40. Integrational Ways to Decompose and Compose

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de

Version 16-0.2, Mai 25, 2016

1. Decomposition and Composition
   1. Example role modeling
2. Systems with Dimensional Decomposition and Composition
   1. LambdaN calculus
   2. Piccola

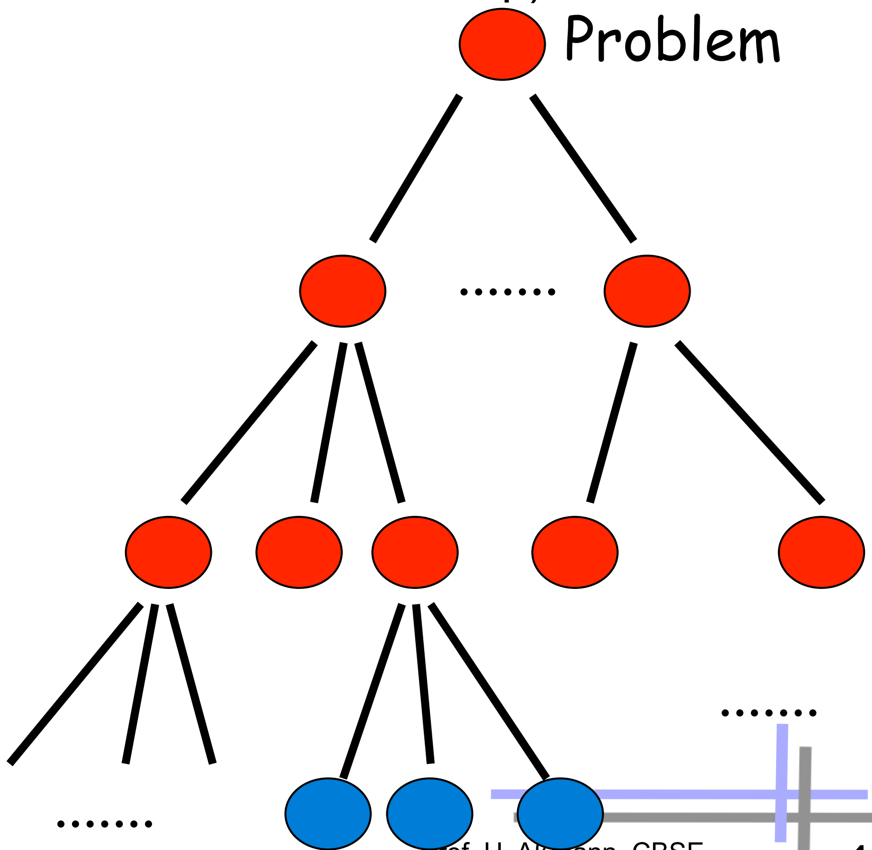
Obligatory Literature

- [Dami95] Laurent Dami. Functions, Records andCompatibility in the Lambda N Calculus in Chapter 6 of “Object-oriented Software Composition”. http://scg.unibe.ch/archive/oosc/PDF/Dami95aLambdaN.pdf
- Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, Object-Based Models and Langages for Concurrent Systems, LNCS 924, pages 147-161. Springer, 1995.
- Optional:
  - Dami, Laurent. Software Composition. PhD University Geneva 1997. The centennial work on the Lambda-N calculus
  - F. Achermann. Forms, Agents, and Channels. Defining Composition Abstraction with Style. PhD thesis. Unversity Berne 2002. Available from Oscar Nierstrasz' Software Composition Group's pages scg.unibe.ch.
    - This web site is great, one of the best sites for composition. Many papers of Nierstrasz and his PhD students show all aspects of composition. Visit it!

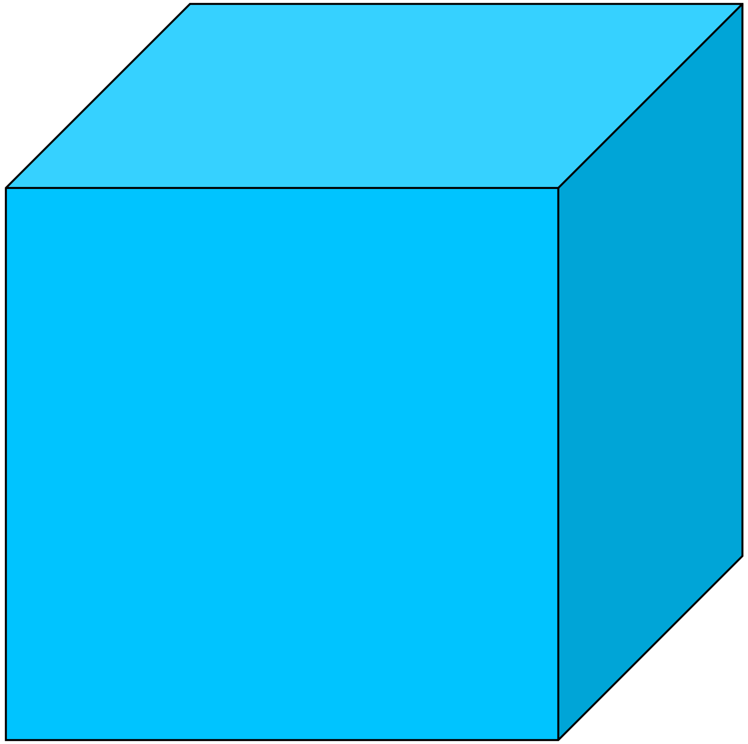
40.1 Decomposition and Composition

Problem Solving with Divide and Conquer Strategy

- **Divide et impera** (from Alexander the Great)
  - divide: problems into subproblems
  - conquer: solve subproblems (hopefully easier)
  - compose (merge): compose the complete solution from the subsolutions
- Methods of (De)composition. We decompose
  - To simplify the problem
  - To find solutions in terms of the abstract machine we can employ
  - When this mapping is complete, we can compose
- Reuse of partial solutions is possible (then the tree is a dag)
- Leafs are operations of a given abstract machine (may be the software or the chip)

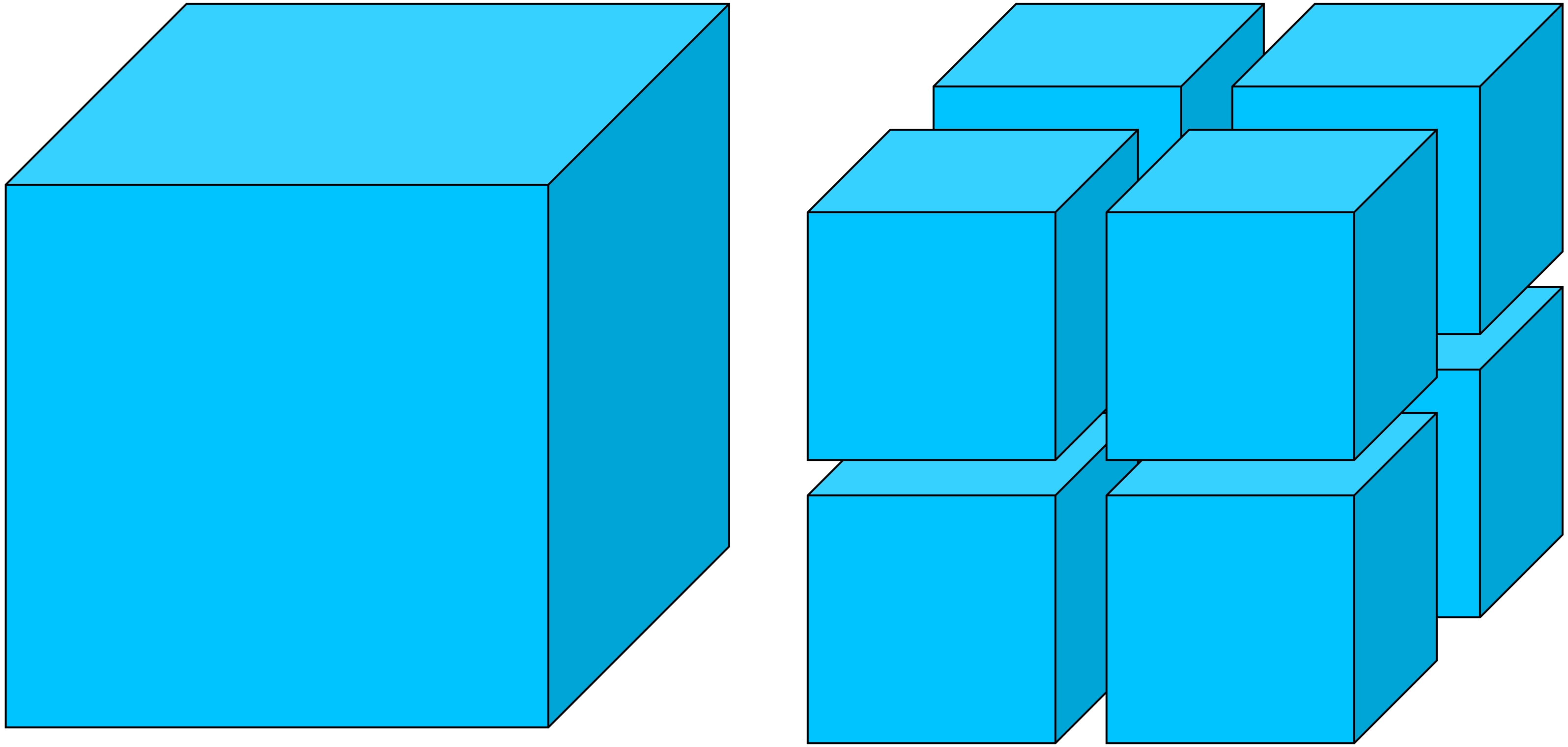


How to Decompose a Cube?



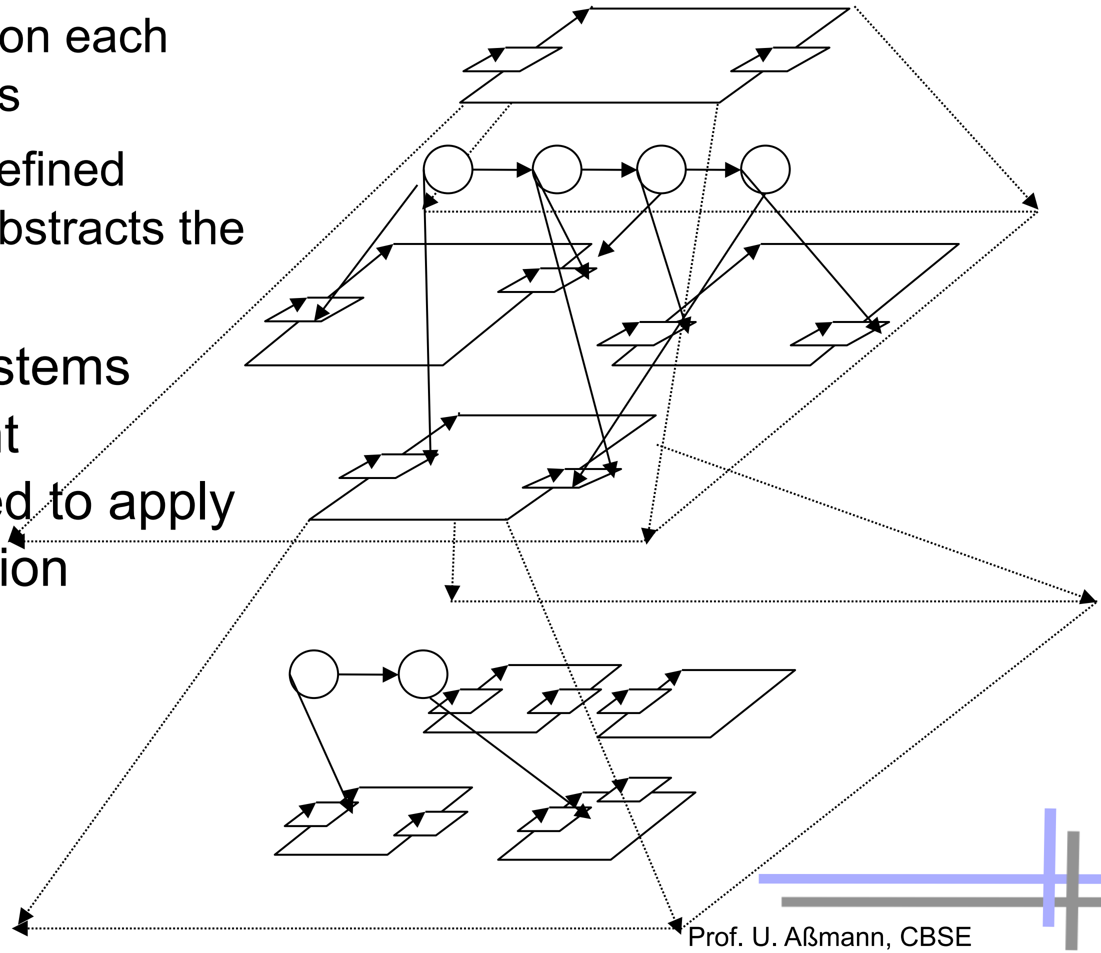
Blockwise Decomposition

- Blockwise decomposition is stepwise refinement
  - Problem size is reduced, dimensionality stays the same



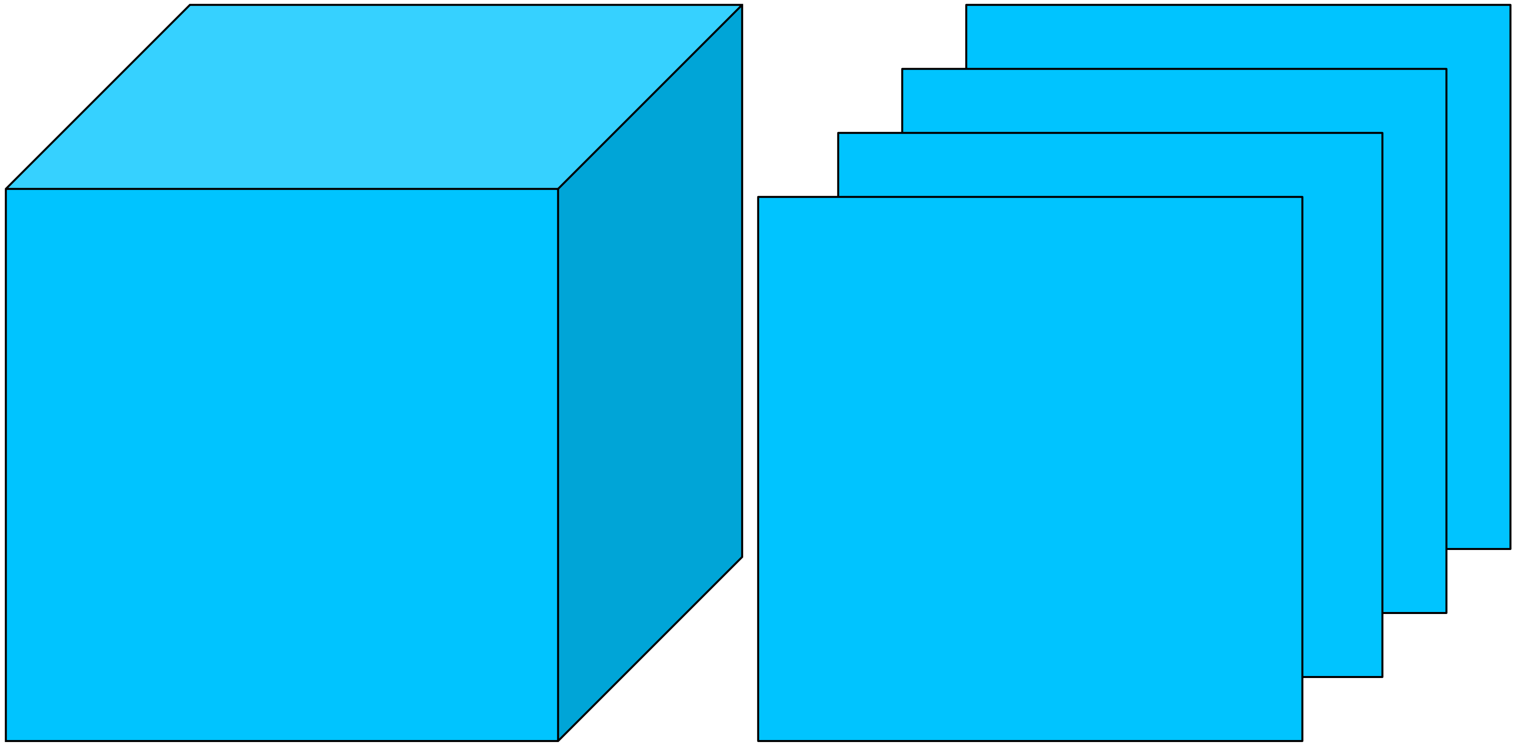
Refinement leads to Reducible Hierarchies and Graphs

- Trees or dags result
  - can be layered
- Reducible graphs result
  - Can be layered too, on each layer there are cycles
  - Every node can be refined independently and abstracts the lower levels
- Component-based systems contain the component hierarchy, so they need to apply blockwise decomposition



Hyperspace Decomposition (Dimensional Decomposition)

- Decomposition is not point-wise
- Problem size is retained; number of dimensions is reduced



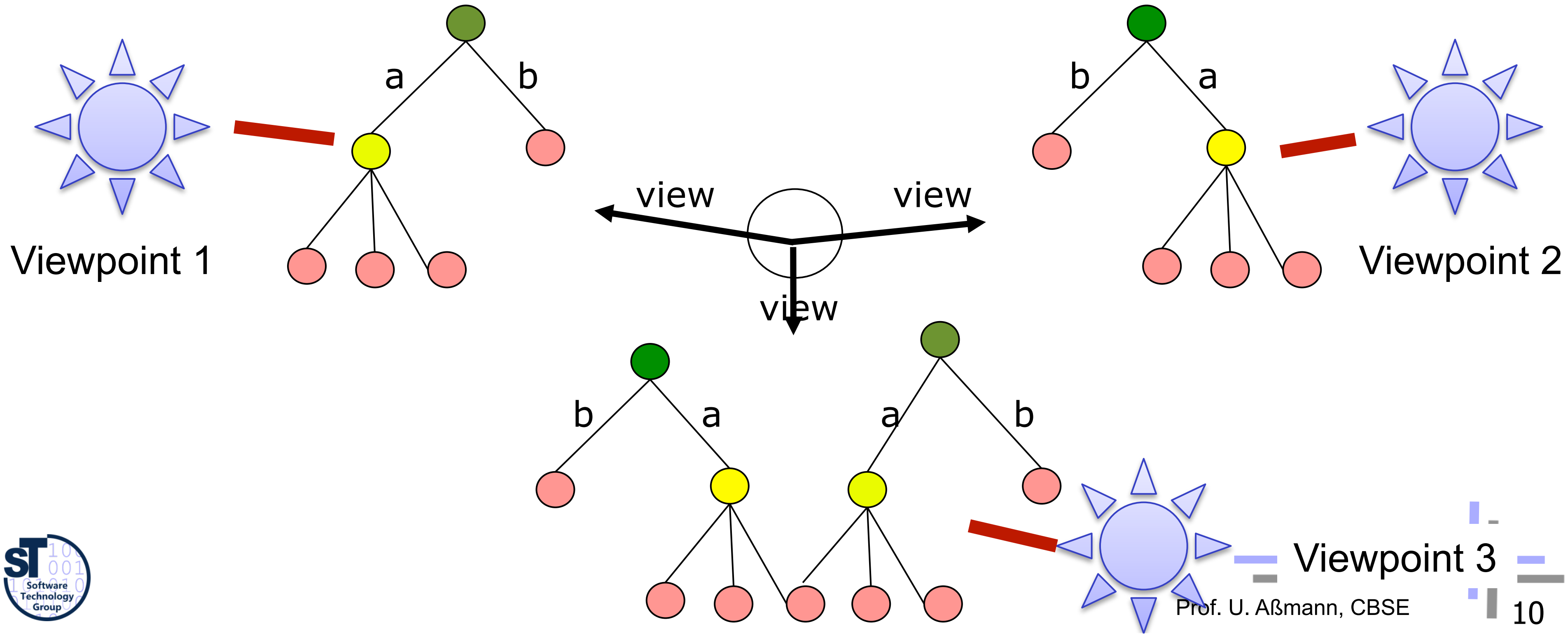
Basic Decomposition Strategy II: Separation of Concerns (SoC)

- **Separation of Concerns** (dimensional divide and conquer, dimensional (de-)composition) splits a problem into hyperplanes (or dimensions)
  - Problem dimension count is reduced
  - Problem size is not reduced
- After separation of concerns, the problem can be solved into subsolutions
- The subsolutions are reintegrated by *grey-box composition* or *integrational composition*
- A **viewpoint** defines a *set of related concerns*, producing a partial representation of a system (**view**)

> A **view** is a representation of a whole system from the perspective of a related set of concerns
> [ISO/IEC 42010:2007, Systems and Software Engineering -- Recommended practice for architectural description of software-intensive systems]

Separation of Concerns leads to Dimensions

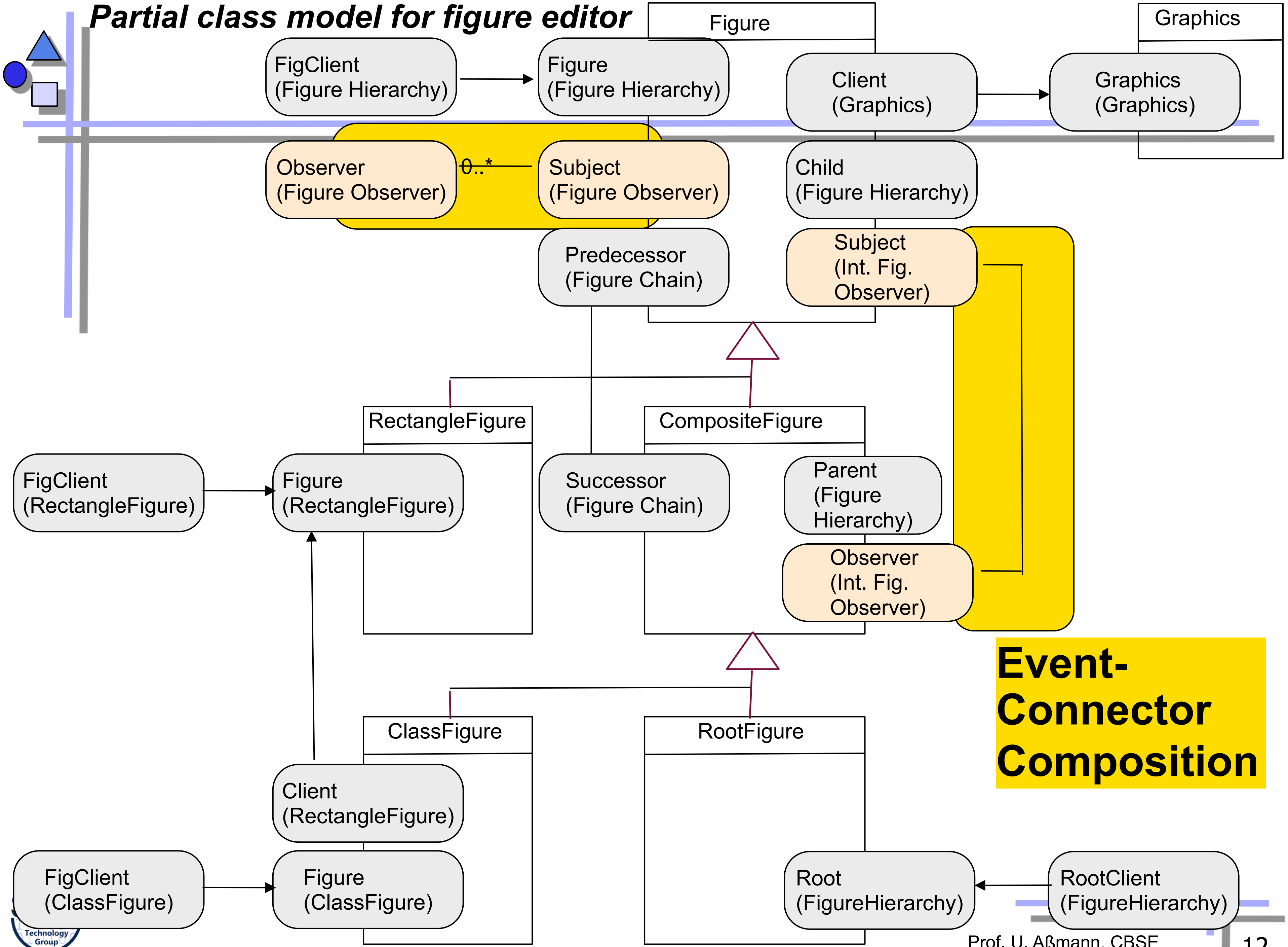
Dimensional (de-)composition (separation of concerns)
needs *projection operators* for decomposition
and *merge operators* for composition



40.1.2 Role Composition and Decomposition in the Role Component Model

Role modeling is a dimensional, view-based specificiation technique

Partial class model for figure editor

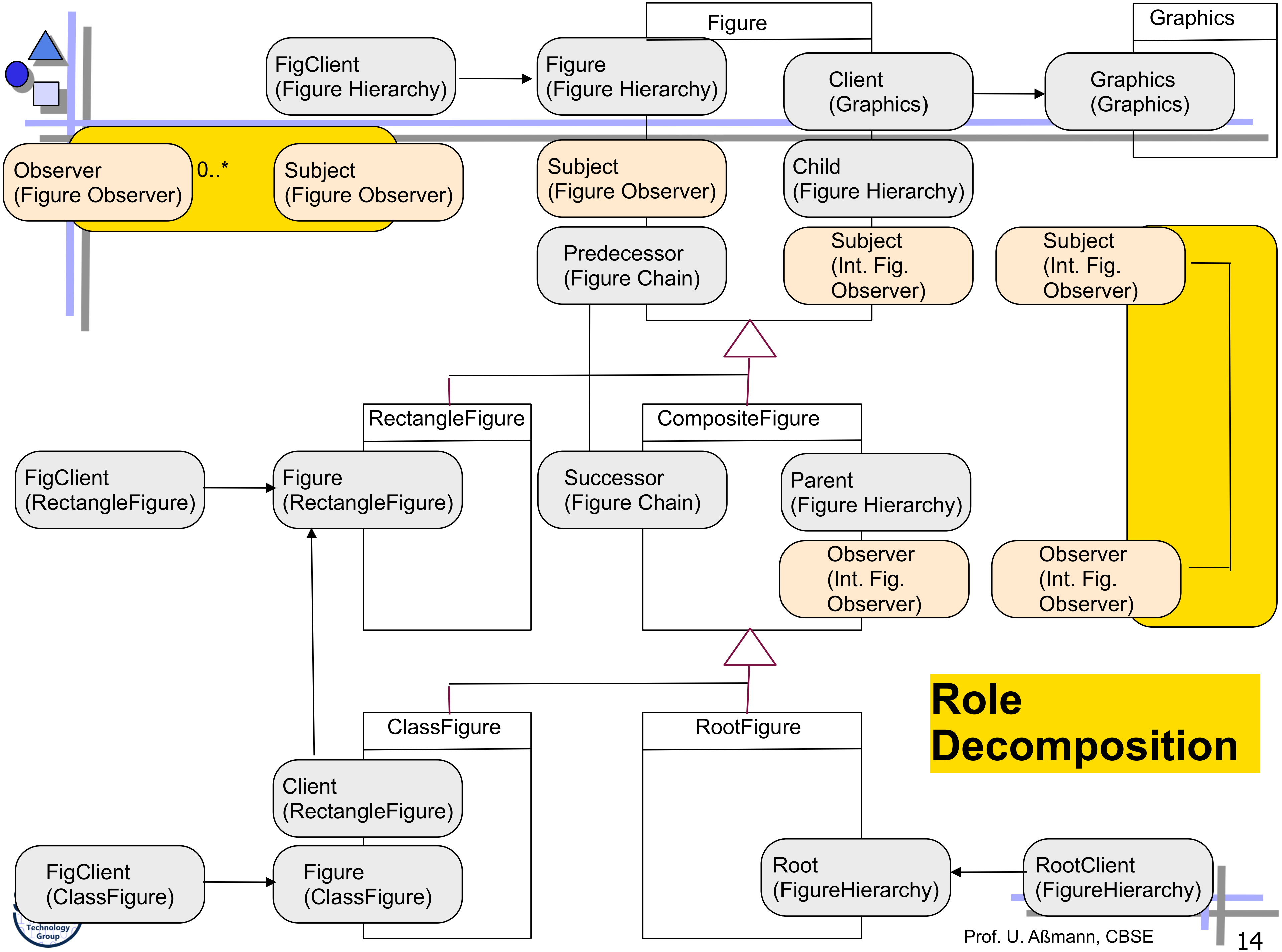


Event-Connector Composition

Role Models are Being Composed

- Roles are *merged to classes* by role allocation (binding)
- Role models can be decomposed (projected)
  - By role splitting
- And integrated
  - By role merge or identification

Role Decomposition



Insight: Role Component Model

- Because their role models are *integrated* with the role model of the component, connectors work with grey-boxes (Integrating)
- Roles are a grey-box component model!

Role-based design relies on a greybox component model: composition by role merging decomposition by role split

40.2 Systems with Composition Languages for Dimensional De- and Composition

Function Merge in the LambdaN Calculus

- An extension of the Lambda-calculus [Dami97]
  - Argument passing by name: arguments have names by which they are handed over to the callee (as in Ada)
  - No positional parameters as in standard lambda calculus

```
f(p1 => value1, p2 => value2);
==  f(p2 => value2, p1 => value1);

f = function (p1, p2) { ...
  implementation ... }
```

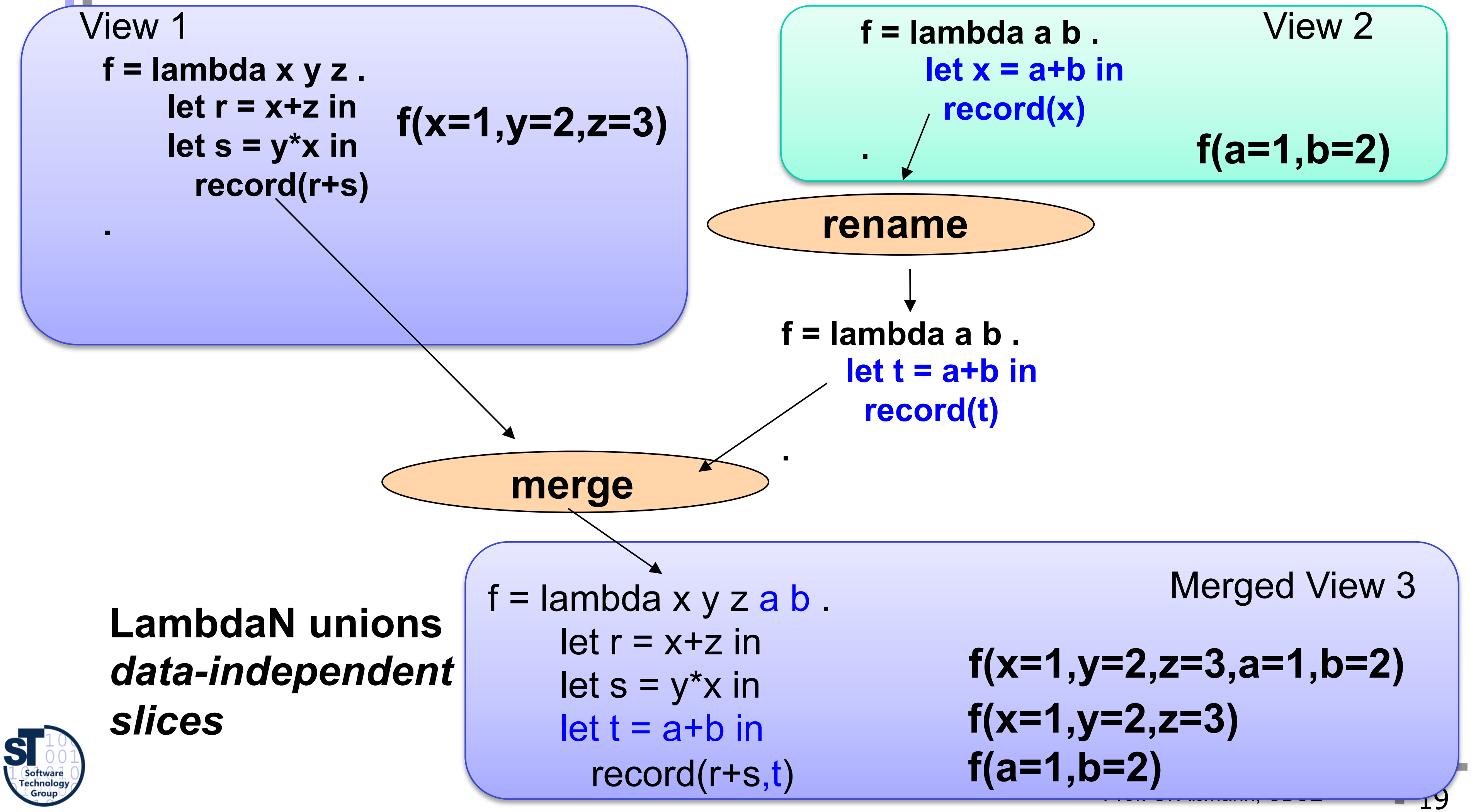
- Some new reduction rules for the calculus that deal with
  - Name-based argument passing
  - Renaming of names
  - Merging of functions

Function Merge in the LambdaN Calculus

- Functions (lambdas) can be multiply defined and *merged*
  - A function is an *open definition,* which can be extended later on
  - The component of LambdaN is an extensible Lambda with positional parameter passing
- The LambdaN-calculus is based on one simple code merge rule, the merging of lambda expressions (*merge operator* for extensible functions)
  - Currying is possible in arbitrary order
- LambdaN is the first code calculus for *merge* of code, i.e., for code composition

Example of Function Merge

- Merging of *slices* (black vs blue): rename variables uniquely, merge
- For partially parameterized calls, the merged f in View 3 returns partially evaluated function



LambdaN unions *data-independent slices*

Class Merge Operator in the LambdaN Calculus

- **Class views:** Classes can be merged by merging the set of functions
  - In LambdaN, a class is just a set of functions
  - The merge operator merges implementations, not only of interfaces
- **Class-role views:** Role types are partial classes, role models are collaborations or roles
  - Role model merge can be reduced to lambda merge of roles models and classes
- **Connector views:** Connectors are special class-role models, so views on connectors can be defined
- **Architectural views:** Views on component-connector configurations
- Therefore, LambdaN can model
  - Role models
  - Classes in object-oriented languages with polymorphism, inheritance, etc.
  - Views on methods and classes
  - Views on components of any grain size
  - Connector views, architecture views can be realized, i.e., the calculus enables view-based architecture systems

The Power of LambdaN

- Consequence: LambdaN is the perfect calculus to model the semantic base for systems with dimensional decomposition and composition

- Hence, **LambdaN can describe many grey-box compositions**
  - Composition Filters (wrapping a filter is a merge of a filter with an object)
  - Parameterizations (well the calculus is higher order, and functions can be passed as arguments)
  - View-based and aspect-oriented programming (see later)
- The LambdaN calculus is *invasive* since functions are merged, i.e., extensions are embedded into extended parts

Sound Composition in the LambdaN

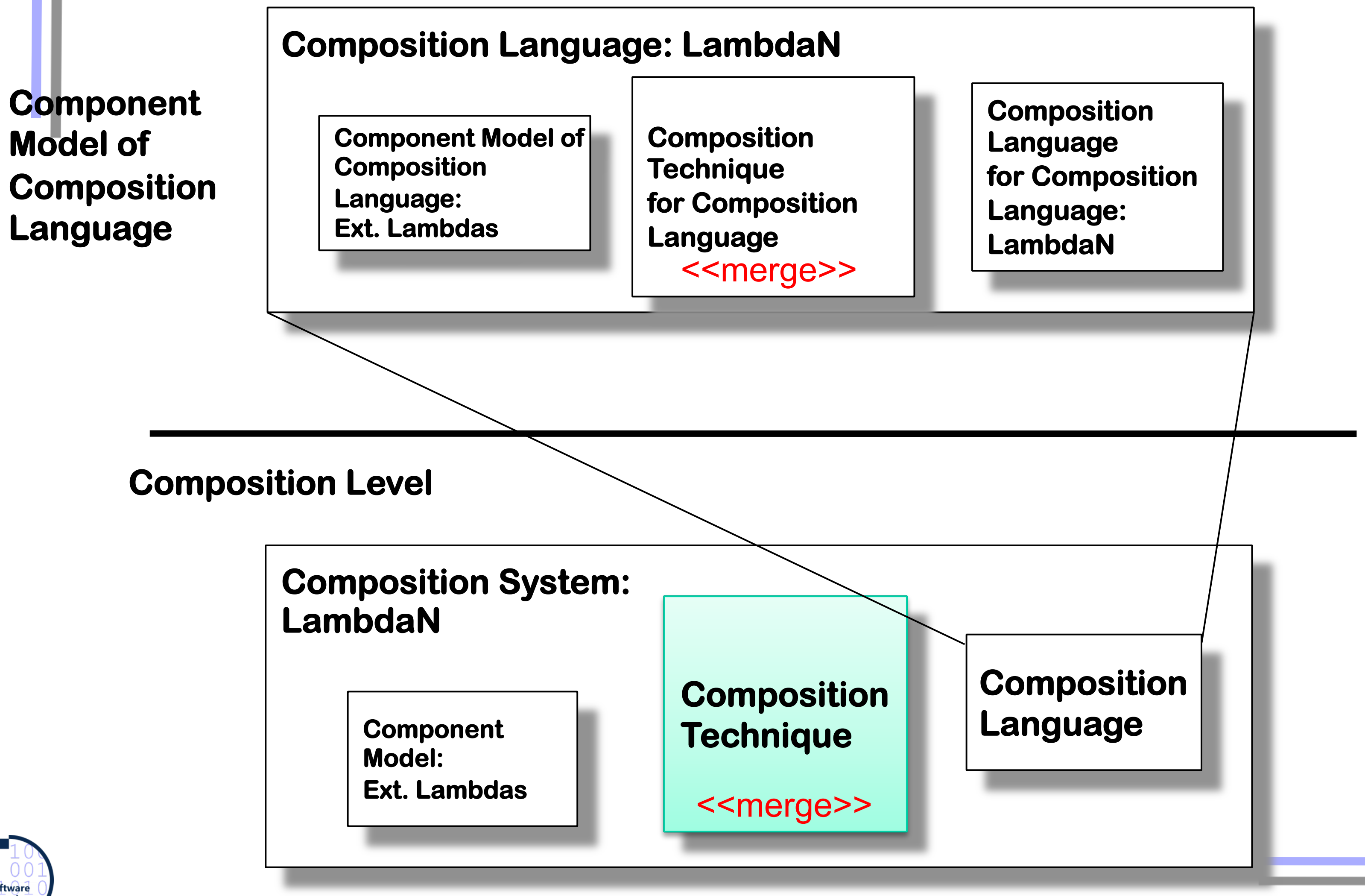
- Def.: A method m is ***conformant*** to a method n if it can safely replace n in all uses.
- **Theorem**: The result of a merge operation in LambdaN is conformant to its operands (its *origins*)
  - The resulting f of the previous example is conformant to both of its “ancestors”

- Conformance allows for *safe* composition operations in LambdaN applicaitons:
  - Extension is safe
  - Merging is safe
  - Adaptation, glueing (with connectors) is safe

The Composition Language of LambdaN

- LambdaN is a higher-order calculus, i.e., is its own composition language
  - Like in lambda-calculus, functions can be composed and merged by composition programs
- The composition language
  - It is turing complete
  - It is confluent, i.e., deterministic
  - It is view-based itself, i.e.,
- LambdaN is a sound basis for the next 700 composition languages

View-Based Programming with LambdaN Relies on the Merge Operator (Safe Merge)



Pi-Calculus

- The pi-calculus is a calculus for parallel processes (from Milner)
  - A process algebra.
  - Similar to CSP of Hoare
  - Channels (streams) for communication, instead of functional application
- Pi-calculus scripts model parallel component semantics
  - But also composition semantics
- The pi-calculus is an “assembler” of composition
  - Non-invasive, i.e., components are black boxes
  - But pi generates glue
  - Higher order, i.e., has its own composition language
- Pi is another base language for composition

Piccola

- [Nierstrasz, Schneider, Lumpe, Achermann] from Bern University
- Derived from Pi-calculus and LambdaN
  - Introduces extensible records for the pi calculus (forms)
  - With these records, all features of LambdaN are inherited
  - Piccola is fully extensible, as LambdaN
  - Higher level language concepts can be mapped to the pi calculus
- More abstract language, much easier to program
- Watch out for that group!

History

- 1988 Inheritance Anomaly, Composition Filters (Aksit)
- Beginning of the 90s: Nierstrasz talks about “Software Composition”
- 1993: Ossher invents subject-oriented programming, an early form of greybox composition
- 1994: Composition Filters (Bergmans, Aksit)
- 1996: Invention of AOP (Kiczales)
- 1997: LambdaN calculus (Dami)
- 2002: Piccola (Achermann, Nierstrasz): parallel extensible software

The End