SLX 2.0

Objektorientierte Modellierung,
Spezifikation und Implementierung II

Anfang
Teil 2

Typ-System

- Vordefinierte Datentypen
- Felder
- Aufzählungstypen
- Zeichenketten
- Klassen
- Objektzeiger
- Mengentypen
- Sim-Typen

Agenda

- Ergänzungen zu Teil 1 (for-each, Referenzzählung)
- Objektorientierte Modellierung (Vererbung, Polymorphie, Konstruktion, Destruktion, Type-Casts)
- Wiederholung Verhaltensmodellierung
- Konzepte zur Verhaltensmodellierung am Beispiel des Fähre-Systems
- Beschreibung anwendungsspezifischer Erweiterungen der Sprache SLX

Ergänzung

for-each-Anweisung

- Iteration über die Elemente einer Menge
- Syntax:
  for (<Pointer> = each {<Klasse> | object}
  in [reverse] <Set>
  [{before | after | from} <Pointer_in_Set>]
  [with <Boolean_Expression>]) { ... }

```
set(*) s;
...
pointer(*) p;
```

```
for (p = each object in s with p->i > 10) {
    ...
}
```

Ergänzung

for-each-Anweisung

- Regeln (Auszug, vollständige Regeln siehe Programmhilfe)
  - bei reverse: Iteration in umgekehrter Reihenfolge
  - Iterations-Variable ist nach Ende der Schleife NULL (außer bei Abbruch der Iteration)
  - aktuelles Element darf aus Set entfernt werden (Nachfolge-Element wird bereits vor Betreten des Schleifenkörpers bestimmt)
  - sollte das Nachfolge-Element jedoch nicht mehr im Set enthalten sein, wird die Iteration von vorn gestartet

Klassenattribute

```
class X {
    static int i;
}
```

print-Anweisung

```
int i = 2;
string(5) s = "hello";

print(i,s) "_ und dann _";
// 2 und dann hello
```

Referenzzählung (use count)

```
pointer(X) x1 = new X(1);
pointer(X) x2;
// RZ von X = 1

x2 = x1;
// RZ von X = 2

x1 = NULL;
// RZ von X = 1

x2 = NULL;
// RZ von X = 0
// X wird vernichtet
```

- jedes Objekt hat einen Referenzzähler (RZ), in SLX auch „use count"
- Anpassung des RZ ist an Zuweisungen an Pointer-Variablen durch den assignment-operator gebunden
- Schritte:
  - 1. Zähler des aktuell ref. Objektes wird verringert
  - 2. Zähler des neu ref. Objektes wird erhöht
- entspricht shared_ptr<T> aus BOOST/C++

Objektorientierung

- Version 1.0 >> objektbasiert
  - keine Vererbung >> keine Polymorphie
  - nur Objekt-Komposition möglich
- Version 2.0 >> objektorientiert
  - Einfachvererbung zwischen Klassen, Mehrfachvererbung zwischen Interfaces (ähnlich wie in Java)

Objektorientierung

- Benutzung der Version 2.0 muss explizit eingeschaltet werden:
  `#define SLX2 ON`

Vererbung zw. Klassen

- zwischen Klassen ist nur Einfachvererbung erlaubt

```
class A {
    int i;
}

class B subclass(A) {
    int j;
}

procedure main() {
    pointer(A) a = new B();
    set(A) as;
    place new B() into as;
}
```

Vererbung zw. Klassen

```
class A {
}

class B (int l) subclass(A) {
    int k = l;
}

procedure main() {
    pointer(A) a = new B(1);
    pointer(B) b = a; // OK: checked at run-time
    pointer(B) bb = new A(); // should not be OK
    print (bb->k) "_";
}
```

- erlaubte Zeiger-zuweisungen

```
class C { }
```

```
pointer(B) pb = new C(); // no runtime error
```

```
Execution begins
11024184
Execution complete
```

Aufruf des Basisklassen-Konstruktors

```
class A (int j) {
    int i = j;
}

class B (int l) subclass(A(l)) {
    int k = l;
}

procedure main() {
    pointer(A) a = new B(1);
}
```

- erlaubte Werte:
  - Parameter des aktuellen Konstruktors,
  - Klassenattribute,
  - Konstanten,
  - oder globale Variablen

Prozedur = Methode

```
class C {
    procedure p() {
    }
    method m() {
    }
}
```

Methoden überschreiben

```
class A {
        overridable method m() {
                print "A.m()\n";
        }
}

class B subclass(A) {
        override method m() {
                print "B.m()\n";
        }
}

procedure main() {
        pointer(A) a = new B();
        a->m();
}
```

```
Execution begins
B.m()
Execution complete
```

- Vererbung zwischen Klassen
  - explizite Kennzeichnung von überschreibbaren und überschriebenen Methoden notwendig
    - überschriebene Methoden können wieder überschrieben werden
    - Schutz gegen unbeabsichtigte Überschreibung

```
class B subclass(A) {
        method m() {
•• Semantic error: this definition overides method "m" in
ancestor class "A",
•• but that method was not declared overridable
                print "B.m()\n";
        }
}
```

Methoden überschreiben

```
class A {
    overridable method m() {}
}

class B subclass(A) {
    override sealed method m() {}
}

class C subclass(B) {
    override method m() {}
•• Semantic error: this definition overides
method "m" in ancestor class "B."
•• The prior definition was sealed, so this
override is not allowed
}
```

- sealed-Methoden dürfen nicht weiter überschrieben werden

Abstrakte Klassen

```
abstract class A {
    abstract method m();
}

class B subclass(A) {
    concrete method m() {}
}
```

- abstrakte Klassen dürfen abstrakte Methoden enthalten
- nicht-abstrakte Spezialisierungen müssen entspr. konkrete Methoden enthalten

Interfaces

```
interface IF1 {
    abstract int i;
    abstract method m();
}

interface IF2 {
    abstract method m();
}

class A implements(IF1, IF2) {
    concrete int i;
    concrete method m() {}
}
```

- Interfaces enthalten nur abstrakte Methoden und abstrakte Variablen
- Mehrfachvererbung zwischen Interfaces erlaubt
- und ein Klasse kann von mehreren Interfaces erben

Aktive & passive Objekte

```
passive class Y {
    int i;
}

class X {
    Y y;
    actions { }
}

procedure main() {
    X x; // Erzeugung
    activate &x; // Aktivierung
}
```

- Objekte von Klassen (class) sind aktiv
  - besitzen Lebenslaufbeschreibung (actions-Property)
  - actions z.B. Wertzuweisung, Warten auf Zustands- oder Zeitereignis
  - Verhaltenausführung beginnt erst mit Aktivierung (activate <Pointer>)
  - ME als Zeiger auf das aktuelle Objekt
- Def. passiver Objekte mit Schlüsselwort passive (kein actions-Property)

Aktives Objekt
= Prozessinstanz

Puck

```
class X {
    actions {
        pointer(puck) p;
        p = ACTIVE;
    }
}
```

- Laufzeitkonstrukt
  - ähnlich GPSS-Transaktion oder Instruction Pointer
  - enthält die aktuelle Ausführungsposition im Lebenslauf eines aktiven Objektes
  - im Gegensatz zu GPSS ist immer ein Objektkontext vorhanden
  - puck ist auch Typ (Position kann in Variablen gespeichert werden)

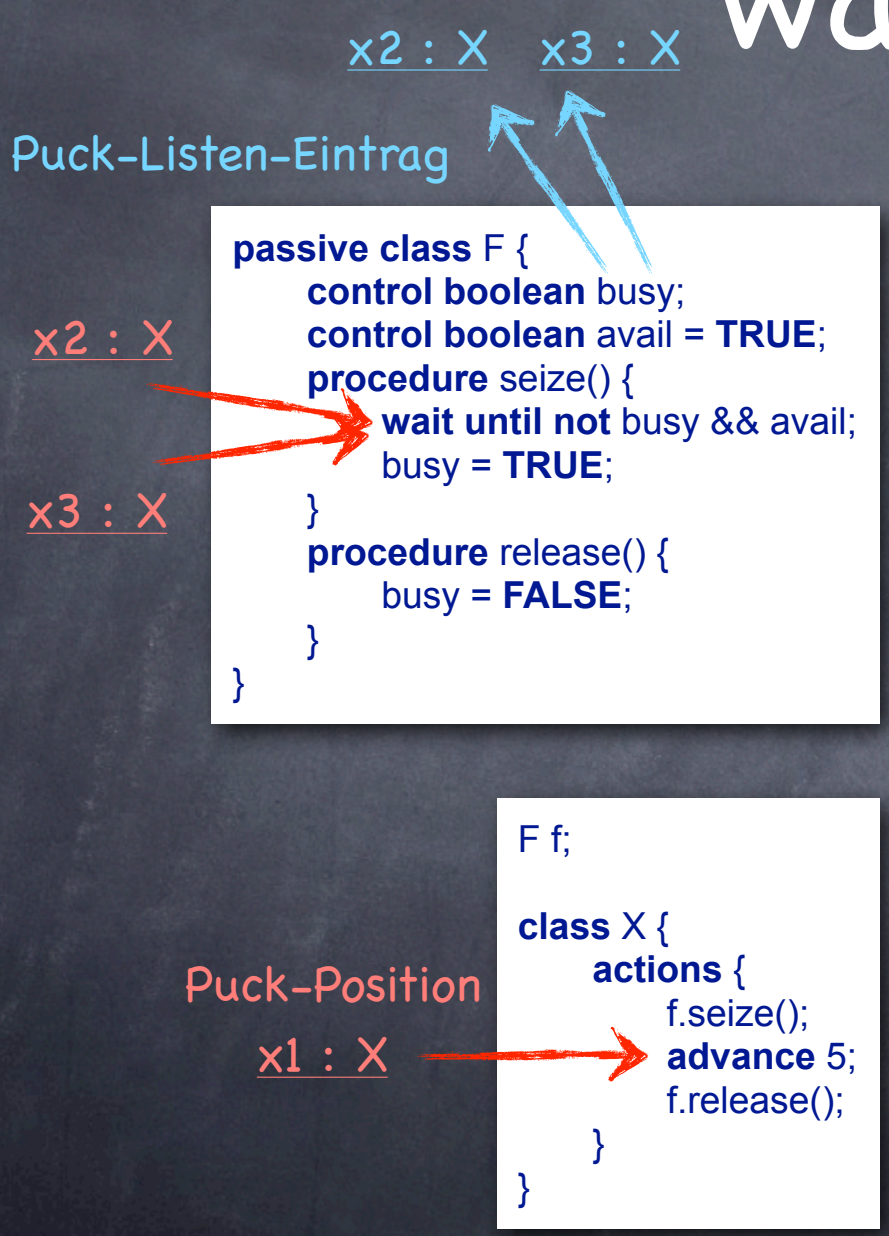
Wieder-holung

Kernkonzepte in Verhaltensbeschreibungen

- GPSS: viele versch. Blocktypen
- SLX: Reduktion auf Basis-Primitive (oder Kernkonzepte)
  - **new, activate, terminate**: Erzeugung, Aktivierung und Vernichtung von Objekten
  - **advance**: Modellierung von Zeitverbrauch
  - **wait until (...)**: Warten bis Zustandsbedingung erfüllt
  - **wait**: Warten auf Reaktivierung durch andere Prozessinstanz
  - **reactivate**: Reaktivierung einer wartenden Prozessinstanz
  - **interrupt, resume, yield, yield to**: Unterbrechung einer anderen Prozessinstanz inkl. Fortsetzung

>> Beschreibung von GPSS-Blöcken mit Basis-Primitiven (Kernkonzepten)

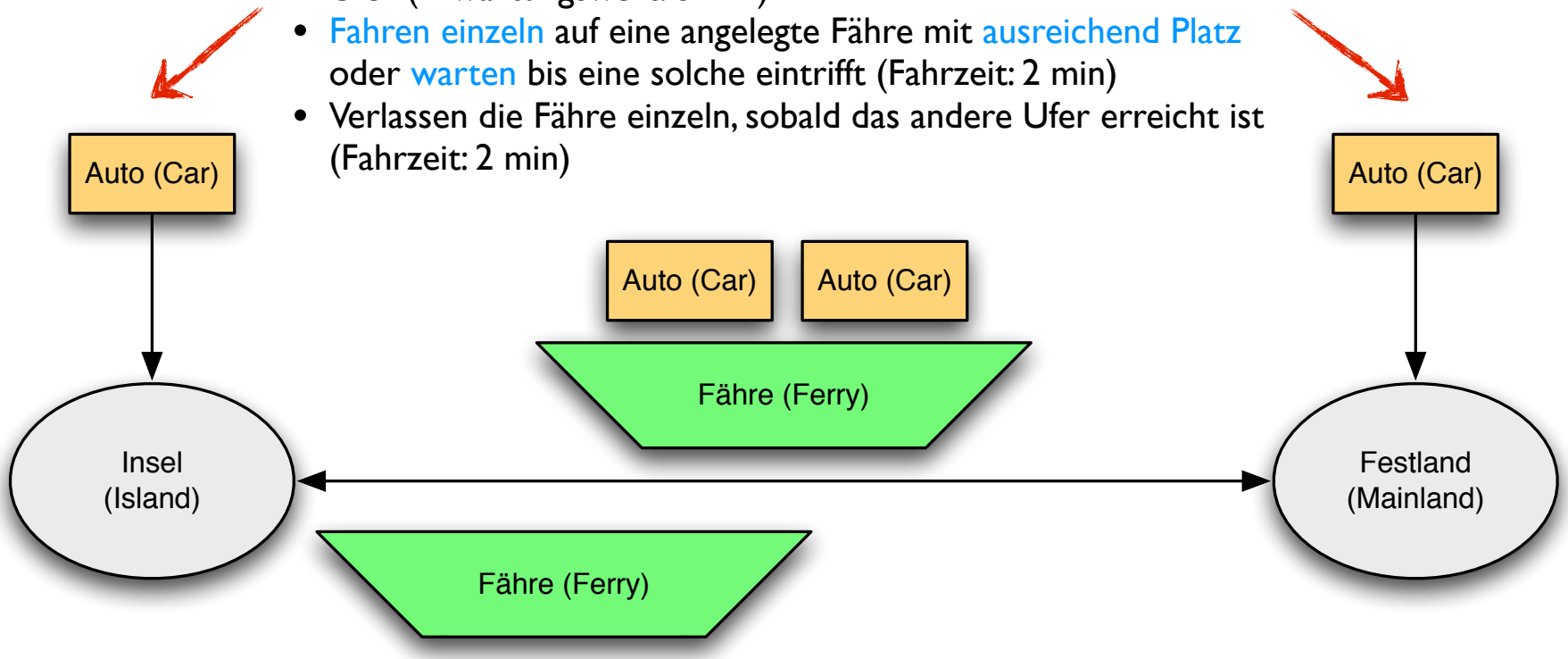
wait until Kernkonzept



- Puck wartet bis Zustandsbedingung (ZB) erfüllt ist
- ZB erfüllt: nächste Anweisung
- ZB nicht erfüllt:
  - für jede Zustandsvariable (ZV) einer ZB gibt es eine Puck-Liste
  - Puck wird in allen ZV-Puck-Listen der ZB eingeordnet (vgl. Retry-Chains in GPSS)
  - Erneute Überprüfung der ZB sobald sich eine ZV ändert
- ZVs für ZBs müssen explizit mit control deklariert werden

Durchgängiges Beispiel: Fähresystem

- Autos erreichen in exponentialverteilten Abständen jedes der Ufer (Erwartungswert: 8 min)
- Fahren einzeln auf eine angelegte Fähre mit ausreichend Platz oder warten bis eine solche eintrifft (Fahrzeit: 2 min)
- Verlassen die Fähre einzeln, sobald das andere Ufer erreicht ist (Fahrzeit: 2 min)



- Mehrere Fähren fahren von Insel nach Festland und zurück
- Zur Vereinfachung starten alle Fähren auf der Insel
- Fähre wartet bis Kapazität erschöpft oder Mindestwartezeit von 15 min abgelaufen
- Überfahrt dauert 20 min
- Überfahrt darf erst beginnen, wenn kein Auto im Begriff ist auf die Fähre zu fahren!
- Fähre wartet bis alle Autos entladen sind
- Fähre wartet wieder auf neue Autos ...

Struktur des Fähresystems

```
passive class Coast {
   pointer(puck) waitingCars; // Liste
   set(Ferry) ferries;
}
```

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   initial {
      coast = inCoast;
   }
}
```

```
class Ferry (int inCapacity) {
   pointer(Coast) currentCoast;
   pointer(Coast) coasts[2];
   int currentCoastIndex;
   set(Car) cars;
   control boolean rampClear = TRUE;
   int capacity;
   initial {
      capacity = inCapacity;
   }
}
```

```
class CarGen (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   rn_stream tba;
   initial {
      coast = inCoast;
   }
}
```

```
procedure main() {
   Coast island;
   Coast mainland;

   Ferry ferry(5);
   ferry.coasts[1] = &island;
   ferry.coasts[2] = &mainland;
   ferry.currentCoast = &island;
   ferry.currentCoastIndex = 1;

   CarGen islandGen(&island);
   CarGen mainlandGen(&mainland);

   wait until (time == 24*60);
}
```

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      • warte auf Fähre mit ausreichend Platz
      • warte auf freie Rampe zur Fähre
      • belege Rampe und fahre auf Fähre
      • gebe Rampe frei
      • warte bis Überfahrt abgeschlossen
      • warte auf freie Rampe
      • belege Rampe und fahre von Fähre
      • gebe Rampe frei
      • terminiere
   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      // Nur erste naive Umsetzung!
      wait until (coast->ferries.size > 0);
      pointer(Ferry) ferry;
      ferry = getFirstFreeFerry(coast);

   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

```
• warte auf Fähre mit ausreichend Platz
• warte auf freie Rampe zur Fähre
• belege Rampe und fahre auf Fähre
• gebe Rampe frei
• warte bis Überfahrt abgeschlossen
```

```
procedure getFirstFreeFerry(in pointer(Coast) coast)
      returning pointer(Ferry) {

   pointer(Ferry) f;
   for (f = each Ferry in coast->ferries) {
      if (f->cars.size < f->capacity) {
         return f;
      }
   }
   return NULL;
}
```

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      // Nur erste naive Umsetzung!
      wait until (coast->ferries.size > 0);
      pointer(Ferry) ferry;
      ferry = getFirstFreeFerry(coast);

      // Was wenn überall die Kapazität
      // bereits erschöpft ist?

   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

- warte auf Fähre mit ausreichend Platz
- warte auf freie Rampe zur Fähre
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- warte bis Überfahrt abgeschlossen
- warte auf freie Rampe
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- terminiere

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      // Nur erste naive Umsetzung!
      wait until (coast->ferries.size > 0);
      pointer(Ferry) ferry;
      ferry = getFirstFreeFerry(coast);

      // Was wenn überall die Kapazität
      // bereits erschöpft ist?

      // Fehlende Ausdruckskraft für
      // komplexe Bedingungen,
      // wie z.B. aus OCL:
      // coast.ferries->exists(
           f | f.cars->size() < f.capacity);

      // Lösung folgt später ...

   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

- warte auf Fähre mit ausreichend Platz
- warte auf freie Rampe zur Fähre
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- warte bis Überfahrt abgeschlossen
- warte auf freie Rampe
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- terminiere

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
  - Conditioned Waiting
  - Scheduled
  - Interrupted

- Zustand nach der Objekterzeugung und vor der Aktivierung
- Es existieren noch keine Pucks

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
  - Conditioned Waiting
  - Scheduled
  - Interrupted

- Zustand nach der Aktivierung
- Puck zeigt auf erste Anweisung im actions-Teil
- Puck ist zur aktuellen Modellzeit für die Ausführung bereit
- Entspricht in GPSS dem Eintrag einer Transaktion in die CEC
- In SLX erfolgt Verwaltung in Liste „Moving Pucks“

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
  - Conditioned Waiting
  - Scheduled
  - Interrupted

- Ein Puck einer Prozessinstanz wird ausgeführt
- Puck wird entsprechend der Lebenslauf-beschreibung von der internen Steuerung verarbeitet

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
  - Conditioned Waiting
  - Scheduled
  - Interrupted

- Es existieren keine aktiven Pucks mehr für die Prozessinstanz
- Pucks wurden entweder explizit vernichtet (terminate) oder implizit (Ende von actions-Teil)
- Pucks und Prozessinstanz können jedoch weiterhin referenziert sein

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
    - Conditioned Waiting
    - Scheduled
    - Interrupted

- Totale Blockierung
- Puck wartet auf eine explizite Reaktivierung
- Keine Entsprechung in GPSS

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
  - Conditioned Waiting
  - Scheduled
  - Interrupted

- Blockierung ist abhängig von einer Bedingung
- Puck wartet an einer wait-until-Anweisung auf die Erfüllung einer Zustandsbedingung
- Entspricht in GPSS z.B. dem Warten auf das Freiwerden einer Facility (Delay-Chain) oder dem Warten auf das Eintreten eines Zustandes (Retry-Chain)

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
  - Conditioned Waiting
  - Scheduled
  - Interrupted

- Blockierung bis Zeitereignis eintritt
- Puck-Ausführung ist im Terminkalender vorgemerkt
- Entspricht in GPSS dem Eintrag einer Transaktion in die FEC

Grundzustände von Prozessinstanzen

- Initialized
- Ready
- Active
- Terminated
- Blocked
  - Waiting
  - Conditioned Waiting
  - Scheduled
  - Interrupted

- Nach Unterbrechung eines Pucks mit der interrupt-Anweisung
- Entspricht in GPSS dem Eintrag einer Transaktion in die Interrupted-Chain einer Facility nach einem PREEMPT-Block

Zustände & Zustandsübergänge von Prozessinstanzen

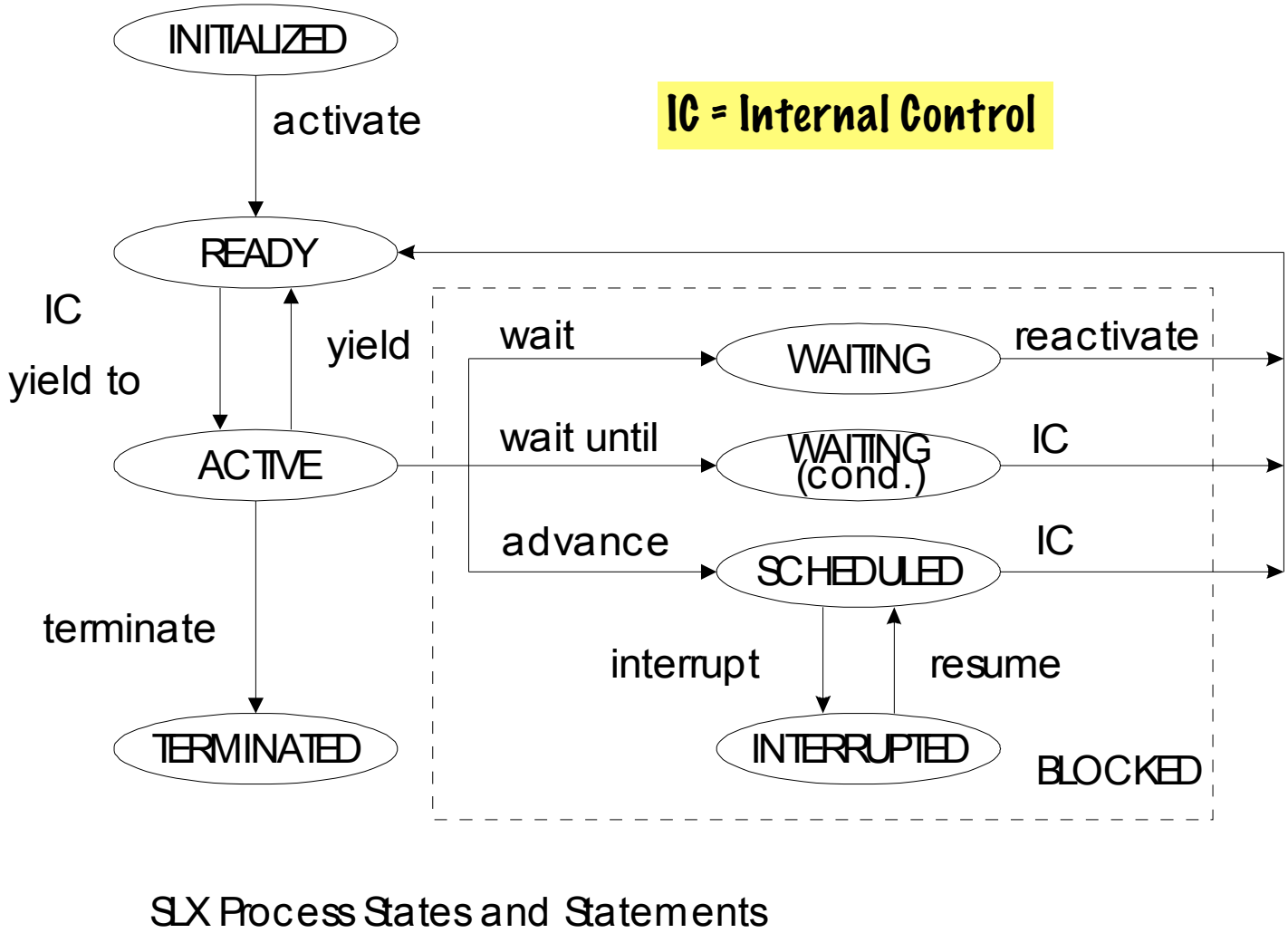


Abbildung 4: Prozessübergänge

Ähnliche Zustände für Pucks

Puck-Liste	Vgl. GPSS	Puck-Zustand
Moving Pucks	CEC	MOVING
Scheduled Pucks	FEC	SCHEDULED
Waiting Pucks	Delay-Chains, Retry-Chains	WAITING
Interrupted Pucks	Interrupted-Chains	INTERRUPTED
		TERMINATED

Kernkonzepte: wait & reactivate

- Totale Blockierung und Explizite Reaktivierung
- Ein Puck führt die wait-Anweisung aus und wartet bis ein anderer Puck die reactivate-Anweisung für ihn ausführt.
- Eigentlich nicht notwendig, wait-until ist theoretisch ausreichend
- Aber:
  - Laufzeit-Vorteile durch explizites Neuanstoßen der Überprüfung von Zustandsbedingungen
  - Fehlende Ausdruckskraft von wait-until bei komplexen Bedingungen
- Zwei Formen:
  wait und **wait list** = *Puck-Zeiger*

```
pointer(puck) pp;
...
```

```
class P1 {
   actions {
      pp = ACTIVE;
      ...
      wait;
   }
}
```

```
class P2 {
   actions {
      ...
      reactivate pp;
   }
}
```

Kernkonzepte: wait & reactivate

- Zwei Formen: **wait** und **wait list** = *Puck-Zeiger*

```
pointer(puck) plist;
...
```

```
class P1 {
   actions {
      pp = ACTIVE;
      ...
      wait list=plist;
   }
}
```

```
class P2 {
   actions {
      ...
      reactivate list=plist;
   }
}
```

- Ausdruck nach **list** muss Bezeichner für einen *Puck-Zeiger* sein
- Dieser Zeiger referenziert den letzten Puck in einer ganzen Liste von Pucks
- **wait list** = *Puck-Zeiger*
  führt zur Eintragung des aktuellen Puck an das Ende der Liste
- **reactivate list** = *Puck-Zeiger*
  führt zur Eintragung aller Pucks der Liste in die Liste der Moving Pucks in der Reihenfolge des Eintrags in die Liste

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      pointer(Ferry) ferry;
      while (ferry == NULL) {
         ferry = getFirstFreeFerry(coast);
         if (ferry == NULL) {
            wait list=coast->waitingCars;
         }
      }
      // verhindert Belegung über Kapazität
      place ME into ferry->cars;

      wait until (ferry->rampClear);
      ferry->rampClear = FALSE;
      advance 2;
      ferry->rampClear = TRUE;

      ...
   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

- warte auf Fähre mit ausreichend Platz
- warte auf freie Rampe zur Fähre
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- warte bis Überfahrt abgeschlossen
- warte auf freie Rampe

```
procedure getFirstFreeFerry(in pointer(Coast) coast)
      returning pointer(Ferry) {

   pointer(Ferry) f;
   for (f = each Ferry in coast->ferries) {
      if (f->cars.size < f->capacity) {
         return f;
      }
   }
   return NULL;
}
```

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      pointer(Ferry) ferry;
      while (ferry == NULL) {
         ferry = getFirstFreeFerry(coast);
         if (ferry == NULL) {
            wait list=coast->waitingCars;
         }
      }
      // verhindert Belegung über Kapazität
      place ME into ferry->cars;

      wait until (ferry->rampClear);
      ferry->rampClear = FALSE;
      advance 2;
      ferry->rampClear = TRUE;

      wait until (ferry->currentCoast != coast
         && ferry->rampClear);
      ferry->rampClear = FALSE;
      advance 2;
      ferry->rampClear = TRUE;
      remove ME from ferry->cars;
      terminate;
   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

```
• warte auf Fähre mit ausreichend Platz
• warte auf freie Rampe zur Fähre
• belege Rampe und fahre auf Fähre
• gebe Rampe frei
• warte bis Überfahrt abgeschlossen
• warte auf freie Rampe
• belege Rampe und fahre auf Fähre
• gebe Rampe frei
• terminiere
```

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      pointer(Ferry) ferry;
      while (ferry == NULL) {
         ferry = getFirstFreeFerry(coast);
         if (ferry == NULL) {
            wait list=coast->waitingCars;
         }
      }
      // verhindert Belegung über Kapazität
      place ME into ferry->cars;

      wait until (ferry->rampClear);
      ferry->rampClear = FALSE;
      advance 2;
      ferry->rampClear = TRUE;

      wait until (ferry->currentCoast != coast
         && ferry->rampClear);
      ferry->rampClear = FALSE;
      advance 2;
      ferry->rampClear = TRUE;
      remove ME from ferry->cars;
      terminate;
   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

- warte auf Fähre mit ausreichend Platz
- warte auf freie Rampe zur Fähre
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- warte bis Überfahrt abgeschlossen
- warte auf freie Rampe
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- terminiere

Beispiel: Lebenslauf von Autos

```
class Car (pointer(Coast) inCoast) {
   pointer(Coast) coast;
   ...
   actions {
      pointer(Ferry) ferry;
      while (ferry == NULL) {
         ferry = getFirstFreeFerry(coast);
         if (ferry == NULL) {
            wait list=coast->waitingCars;
         }
      }
      // verhindert Belegung über Kapazität
      place ME into ferry->cars;

      seize ferry->ramp;
      advance 2;
      release ferry->ramp;

      wait until (
        ferry->currentCoast != coast);
      seize ferry->ramp;
      advance 2;
      release ferry->ramp;
      remove ME from ferry->cars;
      terminate;
   }
}
```

- Zur Erinnerung
  - Nur Einzelauffahrt und Einzelabfahrt erlaubt
  - Auf- und Abfahrzeit 2 min
- Beschreibung erfolgt zunächst nur mit Kernkonzepten

- warte auf Fähre mit ausreichend Platz
- warte auf freie Rampe zur Fähre
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- warte bis Überfahrt abgeschlossen
- warte auf freie Rampe
- belege Rampe und fahre auf Fähre
- gebe Rampe frei
- terminiere

```
class Ferry (int inCapacity) {
   pointer(Coast) currentCoast;
   pointer(Coast) coasts[2];
   int currentCoastIndex;
   set(Car) cars;
   facility ramp;
   ...
}
```

Auszug: Beschreibung der GPSS-Facility

Aufzählungstyp mit den möglichen Werten IDLE, SEIZED und PREEMPTED

```
passive class facility (...) {
   ...
   pointer(puck) owner;
   control facility_state state;
   control enum {AVAIL, UNAVAIL}
         availability;
   ...
   pointer(puck) change_in_seizability;
   ...
   initial {
      state = IDLE;
      availability = AVAIL;
      ...
   }
   report {
      ...
   }
}
```

```
procedure SEIZE (inout facility f) {
   while (f.state != IDLE || f.availability != AVAIL)
      wait list = f.change_in_seizability; // sleep

   f.owner = ACTIVE;
   f.state = SEIZED;
   ...
}
```

```
procedure RELEASE (inout facility f) {
   ...
   f.owner = NULL;
   f.state = IDLE;
   ...
   if (f.availability == AVAIL)
      reactivate list = f.change_in_seizability;
   ...
}
```

Weitere (nicht betrachtete) Kernkonzepte

- interrupt
- resume
- yield, yield to

Inter- und Intra-Objekt-Parallelität

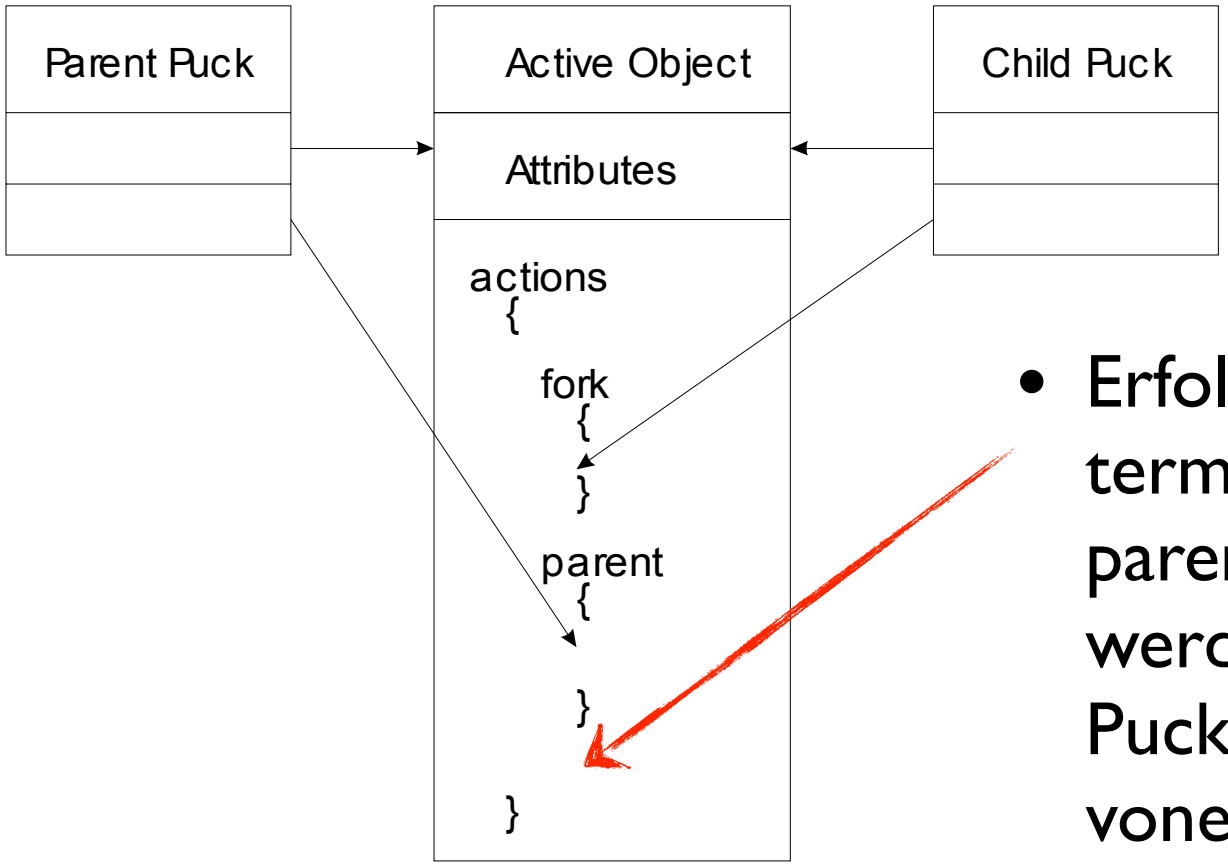
- Inter-Objekt-Parallelität
  - Jede Prozessinstanz für die ein Puck existiert, wird parallel aber sequentiell zu allen anderen Prozessinstanzen mit Pucks zum gleichen Modellzeitpunkt ausgeführt.

Inter- und Intra-Objekt-Parallelität

- Intra-Objekt-Parallelität
  - Für eine Prozessinstanz können mehrere Pucks gleichzeitig existieren
  - Beschreibung prozessinstanzlokaler Parallelität
  - Alle Pucks einer Prozessinstanz operieren dabei auf den gleichen Objektdaten
  - Puck-Erzeugung erfolgt in actions-Teil mit

```
fork { ... }
parent { ... }
```

Inter- und Intra-Objekt-Parallelität



- Erfolgt kein explizites terminate im fork- oder parent-Abschnitt, so werden die zugehörigen Pucks unabhängig voneinander bei der Aktion nach dem parent-Teil weiter fortgeführt

Beispiel: Lebenslauf von Fähren

```
class Ferry (int inCapacity) {
   pointer(Coast) currentCoast;
   pointer(Coast) coasts[2];
   int currentCoastIndex;
   set(Car) cars;
   control boolean rampClear
      = TRUE;
   int capacity;
   initial {
      capacity = inCapacity;
   }
   ...
}
```

```
actions {
   forever {
      place ME into currentCoast->ferries;
      reactivate list=currentCoast->waitingCars;
      control boolean ferryLeaves = FALSE;
      fork {
         wait until (cars.size == capacity || ferryLeaves);
         if (ferryLeaves) terminate; // let parent proceed
      }
      parent {
         advance 15;
         if (cars.size == capacity) {
            terminate; // let child proceed
         }
         else {
            ferryLeaves = TRUE;
            wait until (rampClear);
         }
      }
      remove ME from currentCoast->ferries;
      advance 20;
      currentCoastIndex %= 2;
      currentCoast = coasts[currentCoastIndex];
      wait until (cars.size == 0);
   }
}
```

Anwendungsspezifische Erweiterung

- Buzz-Word:
  domänenspezifisch, domain-specific
- Definition neuer Anweisungen auf der Basis bestehender SLX-Anweisungen
- Definition in zwei Teilen:
  - Beschreibung der Notation
  - Beschreibung der Abbildung auf bestehende SLX-Anweisungen
- Große Ähnlichkeit mit Prozedurdefinition, jedoch Definition einer eigenen Notation möglich

Definition neuer Anweisungen

- Beschreibung der Aufruf-Notation besteht aus
  - Bezeichner
  - Parameterdefinitionen
- Bei Anweisungs-verwendung sind als Werte von Parametern beliebige Zeichenketten erlaubt

Syntax für Anweisungsdefinitionen

```
statement Bezeichner Parameter_Def, ...;
definition {
   Anweisungen;
   ...
}
```

Beispiel einer Anweisungsdefinition

```
statement anweisung1 #p1 capacity=#p2;
definition { ... }
```

Beispiel einer Anweisungsverwendung

```
actions {
   ...
   anweisung1 someText capacity=2;
   ...
}
```

Arten von Parametern

- Stellungsparameter
  - Notation: *#Parametername*
  - Wert muss an angegebener Stelle übergeben werden
- Kennwortparameter
  - Zusätzliche Angabe eines Kennwortes ist bei Anweisungverwendung erforderlich
  - Notation: *Kennwort = #Parametername*
- Füllparameter
  - Notation: *@Bezeichner* #Parametername
  - Zusätzliche Angabe eines Bezeichners ist erforderlich (ähnlich wie Kennwortparameter)
  - Lediglich zur verbesserten Lesbarkeit einer Anweisung

Definition von Parametern

Beispiel

```
statement anweisung1 #p1 capacity=#p2 @with #p3;
definition { ... }

procedure main() {
  anweisung1 4 capacity=2 with 9;
}
```

Definition von Parametern

- Optionale Parameter
  - Notation in eckigen Klammern
    [*Parameter*liste **]**
  - Bsp.-Def.: `statement anw #p1 [#p2];`
- Wiederholung von Parametern
  - Notation in geschweiften Klammern, gefolt von **,...**
    { *Parameterliste* **},...**
  - Bsp.-Def.: `statement anw #p1 {#p2 #p3},...;`
- Alternative Parameter
  - Notation in geschweiften Klammern und **|**-Operator
    { Parameter1 **|** Parameter2 **|** ... **}**
  - Bsp.-Def.: `statement anw {#p1 | #p2};`

Abbildungsdefinition einer Anweisung

- Abbildung erfolgt mit expand-Anweisung:
  expand (#p1, #p2, ...) "*Zeichenkette*"
- Schlüsselwort **expand** gefolgt von Parametern in Klammern, die in der Zeichenkette eingesetzt werden
- Einsetzungsposition wird in Zeichenkette durch # markiert
- Daneben sind auch alle bekannten SLX-Anweisungen und Kontrollstrukturen erlaubt

Definition

```
statement rel #fac;
definition {
   expand (#fac) "RELEASE(#);";
}
```

Anwendung

```
rel Schleifer;
```

Automatische Ersetzung

```
RELEASE(Schleifer);
```

Abbildungsdefinition einer Anweisung

- Optionale Parameter, die beim Aufruf der Anweisung nicht verwendet werden erhalten eine leere Zeichenkette als Wert
- Im Definitionsteil ist ein Test auf die Wertangabe für diese Parameter möglich

Definition

```
statement s #p1 [#p2];
definition {
   if (#p2 == "") {
      expand (#p1) "y #";
   }
   else {
      expand (#p2, #p1) "x # #";
   }
}
```

Anwendung

```
s p1Value 29;
s otherP1Value;
```

Abbildungsdefinition einer Anweisung

- Werte wiederholbarer Parameter werden in einem Feld gespeichert
- Beim Aufruf nicht verwendete Parameter erhalten wieder eine leere Zeichenkette als Wert

```
statement s {#p1 #p2},...;
definition {
   integer i;
   for (i=1; #p1[i] != ““; i++) {
      expand (#p1[i], #p2[i]) “..#..#..“;
   }
}
```

Vergüterei als Beispiel für eine Erweiterung

```
import <h7>

module VergueterеiEinfach {

   control int ringeImLager = 0;

   facility Ofen;
   queue OfenQ;
   facility WBad;
   queue WBadQ;
   facility Kran;

   class RingGen {
      actions {
         pointer(Ring) ring;
         forever {
            ring = new Ring();
            activate ring;
            advance 50;
         }
      }
   }

   ...
```

Ausgangs-situation

```
   ...

   procedure main() {
      RingGen ringGen;
      activate &ringGen;
      wait until (ringeImLa
      report(system);
      exit(0);
   }
}
```

```
   class Ring {
      actions {
         seize Ofen;
         seize Kran;
         advance 5;
         release Kran;
         advance 60;

         seize WBad;
         seize Kran;
         release Ofen;
         advance 5;
         release Kran;
         advance 10;

         seize Kran;
         release WBad;
         advance 5;
         ringeImLager++;
         release Kran;
         terminate;
      }
   }
```

Vergüterei als Beispiel für eine Erweiterung

Definition und Anwendung einer Transport-Anweisung

```
class Ring {
   actions {
      transport Kran 5 toFac=Ofen;
      advance 60;

      transport Kran 5 fromFac=Ofen toFac=WBad;
      advance 10;

      transport Kran 5 fromFac=WBad;
      ringeImLager++;

      terminate;
   }
}
```

```
statement transport #res #duration [fromFac=#from] [toFac=#to];
definition {
   if (#to != "") {
      expand (#to) "seize #;";
   }
   expand (#res) "seize #;";
   if (#from != "") expand(#from) "release #;";
   expand (#duration) "advance #;"
   expand (#res) "release #;";
}
```

Offene Aspekte von SLX

- Ein- und Ausgabe
  - Dateien
  - Reports
  - Histogramme
- Module