Betriebssystemtechnik

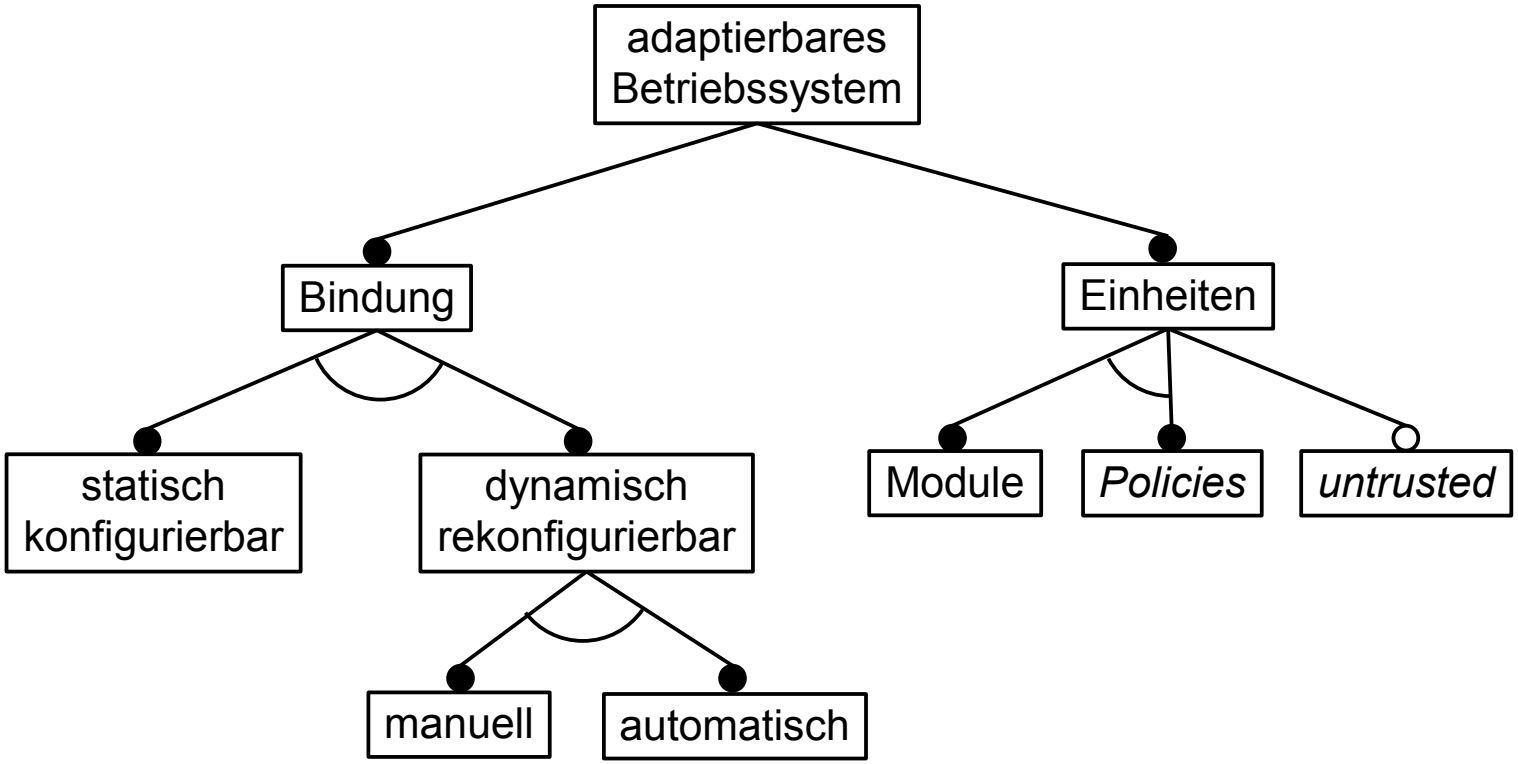
Operating System Engineering (OSE)

Anpassbare Betriebssysteme in der Forschung

Motivation

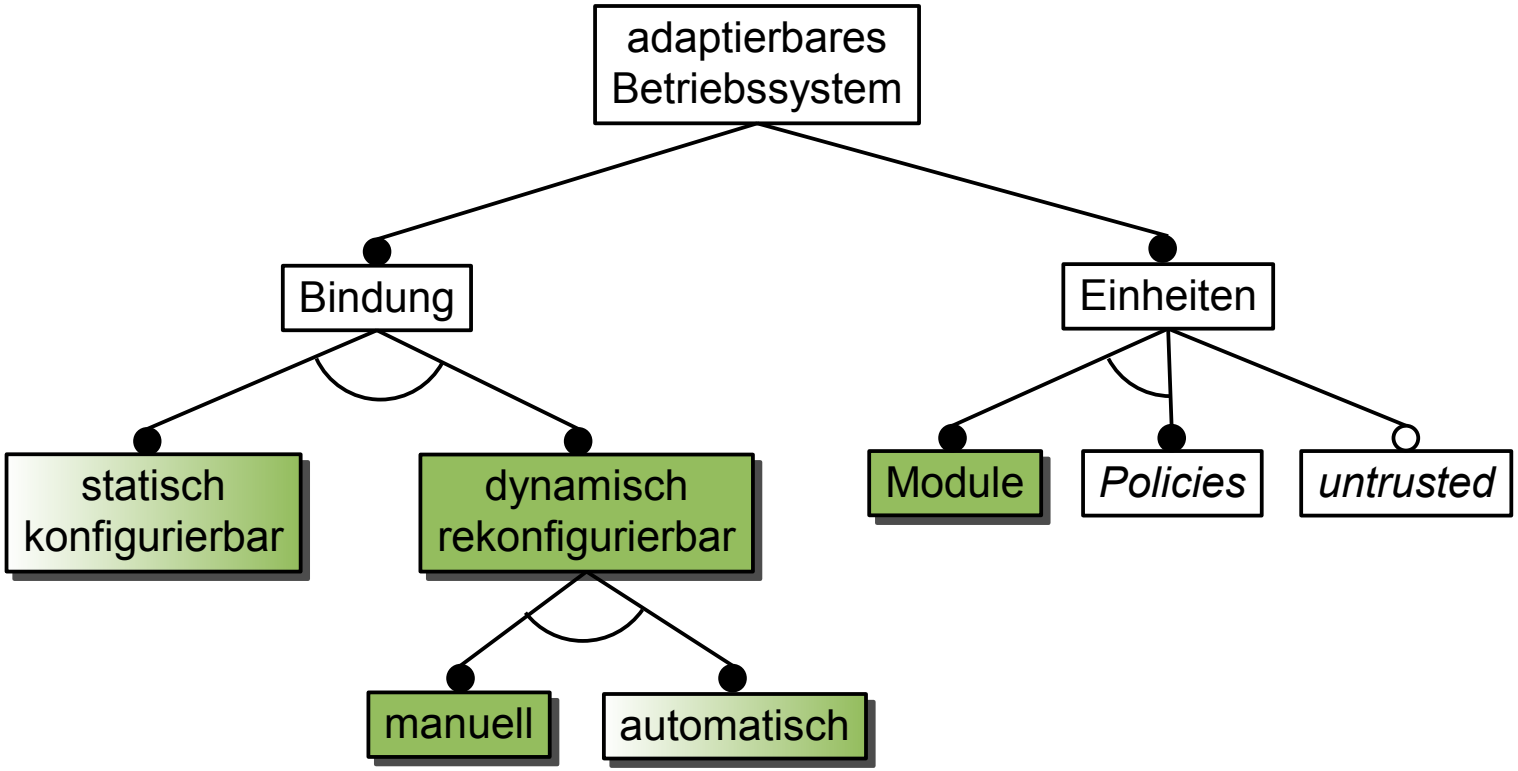
- Bisher betrachtete Konzepte stammen aus der Welt der Softwaretechnik ...
  - Produktlinienentwicklungsprozess
  - Merkmalmodelle
- ... und aus der Welt der Programmiersprachen.
  - Präprozessoren (z.B. *Frame* Prozessoren)
  - Aspektorientierte Programmierung
  - Objektorientierte Programmierung
  - Generative Programmierung
- Welche Lösungen sind in der Betriebssystem-forschungsgemeinde eingesetzt bzw. entwickelt worden?

Variantenvielfalt



Ausgangspunkt für Forschung (1)

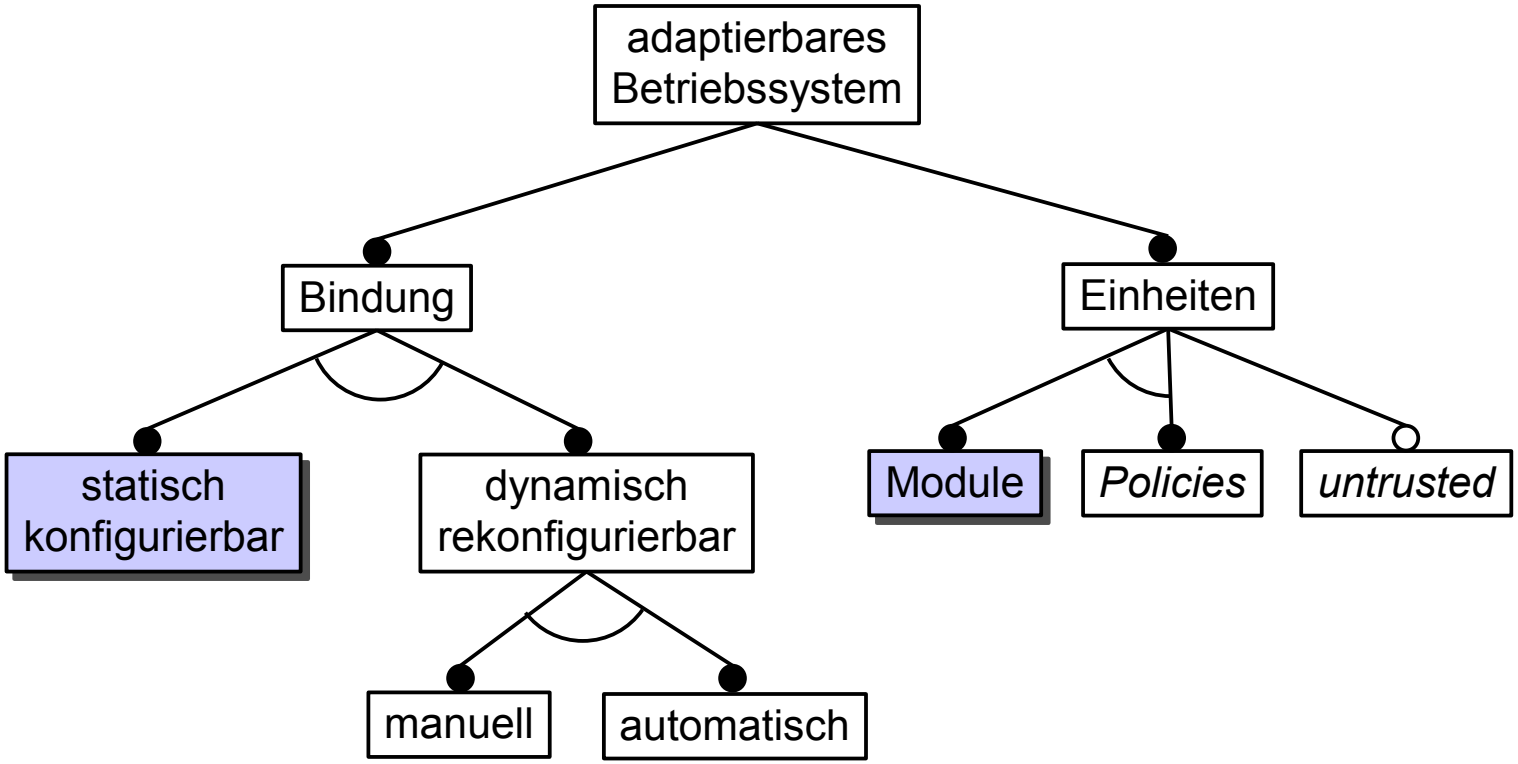
- ... ist der „Stand der Kunst“:



- Großrechner, *Workstation* und PC Betriebssysteme:
  - dynamisches Laden und Entladen von Modulen wie Treibern oder Dateisystemen, teils automatisch, teils auch statisch konfiguriert

Ausgangspunkt für Forschung (2)

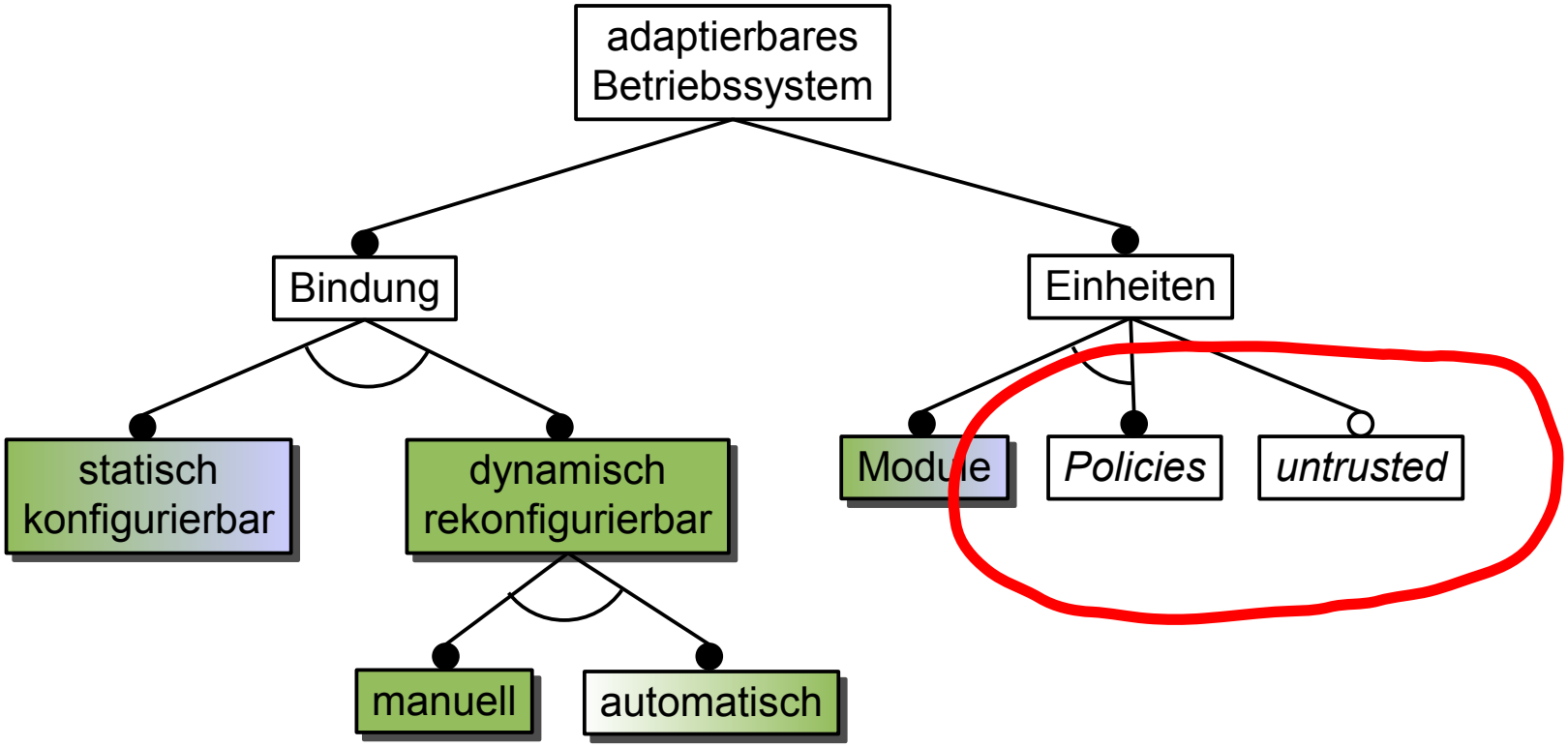
- ... ist der „Stand der Kunst“:



- Eingebettete- und Echtzeit-Betriebssysteme:
  - statische Konfigurierung auf Modulebene und gut modularisierte Strategien (z.B. *Scheduling*), i.d.R. keine Dynamik

Im Zentrum der BS-Forschung [1]

- ... ist, was neu und herausfordernd ist:



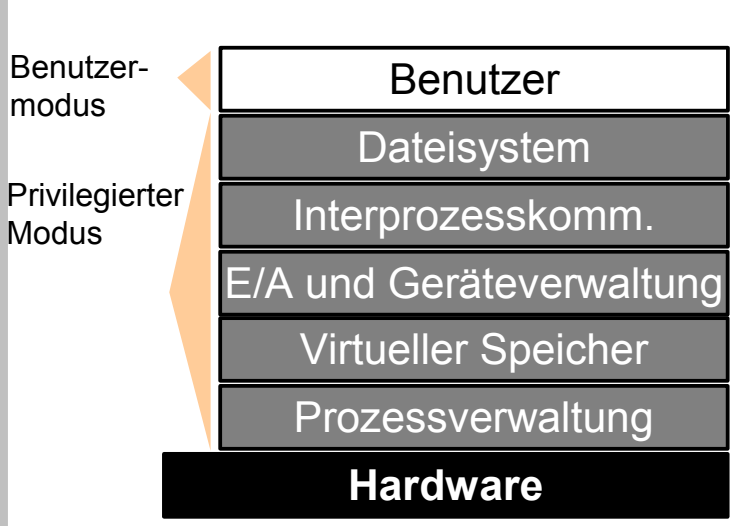
- Ausführung von nicht vertrauenswürdigem Code
- Adaptierbarkeit von *Policies* (Systemstrategien)
  - vor allem dynamisch und im Hinblick auf *Performance*

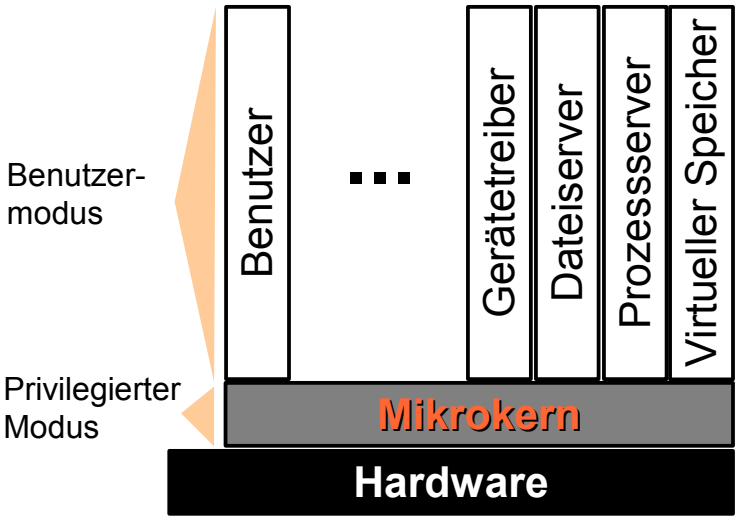
Meilenstein 1: Erweiterbare Kerne

- Problem: nicht vertrauenswürdige Module können einen Systemkern zum Absturz bringen oder lahm legen
  - unkontrollierte Speicherzugriffe
  - exzessiver Ressourcenverbrauch
- Lösungsansatz: Schutz
  - durch Isolation
    - *„Sandboxing“*
  - durch sichere Sprachen
    - Modula 3, wie z.B. in SPIN
    - Java, wie z.B. in JX
- Lösungsansatz: Überprüfung
  - der Quelle (z.B. Systemadministrator, BS-Hersteller)
  - des Verhaltens (*„Proof Carrying Code“*)

Meilenstein 2: Mikrokerne (1)

- Idee: Verkleinerung der *„Trusted Computing Base“* und Schutz durch Isolation

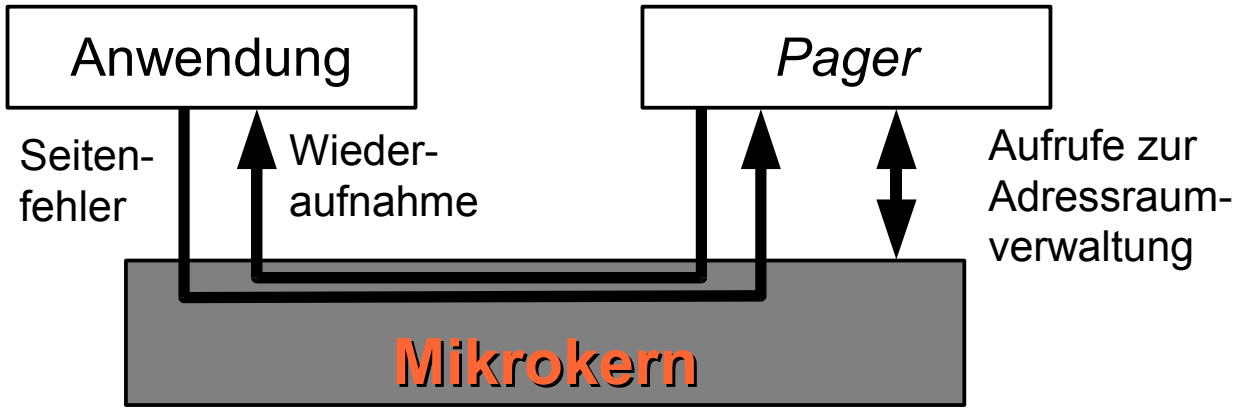




- die Aufgabe des Mikrokerns beschränkt sich im Wesentlichen auf die Realisierung des Nachrichtenaustauschs
- das Konzept sorgt für vergleichsweise einfache (dynamische) Rekonfigurierbarkeit

Meilenstein 2: Mikrokerne (2)

- Beispiel: der externe *Pager* von Mach



- Nutzen: anwendungsspezifische *Paging* Strategien können die *Performance* erheblich steigern
- Problem: viele Kontextwechsel wirken negativ auf die *Performance*

Zwischenfazit

- die klassische Betriebssystemforschung befasste sich hinsichtlich Adaption viele Jahre lang stark mit der dynamischen Erweiterung und Rekonfigurierung unter dem Blickwinkel *Performance* und Sicherheit
  - keine eingebetteten Systeme und statische Konfigurierung
- heute hat sich die Situation verändert
  - Zahlen von David Tennenhouse
  - Rob Pikes *„System Software Research is Irrelevant“*
  - ... und vor allem neue Anwendungsbereiche
    - *Ubiquituous*- und *Pervasive Computing*
    - *Sensor Networks*
- Themen im Bereich der Spezialzwecksysteme und statische Konfigurierung werden inzwischen ernst(er) genommen

Konfigurierbare Forschungssysteme

... einige (bedeutende) Systeme als Beispiel:

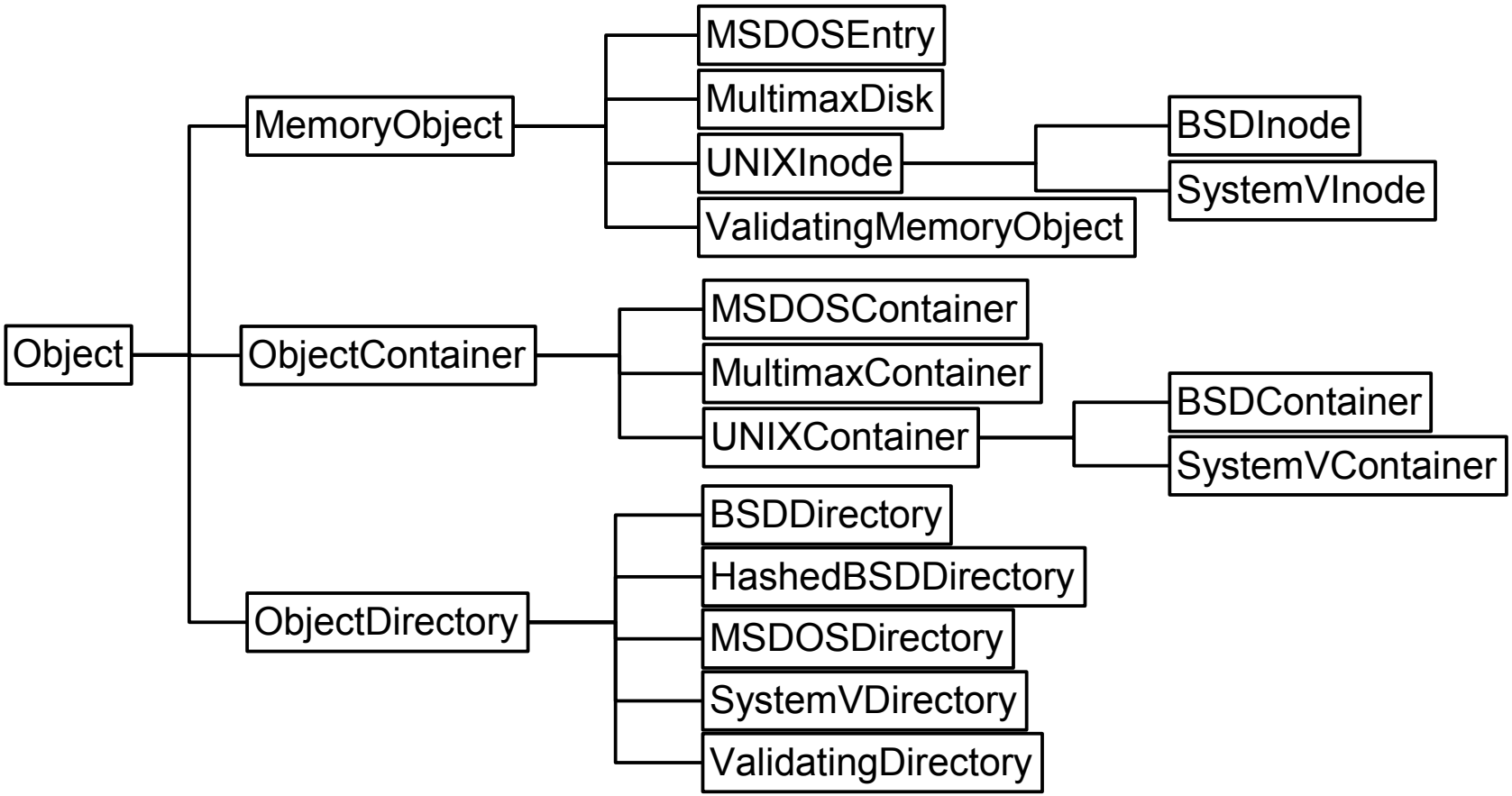
- Choices, 1987
  - eines der ersten objektorientierten Betriebssysteme
- OSKit, 1997
  - Bibliothekssystem, nur schnell dank *„Knit“*
- TinyOS, 2000
  - Minimales BS für Sensornetzwerke, NesC Programmiersprache
- PURE, 1995-2003
  - Merkmalmodell, Aspekte, Minimierung der Größe
- EPOS, 1999
  - Metaprogrammierung, automatische Anwendungsanalyse

Choices [2, 3]

- R. Campbell, 1987
  University of Illinois at Urbana-Champaign
- in C++ mit einigen Erweiterungen implementiert
  - Garbage Collection
  - Klassenobjekte und Laufzeittypprüfungen
  - dynamisches Nachladen von Subklassen
- setzt konsequent den *Framework* Gedanken um
  - anwendungsspezifische Erweiterung und Spezialisierung vorhandener *Framework* Klassenhierarchien
  - Instanziierung der gewünschten Klassen und Verknüpfung der passenden Objekte
  - Hierarchien und einheitliche (abstrakte) Schnittstellen

Choices – ein Beispielsubsystem

- die *Choices* Dateisystem Hierarchie:

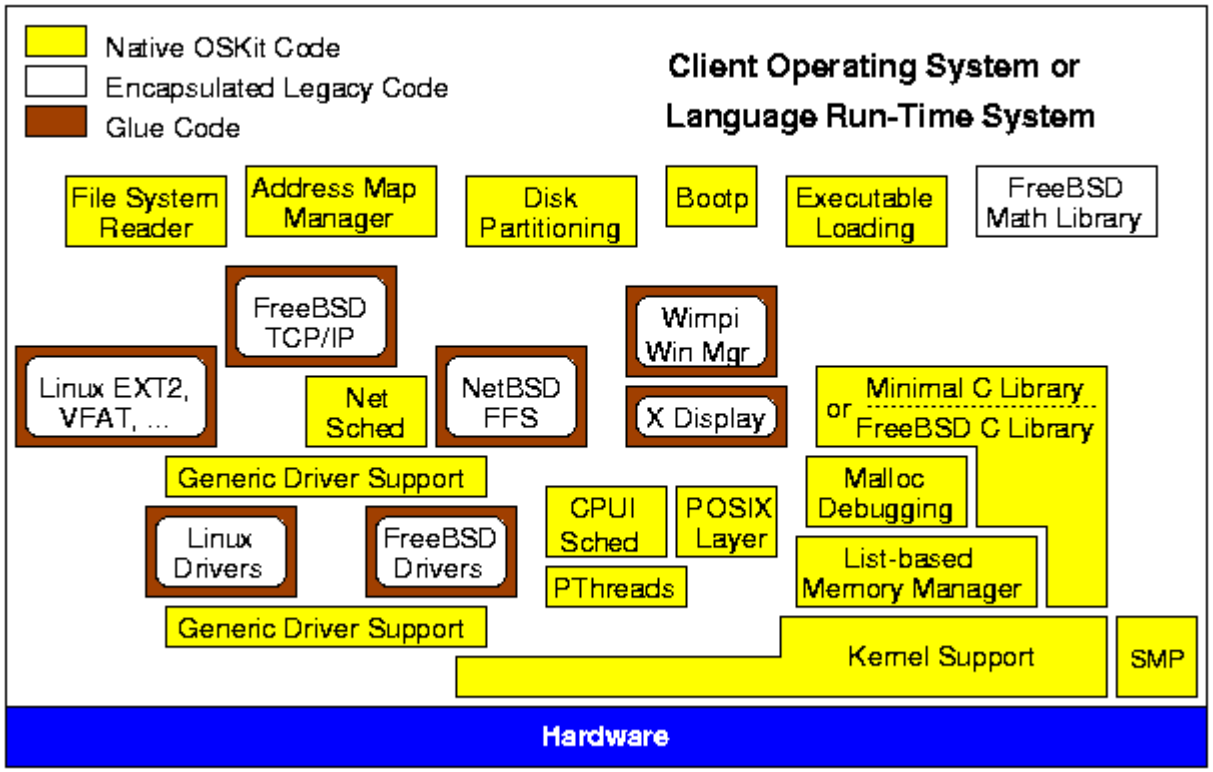


Choices - Fazit

- Choices ist ein als objektorientiertes Framework aufgebautes System
  - erweiterbar
  - anwendungsspezifisch spezialisierbar
- die Entwickler haben den Overhead durch dynamisches Binden in Kauf genommen
  - Laufzeiten (vor allem Kommunikation) werden als gut angesehen
  - Größen werden nicht publiziert
- Einsatzgebiet sind hauptsächlich parallele Systeme, z.B.
  - Intel Hypercube
  - NS 32332
- Warum gibt es heute nicht mehr objektorientierte Betriebssysteme?
  - der Ressourcenverbrauch spielt bei der Auswahl der Methoden im BS-Bereich eine enorme Rolle

OSKit [4]

- *University of Utah*
- Ziel: Schaffung einer Basisplattform für BS-Forschung
  - eine Sammlung von Bibliotheken

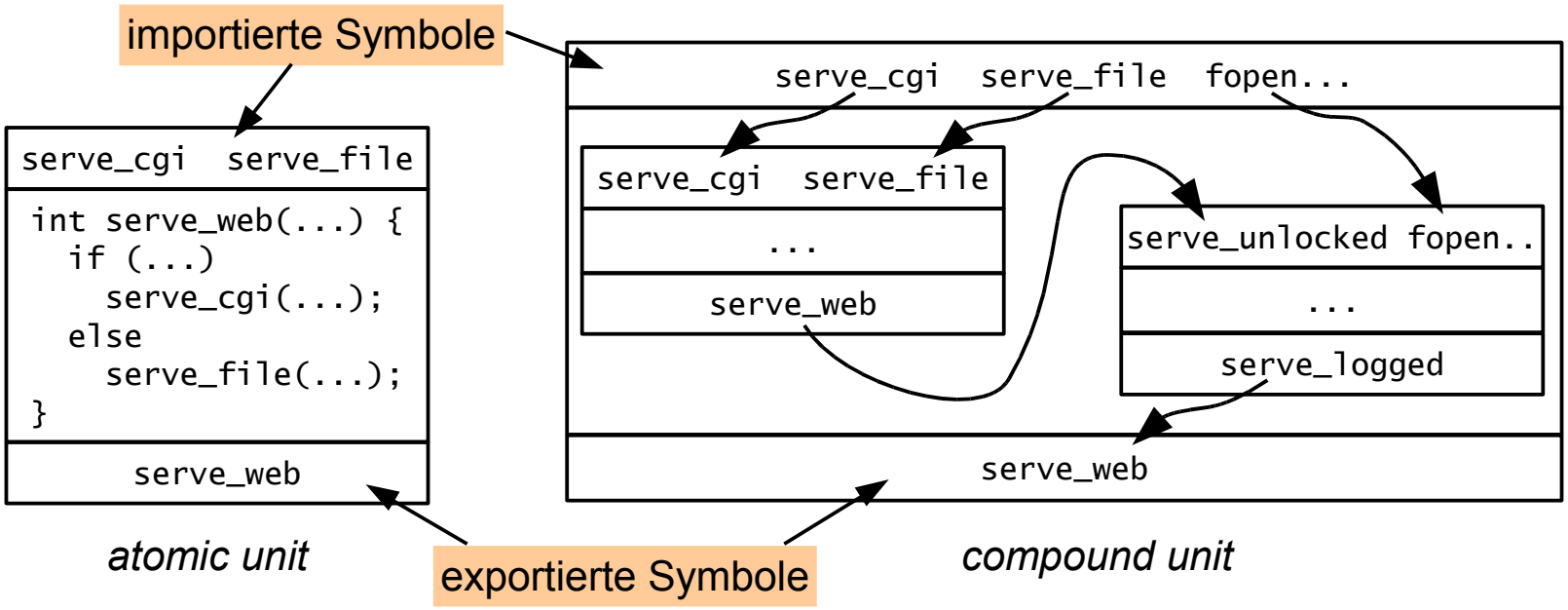


OSKit – Trennbare Module

- alle OSKit Module sollen auch einzeln verwendbar sein
  - Aufrufe über Modulgrenzen werden daher mit einer Indirektionsstufe versehen
- überschreibbare Funktionen (1. Ansatz)
  - Speicheranforderungen in Gerätetreibern erfolgen immer mit fdev_mem_alloc. Der OSKit Speicher-Manager stellt diese Funktion bereit. Der Benutzercode kann sie aber auch selbst bereitstellen.
- dynamisches Binden (2. Ansatz)
  - eine COM Schnittstelle wird beispielsweise beim Zugriff auf Blockgeräte verwendet
  - Dateisysteme erwarten ein kompatibles Blockgerät als Parameter, das auch vom Benutzercode stammen kann.
- die Indirektion macht die Komponentenübergänge teuer
  - Komponenten werden eher grobgranular ausgelegt

OSKit – Knit (1) [5]

- *Knit* ergänzt C um ein Komponentenmodell und verknüpft Komponenten mittels Codetransformation oder Objektcode-Manipulation



OSKit – Knit (2)

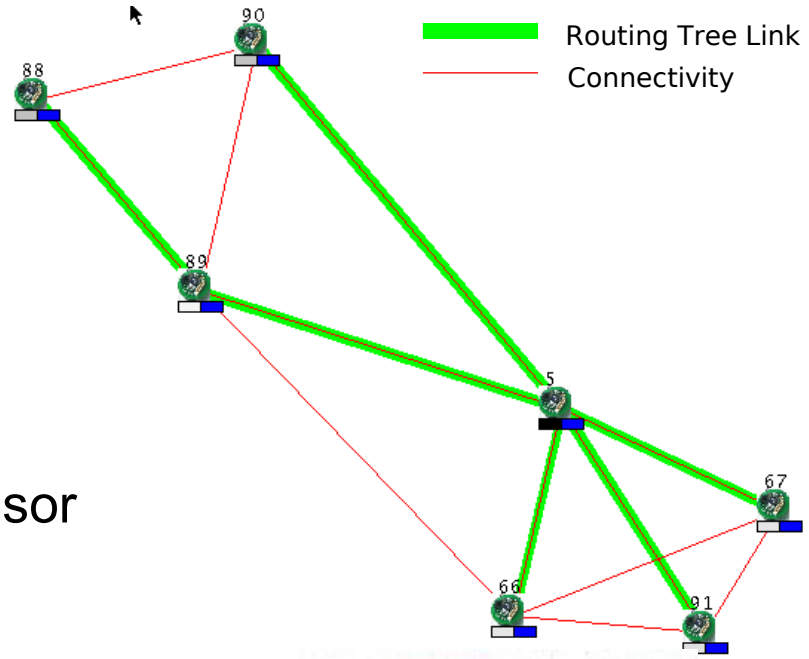
- weitere Aufgaben von *Knit*
  - Generierung von Modul-Initialisierungscode unter Berücksichtigung von Reihenfolgeabhängigkeiten
  - Verarbeitung beliebiger *Properties* und *Constraints*
  - Umbenennungen
  - Verschmelzen von C Implementierungsdateien zwecks *Inlining*
- der Gewinn durch den Einsatz von *Knit* ist beachtlich: (eine modulare *Router* Implementierung [Pentium Pro])
  - *cycles*: 2411 -> 1574
  - *fetch stall cycles*: 781 -> 455
  - *text size*: 109464 -> 106065

OSKit – Fazit

- OSKIT ist eine Sammlung von Komponenten
  - ein Konzept, das von C und typischen Bindern nicht unterstützt wird
- ein besonderes Problem sind Importe
  - Indirektion durch den Binder
  - Indirektion durch Funktionszeiger (z.B. COM Schnittstelle)
  - **Indirektionen sind teuer!**
- Knit schafft Abhilfe
  - C wird um ein Komponentenmodell und eine domänenspezifische Sprache (DSL) zur Komponentenbeschreibung ergänzt
  - den entscheidenden *Performance*-Vorteil konnte *Knit* durch das Vermelzen von Implementierungsdateien und *Inlining* erreichen
- ressourcensparende Systeme sollten daher ...
  - keine (binären) Komponentenmodelle verwenden
  - mit *Inlining* von Funktionen arbeiten

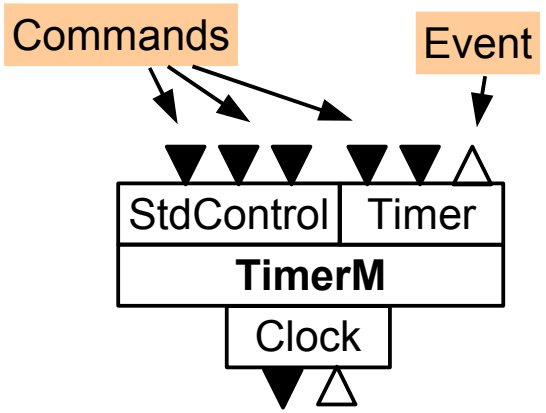
TinyOS [6]

- D. Culler, 2000
  University of California, Berkeley
- Zieldomäne: drahtlose Sensor-Netzwerke
- eine typische Hardware:
  - 4Mhz, 8bit MCU (ATMEL)
    - 512 bytes RAM, 8K ROM
  - 900Mhz Radio (RF Monolithics)
    - 10-100 ft. range
  - Temperature Sensor & Light Sensor
  - LED outputs
  - Serial Port
- trotz Größenminimierung setzt TinyOS auf Komponenten!



TinyOS – Komponentenmodell (1)

- TinyOS Komponenten haben wie *Knit* Komponenten Importe und Exporte.

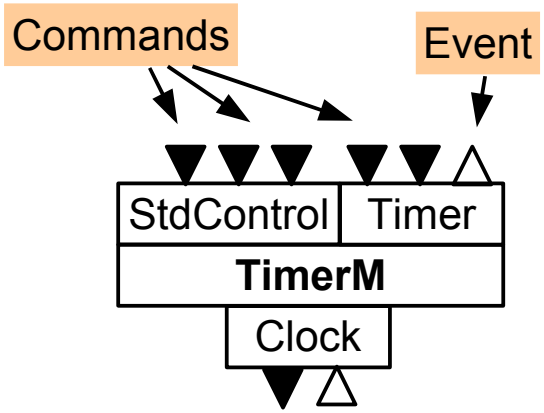



```
module TimerM {
  provides {
    interface StdControl;
    interface Timer[uint8_t id];
  }
  uses interface Clock;
}
implementation {
  ... im C Dialekt NesC
}
```

- *Commands* sind synchrone Funktionsaufrufe
- *Events* signalisieren asynchron das Auftreten eines Ereignisses
- Komponenten implementieren einen Zustandsautomaten

TinyOS – Komponentenmodell (1)

- TinyOS Komponente
  Importe und Exporte

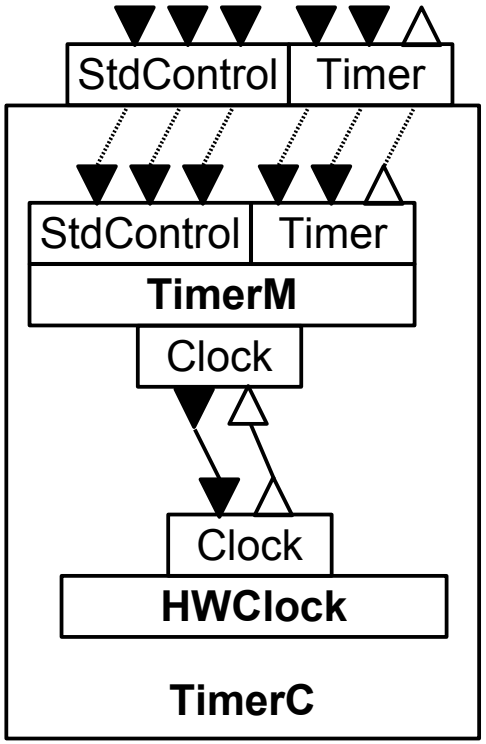



```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}

interface Timer {
  command result_t start(char type,
                         uint32_t interval);
  command result_t stop();
  event result_t fired();
}

interface Clock {
  command result_t setRate(char interval,
                           char scale);
  event result_t fire();
}
```

- *Commands* sind syn
- *Events* signalisieren
  Ereignisses
- Komponenten implementieren einen
  Zustandsautomaten

TinyOS – Komponentenmodell (2)

- TinyOS Komponenten können auch hierarchisch verbunden werden




```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}
implementation {
  components TimerM, HWClock;
  StdControl = TimerM.StdControl;
  Timer      = TimerM.Timer;
  TimerM.Clock -> HWClock.Clock;
}
```

TinyOS – NesC [7]

- das Besondere ist die Implementierung in NesC
  - unterstützt direkt das Komponentenmodell von TinyOS
  - kein dynamischer Speicher
  - keine Aufrufe über Funktionszeiger
  - alle Komponenten werden bei der Übersetzung gemeinsam betrachtet
- Ergebnis: *Whole Program Analysis and Optimization*
  - Eliminierung von unbenutzem Code
  - Optimierung des *Inlinings*
  - automatische Erkennung potentieller *Race Conditions*

App	**Code size**		**Code reduction**	**Data size**	**CPU reduction**
	inlined	*non-inlined*			
surge	14794	16984	12%	1188	15%
Mate	25040	27458	9%	1710	34%
TinyDB	64910	71724	10%	2894	30%

TinyOS – Fazit

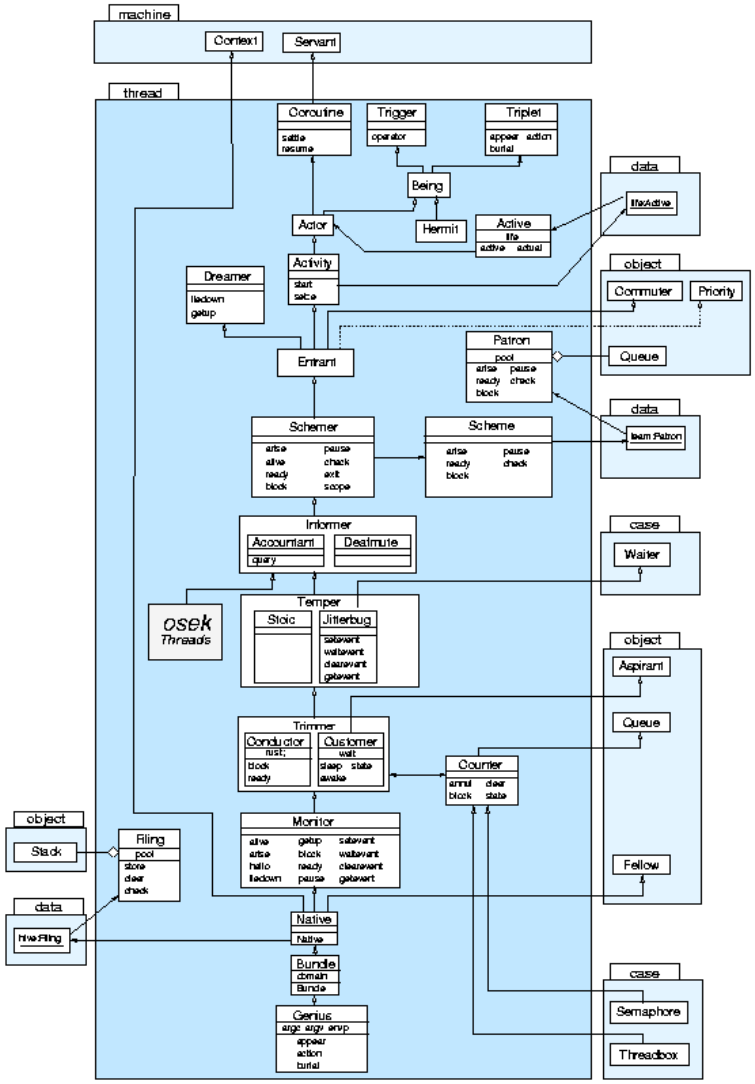
- TinyOS geht mit der Optimierung noch weiter als OSKit mit *Knit*
  - *Whole Program Optimization*
- damit dies möglich ist, müssen die Komponenten aber in der Programmiersprache NesC implementiert sein

PURE

- W. Schröder-Preikschat, 1995-2003
  Universität Potsdam und Magdeburg
- Zieldomäne: kleinste eingebettete Systeme
  - hochgradige Konfigurierbarkeit
- Ansatz: familienbasierter Entwurf
  - Parnas u. Habermann, Vererbung (C++)
- Umfang: ca. 900 Dateien, 300 Klassen
- Herausforderungen:
  - Beherrschung der Konfigurationsvielfalt
  - Umsetzung im Programmcode

PURE – die *Thread*-Hierarchie

- ... extrem feingranular aufgebaut: 13 Ebenen!
- unterstützt z.B. diverse *Scheduling*-Strategien
- Klassenaliase konfigurieren die Schichten
  - anfangs über Präprozessor-Konfigurationsflags
  - durch leere (aber kompatible) Klassen wie Deafmute oder Stoic können Schichten auch inaktiv sein



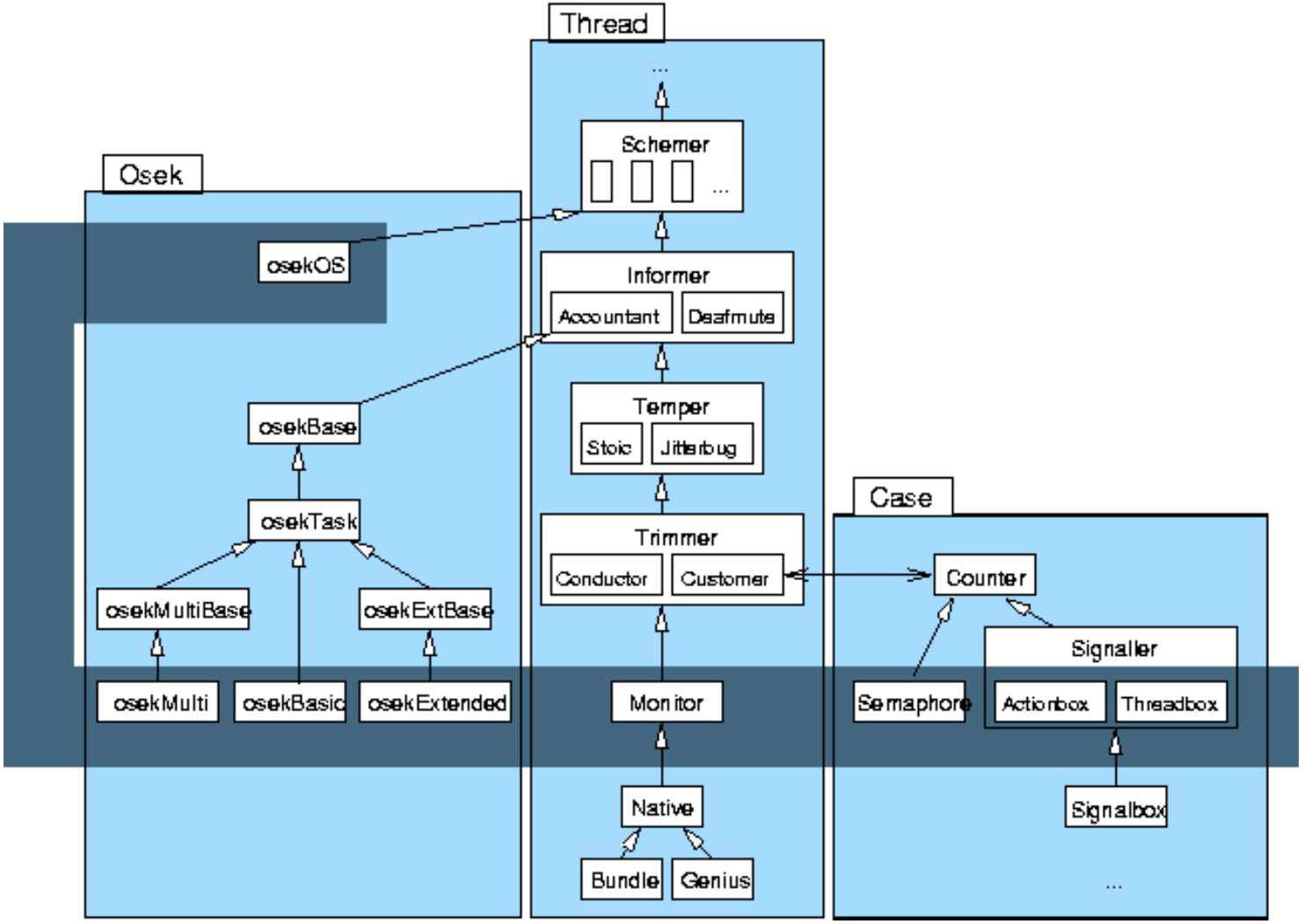
PURE - Konfigurierung

- Problem: je feiner die Granularität der konfigurierbaren Einheiten wird, desto unüberschaubarer ist das System für den Anwender (=Systemkonfigurator).
  - Wie sind bei PURE Klassenaliase zu konfigurieren, um eine gewünschte Wirkung zu erzielen?
  - Nach welchen Regeln werden z.B. Komponenten bei TinyOS und *Knit* verbunden? Nicht nur eine Frage der Schnittstellen!
- Benötigt wird ein Modell und eine „Werkbank“.
  - das Modell wurde übernommen: Merkmalmodelle
  - die Werkbank wurde selbst gemacht: Consulat, jetzt pure::variants

PURE – Aspekte (1)

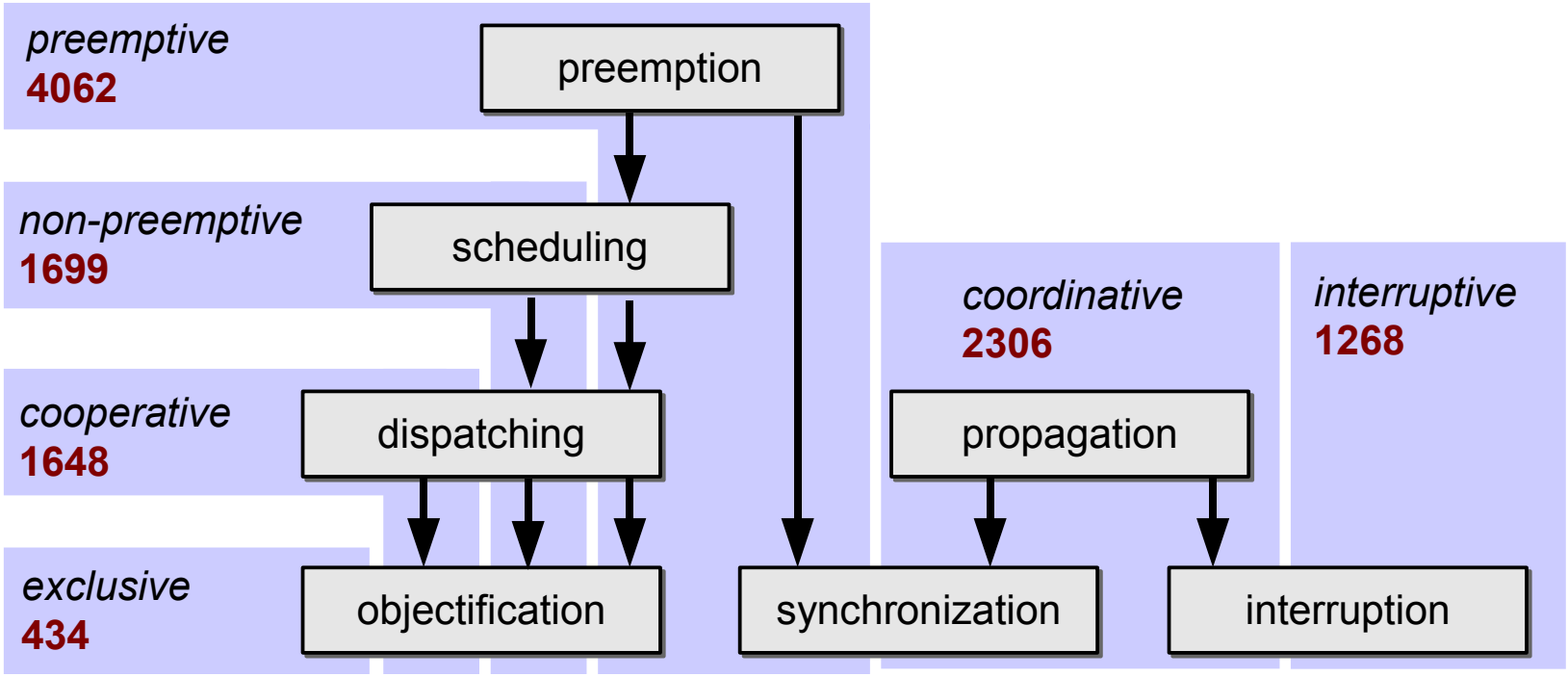
- Problem: quer schneidende Belange
  - redundanter, schwer zu wartender Code
  - schwer wegzulassen
  - Verbindungspunkte nicht konfigurierbar
- Beispiel: Unterbrechungssynchronisation
  - ... beim Eintritt in den Kern
  - 166 Punkte in 15 Klassen
- Benötigt wird ein Modell und eine „Werkzeug“.
  - das Modell wurde übernommen: AOP
  - das Werkzeug wurde selbst gemacht: AspectC++

PURE – Aspekte (2)



PURE - Größen

- ... durchaus mit denen von TinyOS vergleichbar
  - *Inlining* dank C++ und *Inline* Optimierungswerkzeug
  - kaum virtuelle Funktionen durch den familien-basierten Entwurf



PURE - Fazit

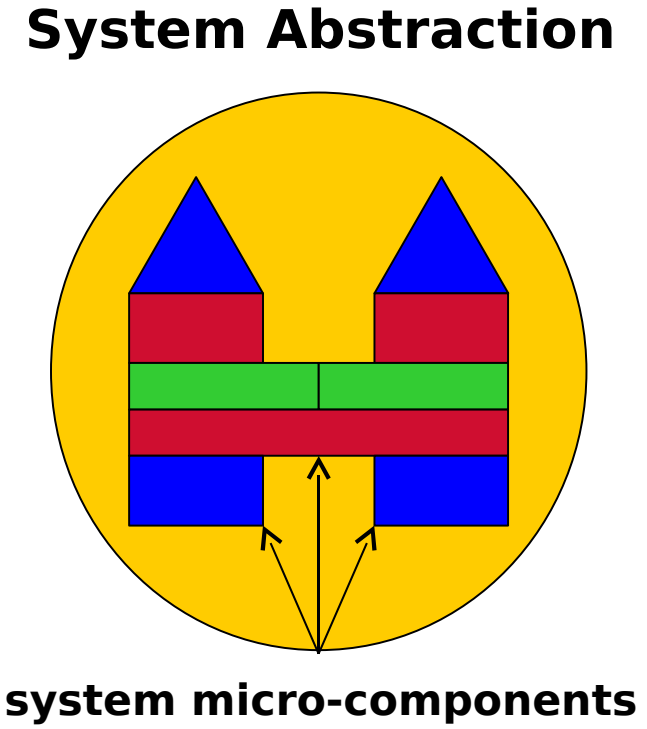
- Feingranular konfigurierbare Systeme werden schnell unüberschaubar
  - Welche Komponenten passen zusammen?
  - Was bedeuten sie für das Gesamtverhalten?
- Der Weg auf eine höhere Abstraktionsebene muss beschritten werden
  - Klassen mit Methoden
  - Komponenten mit Schnittstellen
  - noch ein Schritt höher: abstrakte Merkmale
- Aspekte lösen viele Probleme bei der Umsetzung von Variabilitäten im Programmcode
- PURE hat ca. 250 Merkmale und 300 C++ Klassen

EPOS [8]

- A. Fröhlich, 1999
  GMD FIRST Berlin
- Zieldomäne: eingebettete und parallele Anwendungen
- Ansatz: *Application-Oriented System Design*
  - Szenario-unabhängige Systemabstraktion
  - Szenarioadapter
  - *Inflated Interfaces*
  - statische Anwendungsanalyse
  - statische *Template*-Metaprogrammierung

EPOS - Systemabstraktionen

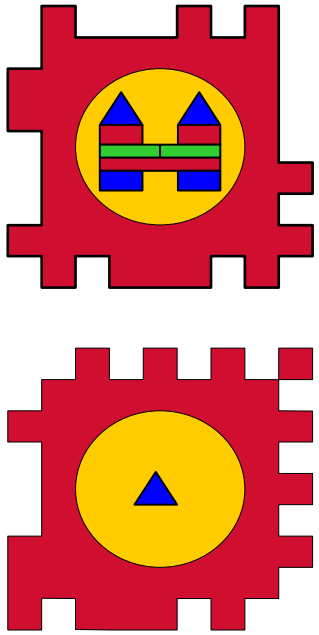
- *„application ready“*
- unabhängig vom Ausführungs-szenario
- Beispiel:
  - ein Faden mit einem bestimmten *Scheduling* Verhalten
  - KEIN Faden für eine Einprozessor- oder Multi-Tasking Umgebung



EPOS - Szenarioadapter

- passt eine Systemabstraktion an eine bestimmte Ausführungs-umgebung an.
- sorgt auch für die Anpassung der Mikro-Komponenten in den Systemabstraktionen
- Beispiele:
  - ein SMP Adapter
  - ein RPC Adapter

Scenario Adapters



EPOS – Anforderungsanalyse (1)

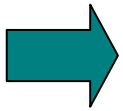
- die Anwendungen werden gegen *Inflated Interfaces* programmiert
  - wohl bekannt
  - leicht verständlich
- nicht alle Systemvarianten haben eine Implementierung für die gesamte Schnittstelle
- Anhand ...
  - der verwendeten Schnittstellen,
  - der referenzierten Funktionen und
  - der Art der Referenzierung
- kann auf die geeignetste Systemkonfiguration geschlossen werden
  - statische Anwendungsanalyse

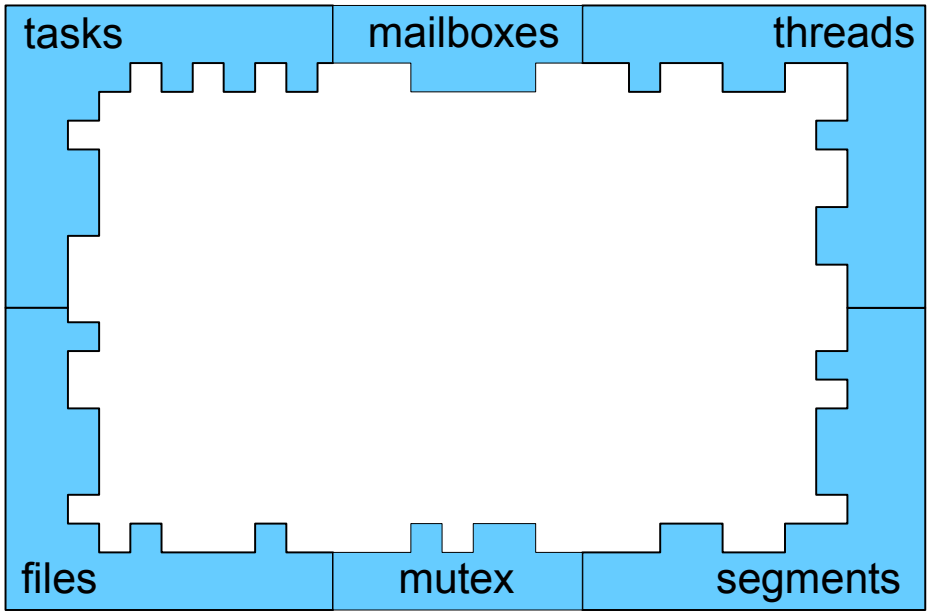
EPOS – Anforderungsanalyse (2)

complex application program

```
code = new Segment(buffer, size);
task = new Task(code, data);
thread = new Thread(task, &entry_func,
      priority, SUSPENDED, arguments);
mutex->entry();
Mailbox mailbox;
mailbox >> message;
File file(name);
file << mailbox;
```

syntactic analyzer



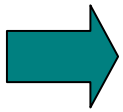


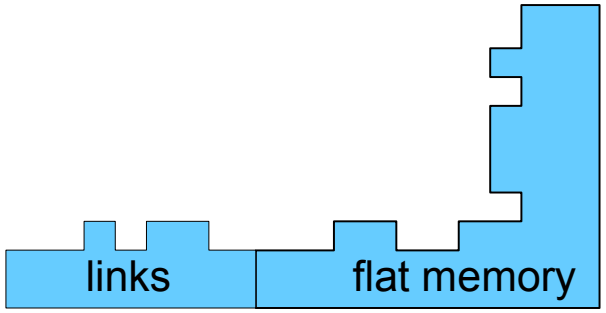
EPOS – Anforderungsanalyse (3)

real application program (MPI)

```
Channel link(destination);
link << message;
```

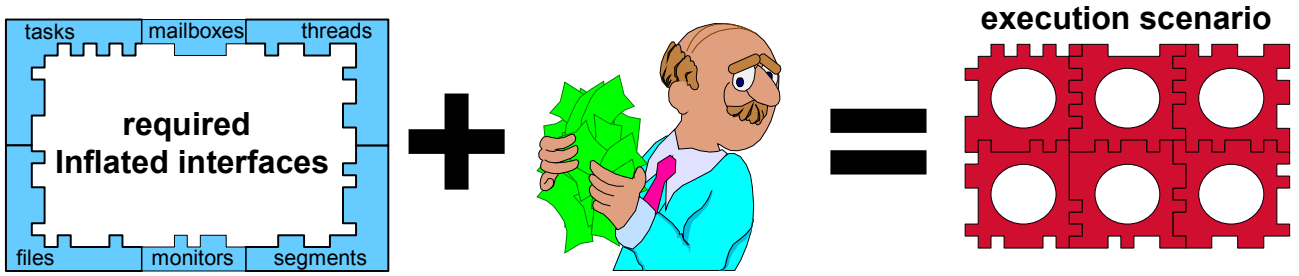
syntactic analyzer





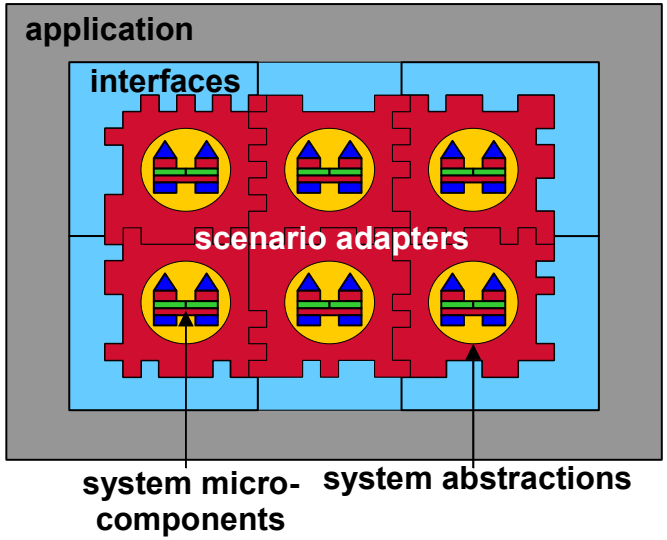
EPOS – Szenarioauswahl

- leider lassen sich nicht alle nötigen Konfigurationsinformationen mit Hilfe der Inflated Interfaces automatisch ableiten
  - Zielhardware
  - zu verwendende Kommunikationsprotokolle
  - Dateisystemtyp
  - ...
- durch manuelle Auswahl wird der passende Szenario-Adapter und die Systemkonfiguration bestimmt
  - durch die Vorauswahl wurde die Komplexität allerdings erheblich reduziert



EPOS – Fazit

- EPOS versucht, es dem Anwendungsprogrammierer leicht zu machen („*application-oriented*“)
  - standardisierte Schnittstellen für alle Mitglieder der (Teil-)Familien
  - automatische Analyse der Anwendung
  - Konfigurierungswerkzeuge
- technisch basiert EPOS auf generischer und generativer Programmierung in C++



Zusammenfassung

- die klassische Betriebssystemforschung fokussiert auf den PC, Workstation und Großrechnerbereich
  - Erweiterbare Systeme
  - Mikrokerne
- neue (bisher vernachlässigte) Anwendungsgebiete geben neue Impulse
  - komponentenbasierte Systeme wie OSKIT und TinyOS
- Baukästen allein tun es aber nicht
  - neue Abstraktion für die Konfigurierung, z.B. Merkmale
  - automatische Anwendungsanalyse zwecks Systemkonfigurierung
- Auffällig:
  - die ideale Programmiersprache für Systemsoftware wurde wohl noch nicht erfunden
  - fast alle Ansätze basieren auf neuen Sprachen

Literatur (1)

[1] G.Denys, F. Piessens, and F. Matthijs, *A Survey of Customizability in Operating Systems Research.* ACM Computing Surveys, Vol. 24, No. 4, Dec. 2002.

[2] R.H. Campbell, G.M. Johnston, P.W. Madany, and V.F. Russo, *Principles of Object-Oriented Operating System Design.* Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[3] R. Campbell, N. Islam, P. Madany, and D. Raila. *Designing and Implementing Choices:an Object-Oriented System in C++*. Communications of the ACM, Sep. 1993.

[4] *The OSKit Project,* http://www.cs.utah.edu/flux/oskit.

Literatur (2)

[5] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide, *Knit: Component Composition for Systems Software.* Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, Oct. 2000.

[6] *The TinyOS Project,* http://www.tinyos.net/.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, *The nesC Language: A Holistic Approach to Networked Embedded Systems*, Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.

[8] A.A. Fröhlich, *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001, ISBN 3-88457-400-0.