Eine neue komponentenorientierte Programmiersprache illustriert anhand einer Verkehrssimulation

Luc Bläser

Institut für Computersysteme
ETH Zürich

blaeser@inf.ethz.ch

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Seminar für Verkehrssimulation
TU Berlin, 25. Mai 2007

Projektübersicht

- Eine neue Programmiersprache
  - Hierarchische und klar strukturierte Komponenten
  - Durchgängige und sichere Nebenläufigkeit
- Verkehrssimulation als Studie hinsichtlich praktischer Einsetzbarkeit der Sprache
  - Simulation als geeigneter Testfall für eine neue Programmiersprache
    - SIMULA: Objekt-Orientierung ist ursprünglich für Simulation erfunden worden
  - Ziele:
    - Natürliche Simulationsbeschreibung
    - Genügende Ausdrucksfähigkeit
    - Vernünftige Ausführungsgeschwindigkeit

Motivation

Mängel in objektorientierten Sprachen

- Referenzen
  - Beliebiges Verlinken von Objekten => Unstrukturierte Abhängigkeiten
  - Fehlende hierarchische Komposition => Objekt kann keine anderen (dynamisch allozierten) Objekte kapseln
- Methoden
  - Blockierende Prozeduraufrufe statt echtem Nachrichtenaustausch
  - Nur einfache Input-Output Interaktion => komplexere zustandsbehaftete Interaktion nicht unterstützt
- Vererbung
  - Unbegründeter Zwang zur Hierarchisierung auf Typebene
  - Unpassende Kombination von Polymorphismus und Code-Wiederverwendung

Die Komponentensprache

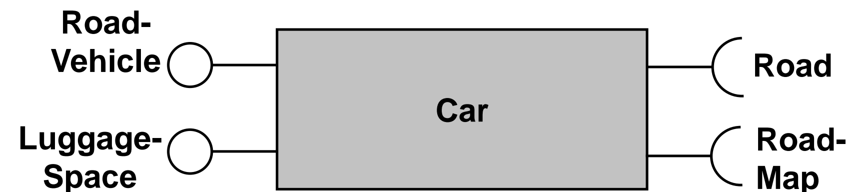
Komponentenbegriff

- Abstraktionseinheit zur Laufzeit
  - Subjekte (z.B. “Person”), aktive Objekte (z.B. “Fahrzeug”), passive Objekte (z.B. “Strasse”), abstrakte Begriffe (z.B. “Route”)
- Strikte Kapselung
  - Äußere Abhängigkeiten nur via expliziten Schnittstellen zugelassen
- Komponente kann Schnittstellen anbieten und erfordern
  - angebotene Schnittstellen repräsentieren eigene Facetten der Komponente
  - erforderte Schnittstellen sind von anderen Komponenten anzubieten

Statische Schablone

COMPONENT **Car**
  OFFERS **RoadVehicle**, **LuggageSpace**
  REQUIRES **Road, RoadMap**;
  (* implementation *)
END Car;

Laufzeitexemplar



Hierarchische Komposition

- Jede Komponente kann beliebig viele Unterkomponenten beinhalten

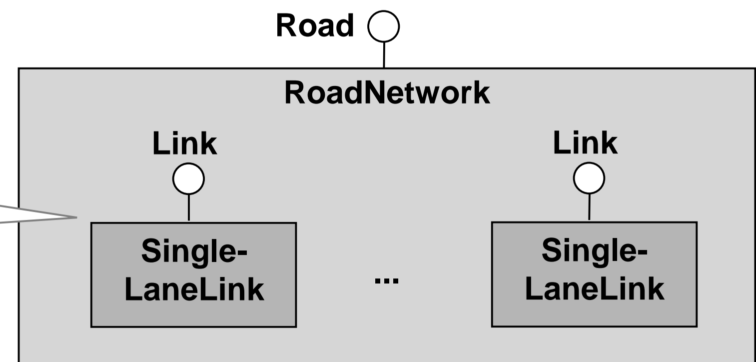
dynamische Kollektion von Unterkomponenten

```
COMPONENT RoadNetwork OFFERS Road;
VARIABLE link[id: INTEGER]: ANY(Link);
BEGIN
  FOREACH link id in simulation file DO
    NEW(link[id], SingleLaneLink)
  END
END RoadNetwork;
```

beliebiger Typ mit postulierten Schnittstellen

Komponenten-Schablone

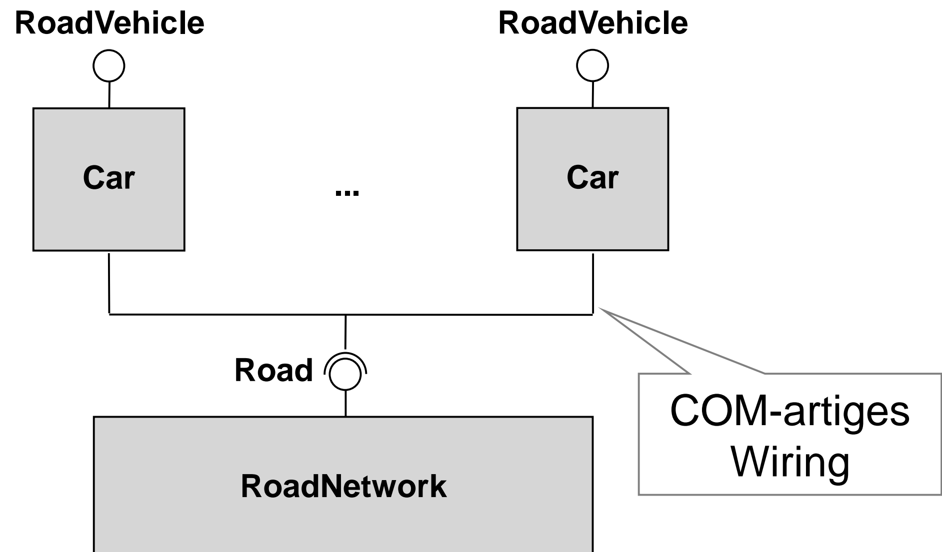
Eingekapselte Unterkomponenten



Schnittstellenverbindungen

- Jede erforderte Schnittstelle kann mit einer angebotenen Schnittstelle des gleichen Namens verbunden werden

```
COMPONENT TrafficSimulation;
VARIABLE
  car[id: INTEGER]: Car;
  road: RoadNetwork
BEGIN
  NEW(road);
  FOREACH car id in simulation file DO
    NEW(car[id]);
    CONNECT(Road(car[id]), road)
  END
END TrafficSimulation;
```

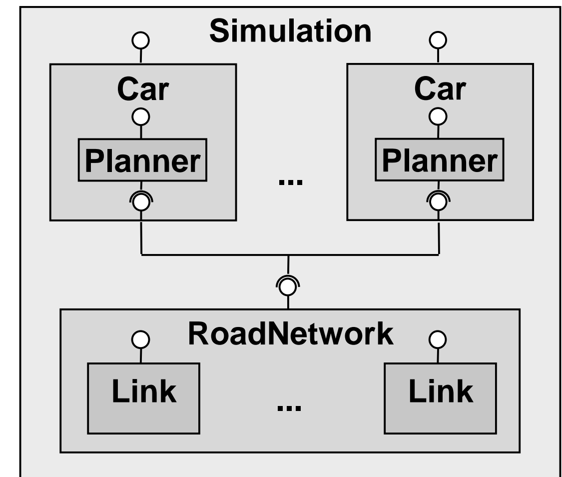


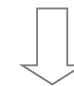
Pointer-Freiheit

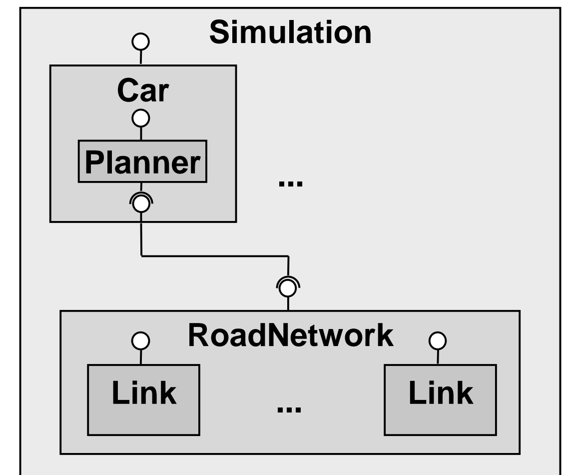
- Verbindungen nur von der umgebenden Komponente gesetzt
- Ausgehende und eingehende Verbindungspunkte für jede Komponente klar deklariert

Speicherverwaltung

- Hierarchie von Komponentennetzwerken
  - Netzwerkstruktur ausschließlich von der umgebenden Komponente kontrolliert
- Hierarchische Abhängigkeit
  - Löschen einer Komponente => Automatisches Löschen der Unterkomponenten
  - Explizites Löschen einer Komponente => Verbindungen mit deren Schnittstellen werden getrennt
  - Speichersicherheit ohne Garbage Collector

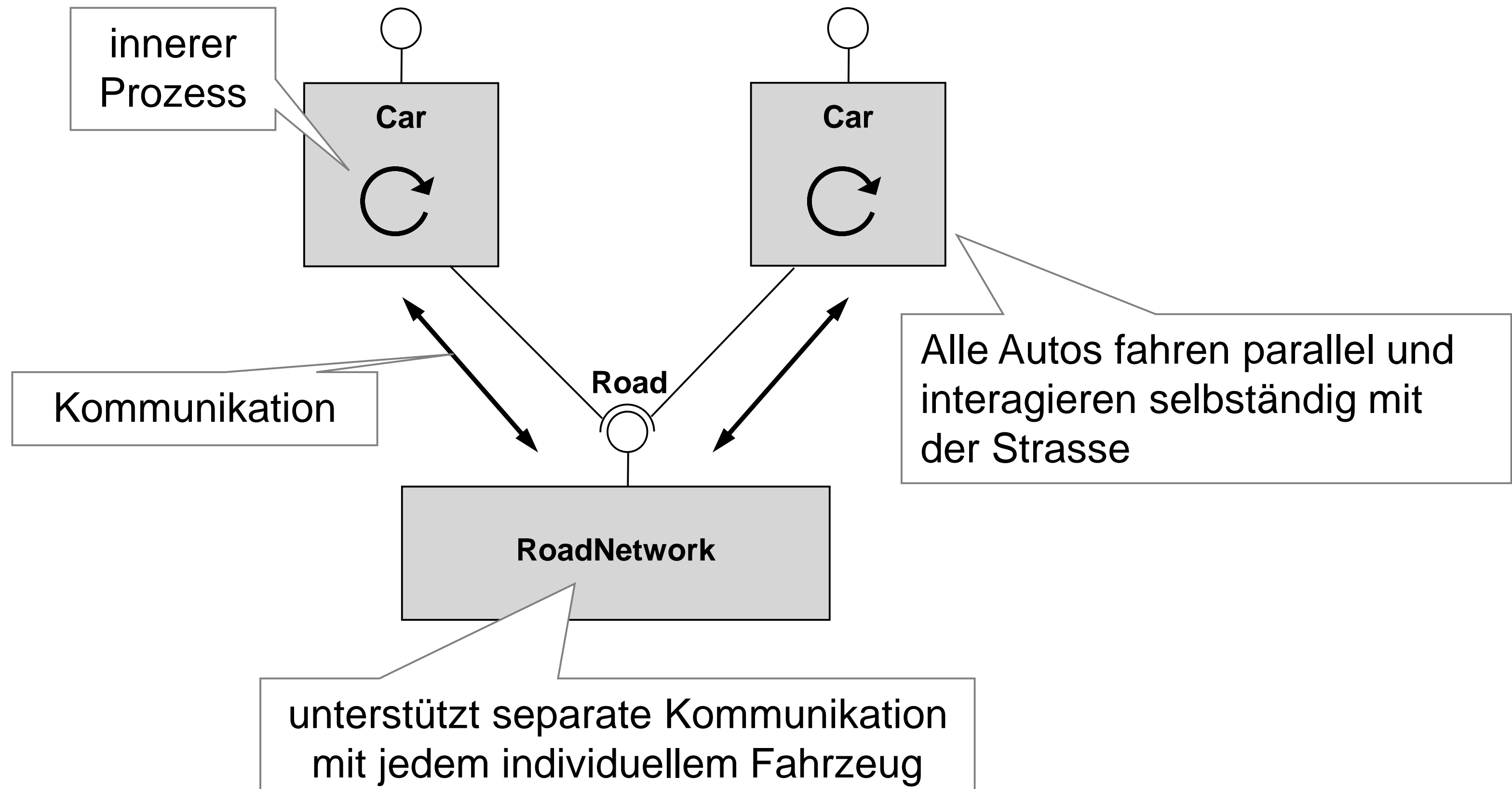






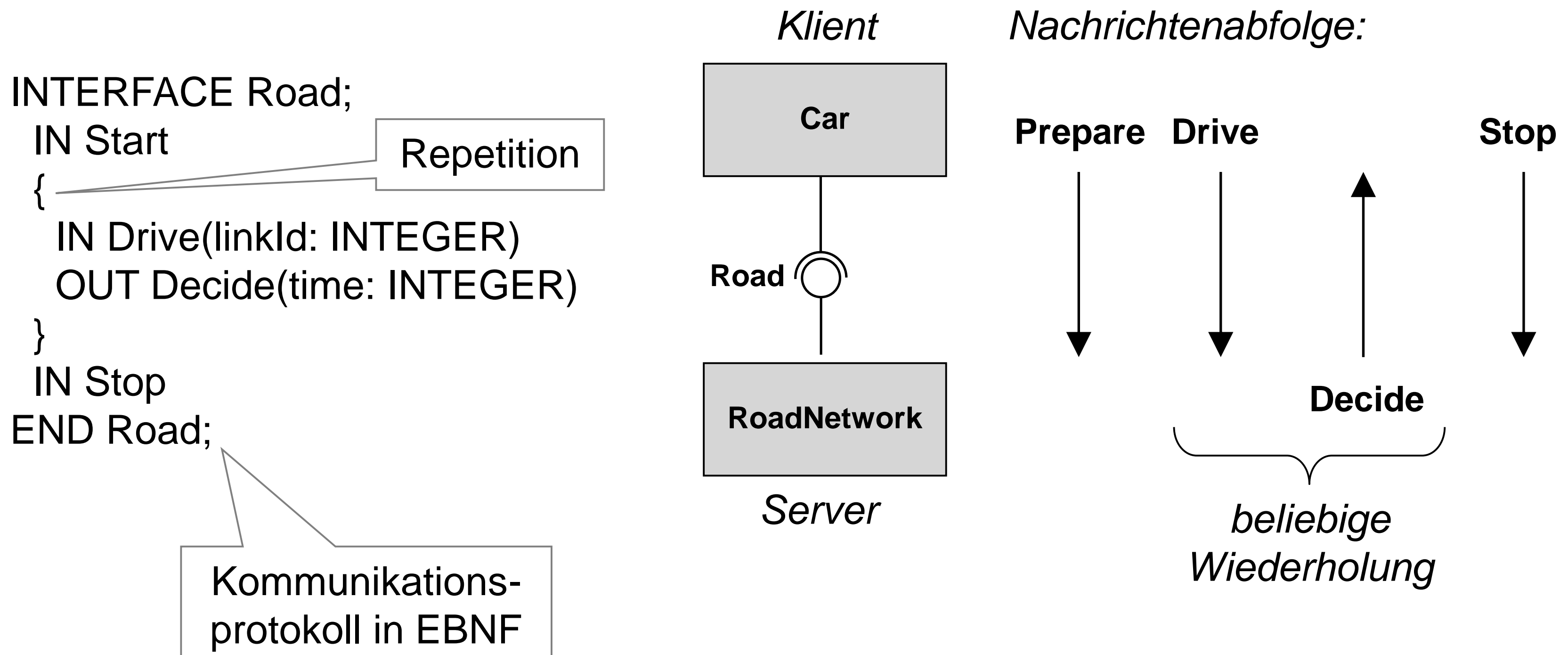
Nebenläufigkeit und Interaktionen

- Jede Komponente führt ihre eigenen inneren Prozesse aus
- Komponenten interagieren mittels Kommunikation über Schnittsstellen

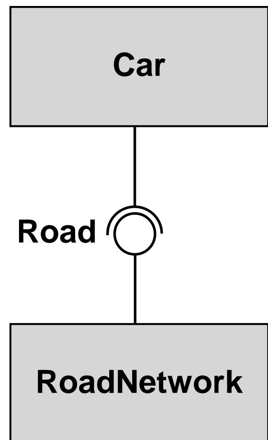


Kommunikation

- Separate Kommunikation zwischen jedem Klienten und Server
- Senden und Empfangen von Nachrichten gemäß eines Protokolls



Implementierung der Komponenten




```
COMPONENT Car (* … *) REQUIRES Road;
BEGIN
  Road!Start;
  WHILE target not reached DO
    Road!Drive(nextLinkId); Road?Decide(time)
  END;
  Road!Stop
END Car;
```

sende Nachricht

empfange Nachricht

```
COMPONENT RoadNetwork OFFERS Road;
  IMPLEMENTATION Road;
  BEGIN
    ?Start;
    WHILE ?Drive DO
      ?Drive(linkId); (* drive *) !Decide(now)
    END;
    ?Stop
  END Road;
END RoadNetwork;
```

separater Service-prozess pro Klient

Empfangstest

Laufzeitsystem

Kleines Betriebssystem

- Leichtgewichtige Prozesse
  - Mikro-Stacks: Größe kann dynamisch wachsen & schrumpfen
  - Ermöglicht hohe Anzahl an Prozessen
- Schnelle Kontext-Wechsel
  - Direkte synchrone Kontext-Wechsel
  - Schnelle Preemption mittels Code-Instrumentierung
- Ausgeklügelter Scheduler
  - Prozesse werden nur parallel ausgeführt, falls unabhängig
  - Vermeidung teurer Synchronisationskosten
  - Abhängigkeiten werden aufgrund der Schnittstellen-Verbindungen bestimmt
- Sichere und effiziente Speicherverwaltung
  - Garbage Collector nicht benötigt
  - Virtuelle Speicherverwaltung nicht benötigt

Verkehrssimulationsstudie

Entwicklung einer Verkehrssimulation in der Komponentensprache

- Selbstaktive Fahrzeuge
  - Alle Fahrzeuge fahren selbständig und parallel
  - Keine globale Programmschleife, die alle Fahrzeuge zentral steuert und bewegt
- Virtuelle Zeit
  - Alle Fahrzeuge laufen mit synchroner virtueller Zeit ab
  - Virtuelle Zeit entspricht der Zeit in der simulierten Welt
- Individuelles Planen und Lernen
  - Fahrer planen Reise, Routen und Abfahrtszeiten individuell
  - Jeder Fahrer berücksichtigt dabei sein eigenes Wissen aus früheren Reisen
  - Keine globale Instanz zur Planung und Ereignisaufzeichnung

Virtuelle Zeit

- Jede Komponente besitzt ihre eigene virtuelle Zeitachse
  - Berechnungsoperationen brauchen keine virtuelle Zeit
  - Fortschreiten der virtuellen Zeit mittels PASSIVATE und AWAIT
- PASSIVATE(duration)
  - Prozess wartet, bis eine virtuelle Zeitperiode verstrichen ist
  - Fährt fort, wenn alle Prozesse derselben Zeitzone auf mindestens diese virtuelle Endzeit warten
- AWAIT(condition)
  - Prozess wartet auf bestimmte Bedingung
  - Während des Wartens kann virtuelle Zeit verstreichen
- Hierarchische Synchronisierung der virtuellen Zeit
  - Komponente kann zeitsynchrone Unterkomponenten deklarieren
  - Zeitwahrnehmung in Unterkomponente bleibt damit unverändert

Straßenstücke haben gleiche virtuelle Zeit wie Straßennetz

```
COMPONENT RoadNetwork;
  VARIABLE link[id: INTEGER]: ANY(Link) {TIMESYNCHRONOUS};
END RoadNetwork
```

Ein einfaches Straßenstück

```
COMPONENT SingleLaneLink OFFERS Link;
  VARIABLE occupied[cell: INTEGER]: BOOLEAN;

  IMPLEMENTATION Link;
  VARIABLE cell: INTEGER;
  BEGIN {EXCLUSIVE}
    AWAIT(~occupied[0]); occupied[0] := TRUE;
    !Entered; PASSIVATE(1); cell := 0;
    WHILE cell < length DO
      AWAIT(~occupied[cell + 1]);
      occupied[cell + 1] := TRUE;
      occupied[cell] := FALSE;
      INC(cell);
      PASSIVATE(1)
    END;
    !EndReached;
    occupied[exit] := FALSE
  END Link;
END SingleLaneLink;
```

Zellulärer Automat

Autonomer Fahrprozess eines Fahrzeugs

Monitorsperre

warte auf nächste freie Zelle

warte eine virtuelle Sekunde

AWAIT & PASSIVATE lösen temporär die Monitorsperre

Ausführungsgeschwindigkeit

skaliert mit Anzahl nebenläufiger Fahrzeuge & Grad der Staus

skaliert mit Länge des Straßennetzes & (späteste Ankunft – früheste Abfahrt)

Ausführungszeit in Sekunden	**Komponenten-basierte Simulation**	**Analoge thread-basierte C# Simulation**	**Klassische sequenzielle C++ Simulation**
1,000 cars	0.778	44.6	28.3
10,000 cars	6.82	14500.0	31.6

Intel P4, 2.4GHz, hyper-threading, 1 GB Hauptspeicher

nahe an Sättigung des “corridor” Straßennetzes

Gewünschte Ankunftszeiten:
- 8:00 für 60% der Autos
- Rest verteilt über den Tag

Schlussfolgerungen

- Natürlichere Simulationsbeschreibung
  - Selbständiges Fahrverhalten pro Fahrzeug
  - Fahrzeuge fahren parallel
  - Fahrverhalten basierend auf virtueller Zeit
  - Individuelles Planen und Lernen
  - Nicht mehr benötigte Konstrukte:
    - Explizite Park- & Warteschlangen
    - Globale Programmschleife über Zeit und alle Straßenzellen sowie zentral gesteuerte Fahrzeugbewegung
    - Zentralisierte Ereignisauszeichnung und Planung, Event-Dateien
- Flexible Programmierung
  - Alle Komponenten konnten einfach und kompakt programmiert werden
  - Neue Strukturen konnten Referenzen (Pointers) nahtlos ersetzen
- Gute Ausführungsgeschwindigkeit
  - Schneller als nebenläufige und (in unseren Fällen) sequenzielle Simulationen
  - skaliert anders als klassisch sequenzielle Simulationsmodelle