

# On Actors and Objects - OOP in System Level Design

Philipp A. Hartmann   Joachim K. Anlauf

Computer Science Department, Division II  
Technical Computer Science  
University of Bonn  
{hartman, anlauf}@cs.uni-bonn.de

Forum on Specification and Design Languages, 2004

# Outline

- Objectives
- Combining Component Based Modelling and Object Orientation
  - Motivation
  - Type, Instance & Object
  - Communication & Connectivity
- Combining Inheritance and Behaviour Alternatives
  - Alternatives vs. Inheritance
  - Implementation graph
  - Example
- Summary

# Objectives

## Common Framework for Hardware and Software Design

- from System Level down to RTL
- existing object-oriented algorithms as starting point

## Internal Representation for System Level Design

- combination of **component based** modelling and **object orientation**
- supporting the whole design process  
→ **behaviour alternatives**
- reconfiguration & mobility
- resource modelling, execution semantics, ...

# Outline

- Objectives
- Combining Component Based Modelling and Object Orientation
  - Motivation
  - Type, Instance & Object
  - Communication & Connectivity
- Combining Inheritance and Behaviour Alternatives
  - Alternatives vs. Inheritance
  - Implementation graph
  - Example
- Summary

# Actors vs. Objects

## Actors

- concurrent, **interconnected** components
- communication e.g. through channels

## Objects

- rather passive, **not explicitly interconnected** data
- behaviour is accessed through abstract “method calls”

## Conclusion

Separate connectivity and behavioural aspects even in OOP!

# Hardware vs. Software Design

## Hardware

- focus on **behaviour**
- components correspond to autonomous entities  
→ **local behaviour**

## Software

- focus on **data**
- shared resources (code, CPU ...) → **shared behaviour**

## Conclusion

Separate representation of data & behaviour!

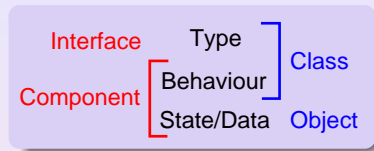
# Hardware vs. Software Design

## Hardware

- focus on **behaviour**
- components correspond to autonomous entities  
→ **local behaviour**

## Software

- focus on **data**
- shared resources (code, CPU ...) → **shared behaviour**



## Conclusion

Separate representation of data & behaviour!

## Type, Instance & Object

Type  $\hat{=}$  Interface

- defines sets of
  - **members**
  - **service** prototypes
  - **ports**
- includes type/inheritance hierarchy

Instance  $\hat{=}$  Behaviour

- allocation of behaviour for (a subset of) services
- members have an associated instance

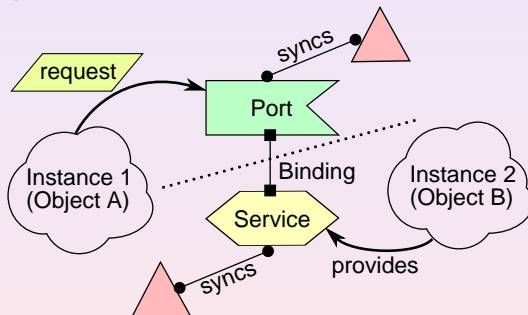
Object  $\hat{=}$  Data

- value/state of a certain type
- can only be accessed through an instance (see below)



# Separating Behaviour and Connectivity

## Service/Request Scheme



**Requests** are always sent to **local** ports,  
which are bound to (external) **behaviour!**

# Binding

## Objects → Instances

- objects have to be bound to instances
- **multiple** objects can be bound to **one** instance simultaneously.

## Ports → Instances

- ports are bound to **instances**, not to objects
- requests to a port are independent of currently bound object

# Outline

- Objectives
- Combining Component Based Modelling and Object Orientation
  - Motivation
  - Type, Instance & Object
  - Communication & Connectivity
- Combining Inheritance and Behaviour Alternatives
  - Alternatives vs. Inheritance
  - Implementation graph
  - Example
- Summary

# Alternatives vs. Inheritance

## Behaviour Alternatives

- Useful during design process to evaluate different implementation variants
  - Hardware vs. Software implementation
  - Different levels of abstraction

## Inheritance

- Encapsulation, Specialisation, Generalisation
- Using inheritance to model different variants?
  - Leads to a different **type**  
→ Unwanted in general!

## Conclusion

Concept of alternatives is needed in addition to inheritance!

# Implementation graph

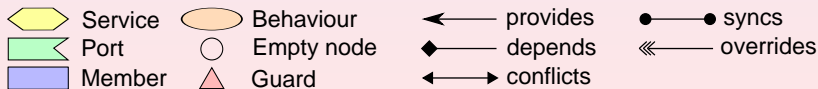
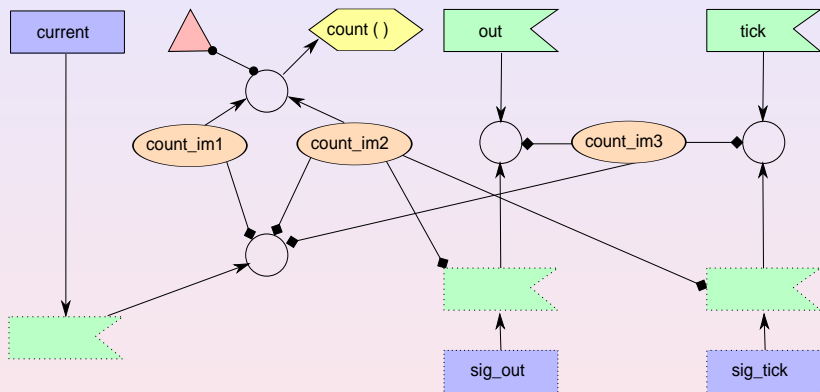
## Representation of Inner Structure

- directed, layered graph
- behaviour and synchronisation of services and its dependencies
- type hierarchy modelled via layers
- alternatives modelled via empty nodes and/or multiple edges

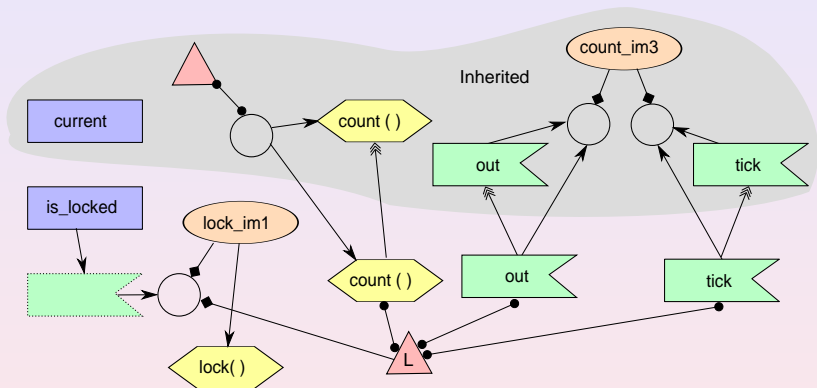
## Resolution of Alternatives

- instances select **valid subgraph** with
  - resolved alternatives
  - consistently allocated nodes  
(no conflicts, all dependencies. . .)

## Implementation Graph - Example



# Implementation Graph - Example



# Outline

- Objectives
- Combining Component Based Modelling and Object Orientation
  - Motivation
  - Type, Instance & Object
  - Communication & Connectivity
- Combining Inheritance and Behaviour Alternatives
  - Alternatives vs. Inheritance
  - Implementation graph
  - Example
- Summary

## Summary (I)

### Actors, Objects & Alternatives

- combination of component based modelling and object orientation
- alternative behaviours can be represented without interfering inheritance

### Reconfiguration & Mobility

- Everything can be changed at run-time:
  - create/modify instances and port bindings
  - create objects and bind them to different instances over the time

## Summary (II)

### Topics, not included in this talk

- modelling of **polymorphism**  
→ polymorphic instances
- modelling pointers and references
- multiple inheritance / parametrised types
- ...

Thank you!

Questions?

# Polymorphism

## Polymorphic Instances

- starting from an ordinary instance
- **allocate**
  - overridden **behaviour** (for the allocated services)
  - members
- for (maybe only a subset of) sub-types **explicitly**
- type (and therefore behaviour) resolution during **binding** of objects

# Pointers & References

## Instance is (statically) known

- local port can be bound as usual
- “position” of an object (within the instance) is sufficient
- even pointer arithmetic becomes possible

## Instance is not known

- local port could **not** be bound statically
- additional level of abstraction
- included in IR to increase flexibility

# Multiple Inheritance / Parametrised Types

## Multiple Inheritance

- implementation graph allows modelling of multiple inheritance
- resolution of multiple paths along inheritance hierarchy required (edges within graph)

## Parametrised Types

- type definition can include a set of parameters similar to C++ templates
- results in new types, depending on the values
- implementation graph concept needs to be adapted (parametrised nodes, out of scope here)