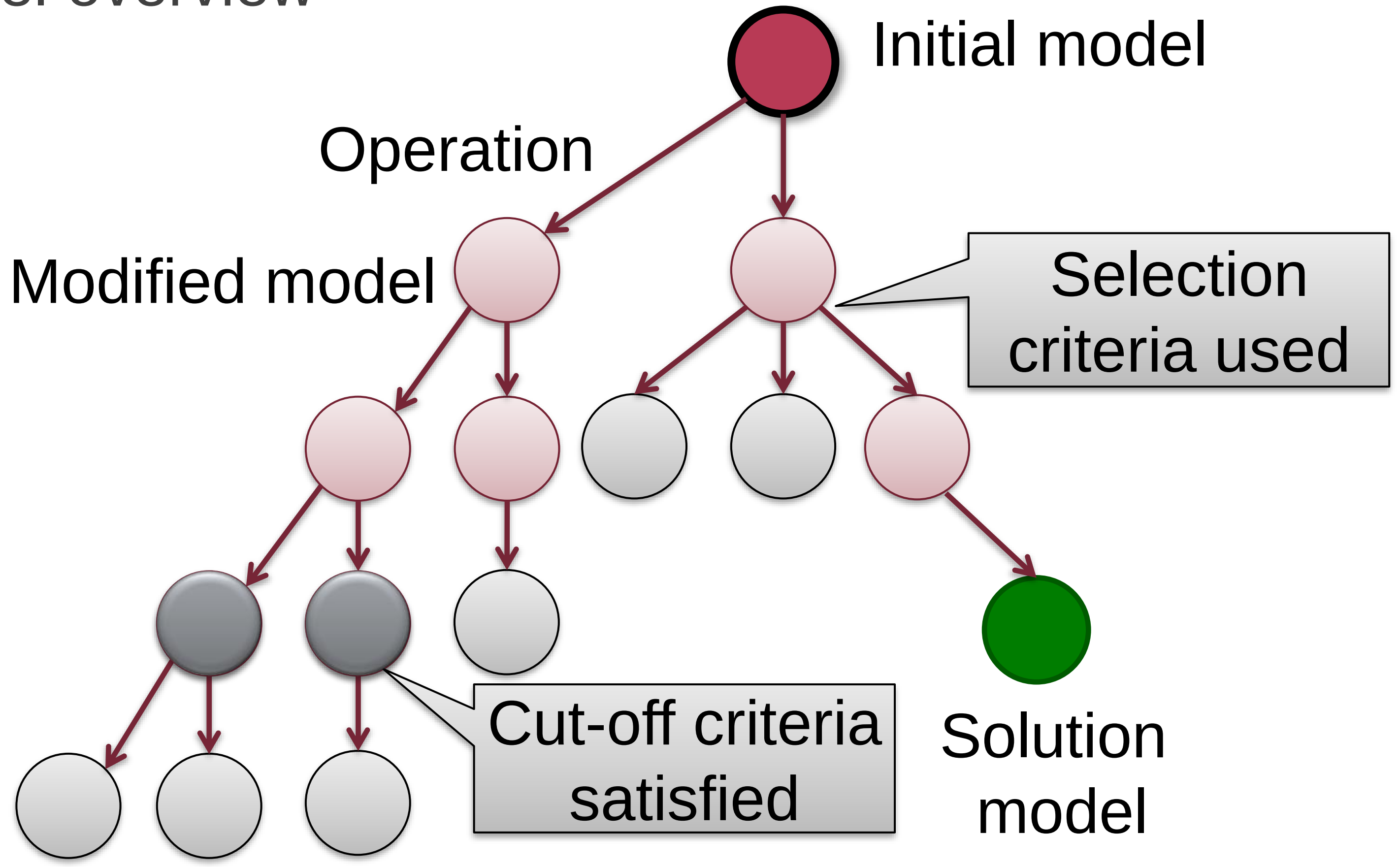
Model Driven Guided Design Space Exploration



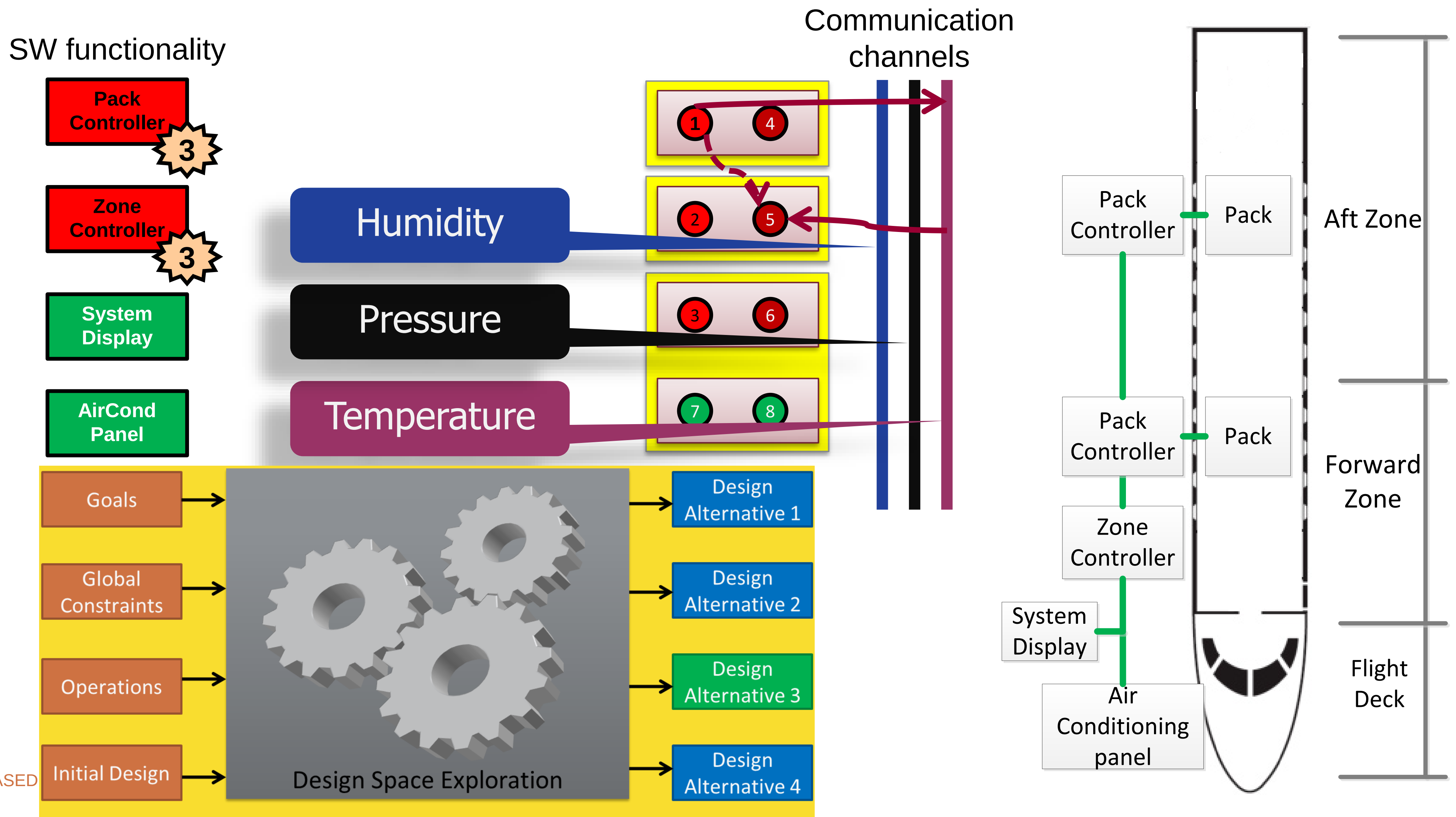
Guidance for exploration: Hints
- designer / end user
- formal analysis

Guided Design Space Exploration

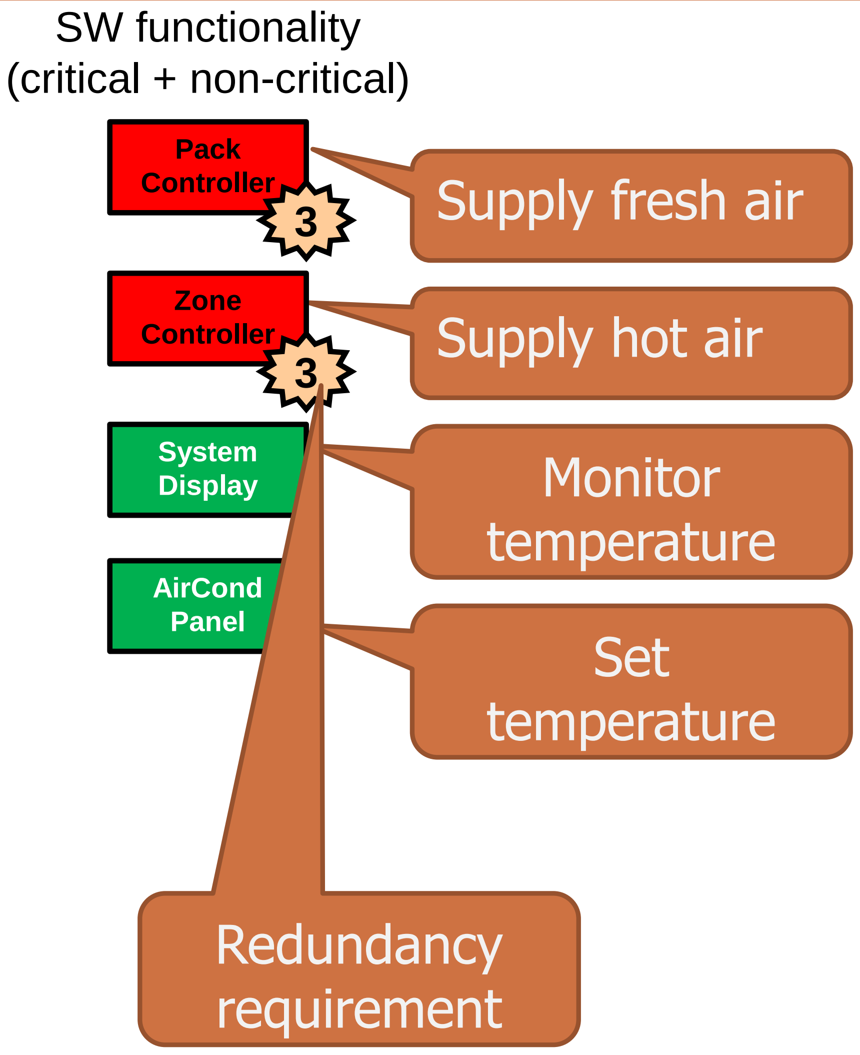
- High-level overview

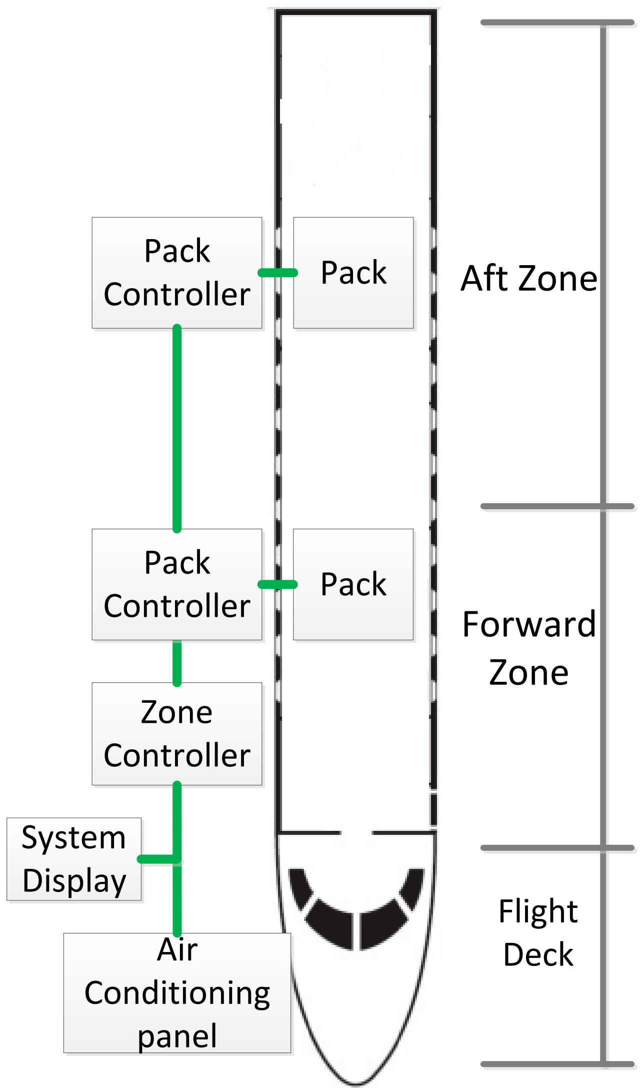


Design Space Exploration for IMA Configuration Design

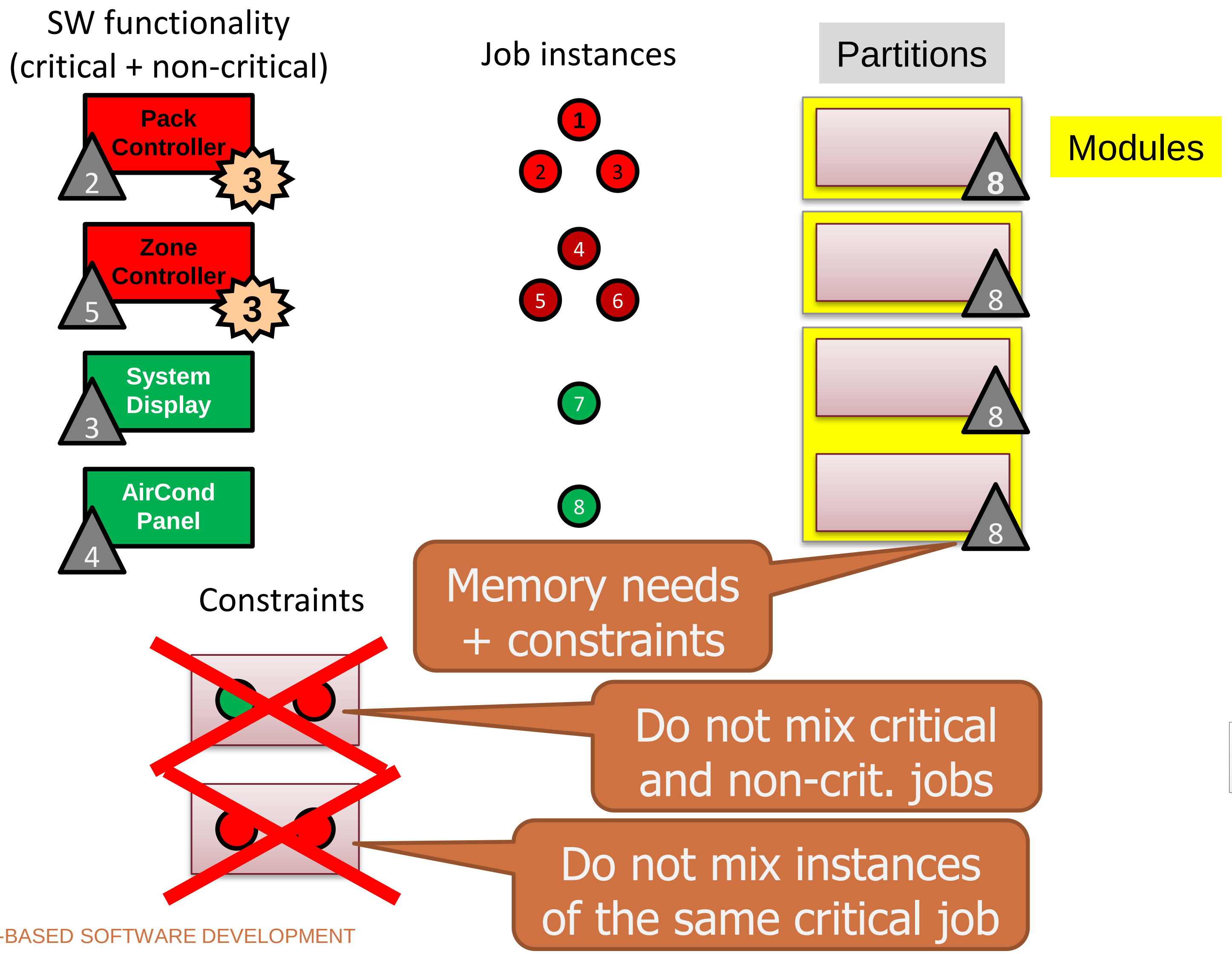


Designing ARINC653 configurations

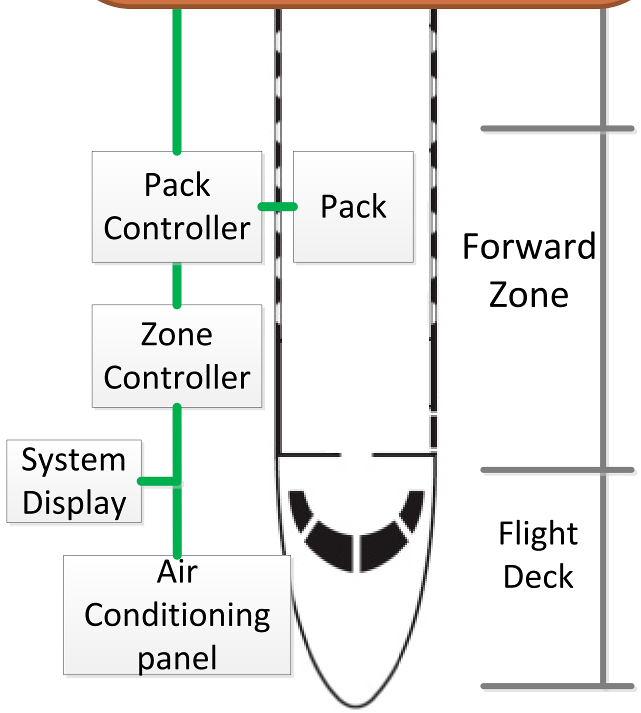




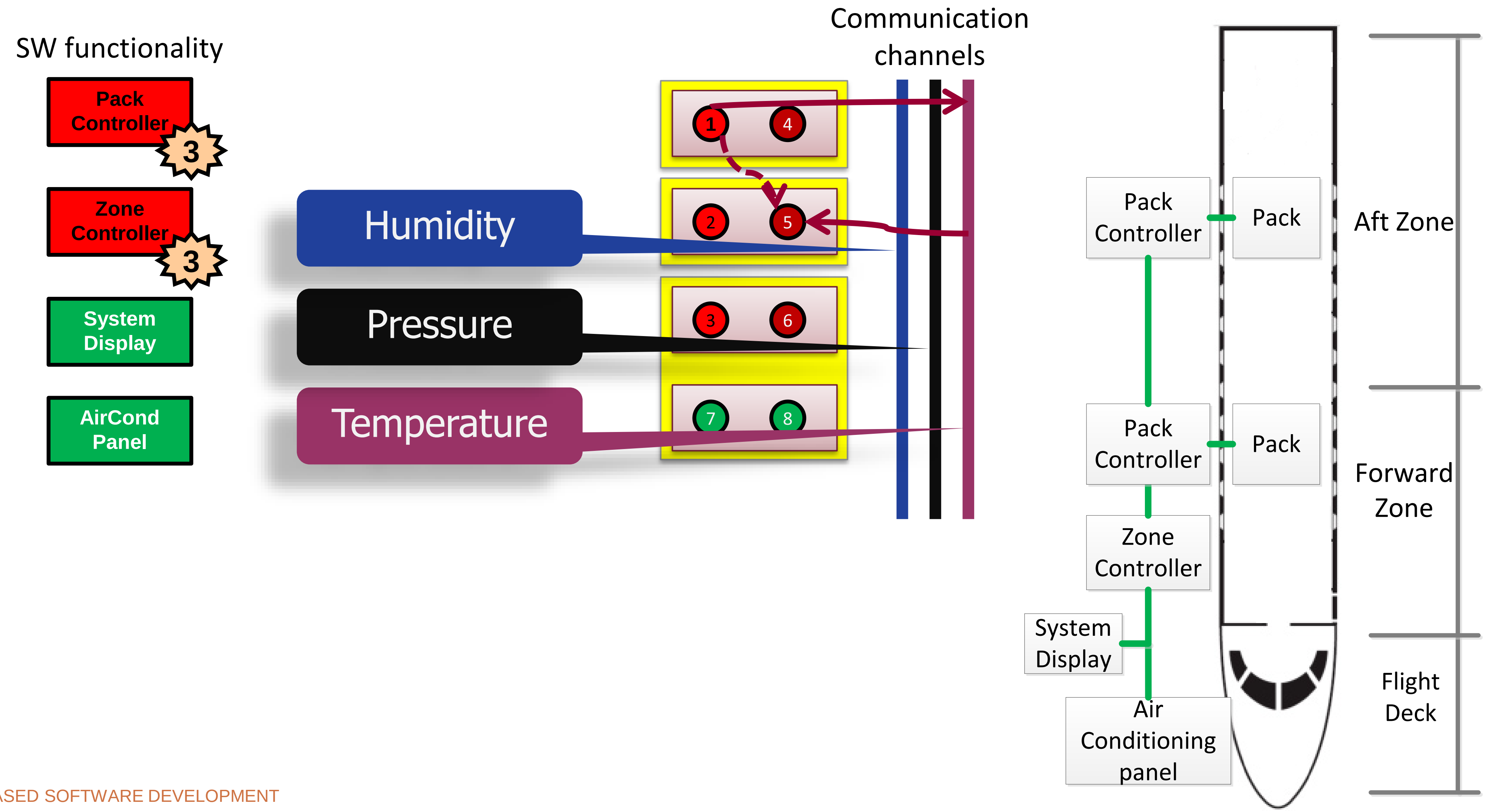
Job instances, Partitions, Modules


SW functionality
(critical + non-critical)
Pack Controller
2
3
Zone Controller
5
3
System Display
3
AirCond Panel
4
Job instances
1
2
3
4
5
6
7
8
Partitions
8
8
8
8
Modules
Constraints
Memory needs + constraints
Do not mix critical and non-crit. jobs
Do not mix instances of the same critical job

Additional constraints
- WCET,
- scheduling, etc.
- interfaces
- datatypes



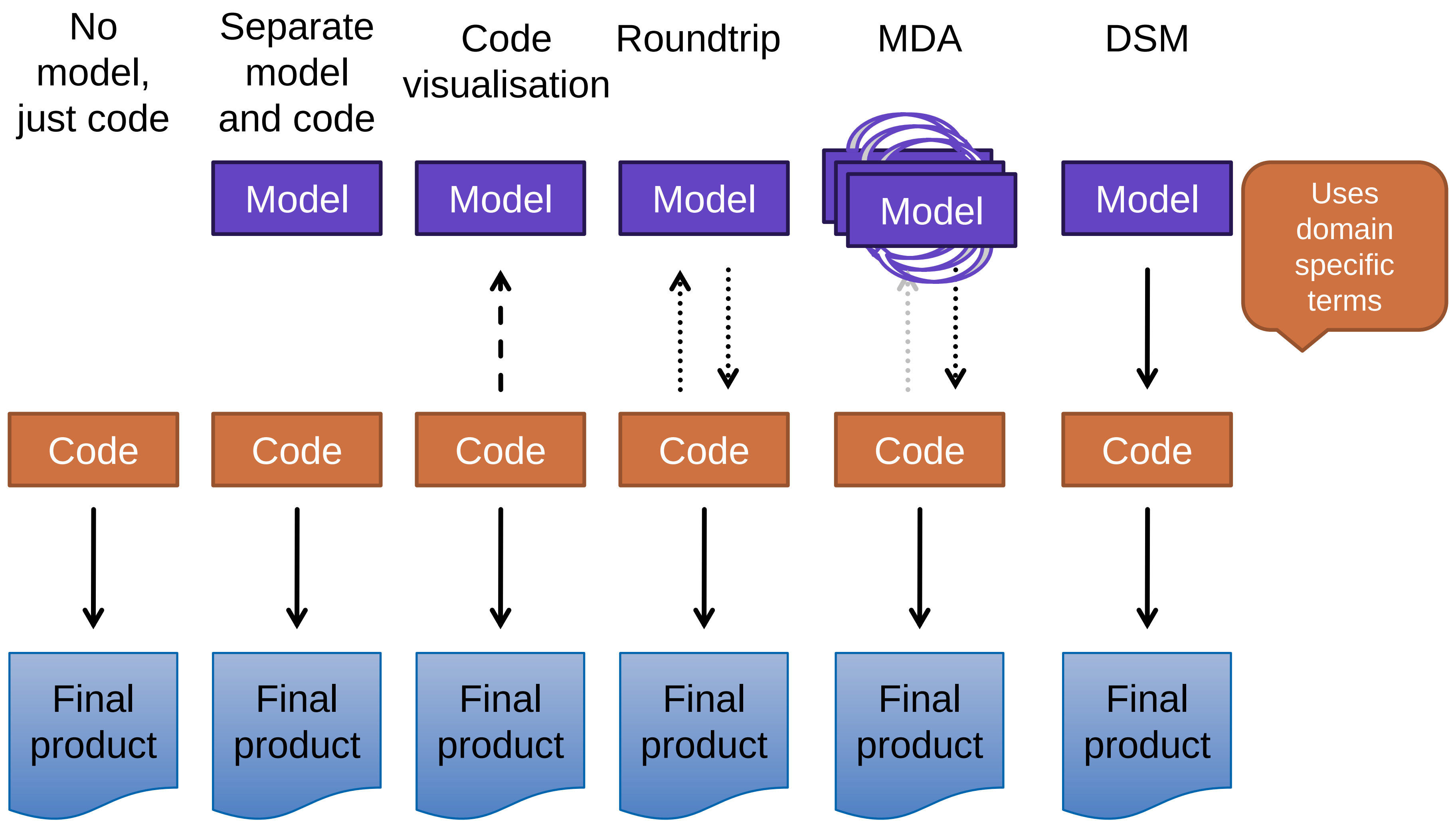
Allocating communication channels



Graph pattern matching, Graph transformation

- Definitions
- Graph pattern matching
- Model transformations
- Incremental transzformations
- Design space exploration

How do we use models?



Thank you for your attention