A Component Language for Structured Parallel Programming

Luc Bläser
Computer Systems Institute, ETH Zurich, Switzerland
blaeser@inf.ethz.ch

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Joint Modular Languages Conference (JMLC)
Oxford, England, September 2006

The Current State: Object-Orientation

Three fundamental problems:

- References
  - Arbitrary object interlinking => Unstructured dependencies
  - No hierarchical composition => Objects can not encapsulate (dynamic) structures of other objects
- Methods
  - Blocking procedure calls instead of real message passing
  - No arbitrarily long “state-full” client communication
- Inheritance
  - Groundless compulsion of hierarchisation and classification
  - Contradictory combination of polymorphism and code reuse

A New Approach

There is a better model for structured parallel programming:

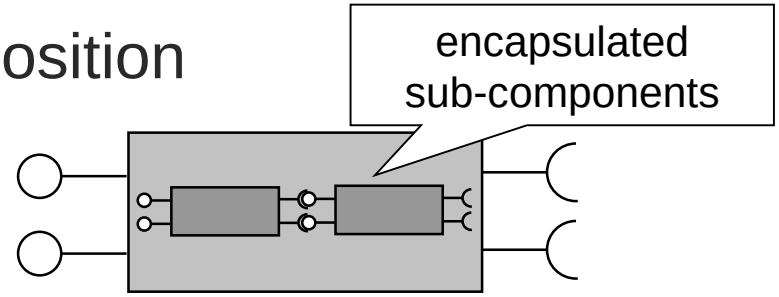
The Component Language

A new programming language with
an integrated general component notion

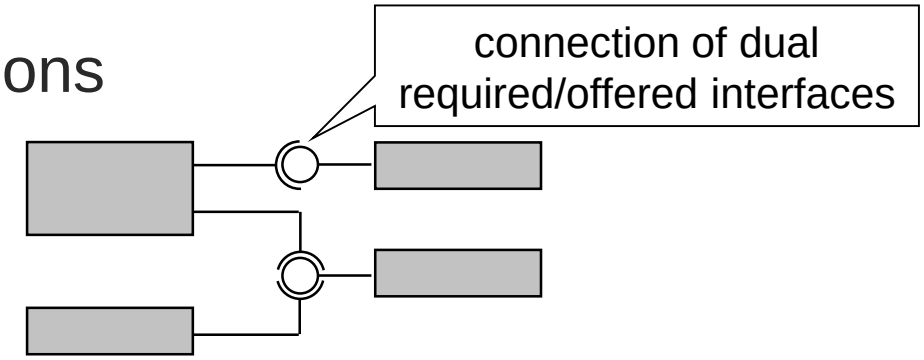
The Component Language

Highlights:

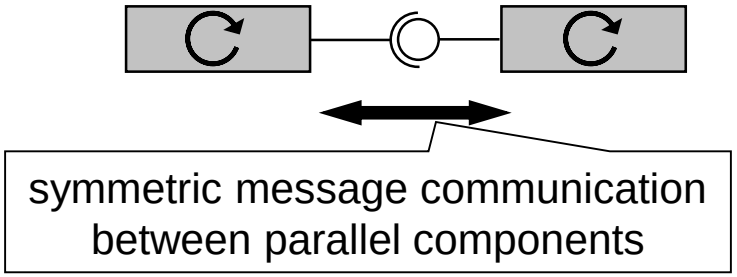
- Hierarchical composition



- Interface connections



- Communication-based interactions



Component Concept

- A **component** is a closed program unit at runtime that encapsulates state and behaviour
- External dependencies only allowed via explicit interfaces
- An **interface** represents an external facet of a component, an interaction point between a component and its external environment
- A component can **offer** own interfaces and **require** foreign interfaces of other external components
- Components (runtime instances) are created from a static template.

static template:

```
COMPONENT StandardHouse
  OFFERS Residence, ParkingPlace
  REQUIRES Water, Electricity;
  (* implementation *)
END StandardHouse;
```

runtime instance:


```
INTERFACE Residence;
  (* … *)
END Residence;
```

```
INTERFACE ParkingPlace; (* … *)
INTERFACE Water; (* … *)
INTERFACE Electricity; (* … *)
```

Component Instances

Multiple component instances can be created from the same template.

Declarations:

house1, house2: StandardHouse;

building: **ANY(**Residence, ParkingSpace **|** Water, Electricity**)**

building is a component of *any* template, which

- offers *at least* Residence and ParkingPlace
- requires *at most* Water and Electricity

townHouse: ANY(Residence | Electricity, Water, CentralHeating)

potential StandardHouse

oldHouse: ANY(Residence | Water)

no StandardHouse

Dynamic **collection** of component instances

- An **index** identifies an instance in the collection

house[number: INTEGER; street: TEXT]: StandardHouse

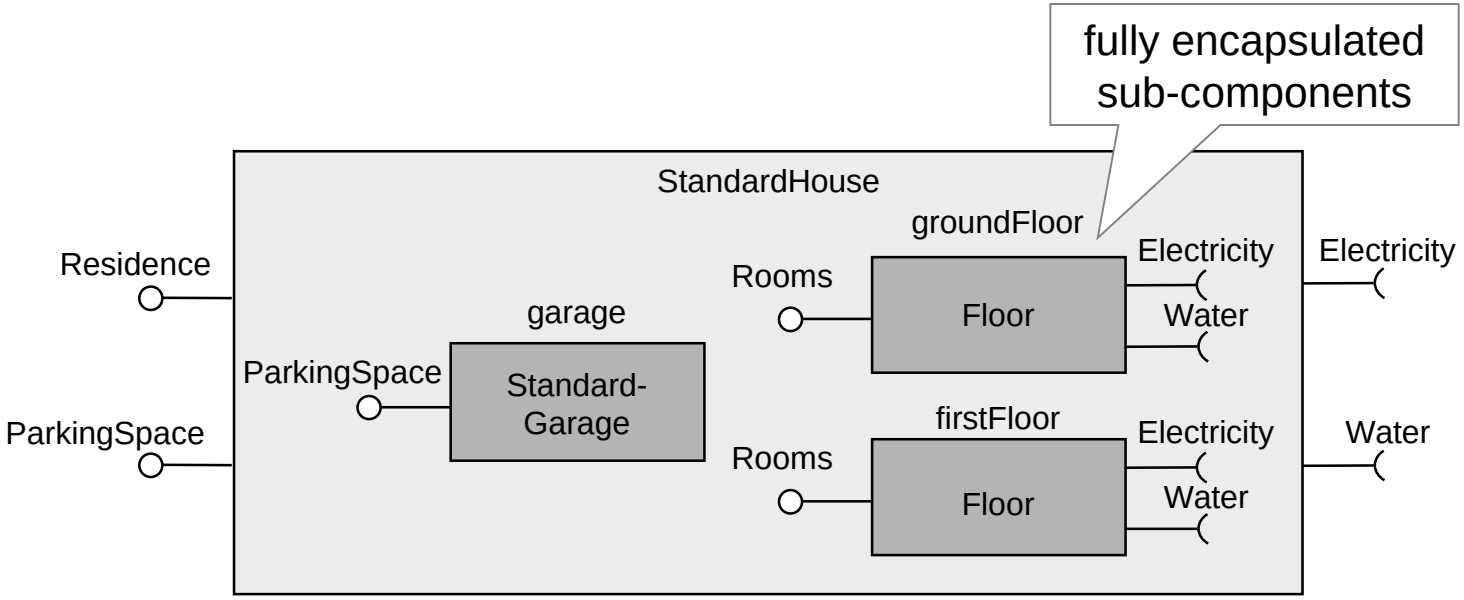
Possible instances:

house[12, “market street”] house[3, “main street”] …

Hierarchical Composition

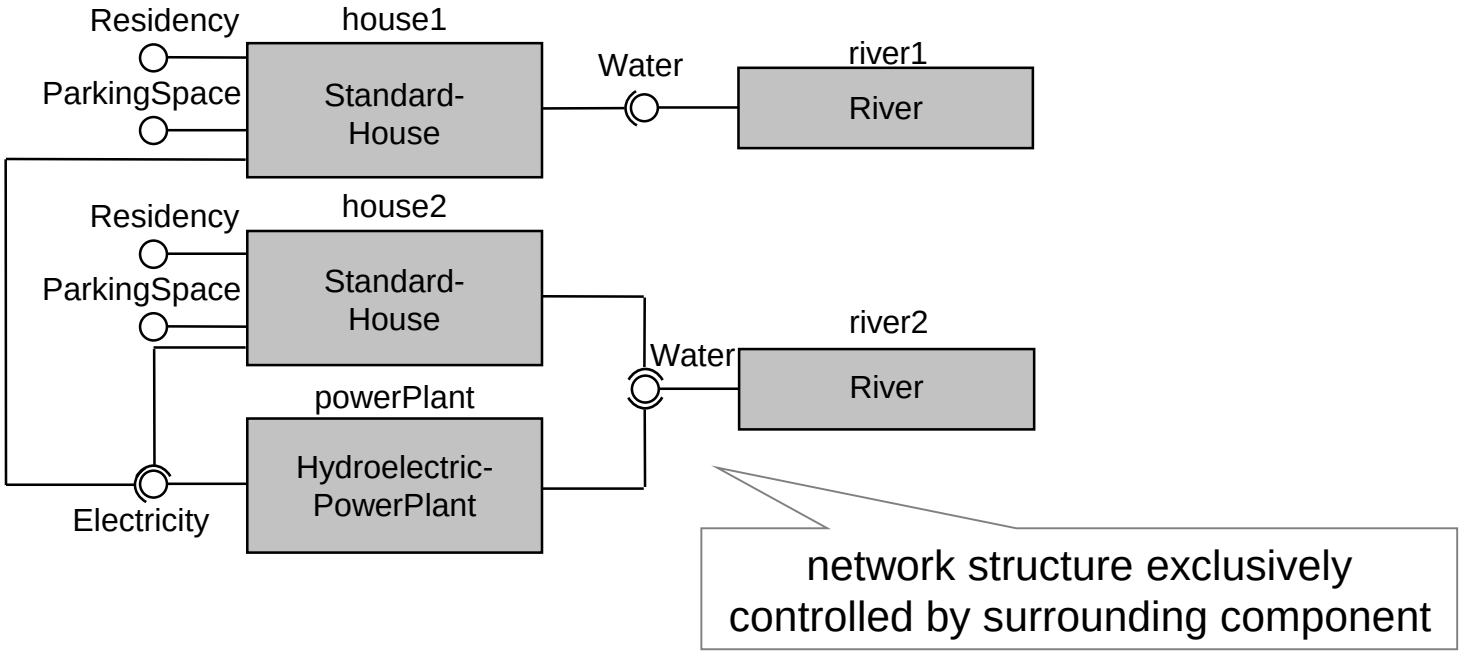
```
COMPONENT StandardHouse
  OFFERS Residence, ParkingSpace
  REQUIRES Electricity, Water;
  VARIABLE
    garage: StandardGarage;
    groundFloor, firstFloor: ANY(Rooms | Electricity, Water);
  BEGIN
    NEW(garage); NEW(groundFloor, Floor); NEW(firstFloor, Floor)
END StandardHouse;
```

variables as containers for components



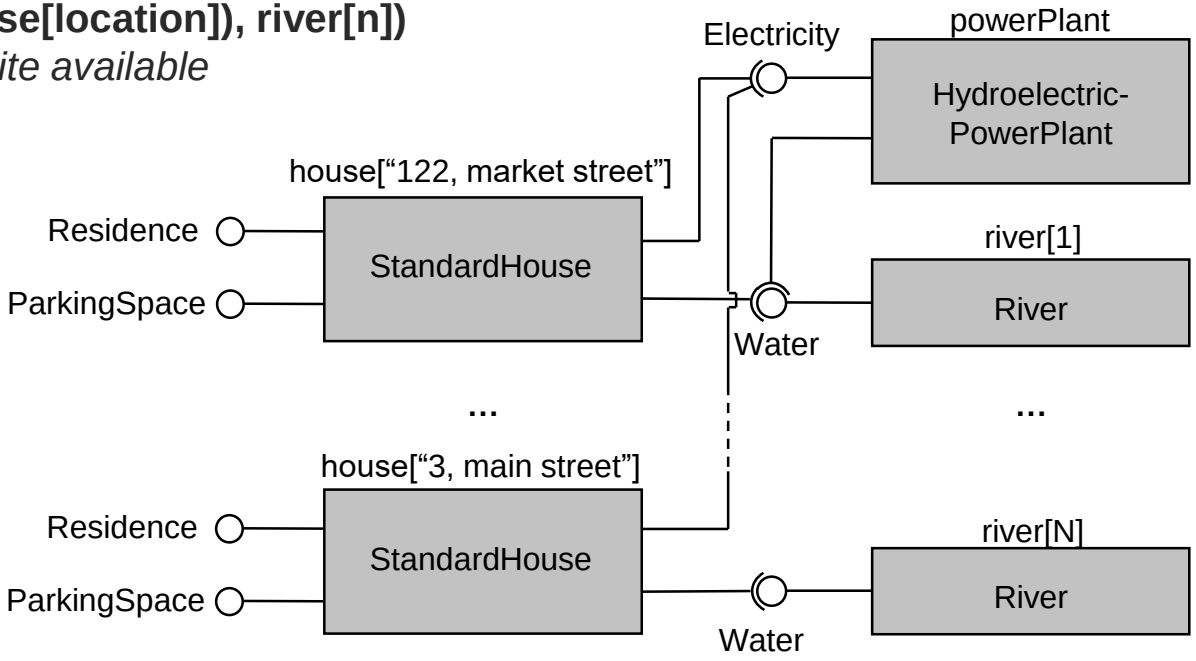
Component Networks

VARIABLE **house1**, **house2**: StandardHouse;
  powerPlant: HydroelectricPowerPlant;
  river1, **river2**: River;
BEGIN
  NEW(house1); NEW(house2); NEW(powerPlant); NEW(river1); NEW(river2);
  CONNECT(Water(house1), river1); CONNECT(Electricity(house1), powerPlant);
  CONNECT(Water(house2), river2); CONNECT(Electricity(house2), powerPlant);
  CONNECT(Water(powerPlant), river2)



Dynamic Network Construction

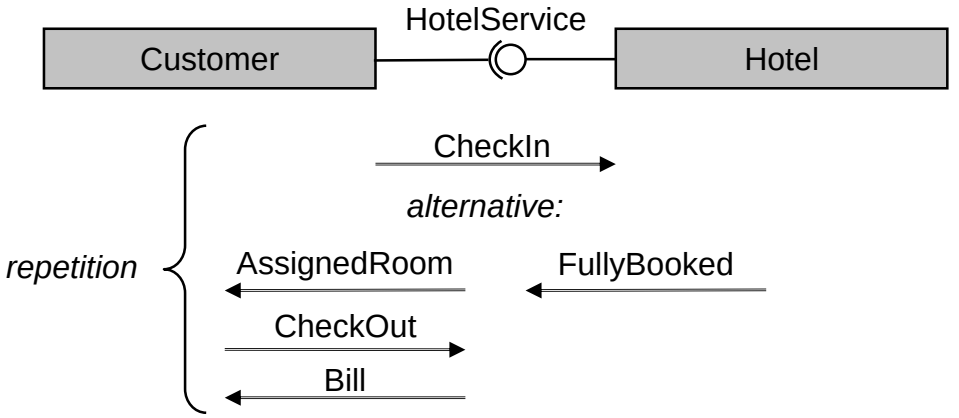
```
VARIABLE
  house[postalAddress: TEXT]: StandardHouse;
  powerPlant: HydroelectricPowerPlant;
  river[number: INTEGER]: River;
BEGIN
  FOR n := 1 TO N DO NEW(river[n]) END; (* N >= 1 *)
  NEW(powerPlant); CONNECT(Water(powerPlant), river[1]);
  REPEAT
    location := postal address of the new house;
    NEW(house[location]); CONNECT(Electricity(house[location]), powerPlant);
    n := number of nearest river;
    CONNECT(Water(house[location]), river[n])
  UNTIL no free building site available
```



Communication-Based Interactions

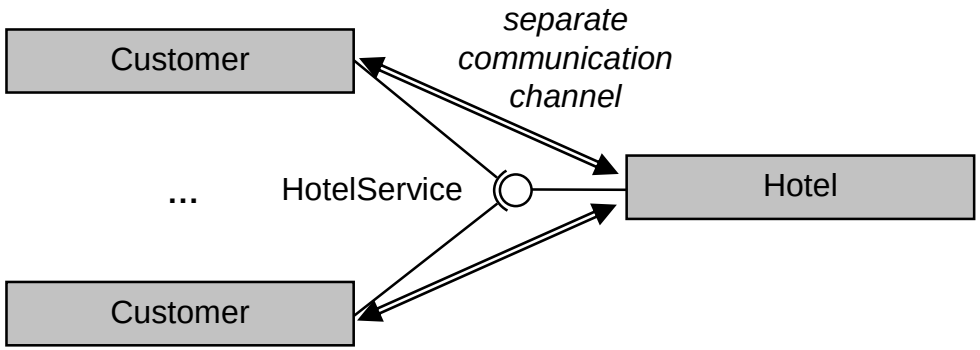
- An interface enables communications between a component and external instances
- A communication involves bidirectional (non-blocking) message exchange, specified by an EBNF protocol
- A message can carry values or components as content

```
INTERFACE HotelService;
 {
   IN CheckIn(name: TEXT)
   (
     OUT AssignedRoom(number: INTEGER)
     IN CheckOut OUT Bill
   |
     OUT FullyBooked
   )
 }
END HotelService;
```



Client-Individual Communications

- A component maintains a separate “state-full” communication with each interface client **individually**
- Multiple clients can use the same offered interface of a component in parallel




```
COMPONENT Customer
  REQUIRES HotelService;
BEGIN
  HotelService!CheckIn;
  IF HotelService?AssignedRoom THEN
    HotelService?AssignedRoom(n) (* … *)
  ELSE HotelService?FullyBooked
  END
END Customer;
```

send

receive guard

receive

```
COMPONENT Hotel
  OFFERS HotelService;

  IMPLEMENTATION HotelService;
  BEGIN
    WHILE ?CheckIn DO
      ?CheckIn(name); (* … *)
    END
  END HotelService;
END Hotel;
```

separate service process per client

Language Implementation

- Virtual machine
  - Intermediate code is generated by a front-end compiler
  - A back-end compiler in the VM generates direct machine code
  - System is currently based on ETH Bluebottle OS
- Structures automatically organised in heap behind the scenes
  - Automatic garbage collection is in fact no longer needed
  - Communication protocol is dynamically monitored
- Concurrency support by underlying Bluebottle / AOS
  - Direct context switches on wait dependencies
  - Still much potential for concurrency improvement in OS

Execution time in seconds:

Windows threads

analogous object-oriented applications

Test application	Component System	Active C#	WinAOS	Native Bluebottle
Producer-consumer	1.6	4.4	10	1.6
Small city simulation	2.9	360	24	2.7
Large city simulation	30	*out of memory*	*out of memory*	28

5000 components, 3000 processes

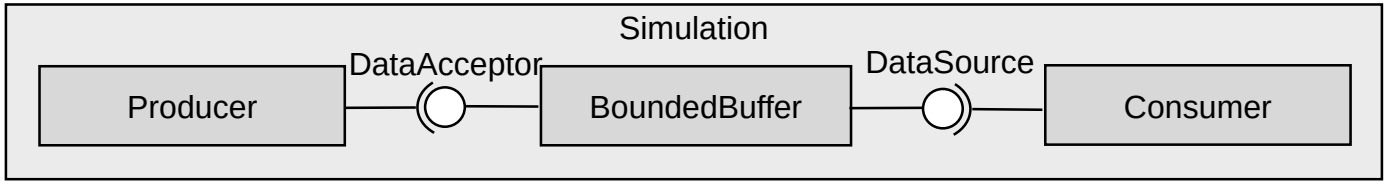
< 10% overhead compared to classical object-orientation

Conclusion

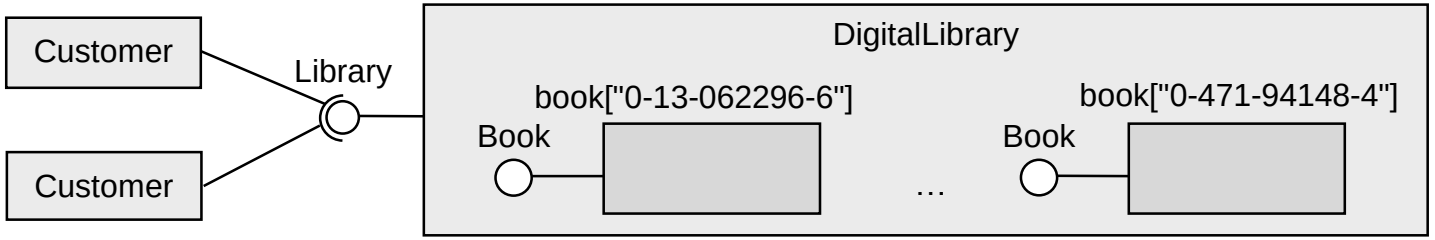
- A substantially new approach towards more structured construction of parallel software
  - Pointer-free structuring
  - Guaranteed hierarchical encapsulation
  - Concurrency with autonomously running components
  - General state-full interactions
  - Flexible polymorphism
- Language report and system download
  - http://www.jg.inf.ethz.ch/components
  - Don’t hesitate to ask for a personal demonstration of the system

Live Demonstration

1. Producer-Consumer



2. Digital Library



3. City Simulation