

Seminar „Visualisierung in Informatik und  
Naturwissenschaften“ im SS 1999

# Visualisierung paralleler bzw. verteilter Programme

Holger Dewes

# Gliederung

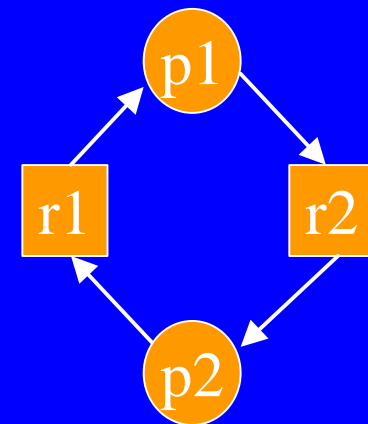
- Zum Begriff
- Motivation
- PARADE
  - Beispiel 1: Thread basierte Programme
  - Beispiel 2: Conch
- GRADE
- Zusammenfassung

# Zum Begriff

- Parallele/Verteilte Programme:  
Programme, die auf mehreren Prozessoren und/oder Rechnern zur Performancesteigerung laufen.
- Zwei Arten:
  - message passing
  - shared memory

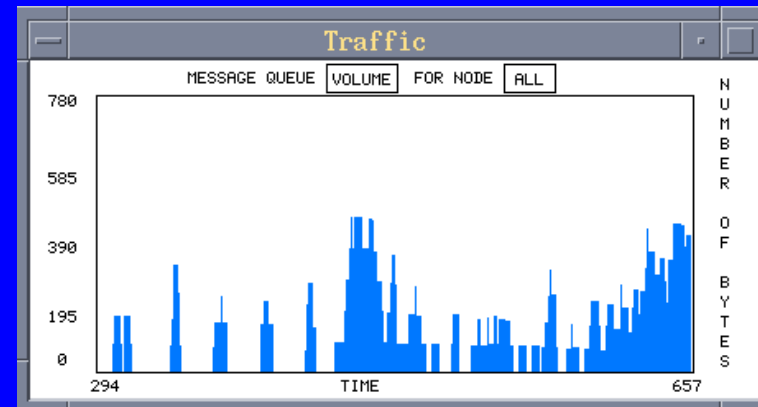
# Motivation

- *Gleichzeitigkeit* paralleler Programme (z. B. Kommunikation) textuell kaum zu veranschaulichen
- viele Probleme paralleler Programmierung sind *inhärent visuell* (z. B. deadlocks)
- *Nichtdeterminismus* erschwert Fehlersuche und erfordert genaues Verständnis des Programms



# Variationen

- Visualisierung der *Performance* (relativ verbreitet)
- Visualisierung der *Semantik* (nicht sehr verbreitet)  
Schwerpunkt dieses Vortrags
- *online*  
Vorteil: Visualisierung während der Ausführung
- *post-mortem*  
Vorteile: geringe Beeinflussung des Programms, mehr Flexibilität bei der Visualisierung



# PARADE

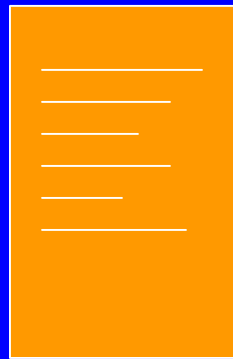
(PARAllel program Animation Development Environment)

Stasko et al., Georgia Institute of Technology

- Flexibles Framework:  
bietet Standardlösungen als auch  
applikationsspezifische Visualisierung
- sowohl semantische als auch Performance-  
Visualisierung
- post-mortem

# PARADE - Architektur

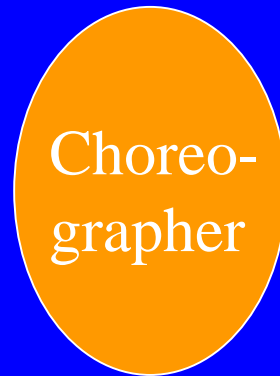
Programm



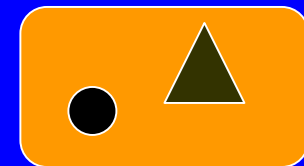
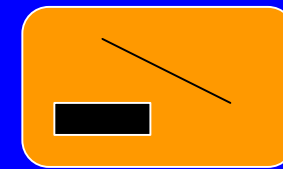
Events



Choreo-  
grapher



Scenes



Sammlung von  
trace Daten

Analyse

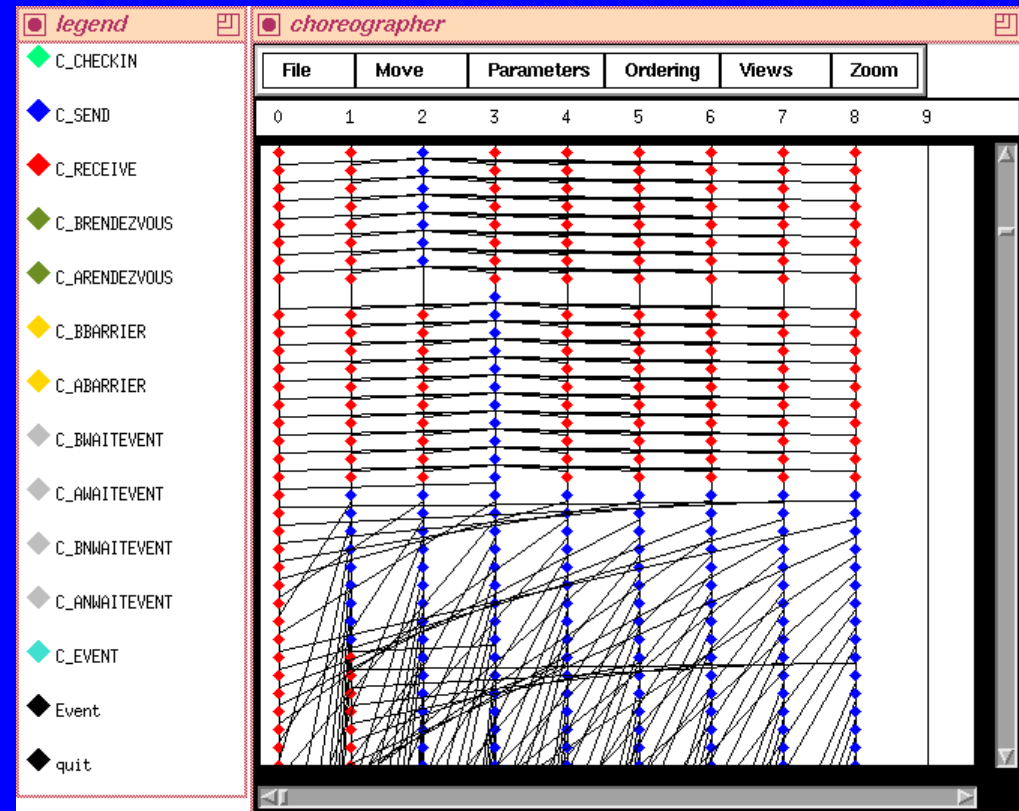
Polka  
Animationen

# Animation Choreographer

- verantwortlich für zeitliche Einordnung der Events (Program-to-Animation Mapping)
- 4 Arten der Sortierung:
  - Timestamp: geordnet nach einer globalen Uhr
  - Serial: seriell nach kausaler Ordnung
  - Minimal distortion: nach globaler Uhr, aber löse dabei kausale/logische Probleme
  - Maximum concurrency: maximale Nebenläufigkeit unter Berücksichtigung der kausalen Ordnung

# Animation Choreographer

- eine Spalte pro Prozeß/Thread
- ein Knoten pro Event
- Kanten für Abhängigkeiten zwischen Events (z.B. send-receive Paar)



# POLKA

(Parallel program-focused Object-oriented Low Key Animation)

- objekt-orientiert (C++)
- high-level Interface
- nebenläufige Animationen
- unterstützt mehrere *Views* auf ein Programm

# Beispiel: Thread basierte Programme

- Basiert auf KSR Pthreads interface (POSIX Threads)
- wrapper library Gthreads sammelt trace Daten
- Funktionsanfänge und -ende müssen von Hand durch Makroaufrufe signalisiert werden

# Thread basierte Programme

The screenshot displays a debugger interface with several panels:

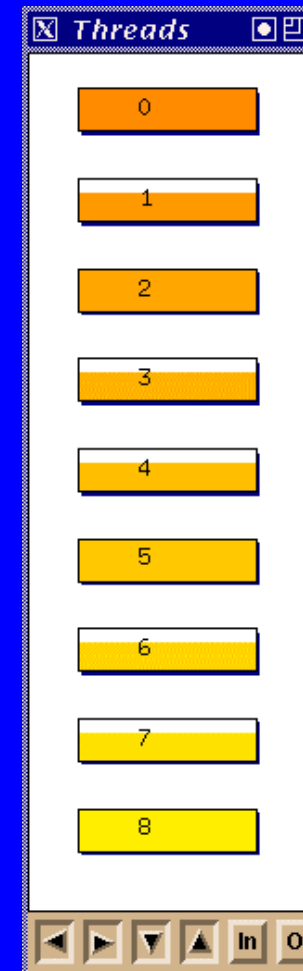
- Polka Control Panel:** Contains control buttons for **Run**, **Step**, **Slow**, **Fast**, and **Quit**.
- Invocation History:** A Gantt chart showing the execution timeline of 9 threads (0-8). Thread 0 is the main thread, and threads 1-8 are spawned. Vertical dashed lines indicate synchronization points.
- Threads:** A list of 9 thread slots, numbered 0 to 8, with progress indicators.
- Functions:** A call stack diagram showing the execution flow: **main** calls **setUp**, which calls **iterate**. The **iterate** function branches into **findBest** and **updateClosest**.
- Mutex 0x81D6600:** A window showing a diagram of a mutex with two orange circles representing threads waiting for the lock.
- Barrier 0x81FDB88:** A table showing the state of a barrier across 9 threads (0-8) in four different stages. The 'In' and 'Out' rows represent thread status.

**Barrier 0x81FDB88 Data:**

	0	1	2	3	4	5	6	7	8
In	0	1	2	3	4	5	6	7	8
Out	0	1	2	3	4	5	6	7	8
In	0	1	2	3	4	5	6	7	8
Out	0	1	2	3	4	5	6	7	8
In	0	1	2	3	4	5	6	7	8
Out	0	1	2	3	4	5	6	7	8

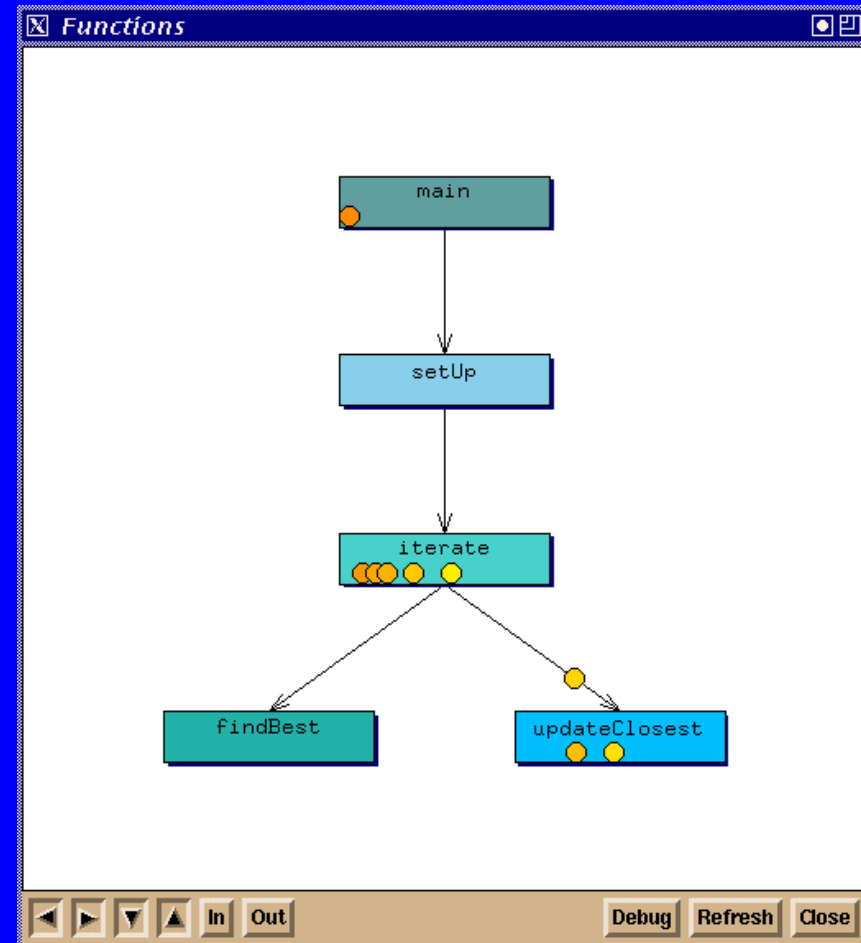
# Thread View

- ein Rechteck pro Thread
- ausgefüllt: bereit zu laufen bzw. läuft
- nicht ausgefüllt: idle oder blockiert



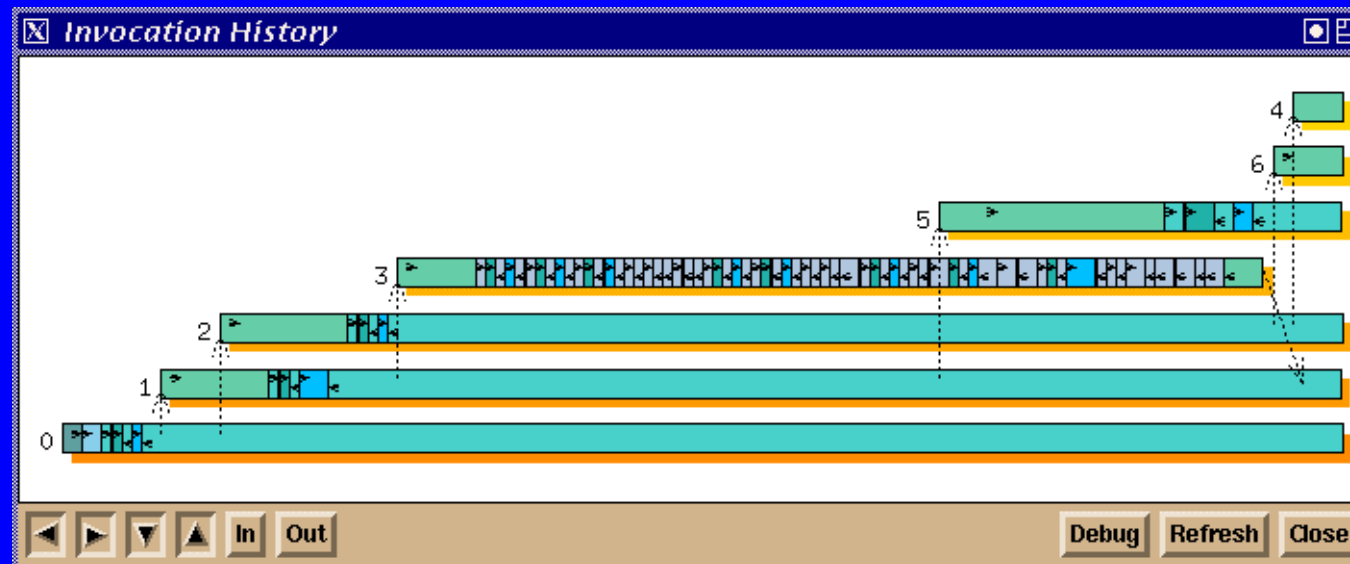
# Function View

- ein Rechteck pro Funktion (wird dynamisch erzeugt)
- ein Kreis pro Thread



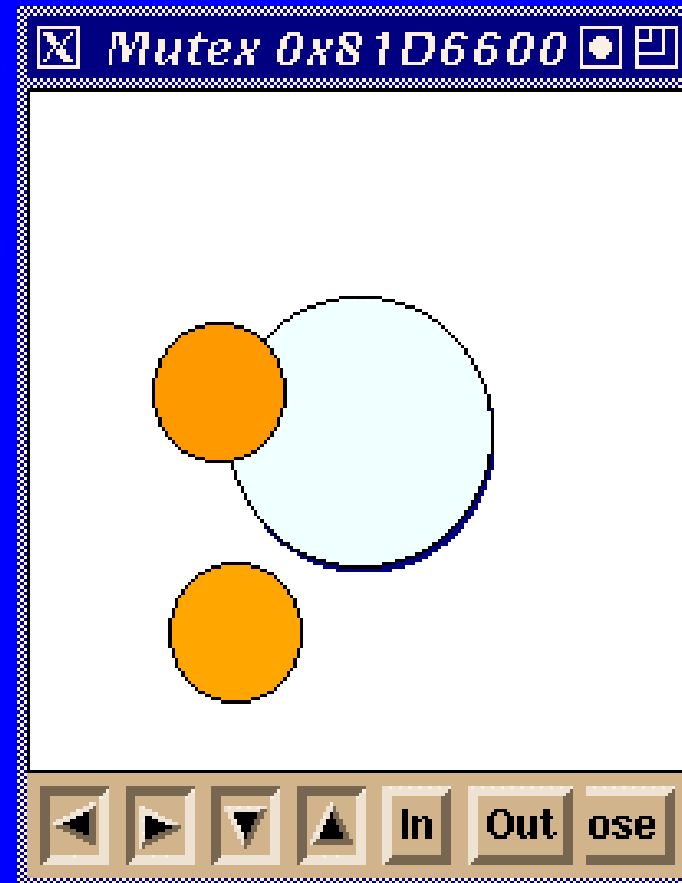
# History View

- eine Reihe pro Thread
- ein Segment pro Funktion
- Pfeile zeigen Thread Erzeugung und Thread joining



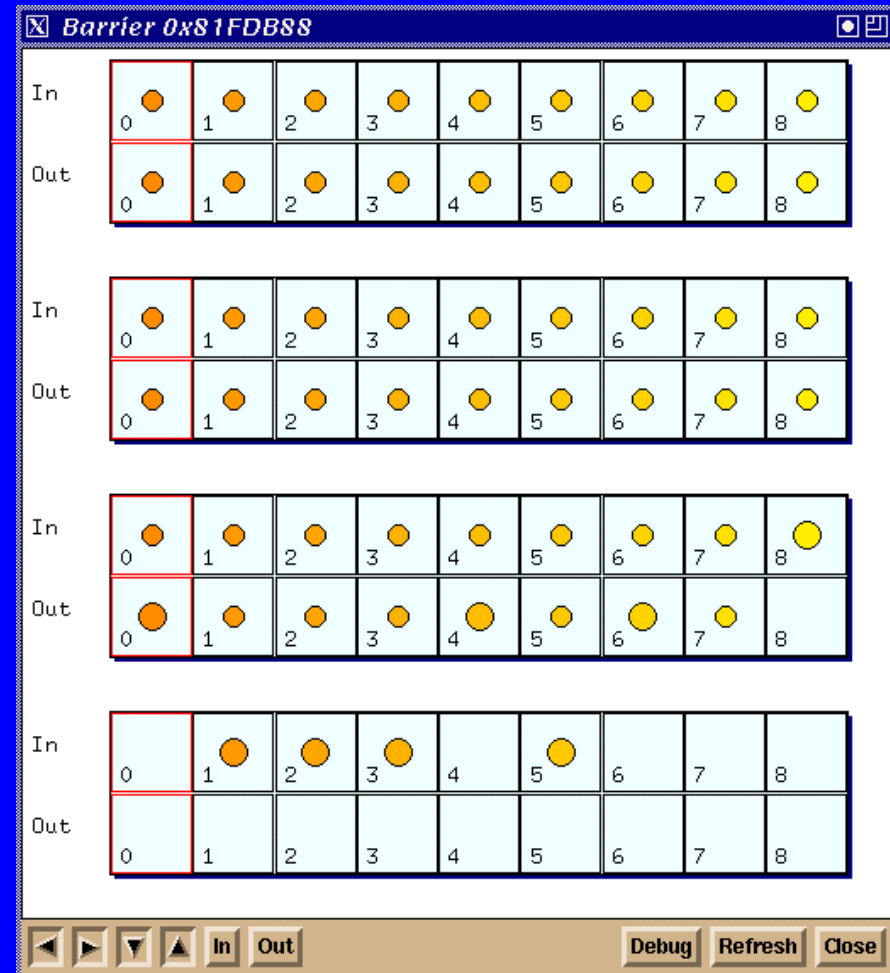
# Mutex View

- ein großer Kreis pro Mutex (dynamisch)
- ein kleiner Kreis pro Thread
- geeignet, um deadlocks zu erkennen?



# Barrier View

- ein Fenster pro Barrier (dynamisch)
- ein Block pro Barrier Synchronisierungsphase
- ein Kreis pro Thread:
  - groß: noch in Phase
  - klein: Phase abgeschlossen



# Beispiel: Conch

- Heterogenes Netzwerk für paralleles verteiltes Rechnen
- trace Datensammlung eingebaut
- Lamport logical clock für timestamping
- Applikationsspezifische Visualisierung möglich

# Conch

**Polka Control Panel**

Run Step Slow Fast

**Conch History View**

13										
12										
11										
10										
9										
8										
7										
6			0	5	3			6		
			777	777	777			777		
5		7	8	4	2			4	5	
		555	555	777	555			555	555	
4	6		5	0	1	8	3	1		
	555		555	555	555	555	555	555		
3	3	4	0	6	7	4	0	8	2	
	555	555	777	555	555	777	555	777	555	
2	2	0		4	5			8	7	
	777	777		777	777			777	777	
1	1				3				6	
	777				777				777	
0	Proc	Proc	Proc	Proc	Proc	Proc	Proc	Proc	Proc	Proc
	0	1	2	3	4	5	6	7	8	FE

**Conch Global View**

0 1 2 3 4 5 6 7 8 FE

Busy Send Rcv Rendezvous Barrier (N)Waitevent Event

**Conch Message Passing**

**Conch Lamport View**

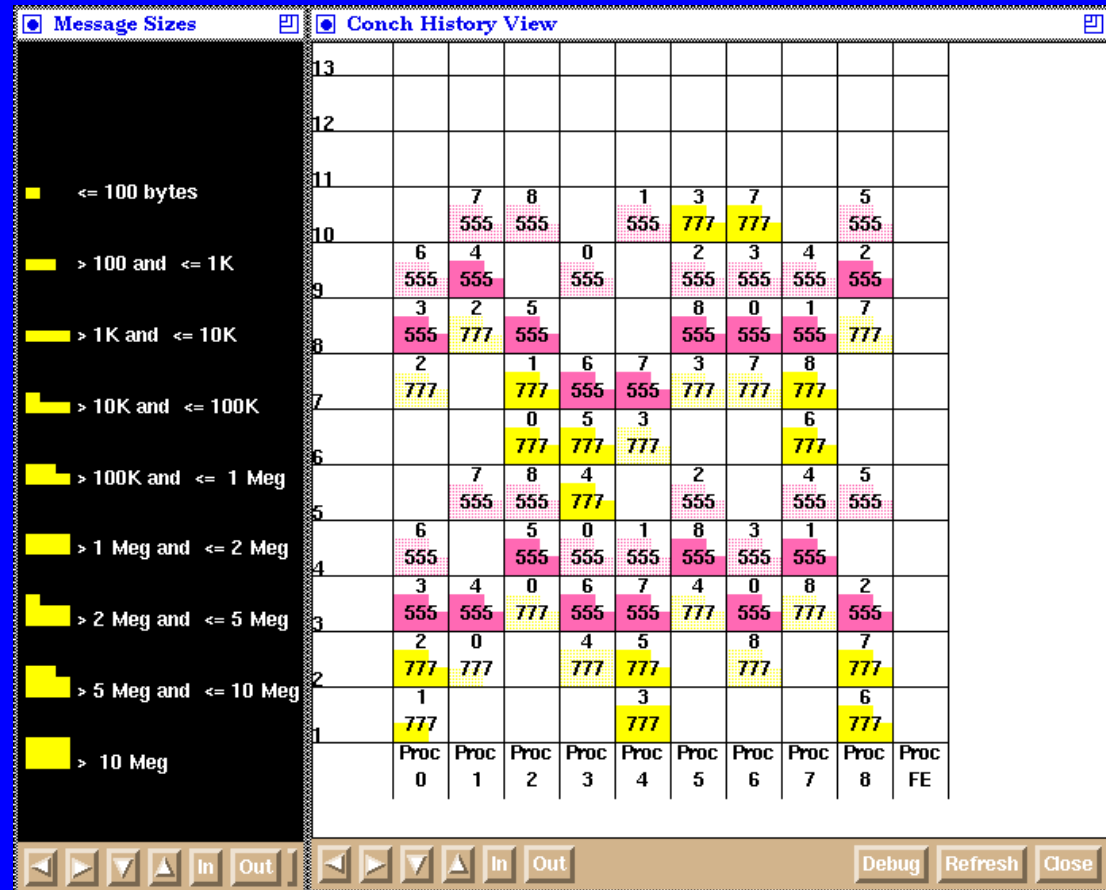
# Global View

- zeigt Zustand der Prozesse an



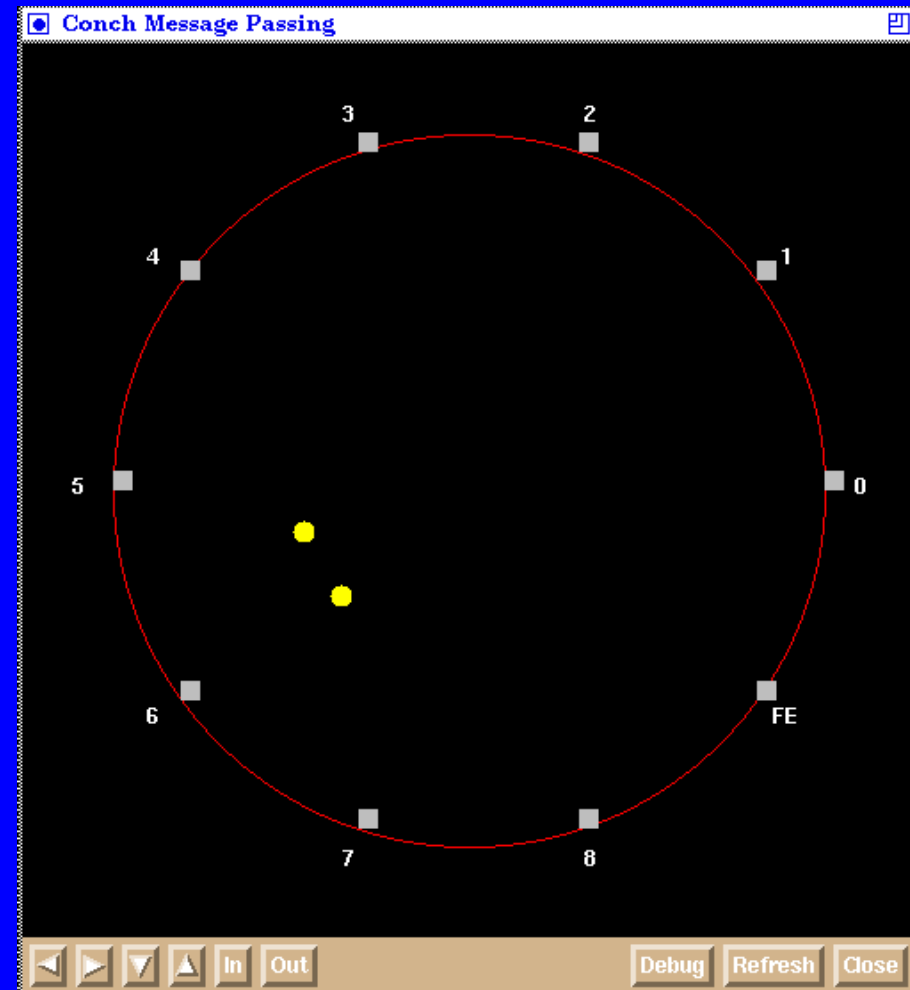
# History View

- Rechtecke sind Nachrichten
  - dunkle Farbe: message send
  - helle Farbe: message receive
- Zeigt Größe und Nachrichtentyp



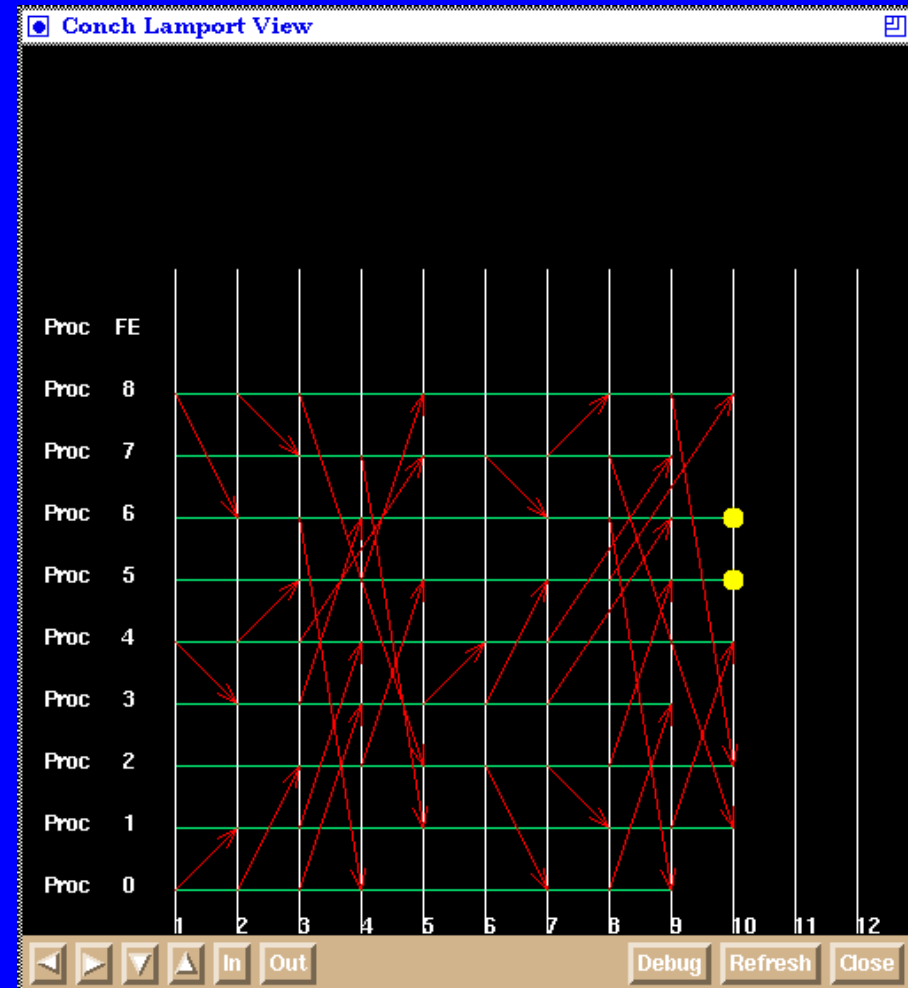
# Message Passing View

- ein Rechteck pro Prozeß
- ein Kreis pro Nachricht (Radius proportional zu Nachrichtengröße)
- message send: Nachricht wandert ins Zentrum
- message receive: Nachricht wandert zum Prozeß



# Lamport View

- Prozesse „wachsen“ horizontal zur Lamport Uhr
- ein Pfeil pro message send/receive Paar
- alternative Sicht zur History View



# GRADE

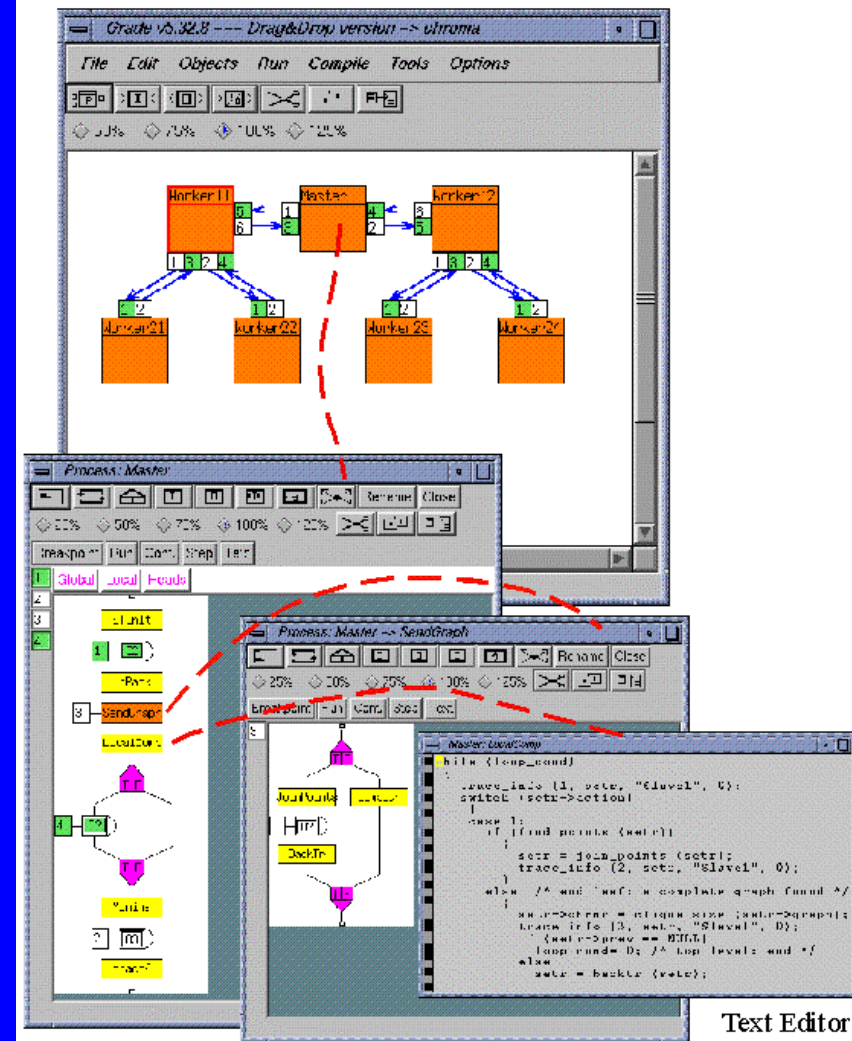
(Graphical Application Development Environment)  
KFKI-MSZKI Institut Budapest, Universität Lissabon

- visuelles Entwicklungssystem für verteilte Programme (PVM: message based)
- umfaßt:
  - GRED: graphischer Editor
  - DDBG: verteilter Debugger
  - PROVE: Visualisierer (haupts. Performance)

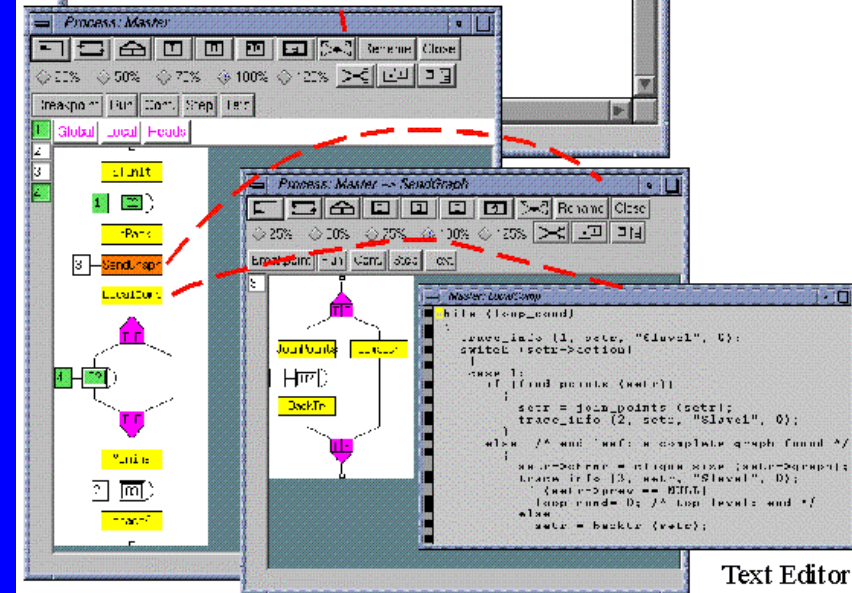
# GRED

- 3 Ebenen:
  - Application Level
  - Process Level
  - Textual Code Level
- richtet Fokus auf die Teile des Programms, die Nachrichten versenden/empfangen

Application Window



Process Window



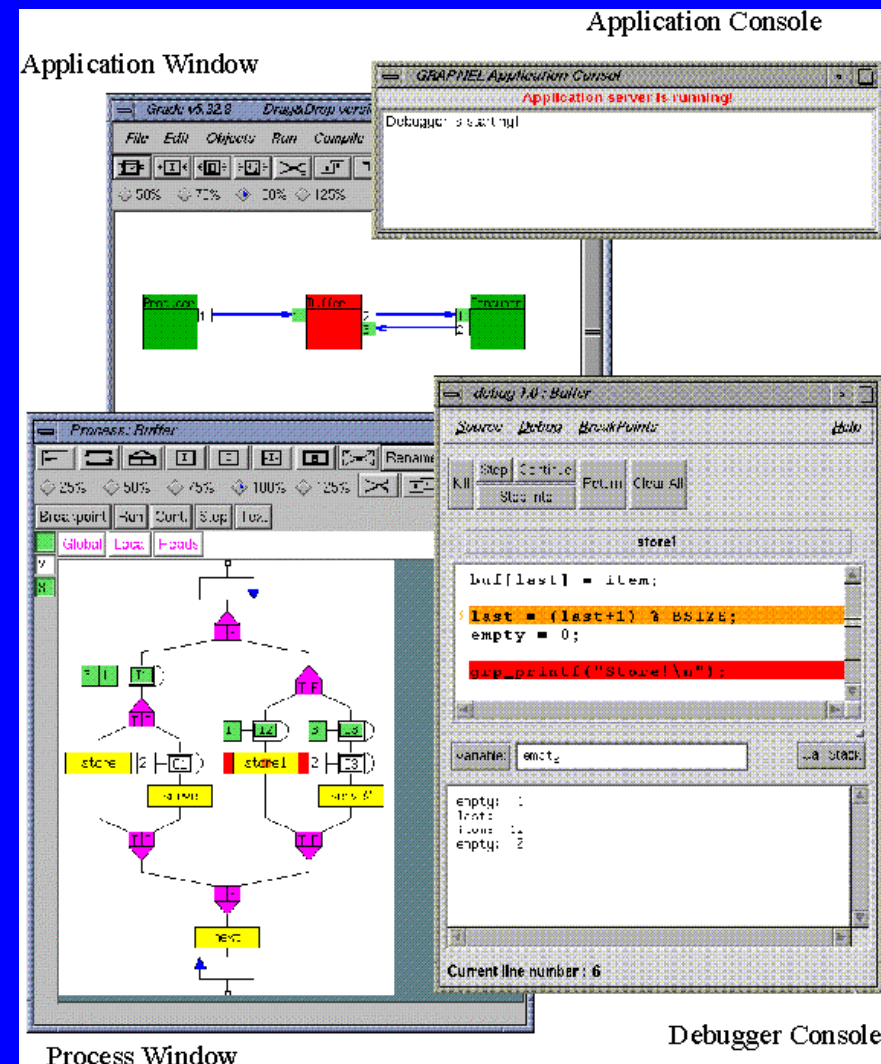
Text Editor

```
while (loop_count)
{
  trace_info (1, setr, "Global", 0);
  switch (setr->action)
  {
    case 1:
      if (find_points (setr))
      {
        setr = join_points (setr);
        trace_info (2, setr, "Global", 0);
      }
      else /* and 'setr' is complete graph found */
      {
        setr->order = unique_size (setr->order);
        trace_info (3, setr, "Global", 0);
        (setr->order) = NULL;
        loop_count = 0; /* loop levels end */
      }
      else
        setr = bucket (setr);
  }
}
```

# DDBG

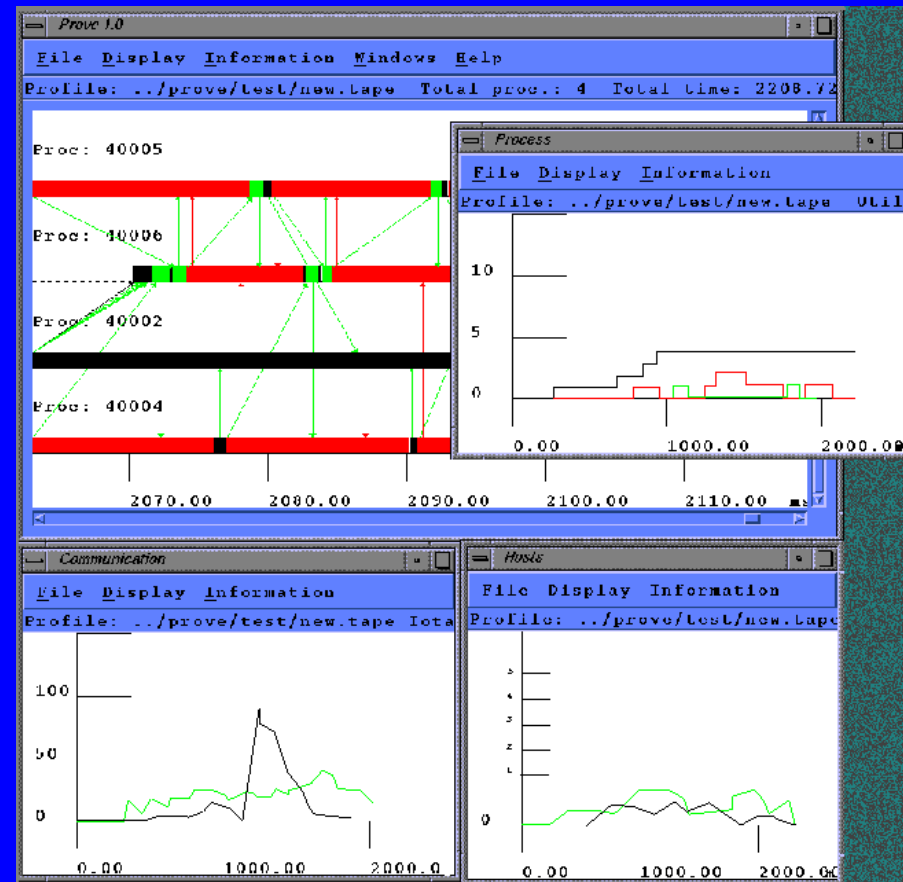
(Distributed DeBuGger)

- verteilter post-mortem Debugger
- ein Debugger pro Prozeß
- integriert in GRED



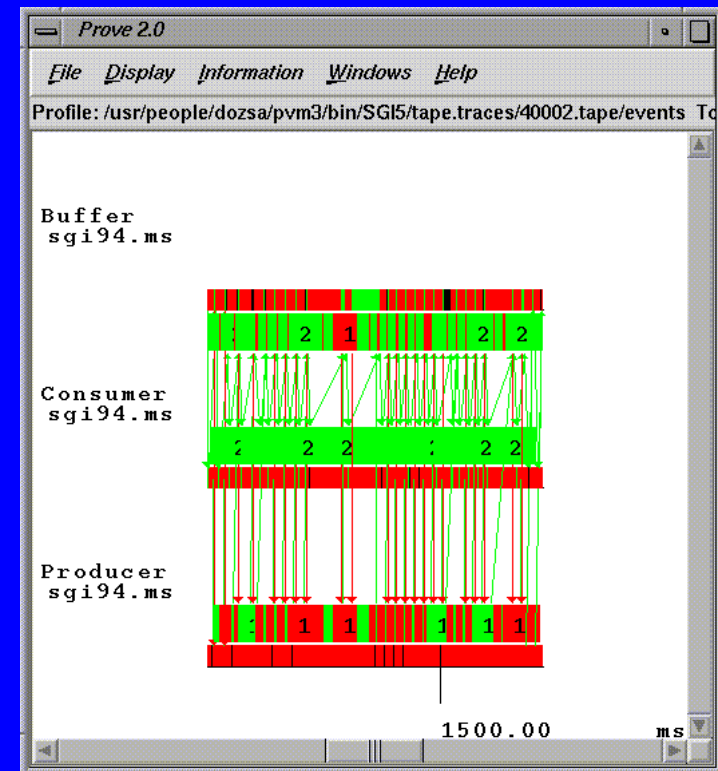
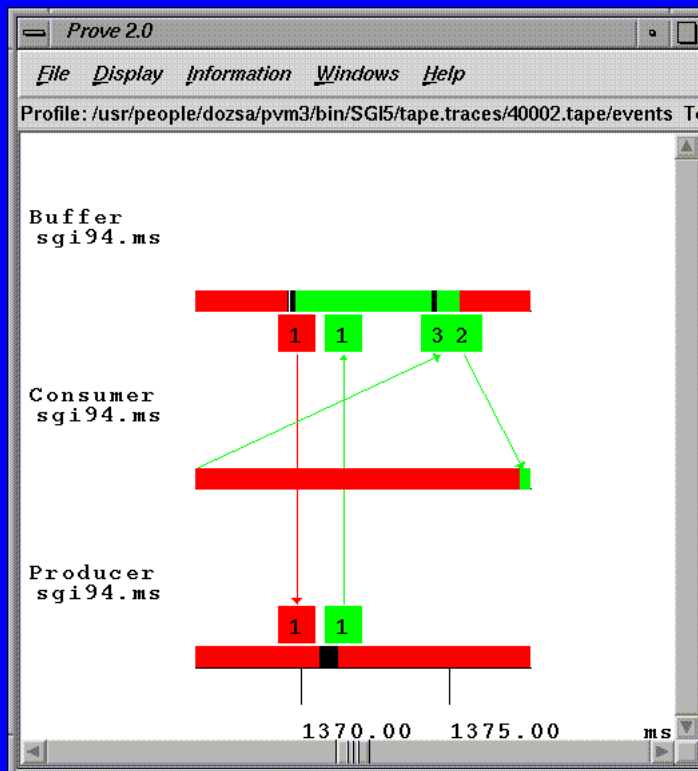
# PROVE

- Behavior Window:  
Prozesse und Events
- Processor Window:  
Auslastung der  
Prozessoren
- Communication Window:  
Datentransferrate  
zwischen Prozessen
- Hosts Window:  
Datentransferrate  
zwischen Prozessoren



# Behavior Window

- horizontale Streifen zeigen Prozesse in ihren verschiedenen Phasen
- Wait: roter Pfeil
- Send: grüner Pfeil



# Zusammenfassung

- Visualisierung hilft bei Entwicklung, Debugging und Verständnis von parallelen Programmen
- Integration von visuellen Elementen in Systeme minimiert Aufwand für Programmierer
- Quellen:
  - PARADE:  
<http://www.cc.gatech.edu/gvu/softviz/parviz/parviz.html>
  - GRADE:  
<http://www.lpds.sztaki.hu/projects/grade/grade.html>