# Kapitel 9

# Fallstudie:
# Koordination verteilter Umweltsimulationen

Prof. Dr. Rolf Hennicker

14.07.2016

# 9.1 Verteilte Umweltsimulationen in GLOWA-Danube

- GLOWA-Danube war ein vom BMBF gefördertes Projekt innerhalb der Initiative GLOWA (Globaler Wandel des Wasserkreislaufs, 2000-2010)
- Projekttitel: "Integrative Techniques, Scenarios and Strategies for the Future of Water in the Upper Danube Basin"
- Projektpartner aus mehr als einem Dutzend verschiedener Fachgebiete aus Naturwissenschaften und Sozialwissenschaften
- Projektziel: Entwicklung des verteilten Systems DANUBIA
  - für integrative Simulationen gekoppelter Simulationsmodelle,
  - zur Analyse von transdisziplinären Effekten wechselseitig abhängiger Prozesse,
  - zur Entscheidungsunterstützung auf der Grundlage wasserbezogener Szenarien.

# GLOWA-Danube Projekt

**Natural Sciences**

- Hydrology
- Plant Ecology
- Glaciology
- Meteorology
- Groundwater
- Surface Water

**Social Sciences**

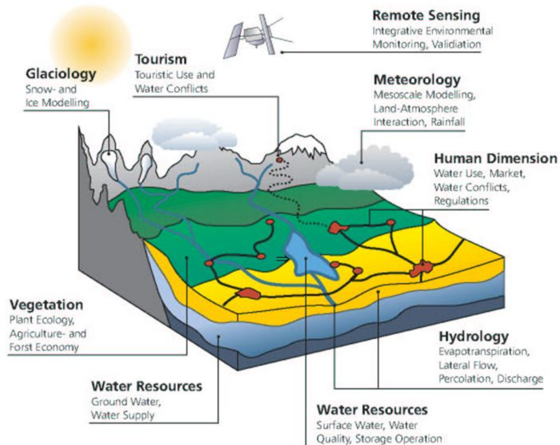
- Environmental Psychology
- Environmental Economy
- Tourism Research
- Water Supply
- Agricultural Economics

+ Informatics

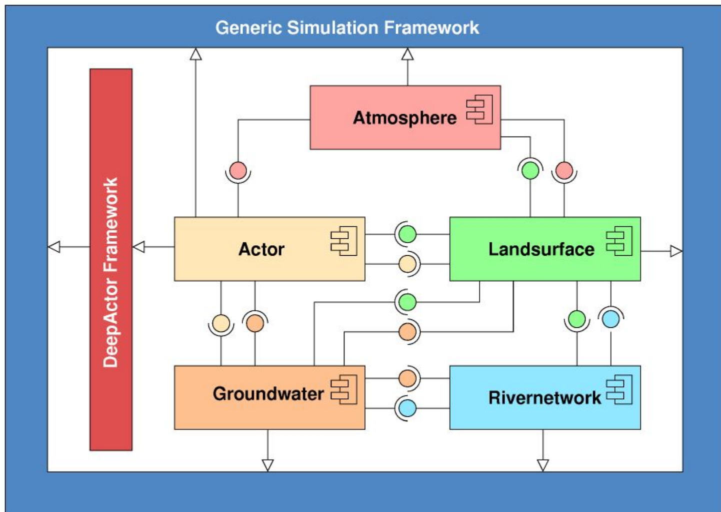
**Upper Danube Basin:**

- Area: 77.000 km²
- Population: 8.2 Mio.
- Elevation Gradient: 3300 m

# Wechselseitig anhängige Prozesse in Natur und Gesellschaft



# Das DANUBIA-System



# 9.2 Simulationsmodelle und integrative Simulationen

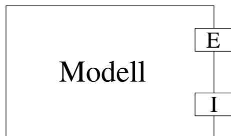
## Simulationszeit

- Ein Simulationsmodell simuliert einen physikalischen oder sozialen Prozess über eine bestimmte Zeitspanne (*Simulationszeit*).
- Die Simulationszeit ist endlich, d.h. es gibt einen Anfangs- und einen Endzeitpunkt einer Simulation.
- Ein Simulationsmodell hat einen individuellen lokalen *Zeitschritt*, der den Abstand zweier aufeinanderfolgender Simulationszeitpunkte bestimmt (z.B. stündliche Temperaturwerte, Wasserverbrauch pro Monat).
- Wir nehmen an, dass der Zeitschritt eines Modells während der gesamten Simulation gleich bleibt.

## Datenaustausch

Ein Simulationsmodell

- stellt über Export-Interfaces Daten (für andere Modelle) zur Verfügung,
- holt über Import-Interfaces Daten (von anderen Modellen), die es für die eigenen Berechnungen benötigt.



## Lebenszyklus eines Simulationsmodells

- Nachdem ein Modell gestartet wurde, stellt es zunächst bestimmte Anfangswerte über seine Export-Interfaces zur Verfügung.
- Bis zum Simulationsende führt es dann zyklisch, entsprechend des lokalen Modellzeitschritts, folgende Aktivitäten durch:
  1. Holen der (für die eigenen Berechnungen) benötigten Daten über die Import-Interfaces.
  2. Berechnung neuer Simulationsdaten, die zum nächsten Simulationszeitpunkt gültig sind.
  3. Bereitstellen der neu berechneten Werte über die Export-Interfaces.

# Modellierung von Simulationsmodellen in FSP

```
//Simulationszeit
const SimStart = 0
const SimEnd = 6
range Time = SimStart..SimEnd

//Berechnungszyklus
MODEL(Step=1) = (start → INIT),
INIT = (prov[SimStart] → M[SimStart]),
M[t:Time] = if (t+Step <= SimEnd)
            then (get[t] → compute[t] → prov[t+Step] → M[t+Step])
            else STOP.
```

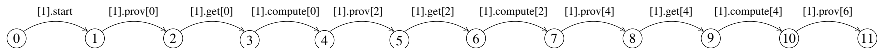
**Beachte:**

- prov[x] repräsentiert das Bereitstellen von Export-Daten, die zum Zeitpunkt x gültig sind.
- get[x] repräsentiert das Holen von Import-Daten, die zum Zeitpunkt x gültig sind.
- compute[x] repräsentiert die Berechnung von neuen Daten, auf der Grundlage der zum Zeitpunkt x gültigen Import-Daten.

# Modellierung von Instanzen von Simulationsmodellen

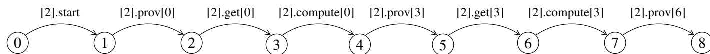
Simulationsmodell mit Nummer 1 und Zeitschritt 2:

[1]:MODEL(2)

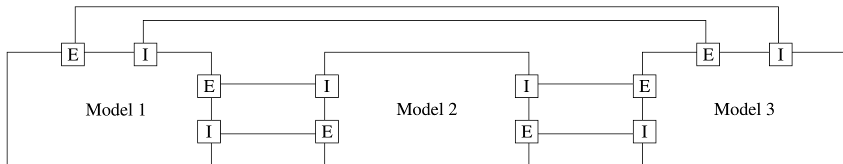



Simulationsmodell mit Nummer 2 und Zeitschritt 3:

[2]:MODEL(3)



# Integrative Simulationen




- In einer integrativen Simulation arbeiten verschiedene Simulationsmodelle zusammen, indem sie (zur Laufzeit) gegenseitig Daten austauschen.
- Bei einer integrativen Simulation haben alle Modelle denselben Simulationszeitraum, jedoch (i.a.) verschiedene lokale Zeitschritte.
- Die Modelle müssen geeignet koordiniert werden.

# Parallele Komposition von zwei Simulationsmodellen

const nrModels = 2      range Models = 1..nrModels

||SYS = ([1]:MODEL(2) || [2]:MODEL(3)) / {start/[Models].start}.

**Mögliche Abläufe:**

(1) *Zu alte Daten:*

start → [2].prov[0] → [1].prov[0] → [1].get[0] → [1].compute[0] → [1].prov[2] → [1].get[2] → [1].compute[2] → [1].prov[4] → [1].get[4] →

Modell 1 holt Daten, während Modell 2 noch nicht die aktuellsten Daten bereitgestellt hat.

(2) *Überschriebene Daten:*

start → [2].prov[0] → [1].prov[0] → [2].get[0] → [2].compute[0] → [2].prov[3] → [1].get[0] →

Modell 2 liefert Daten, während Modell 1 die zum gewünschten Zeitpunkt gültigen Daten noch nicht geholt hat.

# 9.3 Formalisierung des Koordinatenproblems

- Betrachte ein Simulationsmodell unter verschiedenen Rollen, als Benutzer (“User“) oder als Lieferant (“Provider“) von Daten.
- Betrachte die Anforderungen paarweise für einen Benutzer und einen Lieferanten.

**Anforderungen zur Gültigkeit von Daten**

(1) **Für den Benutzer (U):**
U holt nur dann Daten, die zu einem Zeitpunkt $t_U$ gültig sein sollen, wenn gilt: Die nächsten Daten, die P liefert, sind gültig zu einem Zeitpunkt $t_P$ mit $t_U < t_P$.

*Wenn die nächsten Daten, die P liefert, zu einem Zeitpunkt in der Zukunft gültig sind, können die gerade vorhanden nicht veraltet sein und U kann sie jetzt holen.*

(2) **Für den Lieferanten (P):**
P liefert nur dann Daten, die zu einem Zeitpunkt $t_P$ gültig sind, wenn gilt: Die nächsten Daten, die U holt, sollen gültig sein zu einem Zeitpunkt $t_U$ mit $t_U >= t_P$.

*Wenn die nächsten Daten, die U holt, zu einem Zeitpunkt ab jetzt gültig sein sollen, kann P Daten jetzt liefern ohne noch benötigte Daten zu überschreiben.*

## Modellierung der legalen Abläufe durch Property-Prozesse

```
property VALIDDATA(User=1,StepUser=1,Prov=1,StepProv=1) =
  VD[SimStart][SimStart],

VD[nextGet:Time][nextProv:Time] =
 (when (nextGet<nextProv)
       [User].get[nextGet] -> VD[nextGet+StepUser][nextProv]
 |when (nextGet>=nextProv)
       [Prov].prov[nextProv] -> VD[nextGet][nextProv+StepProv]).
```

Instanziierung, z.B. `VALIDDATA(1,2,2,3)`.

Zur Vereinfachung ist oben das Simulationsende nicht berücksichtigt.

*Mit Simulationsende:*

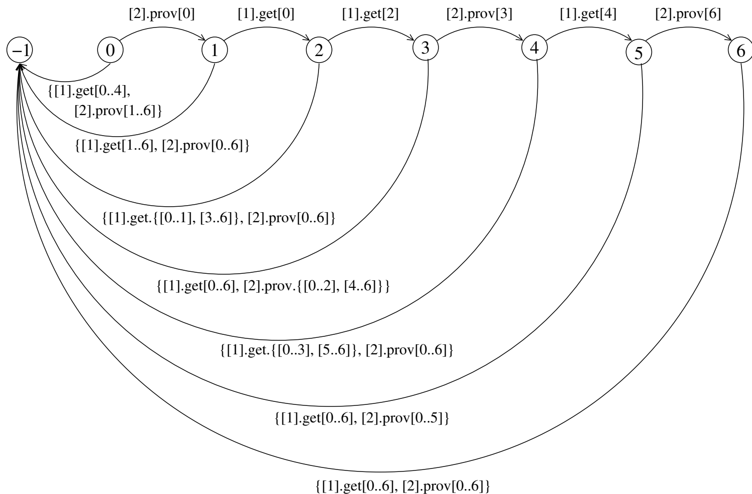
```
property VALIDDATA(User=1,StepUser=1,Prov=1,StepProv=1) =
  VD[SimStart][SimStart],

VD[nextGet:Time][nextProv:Time] =
 (when (nextGet<nextProv)
    [User].get[nextGet] ->
        if (nextGet+2*StepUser<=SimEnd)
        then VD[nextGet+StepUser][nextProv] else PROVFINISH[nextProv]
 |when (nextGet>=nextProv)
    [Prov].prov[nextProv] ->
           if (nextProv+StepProv<=SimEnd)
           then VD[nextGet][nextProv+StepProv] else USERFINISH[nextGet]

PROVFINISH[nextProv:Time] =
 ([Prov].prov[nextProv] -> if (nextProv+StepProv<=SimEnd)
                           then PROVFINISH[nextProv+StepProv]),

USERFINISH[nextGet:Time] =
 ([User].get[nextGet] ->  if (nextGet+2*StepUser<=SimEnd)
                          then USERFINISH[nextGet+StepUser]).
```

# LTS des Property-Prozesses `VALIDDATA(1,2,2,3)`, d.h. User = Modell 1 mit Zeitschritt 2 und Provider = Modell 2 mit Zeitschritt 3.



# 9.4 Systementwurf

**Idee:**

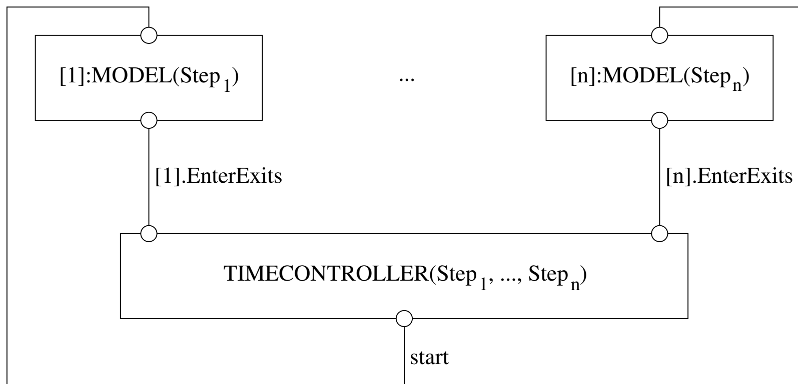
Verwende einen globalen Timecontroller zur Koordination aller an einer verteilten Simulation teilnehmenden Simulationsmodelle.

**Ziel:**

Konstruiere einen FSP-Prozess TIMECONTROLLER, so dass für n Modelle das folgende System alle Sicherheitseigenschaften erfüllt:

$$
\begin{aligned}
\|\text{SYS}=(&[1]\text{:MODEL}(\text{Step}_1) \parallel \ldots \parallel [n]\text{:MODEL}(\text{Step}_n)\\
&\parallel \text{TIMECONTROLLER}(\text{Step}_1,\ldots,\text{Step}_n))\\
&/\{\text{start}/[\text{Models}].\text{start}\}
\end{aligned}
$$

# Strukturdiagramm




wobei
set EnterExits = {{enterGet,exitGet,enterProv,exitProv}[Time]}

# Konstruktion des Timecontroller-Modells

- Der Timecontroller ist ein Monitor, der die Operationen enterGet, exitGet, enterProv und exitProv anbietet.
- enter-Aktionen werden bewacht durch eine Bedingung, die die Anforderungen bzgl. der Gültigkeit von Daten gewährleisten soll.
- Die Bedingungen sind abhängig vom Monitorzustand, der durch Indexvariablen lokaler Prozesse modelliert wird.
- Der Monitor merkt sich für jedes an einer verteilten Simulation teilnehmende Modell $m \in$ Models den Zeitpunkt, an dem das Modell zum nächsten Mal Daten holt (nextGetm) und an dem es zum nächsten Mal Daten liefert (nextProvm).
- Die Zeitschritte der an einer verteilten Simulation teilnehmenden Modelle werden dem Timecontroller über Prozessparameter bekannt gegeben, z.B. Timecontroller für zwei Modelle mit den Zeitschritten 2 und 3: TIMECONTROLLER(2,3).

# Modell des Timecontrollers (für 2 Simulationsmodelle)

```
TIMECONTROLLER(Step1=1,Step2=1) =
  (start -> TC[SimStart][SimStart][SimStart][SimStart]),

TC[nextGet1:Time][nextProv1:Time][nextGet2:Time][nextProv2:Time] =
  (dummy[t:Time] ->
    //enterGet
    (when (t<nextProv1 && t<nextProv2)
           [Models].enterGet[t] ->
           TC[nextGet1][nextProv1][nextGet2][nextProv2]
    //exitGet
    |[1].exitGet[t] -> TC[t+Step1][nextProv1][nextGet2][nextProv2]
    |[2].exitGet[t] -> TC[nextGet1][nextProv1][t+Step2][nextProv2]
```

```
  //enterProv
  |when (nextGet1>=t && nextGet2>=t)
        [Models].enterProv[t] ->
        TC[nextGet1][nextProv1][nextGet2][nextProv2]
  //exitProv
  |[1].exitProv[t] ->
        if (t+Step1<=SimEnd)
        then TC[nextGet1][t+Step1][nextGet2][nextProv2]
        else TC[SimStart][SimStart][SimStart][SimStart]
  |[2].exitProv[t] ->
        if (t+Step2<=SimEnd)
        then TC[nextGet1][nextProv1][nextGet2][t+Step2]
        else TC[SimStart][SimStart][SimStart][SimStart]
  |dummy[t] -> TC[nextGet1][nextProv1][nextGet2][nextProv2])
)\{dummy[Time]}.
```

## Integration von enter- exit-Aktionen in ein Modell

MODEL(Step=1) = (start → INIT),
INIT = (enterProv[SimStart]→prov[SimStart]→exitProv[SimStart]→M[SimStart]),
M[t:Time] = if (t+Step <= SimEnd)
    then (enterGet[t] → get[t] → exitGet[t] →
        compute[t] →
        enterProv[t+Step] → prov[t+Step] → exitProv[t+Step] → M[t+Step])
    else STOP + EnterExits.

## Modell einer verteilten Simulation (mit 2 Simulationsmodellen)

const StepModel1 = 2

const StepModel2 = 3

$\|$SYS = ([1]:MODEL(StepModel1) $\|$ [2]:MODEL(StepModel2)
$\|$ TIMECONTROLLER(StepModel1,StepModel2)) / {start/[Models].start}.

## Nachweis der Sicherheitseigenschaften

$\|$CHECK_VALIDDATA_USER1_PROV2 =
(SYS $\|$ VALIDDATA(1, StepModel1, 2, StepModel2)).

Fehlerzustand nicht erreichbar.

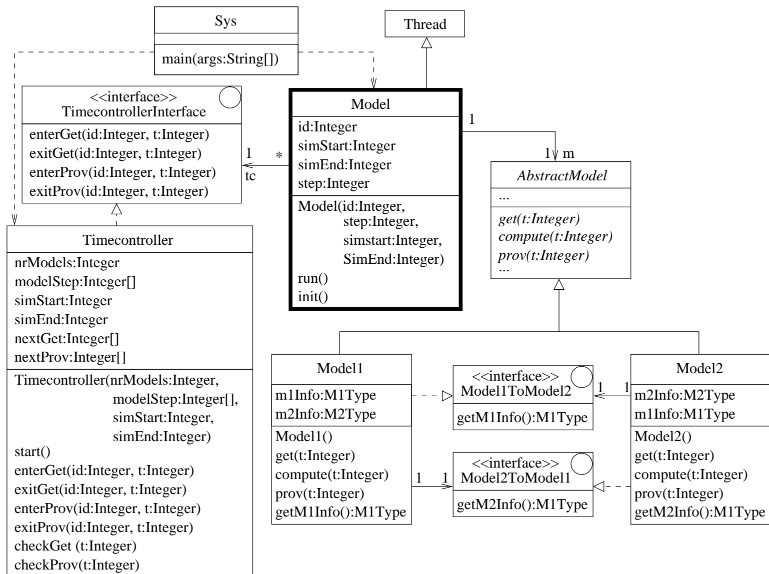
$\|$CHECK_VALIDDATA_USER2_PROV1 =
(SYS $\|$ VALIDDATA(2, StepModel2, 1, StepModel1)).

Fehlerzustand nicht erreichbar.

# 9.5 Systemimplementierung

- Simulationsmodelle werden durch Threads implementiert (aktive Objekte).
- Der Timecontroller wird durch einen Monitor implementiert (passives Objekt), der beliebig viele Simulationsmodelle koordinieren kann.
- Für den Zugriff der Modelle auf den Timecontroller wird ein Interface eingeführt, das die enter- und exit-Operationen anbietet.
- Für die Ausführung der get, compute und prov Operationen wird eine abstrakte Klasse eingeführt (*AbstractModel*), die entsprechend konkreter, fachspezifischer Simulationen implementiert wird (Framework-Idee!).
- Für den Datenaustausch zwischen fachspezifischen Simulationen werden entsprechende Interfaces verwendet.

# Architektur der Implementierung



# Java-Code

**main-Methode der Klasse Sys (bei 2 Simulationsmodellen):**

```
public static void main(String[] args) {
  int simStart = 0;
  int simEnd = 6;
  int nrModels = 2;
  int[] modelStep = new int[] { 2, 3 };
  Timecontroller tc =
    new Timecontroller(nrModels, modelStep, simStart, simEnd);
  tc.start();
  Model m1 =  new Model(1, 2, simStart, simEnd, tc);
  m1.start();
  Model m2 = new Model(2, 3, simStart, simEnd, tc);
  m2.start();
}
```

## run-Methode der Klasse Model

```
public void run() {
  init();
  int t=0;
  while (t+step<=simEnd) {
    try {
      tc.enterGet(id, t);
    } catch (InterruptedException e) {}
    m.get(t);
    tc.exitGet(id, t);
    m.compute(t);
    try {
      tc.enterProv(id, t+step);
    } catch (InterruptedException e) {}
    m.prov(t+step);
    tc.exitProv(id, t+step);
    t = t+step; }
 }
```

## Methoden der Klasse Timecontroller

```
public void start() {
  for (int i=0; i<nrModels; i++) {
    nextGet[i]=simStart;
    nextProv[i]=simStart; }
}

public synchronized void enterGet(int id, int t)
  throws InterruptedException {
  while (!checkProv(t)) wait();
}

private boolean checkProv(int t) {
  boolean b = true;
  for (int i = 0; i < nrModels; i++) {
    b = (b && (t < nextProv[i]));
  }
  return b;
}
```

```
public synchronized void exitGet(int id, int t) {
  // subtract 1 to match model id!
  nextGet[id-1] = nextGet[id-1] + modelStep[id-1];
  notifyAll();
}

public synchronized void enterProv(int id, int t)
  throws InterruptedException {
  while (!checkGet(t)) wait();
}

private boolean checkGet(int t) {
  boolean b = true;
  for (int i = 0; i < nrModels; i++) {
    b = (b && (nextGet[i] >= t));
  }
  return b;
}

public synchronized void exitProv(int id, int t) {
  nextProv[id-1] = nextProv[id-1] + modelStep[id-1];
  notifyAll();
}
```