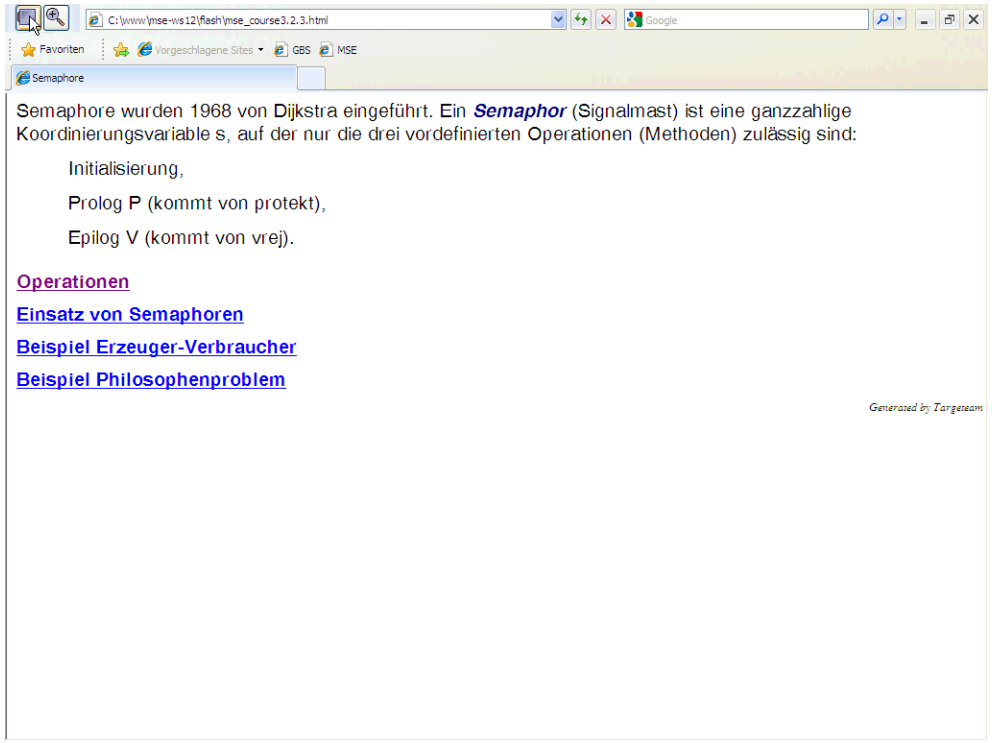
Script **generated by TTT**

Title: Eingebettete_Systeme (23.10.2012)

Date: Tue Oct 23 16:10:01 CEST 2012

Duration: 64:54 min

Pages: 32



Einsatz von Semaphoren

Notation: vordefinierter Typ `semaphor(int s)` . ***Semaphor-Objekt*** ist Instanz des Typs `semaphor` ; Semaphor wir mit Parameter s initialisiert.

Zugang zu kritischen Abschnitten

Realisierung der kritischen Abschnitte von Prozessen, in denen auf eine exklusiv benutzbare Ressource X zugegriffen wird:

1. Definition eines Semaphor-Objekts wa: semaphor(1), d.h. Initialisierung der Kontrollvariable des Semaphor-Objekts wa mit 1.
2. Klammern der kritischen Abschnitte, in denen auf die Ressource X zugegriffen wird, mit P und V Operationen:

   wa.P

   Code mit Zugriffen auf X

   wa.V

Semaphor-Konzept erfüllt w.A. Lösungsanforderungen

Wechselseitiger Ausschluss für alle kritischen Abschnitte.

keine Reihenfolge-Annahmen.

keine Ausführungszeit-Annahmen.

kein Verhungern.

Generated by Targeteam

Semaphore wurden 1968 von Dijkstra eingeführt. Ein ***Semaphor*** (Signalmast) ist eine ganzzahlige Koordinierungsvariable s, auf der nur die drei vordefinierten Operationen (Methoden) zulässig sind:

Initialisierung,

Prolog P (kommt von protekt),

Epilog V (kommt von vrej).

Operationen

Einsatz von Semaphoren

Beispiel Erzeuger-Verbraucher

Beispiel Philosophenproblem

Generated by Targeteam

Zugriff auf Puffer W erfolgt durch Semaphor wa: semaphor(1);

```
Erzeuger E :                    Verbraucher V :
while (true) {                  while (true) {
    produziere                      wa.P
    wa.P                            entnimm aus W, falls Element da; sonst warte
    schreibe nach W                 wa.V
    wa.V                            verarbeite
}                               }
```

Problem : es kann eine Verklemmung auftreten, wenn der Verbraucher wa.P ausführt und warten muss, weil der Puffer kein Element enthält.

Variante 2

Einführen eines zusätzlichen Semaphors voll: semaphor(0), das die Datenelemente im Puffer zählt:

```
Erzeuger E :                    Verbraucher V :
while (true) {                  while (true) {
    produziere                      voll.P
    wa.P                            wa.P
    schreibe nach W                 entnimm aus W
    wa.V                            wa.V
    voll.V                          verarbeite
}                               }
```

Für den Erzeuger ergibt sich natürlich ein analoges Problem, falls der Puffer W nur eine beschränkte Kapazität besitzt.

Variante 3

Einführen eines zusätzlichen Semaphors leer: semaphor(n), das die Anzahl der freien Elemente im Puffer zählt:

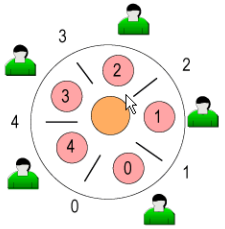
```
wa.semaphor(1); //kontrolliert den Zugang zum kritischen Bereich
voll.semaphor(0); //zählt die Anzahl der Einheiten im Puffer
leer.semaphor(n); //zählt die Anzahl der freien Pufferplätze

Erzeuger E :                    Verbraucher V :

while (true) {                  while (true) {
    produziere Einheit ;            voll.P;
    leer.P;                         wa.P;
    wa.P;                           entnimm Einheit aus W
    schreibe Einheit nach W ;       wa.V
    wa.V;                           leer.V;
    voll.V;                         verarbeiteEinheit ;
}                               }
```


Beispiel Philosophenproblem

Zu den klassischen Synchronisationsproblemen zählt das Problem der ***speisenden Philosophen*** ("Dining Philosophers"). In einem Elfenbeinturm leben fünf Philosophen. Der Tageszyklus eines jeden Philosophen besteht abwechselnd aus Essen und Denken. Die fünf Philosophen essen an einem runden Tisch, auf dem in der Mitte eine Schüssel voller Reis steht. Jeder Philosoph hat seinen festen Platz an dem Tisch und zwischen zwei Plätzen liegt genau ein Stäbchen. Das Problem der Philosophen besteht nun darin, dass der Reis nur mit genau zwei Stäbchen zu essen sind. Darüber hinaus darf jeder Philosoph nur das direkt rechts und das direkt links neben ihm liegende Stäbchen zum Essen benutzen. Das bedeutet, dass zwei benachbarte Philosophen nicht gleichzeitig essen können.



Realisierung mit Semaphoren

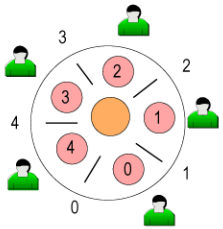
- Variante 1
- Variante 2

Variante 1

Für eine Lösung des Philosophenproblems seien die folgenden 5 Semaphore definiert: stab_0, stab_1, ..., stab_4, wobei jedes der 5 Semaphore mit 1 initialisiert ist. Jeder Philosoph j, mit j ∈ {0,...,4}, führe den folgenden Algorithmus aus:

```
philosoph_j:
    while (true) {
       Denken;
       stab_i.P;                    mit i = j
       stab_i.P;                    mit i = j + 1 mod 5
       Essen
       stab_i.V;                    mit i = j
       stab_i.V;                    mit i = j + 1 mod 5
    }
```


Zu den klassischen Synchronisationsproblemen zählt das Problem der ***speisenden Philosophen*** ("Dining Philosophers"). In einem Elfenbeinturm leben fünf Philosophen. Der Tageszyklus eines jeden Philosophen besteht abwechselnd aus Essen und Denken. Die fünf Philosophen essen an einem runden Tisch, auf dem in der Mitte eine Schüssel voller Reis steht. Jeder Philosoph hat seinen festen Platz an dem Tisch und zwischen zwei Plätzen liegt genau ein Stäbchen. Das Problem der Philosophen besteht nun darin, dass der Reis nur mit genau zwei Stäbchen zu essen sind. Darüber hinaus darf jeder Philosoph nur das direkt rechts und das direkt links neben ihm liegende Stäbchen zum Essen benutzen. Das bedeutet, dass zwei benachbarte Philosophen nicht gleichzeitig essen können.



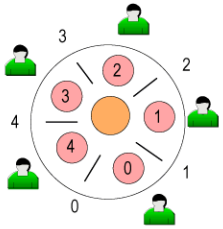
Realisierung mit Semaphoren

Nur vier Philosophen dürfen gleichzeitig zu ihrem linken Stäbchen greifen. Dies wird durch Einführung eines weiteren Semaphors `tisch: semaphor(4)`, das mit 4 initialisiert wird, erreicht. Der "Anweisungsteil" jedes Philosophen wird zusätzlich mit `tisch.P` und `tisch.V` geklammert. Es ergibt sich also folgende Lösung: Jeder Philosoph j, mit j ∈ {0,...,4} führt den folgenden Algorithmus aus:

```
philosoph_j:
    while (true) {
       Denken;
       tisch.P
       stab_i.P;                    mit i = j
       stab_i.P;                    mit i = j + 1 mod 5
       Essen
       stab_i.V;                    mit i = j
       stab_i.V;                    mit i = j + 1 mod 5
       tisch.V
    }
```


Beispiel Philosophenproblem

Zu den klassischen Synchronisationsproblemen zählt das Problem der ***speisenden Philosophen*** ("Dining Philosophers"). In einem Elfenbeinturm leben fünf Philosophen. Der Tageszyklus eines jeden Philosophen besteht abwechselnd aus Essen und Denken. Die fünf Philosophen essen an einem runden Tisch, auf dem in der Mitte eine Schüssel voller Reis steht. Jeder Philosoph hat seinen festen Platz an dem Tisch und zwischen zwei Plätzen liegt genau ein Stäbchen. Das Problem der Philosophen besteht nun darin, dass der Reis nur mit genau zwei Stäbchen zu essen sind. Darüber hinaus darf jeder Philosoph nur das direkt rechts und das direkt links neben ihm liegende Stäbchen zum Essen benutzen. Das bedeutet, dass zwei benachbarte Philosophen nicht gleichzeitig essen können.



Realisierung mit Semaphoren

Variante 1
Variante 2

Generated by Targeteam

Variante 2

Nur vier Philosophen dürfen gleichzeitig zu ihrem linken Stäbchen greifen. Dies wird durch Einführung eines weiteren Semaphors `tisch: semaphor(4)`, das mit 4 initialisiert wird, erreicht. Der "Anweisungsteil" jedes Philosophen wird zusätzlich mit `tisch.P` und `tisch.V` geklammert. Es ergibt sich also folgende Lösung: Jeder Philosoph j, mit j ∈ {0,...,4} führt den folgenden Algorithmus aus:

```
philosoph_j:
    while (true) {
       Denken;
       tisch.P
       stab_i.P;                mit i = j
       stab_i.P;                mit i = j + 1 mod 5
       Essen
       stab_i.V;                mit i = j
       stab_i.V;                mit i = j + 1 mod 5
       tisch.V
    }
```

Generated by Targeteam

Semaphore

Semaphore wurden 1968 von Dijkstra eingeführt. Ein ***Semaphor*** (Signalmast) ist eine ganzzahlige Koordinierungsvariable s, auf der nur die drei vordefinierten Operationen (Methoden) zulässig sind:

- Initialisierung,
- Prolog P (kommt von protekt),
- Epilog V (kommt von vrej).

Operationen

Einsatz von Semaphoren

Beispiel Erzeuger-Verbraucher

Beispiel Philosophenproblem

Generated by Targeteam

Synchronisierung von Java Threads

Java unterstützt synchronisierte Methoden.

```
public synchronized void methodname(...) { ... }
```

Eine synchronisierte Methode kann nur exklusiv von einem Java Thread betreten werden.

Beispiel TakeANumber Class

Java Monitor

Ein Monitor ist ein Java-Objekt, das synchronisierte Methoden enthält.

Ein Monitor stellt sicher, dass nur ein Thread zur Zeit in einer der synchronisierten Methoden sein kann. Bei Aufruf einer synchronisierte Methode wird das Objekt gesperrt.

Während das Objekt gesperrt ist, können keine anderen synchronisierten Methoden des Objekts aufgerufen werden.

Kritische Abschnitte können in Java als Objekte mit den zugehörigen synchronisierten Methoden spezifiziert werden.

Generated by Targeteam

Nur ein Thread kann zu einem Zeitpunkt eine Nummer ziehen

```
class TakeANumber {
    private int next = 0; //next place in line
    public synchronized int nextNumber() {
       next = next + 1; return next;
    } //nextNumber
} //TakeANumber

public class Customer extends Thread {
    private static int number = 1000; //initial customer ID
    private int id;
    private TakeANumber takeANumber;
    public Customer(TakeANumber gadget) {
       id = ++number;
       takeANumber = gadget;
    } //Customer constructor
    public void run() {
       try {
          sleep( (int)(Math.random() * 1000) );
          System.out.println("Customer "+id+" takes ticket " +
          takeANumber.nextNumber());
       } catch ......
```

Beispiel TakeANumber Class

```
        next = next + 1; return next;
    } //nextNumber
} //TakeANumber

public class Customer extends Thread {
    private static int number = 1000; //initial customer ID
    private int id;
    private TakeANumber takeANumber;
    public Customer(TakeANumber gadget) {
        id = ++number;
        takeANumber = gadget;
    } //Customer constructor
    public void run() {
        try {
            sleep( (int)(Math.random() * 1000) );
            System.out.println("Customer "+id+" takes ticket " +
            takeANumber.nextNumber());
        } catch ......
    } //run
} //Customer

public class Bakery {
    public static void main(String args[]) {
```

Beispiel TakeANumber Class

```
        takeANumber = gadget;
    } //Customer constructor
    public void run() {
        try {
            sleep( (int)(Math.random() * 1000) );
            System.out.println("Customer "+id+" takes ticket " +
            takeANumber.nextNumber());
        } catch ......
    } //run
} //Customer

public class Bakery {
    public static void main(String args[]) {
        System.out.println("Starting Customer threads");
        TakeANumber numberGadget = new TakeANumber();
        .........
        for (int k = 0; k < 5; k++) {
            Customer customer = new Customer(numberGadget);
            customer.start();
        }
    } // main
} //Bakery
```

Java unterstützt synchronisierte Methoden.

```
public synchronized void methodname(...) { ... }
```

Eine synchronisierte Methode kann nur exklusiv von einem Java Thread betreten werden.

Beispiel TakeANumber Class

Java Monitor

Ein Monitor ist ein Java-Objekt, das synchronisierte Methoden enthält.

Ein Monitor stellt sicher, dass nur ein Thread zur Zeit in einer der synchronisierten Methoden sein kann. Bei Aufruf einer synchronisierte Methode wird das Objekt gesperrt.

Während das Objekt gesperrt ist, können keine anderen synchronisierten Methoden des Objekts aufgerufen werden.

Kritische Abschnitte können in Java als Objekte mit den zugehörigen synchronisierten Methoden spezifiziert werden.

Eine wichtige Systemeigenschaft betrifft die Synchronisation paralleler Ereignisse. Mehrere Prozesse (oder Threads) konkurrieren um eine gemeinsame Ressource (CPU), oder greifen auf gemeinsame Daten zu.

Beispiel: gemeinsame Daten

Synchronisierungskonzepte

Semaphore

Synchronisierung von Java Threads

Allgemeines

Es lässt sich zeigen, dass die folgenden Bedingungen ***notwendig und hinreichend*** dafür sind, dass eine Verklemmung auftreten kann.

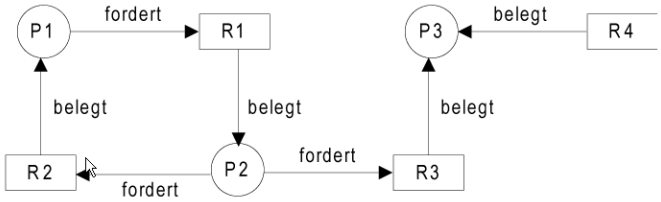
1. Die gemeinsam benutzbaren Ressourcen können nicht parallel genutzt werden, d.h. sie sind nur ***exklusiv*** benutzbar.
2. Die zugeteilten/belegten Ressourcen können ***nicht entzogen*** werden, d.h. die Nutzung ist nicht unterbrechbar.
3. Prozesse ***belegen*** die schon zugeteilten Ressourcen auch dann, wenn sie auf die Zuteilung weiterer Ressourcen warten, d.h. wenn sie weitere Ressourcen ***anfordern*** .
4. Es gibt eine ***zyklische Kette*** von Prozessen, von denen jeder mindestens eine Ressource belegt, die der nächste Prozess in der Kette benötigt, d.h. zirkuläre Wartebedingung.

Verklemmungen

Verklemmung = Prozesse warten wechselseitig auf das Eintreten von Bedingungen; gemeinsame Verwendung von Ressourcen (CPU, Speicherzellen, EA-Geräte, Dateien) kann zu Verklemmungen führen.

Beispiel

Seien P = {P1, ... , Pn} Prozesse und R= {R1, ... , Rm} Ressourcen, z.B. n = 3 und m = 4. Beispiel eines Belegungs/Anforderungsgraphen.



Verklemmung = Prozesse warten wechselseitig auf das Eintreten von Bedingungen; gemeinsame Verwendung von Ressourcen (CPU, Speicherzellen, EA-Geräte, Dateien) kann zu Verklemmungen führen.

Allgemeines

Belegungs-Anforderungsgraph

Die Zuteilung/Belegung und Anforderung von Ressourcen kann man sich an einem Graphen, dem Belegungs -Anforderungsgraph, veranschaulichen. Die Knoten sind die Prozesse und Ressourcen, die Kanten spiegeln Belegungen und Anforderungen wider.

Beispiel

Verklemmungs-Ignorierung

Vogel-Strauß-Politik und hoffen, dass nichts passiert.

Verklemmungs-Erkennung

Verklemmungs-Verhinderung

Verklemmungs-Vermeidung

Vergleich der Ansätze

In der Praxis häufig angewendete Strategie: Verklemmungen in Kauf nehmen, sie erkennen und beseitigen.

Erkennungs-Algorithmus

Ansatz 1: Suche nach Zyklen im Belegungs/Anforderungsgraph.

Ansatz 2: Prüfen, ob es eine Reihenfolge für die Ausführung der Prozesse gibt, so dass alle Prozesse terminieren können.

Vorgehen für Ansatz 2

1. Starte mit Prozessmenge P, die alle Prozesse enthält,
2. suche Prozess p aus P, dessen zusätzliche Anforderungen im aktuellen Zustand erfüllbar sind,
3. falls gefunden, simuliere, dass p seine belegten Ressourcen wieder freigibt,
4. entferne p aus P und gehe zu 2
5. falls kein Prozess mehr in P, dann terminiert Suche: keine Verklemmung,
6. falls $P \neq \emptyset$ und in Schritt 2 kein Prozess mehr gefunden wird, dessen Anforderungen erfüllbar sind, dann terminiert die Suche; P enthält die Menge der verklemmten Prozesse.

Auflösung einer Verklemmung in der Regel durch Abbruch einzelner Prozesse.

Verklemmungs-Verhinderung

Die Verhinderungssverfahren beruhen darauf, dass man durch die Festlegung von Regeln dafür sorgt, dass mindestens eine der für das Auftreten von Deadlocks notwendigen Bedingungen nicht erfüllt ist.

Festgelegte lineare Reihenfolge

Bedingung "Zyklus tritt auf" in Belegungs-/ Anforderungsgraph darf nicht erfüllt werden. Dazu wird eine lineare Ordnung über den Ressourcen definiert: R1 < R2 < ... Rm.

Die Prozesse dürfen dann Ressourcen nur gemäß dieser Ordnung anfordern,

d.h. ein Prozess, der Ressource Ri belegt, darf nur Ressourcen Rj anfordern, für die gilt: Rj > Ri.

Andere Möglichkeiten sind:

a) Zuteilung aller benötigten Ressourcen zu einem Zeitpunkt.

b) zwangsweiser Entzug aller belegter Ressourcen, falls eine Ressourcen-Anforderung nicht erfüllt werden kann.

c) Spooling: nur der Spooler Prozess hat als einziger die Ressource zugeteilt; Zugriffe anderer Prozesse gehen über diesen Prozess. Beispiel ist das Spooling von Druckaufträgen.

Die Vermeidungsverfahren basieren auf der Idee,

die zukünftigen Betriebsmittelanforderungen von Prozessen zu analysieren (bzw. diese geeignet abzuschätzen) und

solche Zustände zu verbieten (sie also zu verhindern), die zu Verklemmungen führen könnten.

Ein Beispiel ist der Bankiers-Algorithmus, der 1965 von Dijkstra entwickelt wurde.

Veranschaulichung des Algorithmus

Veranschaulichung des Verfahrens anhand eines Bankenszenarios.

Idee: Verwaltung von nur einer Ressourcen-Klasse, nämlich den Bankkrediten.

Bankier besitzt festen Geldbetrag und verleiht Geld an seine Kunden.

Alle Kunden sind dem Bankier bekannt, jeder Kunde hat einen eigenen maximalen Kreditrahmen, der kleiner als die zur Verfügung stehende Geldmenge des Bankiers ist.

Bankier hat weniger Geld als die Summe dieser Kreditrahmen.

Kunden können jederzeit Geld in der Höhe ihres jeweiligen Kreditrahmens fordern, müssen aber ggf. in Kauf nehmen, dass der Bankier diese Forderung erst nach endlicher Zeit erfüllt.

Aufgabe des Bankiers

Verleihen des Geldes so, dass jeder Kunde seine Geschäfte in endlicher Zeit durchführen kann und Kunden möglichst parallel bedient werden.

Idee : Reihenfolge für Kreditvergabe finden, so dass selbst bei denkbar ungünstigsten Kreditforderungen die Durchführung aller Geschäfte sichergestellt ist.

ungünstigster Fall : alle Kunden fordern Geld bis zu ihrem jeweiligen max. Kreditrahmen, ohne Kredite zurückzuzahlen.

Grobes Vorgehen

1. falls ein Kunde (Prozess) eine Anforderung hat, die aktuell erfüllbar ist, so teilt man das Geld (die Ressource) probeweise zu und
2. untersucht für den sich damit ergebenden Zustand, ob jetzt eine Verklemmung vorliegt, indem
3. für alle anderen Kunden von deren ungünstigsten Anforderungen ausgegangen wird und
4. ein Erkennungs-Algorithmus ausgeführt wird.

Falls keine Verklemmung auftritt, kann die Zuteilung tatsächlich erfolgen, anderenfalls könnte bei einer Zuteilung ein Deadlock auftreten (muss aber nicht), weshalb die Anforderung nicht erfüllt wird.

Beispiel

Ausgangspunkt ist die folgende Situation der vier Kunden A, B, C, D (Einheiten jeweils in Tausend Euro):

Kunde	aktueller Kredit	max. Kreditrahmen
A	1	6
B	1	5
C	1	4
D	4	7

Es seien noch 3 Einheiten (Tausend Euro) in der Bank als Kredit verfügbar.

Annahme: Kunde C fordere eine weitere Einheit als Kredit an. Diese Anforderung wird probeweise zugeteilt und mündet nicht in einem Deadlock, da zuerst C (max noch 2 Einheiten bis Kreditrahmen) bedient werden kann.

Wenn C seine Einheiten wieder zurückgezahlt hat, können B oder D und schließlich A bedient werden.

Ansatz	Verfahren	Vorteile	Nachteile
Erkennung	periodischer Aufruf des Erkennungs Algorithmus	Prozesserzeugung wird nicht verzögert; erleichtert interaktive Reaktion	Verlust durch Abbruch
Verhinderung	feste Reihefolge bei der Zuteilung	keine Verklemmungsanalyse zur Laufzeit notwendig; Überprüfung während Übersetzung	keine inkrementellen Anfragen für Ressourcen möglich
	Zuteilung aller Ressourcen auf einmal	keine Präemption (Entzug) von Ressourcen notwendig; gut für Prozesse mit einzelner Aktivitätsphase (single burst)	ineffizient; verzögert Prozesserzeugung; Bedarf für Ressourcen muss bekannt sein
Vermeidung	Bankiers-Algorithmus	keine Präemption (Entzug) von Ressourcen notwendig	zukünftiger Bedarf muss bekannt sein; Prozesse können längere Zeit blockiert werden

Teil I: Betriebssysteme und Systemsoftware

- Prof. J. Schlichter
  - Lehrstuhl für Angewandte Informatik / Kooperative Systeme, Fakultät für Informatik, TU München
  - Boltzmannstr. 3, 85748 Garching

    Email: schlichter@in.tum.de

    Tel.: 089-289 18654

    URL: http://www11.informatik.tu-muenchen.de/