Kurs OMSI
im WiSe 2010/11

Objektorientierte Simulation mit ODEMx

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

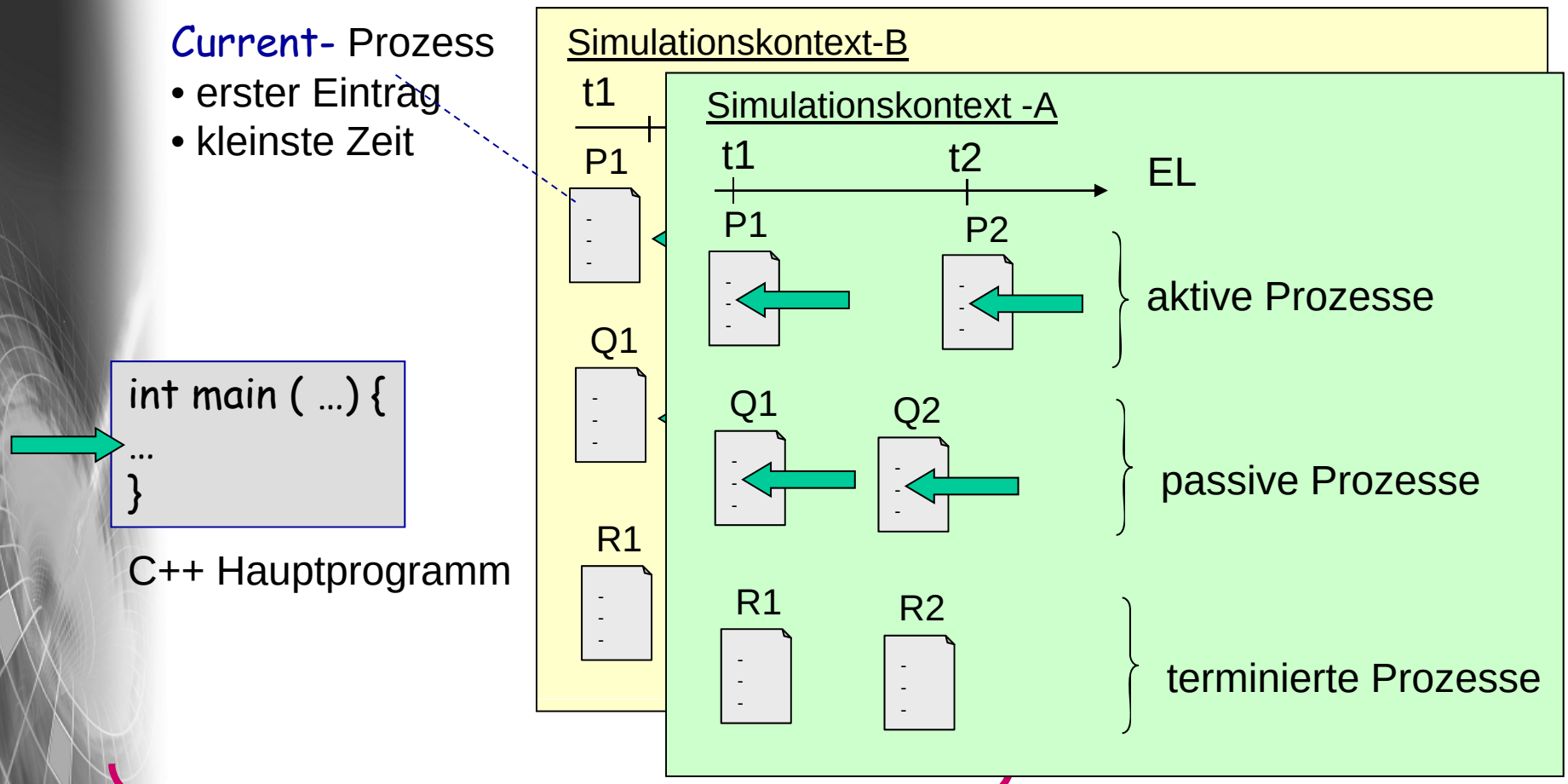
fischer|ahrens|eveslage@informatik.hu-berlin.de

2. Prinzip der Next-Event-Simulation

1. Charakterisierung der Next-Event-Simulation
2. Umsetzung des Prinzips in ODEMx
   - Aufbau von ODEMx
   - Simulationskontext
   - Simulationskontext (Barrenbeispiel)

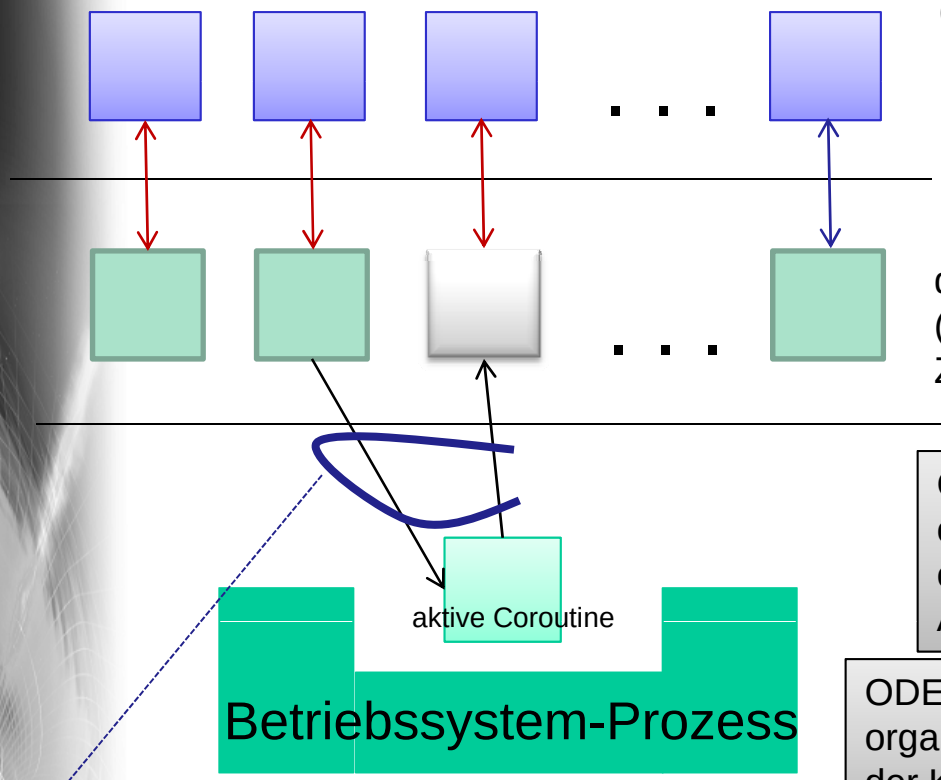
Grundidee einer hierarchischen Prozessverwaltung

Current- Prozess
- erster Eintrag
- kleinste Zeit



Hauptprogramm und Prozesse aller Simulationskontexte bilden ein hierarchisches Coroutinensystem

Universelle ODEMx- Urvariante



dynamisches Ensemble von realen/gedachten Prozessen (zeitskret/zeitkontinuierlich) (Original)

Struktur-Äquivalenz (1:1)

dynamisches Ensemble von ODEMx-Prozessen (Simulationsmodell)
Zustand jedes Prozesses auf der Halde

ODEMx-Laufzeitsystem
organisiert Scheduling der ODEMx-Prozesse entsprechend ihrem Modellzeitverbrauch für Änderung ihrer Zustandsgrößen

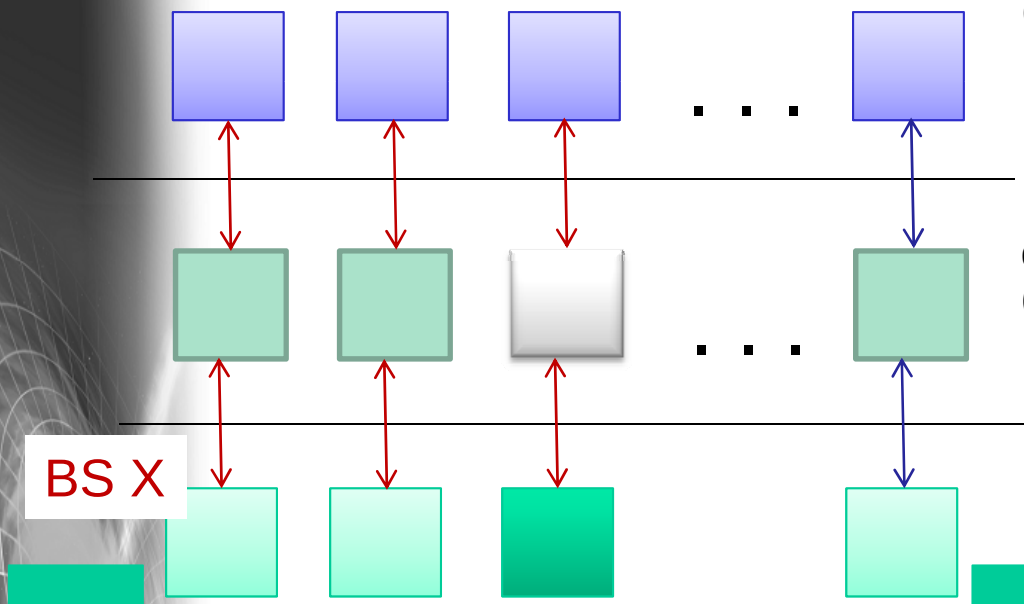
ODEMx-Laufzeitsystem
organisiert Speicherung und Restaurierung der Koroutinen-Zustände (Laufzeitkeller, Register)

rechenzeitaufwändiger

… im Vergleich zu prozeduralen Next-Event-Verfahren

aber hoher Grad an auf natürliche Weise erreichbarer Strukturäquivalenz

Laufzeitverbesserte Varianten



dynamisches Ensemble von realen/gedachten Prozessen (zeitdiskret/zeitkontinuierlich) (Original)

Struktur-Äquivalenz (1:1)

dynamisches Ensemble von ODEMx-Prozessen (Simulationsmodell)

Struktur-Äquivalenz (1:1)

BS X

Betriebssystem-Prozess

ODEMx-Laufzeitsystem
organisiert Scheduling der ODEMx-Prozess- entsprechend ihrem Modellzeitverbrauch für Änderung ihrer Zustandsgrößen

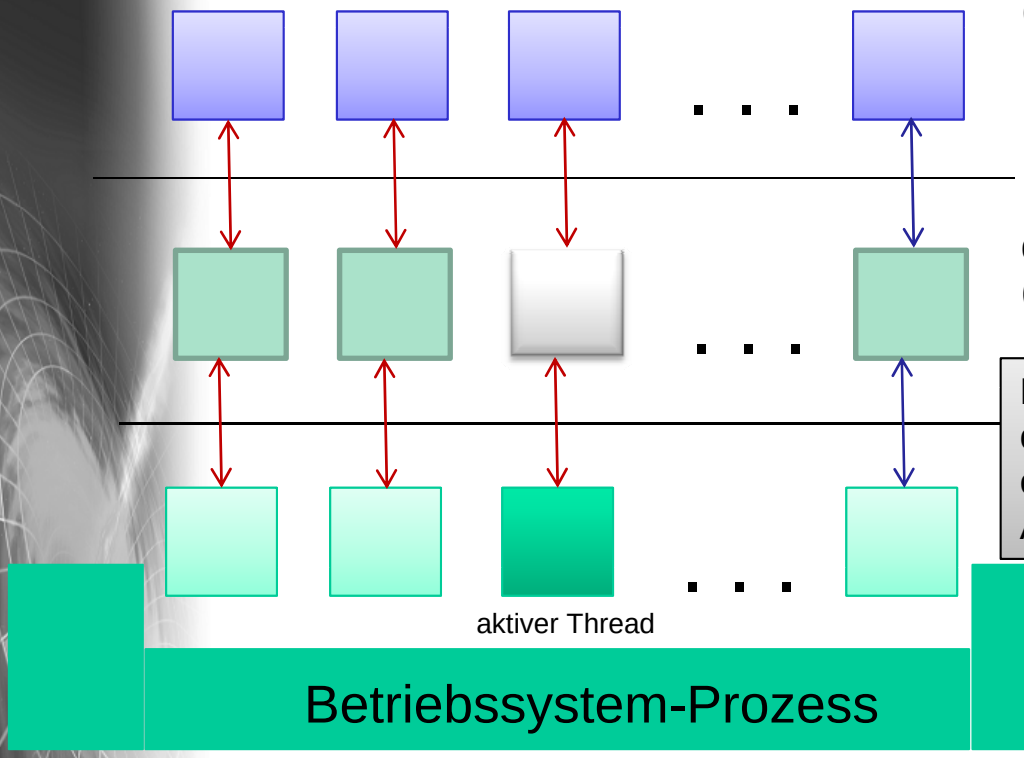
X= Fibres, MS Windows

X= uContext, Linux

ODEMx-Laufzeitsystem
organisiert Umsetzung von Zeigern zu Koroutinen-Zustände (Laufzeitkeller) und speichert und restauriert lediglich Register

Java-basierte Variante (DESMO-J)

DESMO-J = ODEMx – {zeitkontinuierl., Protokoll-Konzepte}



dynamisches Ensemble von realen/gedachten Prozessen (zeitdiskret/zeitkontinuierlich) (Original)

Struktur-Äquivalenz (1:1)

dynamisches Ensemble von DESMO-J-Prozessen (Simulationsmodell)

DESMO-J-Laufzeitsystem
organisiert Scheduling der DESMO-J-Prozesse entsprechend ihrem Modellzeitverbrauch für Änderung ihrer Zustandsgrößen

aber laufzeitschlechter als ODEMx-Urvariante (schwergewichtigere Lösung)
Anzahl von DESMO-J-Prozessen ist begrenzt

Klasse Sched, Event, Process

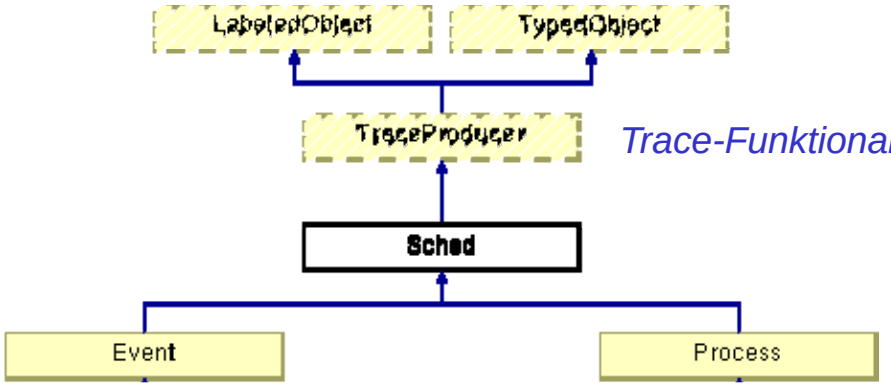
Sched

- abstrakte Klasse
- Objekte werden im Kalender in chronologischer Reihenfolge erfasst

Simulationslauf

- ist die Ausführung (execute) von Sched-Objekten
- in Abhängigkeit von
  - der jeweiligen Kalenderkonstellation und
  - der Typen der Sched-Objekte

Namensfunktionalität *Typerkennung*



dabei können neben Zustandsänderungen auch
- Eintragungen,
-Verschiebungen und Streichungen von Sched-Objekten vorgenommen

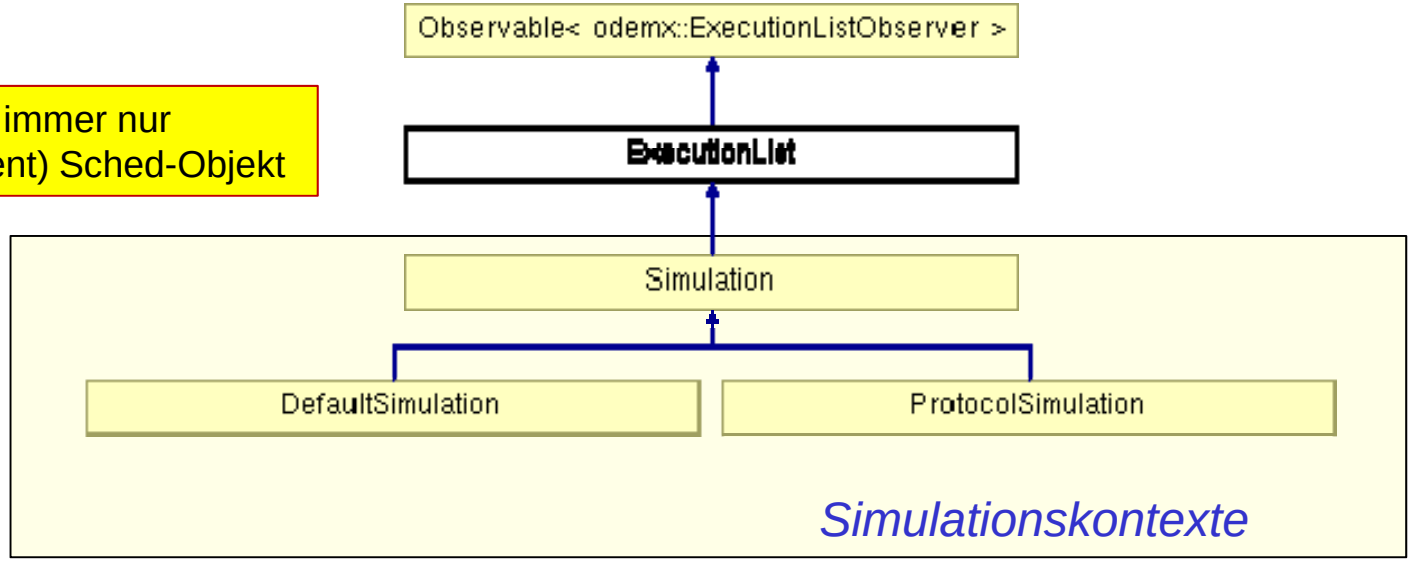
```
virtual SimTime getExecutionTime () const =0          // Get model time.
virtual SimTime setExecutionTime (SimTime time)=0     // Set model time.
virtual Priority getPriority () const =0              // Get priority.
virtual Priority setPriority (Priority newPriority)=0 // Set new priority.
bool isScheduled () const                             // Check if Sched object is in schedule.
SchedType getSchedType () const                       // Determine the Sched object's type.
virtual void execute ()=0                             // Execution of Sched object.
```

Die Klasse ExecutionList *(Ereignisliste, Kalender)*

```
Sched * getNextSched ()    // top most Sched in ExecutionList
bool isEmpty ()            // check if ExecutionList is empty
virtual SimTime getTime () // const =0  get model time
```

Vergangenheit wird nicht konserviert

ausgeführt wird immer nur das **erste** (current) Sched-Objekt



jeder Simulationskontext (Objekt von Simulation bzw. Ableitung) verfügt über eine eigene ExecutionList-Funktionalität

Scheduling-Prinzip: A Sched object is scheduled
- at a given time
  considering its priority and a FIFO- or LIFO- strategy,
  or
  in relation to an already scheduled object.

Modellzeit

Wert vom Typ SimTime

getTime() liefert stets die aktuelle Modellzeit (größer gleich Null)

ODEMx-Bibliothek in zwei Varianten:

- diskrete Zeit: long
- kontinuierliche Zeit: double

ODEMx-Bibliothek mit unterschiedlichen Ausbaustufen:

- zeitdiskrete Zustandsänderungen
- zeitdiskrete und zeitkontinuierliche Zustandsänderungen

2. Prinzip der Next-Event-Simulation

1. Charakterisierung der Next-Event-Simulation
2. Umsetzung des Prinzips in ODEMx
   - Aufbau von ODEMx
   - Simulationskontext (Trivialbeispiel)
   - Simulationskontext (Barrenbeispiel)

Uhren-Beispiel: Einrichten und Start einer Simulation

```
class Clock : public Process {
public:
    Clock (Simulation* sim) :
        Process(sim, "Clock") {}

    virtual int main() {
        while (true) {
            holdFor(1.0);

            cout << '.';
        }

        return 0;
    }
};
```

Zuordnung eines Simulations-kontextes
als Simulation- Zeiger

Prozessobjekte
(hier: Instanzen von Clock)
erhalten ein 1-deutiges Label
(Bezeichner&Nummer)

Verhalten des Nutzerprozesses
als Redefinition
der (pure virtual) Funktion int main().

Rückgabewert bei Beendigung
- wird als long gespeichert,
- zugreifbar mit getReturnValue()
- Test eines Prozess-Objektes mit hasReturned()

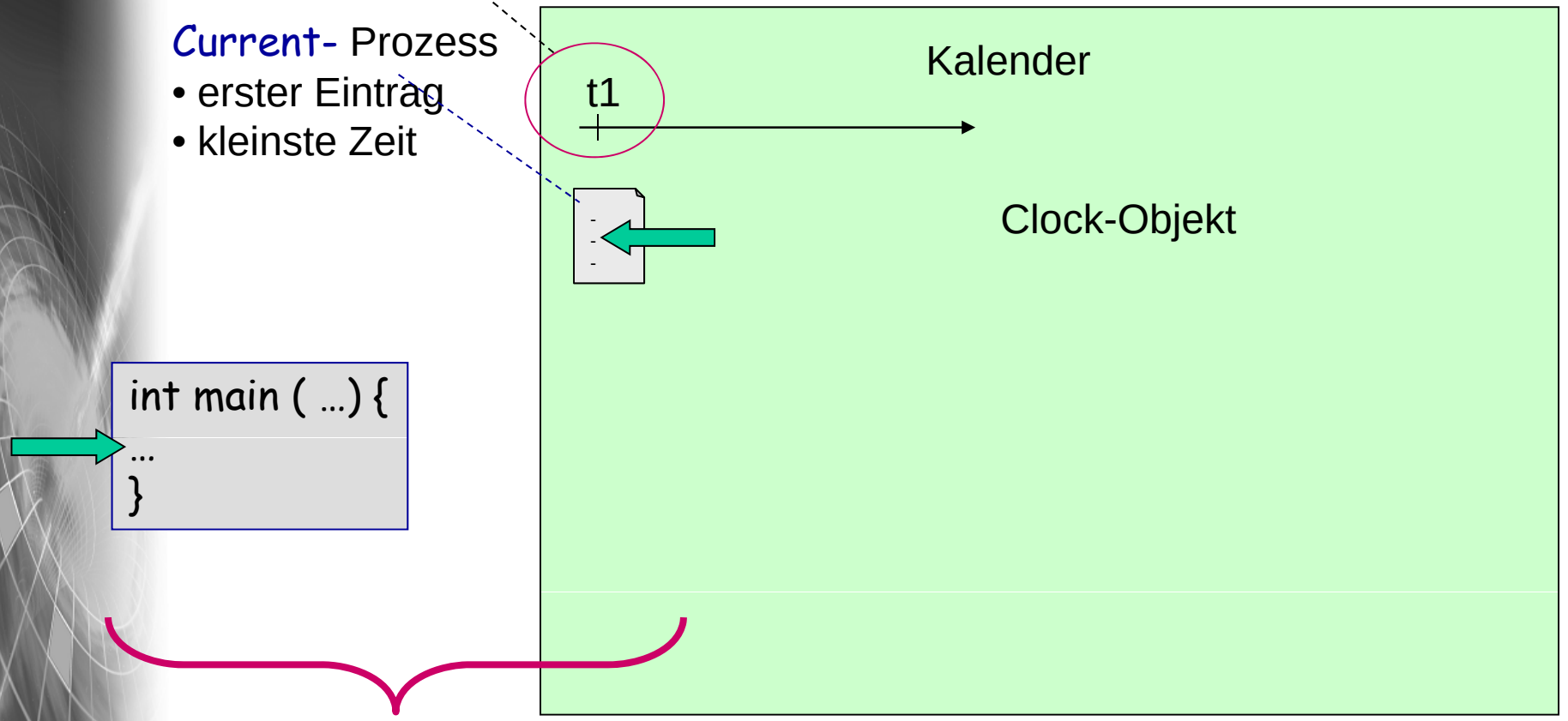
eine von mehreren Scheduling-Operationen, Verzögerung um eine Zeitdauer mit evtl. Prozesswechsel

Einfaches Prozess-Ensemble

getTime(): Time

Simulationskontext (DefaultSimulation-Objekt)

Current- Prozess
- erster Eintrag
- kleinste Zeit



C++ Hauptprogramm und der einzige Prozess des Simulationskontextes (d.h. dessen Memberfunktion main) bilden ein Koroutinensystem

Uhren-Beispiel: C++ Hauptprogramm

```
int main(int argc, char* argv[]) {
    Clock* myClock=
            new Clock (getDefaultSimulation());
    myClock->activate();

    cout << "Basic Simulation Example" << endl;
    cout << "========================" << endl;

    for (int i=1; i<5; ++i) {
            getDefaultSimulation() ->step();
            cout << endl << i << ". step time=" << getDefaultSimulation()->getTime()
    << endl;
    }

    cout << endl;
    cout << "continue until SimTime 13.0 is reach
    getDefaultSimulation() ->runUntil(13.0);
    cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

    cout << "========================" << endl;
    return 0;
}
```

Zuordnung des Default-Simulations-kontextes

Objekt wird erzeugt, aber nicht aktiviert

Scheduling-Operation: Objekt wird zum aktuellen Zeitpunkt in EL[1] als 1.Eintrag vermerkt (aber nicht aktiviert)

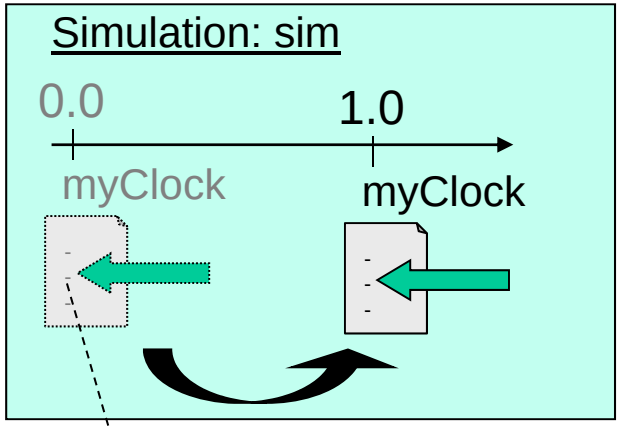
1.Eintrag der EL wird ausgeführt,
- dieser realisiert seine Zustandsänderungen
- nach Ausführung eines Scheduling-Schrittes kehrt die Steuerung ins C++ Hauptprogramm zurück

1.Eintrag der EL wird ausgeführt,
- solange die Modellzeit des Simulationskontextes <= 13.0 ist

[1]EL= ExecutionList/Kalender

Standardfall: DefaultSimulation als Kontext

- **Konvention**: Erzeugung und Aktivierung der Prozesse bei Nutzung von Default-Simulation erfolgt in main (C++ -Hauptprogramm)
- step() aktiviert das Prozess-Ensemble, d.h. 1.Kalender-Eintrag: myClock (zum Zeitpunkt 0.0) ➔ Koroutinenwechsel
- myClockverzögert sich um 1.0 ZE, d.h.
  - Umsortierung im Kalender
  - Step-Modus: Rückgabe der Steuerung an main() ➔ Koroutinenwechsel
  - Ausgabezeile und erneuter step()
- 5-maliger Wechsel zwischen Hauptprogramm und myClock
- bis Zeitpunkt 5.0 erreicht wird

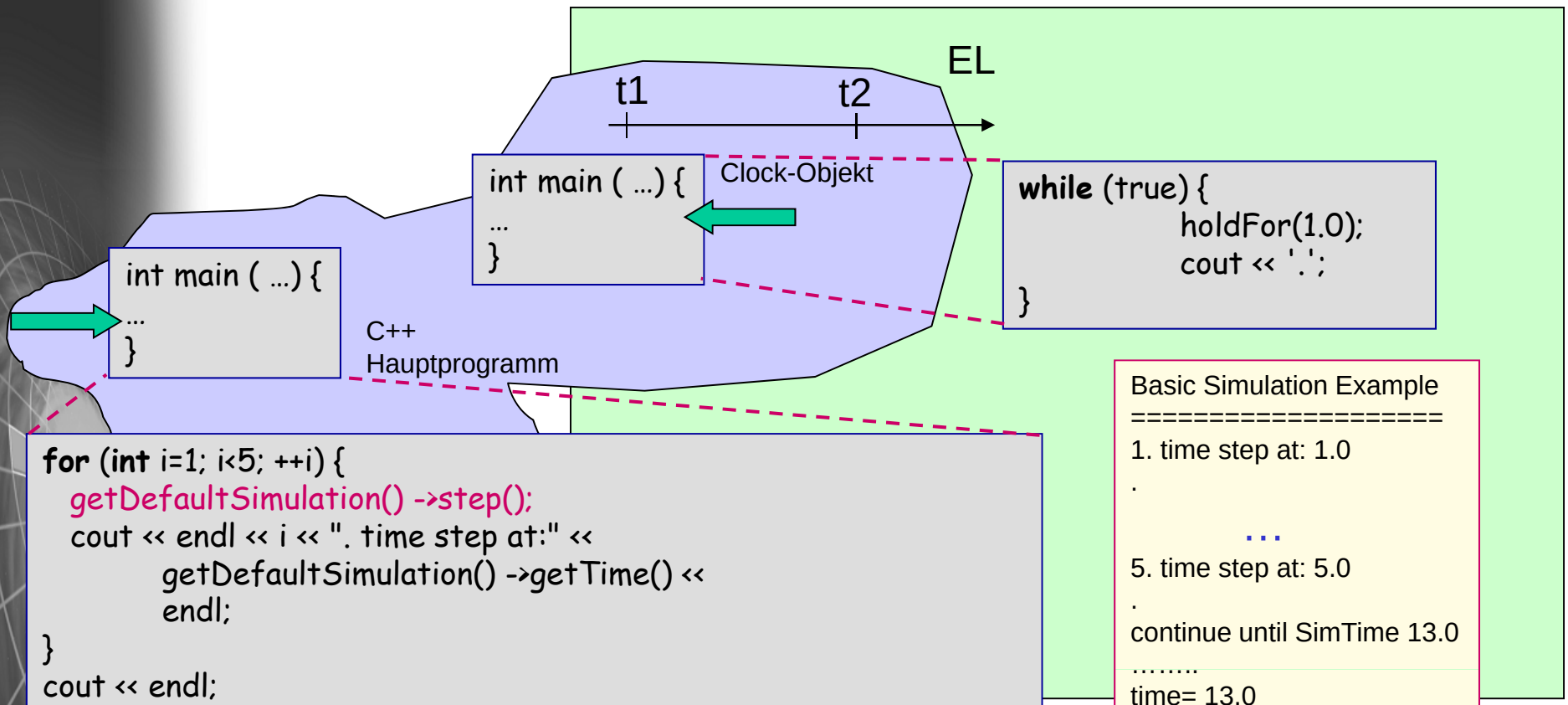



```
for (int i=1; i<5; ++i) {
   getDefaultSimulation() ->step();
   cout << endl << i << ". step time=" <<
           getDefaultSimulation() ->getTime() <<
           endl;
}
```

```
virtual int main() {
   while (true) {
           holdFor(1.0);
           cout << '.';
   }
}
```

Beispiel: Einfaches Prozess-Ensemble

Simulationskontext (DefaultSimulation-Objekt)




```
for (int i=1; i<5; ++i) {
  getDefaultSimulation() ->step();
  cout << endl << i << ". time step at:" <<
          getDefaultSimulation() ->getTime() <<
          endl;
}
cout << endl;
cout << "continue until SimTime= 13.0";
getDefaultSimulation()->runUntil(13.0);
cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;
cout << "======================" << endl;
```

```
Basic Simulation Example
====================
1. time step at: 1.0
.
...
5. time step at: 5.0
.
continue until SimTime 13.0
……..
time= 13.0
====================
```

Varianten der Simulationsberechnung

bisher besprochen

1. Einzelschrittausführung: `step()`

2. Lauf bis zum Erreichen/Überschreiten einer vorgegebenen Modellzeit (`SimTime`): `runUntil(...)`

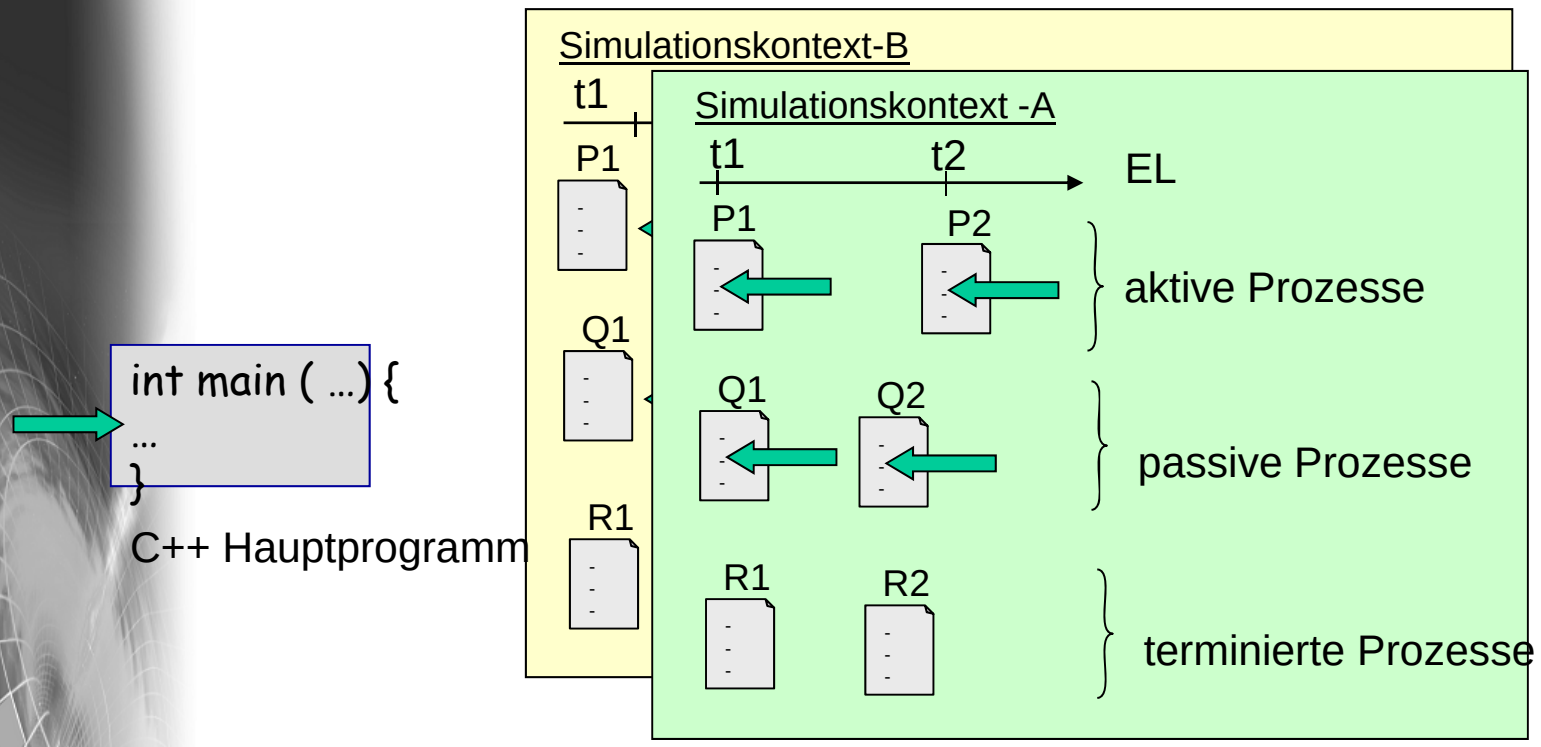
3. Lauf bis zum Ende der Simulation: `run()`

   Rückkehr ins C++ Hauptprogramm:

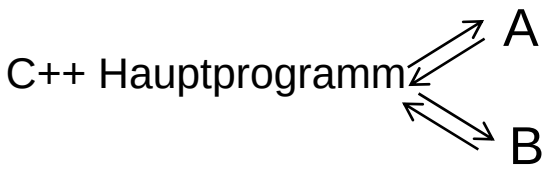
   – ***implizit***:
   es gibt keinen **aktiven** Prozess mehr im zugehörigen Simulationskontext (Kalender ist leer)

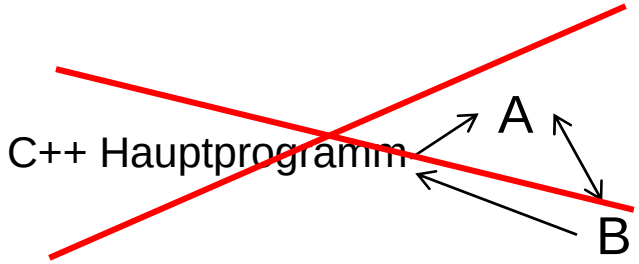
   – ***explizit***: die Simulation wurde mit `exitSimulation()` durch einen Prozesses des Simulationskontextes beendet

Verwaltung mehrerer Simulationskontexte



Steuerungszenarien:



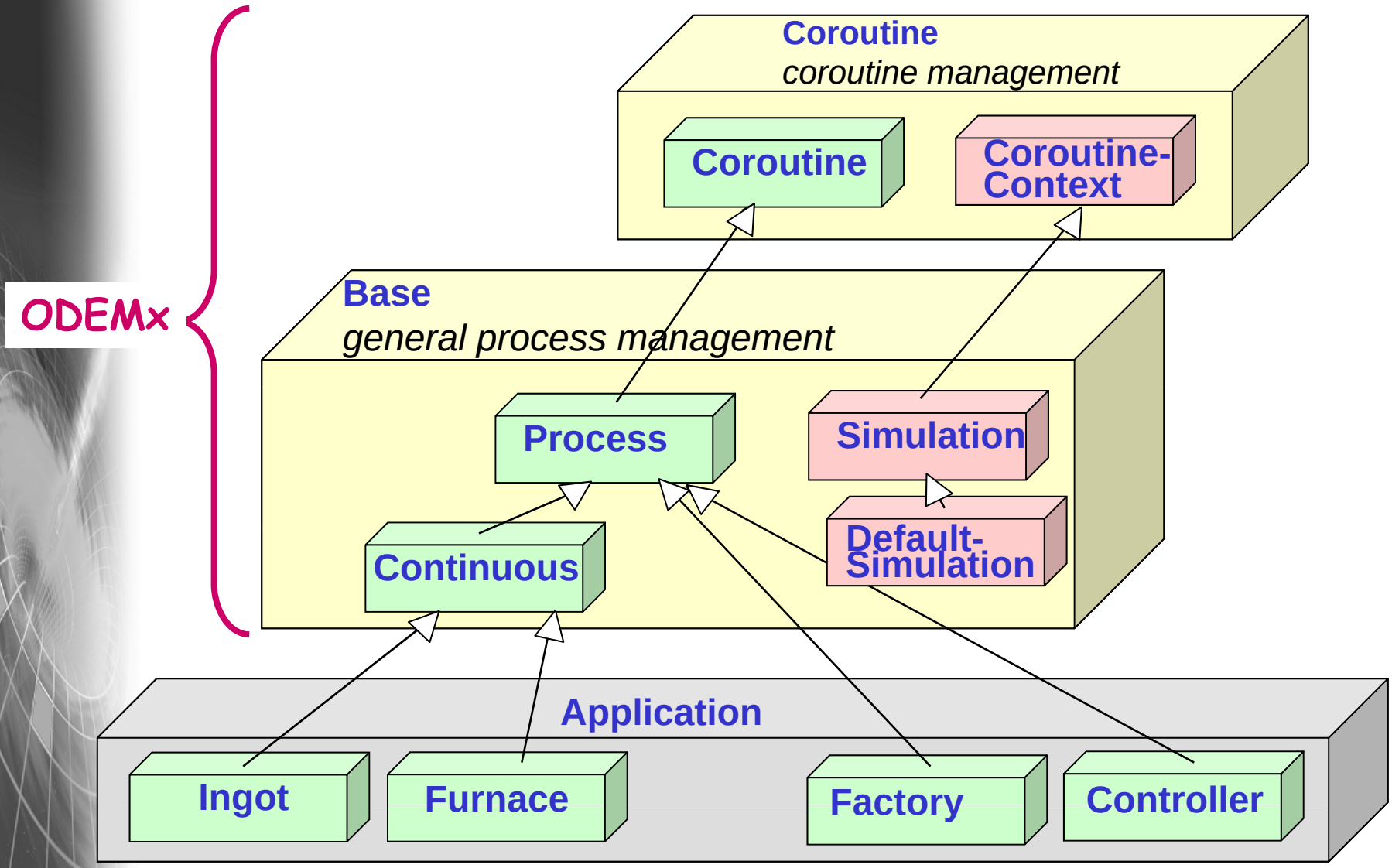


2. Prinzip der Next-Event-Simulation

1. Charakterisierung der Next-Event-Simulation
2. Umsetzung des Prinzips in ODEMx
   - Aufbau von ODEMx
   - Simulationskontext
   - Simulationskontext (Barrenbeispiel)

3. Schritt: Softwaretechnische Umsetzung

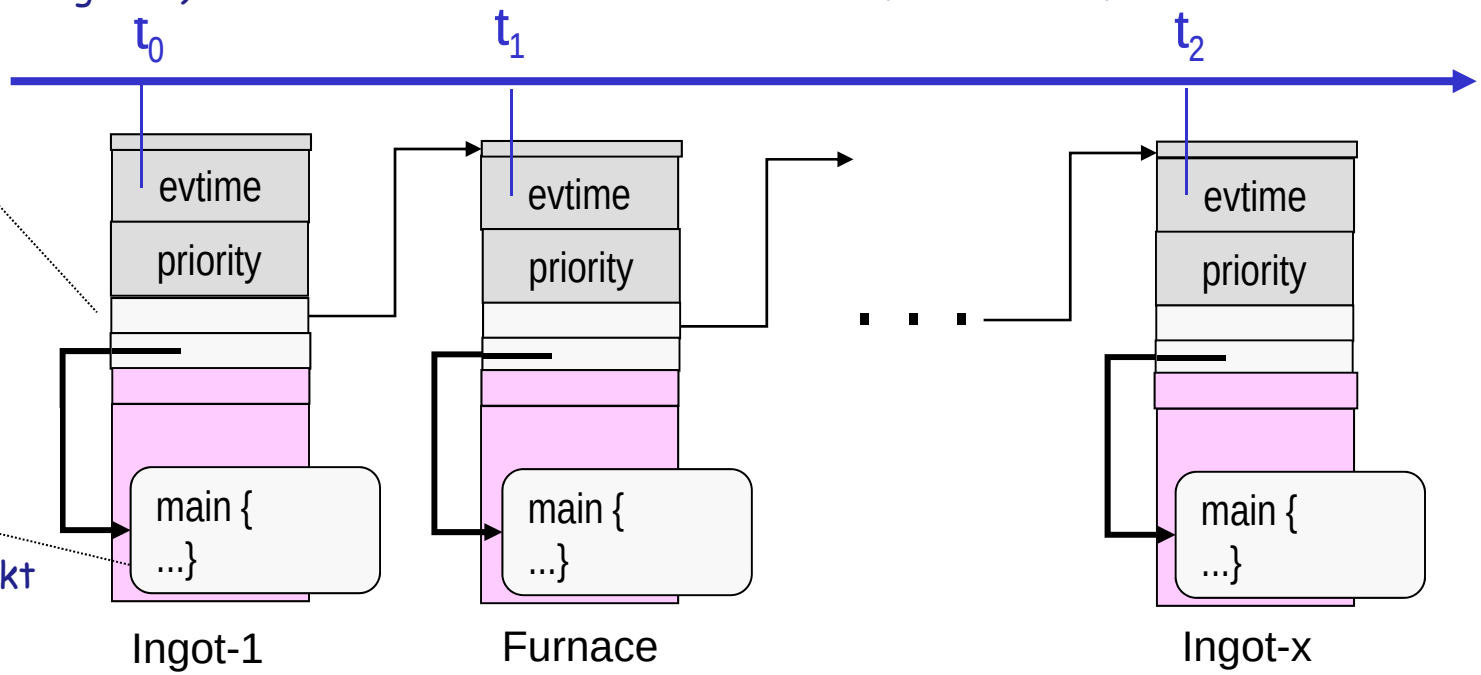
- bei Nutzung von ODEMx -



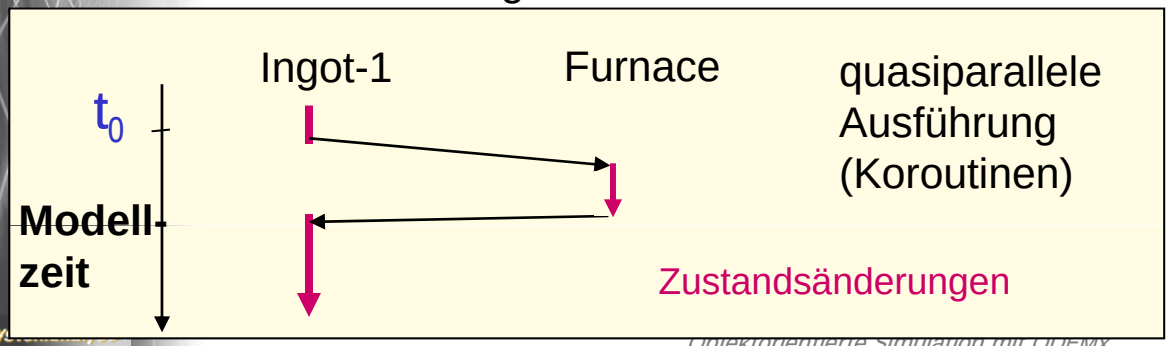
Process-Scheduling

Ereignisliste (Kalender):
Verwaltung von Objekten
von Sched-Ableitungen
in chronologischer Reihenfolge
(Modellzeit)

Verwaltungsinformation
(Stackaufbau, Register,
insbesondere Befehlsregister)



Unterbrechbarkeit
des Lebenszyklus,
Fortsetzung nach
Unterbrechungspunkt

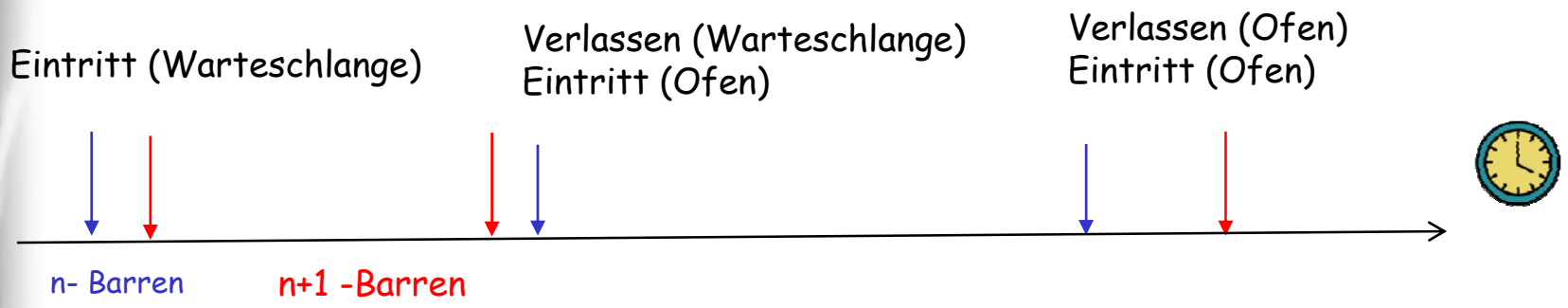


entscheidende
konzeptuelle Grundlage:
Zustandsänderungen
in individuellen Lebenszyklen
sind modellzeitverbrauchend
(Spezialfälle: Null-Zeit)

Ausführung des Process-Scheduling

nach dem Prinzip der Next-Event-Simulation

getTime() liefert Werte einer monoton wachsenden SimTime-Folge



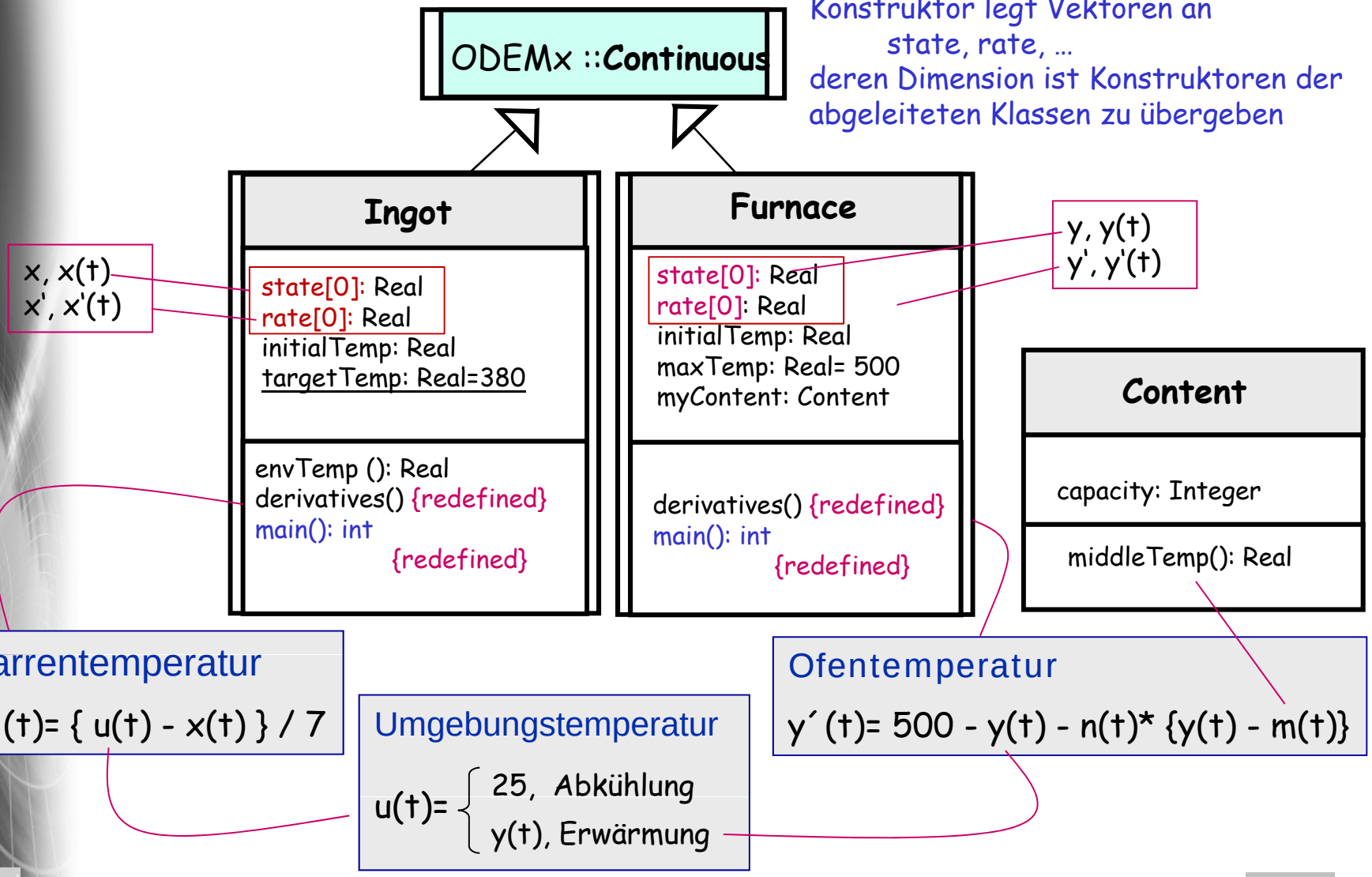
am Ereignis:
‚Verlassen des Ofens' klarmachen !

Unterscheidung zwischen
- Zeitereignissen und
- Zustandsereignissen

Ausführung einer Aktion
zu einem vorgegeben Zeitpunkt

Ausführung einer Aktion
in Abhängigkeit eines erreichten Zustandes

Modellierung zeitkontinuierlicher Prozesse



ODEMx-Modellierung der Barrenprozessklasse Ingot

```
Ingot::Ingot () : Continuous("Ingot", 1) {
    set_minmax (0.001, 10.0);
    set_errorlimit(1, 0.01);

    state[0]= temp->sample();
    cold    = false;
    add_peer(contr);
};

int Ingot::main() {

    integrate (0.0);
    return 0;
};

void Ingot::derivatives (SimTime t) {
    double ambient;
    ambient= cold ? 25.0 : furn->state[0];
    rate[0]= (-state[0] + ambient) / 7;
};
```

```
class Ingot :  public Continuous {
public:
    bool cold;
    Ingot ();
    void derivatives (double);
    int main();
};
```

Ensemble von main()-Funktionen aller existenten Prozess-Objekte wird zusammen
mit eigentlichem C++ Hauptprogramm
als Ensemble von Koroutinen
quasiparallel ausgeführt

ODEMx-Modellierung der Barrenprozessklasse Ingot

```
Ingot::Ingot () : Continuous("Ingot", 1) {
    set_minmax (0.001, 10.0);
    set_errorlimit(1, 0.01);
    set_tracelevel(0);
    state[0]= temp->sample();
    cold    = false;
    add_peer(contr);
};

int Ingot::main() {

    integrate (0.0);
    return 0;
};

void Ingot::derivatives (double) {
    double ambient;
    ambient= cold ? 25.0 : furn->state[0];
    rate[0]= (-state[0] + ambient) / 7;
};
```

```
class Ingot :  public Continuous {
public:
    bool cold;
    Ingot ();
    void derivatives (double);
    int main();
};
```

Ingot-Klasse

- als kontinuierlicher Prozess
- mit Differentialgleichungssystem (1.Ordnung) der Dimension n=1

Satz von Vektoren der Dimension 1
state, rate, s1, s2, s3, s4, …, initialState, errorVector

ODEM-Modellierung der Barrenprozessklasse Ingot

```
Ingot::Ingot () : Continuous("Ingot", 1) {
    set_minmax (0.001, 10.0);
    set_errorlimit(1, 0.01);
    set_tracelevel(0);
    state[0]= temp->sample();
    cold    = false;
    add_peer(contr);
};
int Ingot::main() {
    set_tracelevel(0);
    integrate (0.0);
    return 0;
};

void Ingot::derivatives (double) {
    double ambient;
    ambient= cold ? 25.0 : furn->state[0];
    rate[0]= (-state[0] + ambient) / 7;
};
```

```
class Ingot :  public Continuous {
public:
    bool cold;
    Ingot ();
    void derivatives (double);
    int main();
};
```

Ingot-Klasse
- Verfahrensparameter (Schrittweite, Fehler, Ausgabeart)

ODEM-Modellierung der Barrenprozessklasse Ingot

```
Ingot::Ingot () : Continuous("Ingot", 1) {
    set_minmax (0.001, 10.0);
    set_errorlimit(1, 0.01);
    set_tracelevel(0);
    state[0]= temp->sample();
    cold    = false;
    add_peer(contr);
};
int Ingot::main() {
    set_tracelevel(0);
    integrate (0.0);
    return 0;
};

void Ingot::derivatives (double) {
    double ambient;
    ambient= cold ? 25.0 : furn->state[0];
    rate[0]= (-state[0] + ambient) / 7;
};
```

Zufallswert

```
class Ingot :  public Continuous {
public:
    bool cold;
    Ingot ();
    void derivatives (double);
    int main();
};
```

Ingot-Klasse

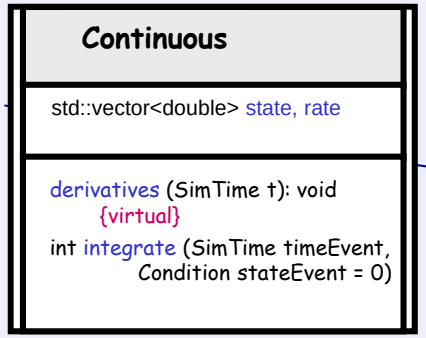
- state[0]
  Zustandsvektor, zeigt den Wert der Temperatur des Barren-objektes
- wird mit Zufallswert initialisiert
- zu synchronisieren mit Prozess contr

ODEM-Modellierung der Barrenprozessklasse Ingot

```
Ingot::Ingot () : Continuous("Ingot", 1) {
    set_minmax (0.001, 10.0);
    set_errorlimit(1, 0.01);
    set_tracelevel(0);
    state[0]= temp->sample();
    cold    = false;
    add_peer(contr);
};
int Ingot::main() {

    integrate (0.0);
    return 0;
};

void Ingot::derivatives (double) {
    double ambient;
    ambient= cold ? 25.0 : furn->state[0];
    rate[0]= (-state[0] + ambient) / 7;
};
```




```
class Ingot :  public Continuous {
public:
     bool cold;
     Ingot ();
     void derivatives (double);
     int main();
};
```

Ingot-Klasse

- Lebenslauf:

startet
numerische Integration parallel
zu anderen Prozessen
→ dabei Temperaturänderung mit
Fortschreiten der Modellzeit
(wie durch DGL beschrieben)

ODEM-Modellierung der Barrenprozessklasse Ingot

```
Ingot::Ingot () : Continuous("Ingot", 1) {
    set_minmax (0.001, 10.0);
    set_errorlimit(1, 0.01);
    set_tracelevel(0);
    state[0]= temp->sample();
    cold    = false;
    add_peer(contr);
};
int Ingot::main() {
    set_tracelevel(0);
    integrate (0.0);
    return 0;
};

void Ingot::derivatives (double) {
    double ambient;
    ambient= cold ? 25.0 : furn->state[0];

    rate[0]= (ambient - state[0] ) / 7.0;
};
```

```
class Ingot :  public Continuous {
public:
    bool cold;
    Ingot ();
    void derivatives (double);
    int main();
};
```

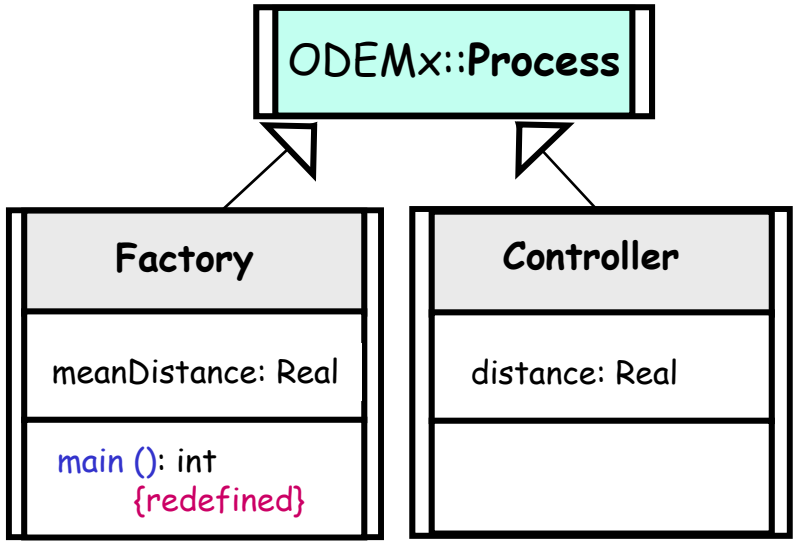
Ingot-Klasse

- Zustandsgleichungen:
  die der
  numerischen Integration
  als virtuelle Funktion zugrunde
  gelegt werden

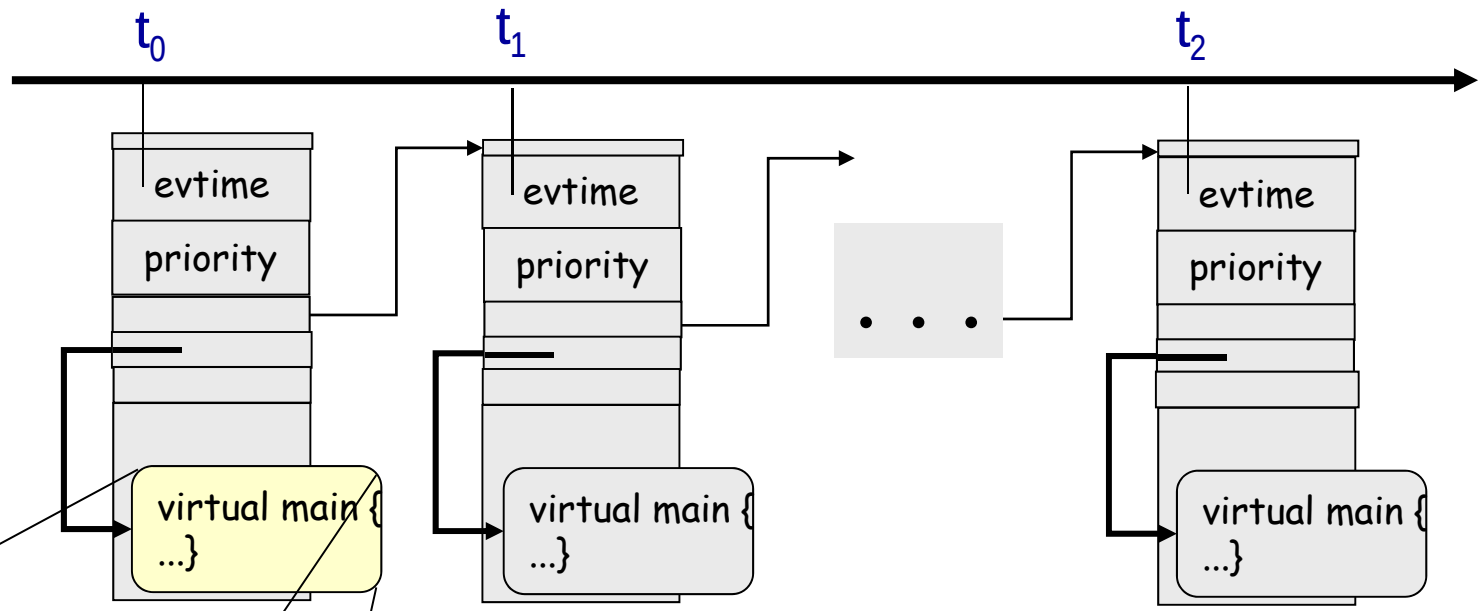
Barrentemperatur

$x'(t)= \{u(t) - x(t)\} / 7$

Lebenslauf zeitdiskreter Prozesse



ODEM-Modellierung der Prozessklasse Factory




```
virtual main {
    for (;;) {
        ing= new Ingot (...);
        ing->into (ingotQ);
        ing->cold:= true;
        ing->start(NOW);
        hold (arrive->sample());
    }
}
```

Zufallswert