

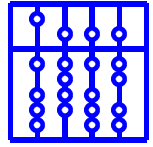
# Middleware-Konzepte für interoperable Entwicklungswerkzeuge

Thomas Ludwig

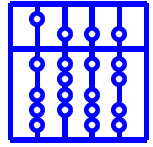
Institut für Informatik

Technische Universität München

[ludwig@in.tum.de](mailto:ludwig@in.tum.de)



- Entwicklungswerkzeuge
- Interoperable und universelle Werkzeuge
- Konventioneller Werkzeugentwurf
- Middleware-basierter Werkzeugentwurf
- Systemmodell und objektbasierte Schnittstelle
- Interoperable Werkzeuge
- Entwurfskonzepte der Middleware-Komponenten
- Ausblick



# Entwicklungswerkzeuge

---

## Einsatzbereiche

- Entwicklung, Optimierung und Laufzeitunterstützung paralleler und verteilter Programme

## Benutzungsmethodik

- Interaktiv — Automatisch

## Benutzungszeitpunkt

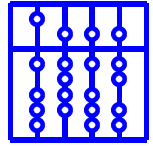
- Zur Laufzeit (online) — Nach Programmende (offline)

## Programminteraktion

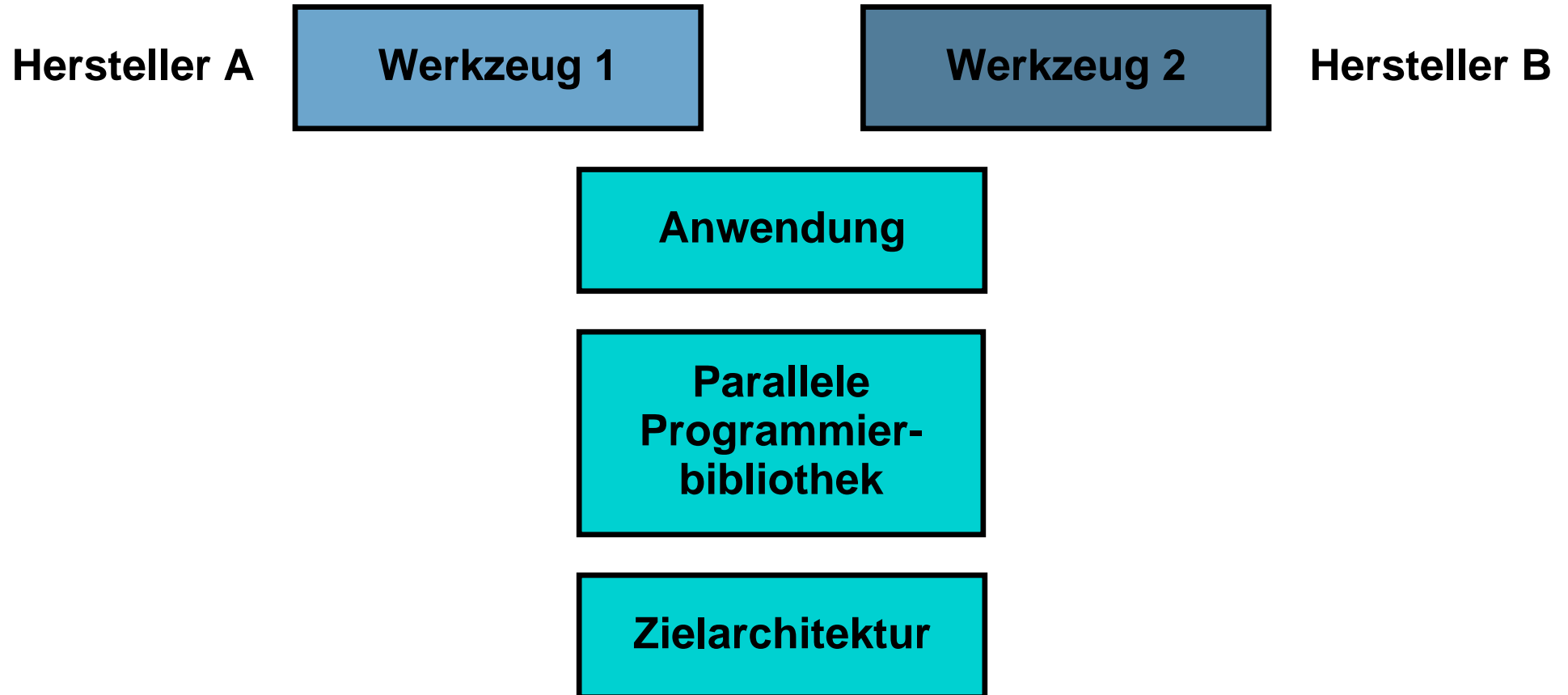
- Beobachtung — Manipulation

## Programmiermodell

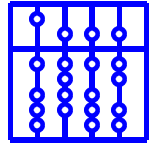
- Parallele und verteilte Programme mit Nachrichtenaustausch



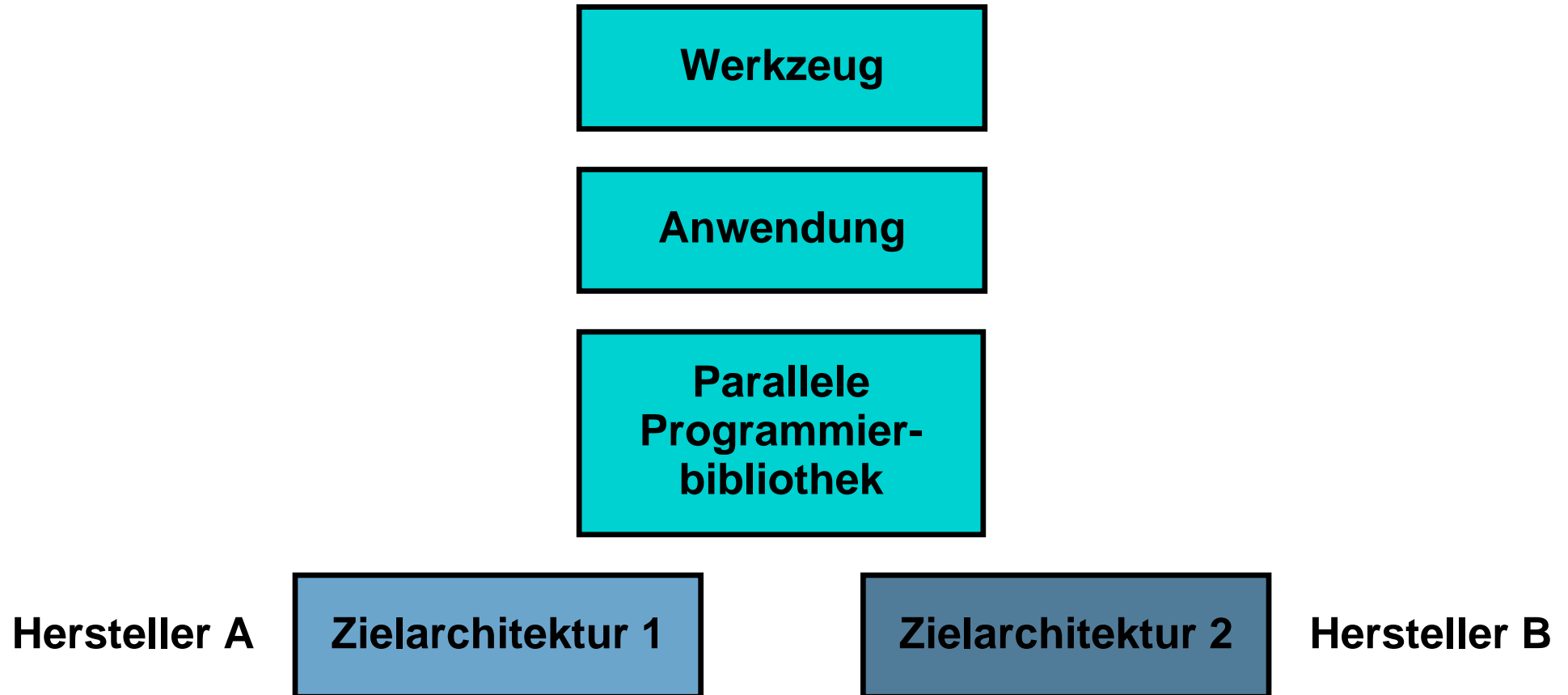
# Interoperable Werkzeuge



Verschiedene Werkzeuge arbeiten kooperierend auf demselben Programm

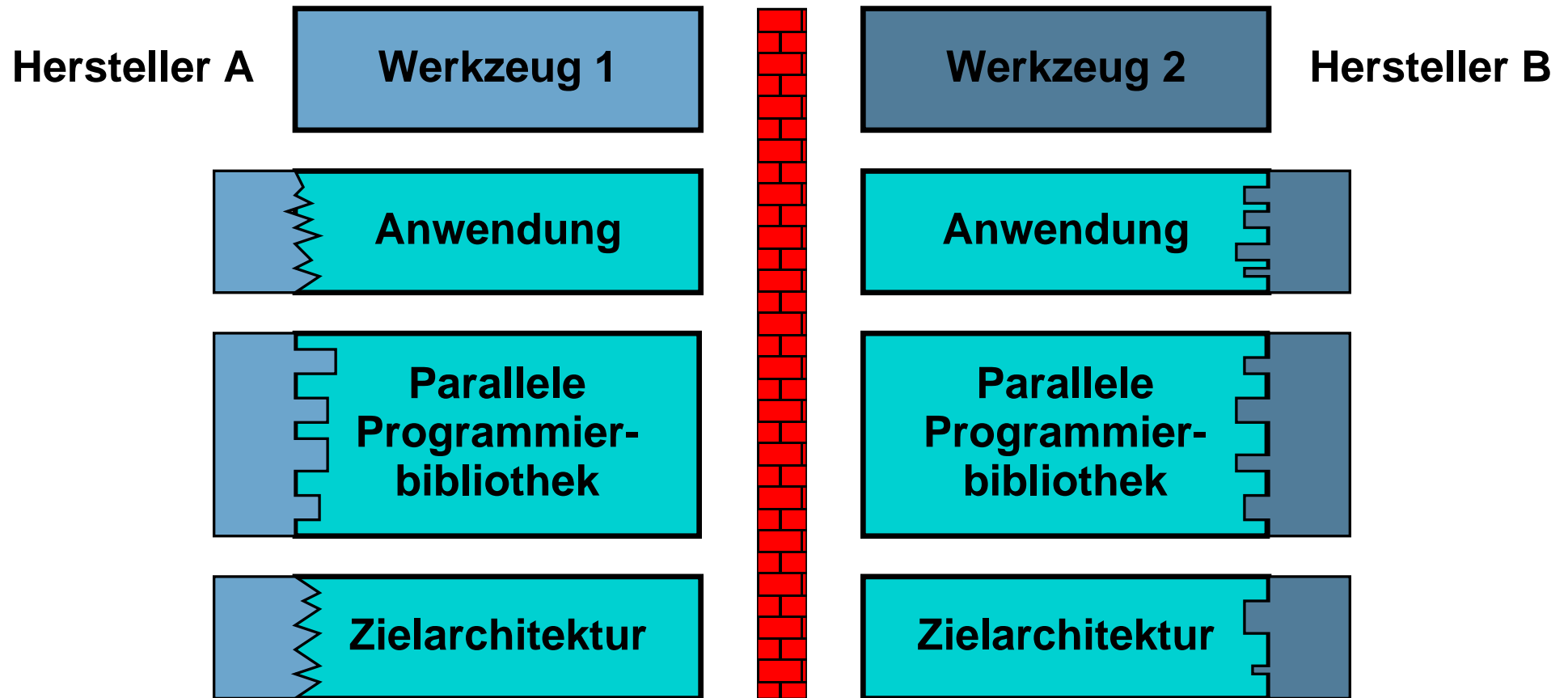


# Universelle Werkzeuge

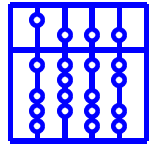


Dasselbe Werkzeug arbeitet auf verschiedenen Zielarchitekturen

# Konventioneller Werkzeugentwurf



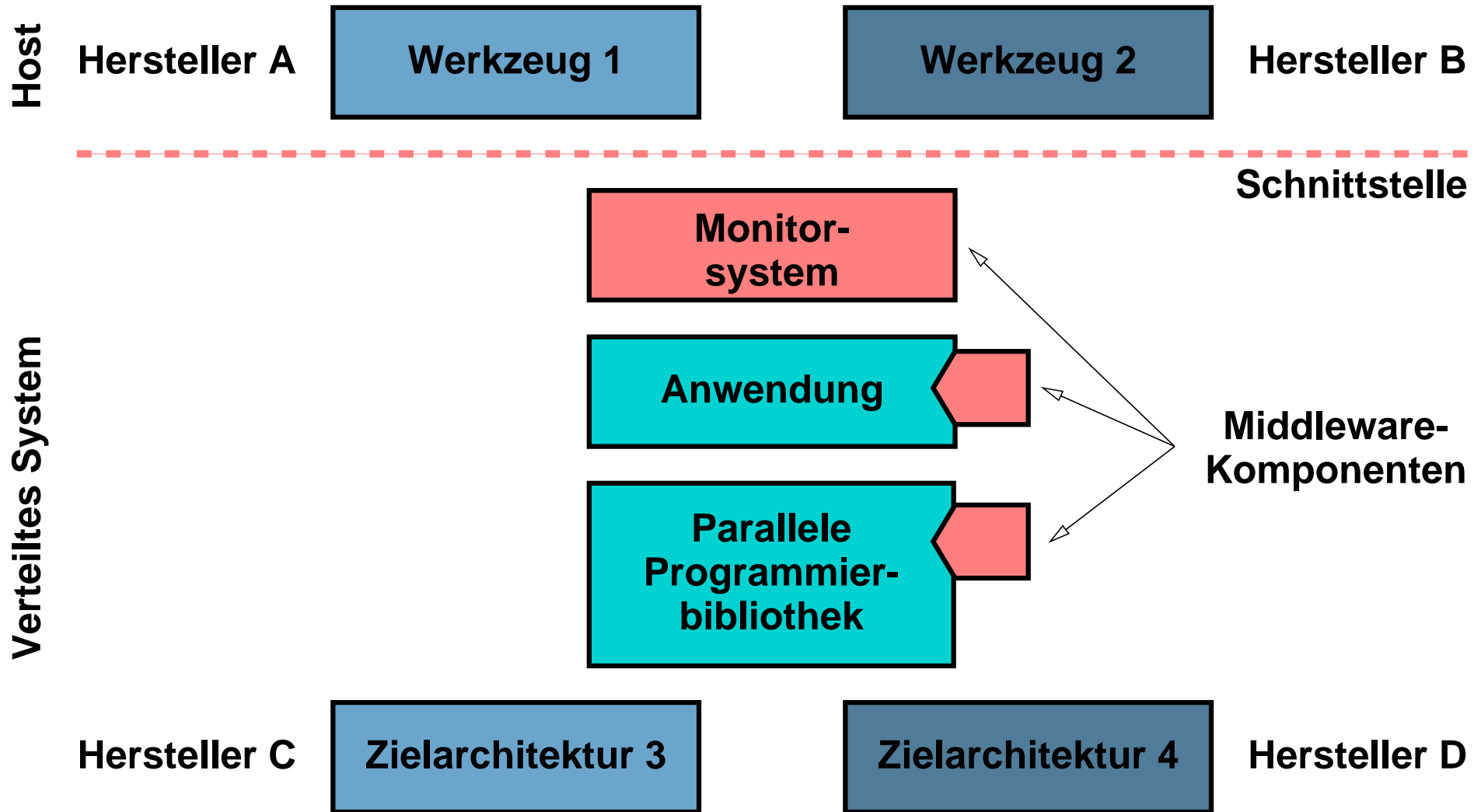
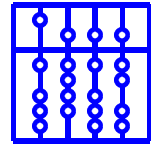
Werkzeuge sind in Implementierung und Anwendung miteinander unvereinbar

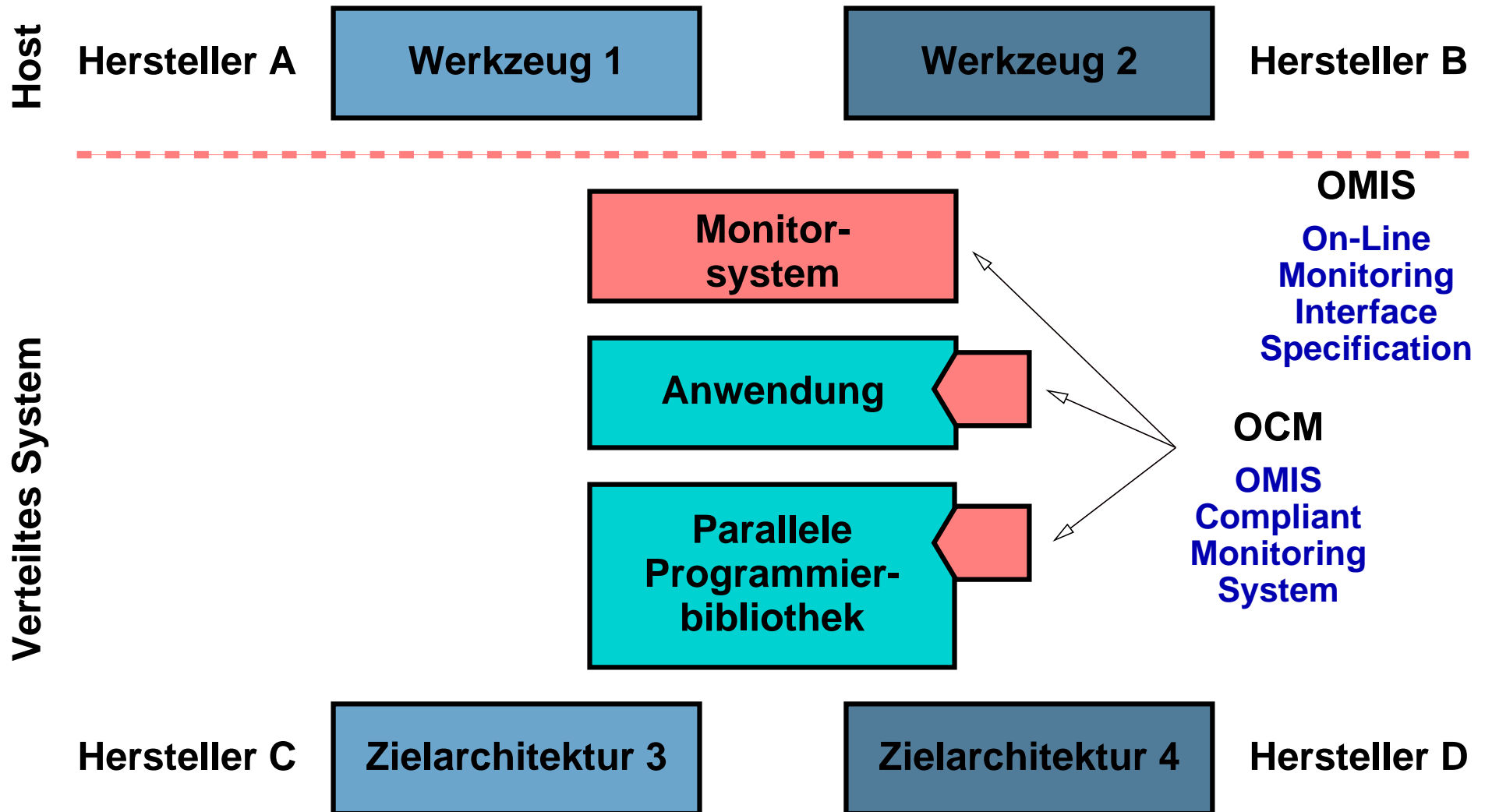
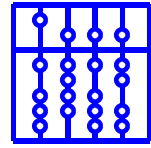


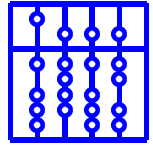
## Vorgehensweise

1. Identifikation von Infrastrukturteilen, die vielen Werkzeugen gemeinsam sind
2. Zusammenführen gemeinsamer Komponenten in einer einheitlichen Middleware-Architektur
3. Spezifikation einer genormten Schnittstelle zwischen Werkzeugen und Middleware
4. Anpassung neuer oder existierender Werkzeuge an diese Schnittstelle

⇒ Schnittstellen + Middleware-Komponenten







# Anforderungen an die Schnittstelle

---

## Unterstützung von Interoperabilität und Universalität

- Mehrere Werkzeuge nebenläufig anwendbar
- Werkzeuge auf mehreren Zielarchitekturen ablauffähig

## Allgemeingültigkeit und Flexibilität

- Verschiedene Werkzeuge zu unterschiedlichen Zwecken
- Verschiedene Werkzeugarchitekturen
- Verschiedene programmiersprachliche Objekte

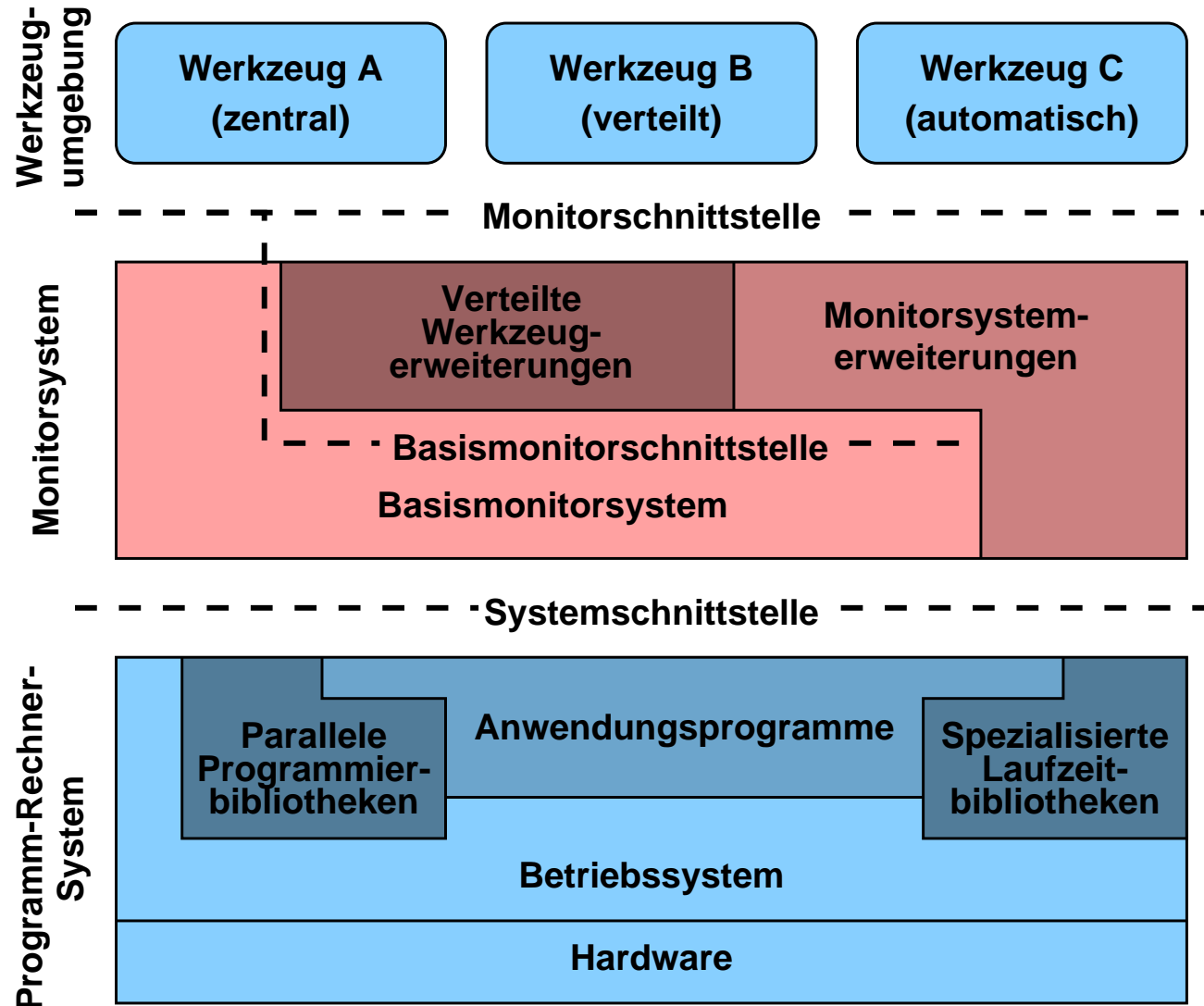
## Erweiterbarkeit

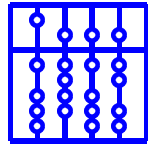
- Neue Werkzeugkonzepte
- Neue zu überwachende Objekte

## Effizienz

- Geringes Kommunikationsaufkommen zwischen Werkzeugen und Überwachungssystem

# Das Systemmodell





## Objektklassen

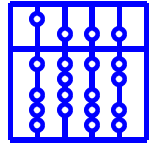
- System, Knoten
- Prozesse, Threads
- Nachrichtenpuffer, Nachrichten
- Monitorobjekte

## Dienstklassen:

- Informationsdienste (z.B. liefere Liste der Prozesse)
- Manipulationsdienste (z.B. halte Prozeß an)
- Benachrichtigungsdienste (z.B. melde, daß Prozeß terminiert)

## Sicht des Werkzeugs:

- Monitorsystem ist Server, der Dienste an Objekten anbietet

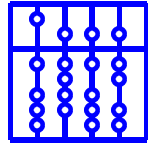


## Arbeitsprinzip: Ereignis/Aktions-Modell

- **Ereignis** = interessierender Zustandsübergang im überwachten Programm
- **Aktion** = erwünschte Beobachtung oder Manipulation des Programms

## Dienstanforderungen:

- Verhalten des Monitorsystems ist durch Ereignis/Aktions-Relationen bestimmt
- Programmierung der Relationen über die definierte Schnittstelle
- Aktionen jeweils ausgelöst, wenn das Ereignis erkannt wird
- Leere Ereignisdefinition  $\Rightarrow$  unbedingte Aktion



# Die Schnittstellenfunktion

## Syntax der Schnittstellenfunktion

Omis\_reply

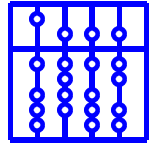
```
omis_request( char * request ,
              void (* callback)(Omis_reply reply, void *param),
              void *param,
              Omis_flags flags )
```

## Syntax der Ereignis/Aktions-Relationen

```
request ::= [ event_definition ] : action_list
event_definition ::= service_name ( parameters )
action_list ::= action | action [ ; ] action_list
action ::= service_name ( parameters )
```

## Semantische Details

- Dienste arbeiten auf Einzelobjekten und Objektmengen
- Dienste arbeiten auf Objekten beliebiger Hierarchiestufen
- Aktionen können potentiell nebenläufig ausgeführt werden



## Beispiel: Leistungsanalyse

---

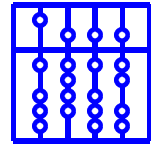
```
thread_has_started_lib_call([p_21], "MPI_Send") :  
  pt_integrator_start(pt_i_1) pt_counter_add(pt_c_1, $par5)
```

Wenn Prozeß p\_21 einen Sendeaufruf startet: aktiviere einen integrierenden Zähler und addiere die Nachrichtenlänge (\$par5) auf einen Zähler

```
thread_has_ended_lib_call([p_21], "MPI_Send") : pt_integrator_stop(pt_i_1)
```

Wenn der Prozeß den Aufruf beendet: halte den integrierenden Zähler an

**Resultat:** Verweilzeit im Sendeaufruf und gesendete Nachrichtenmenge



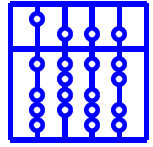
## Koppelung automatischer und interaktiver Werkzeuge

### Fehlersuche und Sicherungspunktgenerierung

- Abspeichern von Programm- und Sitzungszustand
- Wiederaufsetzen der Sitzung von Sicherungspunkten aus
- Aufnahme einer Fehlersuch Sitzung von Sicherungspunkten aus

### Leistungsanalyse und Lastausgleich

- Bewertung der Programmleistung unter Produktionsbedingungen
- Erfassung der Leistung des Lastverwaltungssystems
- Einblenden/Ausblenden des Wirkens der Lastverwaltung



## Koexistierende Werkzeuge

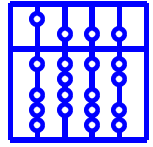
- Werkzeuge wissen nichts voneinander
- Beliebige Wechselwirkungen möglich

## Konsistente koexistierende Werkzeuge

- Werkzeuge können Aktionen anderer überwachen
- Manipulationen anderer Werkzeuge werden berücksichtigt

## Kooperierende Werkzeuge

- Werkzeuge können sich gegenseitig Nachrichten zusenden
- Beliebige Interaktionen sind realisierbar
- Mechanismus einfach; Semantik kritisch



## Koexistierende Werkzeuge

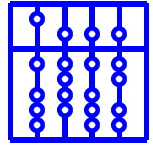
- Gemeinsame Middleware-Komponenten

## Konsistente koexistierende Werkzeuge

- Methoden zur Überwachung der Aktionen anderer Werkzeuge
- Methoden zum exklusiven Ausführen der eigenen Aktionen
- Informationsmodell der Werkzeuge (ähnlich Cache-Kohärenz)

## Kooperierende Werkzeuge

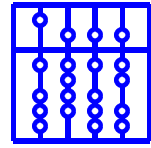
- Methoden zum Nachrichtenaustausch
- Dynamische Dienstverwaltung



## Implementierung für PVM-basierte Anwendungen auf Rechnernetzen

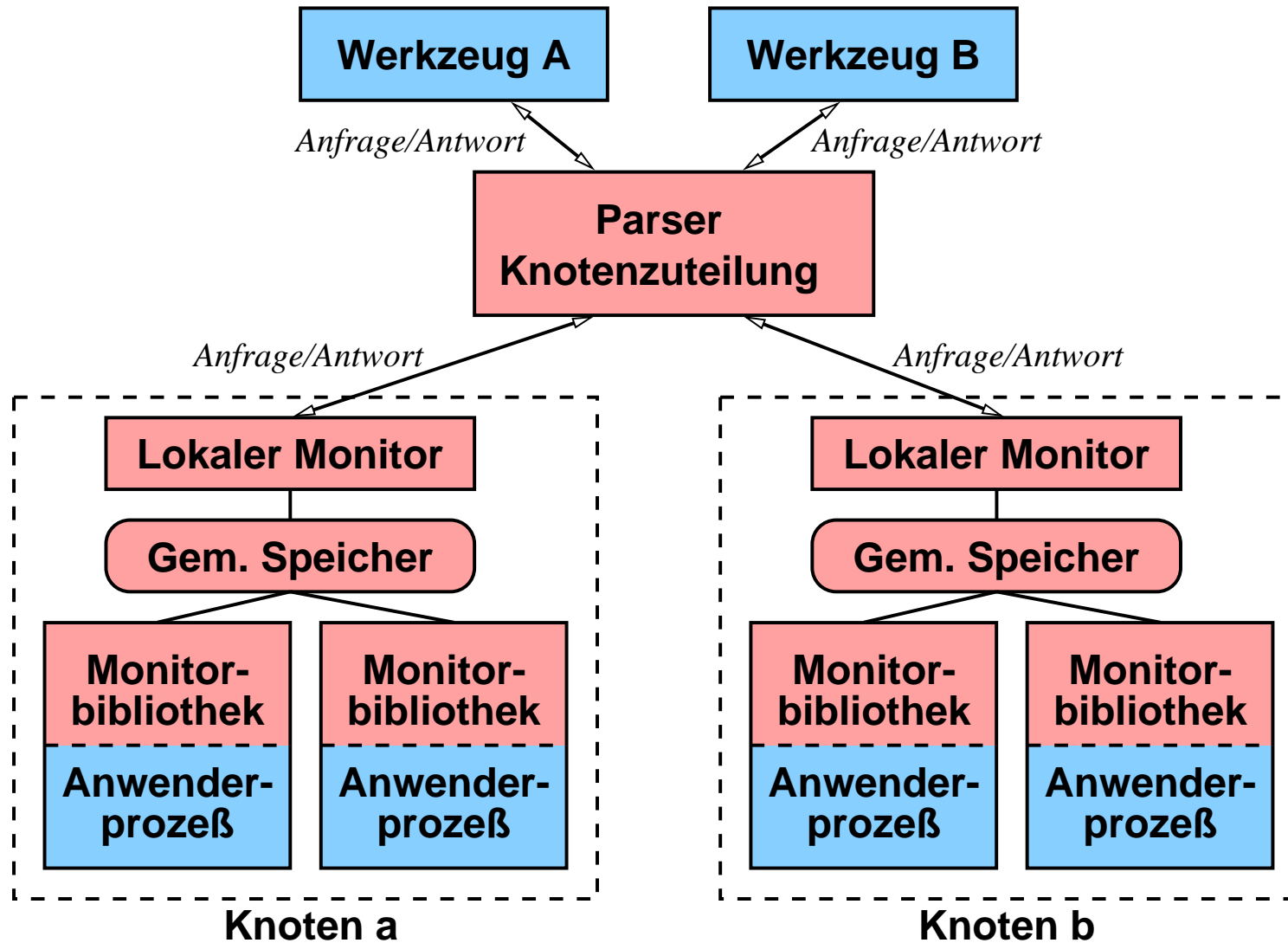
### Projektziele:

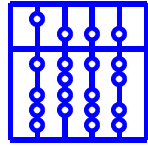
- Verifikation der Anwendbarkeit der Spezifikation
- Implementierungsplattform für THE TOOL-SET
- Implementierungsplattform für Werkzeuge Dritter
- Referenz für andere OMIS-konforme Implementierungen



- Das Monitorsystem wird aus kommunizierenden Monitoren gebildet, jeweils einem pro Rechnerknoten der virtuellen Maschine
- Globale Aktionsspezifikationen werden zur Definitionszeit an die Knoten verteilt.
- Kommunikation zwischen Monitoren mittels PVM-Mechanismen
- Im ersten Schritt: Zentrales Parsen der Dienstanforderungen
- Ereignisgetriebenes Ausführungsmodell der (Einzelknoten-)Monitore
- Ereigniserkennung und Aktionsausführung sowohl im Monitor als auch in der Anwendung
- Kommunikation zwischen knotenlokalen Monitorkomponenten über gemeinsamen Speicher und Signale

# Grobstruktur des Monitorsystems





## OMIS

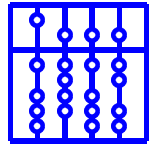
- Version 1.0 veröffentlicht Februar 1996
- Überarbeitung: Version 2.0 veröffentlicht Juli 1997

## OCM

- Implementierung begonnen Januar 1997
- Erster funktionsfähiger Prototyp: Juli 1997
- Erste interoperable Werkzeuge des TOOL-SET: Oktober 1997

## Kooperationen

- KFKI-MSZKI Research Institute, Budapest, Ungarn (GRADE)
- Institute of Computer Science und Academic Computer Center CYFRONET, Krakau, Polen
- University of Westminster, London, England
- Univeristat Autònoma de Barcelona, Barcelona, Spanien



- Abschluß der OCM-Implementierung und Werkzeuganpassung
- Ausbau der Konzepte der Interoperabilität der Werkzeuge
- Integration neuer Werkzeuge: Sicherungspunkterstellung, Lastausgleich, deterministische Programmausführung
- OMIS für Architekturen mit verteiltem gemeinsamen Speicher
- Kooperationen mit anderen Forschungsgruppen

<http://www.bode.informatik.tu-muenchen.de/~omis>