

# Multi-level Language Descriptions

Andreas Prinz

Introduction

Tools

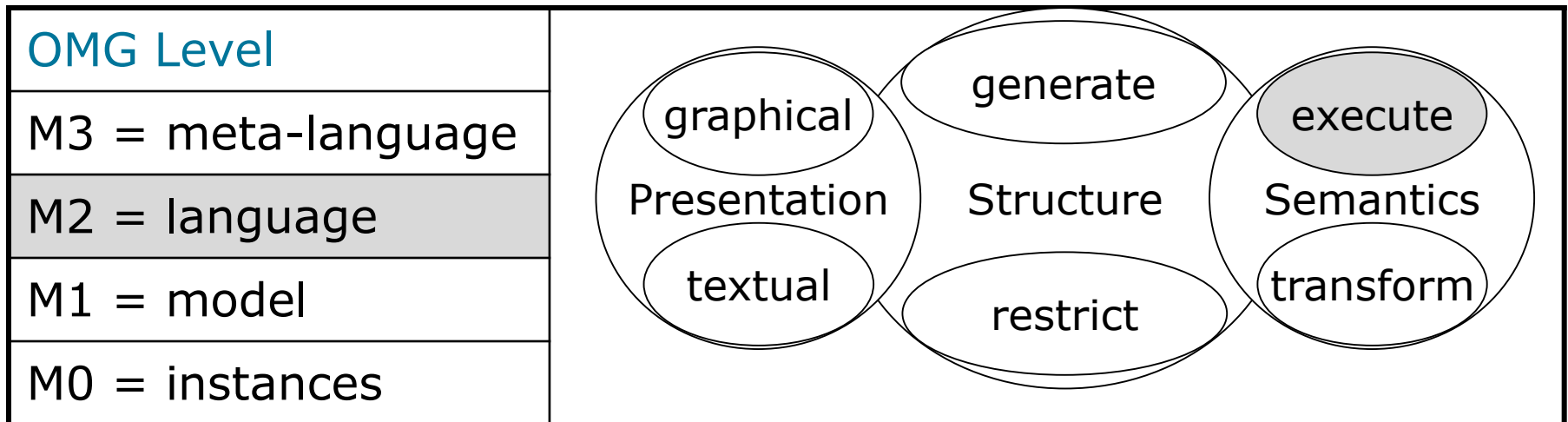
Runtime states

Instantiation

Summary



# Defining language semantics



- In meta-modelling, semantics is often given by transformation.
- We want to describe execution semantics with two parts: (1) runtime states and (2) runtime state changes.
- We assume that the language structure (meta-model) is given.
- OMG levels are absolute with instantiation between the levels.
- Only execution semantics crosses 2 levels.

# Relative architecture

Relative Level	Example 1	Example 2	Example 3
n+2 (language)	MOF	UML	MOF
n+1 (model)	UML	UML model	MOF
n (instances)	UML model	UML objects	UML

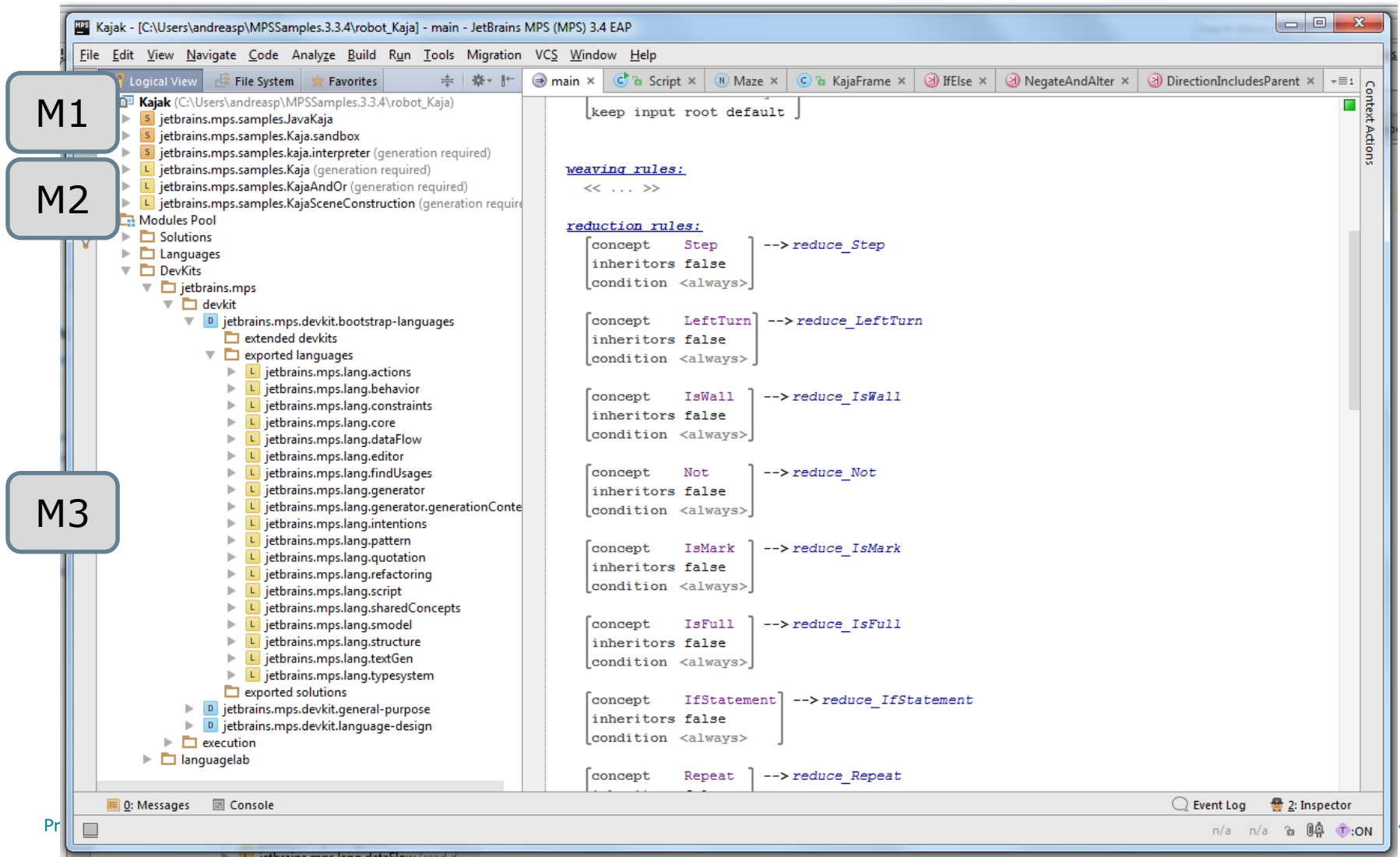
Levels are placed relatively.

Between two adjacent levels there is instantiation.

Instantiation semantics describes how to go from n+1 to n.

Execution semantics also describes going from n+1 to n.

# MPS (conceptually absolute)



The screenshot shows the JetBrains MPS IDE interface. On the left, the 'Logical View' pane displays a project structure for 'Kajak'. Three callout boxes are overlaid on the left side:

- M1** points to the 'jetbrains.mps.samples.Kaja' directory.
- M2** points to the 'jetbrains.mps.samples.KajaAndOr' directory.
- M3** points to the 'jetbrains.mps.lang.actions' directory.

The main editor window shows the following code:

```

[keep input root default ]

weaving rules:
<< ... >>

reduction rules:

[concept Step ] --> reduce_Step
inheritors false
[condition <always>]

[concept LeftTurn ] --> reduce_LeftTurn
inheritors false
[condition <always>]

[concept IsWall ] --> reduce_IsWall
inheritors false
[condition <always>]

[concept Not ] --> reduce_Not
inheritors false
[condition <always>]

[concept IsMark ] --> reduce_IsMark
inheritors false
[condition <always>]

[concept IsFull ] --> reduce_IsFull
inheritors false
[condition <always>]

[concept IfStatement ] --> reduce_IfStatement
inheritors false
[condition <always>]

[concept Repeat ] --> reduce_Repeat
  
```

The bottom status bar shows 'Event Log' and 'Inspector' tabs, with a page number '4' in the bottom right corner.

# LanguageLab (relative)

The screenshot shows the LanguageLab IDE with the following components:

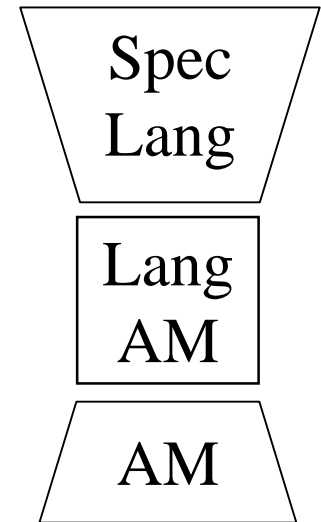
- Top Panel (Mn+1):** Contains modules `BasicStructure`, `BasicTextEditor`, and `BasicMapping`. A callout box labeled "Mn+1" points to this area. A grey box labeled "Interface between Mn and Mn+1" is positioned between the top and middle panels.
- Middle Panel (Mn):** Contains modules `BasicStructureEditor`, `BasicTextEditorEditor`, and `BasicMappingEditor`. A callout box labeled "Mn" points to this area. A grey box labeled "Interface between Mn-1 and Mn" is positioned between the middle and bottom panels.
- Left Panel (Mn):** Shows a tree view of the `RuntimeStateInfoTree` module, listing various objects like `MapOperation o111`, `ObjectMap o112`, etc.
- Bottom Panel (Mn):** Shows the `LowerInterfaceView` module, listing interfaces like `Transition`, `Place`, `Syntax PetriNetEditor`, and `prepareToRun()`.
- Code Editor:** Displays the following code:
 

```

type Transition {
  att tname: llString;
  ref inputPlace: Place;
  ref outputPlace: Place;
}
type Place {
  att pname: llString;
  att tokens: llInt;
}
      
```

# Underlying abstract machine

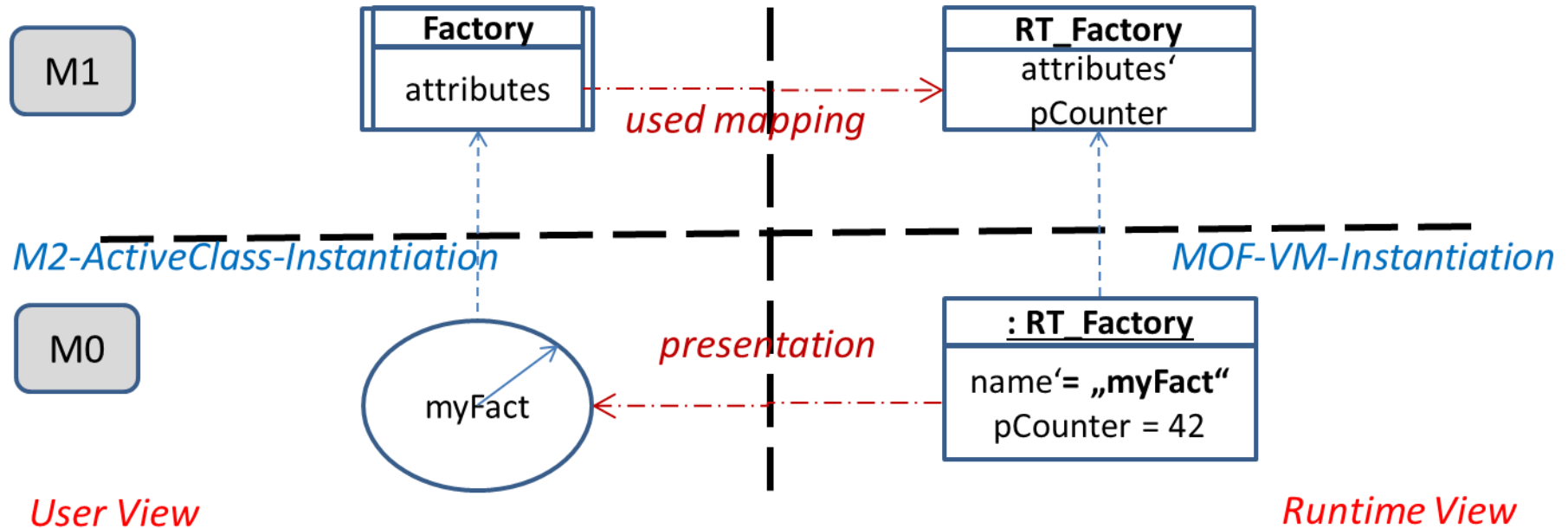
- There is always an underlying (abstract) machine AM.
- What does the AM provide?
  - Functions, parameters
  - Instantiation
- There are runtime states for the underlying machine as well.
- We use a special object-oriented underlying machine: MOF-VM.



# What are runtime states (RTE)?

- *Read-only program* included in the RTE
- *Global elements*: independent of the specific program
- *Local elements*: related to language concepts but independent of the instance
  - *None-elements* are not existing at runtime (1:0).
  - *One-elements* are existing at runtime (1:1)
  - *Many-elements* are existing at runtime (1:n)
- *Dependent elements*: related to language concepts and dependent of the instance

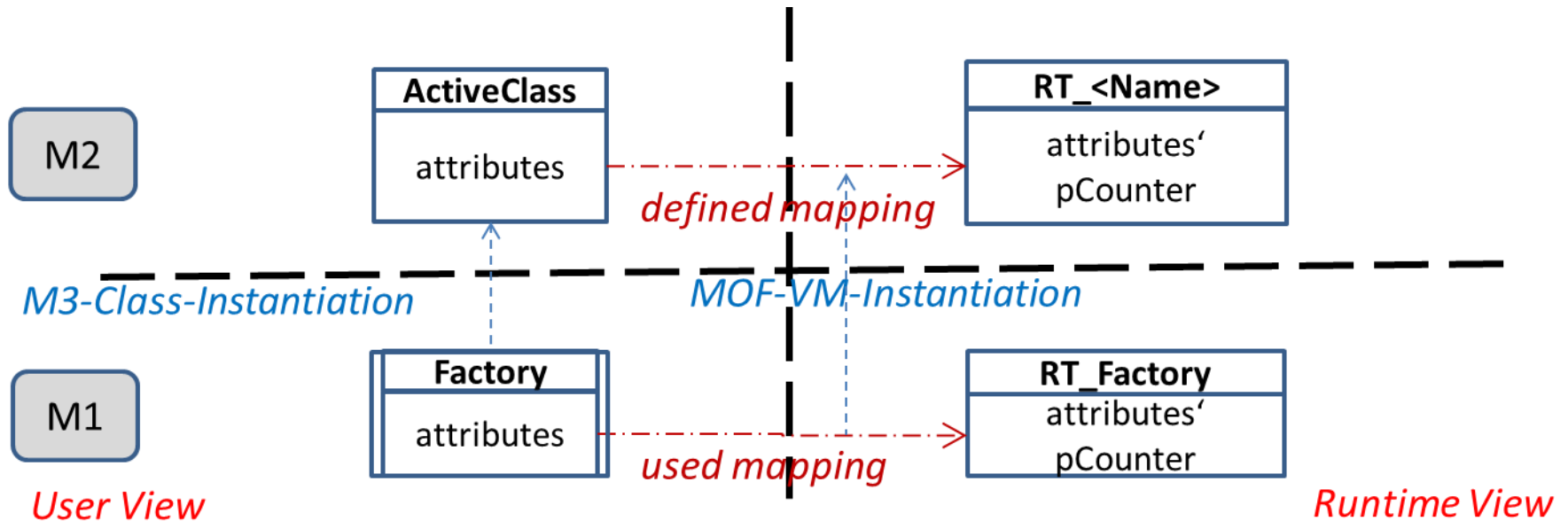
# Language instantiation



- Language instantiation (linguistic) is based on the underlying machine instantiation (runtime = MOF-VM).

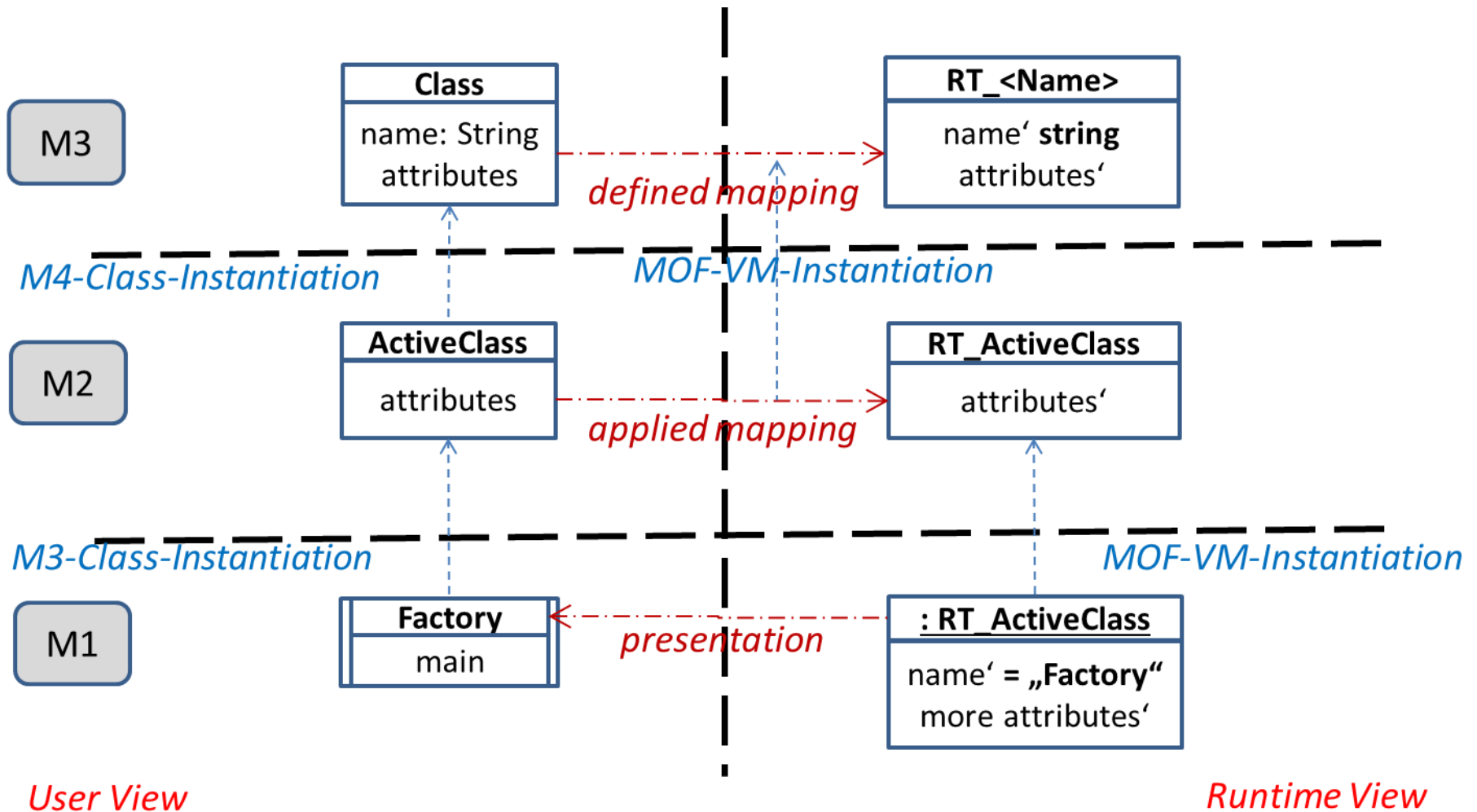


# Defining instantiation

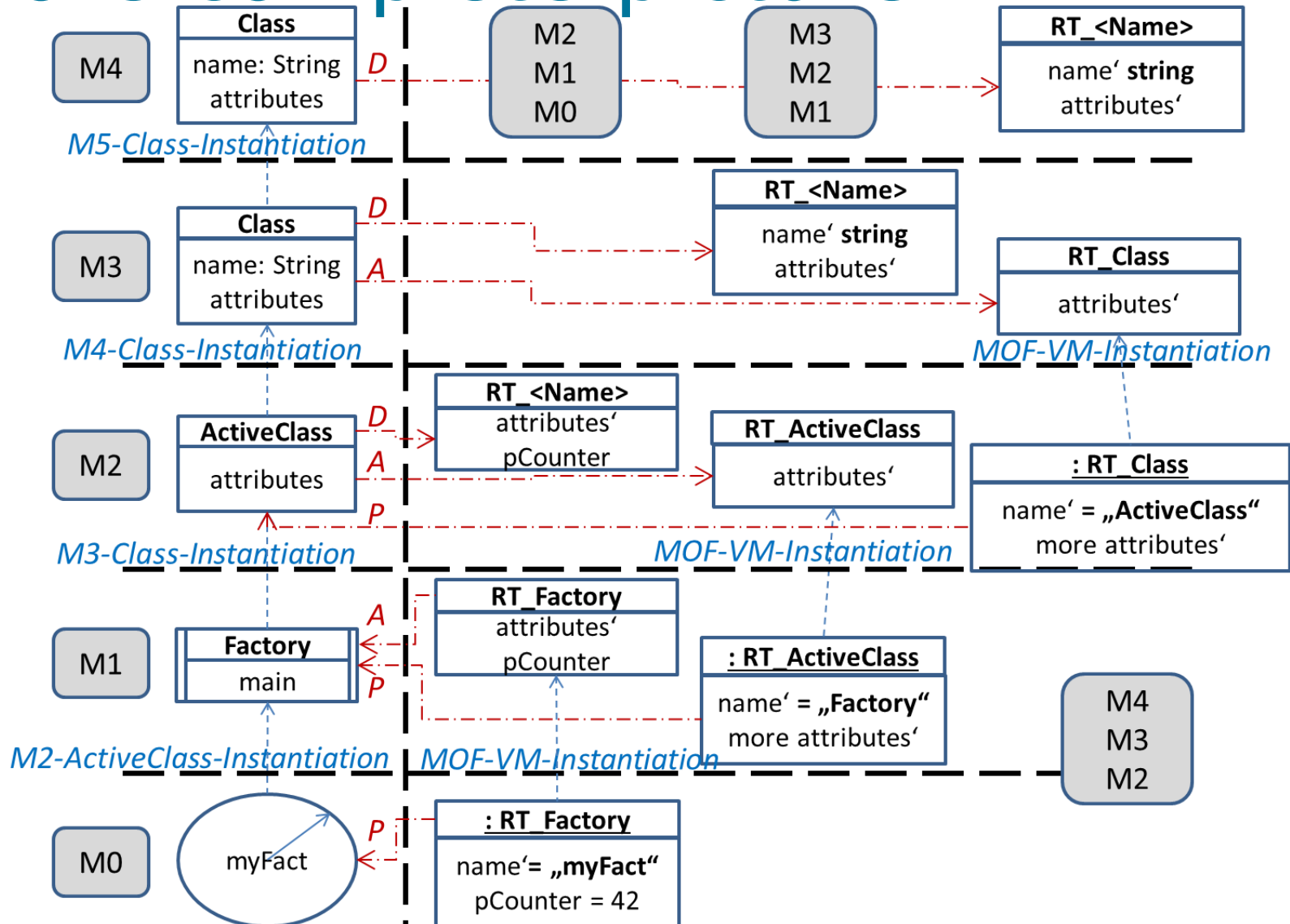


- Instantiation is defined by mapping the language to the underlying machine.
- The mapping is applied on the level below.

# Instantiation for metalanguages



# More complete picture



# Summary

- Language execution semantics needs instantiation semantics and dynamic semantics.
- Instantiation semantics (RTE) is based on an underlying machine (MOF-VM) instantiation.
- Several possible kinds of instantiation relations between specification and RTE were identified.
- They are specified for a language as a mapping between specification and MOF-VM, which is defined at language level and used at specification level.
- There are three kinds of instantiation: linguistic, ontological, and runtime instantiation.

# Instances on several levels

