S. Diehl

Java-Netzwerkprogrammierung am Beispiel einer einfachen verteilten, virtuellen Welt

Einleitung

Java bietet dem Programmierer ein reichhaltiges Angebot an Protokollen zur Netzwerkkommunikation. Diese reichen von HTTP-Verbindungen, über TCP, UDP und Multicast UDP, bishin zu verteilten Objekten, die mit Hilfe von Remote Methode Invocation RMI und CORBA implementiert werden können. 3D-Browser, die auf dem VRML97 Standard beruhen, bieten Schnittstellen, wie das External Authoring Interface EAI, zu benutzerprogrammierten Java-Programmen. Dadurch ist es möglich, die 3D-Szene von Java [4] aus zu steuern und sie über Java-Netzwerkprogrammierung mit anderen Instanzen der Szene auf anderen Rechnern im Netz kommunizieren zu lassen. In diesem Artikel stellen wir die Architektur und einige Implementierungsdetails einer solchen verteilten, dreidimensionalen Szene vor.

Miss Connectivity

In Anlehnung an Miss Liberty wurde Miss Connectivity entwickelt als ein Symbol für den freien, weltweiten Zugriff auf Daten im Internet und die Vielzahl an Möglichkeiten über das Internet zusammenzuarbeiten. Ähnlich wie sich die Informationen im Internet ständig verändern, so verändert sich auch das Erscheinungsbild von MissC. Jeder Besucher kann nicht nur MissC besichtigen, sondern er kann auch ihr Aussehen ändern. Jede Änderung wird sofort an alle anderen Besucher propagiert und die

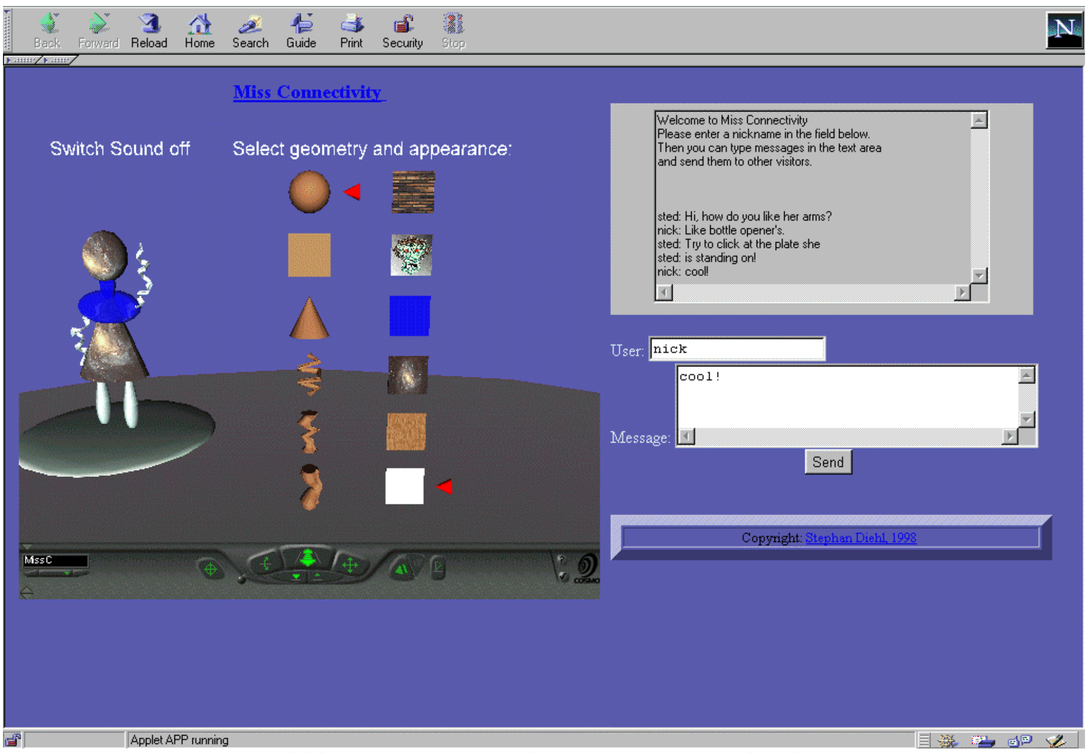

Abbildung 1: Screenshot of MissC's Webseite

Benutzer können mittels eines Textchats ihre Änderungen an MissC koordinieren. MissC gehört zu den ersten dreidimensionalen, interaktiven, kollaborativen Kunstwerken im Internet. Sicherlich werden wir mehr ihrer Art sehen, sobald die zugrundeliegende Technologie (Netscape 4.0.x, SGI CosmoPlayer 2.x

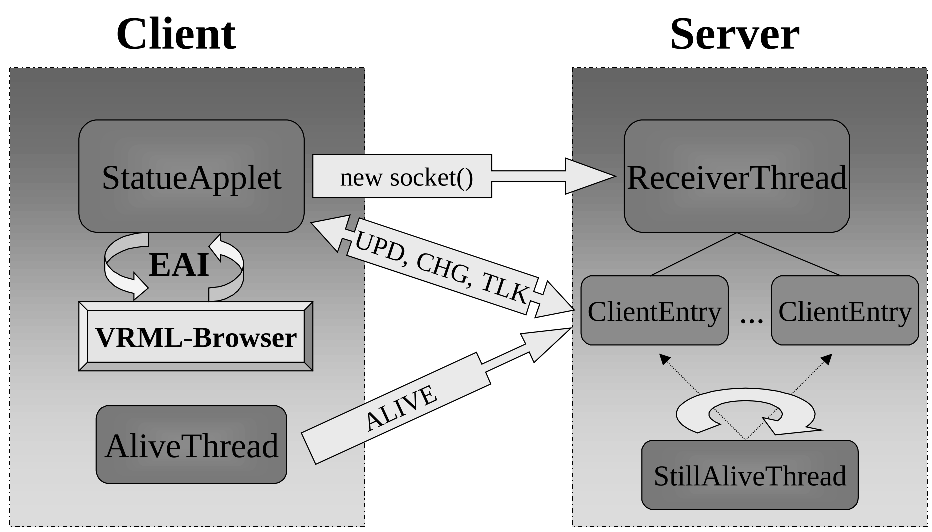

Abbildung 2: Architektur von MissC

PlugIn und Java JDK1.1) reifer wird und mehr Verbreitung findet. MissC kann auf der Webseite http://www.cs.uni-sb.de/~diehl/Statue/MissC.html besichtigt werden. In erster Linie dient MissC allerdings Ausbildungszwecken, da ihr Programmcode überschaubar ist, jedoch eine Vielzahl verschiedener Webtechnologien vereinigt: HTML, HTTP, Java-Networking, Java-Applets, Java-Threads, VRML und das EAI.

Die Architektur von MissC

MissC verwendet eine Client/Server-Architektur. Sie wird als HTML-Seite von einem Webserver geladen. In dieses HTML-Dokument sind eine 3D-Szene (mit `<EMBED>`) und ein Java-Applet (mit `<APPLET>`) eingebunden. In Abbildung 2 sind die Architektur und das Protokoll von MissC dargestellt. Das Java-Applet initiiert die Kommunikation mit dem Server durch Öffnen eines TCP-Sockets für den Serverport, auf dem der Server nach Anfragen lauscht. Der Server erzeugt daraufhin einen Eintrag der Klasse `ClientEntry` und einen zugehörigen Thread zur Kommunikation mit dem anfragenden Client. Diesem Thread wird ein neuer Port zugeordnet und der Client und der ihm auf dem Server zugeordnete `ClientEntry`-Thread kommunizieren nun nur noch über diesen Port.

Zuerst schickt der Server den aktuellen Zustand (`UPD`) von MissC, d.h. für jeden ihrer Körperteile enthält die Nachricht das aktuelle Aussehen (Farbe, Textur) und die aktuelle Geometrie (Kugel, Box, Korkenzieher). Ändert der Benutzer ein Teil der Statue, dann schickt der Client eine Nachricht (`CHG`) mit der Nummer des geänderten Körperteils und des gewählten Aussehens und der gewählten Geometrie an den Server. Dieser ändert intern den Zustand der Statue und sendet die empfangene Nachricht allen anderen Clients weiter. Tippt der Benutzer eine Textzeile im Chatfenster ein, so schickt der Client eine Nachricht (`TLK`) an den Server, der diese an alle Clients weiterreicht. Die im Protokoll vorgesehenen Nachrichten sind wie folgt:

```
UPD APP1;GEO1;APP1;GEO1; …;APP2;GEO3;
CHG 1:APP1;GEO3;
TLK sted: Try to click at the plate!
ALIVE
KILL ME
```

Totgesagte leben länger

Da der Server lange Zeit, ideallerweise unendlich lange, laufen soll, aber dabei nicht mehr benötigte Ressourcen wieder freigeben sollte, müssen Vorkehrungen getroffen werden, um festzustellen, wann ein Client die Verbindung beendet. Im einfachsten Fall teilt der Client dies explizit mit (`KILL ME`). Man kann aber leider nicht davon ausgehen, daß ein Client immer zum Absenden einer solchen Nachricht vor Beendigung der Kommunikation kommt, z.B. könnte ein Benutzer seinen Rechner zuvor ausschalten. Daher sendet jeder Client in regelmäßigen Abständen Lebenszeichen (`ALIVE`), manchmal auch als Herzschlag (heartbeat) bezeichnet. Auf dem Server läuft ein Thread der Klasse `StillAliveThread`, der alle Clients der Reihe nach überprüft, wann sie das letzte Lebenszeichen von sich gegeben haben. Ist dies zu lange her, wird der `ClientEntry`-Thread für diesen Clients beendet und der Socket (Port) wieder freigegeben.

Einige Implementierungsdetails für TCP mit Java

Wir beschränken uns aus Platzgründen hier auf den Server. Zur besseren Lesbarkeit des Quellkodes haben wir im Folgenden den Programmtext für die Ausnahmebehandlung entfernt.

Schleife des Server-Threads:

```
public void run() {
 StillAliveThread checkThread=new StillAliveThread(this);
 acceptsock=new ServerSocket(3000);
 while(true) { newsock = acceptsock.accept();
               clients.addElement(new ClientEntry(newsock));
               Thread.yield(); }
 }
```

Der Server erzeugt zuerst einen ServerSocket, um auf dem Port 3000 Anfragen für neue Verbindungen zu empfangen (`accept()`). Liegt eine Anfrage vor, erzeugt er einen neuen `ClientEntry` und übergibt an diesen, den für die neue Verbindung vorgesehenen Socket `newsock`. Außerdem fügt er diesen `ClientEntry` der Liste der aktuellen Clients hinzu.

Konstruktor des `ClientEntry`-Threads:

```
public ClientEntry(Socket newsock) {
  tcpsock=newsock;
  connect();
  this.start();
}

public  void connect() {
   tcpinput = new BufferedReader(
               new InputStreamReader(tcpsock.getInputStream()));
   tcpoutput = new PrintWriter(tcpsock.getOutputStream(),true);
   String update="UPD ";
```

```
    update = … // aktueller Zustand
    tcpoutput.println(update);
  }
```

Beim Erzeugen eines neuen `ClientEntry` speichert der Konstruktor den für diesen Client vorgesehenen Socket in der Objektvariablen `tcpsock`. Dann öffnet er einen Eingabe- und einen Ausgabestrom, um vom Client gesendete Daten zu lesen und Nachrichten an den Client zu schicken. Als erste Nachricht schickt der Konstruktor den aktuellen Zustand, indem er mit `println()` auf den Ausgabestrom schreibt.

Die Schleife des `ClientEntry`-Threads:

```
public void run() {
 boolean alive=true;
 lastAlive=System.currentTimeMillis();
 String str;
 while (alive)
 { str= tcpinput.readLine();
   if (str==null) {
    if (System.currentTimeMillis()-lastAlive)>1000) alive=false;
    Thread.yield();
    continue; }
   if (str.equals("KILL ME")) { lastAlive=(long) 0;
                                alive=false;}
   if (str.equals("ALIVE"))
      lastAlive=System.currentTimeMillis();
   if (str.indexOf("TLK")==0)
      StatueServer.sendUpdates(null,str);
   if (str.indexOf("CHG")==0)
        { StatueServer.sendUpdates(this,str);
          // ändere im aktuellen Zustand }
   Thread.yield(); }
}
```

Der `ClientEntry`-Thread liest Nachrichten vom Clients mit `readline()` aus dem Eingabestrom. Je nach Inhalt der Nachricht führt er dann verschiedene Aktionen aus. Die ersten drei IF-Anweisungen behandeln den Fall, daß die Verbindung beendet wurde. Die Nachrichten `TLK` und `CHG` werden mit der Methode `sendUpdates()` des Servers an alle anderen Clients weitergeschickt. Im Falle von `TLK` auch an den Clients selbst:

```
static void sendUpdates(ClientEntry sender, String str)
{ int i;
  synchronized (clients)
  { for(i=0; i<clients.size();i++)
    { ClientEntry cl=(ClientEntry) clients.elementAt(i);
      if (cl!=sender) cl.tcpoutput.println(str);
    }
  }
}
```

Da mehrere Clients gleichzeitig Nachrichten an den Server schicken können und die zugehörigen `ClientEntry`-Threads gleichzeitig laufen, muß man dafür sorgen, daß sie nicht konkurrierende Nachrichten an die anderen Clients schicken. Dies wird hier dadurch erreicht, daß die Sendeschleife über die Klassenvariable `clients` des Servers synchronisiert wird.

Kommunikation zwischen VRML-Browser und Java

MissC besteht im wesentlichen aus drei Dateien: einer HTML-Datei, einer VRML-Datei und einer Java-Datei (Applet). Sie zusammen implementieren eine Webseite, in der man eine dreidimensionale Statue manipulieren und Textnachrichten eintippen kann und diese Änderungen und Nachrichten werden an alle Teilnehmer weitergeleitet, die die gleiche Seite geladen haben. MissC verwendet HTML, JavaScript, Java, VRML, das EAI und Java. Wer mehr zu diesen Webtechnologien wissen will, findet in "Java&Co" [1] dazu die wichtigsten Informationen.

Eine zentrale Rolle in 3D-Grafiksystemen spielt der Szenengraph. Die Anzeigesoftware traversiert diesen Graphen und akkumuliert entlang eines jeden Pfades, die in den Knoten enthaltenen Informationen, etwa Transformationen, Farbinformation und geometrische Daten. Ein Knoten kann über verschiedene Pfade erreicht werden, d.h. er kann in unterschiedlichen Kontexten ausgewertet werden. In klassischer 3D-Software wurde der Szenengraph im Programmcode des 3D-Systems selbst aufgebaut, er war also in das 3D-System integriert. Mit Hilfe von VRML [5] kann ein Szenengraph unabhängig von der Implementierung des Grafiksystems angegeben werden. Die Konstrukte von VRML heißen in Anlehnung an den durch sie spezifizierten Szenengraphen Knoten und Felder, wobei Felder den Kanten im Graphen entsprechen. Einen Knoten definiert man durch Angabe des Knotentyps und der Werte für die Felder des Knotens. Diese Werte können wieder Knoten sein.

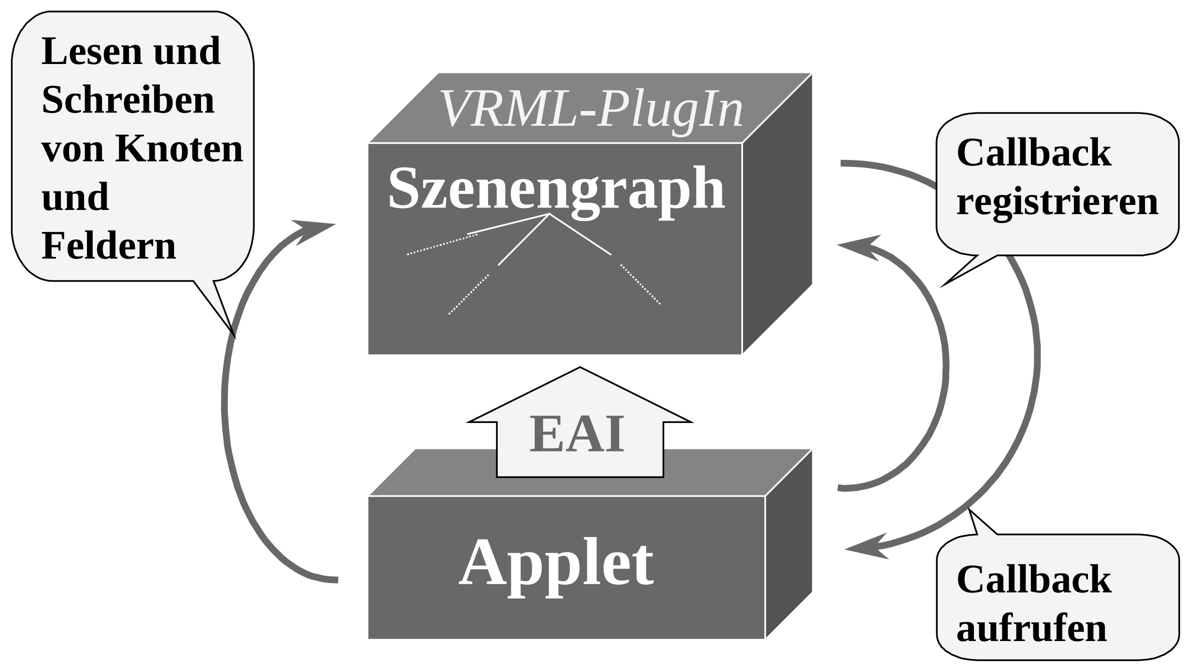

Abbildung 3: Kommunikation zwischen Applet und VRML-Browser

Das External Authoring Interface (EAI) [3] ist eine Programmierschnittstelle über die externe Programme mit dem VRML-Browser kommunizieren können. In MissC ist ein Java-Applet ein solches externes Programm. Das EAI erfüllt zwei wichtige Aufgaben:

- Es läßt das externe Programm lesend und schreibend auf den Szenengraph zugreifen.
- Es erlaubt dem externen Programm eigene Funktionen, sogenannte Callbacks, für bestimmte Events anzumelden. Der VRML-Browser ruft diese auf, sobald der entsprechende Event auftritt.

Alternativen zur Implementierung des Protokolls

Java bietet außer TCP auch andere Protokolle/Architekturen zur Netzwerkprogrammierung: URL (CGI-Skripte), UDP, Multicast UDP, RMI (Remote Methode Invocation) und CORBA (JavaIDL).

URL

Mit URL kann man eine Verbindung mit einem HTTP-Server aufbauen. Es wird eine Anfrage geschickt, der Server leitet diese an ein CGI-Skript weiter und dieses verarbeitet die Eingaben und schickt eine Antwort. Der Vorteil an URL besteht darin, daß auch Rechner hinter einer Firewall, die nur HTTP-Zugriffe erlaubt, MissC benutzen können. Dazu könnte man zwei CGI-Skripte schreiben: `sendUpdate` und `getUpdate`. Die Clients müssen regelmäßig das `getUpdate`-CGI-Skript aufrufen, um die letzten Nachrichten der anderen Clients zu erhalten, d.h. der Server schickt nicht wie bei TCP die Nachrichten automatisch an alle Clients weiter. Ähnlich wie TCP-Sockets bietet auch eine URL-Verbindung Ein- und Ausgabeströme. Im Gegensatz zur TCP-Verbindung besteht die URL-Verbindung nur für die Dauer einer Übertragung, d.h. für jede Übertragung müssen die Ströme neu geöffnet werden. Auf dem Client könnte dann folgende Methode Nachrichten an den Server schicken.

```
public void send(String upd) {
 URL script =
   new URL("http://www.dings.com/cgi-bin/sendUpdate");
 URLConnection uc = script.openConnection();
 uc.setDoOutput(true);
 PrintWriter out = new PrintWriter(uc.getOutputStream());
 out.println(upd);
 out.close(); }
```

UDP und Multicasting UDP

Die Verwendung von UDP hat den Vorteil der höheren Übertragungsgeschwindigkeit und - bei Verwendung von Multicasting – der besseren Lastverteilung. Java sieht die Methoden `send()` und `receive()` zum Senden und Empfangen von UDP-Paketen vor. Um UDP einfach in die bestehende MissC-Implementierung einzubauen, könnte man eigene Ein- und Ausgabeströme implementieren, die Zeichenketten automatisch in UDP-Pakete zerlegen und umgekehrt. Verwendet man Multicasting so schickt man die Nachricht nicht mehr direkt an einen Empfänger, sondern an eine Multicastgruppe, d.h. eine spezielle IP-Adresse (224.0.0.0 bis 239.255.255.255). Zum Empfangen von Nachrichten muß ein Empfänger zuerst Mitglied einer Multicastgruppe werden (mit der Methode `joinGroup()`) und am Ende einer Sitzung wieder aus der Gruppe austreten (mit der Methode `leaveGroup()`). Eine entsprechende Methode eines Clients zum Empfangen von Nachrichten sieht wie folgt aus:

```
public String receive() {
  MulticastSocket socket = new MulticastSocket(4446);
  InetAddress group = InetAddress.getByName("230.0.0.1");
  socket.joinGroup(group);
  DatagramPacket packet
    = new DatagramPacket(buf, buf.length);
  socket.receive(packet);
  String msg = new String(packet.getData());
  socket.leaveGroup(group);
  socket.close();
  return msg; }
```

Verteilte Objekte (RMI oder CORBA)

Mit RMI oder CORBA ersetzt man die ASCII-Befehle des Protokolls durch Methodenaufrufe. Das Objekt, dessen Methode man aufruft, befindet sich dabei auf einem anderen Rechner und die Argumente des Methodenaufrufs werden zu diesem Rechner übertragen, die Methode dann dort ausgeführt und das Ergebnis wieder zurück an den aufrufenden Rechner übertragen. Für einfache Anwendungen verlangt RMI und vor allem CORBA zuviel Einarbeitungszeit, bietet aber auch interessante Vorteile (siehe [2]), etwa das Erweitern von Protokollen durch Vererbung und die dynamische Protokollverhandlung.

Der einfachste Ansatz besteht darin ein Anwendungsserverobjekt zu implementieren, das eine Methode `sendUpdate()` und eine Methode `getUpdate()` hat. Das Serverobjekt existiert auf dem Server und auf dem Client wird über ein Proxyobjekt, das manchmal auch als Stub (Stumpf) bezeichnet wird, auf das Serverobjekt zugegriffen. Server- und Proxyobjekt müssen das gleiche, anwendungsspezifische Interface implementieren:

```
interface SimpleServe extends java.rmi.Remote
   { public String getUpdate();
     public void sendUpdate(String upd); }
```

Auf dem Server wird eine Klasse definiert, die das eigentliche Serverobjekt implementiert (hier stark vereinfacht). Dann wird ein solches Objekt erzeugt und bei einem Naming-Service registriert. In unserem Beispiel ist das erzeugte Objekt unter dem Namen `“SimServObj”` eingetragen.

```
class OurServer extends java.rmi.UnicastRemoteObject
      implements SimpleServe
 { String update;
   public String getUpdate() { return(update); }
   public void sendUpdate(String upd) { update=upd; }
 }

myinst=new OurServer();
Naming.rebind(“//names.host.com/SimServObj”,myinst);
```

Der Client fragt beim Naming-Service nach, ob ein Objekt unter dem Namen `SimServObj` eingetragen ist und erzeugt dann das zugehörige Proxyobjekt. Das Proxyobjekt wird explizit in den richtigen Typ umgewandelt, so daß der Client die in der Schnittstelle deklarierten Methoden aufrufen kann:

```
SimpleServe stub =
(SimpleServe)Naming.lookup("http://names.host.com/SimServObj");

public void send(String upd) { stub.sendUpdate(upd); }
```

Zusammenfassung

In diesem Papier haben wir die Architektur einer einfachen, verteilten, dreidimensionalen Szene vorgestellt. Für die Beschreibung der Szene wurde VRML, für die Implementierung der Netzwerkkommunikation Java verwendet. Bei der Darstellung wurde der Schwerpunkt auf die verschiedenen Möglichkeiten gelegt, die Java zur Netzwerkkommunikation bietet.

Literatur

[1] Stephan Diehl, Java & Co, Addison-Wesley, 1997
[2] Stephan Diehl, Towards Lean and Open Multi-User Technologies, International Symposium on Internet Technologies ISIT98, Taipei, Taiwan, 1998
[3] Marrin Chris, External Authoring Interface Reference, Silicon Graphics Inc., http://www.vrml.org/WorkingGroups/vrml-eai/ExternalInterface.html
[4] Sun, Dokumentation zu Java und seinen APIs, http://www.java.sun.com/docs/index.html
[5] VRML-Konsortium, Spezifikationen und Arbeitsgruppen, http://www.vrml.org

Autor:
Dr. Diehl, Stephan
Universität des Saarlandes, Fachbereich Informatik, Postfach 15 11 50,
66041 Saarbrücken
Tel.: 0681-302-3915
e-mail: diehl@cs.uni-sb.de , web: http://www.cs.uni-sb.de/~diehl