Kurs OMSI
im WiSe 2010/11

Objektorientierte Simulation mit ODEMx

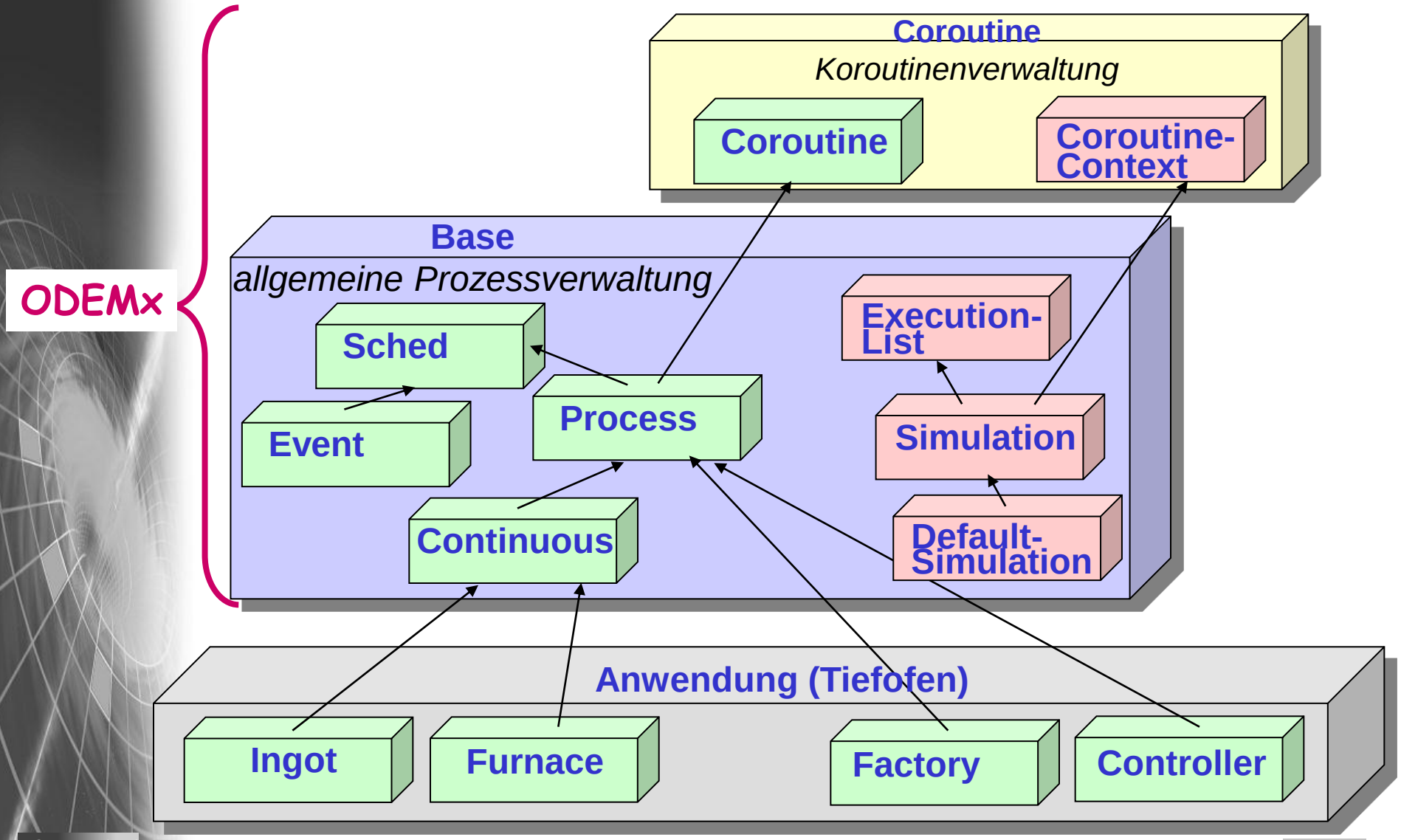
Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

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

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

ODEMx-Prozessverwaltung (Wdh.)



Varianten der Simulationsausführung (Wdh.)

1. Einzelschrittausführung: step()
   - Rückkehr ins Hauptprogramm nach bereits einer einzigen Ereignisrealisierung

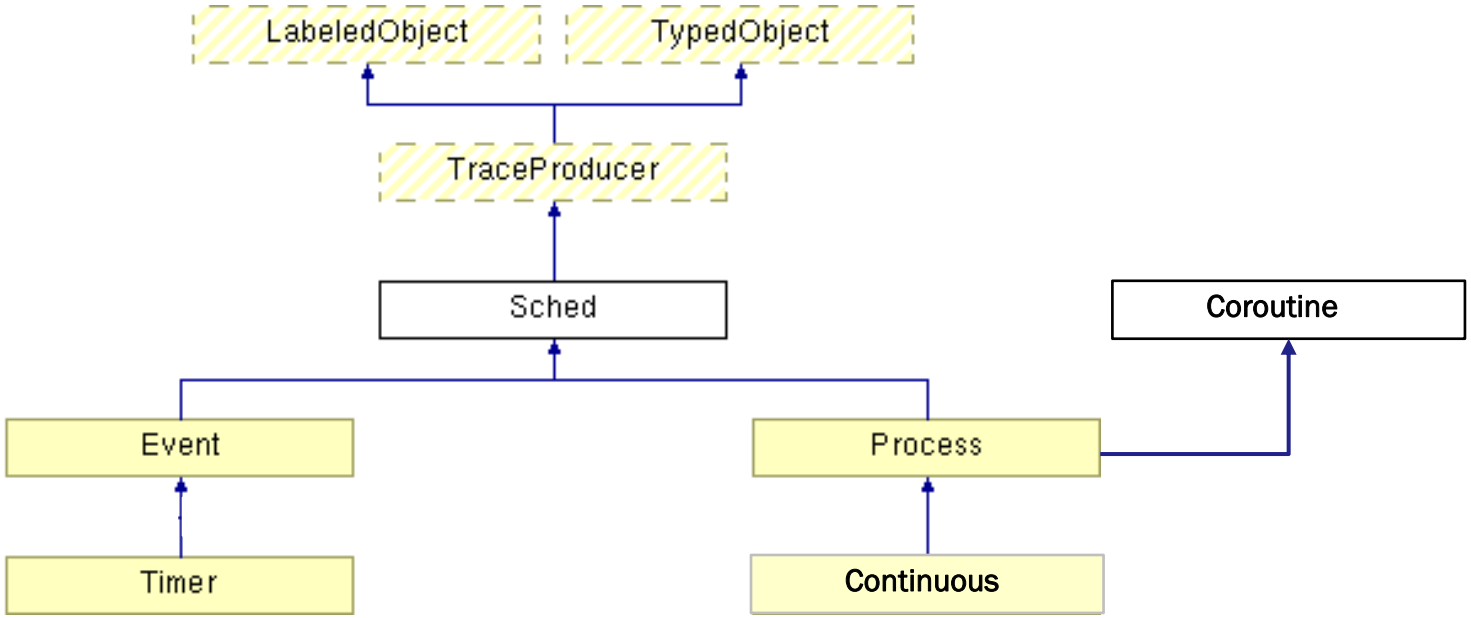
2. zeitabhängige Ausführung: runUntil(…)
   - Rückkehr ins Hauptprogramm nach Erreichen/Überschreiten einer vorgegebenen Modellzeit

3. geschlossene Ausführung: run()
   - ***implizite Beendigung***:
     es gibt keinen aktiven Prozess mehr im zugehörigen Simulationskontext (ExL ist leer)
   - ***explizite Beendigung***:
     Aufruf von exitSimulation() durch einen Prozess des Ensembles

   mit Rückkehr ins Hauptprogramm

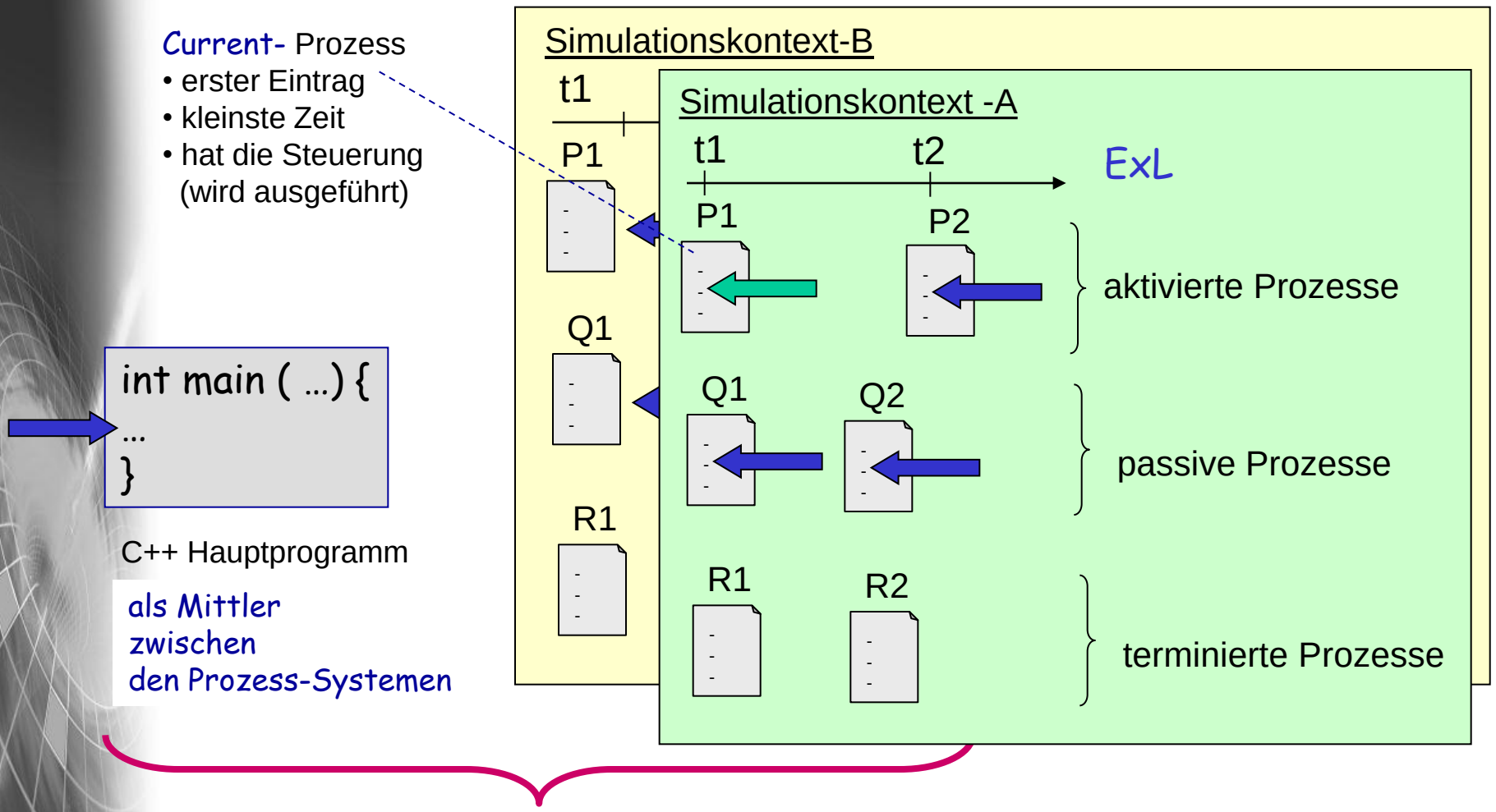
Sched-Objekte und ExecutionList (Wdh.)

Ausschnitt der ODEMx-Klassenhierarchie: Sched



Kontextbasierte Prozessverwaltung (Wdh.)

Current- Prozess
- erster Eintrag
- kleinste Zeit
- hat die Steuerung (wird ausgeführt)



Hauptprogramm und Prozesse aller Simulationskontexte bilden ein hierarchisches Prozess-System

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Grundstrategie

ein Prozess (d.h. Pointer zum Process-Objekt)

- bleibt in seinem gesamten Lebenslauf **einem einzigen Simulationskontext** zugeordnet

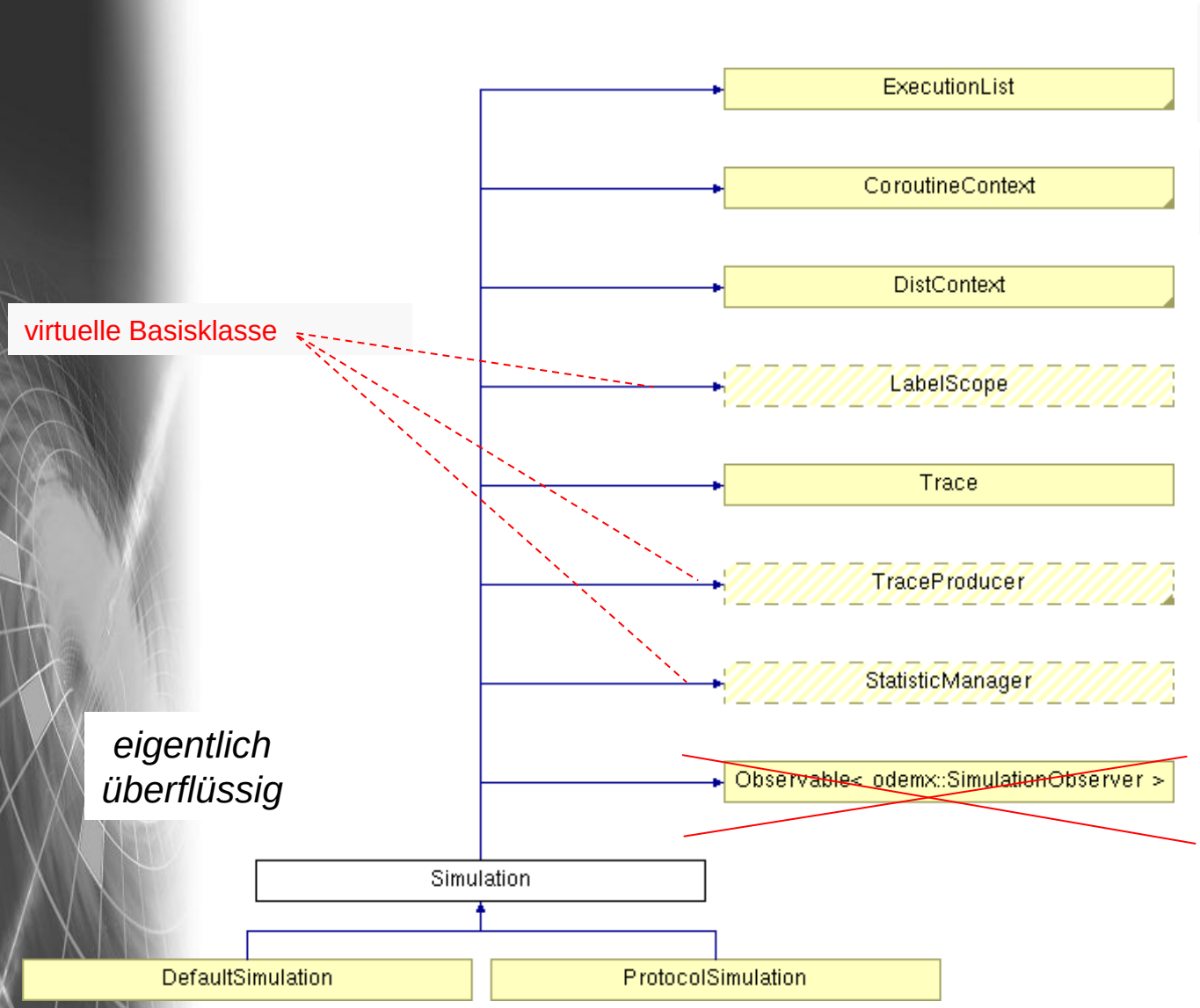
- ist während seines Lebenslaufes (in Abhängigkeit seines Grundzustandes) in vier unterschiedlichen **Listen** seines Simulationskontextes erfasst.

  Grundzustände
  - Created
  - Runable, dann auch in ExL
  - Idle, dann meist auch in dezentralen Synchronisationslisten
  - Terminated

Process-Member-Funktion
`State getState() const;`

zeitgleich kann ein blockierter Prozess (Idle) in weiteren Warteschlangen erfasst sein.

Simulationskontext



Prozess-Scheduling
nach Zeit und Priorität
std::list<Sched*>

ExL

Koroutinen-Ensemble,
inkl. C++ Hauptprogramm

Verwaltung aller erzeugten
Zufallszahlengeneratoren

Objektnamenverwaltung

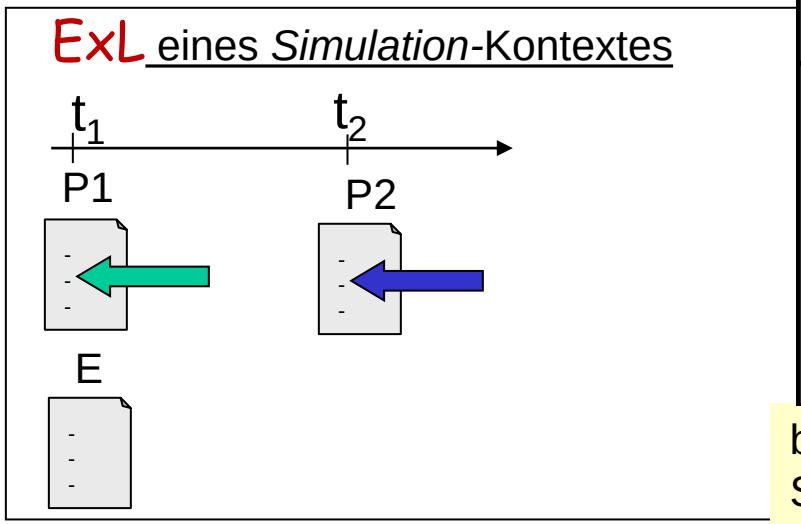
Trace-Manager um
Ereignisse/Zustandsänderungen
von Objekten zu registrieren

Erzeuger von Markierungen
(marks) für Trace

Manager von Statistik-Objekten

Beobachtung registrierter
Objekte

Simulationsparadigma: NextEvent-Simulation



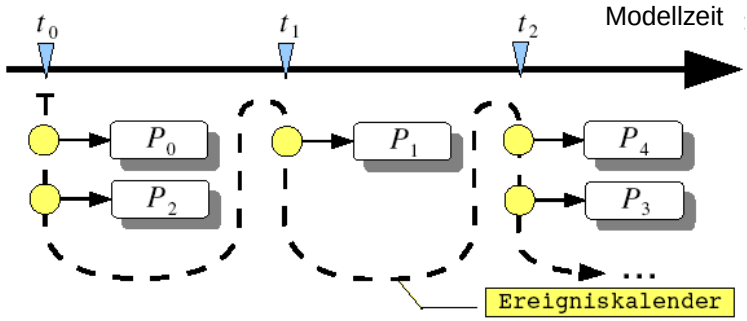
Sched
-schedType: SchedType
+getExecutionTime(): SimTime +setExecutionTime(): SimTime +getPriority(): Priority +setPriority(): Priority +getSchedType(): SchedType

bestimmt das Simulationsprinzip der sogenannten **Next-Event-Simulation**

Ist ein Ereignis (zustandsändernde Aktionen eines Zeitpunktes) realisiert, wird das unmittelbar nächste Ereignis (Ereigniskalender/ExecutionList) ausgeführt,

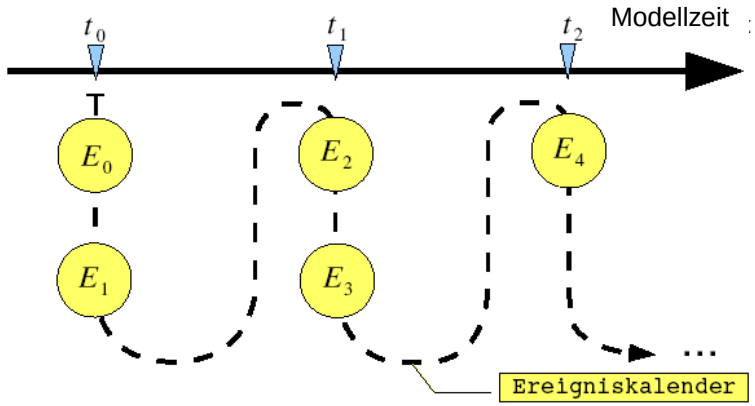
unabhängig davon, ob die Ereignisse als Prozesslebenszyklus (main) strukturiert sind oder nicht

Realisierungen der Next-Event-Simulation



Prozess-Scheduling

(Prozess als Folge von Ereignissen)



Ereignis-Scheduling

(Prozess als Folge von Ereignissen)

ODEMx erlaubt beide Varianten (auch im Mix)

[Sched als abstrakte Basisklasse von Process und Event]

Zugriffsfunktionen für Process-Listen

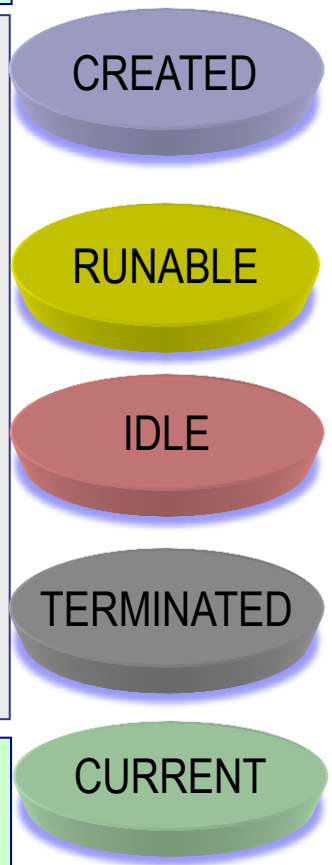
generell
Jeder Prozess wird in seinem gesamten Lebenslauf von seinem Simulationskontext verwaltet *(Zustandslisten eigentlich überflüssig)*

```
std::list<Process*>& Simulation::getCreatedProcesses() {
    return created;
}
std::list<Process*>& Simulation::getRunnableProcesses() {
    return runnable;
}
std::list<Process*>& Simulation::getIdleProcesses() {
    return idle;
}
std::list<Process*>& Simulation::getTerminatedProcesses() {
    return terminated;
}
```

ExL

```
Process* Simulation::getCurrentProcess()
                    Get currently executed process.
Sched * getCurrentSched ()
                    Get currently executed Sched object.
```

Prozess-grundzustand



Zeitbezug

SimTime

Modellzeit: Datentyp bestimmt Varianten von ODEMx: int, double

- now - aktuelle Modellzeit (private Simulation Member-Variable)

Zugriff (nur lesend)

- getCurrentTime()
- getSimulation()->getTime()

geplante Aktivierungszeit eines beliebigen Prozesses p in der ExL

- p->getExecutionTime()

semantisch äquivalent:
now==
getCurrentTime()==
getCurrentProcess()->getExecutionTime() ==
getSimulation()->getTime()

Funktionssignaturen

Process-Member-Funktion

```
SimTime Process::getExecutionTime() const;
        // aktuelle Ereigniszeit
        //      0.0, falls Prozess nicht in ExL eingetragen ist
        // (Vorsicht: 0.0 legt allein noch nicht den Grundzustand fest)
```

Simulation-Member-Funktion

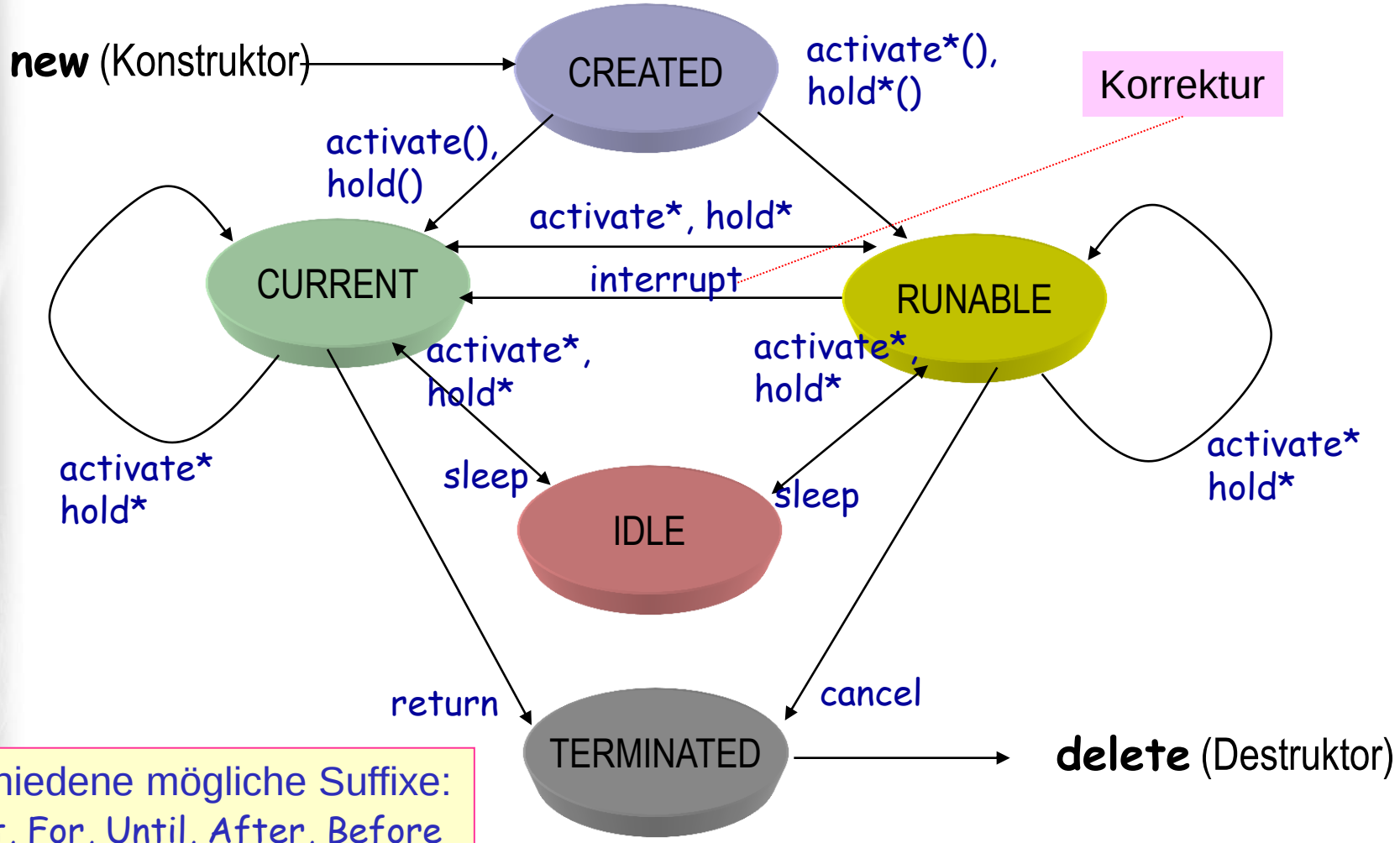
```
Process* Simulation::getCurrentProcess();
        // liefert Zeiger zum aktuellen Prozess der ExL

Simulation* getSimulation();
        // liefert Zeiger zum aktuellen Simulationskontext
```

globale Funktion

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Überblick: Zustände und Scheduling-Operationen



* verschiedene mögliche Suffixe:
In, At, For, Until, After, Before

Process: Scheduling-Operationen (1)

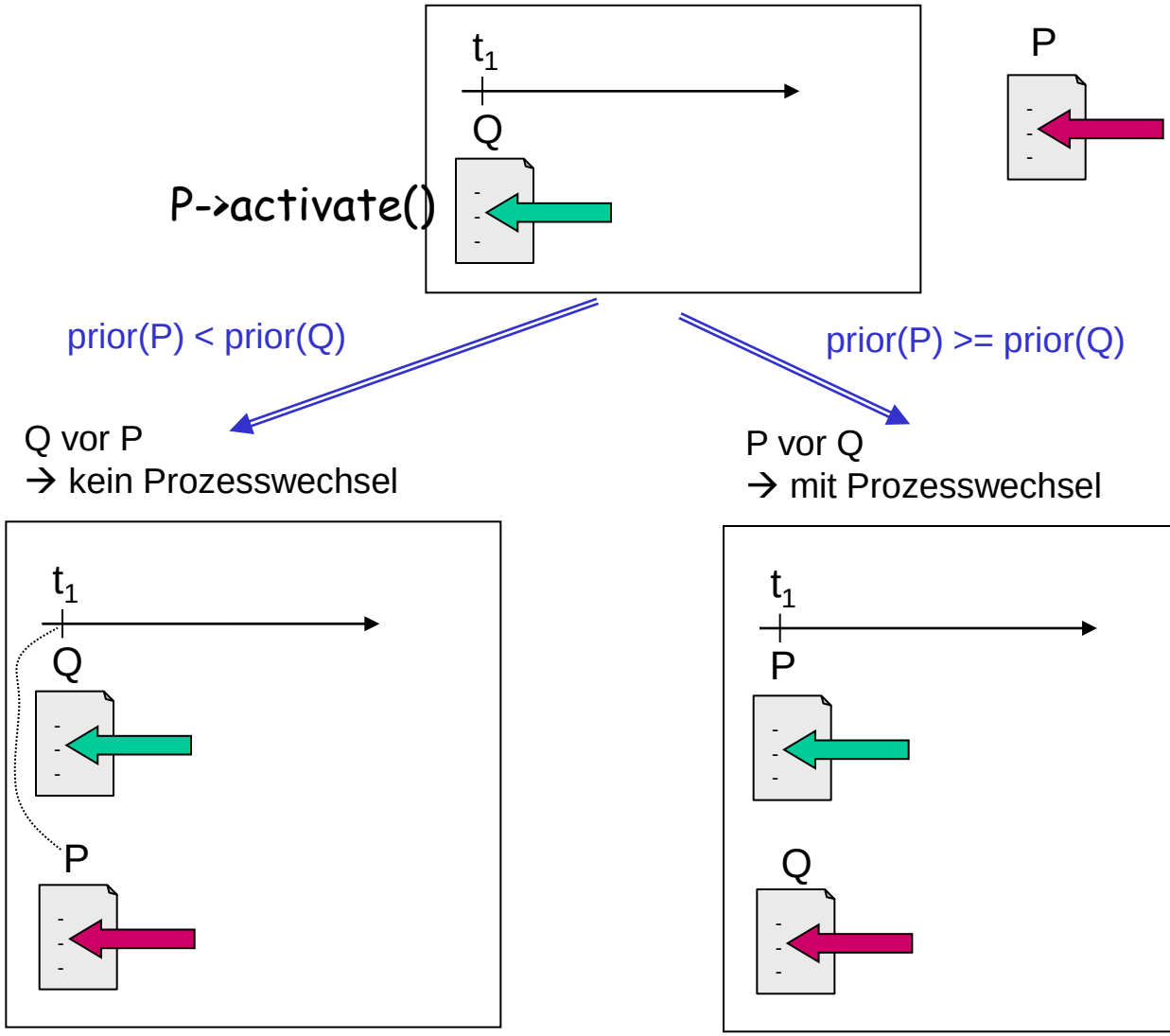
Prozessaktivierungen nach dem LIFO-Prinzip

```
void activate();      // Eintrag in ExL zur aktuellen Ereigniszeit now
                      // nach dem LIFO-Prinzip
                      // Prozesswechsel (falls kein Prioritätskonflikt)

void activateIn (SimTime t);
                      // Eintrag in ExL zur Ereigniszeit now + t
                      // nach dem LIFO-Prinzip bei Gleichzeitigkeit und
                      // Prioritätsgleichheit
                      // falls t<0.0, dann t= 0.0

void activateAt (SimTime t);
                      // Eintrag in ExL zur absoluten Ereigniszeit t
                      // nach dem LIFO-Prinzip bei Gleichzeitigkeit und
                      // Prioritätsgleichheit
                      // falls t<now, dann t= now
```

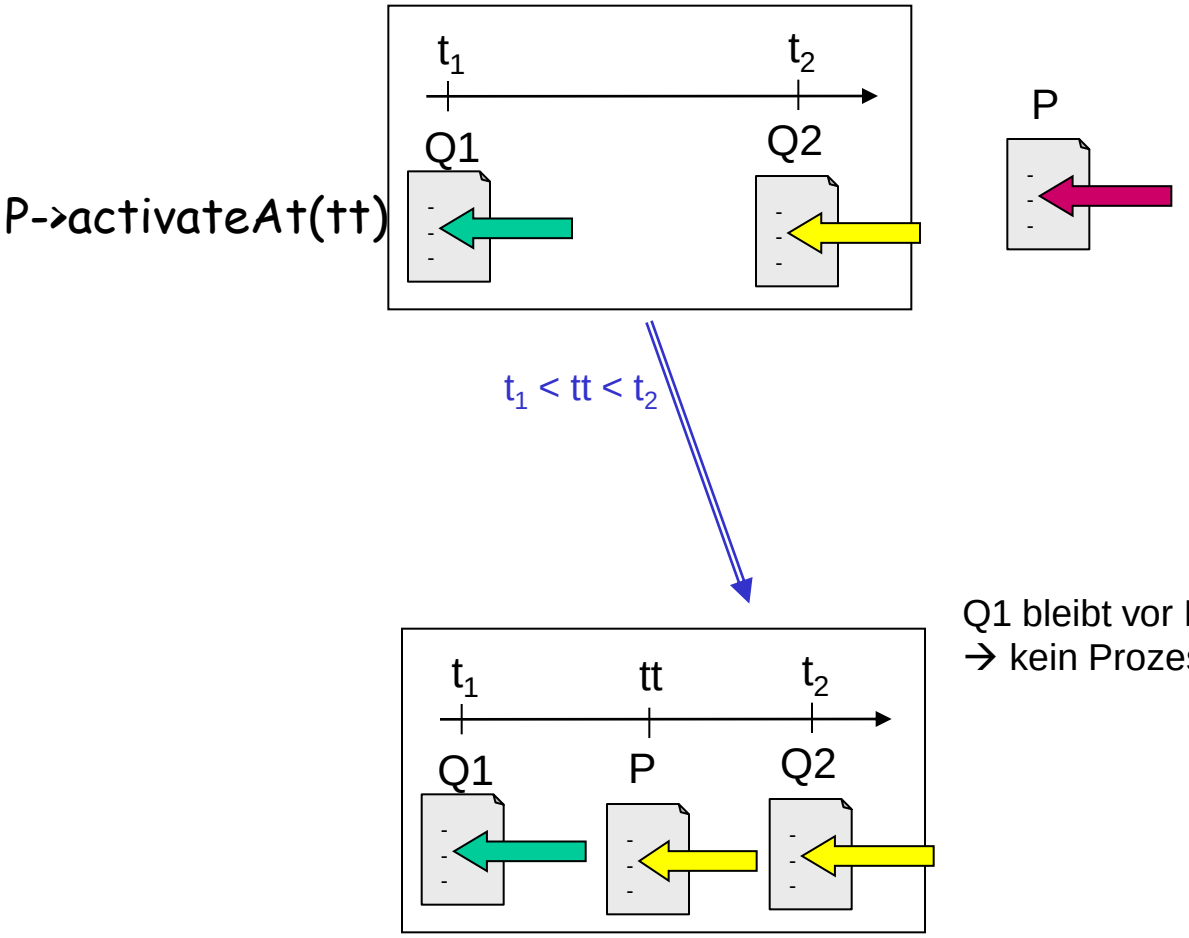
Achtung:
nur aus einem Simulationskontext heraus,
nicht bei Aktivierung aus dem Hauptprogramm

semantisch äquivalent:
P->activate() == P->activateIn(0.0) == P->activateAt(now)

Activate innerhalb eines Simulationskontextes



Activate innerhalb eines Simulationskontextes

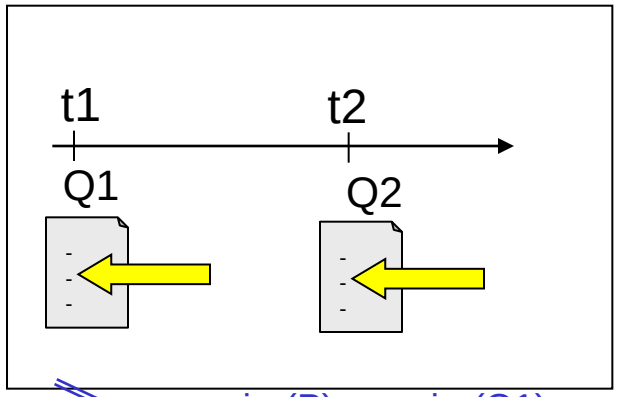


Q1 bleibt vor P
→ kein Prozesswechsel

Activate außerhalb eines Simulationskontextes

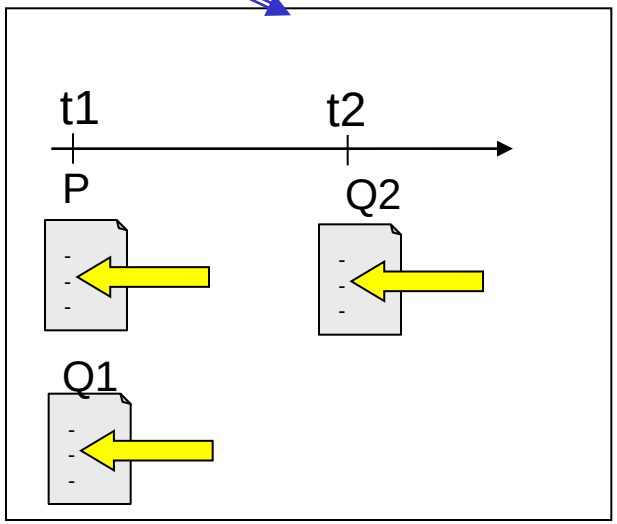
Simulationskontext (DefaultSimulation-Objekt)

```
int main ( ...) {
P->activate()
...
}
```



prior(P) >= prior(Q1)

```
int main ( ...) {
... P->activate()
..
}
```



trotzdem
noch kein
Prozesswechsel !
Hauptprogramm
setzt Ausführung fort

Process: Scheduling-Operationen (2)

Prozessaktivierungen nach dem Vorher/- Nachherprinzip

```
void activateBefore (Process* p);
                    // unmittelbarer Eintrag vor p mit Ereigniszeit von p,
                    // ggf. Übernahme der Priorität von p
                    // falls p == this: leere Anweisung
                    // falls p nicht in der ExL: Fehlermeldung

void activateAfter (Process* p);
                    // unmittelbarer Eintrag nach p mit Ereigniszeit von
                    // p, ggf. Übernahme der Priorität von p
                    // falls p == this: leere Anweisung
                    // falls p nicht in der ExL: Fehlermeldung
```

Process: Scheduling-Operationen (3)

Prozessverzögerungen nach dem FIFO-Prinzip

```
void hold();
                    // Eintrag zur aktuellen Ereigniszeit
                    // als letzter bei gleicher oder niedrigerer Priorität
    (FIFO)

void holdFor (SimTime t);
                    // Eintrag zur Ereigniszeit now + t
                    // als letzter bei gleicher oder niedrigerer Priorität
                    // falls t<0.0, dann t= 0.0

void holdUntil (SimTime t);
                    // Eintrag zur absoluten Ereigniszeit t
                    // als letzter bei gleicher oder niedrigerer Priorität
                    // falls t<now, dann t= now
```

semantisch äquivalent:

```
p->hold() == p->holdFor(0.0) == p->holdUntil(now)
p->holdUntil(t) == p->holdFor(t-now)
```

Process: Scheduling-Operationen (4)

Prozessunterbrechungen

```
void sleep();
        // Entfernung von currentProcess()  / runable-Prozess aus der ExL
        // Zustandswechsel in idle, Ereigniszeit 0.0
        // Aktivierung des ersten ExL-Eintrages
        //          falls Process, dann auch Prozesswechsel )
        //          falls ExL leer, dann Rückkehr ins Hauptprogramm

virtual void interrupt();
        // runable-Prozess wird in seiner hold/activate-Phase unterbrochen
        // wird evtl. zum neuen Current-Prozess (aus Zukunft zurückgeholt), falls kein
        // Prioritätskonflikt
        // und kann mit getInterrupter() die erfolgte Unterbrechung erkennen und
        // selbst behandeln

void cancel();
        // Prozessabbruch, Entfernung aus der ExL
        // Zustandswechsel in terminated
        // erneute Aktivierung führt zum Fehler
```

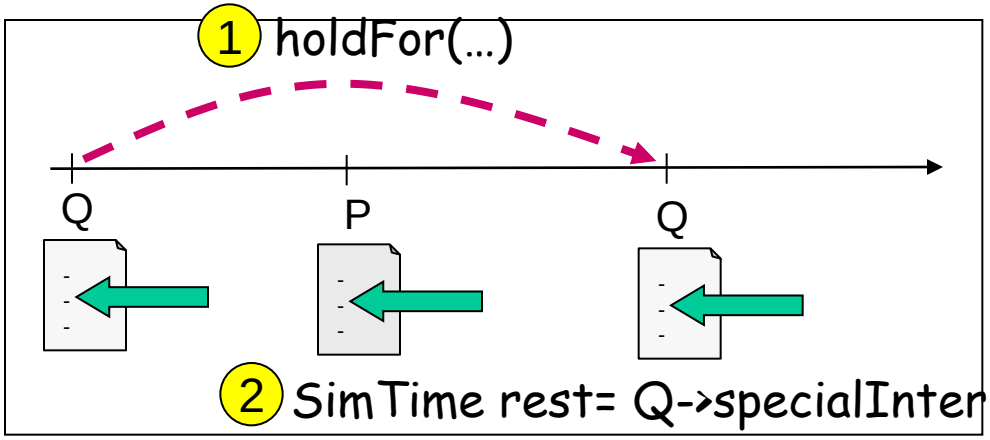
geplante Ereigniszeit des zu unterbrechenden Prozesses

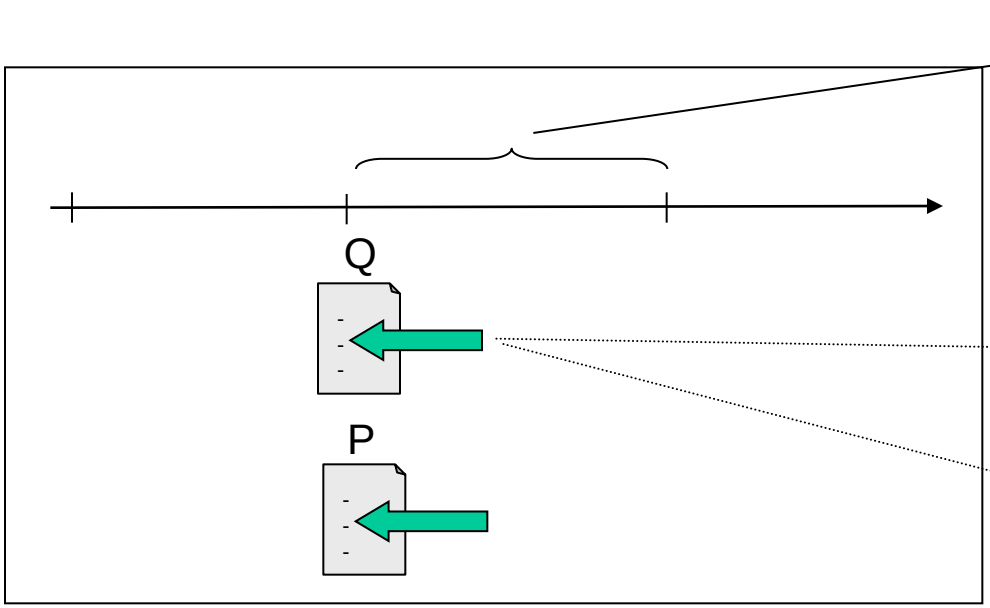
Kaskadierung von interrupt könnte die ausstehende Restzeit ermitteln:

```
SimTime specialInterrupt() {
        SimTime t= getExecutionTime() – getSimulation()->getTime();
        interrupt();
        return t;
}
```

Unterbrechungszeitpunkt, (aktuelle Modellzeit des interrupt-Rufers)

Process: Interrupt-Mechanismus (1)





Process: Interrupt-Mechanismus (2)

Unterbrechungsbehandlung

```
bool isInterrupted() const {return interrupted;}
            // Abfrage eines Interrupt-Zustandes (nach erfolgtem interrupt)
            // true, falls Unterbrechung erfolgte und noch keine Verzögerung
            // stattgefunden hat


Sched* getInterrupter() const {return interrupter;}
            // Anzeige des Prozesses/Ereignisses, der/das interrupt() gerufen
    hat
            // falls isInterrupted() == true und
            //        getInterrupter()==0: dann war Interrupter der
            //  Simulationskontext


void resetInterrupt() {interrupted=false; interrupter=0;}
            // löscht Interrupt-Zustandseinträge
            // implizit bei jeder Scheduling-Operation
```

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Process: Lebenslauf und Rückgabewert

Process-Verhaltensfunktion

```
protected:
    virtual int main() = 0;
```

… bei Terminierung mittels return

```
bool hasReturned() const {return validReturn;};
            // Test auf Beendigung

int getReturnValue() const;
            // liefert Rückgabewert (ohne evtl. Blockierung des Rufers)
            // Warnung für Nutzung eines ungültigen Wertes
            // vorheriger Test mit hasReturned()
```

Bedingungen zur Prozessauswahl

```
class Process : …. {
    public:
        // Funktionstypen zur Codierung von Bedingungen für Prozessauswahl
        typedef bool (Process::*Selection)(Process* partner);
        typedef bool (Process::*Condition)();

        // Prozessgrundzustand
        enum ProcessState {CREATED, CURRENT, RUNNABLE,
                           IDLE, TERMINATED                    };

        Process (Simulation* s, Label l, ProcessObserver* o = 0);
        ~Process();

        ProcessState getProcessState() const;
```

- Funktionstyp heißt Selection,
- Wert einer Variable oder eines Parameters **s** von diesem Typ muss eine Adresse einer Memberfunktion einer Prozess-Ableitung sein mit der Signatur (Process*):bool
- **Beispiel:** Selection s = & processSpecial::mF
  ein Aufruf erfolgt mittels (p->*s) (aktuellerPartner)

Anwendung der Funktionstypen in ODEMx

… der Funktionstypen

- Selection
- Condition

Member-Funktionen von Process-Ableitungen

dieser Funktionstypen werden durch konkrete Anwendungen festgelegt:

- Überprüfung von globalen Bedingungen (Condition) in Form von Zustandsereignissen
- Überprüfung von Eigenschaften eines Prozesses (Selection)

➔ Result vom Typ Boolean

Der Aufruf dieser Member-Funktionen der Anwendung soll aber

- im Funktionskörper von Funktionen der ODEMx-Bibliothek erfolgen,
  obwohl sie die konkreten Memberfunktionen nicht kennen können
  (Nutzung von Funktionsvariablen)

Klassendefinition (Auszug)

Private Member-Variablen

```
private:
        ProcessState processState;  //process state
        Priority p;                 // process priority
        SimTime t;                  // process execution time
        Simulation* env;            // process simulation
        int returnValue;            // return value of main()
        ProcessQueue* q;            // pointer to queue if process is waiting
        SimTime q_int;              // enqueue-time
        SimTime q_out;              // dequeue-time
        bool validReturn;           // return value is valid
        bool interrupted;           // Process was interrupted
        Process* interrupter;       // Process was interrupted by
                                    // interrupter
                                    // (0 -> by Simulationkontext)
```

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue
6. Spezielles Process-Scheduling (Memory)

Lokalisierung eines Prozesses

ein beliebiger Prozess kann zu einem Zeitpunkt in verschiedenen Listen erfasst sein:

- in höchstem einem Terminkalender (ExL) (seines Simulationskontextes, falls aktiviert)

list <sched*>

überflüssig

- in genau einer der vier Prozesszustandslisten seines Simulationskontextes

list <process*>

- in genau einer Warteschlange vom Typ ProcessQueue (oder Ableitung)

list <process*>

- in beliebig vielen Memory-Listen der Typen PortTail, PortHead, Timer, WaitCondition (oder Ableitungen)

list <process*>

Modul Synchronistation definiert Queue als ProcessQueue-Ableitung, ProcessQueue-Objekte werden insbesondere zur Erfassung (inkl. Statistik) von blockierten Prozessen in vordefinierten Synchronisationsklassen (Bin, Res, Waitq, Condq, …) benutzt

list <process*>

ProcessQueue

- **sortierte Liste** von Prozessen (Zeigern)
  in Form nutzergesteuerter Halbordnungen
  - DefaultOrder: sortiert nach Ausführungszeiten
    (u. bei Gleichzeitigkeit nach Priorität) *überflüssig*
  - PriorityOrder: sortiert nur nach Priorität
  - zusätzlicher LIFO/FIFO- Auswahlparameter
    (ähnlich zum Konzept des ExL)
- **Anwendung**
  - Spezialisierung zur Klasse Queue
    im Modul Synchronisation zur Verwaltung schlafender/blockierter Prozesse (IDLE-Zustand)
  - nutzereigene Listen: nächstes Beispiel
- **Funktionen** (Varianten) zur Reaktivierung blockierter Prozesse
  - awake

ProcessQueue

Member-Funktionen

```
// Zugriffsmethoden
        Process* getTop() const;
        const std::list<Process*>& getList() const;

        bool isEmpty() const;

        unsigned int getLength() const {return (unsigned int)l.size();}

// Manipulationsmethoden
        virtual void popQueue(); // entfernt getTop()
        virtual void remove(Process* p);
        virtual void inSort(Process* p,  bool fifo = true);
```

ProcessQueue

- Ein Prozess kann zu einem Zeitpunkt immer nur in einer ProcessQueue enthalten sein

- Fehlbenutzung
  - bei Eintrag in zweite ProcessQueue:
    (ohne aus der ersten entfernt worden zu sein)
    Abbruch mit Fehlermeldung

- Prioritätsänderung
  - verursacht automatische Positionsänderung in der jeweiligen ProcessQueue

ProcessQueue

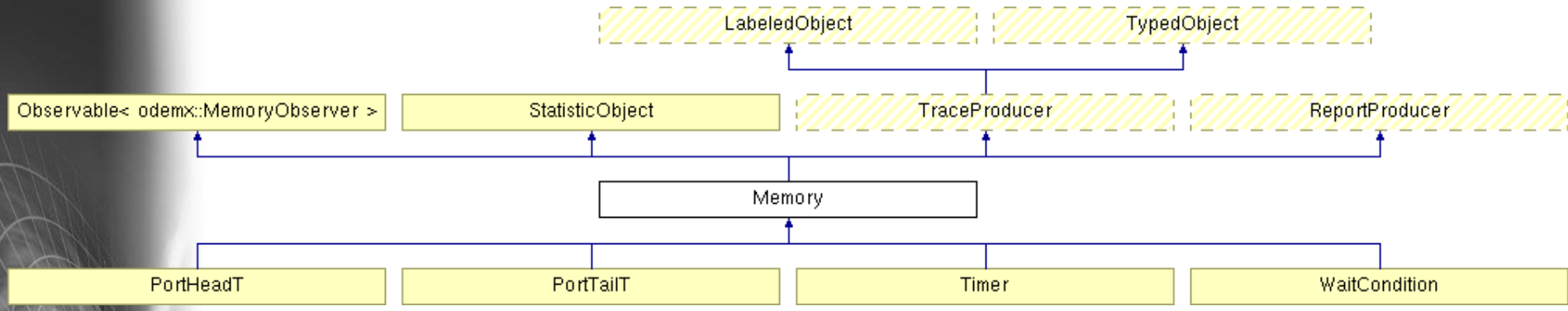
Process-Warteschlange (Benutzung im Modul Synchronisation)

```
ProcessQueue* getQueue() const {return q;}
            // liefert Zeiger zur Warteschlange, in der sich der Prozess
            // befindet

SimTime getEnqueueTime() const {return q_int;}
            // liefert Eintrittszeit in die Warteschlange, in der sich der Prozess
            // befindet

SimTime getDequeueTime() const {return q_out;}
            // liefert Zeit des Verlassens der Warteschlange, in der sich der
            // Prozess befand
```

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue
6. Spezielles Process-Scheduling (Memory)

Klasse Memory



abstrakte Basisklasse
zur Erfassung blockierter/schlafender Prozesse,
die unter bestimmten Bedingungen wieder zu aktivieren sind:

- Timer (nach Ablauf einer Zeitspanne)
- allgemeine Bedingung (Erfüllung der Bedingung)
- Nachrichten-Eingabewarteschlange (Vorhandensein einer Nachricht)
- Nachrichtenausgabewarteschlange (vorhandener Platz in der Warteschlange)
- Prozess (erfolgreiche Beendigung eines Prozesses)
- …

Empfohlene Anwendung: wait()-Funktion.

Motivation

typisches Synchronisationsproblem

- Prozesse (z.B. Zustandsmaschinen) setzen ihren Lebenslauf (Zustandsübergänge) nur unter bestimmten Bedingungen fort:
  - in einem von mehreren Eingangspuffern wurde eine Nachricht/Ereignis/Anforderung hinterlegt
  - ein oder mehrere Zustandsereignisse sind eingetreten
  - ein oder mehrere Zeitereignisse (ausgelöst durch Wecksignale von Uhren) sind eingetreten

- dabei kann genau ein Prozess oder aber auch mehrere betroffen sein, wobei auch eine selektive Auswahl der Prozesse modellierbar sein sollte

Im Lebenslauf eines Prozesses

```
...
result= wait (buffer1, buffer2, timer1, cond1);
switch (result->getType() ) {
   case TIMER: ...
   case PORTHEAD: ...
   case CONDITION
   default: ...
}
...
```

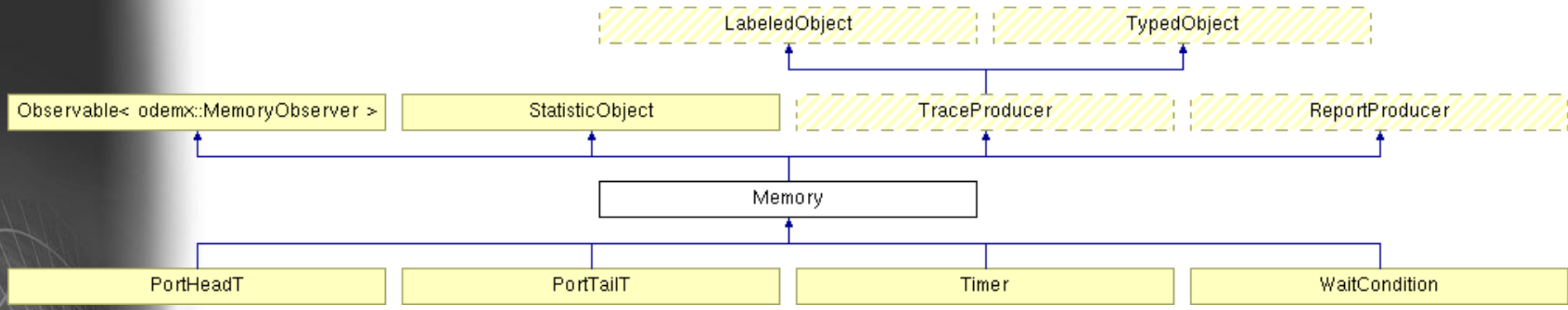
Aufrufer müsste sich jeweils registrieren bei buffer1, buffer2, timer1, cond1

bei diesen Objekten sollten sich weitere Prozesse/Ereignisse registrieren können

Objekte könnten für Aktivierung der wartenden Prozesse sorgen

Klasse Memory

Basisklasse für
- PortHeadT, PortTailT
- Timer, WaitCondition



Spezialisierungen legen Semantik von available fest

Protected Member-Functionen

Memory (Simulation *sim, MemoryType t, MemoryObserver *mo=0)
virtual ~Memory ()
virtual void alert () //reaktiviert vermerkte Sched-Einträge
virtual Trace * getTrace () const
bool remember (Sched *newObject)
bool forget (Sched *rememberedObject)
void eraseMemory () //löscht vermerkte Sched-Einträge

} Verwaltung von memoryList

Statusinformation eines Memo-Objektes

virtual bool available ()=0 // testet die Bereitschaft des Memo-Objektes.
MemoType getMemoryType () // liefert den Typ.

Protected Attribute

std::list< Sched * > memoryList //Liste vermerkter Sched-Objekte (Zeiger)

```
enum MemoryType {
    TIMER,
    PORTHEAD,
    PORTTAIL,
    CONDITION
};
```

Beispiel: Autofähre

- Wortmodell und informale Darstellung