Kurs OMSI
im WiSe 2014/15

Objektorientierte Simulation mit ODEMx

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

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

6. ODEMx-Modul Synchronisation:

WaitQ, CondQ

- Konzept WaitQ

  Beispiel: Tankerflotte / Hafen / Raffinerie

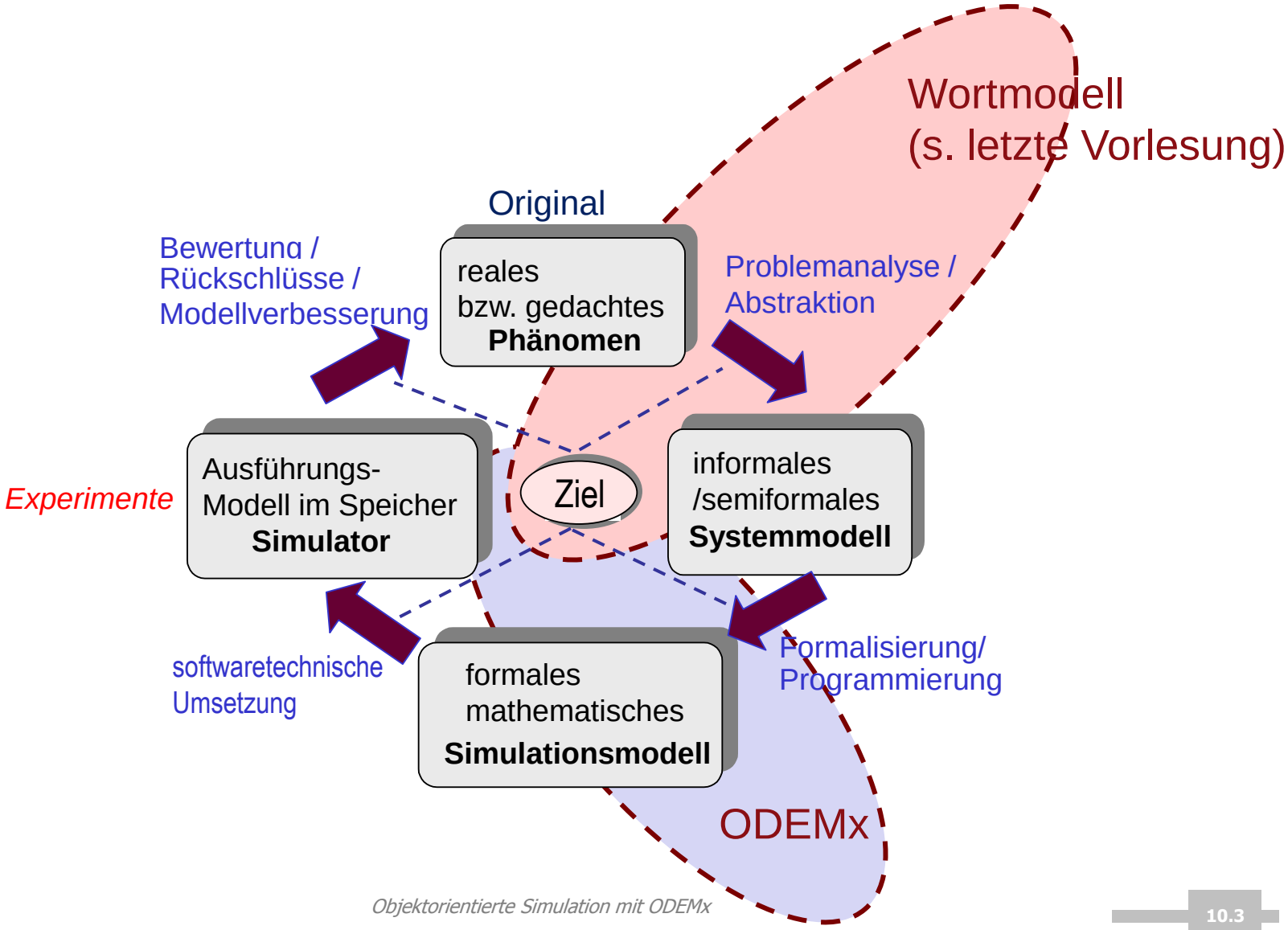
- Konzept CondQ

  Beispiel: Hafen / Schlepper / Gezeiten

- Weitere Anwendungsbeispiele für WaitQ u. CondQ

- Zusammenfassung/einheitliche Betrachtung

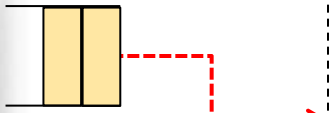
Erinnerung: Vorgehensweise bei der Systemsimulation



Beispiel: Tanker – Tank – Raffinerie

Ladung: Gleichverteilung
15tb, 20tb, 25tb

Zwischenankunftszeit:
n.-exponential verteilt



(Kap 70 tb)

T5
T4
T3
T2
T1

Pumprate
1 tb/h

4 tb/h

Raffinerie

Bed.2: Ende der Tank-Befüllung
- ist Tank nahezu voll (≈70%), wird Öl zur Raffinerie abgepumpt
- Beladung wird gestoppt

Bed. 1: Tankauswahl
(1) Tank,
der die längste Zeit leer war
und
(2) dessen frei gebliebene
Kapazität die Schiffsladung
komplett übernehmen kann
(3) konstante Vorbereitungszeit: 0,5h

Ziel:
1000 h Simulation, bei besonderer Ausgangssituation:
(1) Füllstand der Tankbehälter
- zwei sind leer (70tb frei)
- einer wird in 8h leer (70tb frei)
- einer wird in 12 h mit Befüllung fertig (45tb frei)
- einer wird in 3.5h mit Befüllung fertig (25tb)

(2) erste Tankerankunft: 0.0

Beispiel zeigt typisches Problem

- Prozess-Objekte kooperieren zeitweilig miteinander
  (sogar starke Kopplung, da Rückkopplung)

  - hier: Tanker ←→ Tankbehälter
    Tankbehälter ← → Raffinerie
    (falls Raffinerie als eigenständiges Objekt)

  - Zustandsänderung des einen Objektes (Tanker-Inhalt) ist abhängig von Zustandsänderung des anderen Objektes (Tank-Inhalt) und umgekehrt

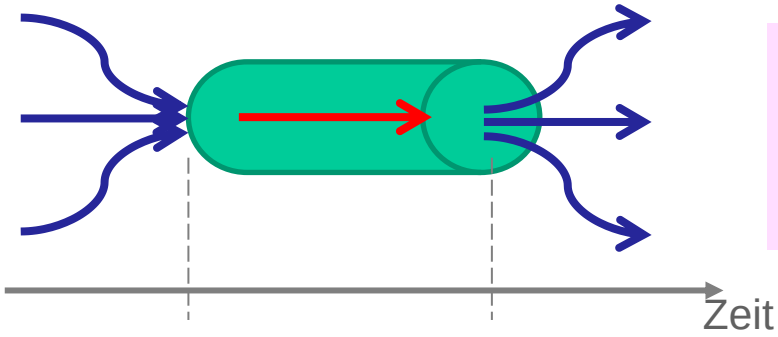
- Daraus entsteht ein grundsätzliches Synchronisationsproblem:
  bei quasiparalleler Ausführung von n Prozessen in ihrer Kooperationsphase

- Ein Prozess-Objekt kann in seiner Lebenszeit verschiedene zeitweilige Kooperationsbeziehungen eingehen

Nützliches Modellierungsmuster

- n (>=2) Prozesse kooperieren ab einem Zeitpunkt für eine gewisse Dauer
- **Bed**.: (1) Zum Startzeitpunkt der Kooperation sind alle n Prozesse
  verfügbar/für die Kooperation bereit
  falls nicht, müssen die bereits verfügbaren auf die anderen warten

  (2) Zustandsänderungen der Prozesse sind voneinander abhängig



Entschärfung der Parallelität
der synchronen Wechselwirkungen
bei Zustandsänderungen
im Simulator

die Slave-Prozesse sind in der Kooperationsphase passiv

- Effiziente simulative Umsetzung auf einer Ein-Prozessor-Maschine
  - einer der n Prozesse übernimmt als **Master** (aktiv) die Ausführung der Zustandsänderungen sämtlicher Prozesse in Abhängigkeit der Modellzeit
  - alle anderen n-1 Prozesse warten als **Slave** (passiv) auf die Beendigung der Kooperation durch den **Master**

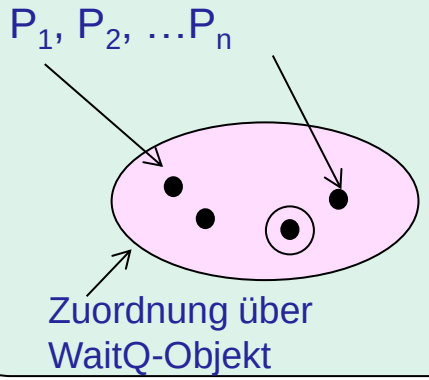
ACHTUNG: Master und Slave sind nur Rollen, die Prozesse zeitweilig spielen

Motivation der ODEMx-Klasse WaitQ

- Realisierung des Modellierungsmusters „Master-Slave“ zur Prozesssynchronisation
  bei Auszeichnung dynamischer **Rollen** zur Erbringung einer gemeinsamen Kooperationsleistung

  – **Rolle** Slave :
  sind spezielle Ressourcen/Kooperationspartner zugeordneter Master-Prozesse zur Realisierung der gemeinsamen (zeitlich befristeten) Kooperationsleistung

  – **Rolle** Master :
  sind **alleinige** Erbringer dieser Kooperationsleistung, Slave-Partner bleiben während der Kooperation passiv, müssen nach Abschluss der Kooperation vom Master aktiviert werden



Master-Rolle, Slave-Rolle

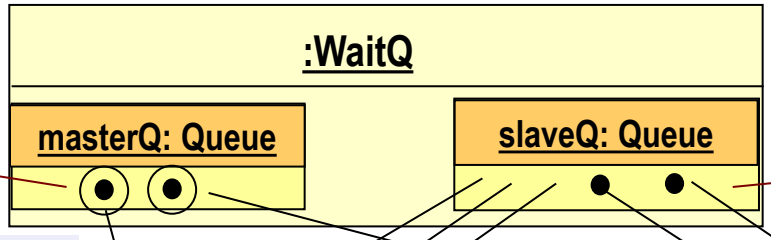
temporäres Prozessensemble
bei Auszeichnung eines Prozesses als
Stellvertreter (Master) zur Realisierung wechselseitig abhängiger Zustandsänderungen

WaitQ-Konzept

Synchronisationsklasse

zur Erfassung von Prozessen und Bildung zeitweiliger Kooperationsgemeinschaften mit unterbrechbarem Warten auf das Zustandekommen der Kooperation, falls Kooperationspartner momentan nicht zur Verfügung stehen

- jeweils **einem** Master lassen sich beliebig **viele** Slave-Prozesse zuordnen



ungebundene (wartende) Master-Prozesse

ungebundene (wartende) Slave-Prozesse

gebundener Master verwaltet selbstständig temporär ausgewählte Slaves

gebundene Master-Slave-Ensemble (nur der Master ist aktiv)

ein Slave ist zu einem Zeitpunkt höchstem einem Master zugeordnet

Zusammenfassung: WaitQ-Anforderungen

1. über ein WaitQ-Objekt sollen sich gleichzeitig oder nacheinander **beliebig viele** temporäre Master-Slave-Ensemble bilden können

2. **einem** Master lassen sich beliebig **viele** Slave-Prozesse zuordnen
   - Ist die **Zuordnung komplett erfolgt**,
     sind im WaitQ-Objekt weder Master, noch seine erwählten Slaves weiter erfasst
     Allein der Master eines kompletten Ensembles ist aktiv, die erwählten Slaves bleiben passiv.
   - **Master** bestimmt **allein** die Realisierung und Dauer der Kooperationsleistung (benötigt dafür entsprechende Zugriffsrechte für seine Slaves) und gibt danach die Slaves per activate wieder frei

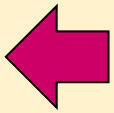
3. folgende Teilaktivitäten bei Nutzung eines WaitQ-Objektes sollen extern (z.B. Timer) vorzeitig **unterbrechbar** sein:
   - Warten eines Prozesses als Master auf die Verfügbarkeit eines Slaves
   - Warten eines Prozesses als Slave auf die Verfügbarkeit eines Masters
   - Erbringung der laufenden Kooperationsleistung durch den Master

Bagger-Beispiel

Unterbrechung erfolgt durch Anwendung von interrupt
~ läuft auf ein activateAt zur aktuellen Zeit hinaus
bei vorherigem Setzen des Flags interupted

Zusammenfassung: WaitQ-Anforderungen

4. ein Master sollte über ein waitQ-Objekt seine Slaves mit bestimmten Eigenschaften bestimmen dürfen
   - bestimmter Prozesstyp (abgeleitete Klasse)
   - bestimmte Attribut-Belegungen (Zustand)



Ölhafen-Beispiel

5. ein Master sollte über ein waitQ-Objekt die Verfügbarkeit eines Slaves mit bestimmten Eigenschaften vorab (ohne Blockierung) prüfen können
   - bestimmter Prozesstyp (abgeleitete Klasse)
   - bestimmte Attribut-Belegungen (Zustand)

6. Umgekehrt sollte die Reihenfolge des Erweckens der Master durch Slaves beeinflussbar sein

WaitQ- Member-Funktionen

Konstruktor / Destruktur

```
                    WaitQ (base::Simulation &sim, const data::Label &label, WaitQObserver *obs=0)
                                        // Construction for user-defined Simulation.
                    ~WaitQ ()
                                        // Destruction.
const base::ProcessList &   getWaitingSlaves () const
                                        // List of blocked slaves
const base::ProcessList &   getWaitingMasters () const
                                        // List of blocked masters.
// Master-slave synchronisation
                bool    wait ()
                                        // Wait for activation by a 'master' process.
                bool    wait (base::Weight weightFct)
                                        // Wait for activation by a 'master' process.
        base::Process *     coopt (base::Selection sel=0)
                                        // Get a 'slave' process
                    coopt (base::Weight weightFct)
                                        // Get a 'slave' process by evaluating a weight function.
        base::Process *     avail (base::Selection sel=0)
                                        // Get available slaves without blocking (optional: select slave)
                void    signal ()
                    //reactivate all master for rechecking of modified selection or weight conditions
```

aktiviert den am längsten wartenden Master (also nicht alle wartenden!)

Aufrufer wird zum Slave

aktiviert den Master, für den die Gewichtsfunktion den Maximalwert liefert

WaitQ- Member-Funktionen

```
                    WaitQ (base::Simulation &sim, const data::Label &label, WaitQObserver *obs=0)
                                        // Construction for user-defined Simulation.
                    ~WaitQ ()
                                        // Destruction.
const base::ProcessList &   getWaitingSlaves () const
                                        // List of blocked slaves.
const base::ProcessList &   getWaitingMasters () const
                                        // List of blocked masters.
// Master-slave synchronisation
                bool    wait ()
                                        // Wait for activation by a 'master' process.
                bool    wait (base::Weight weightFct)
                                        // Wait for activation by a 'master' process.
    base::Process *     coopt (base::Selection sel=0)
                                        // Get a 'slave' process
                    coopt (base::Weight weightFct)
                                        // Get a 'slave' process by evaluating a weight function.
    base::Process *     avail (base::Selection sel=0)
                                        // Get available slaves wit
                void    signal ()
                    //reactivate all master for rechecking of modified selection or weight conditions
```

wählt den ersten Slave, für den die Auswahlfunktion des rufenden Masters wahr ist
bzw. den, der am längsten gewartet hat

Aufrufer wird zum Master

wählt unter allen Slaves denjenigen, für den die Gewichtsfunktion den Maximalwert liefert

WaitQ- Member-Funktionen

```
                    WaitQ (base::Simulation &sim, const data::Label &label, WaitQObserver *obs=0)
                                        // Construction for user-defined Simulation.
                    ~WaitQ ()
                                        // Destruction.
const base::ProcessList &   getWaitingSlaves () const
                                        // List of blocked slaves.
const base::ProcessList &   getWaitingMasters () const
                                        // List of blocked masters.
// Master-slave synchronisation
                    bool    wait ()
                                        // Wait for activation by a '
                    bool    wait (base::Weight weightFct)
                                        // Wait for activation by a 'master' process.
          base::Process *   coopt (base::Selection sel=0)
                                        // Get a 'slave' process
                    coopt (base::Weight weightFct)
                                        // Get a 'slave' process by evaluating a weight function.
          base::Process *   avail (base::Selection sel=0)
                                        // Get available slaves without blocking (optional: select slave)
                    void    signal ()
                    //reactivate all master for rechecking of modified selection or weight conditions
```

prüft lediglich Slave-Verfügbarkeit ohne Blockierung oder Bindung

Process: Klassendefinition (Auszug), Wdh.

Private Member-Variablen

```
private:
        ProcessState processState;   //process state
        Priority p;                  // process priority
        SimTime t;                   // process execution time
        Simulation* env;             // simulation context
        …
        bool interrupted;            // Process was interrupted
        Process* interrupter;        // Process was interrupted by interrupter
                                     // (0 -> by Simulationkontext)
```

Process: Interrupt-Mechanismus (Wdh.)

Unterbrechungsbehandlung

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

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
```

Bedingungen zur Prozessauswahl

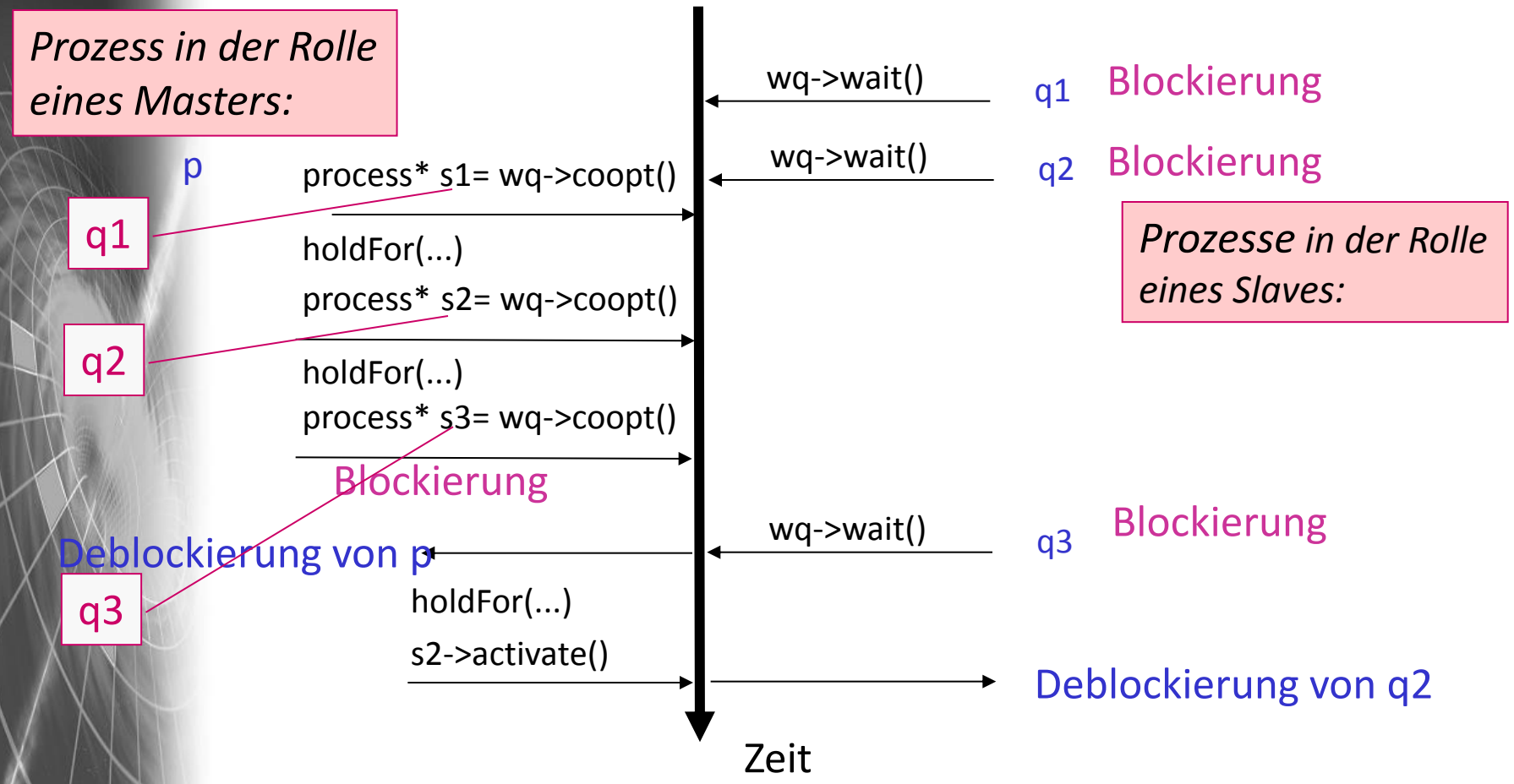
```
class Process : .... {
    public:
        // Funktionstypen zur Codierung von Bedingungen für Prozessauswahl
        typedef bool (Process::*Selection)(Process* partner);
        typedef bool (Process::*Condition)();

        // 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 vom Typ Selection muss eine Adresse einer Memberfunktion (hier: mF) einer Prozess-Ableitung (hier: processSpecial) sein mit der Signatur (Process*):bool

- **Beispiel:** Selection s = & processSpecial::mF
  ein Aufruf erfolgt mittels (p->*s) (aktuellerPartner)

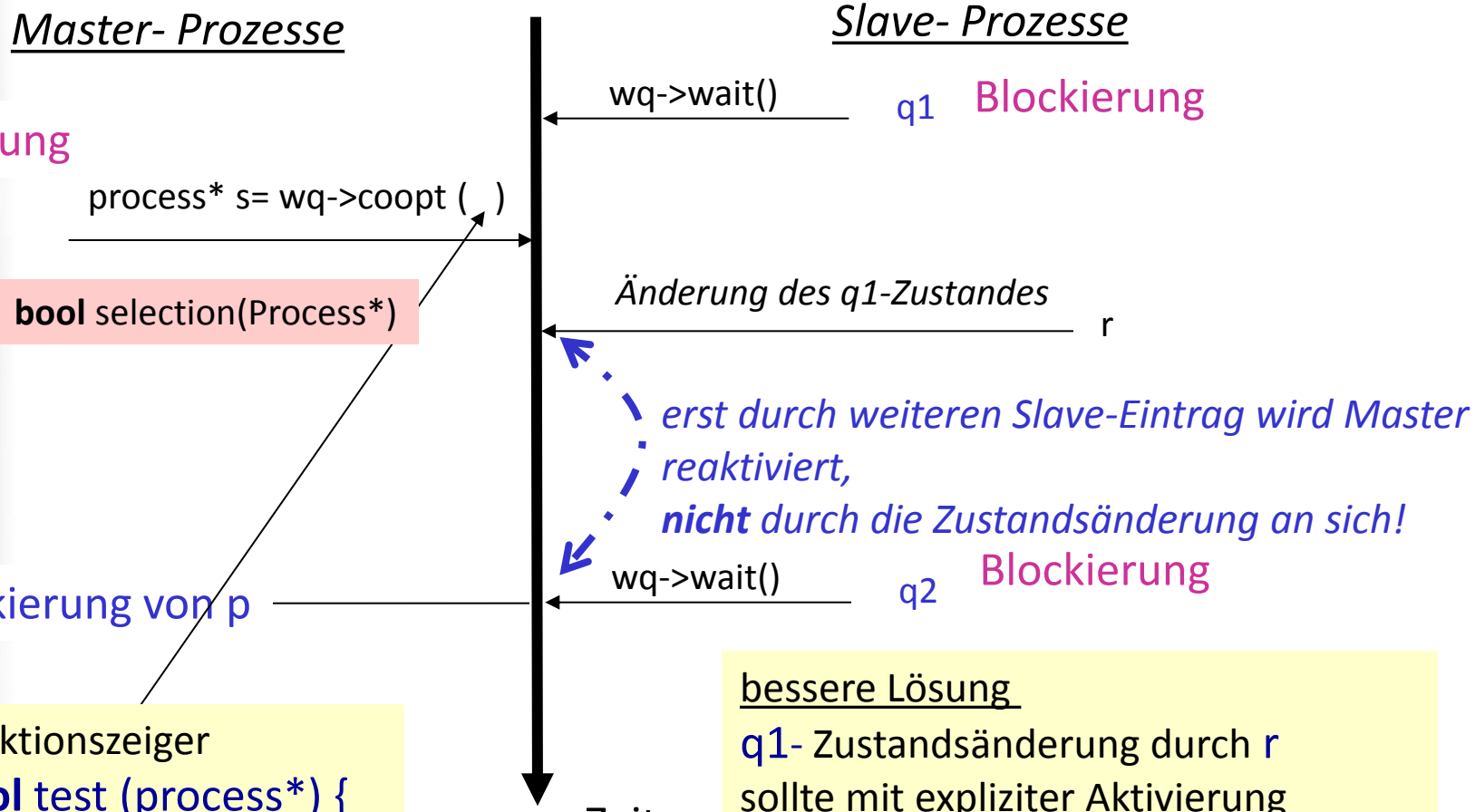
WaitQ-Synchronisation

```
Process *p, *q1, *q2, *q3;      // Zeiger auf Prozessobjekte
WaitQ *wq;
```



WaitQ-Synchronisation: coopt(&processSpecial::test)

```
Process *p, *q1, *q2, *r;      // Zeiger auf Prozessobjekte
WaitQ *wq;
```



Funktionszeiger

```
bool test (process*) {
    return q1->x > wert;
}
```

bessere Lösung

q1- Zustandsänderung durch r sollte mit expliziter Aktivierung der blockierten Masterprozesse in wq verbunden sein: wq->signal()

Unterbrechung *wartender Master- bzw. Slave-Prozesse*

- das evtl. Warten auf den Partner-Prozess
  kann sowohl beim Master- als auch beim Slave-Prozess mittels interrupt abgebrochen werden:

  in diesem Fall liefert

  – coopt() einen NULL-Zeiger

  – wait() den Wert **false**

  ACHTUNG:

  ein Master sollte jedoch seinen erwählten Slave nicht per interrupt() aktivieren !!! (sondern per activate())

  nur dann liefert wait() den Wert **true**

Unterbrechung *der Kooperation von Master- und Slave-Prozessen*

- der Master befindet sich im Terminkalender (execution list), seine Ereigniszeit markiert einen Zwischen- oder Endzustand der Kooperation
  (i.d.R.: realisiert der Master hold_for()
- die jeweils per Coopt ermittelten Slaves bleiben passiv (blockiert)
- weder Master noch seine erwählten Slaves sind mehr in dem einst zusammenführenden WaitQ-Objekt registriert
- eine Unterbrechung der Kooperation kann in ODEMx nur durch Unterbrechung (interrupt) des Master eingeleitet werden,
  sowohl die Komplettierung der Unterbrechung, als auch die Unterbrechungsbehandlung muss im Aktionscode (main) des Masters umgesetzt werden

```
Process *s;
WaitQ wq;
…
// Vorbereitung der Kooperation
s= wq.coopt();
                    // evtl. Blockierung, Fortsetzung sobald s verfügbar
// Beginn der unterbrechbaren Kooperation ohne Unterbrechungsbehandlung
holdFor (dt);  //markiert das Ende der Kooperation
…                   // Kooperation: Aktionen zur Änderung der Attribute von Master und Slave
s->activate(); // Freigabe des slaves
// Ende der Kooperation
```

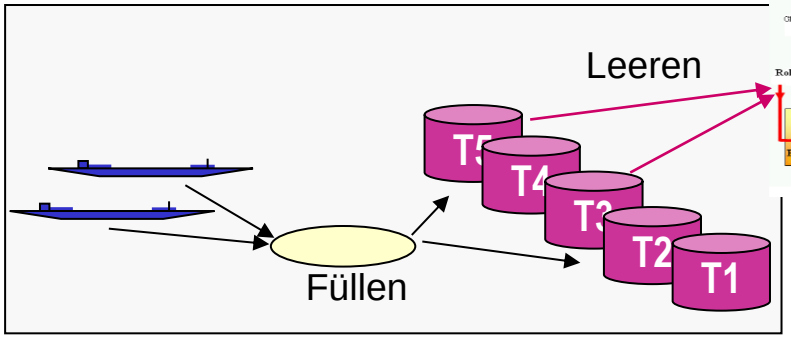
bei einer Unterbrechung wird
lediglich die Zeit der Kooperation reduziert,
die aktuellen Zustände bleiben unverändert

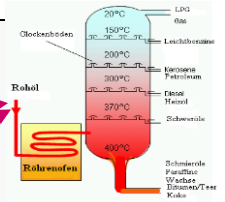
Unterbrechung *der Kooperation von Master- und Slave-Prozessen*

- Unterbrechungsbehandlung

```
Process *s;
WaitQ wq;
…
// Vorbereitung der Kooperation
s= wq.coopt();
                    // evtl. Blockierung, Fortsetzung sobald s verfügbar
// Beginn der unterbrechbaren Kooperation mit Unterbrechungsbehandlung
holdFor (dt);  //markiert das Ende der Kooperation
if (interrupted) {
                …                  // Aktionen zur  modifizierten Änderung der Attribute von Master
                                   // und Slave
                                   // bei Berücksichtigung der reduzierten Kooperationszeit und
                                   // der Unterbrechungsursache (Zeiger auf Unterbrecher-Prozess)
}
else {
                …                  // ursprüngliche Aktionen zur Änderung der Attribute von Master
                                   //und Slave
s->activate(); // Freigabe des slaves
// Ende der Kooperation sowohl für die volle als auch reduzierte Zeit
```

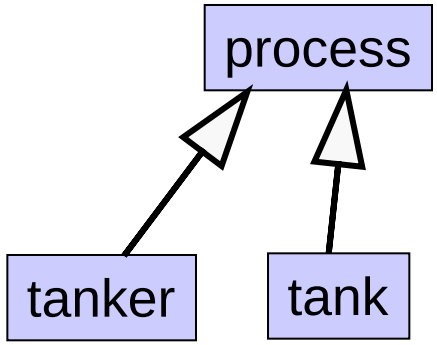
Informales Modell → Simulationsmodell



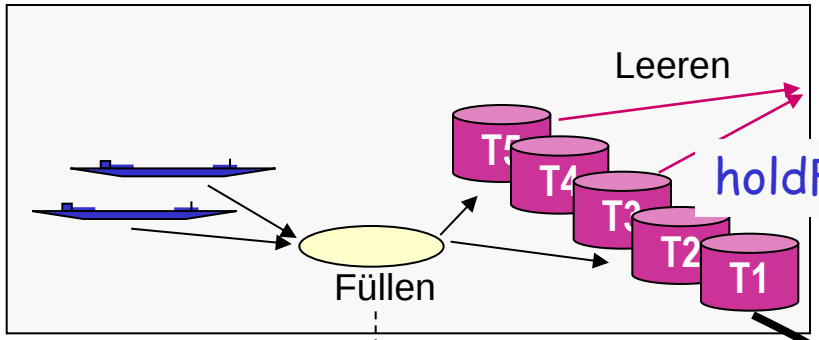


- Kooperationsaktivität: Füllen
  - Tanker und leerer Tank
- Kooperationsaktivität: Leeren
  - „voller“ Tank (und Raffinerie, die aber außerhalb des Systems liegt)
- Master/Slave-Prinzip beim Füllen
  - warum sollten Tanker die Master-Rolle übernehmen ?
  - haben Tank-Objekte nach Kriterien auszuwählen!

    ➔ Anwendung von coopt mit selection-Funtion



Informales Modell → Simulationsmodell



Prozess-terminierung nach Tank-aktivierung

Tank* t= tankQ-> coopt(?)

tankQ: WaitQ
:ProcessList
:ProcessList

tankQ->wait()

beladene Tanker

befüllbare Tankbehälter

Einsortierung
FIFO: Reihenfolge
nach der Wartedauer

Tankauswahl:
- (1) Tank,
  der die längste Zeit leer war
- (2) dessen frei gebliebene Kapazität die Schiffsladung komplett übernehmen kann

```
//Selektionsfunktion als Member-Fkt von Tanker
bool suitableTank (Process* partner) {
    Tank*  selected= dynamic_cast<Tank*> (partner);
    return selected->freeCap >= load;
};
```

freie Tankkapazität

Tankerladung

Umsetzung: Master-Slave-Synchronisation (1)

Tanker-Objekte als Master

```
class Tanker : public Process {
public:
      Tank *myTank; //Tank zur Entladung
      double load; //Fassungsvermoegen[tb]

      Tanker() : Process(sim, "Tanker"),
                  load (5.0*size->sample()){}
protected:
      int main();

//Selektionsfunktion
      bool suitableTank (Process* partner) {
                  ...
      };
};
```

Tank-Objekte als Slave

```
class Tank : public Process {
      double maxCap;   //max. Fassungsverm.
public:
      double freeCap; //akt. Freiraum[tb]

      Tank (double f) : Process(sim, "Tank"),
                  maxCap(70), freeCap(f) {}
protected:
      int main();
};
```

coopt- Implementierung iteriert über die Slave-Prozesse
und wendet auf jedes Element suitableTank(...) an

Umsetzung: Master-Slave-Synchronisation (2)

Tanker-Objekte als Master

```
int Tanker::main() {
    //Ankunft im Hafen

    //Auswahl des Tanks und Synchronisation
    myTank = dynamic_cast <Tank*>
            (tankq->coopt(
                (Selection) &Tanker::suitableTank));
    //Entladung
    holdFor(setuptime +
                load*tankerPumpRate);
    //Zeitverbrauch zur Entladung
    myTank->freeCap =
                myTank->freeCap - load;
    //Beendigung der Kopplung zum Tank
    myTank->holdFor();

    return 0;
}
```

Tank-Objekte als Slave

```
int Tank::main() {
  for (;;) {
    // Warten auf Synchronisation mit Tanker
    // bei anschl. Befuellung des Tanks
    // durch Tanker
    // bis weniger als 20 tb frei sind
    while (freeCap > 20.0) tankq->wait();

    // Entleerung des Tanks
    holdFor( (maxCap - freeCap) *
                        raffPumpRate);
    freeCap = maxCap;
  }
  return 0;
}
```

Umsetzung: Startsituation

Ziel:
1000 h Simulation, bei **besonderer Startsituation**:

(1) Füllstand der Tankbehälter
- zwei sind leer
- einer wird in 8h leer
- einer wird in 12 h voll (45tb)
- einer wird in 3.5h voll (25tb)

(2) erste Tankerankunft: 0.0

```
Tank *t;
    t = new Tank(70.0); t->hold();
    t = new Tank(70.0); t->hold();
    t = new Tank(45.0); t->holdUntil(12.0);
    t = new Tank(25.0); t->holdUntil(3.5);
    t = new Tank(70.0); t->holdUntil(8.0);
```

Aktionen können entweder
- im Hauptprogramm vor Start des Simulationskontextes

oder
- von einem Konfigurationsprozess übernommen werden, der wiederum vom Hauptprogramm zur Zeit 0 in den Ereigniskalender aufzunehmen ist

Report-File

Random Number Generators

Name	Reset at	Type	Uses	Seed	Parameter 1	Parameter 2	Parameter 3
nextTanker	0	Negexp	128	33427485	0.125	0	0
size	0	Randint	129	22276755	3	5	0

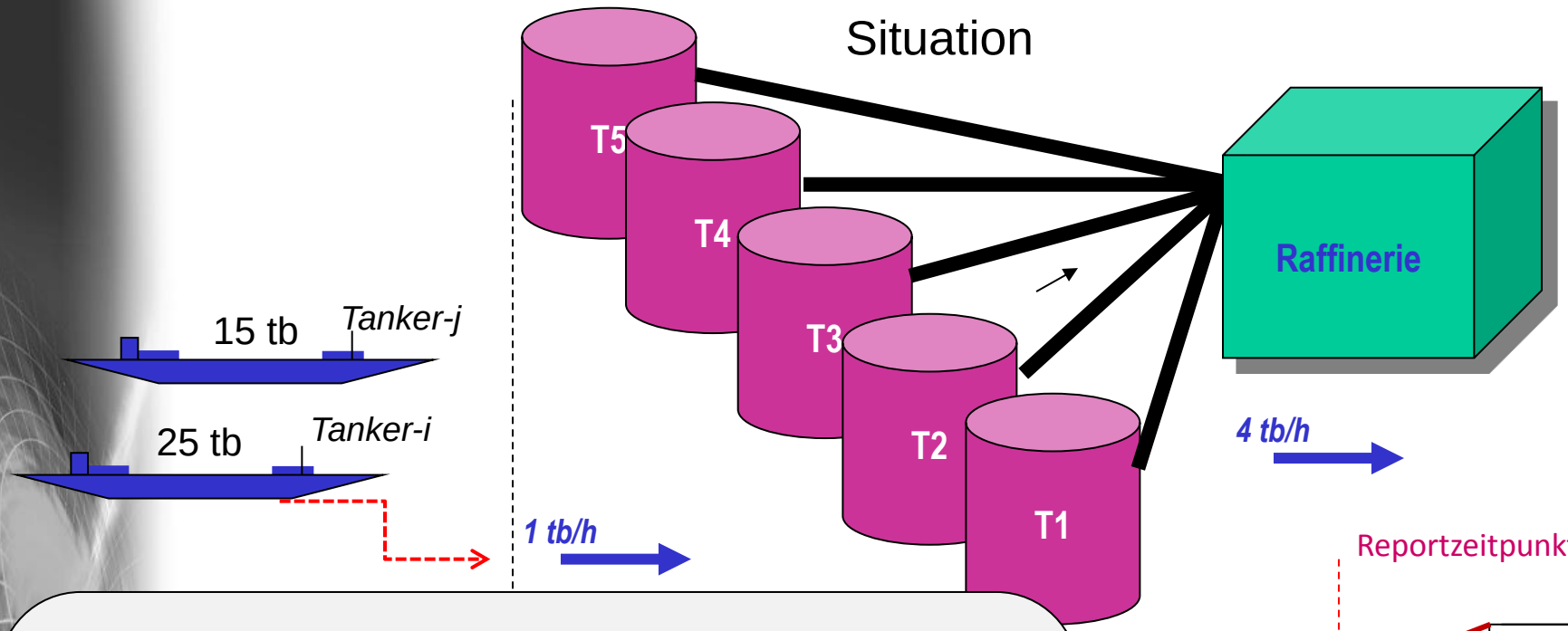
Queue Statistics

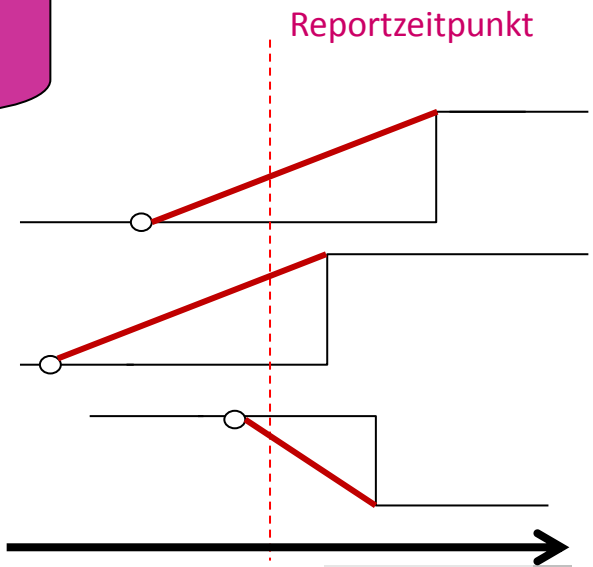
Name	Reset at	Min queue length	Max queue length	Now queue length	Avg queue length
shoreTanks_master_queue	0	0	5	0	0.184572
shoreTanks_slave_queue	0	0	5	2	1.82777

Waitq Statistics

Name	Reset at	Master Queue	Slave Queue	Number of Synch.	Zero wait masters	Avg masters wait	Zero wait slaves	Avg slaves wait
shoreTanks	0	shoreTanks_master_queue	shoreTanks_slave_queue	128	102	1.46019	27	14.2281

Problem: *korrekte Zustandserfassung zu einem beliebigen Zeitpunkt*





bisher

- Belegungszustände und deren Änderungen werden exakt erfasst
- Füllstände der jeweils aktuellen
  - Tank-Inhalte,
  - Tanker-Inhalte und
  - die Menge geflossenen Öls an die Raffinerie

  werden **nicht synchron** dargestellt,

- stetige Zustandsänderungen sind **Sprungfunktionen**

Allg. Vorgehensweise

- **Master- Prozesse** (und Tank-Prozesse bei Raffinerieversorgung)
  registrieren sich stets als Prozesse beim Report-Prozess sobald sie aktiv werden, dabei vermerken sie die **Startzei**t der Kooperation
  (sie streichen sich sobald sie passiv werden)

- Erhält der **Report-Prozess** zum geplanten Report-Zeitpunkt die Steuerung,
  - unterbricht er per interrupt **alle registrierten** arbeitenden Prozesse
  - diese werden re-scheduled
    (erkennen bei Fortsetzung nach holdFor) am Flag, dass eine Unterbrechung vorliegt und
  - **aktualisieren** Ihre Zustände per **linearer Interpolation** und die ihrer **Slaves**,
  - merken sich die **Restzeit**

  Damit liegen für den Report-Prozess die aktuellen Zustandswerte abholbereit vor

  Die **unterbrochenen** (aber aktiven Prozesse) setzen danach die Koop.Phase mit der **Restzeit** fort, aktualisieren danach ihre **Zustandsgrößen**.

Kritik an der aktuellen ODEMx-Lösung

- WaitQ sollte im Bedienungsmodus veränderbar sein

  – setStatus_OneMaster()

    so wie aktuelle Semantik:
    Der am längsten wartende Master erhält die Steuerung durch den nächsten eintreffenden Slave.
    Sollte die Selection- Funktion für alle erfassten Slaves False ergeben, blockiert dieser Master wieder ohne seine Nachfolger in der masterQ zu aktivieren

  – setStatus_AllMaster()

    Der am längsten wartende Master erhält die Steuerung durch den nächsten eintreffenden Slave (wie oben).
    Sollte die Selection- Funktion für alle erfassten Slaves False ergeben, blockiert dieser Master, aktiviert aber zuvor seinen masterQ-Nachfolger.