Model-Driven Development (MDD) meets Component Based Software Development (CBSD)

Prof. Dr. Uwe Aßmann
Institut für Software- und Multimediatechnik (SMT)
Fakultät für Informatik
TU Dresden
1.0, Sep 21, 2006

Introduction

- MDD and CBSE, where are we going from here?
  - We will mainly talk about Model-Driven Architecture (MDA ® OMG)
- How to have reuse of models in MDA?
  - How to realize the MDD components?
  - How to unify MDD and CBSE?
- However,
  - Different abstraction levels
    - MDA is about *design reuse*
    - Components about *code reuse*
  - Different instantiation mechanisms
    - MDA: translation
    - Components: connection

Old Tales About Change-Oriented Design

Parnas information hiding principle
for change-oriented design

Modules (Information-Hiding-Based Design a la Parnas)

- Every module hides the an important design decision behind a well-defined interface which does not change when the decision changes.

We can attempt to define our modules “around” *assumptions which are likely to change*.

One then designs a module which “hides” or contains each one.

Such modules have rather abstract interfaces which are relatively unlikely to change.

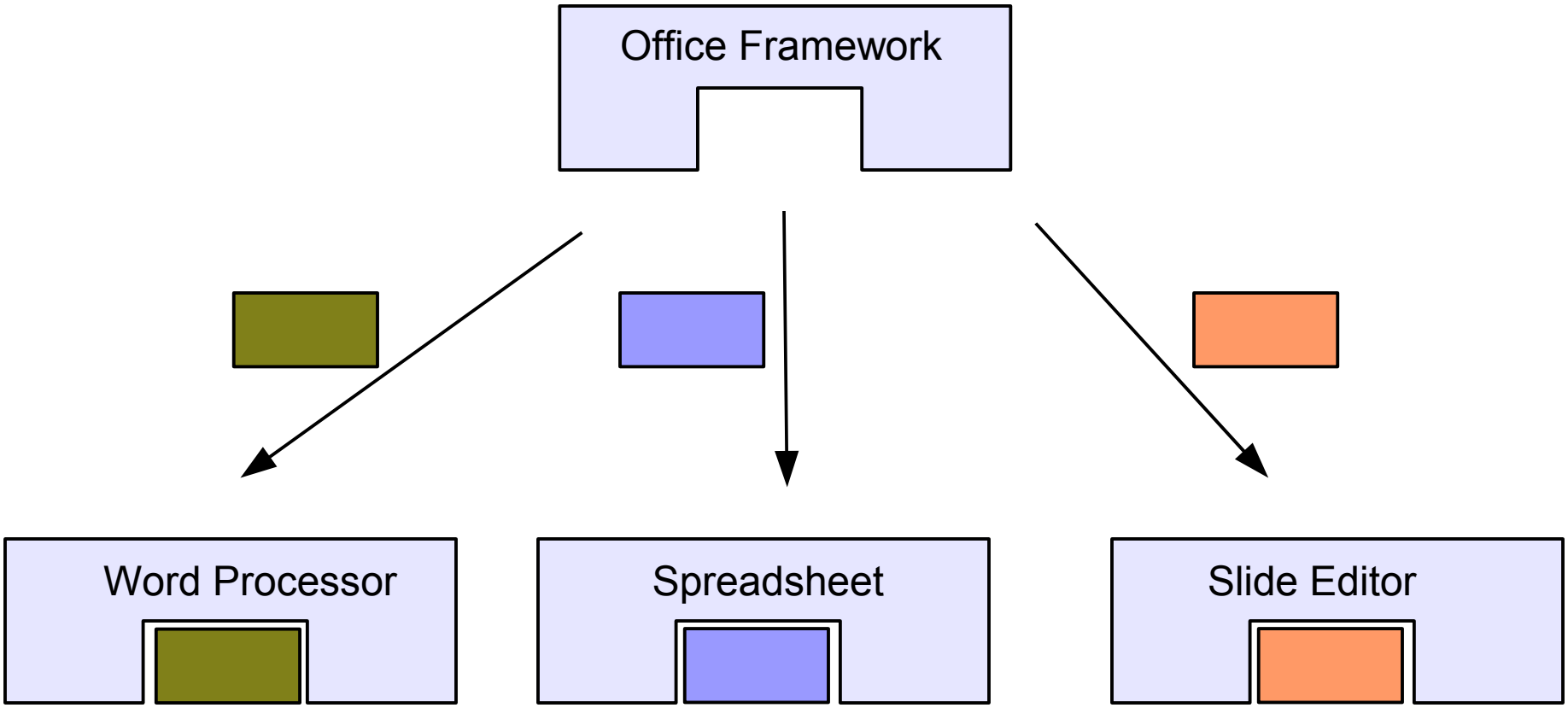
Commonality-Variability Analysis (CVA)

- The Parnas principle has been refined in **Commonality-Variability Analysis**
- Finding common assets for a product line
  - Separating them from variable assets
- CVA can be realized with many technologies

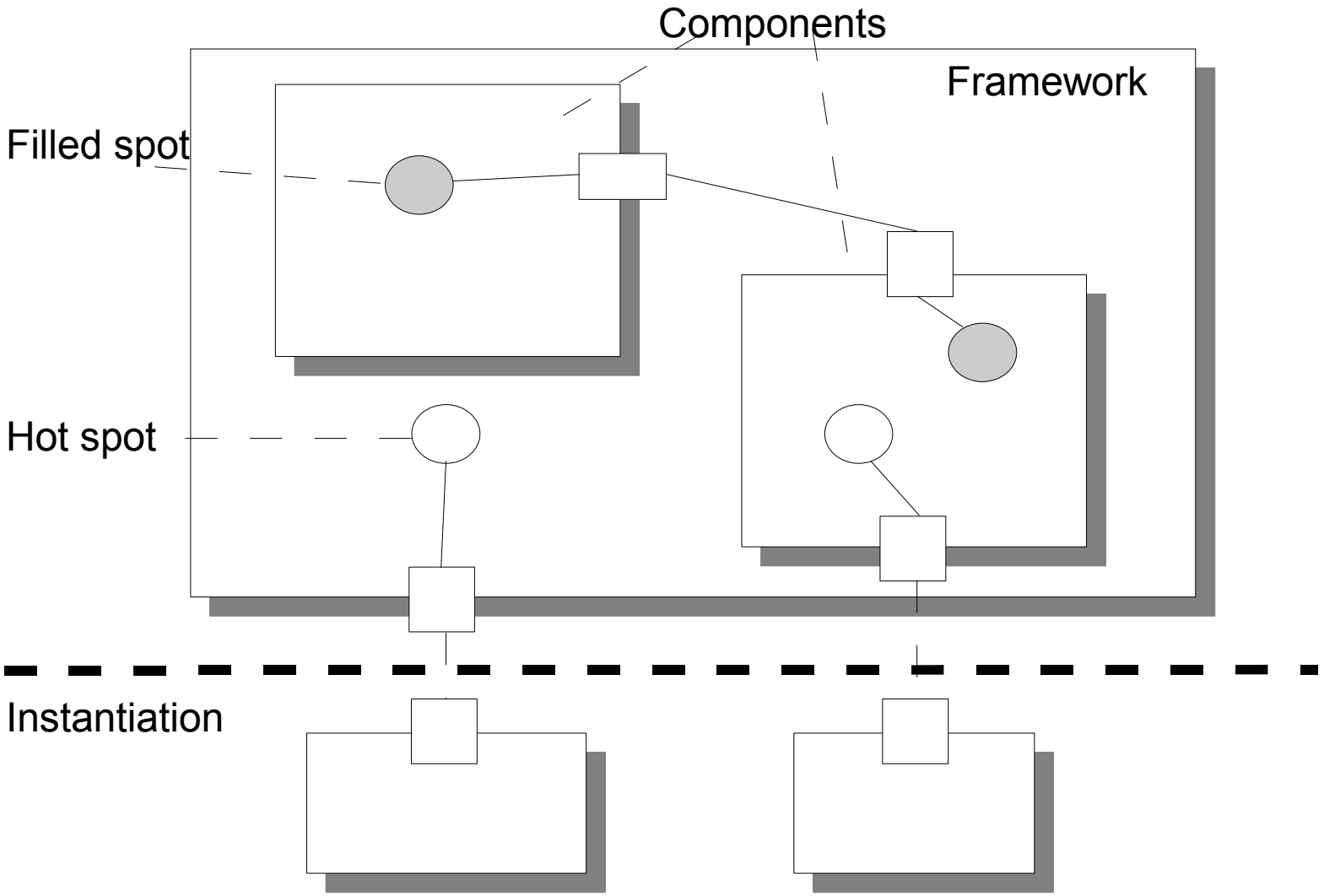
CBSE is not about COM++, but about CVA

Frameworks, Components, and Products

- Planning several products from one common code base, but several variant parameterizations

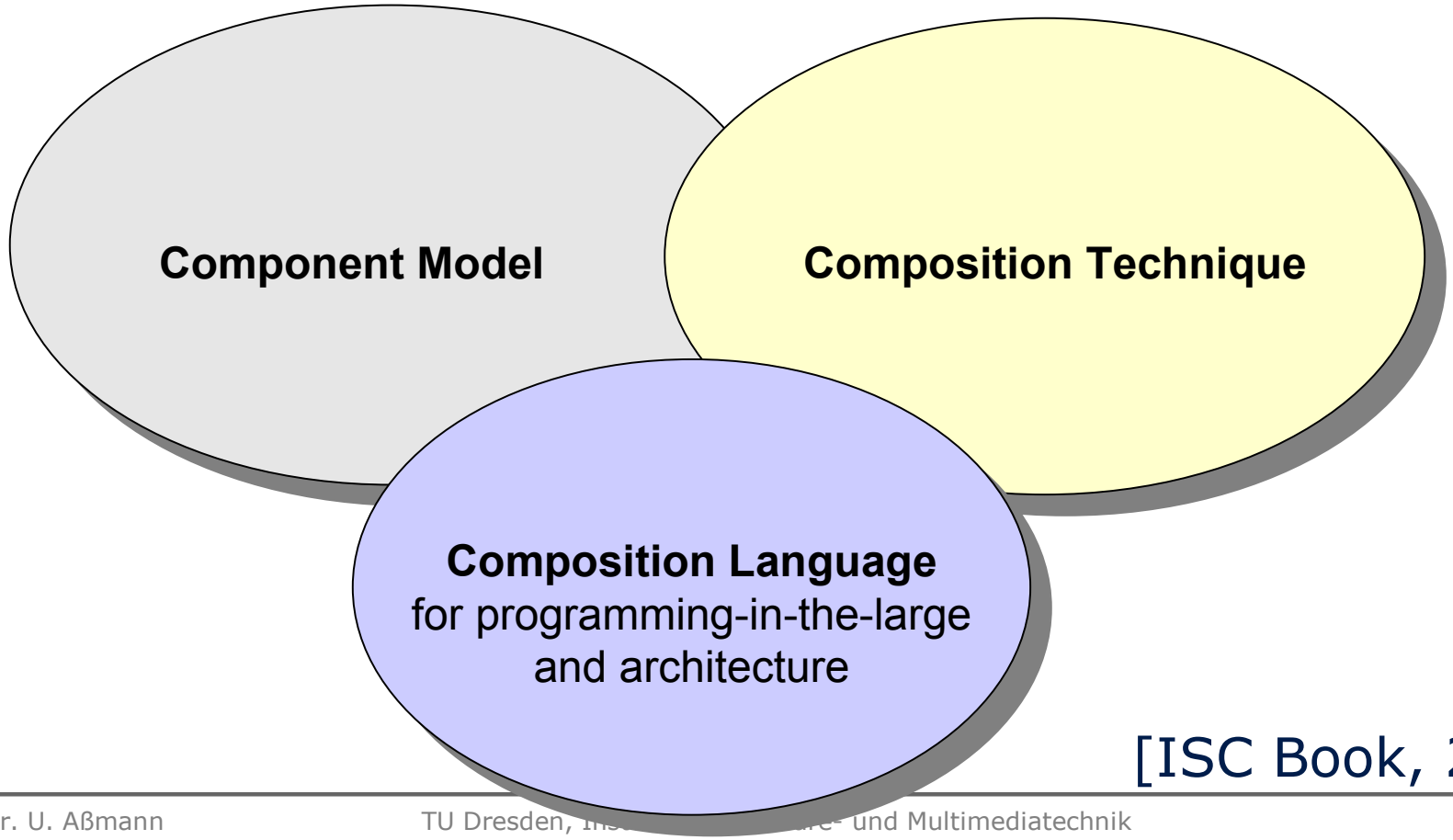


Frameworks are Larger Components



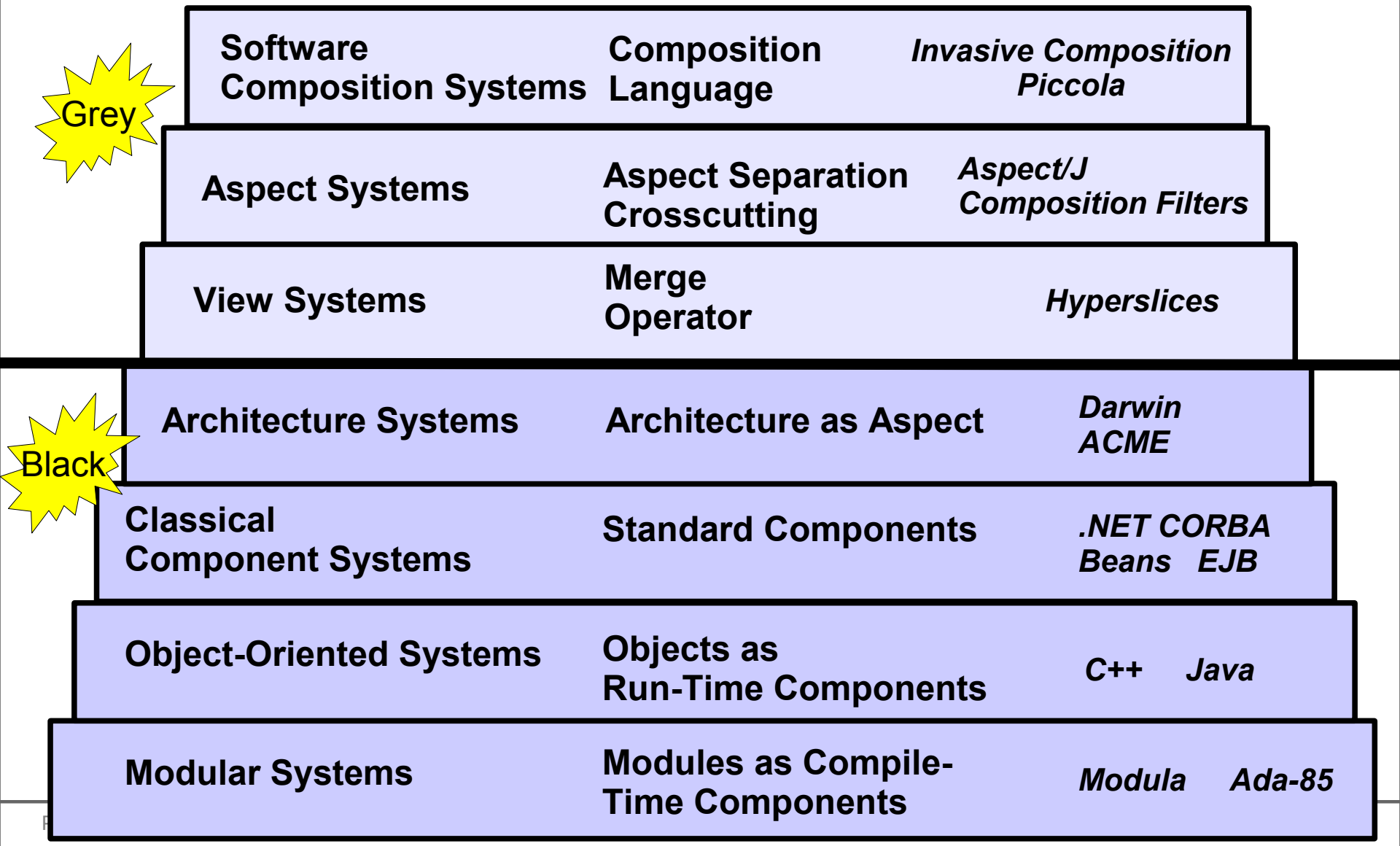
Software Composition Systems

- A composition system has

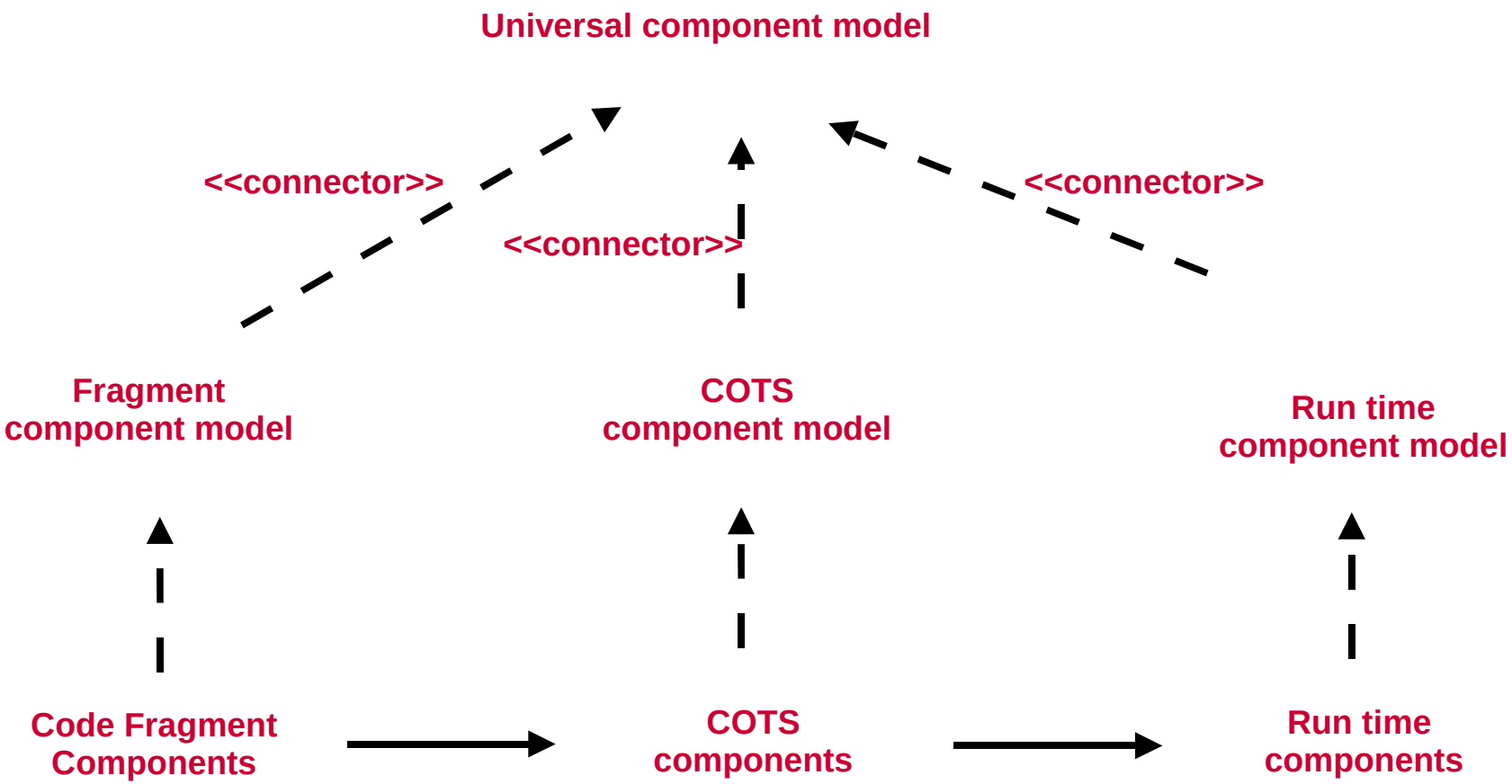


[ISC Book, 2003]

The Ladder of Composition Systems



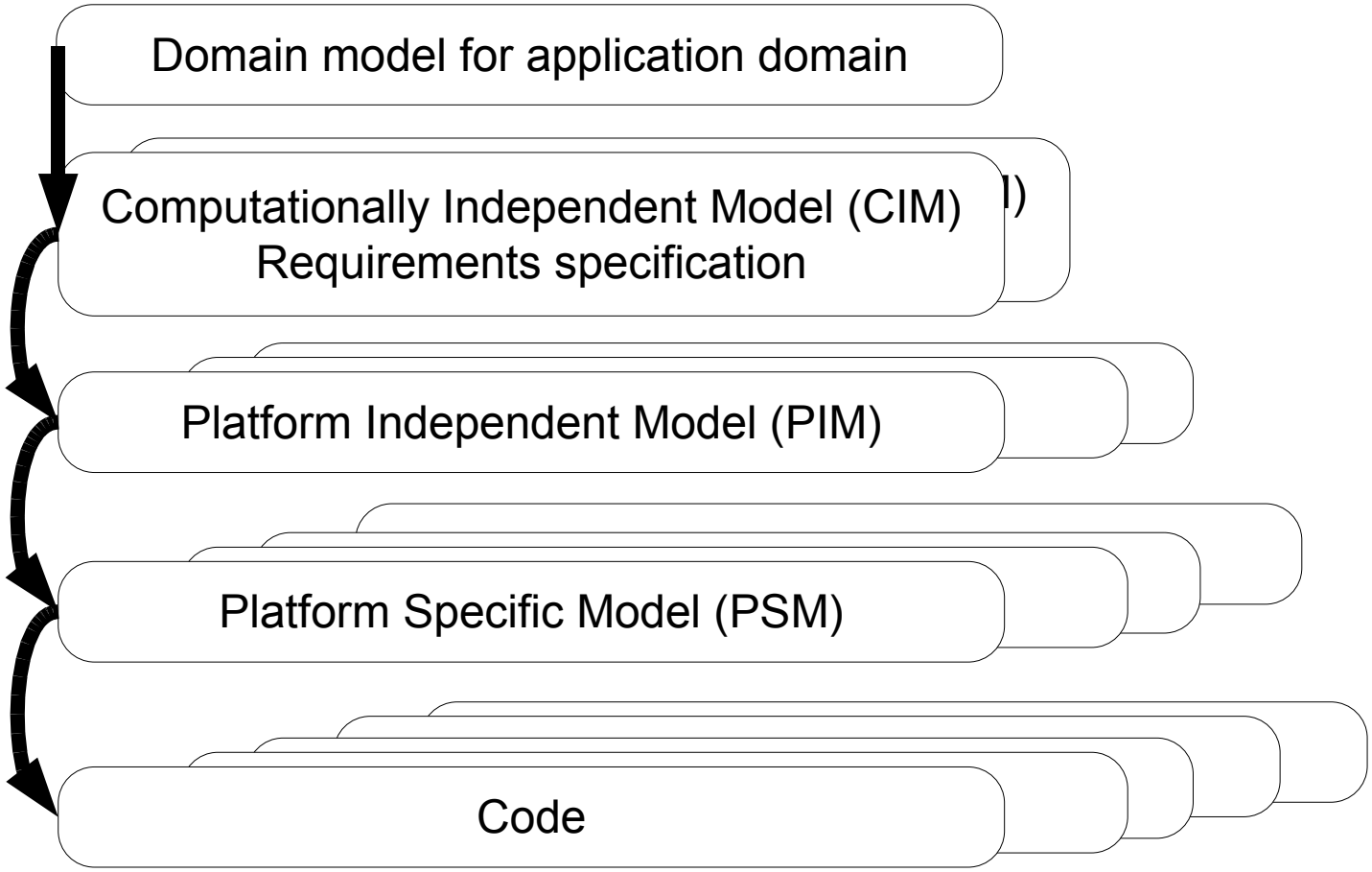
Component Models at Different Composition Times



MDA is not about Platforms, but about CVA

MDA Describes Product Lines

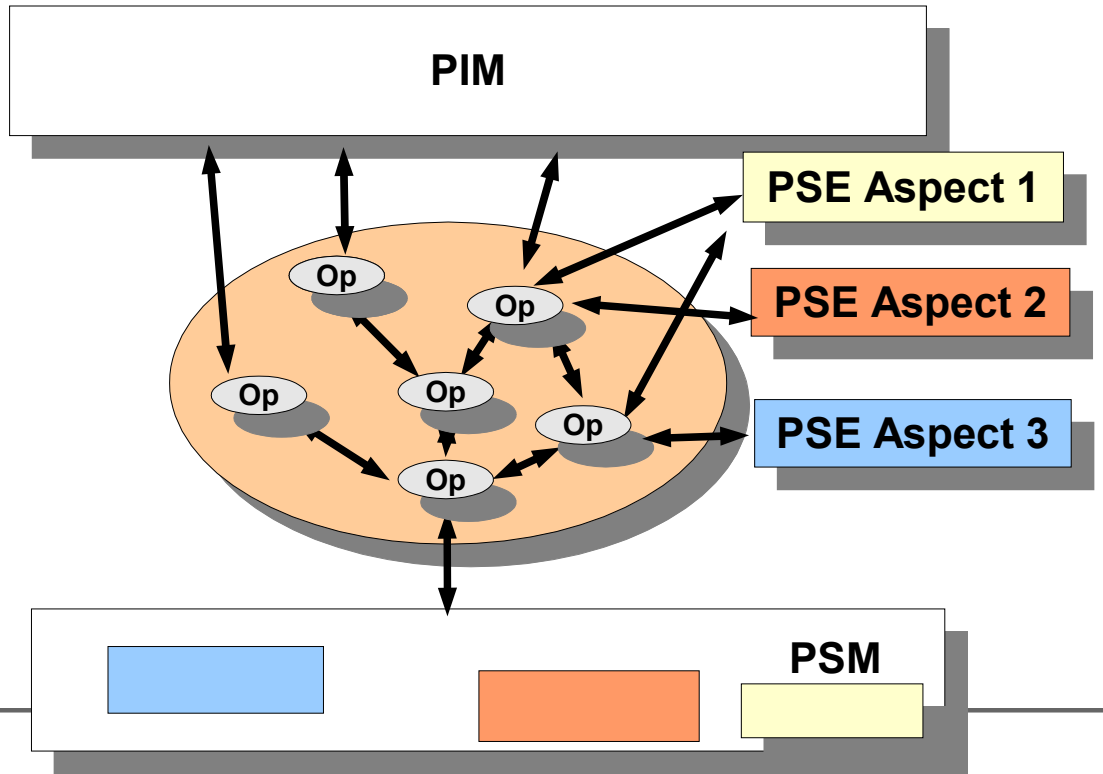
- The platform stack is a *translational framework*



The products of the product line

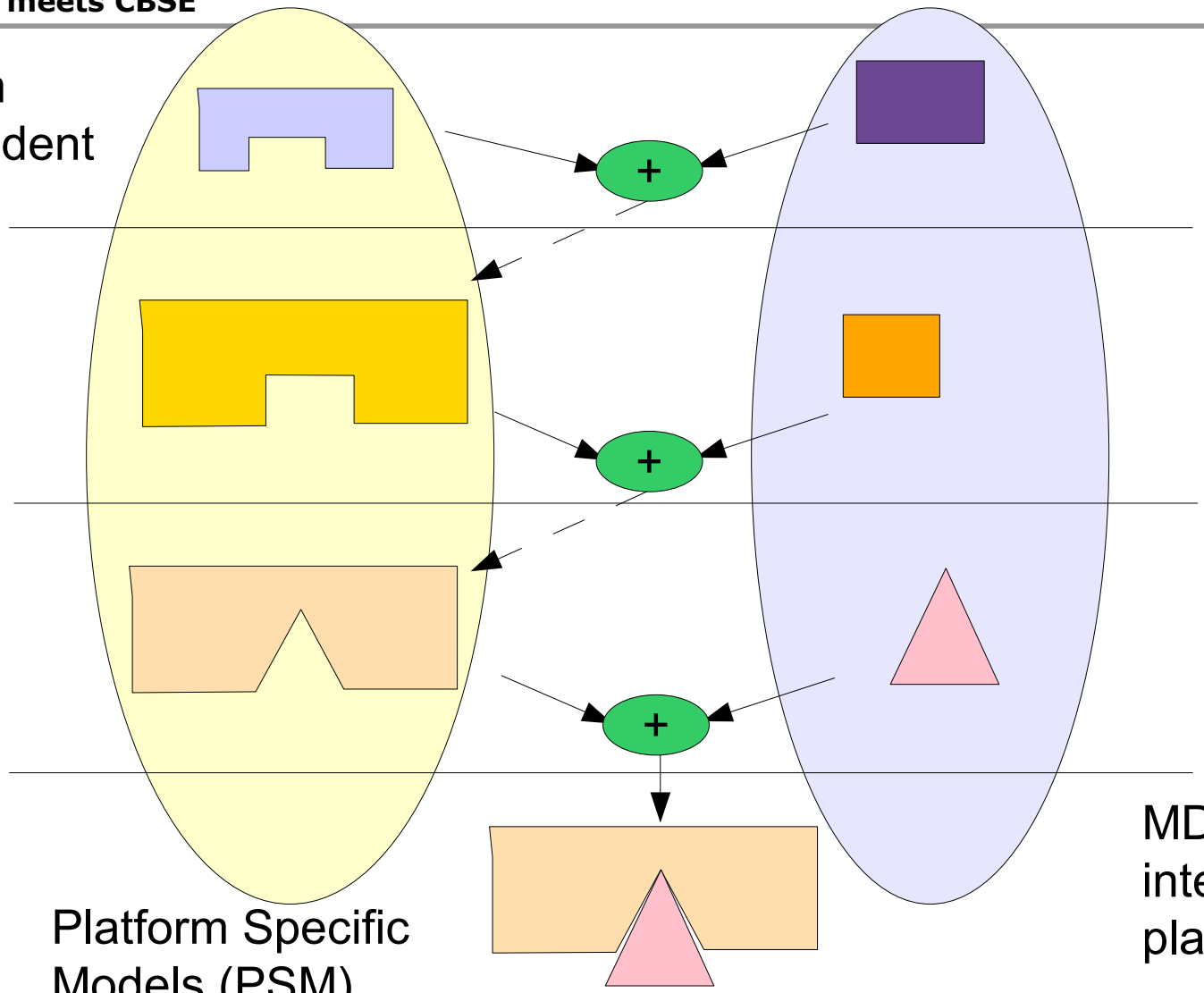
MDA with Aspect Mappings

- Describe *platform specific extension (PSE)* as *aspects:*
  - The PIM is the *core, t*he PSM the *weaved system*
  - The model mapping becomes an *aspect weaver*



A Weaving Architecture: MDA

MDA meets CBSE



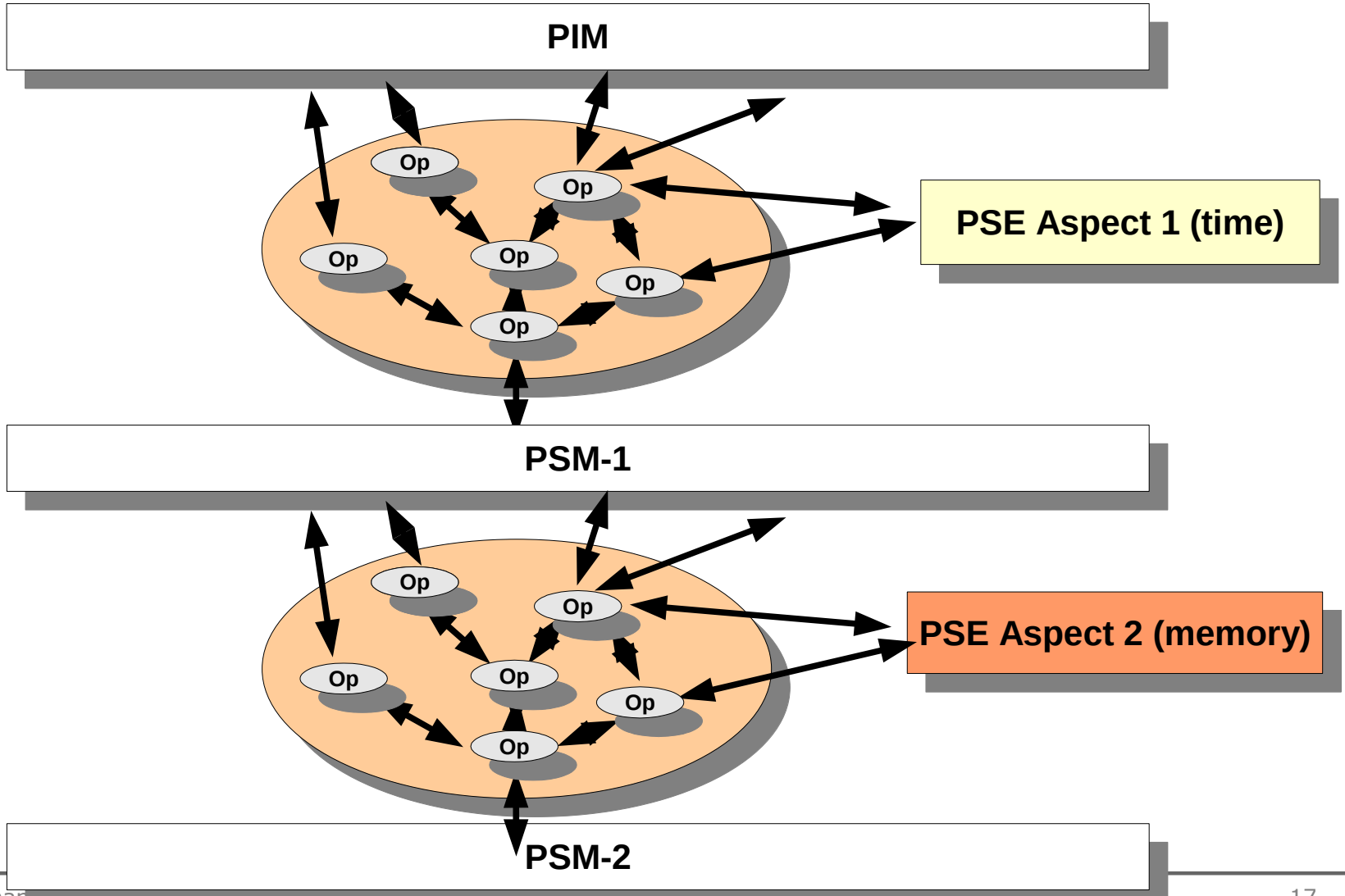
Aspects Important for Embedded Systems

- Life-time aspects in MDA
  - How long does an object live?
- Resource aspects in MDA
  - Representation of collections (SetL)
- Middleware aspects
  - Representation of connectors
  - Transactions, persistency, ...
- Real-time aspects in MDA
  - Profile info
  - OCL-RT info
- Quality aspects in MDA
  - Contracts
  - Security

MDA With Several Layers for Resource-Constrained Systems

TECHNISCHE UNIVERSITÄT DRESDEN

MDA meets CBSE



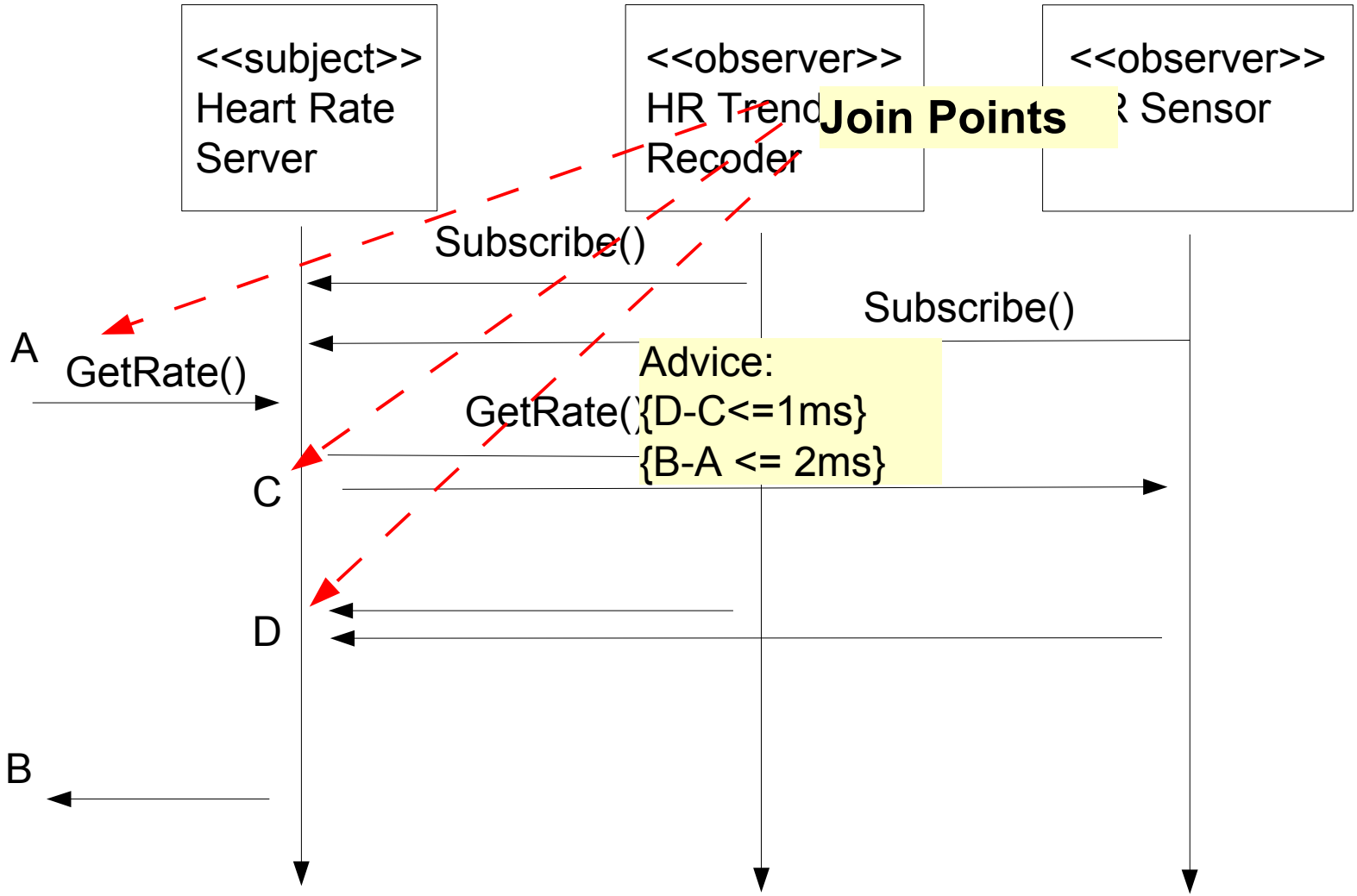
Example: MDA for RT-UML

- EU project *High Integrity Distributed Object-Oriented Real-Time Systems (HIDOORS)* http://www.hidoors.org
- German BMBF project SuReal http://www.sureal-projekt.org

- MDA for RT-UML
  - Realtime sequence diagrams (MSC)
  - UML realtime statecharts
- Mapping to timed automata of Uppaal model checker

RT Sequence Diagram (UML)

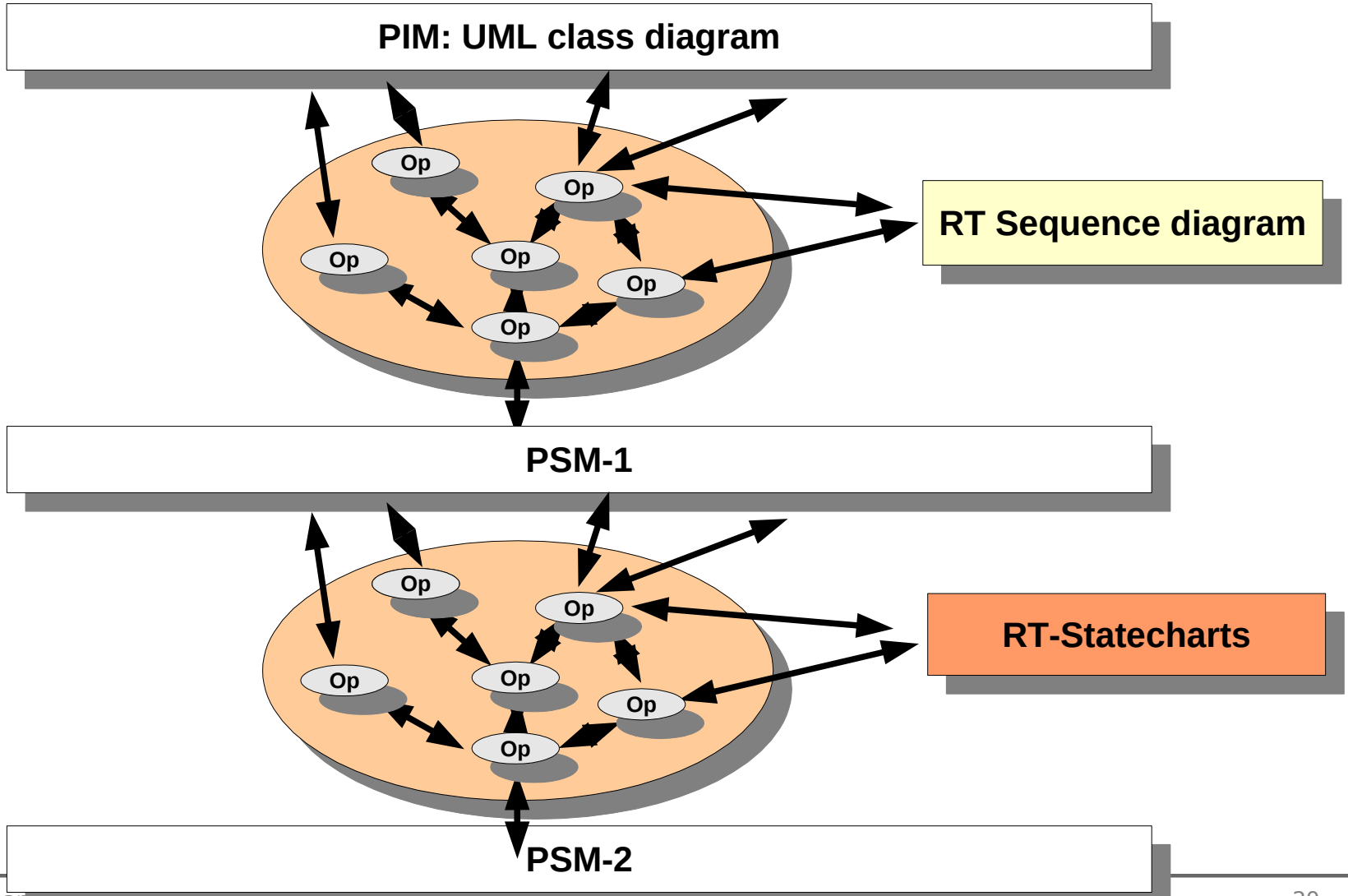
RT Extension Aspect



RT-SD und RT-Statecharts are Platform Specific Aspects

TECHNISCHE UNIVERSITÄT DRESDEN

MDA meets CBSE



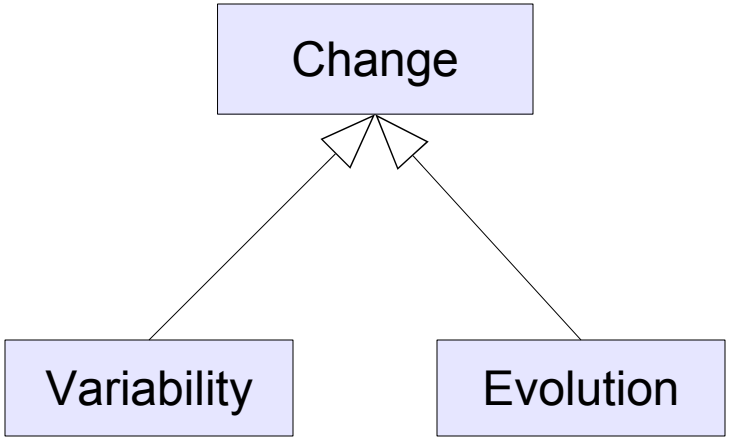
What MDA Really Is About

- MDA is about *weaving platform-extension model aspects*

- Problem:
  - We need weavers for every level
  - With different modeling languages
  - Who will build them?

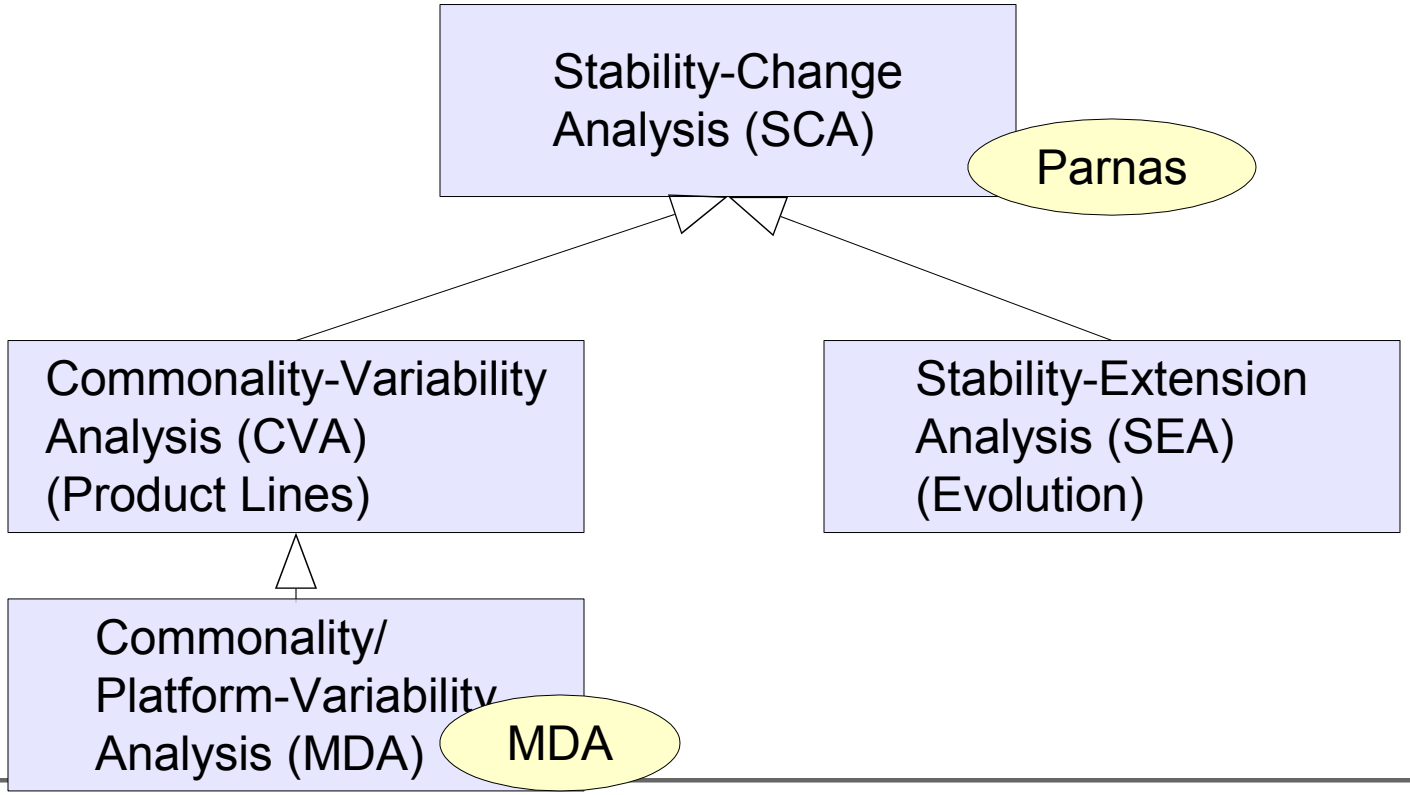
Stability/Change Analysis (SCA)

What is Change?



From CVA to SCA

- [Parnas 72] tells us that commonality/variability is just a special case
- Variability is *preplanned,* evolution is *unforeseen*



Planning Variations - Unforeseen Extensions

- Planning relies on *variation points*
  - Templates with slots
  - Frameworks with Pree-like metapatterns

- Extension relies on *extension points*
  - Cores with *hooks* or *joinpoints*
  - Cores with extension *views*

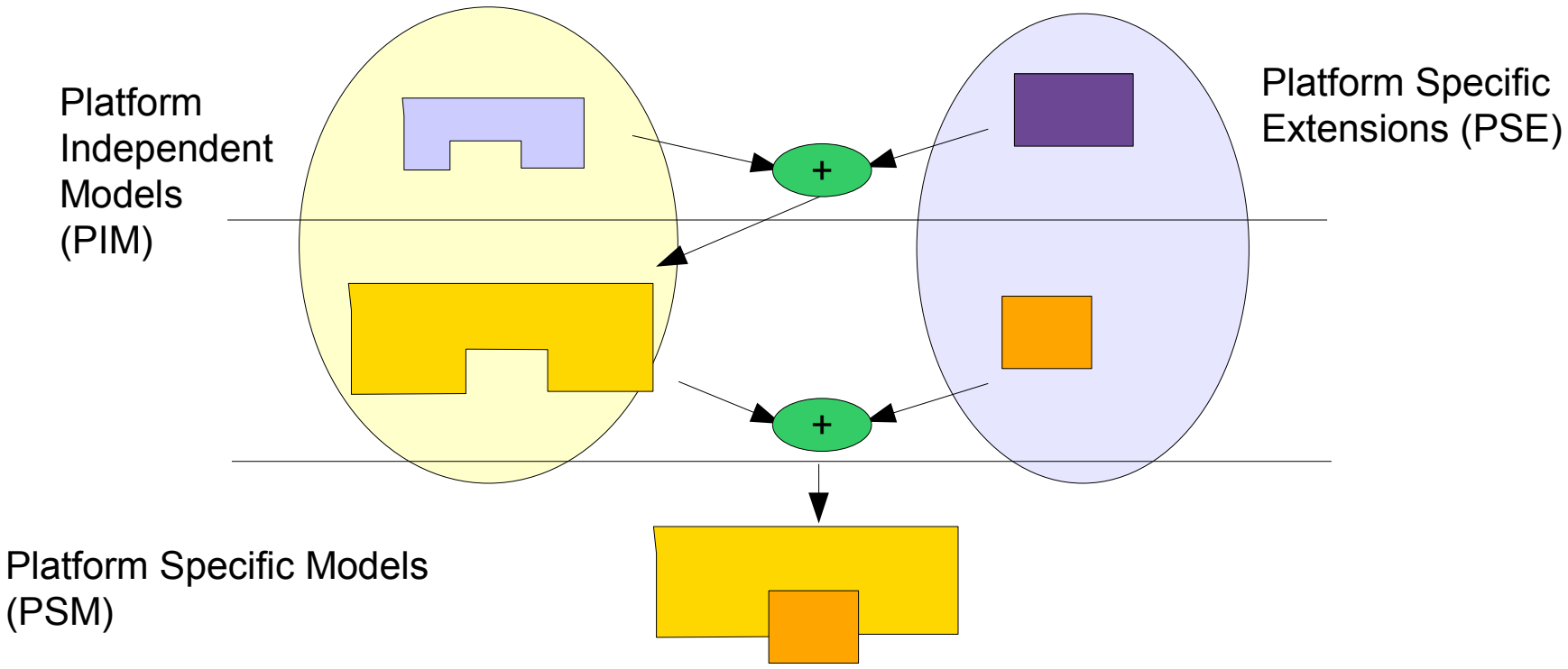
Planned and Unplanned MDA for SCA (CVA and SCE)

Problem

- Parnas requires *interfaces* between modules
  - which are not in MDA which relies on *implicit extension points (implicit hooks on models)*
- Parametric MDA?
  - For planned variants
  - Needs explicit composition interfaces between PIM and PSM

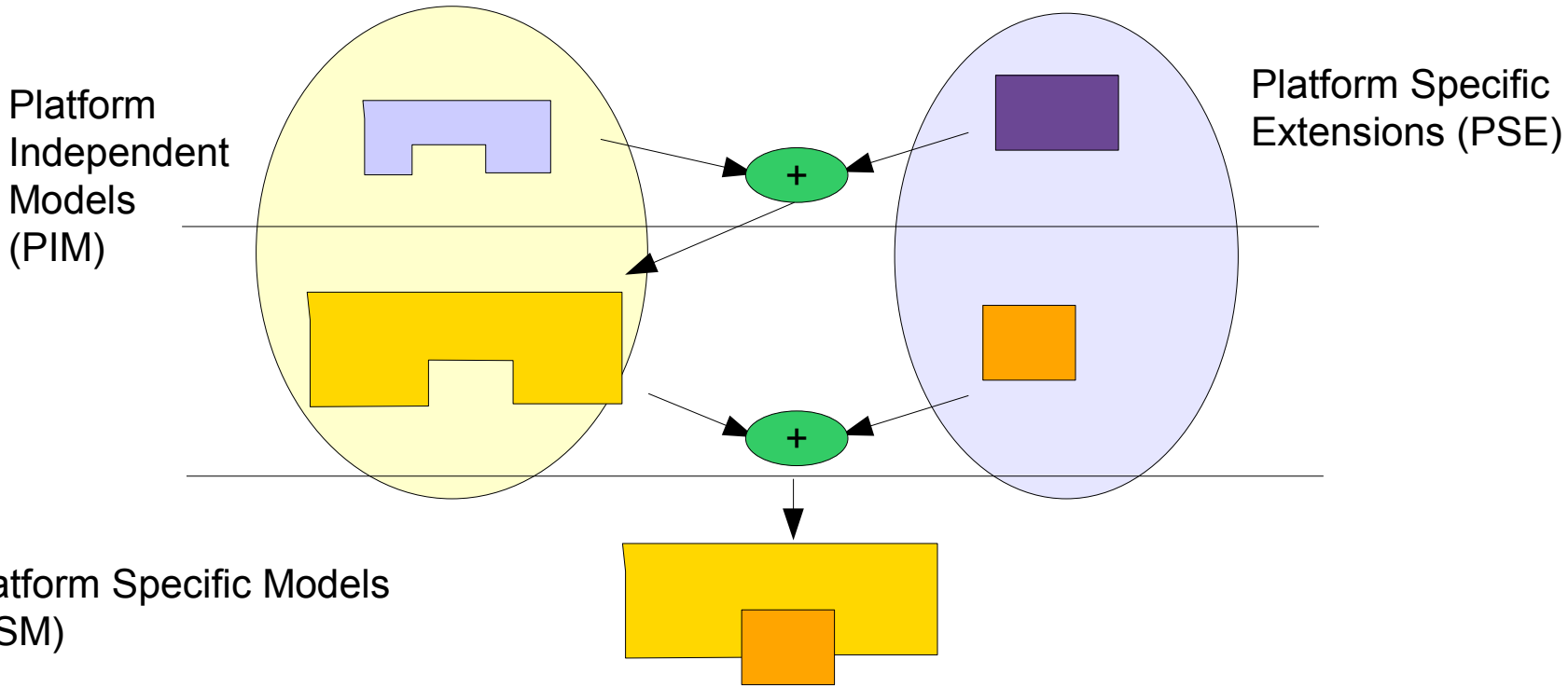
Unforeseeable Extensible MDA

- Aspect-weavers *extend* in *unforeseen ways*



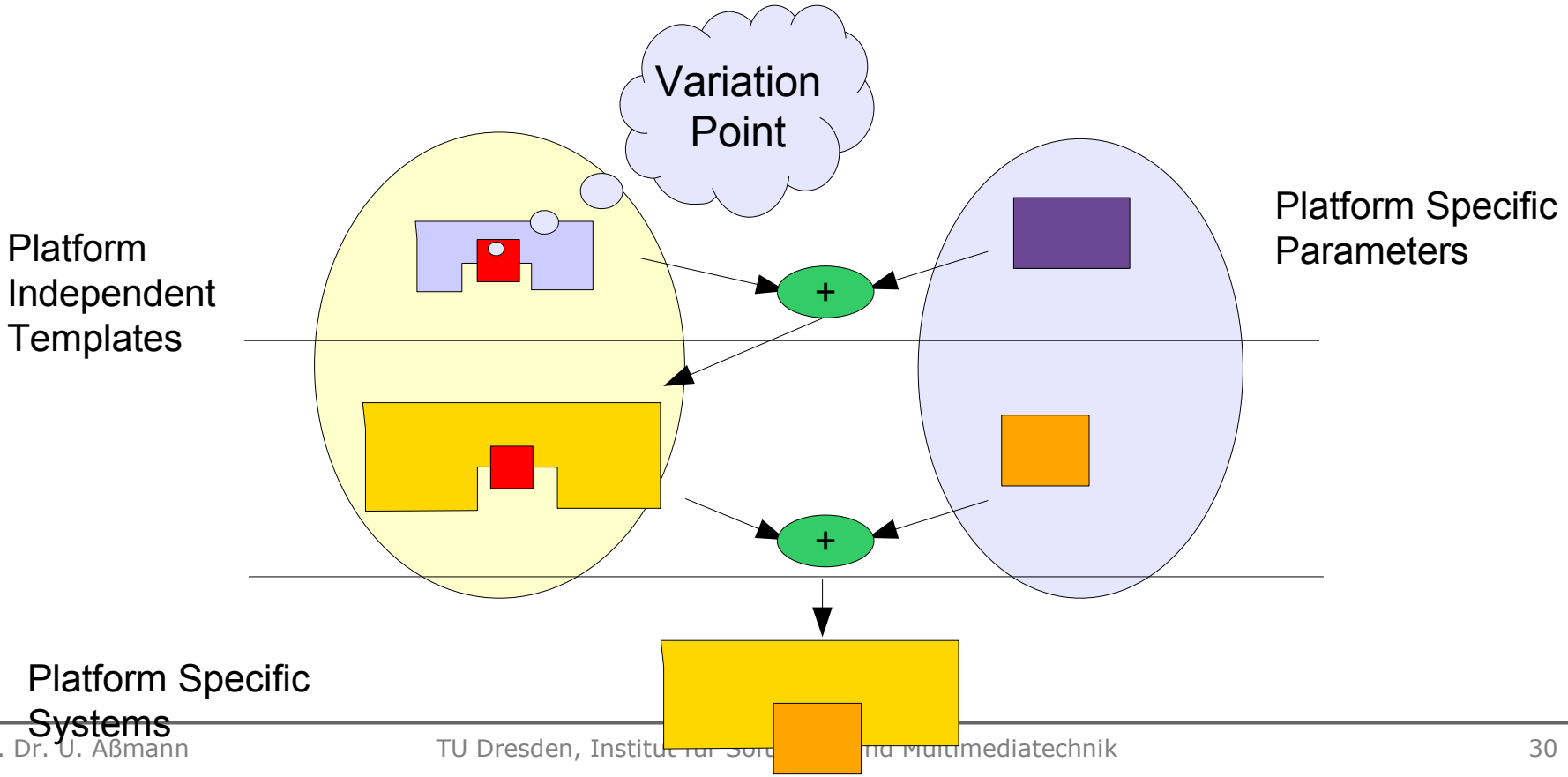
Unforeseeable Extensible MDA

- Hyperslices *extend* in *unforeseen ways,*
  - but not with crosscutting



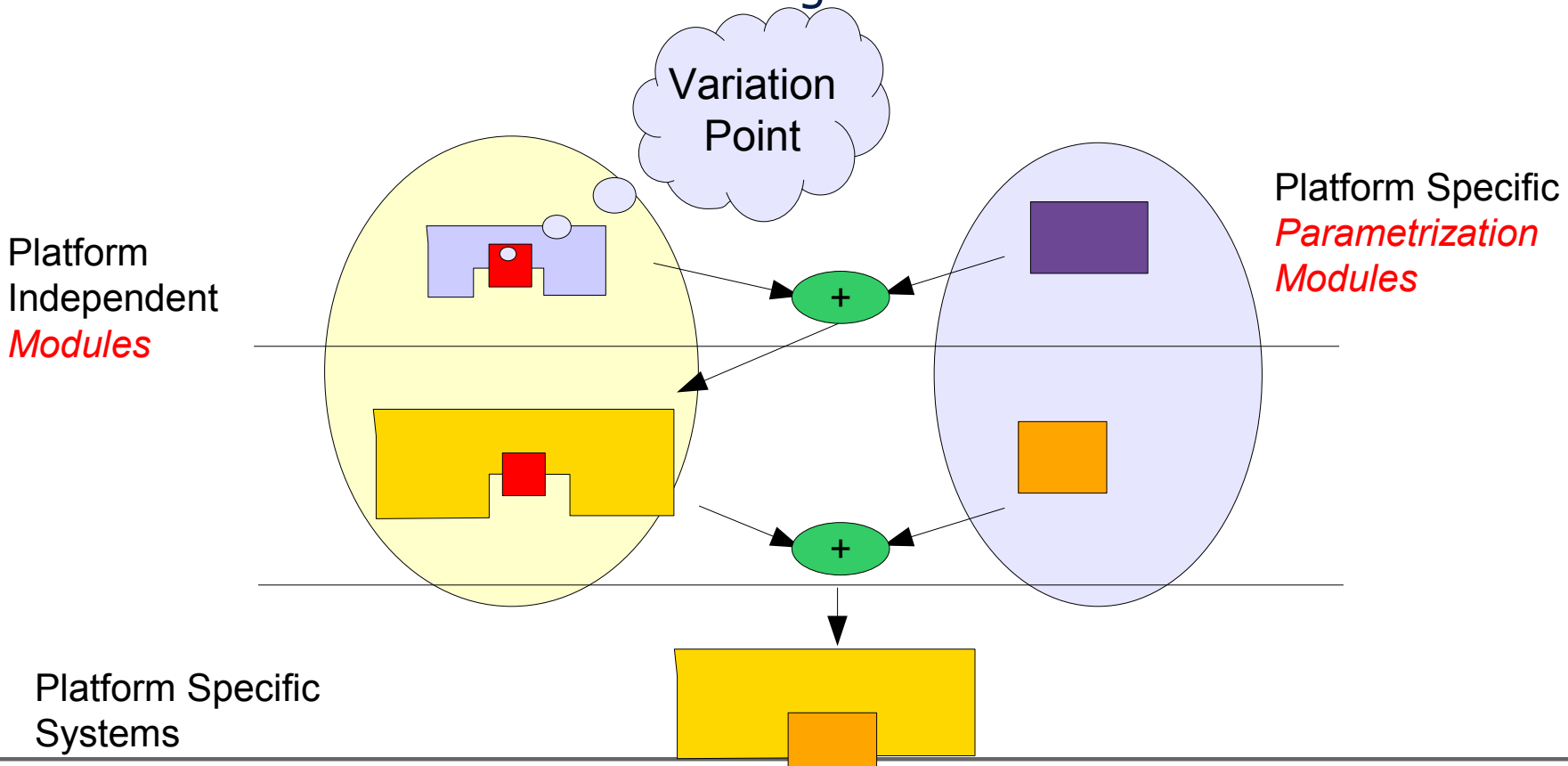
Planned MDA (Generic MDA)

- Templates *parameterize* in *planned ways*
  - Template interface is composition interface
  - but not with crosscutting



Planned MDA with Parnas

- Modules *parameterize* in *planned ways*
  - Module interface is composition interface
  - but not with crosscutting



Planning Variations - Unforeseen Extensions

- Planning relies on *variation points*
- MDA with
  - templates
  - modules
  - blackbox component models

- Extension relies on *extension points*
- MDA with
  - views
  - aspects

Conclusion: It's Your Choice!

- MDA can be employed for CVA and SCE!
  - with many different CBSE technologies
- MDA for commonality/variability-based design can be *planned*
  - Using a composition interface with declared variation points
- MDA for stability/extension-based design can be done
  - for *unforeseen extensions*
  - using a composition interface with *implicit hooks (join points)*

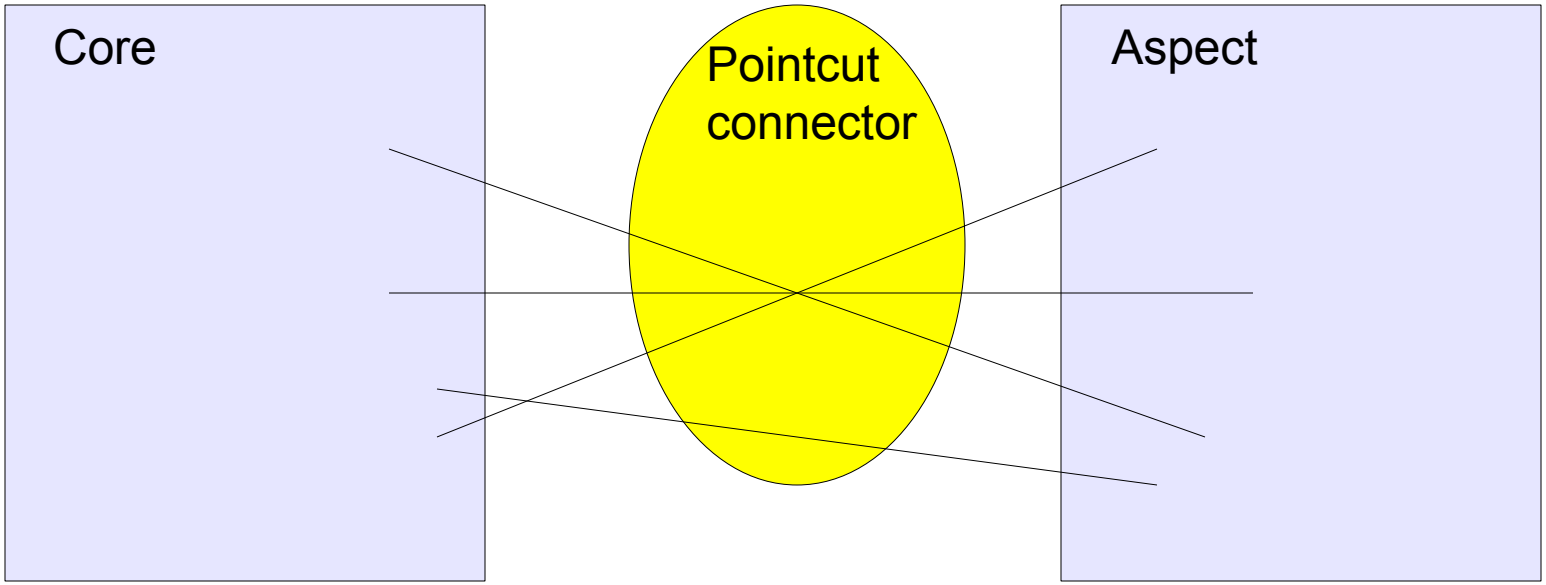
Tool Problems

Problem: Weaver Proliferation

- Who builds all these weavers and template expanders?
- Answer:
  - Universal pointcut languages
  - Universal composability add-ons

Universal Pointcut Languages

- Weaver proliferation can be avoided by *universal pointcut languages* that *interconnect* base languages



Universal Pointcut Languages

- A pointcut language connects *names* of the core and the aspect
  - does not know more concepts
- It can be used universally

- Example:
  - Xpath, can it be used as pointcut language?
  - Can you separate pointcuts from Aspect/J advices and address advice joinpoints?

Universal Composability Add-Ons

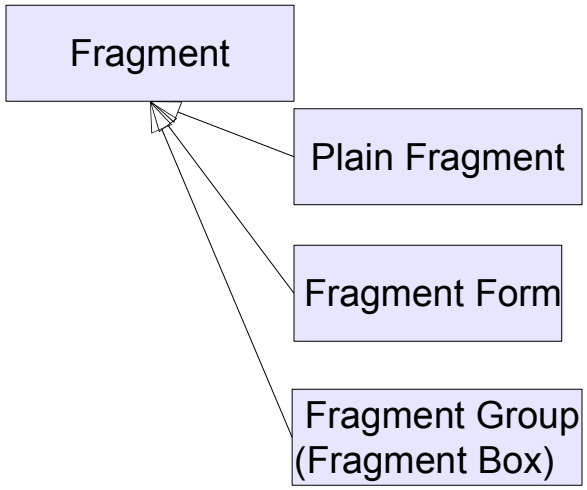
- Based on Universal Genericity and Extension
- Basic idea: provide templates and extensible fragments for every language

Universal Genericity

Component-Based Engineering with Fragments
The Universally Generic Language BETA

The Component Model of BETA

- The basic module in the BETA system is a *fragment*
  - **Plain Fragment**: Sentential form, a partial sentence derived from a nonterminal
  - **Generic Fragment** (fragment form, template): Fragment that still contains nonterminals (***slots***)
  - **Fragment Group** (fragment box): Set of fragments

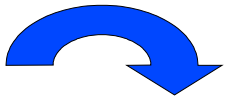



```
define fragment component PersonTemplate = {
  name '/home/assmann/PersonTemplate'
  Person : PatternDecl
  Person : begin
    PersonMembers : begin
      name : @String
      <<EmployerSlot : Attribute>>
    end
  end
}
```

Example of Composition in BETA


```
define fragment component PersonTemplate = {
  name '/home/assmann/PersonTemplate'
  Person : PatternDecl
  Person : begin
    PersonMembers : begin
      name : @String
      <<EmployerSlot : Attribute>>
    end
  end
}
```

```
define fragment component PersonFiller = {
  name `/home/assmann/PersonFiller'
  origin `/home/assmann/PersonTemplate'
  EmployerSlot: Attribute
  EmployerSlot: begin
    employer: @Employer;
    salary: Integer
  end
}
```



```
Person : PatternDecl
Person : begin
  PersonMembers : begin
    name : @String
    employer: @Employer;
    salary: Integer
  end
end
```

Example of Composition in BETA


```
define fragment component PersonTemplate = {
  name '/home/assmann/PersonTemplate'
  Person : PatternDecl
  Person : begin
    PersonMembers : begin
      name : @String
      <<EmployerSlot : Attribute>>
    end
  end
}
```

```
define fragment component PersonFiller = {
  name `/home/assmann/PersonFiller'
  origin `/home/assmann/PersonTemplate'
  EmployerSlot: Attribute
  EmployerSlot: begin
    employer: @Employer;
    salary: Integer
  end
}
```

```
fragment Person = PersonTemplate.
EmployerSlot.bind(PersonFiller);
```

```
Person : PatternDecl
Person : begin
  PersonMembers : begin
    name : @String
    employer: @Employer;
    salary: Integer
  end
end
```

Advantages

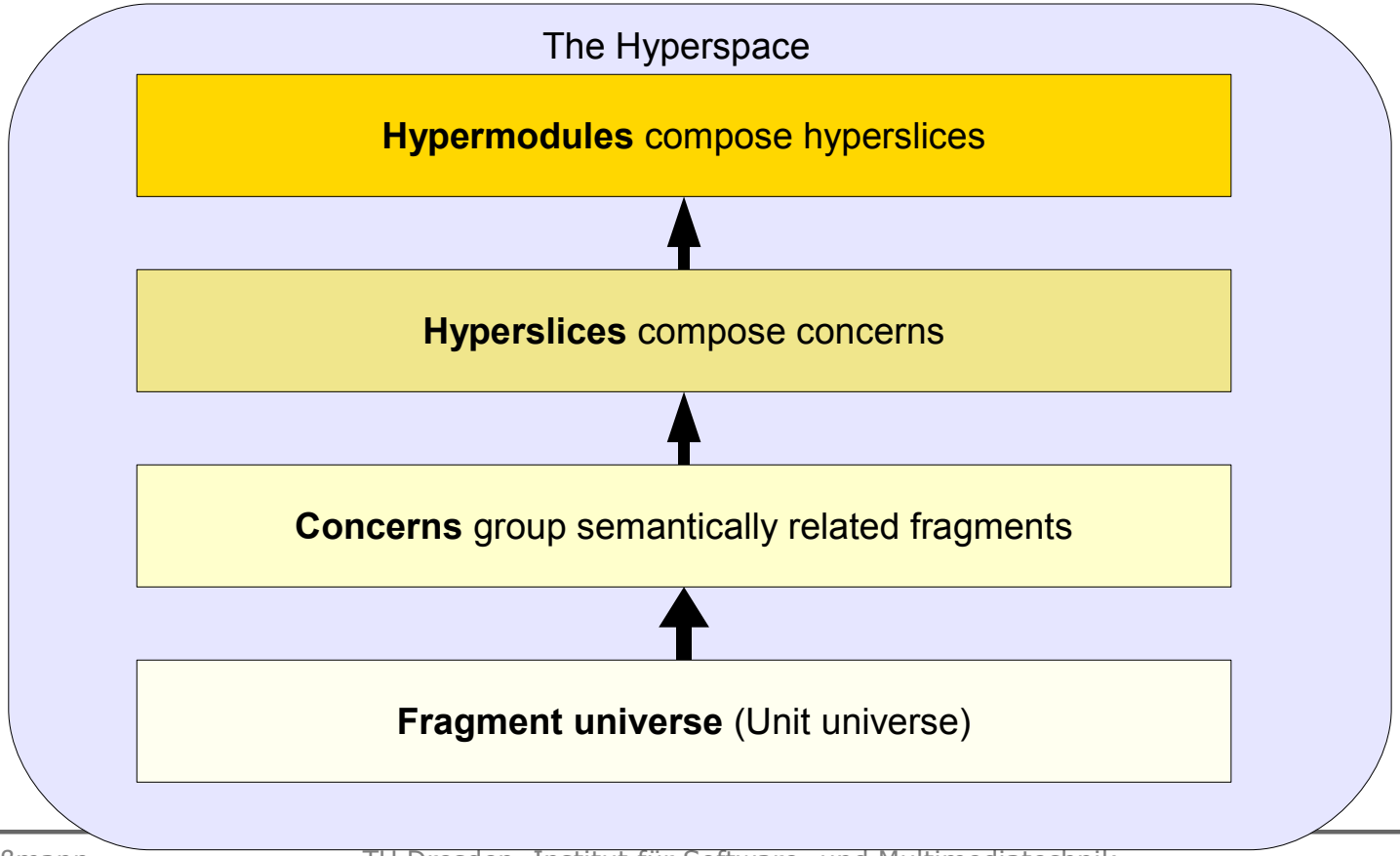
- Fine-grained *fragment component model*
  - The slots of a beta fragment form its *parameterization interface*
  - The BETA compiler can compile all fragments separately
  - All language constructs can be reused
  - Type-safe composition with composition operation *bind-fragment*
  - Mjölner metaprogramming environment is one of the most powerful software IDE in the world (even after 15 years)

Universal genericity: A language is called *universally generic*, if it provides genericity for every language construct.

Universally Extensibility in Hyperspace Programming

The Hyperspace, a Fragment Space

- View-based programming
- Composition operation: *merge of fragments* in concerns



```
concern PersonalConcern = {
    class Person {
       String name;
       int age;
    }
}
```

```
concern EmploymentConcern = {
    class Person {
       Employer employer;
       int salary;
    }
    class Employer { }
}
```

```
hyperslice Employment  =
PersonalConcern.merge(EmploymentConcern);
```

```
concern PoliticalConcern = {
    class Person {
       string politicalParty;
       int contribution;
    }
}
```

```
hyperslice Employment = {
    class Person {
       String name;
       int age;
       Employer employer;
       int salary;
    }
}
```

```
hyperslice PersonInfo  =
Employment.merge(PoliticalConcern);
```

```
hyperslice PersonInfo = {
    class Person {
       String name;
       int age;
       string politicalParty;
       int contribution;
       Employer employer;
       int salary;
    }
}
```

Advantages

- Compositional merge resp. extension of fragment sets
  - Classes
  - Packages
  - Methods
  - Hyperslices!

Universal extensibility: A language is called *universally extensible*, if it provides extensibility for every collection-like language construct.

Universal Genericity vs Universal Extension

- BETA and hyperspaces look really similar
  - Fragment components
  - *slots* vs *hooks* (parameterization vs extension interface)
  - *bind* vs *merge* composition operations
- BETA is a *generic* component approach
- Hyperspaces is an *extensible* component approach

Universal composability: A language is called *universally composable*, if it provides universal genericity and extension.

Universal Reuse Add-On Languagess

for universal genericity and extension

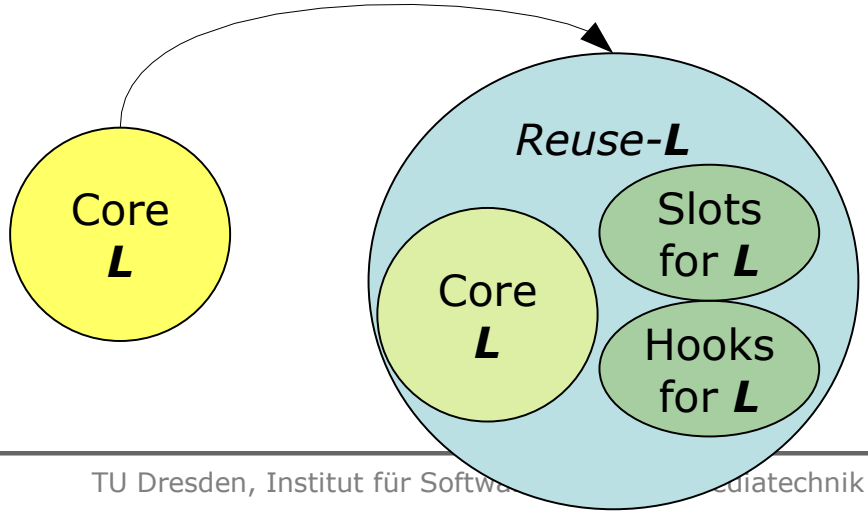
Universally Composable Languages

Universally composable: A language is called *universally composable*, if it provides universal genericity and universal extensibility

Reuse add-on language: Given a meta-model of a *core* language L, a metamodel of a universally composable language can be generated (Reuse-L)

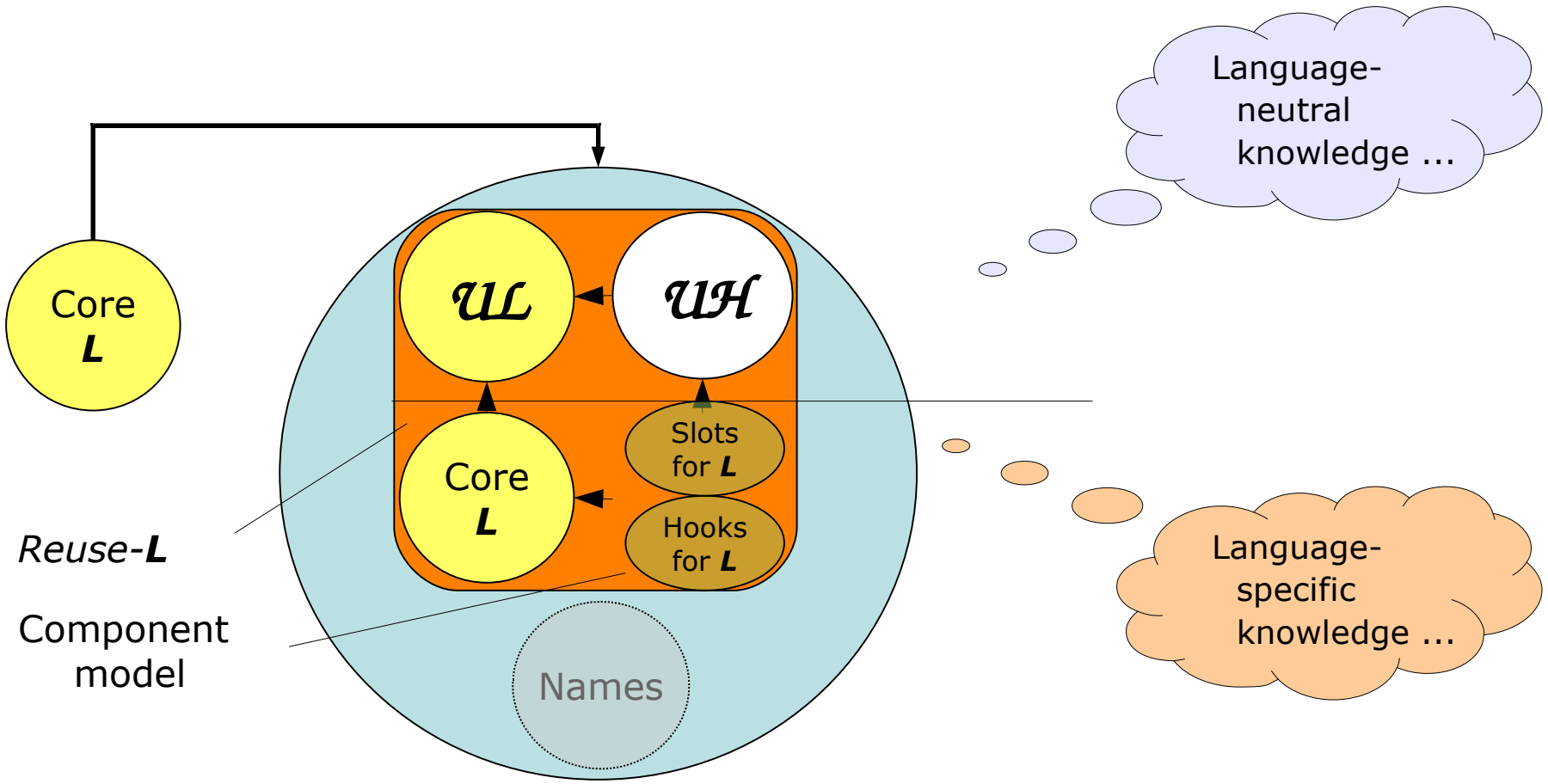
Slot and Hook model: Generated from the core language metamodel

- realizes universal composability by defining *slots* and *hook constructs*, one for each construct in the core language



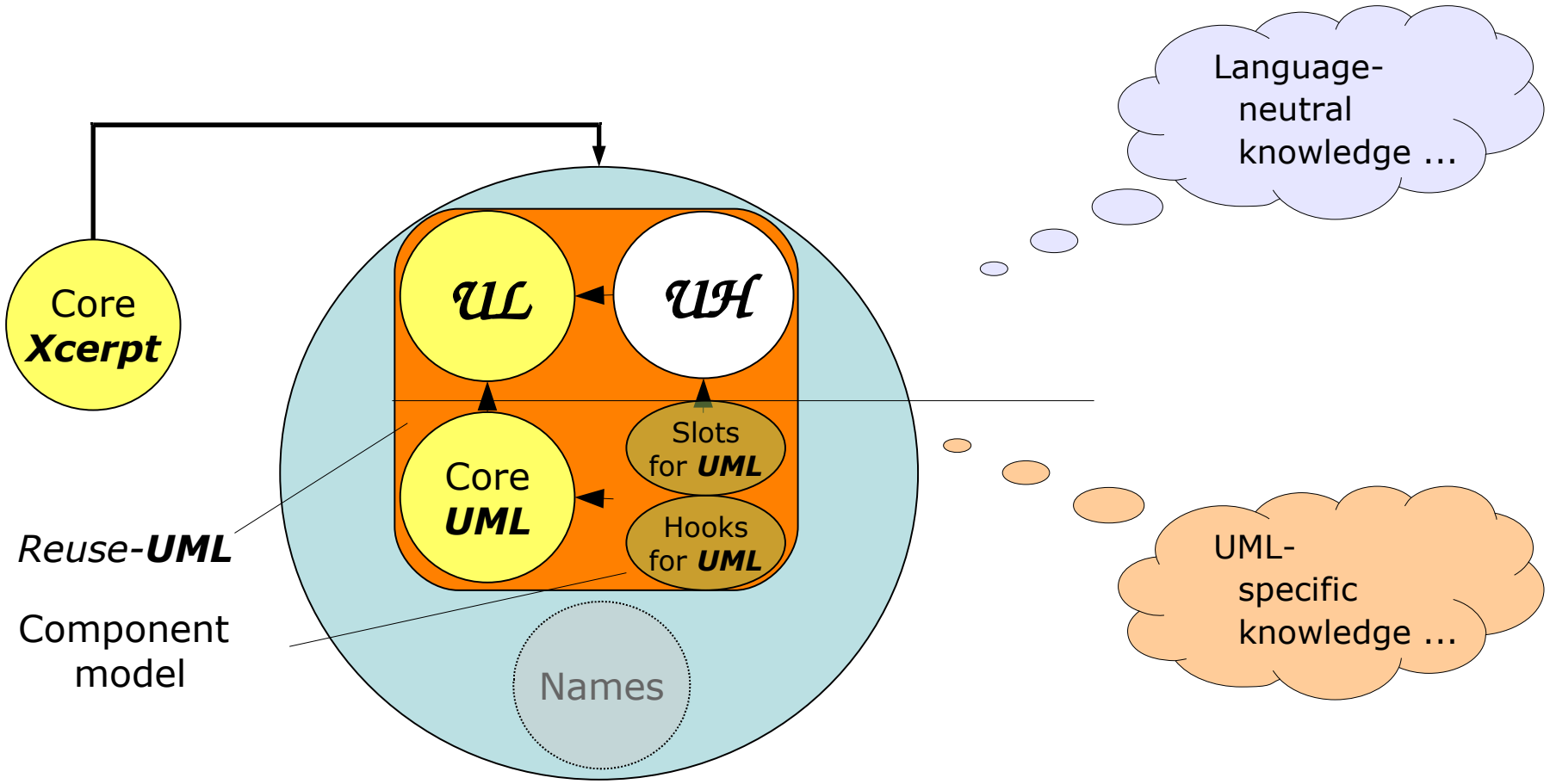
Structure of a Universally Composable Language

- The reuse language has two levels...

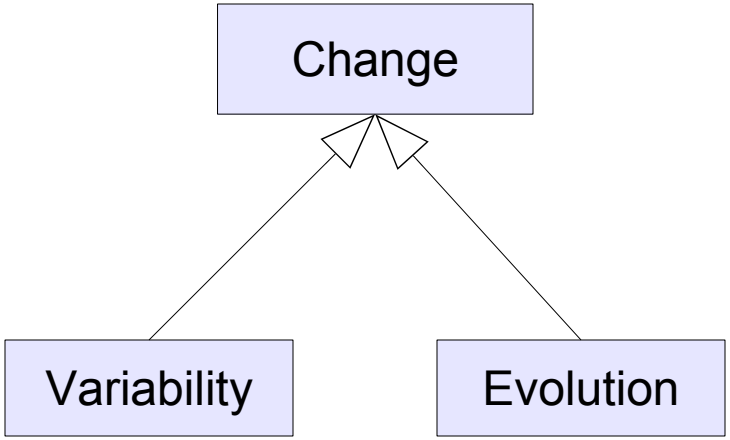


Reuse-UML, a Universally Composable Language

- .. an extension of UML with slot and hook model

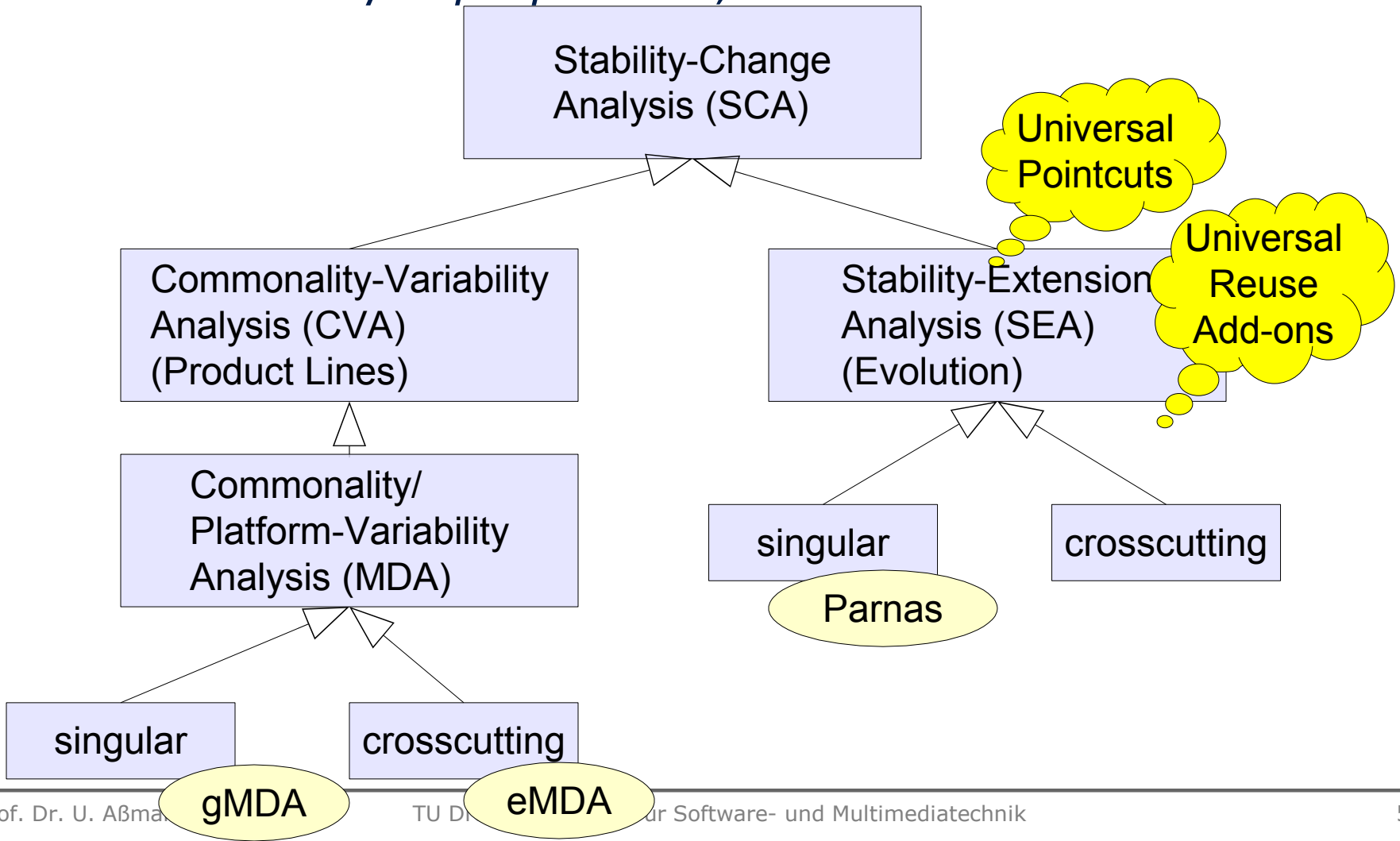


Change



Future MDA with SCA

- Variability is *preplanned,* evolution is *unforeseen*



MDA is Concern Separation

- MDA is **not about platforms**
- MDA is a design framework approach with subcategories
  - Generic MDA (for variation)
    - Templates (mappings are parameterizations)
    - Modules
    - Connector refinements (mappings are refinements)
  - Extensible MDA (for unforeseen extension)
    - Views (mappings are extensions)
    - Aspects
  - Translation-based, Rewrite-based MDA

The Real MDA

- MDA is a CBSE approach

- Greybox composition
  - PIM and PSE are *greybox components*
- Blackbox composition
  - PIM and PSE are *blackbox components*
  - Without unforeseen extension – everything is planned
  - You can do it in C, but you must plan carefully

The End

- http://st.inf.tu-dresden.de
- U. Aßmann. Invasive Software Composition. Springer, 2003
- Aßmann, Uwe, Johannes, Jendrik, Henriksson, Jakob, Savga, Ilie, Composition of Rule Sets and Ontologies. Reasoning Web, Second International Summer School 2006, pp. 68-92, 2006, LNCS 4126, Springer