

# Synthese Eingebetteter Systeme

Sommersemester 2011

## 6 – SystemC-TLM

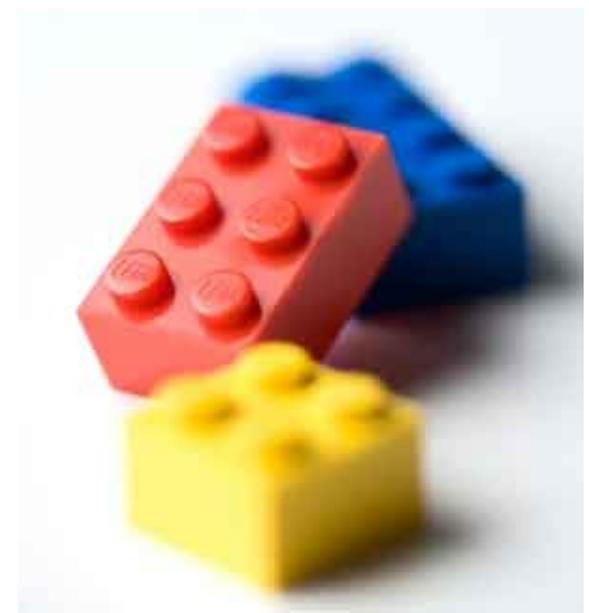
Michael Engel  
Informatik 12  
TU Dortmund

2011/04/27

# SystemC-TLM

---

- Was ist TLM?
- Entwurfsebenen
- TLM in SystemC
- TLM-Beispiel



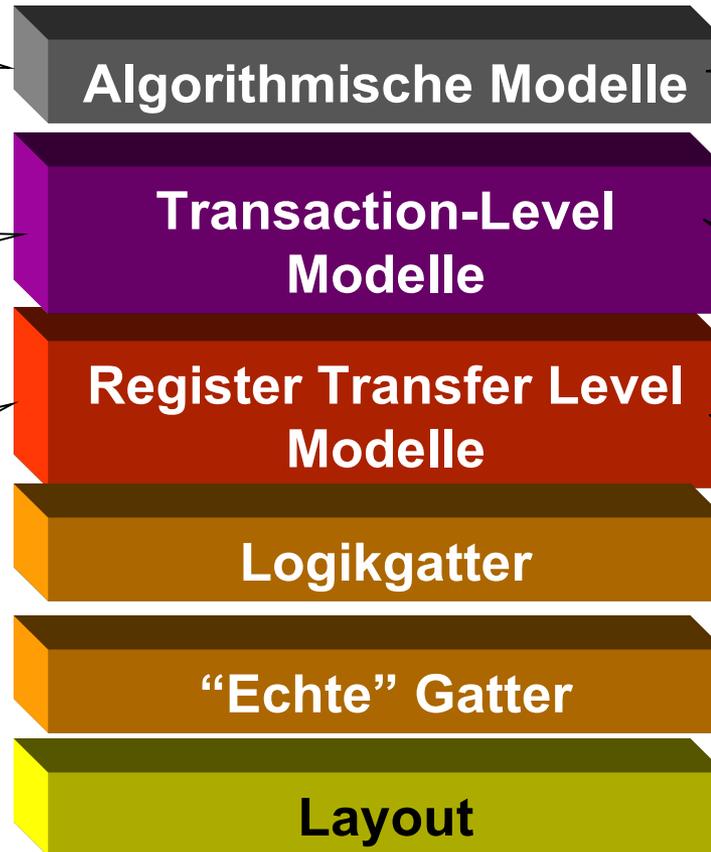
# Abstraktionsebenen

## Kommunikation:

Gemeinsame Variablen

Methoden von Channels

Leitungen und Register



## Sprachen:

C/C++,  
Matlab

SystemC,  
SpecC,  
Metropolis

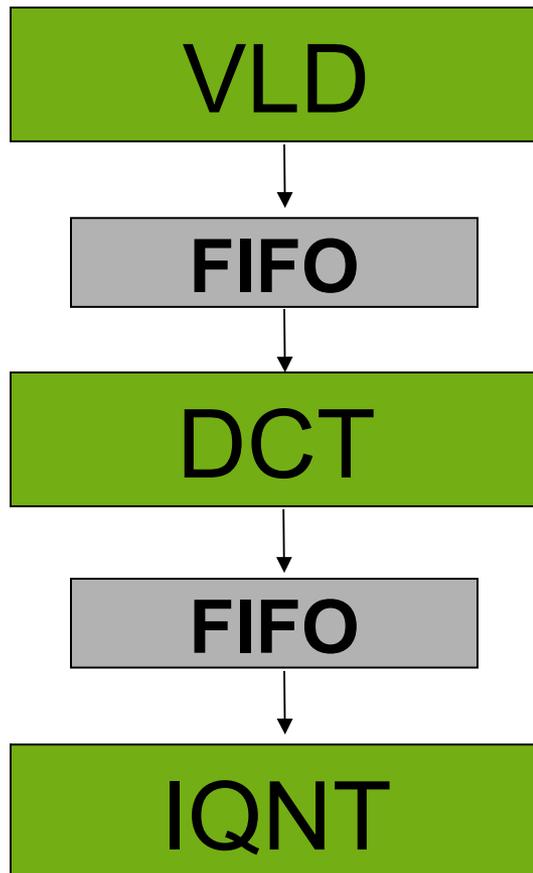
Verilog,  
VHDL

# Transaction-Level-Modeling

---

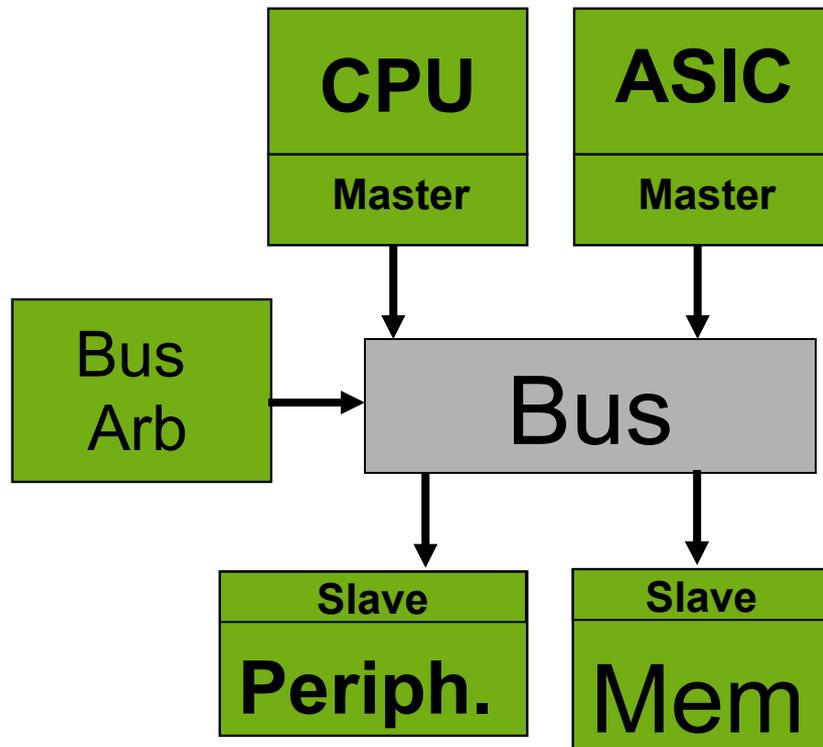
- Bisher: low-level Kommunikationsmodellierung
  - Ports, Channels, Signals
- Komplexe prozessorbasierter Systeme
  - überfordern die Entwickler
  - benötigen höhere Abstraktionsebene
- Transaction-Level Modeling (TLM)
- TLM-Klassenbibliothek für SystemC
  - Teil der Open SystemC Initiative (OSCI)
  - SystemC-Erweiterung zur Modellierung Prozessorbasierter Systeme

# Software: Nebenläufige Prozesse



- Explizite Nebenläufigkeit
  - Einfacher handhabbar als Extraktion aus sequentiellem Quellcode
  - Gut zur Modellierung von Hardware geeignet
- Kommunikation via channels
  - Punkt-zu-Punkt-Kommunikation
  - Communication
  - Methodenaufrufe an Stelle von Signalen
  - Protokolle werden wegabstrahiert
- Mit und ohne Zeitabhängigkeiten

# Sicht des Programmierers



- Ähnlich zur Architektur
  - Registergenau
  - Nützlich für SW-Entwicklung und Prototyping
- Gemeinsame Kommunikation
  - Potentiell Arbitrierung
  - Blockierend/nicht-blockierend
- Mit und ohne Zeitabhängigkeiten

# Ziele von Transaction-Level-Modeling

---

- Ereignisgetriebene Simulation
  - Daher “Transaktion”
  - Simulation wird durch Datenaustausch getriggert
  - Unterstützung von Bus-Events
- Ziel
  - Hohe Simulationsgeschwindigkeit
  - Vereinfachte Modellierung
  - Systemanalyse in frühen Entwurfsphasen ermöglichen

# Definitionen

---

- Transaktion
  - Austausch von Daten oder Ereignissen zwischen zwei Komponenten eines modellierten und simulierten Systems
- Modul
  - Strukturelle Einheit, die Prozesse, Ports, Channels und andere Module enthält
- Channel
  - Implementiert ein oder mehrere Interfaces, dient als Container für Kommunikationsfunktionalität
- Port
  - Objekt, durch das ein Modul auf das Interface eines Channels zugreifen kann

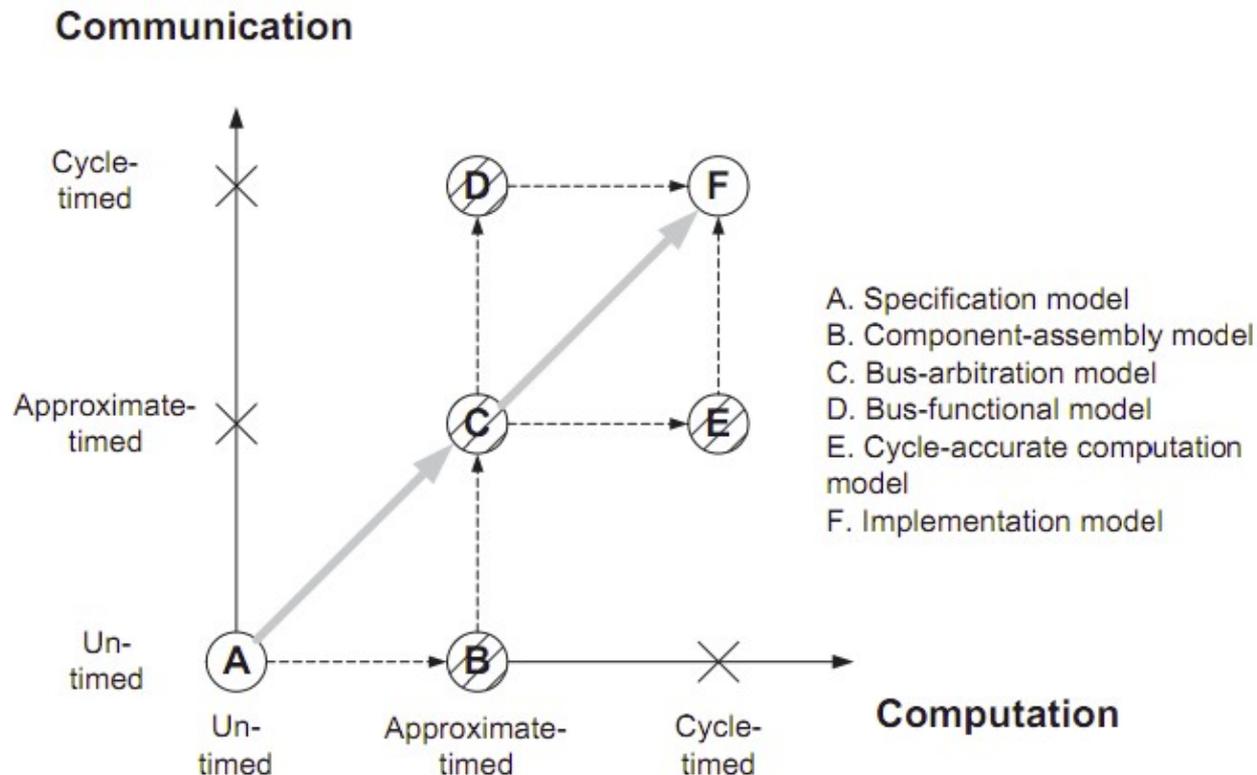
# Definition von TLM

---

- TLM =  $\langle \{\text{Objekte}\}, \{\text{Kompositionen}\} \rangle$
- Objekte
  - Berechnungsobjekte + Kommunikationsobjekte
- Komposition
  - Berechnungsobjekte lesen/schreiben abstrakte Datentypen (oberhalb von Pin-Genauigkeit) mittels Kommunikationsobjekten
  - Explizite Trennung von Kommunikation & Berechnung
- Vorteile
  - Objektunabhängigkeit
    - Objekte unabhängig voneinander modellierbar
  - Abstraktionsunabhängigkeit
    - Unterschiedliche Objekte auf unterschiedlichen Abstraktionsebenen modellierbar

# Abstraktionsmodelle

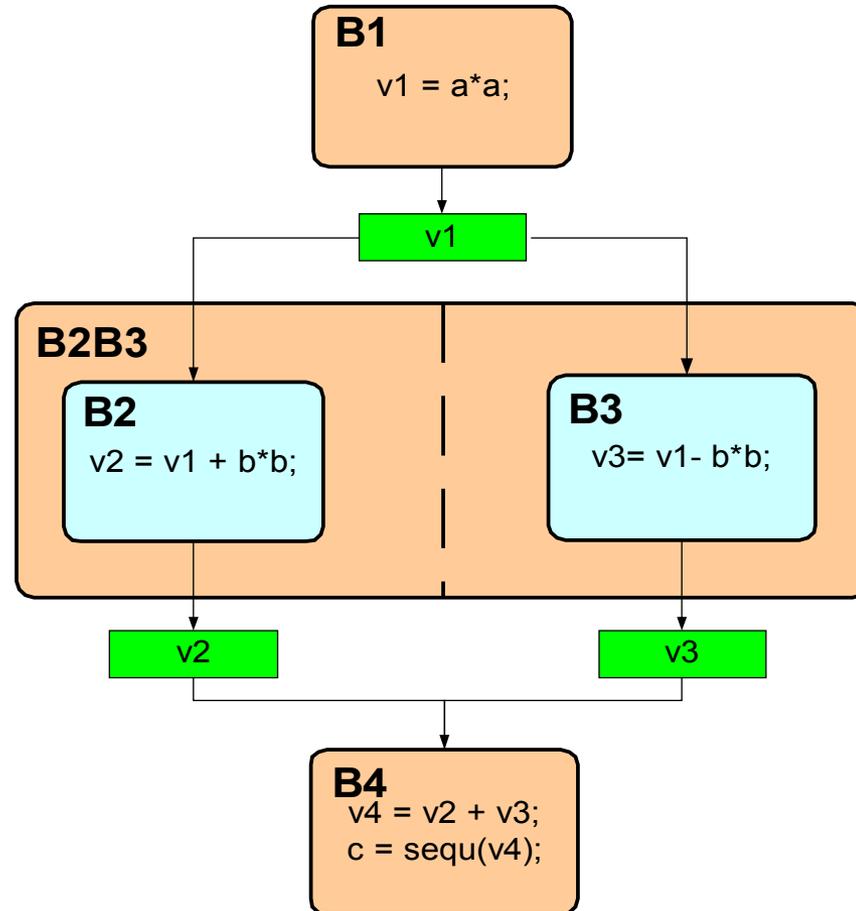
- Zeitgranularität von kommunizierenden/rechnenden Objekten läßt sich in drei Kategorien einteilen
- Modelle B, C, D und E könnten als TLMs klassifiziert werden



# A: Spezifikationsmodell

## Objekte

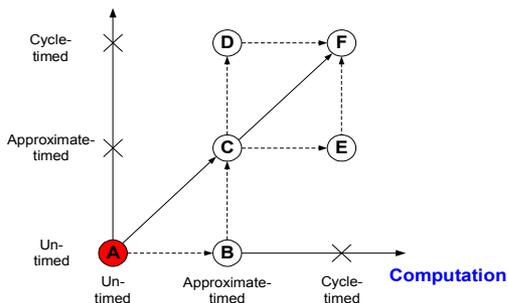
- Berechnung
  - Verhalten
- Kommunikation
  - Variablen



## Komposition

- Hierarchie
- Reihenfolge
  - Sequentiell
  - Parallel
  - Piped
  - Zustände
- Übergänge
  - TI, TOC
- Synchronisation
  - Notify/Wait

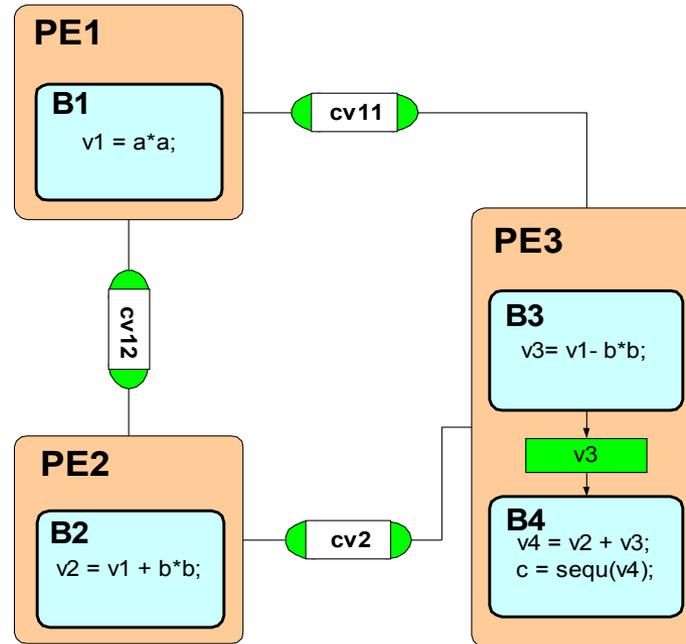
### Communication



# B: Komponentenmodell

## Objekte

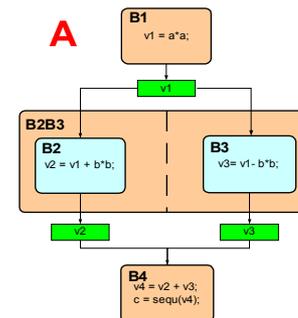
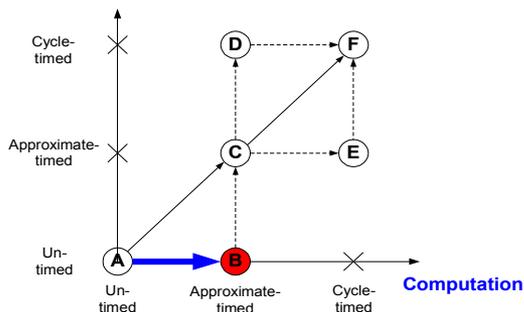
- Berechnung
  - Prozessoren
  - IPs
  - Speicher
- Kommunikation
  - Variablen
  - Kanäle



## Komposition

- Hierarchie
- Reihenfolge
  - Sequentiell
  - Parallel
  - Piped
  - Zustände
- Übergänge
  - TI, TOC
- Synchronisation
  - Notify/Wait

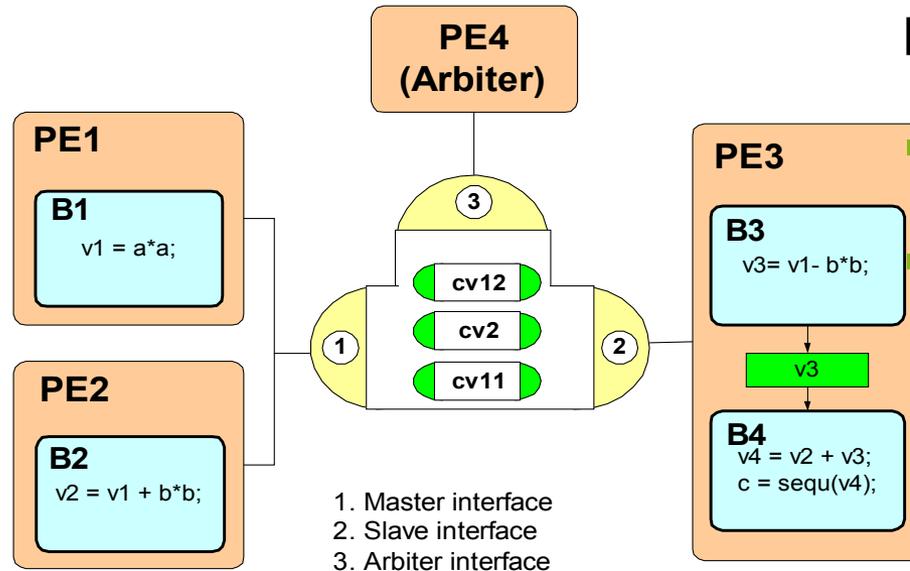
### Communication



# C: Busarbitrierungsmodell

## Objekte

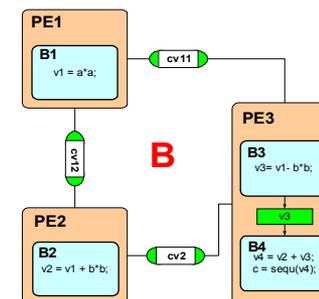
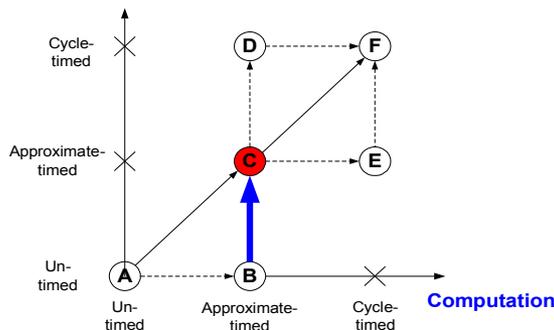
- **Berechnung**
  - Prozessoren
  - IPs (Arbitrierer)
  - Speicher
- **Kommunikation**
  - Abstrakte Busse
  - Kanäle



## Komposition

- **Hierarchie**
- **Reihenfolge**
  - Sequentiell
  - Parallel
  - Piped
  - Zustände
- **Übergänge**
  - TI, TOC
- **Synchronisation**
  - Notify/Wait

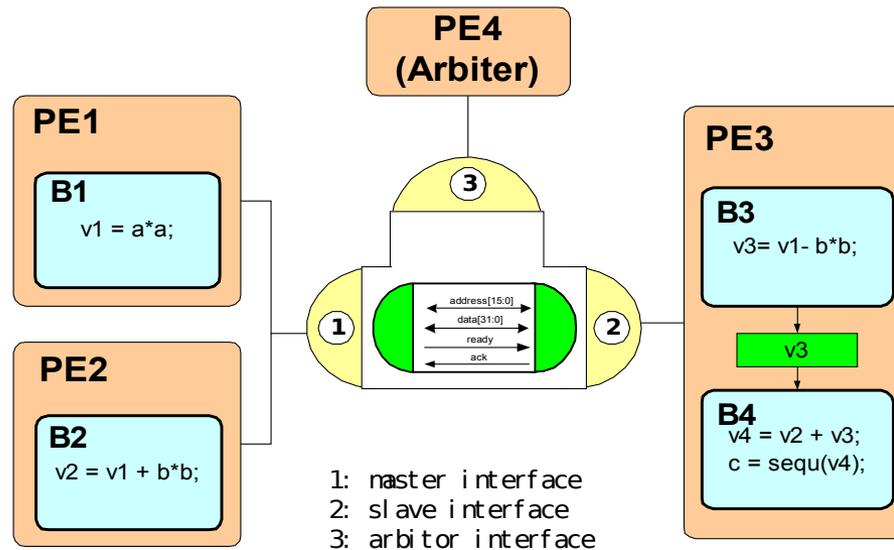
## Communication



# D: Funktionales Busmodell

## Objekte

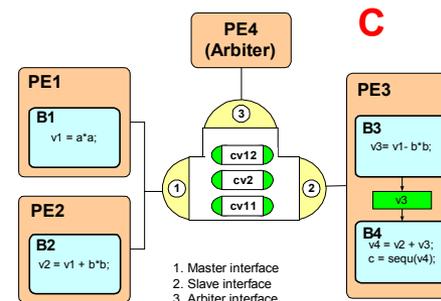
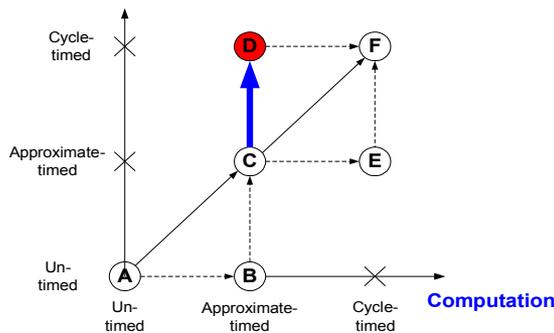
- Berechnung
  - Prozessoren
  - IPs (Arbitrierer)
  - Speicher
- Kommunikation
  - Protokolle
  - Busse
  - Kanäle



## Komposition

- Hierarchie
- Reihenfolge
  - Sequentiell
  - Parallel
  - Piped
  - Zustände
- Übergänge
  - TI, TOC
- Synchronisation
  - Notify/Wait

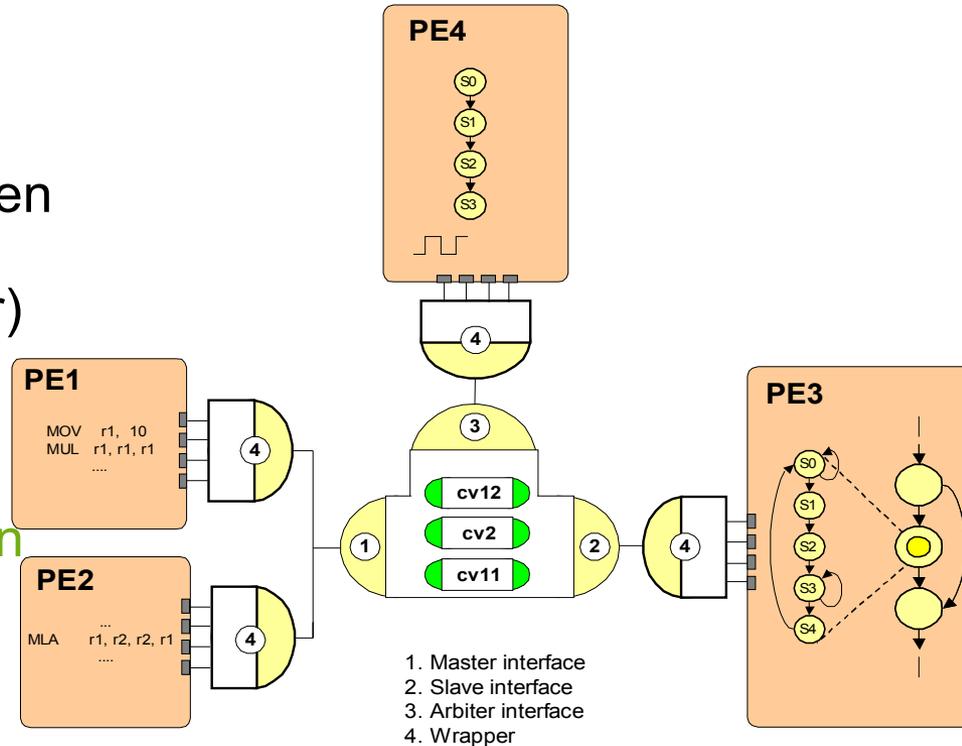
## Communication



# E: Zyklengenaues Berechnungsmodell

## Objekte

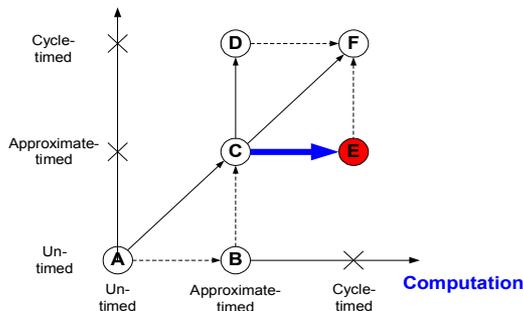
- **Berechnung**
  - Prozessoren
  - IPs (Arbitrierer)
  - Speicher
  - Wrapper
- **Kommunikation**
  - Abstrakte Busse
  - Kanäle



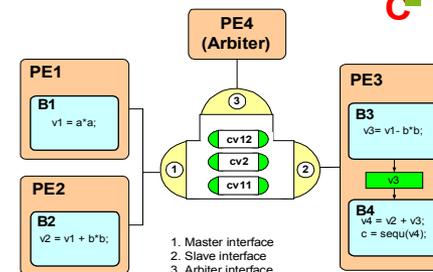
## Komposition

- Hierarchie
- Reihenfolge
  - Sequentiell
  - Parallel
  - Piped
  - Zustände
- Übergänge
  - TI, TOC

### Communication



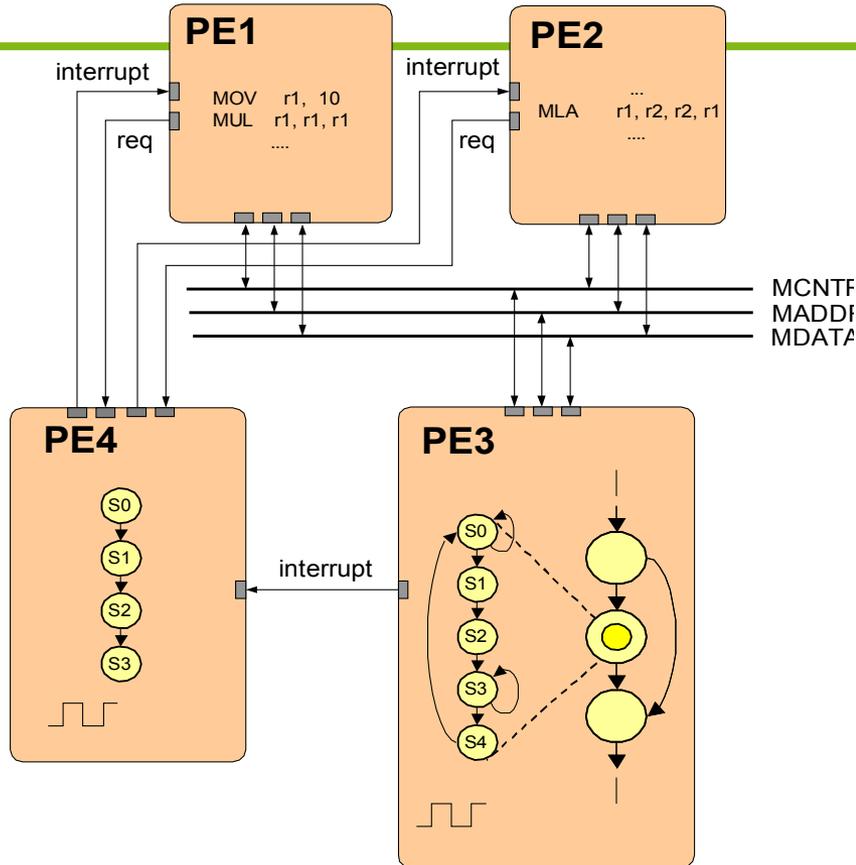
- **Synchronisation**
  - Notify/Wait



# F: Implementierungsmodell

## Objekte

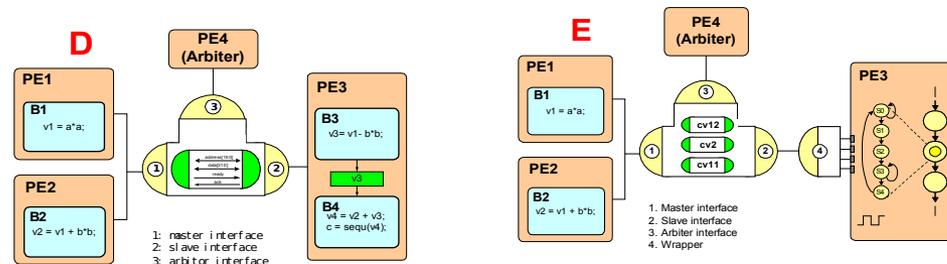
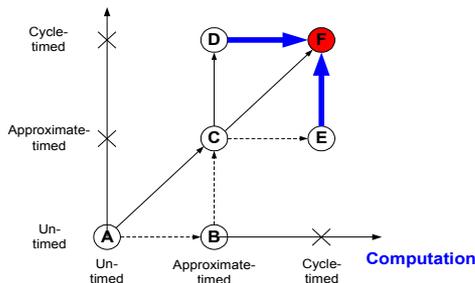
- Berechnung
  - Prozessoren
  - IPs (Arbitrierer)
  - Speicher
- Kommunikation
  - Busse (Leitungen)



## Komposition

- Hierarchie
- Reihenfolge
  - Sequentiell
  - Parallel
  - Piped
  - Zustände
- Übergänge
  - TI, TOC
- Synchronisation
  - Notify/Wait

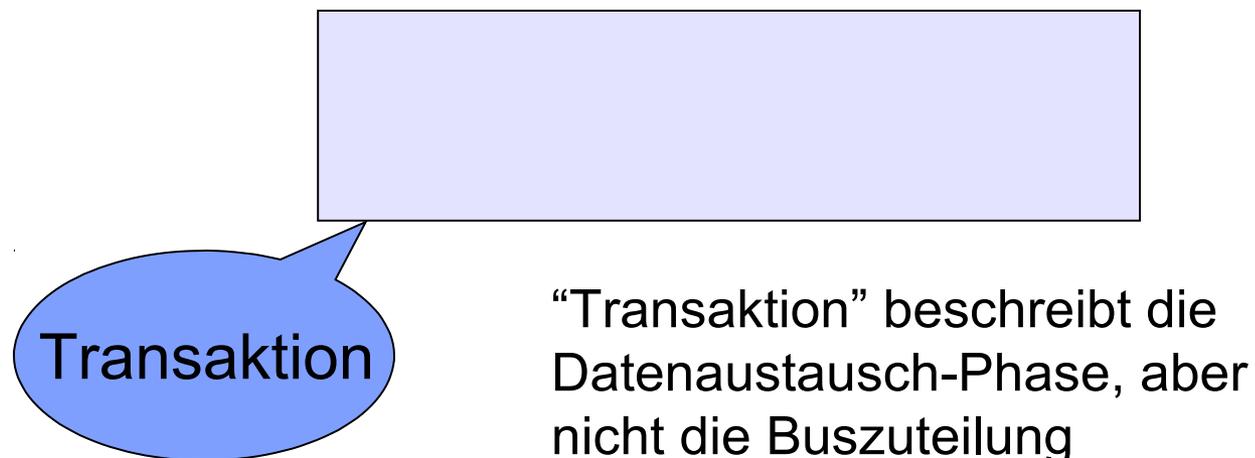
### Communication



# Beispiel: Bussystem – TLM-Sicht

---

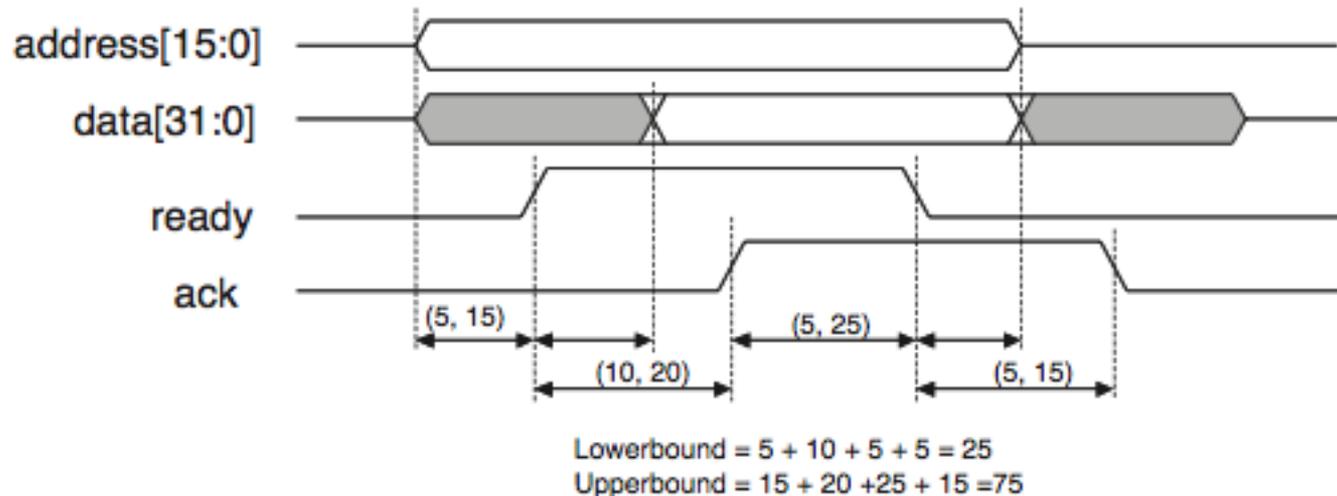
- Abstraktion von Implementierungsdetails



# Beispiel: Bussystem

## Tatsächlicher Ablauf – Zeitdiagramm

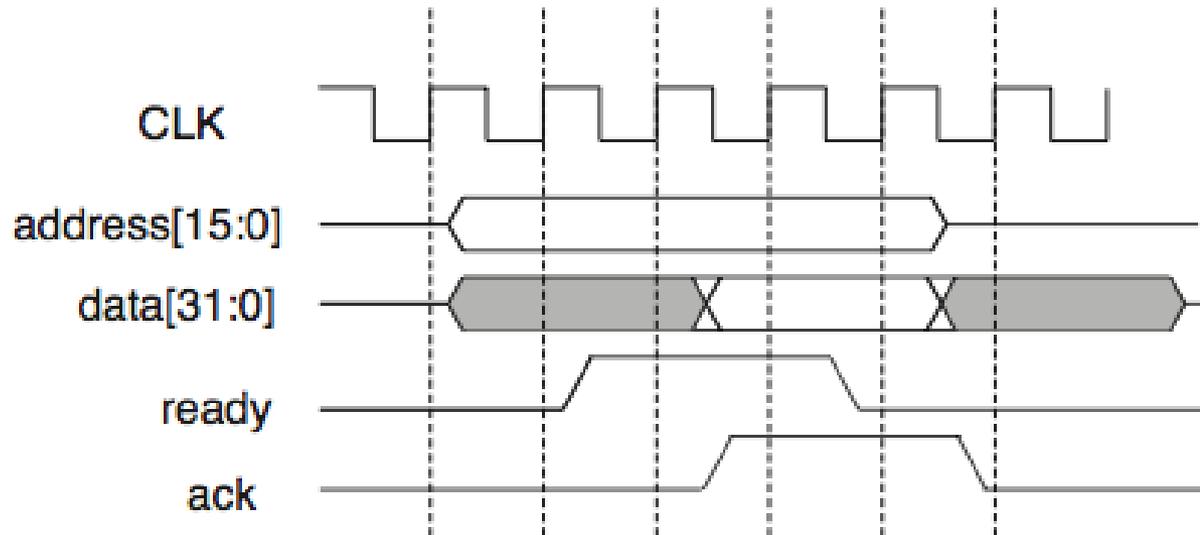
- Komplexe Abläufe und Reihenfolgen, Zeitabhängigkeiten, Synchronisation



# Beispiel: Bussystem

## Tatsächlicher Ablauf – Zyklengenau

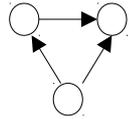
- Berücksichtigung von Takt (CLK)



# Charakteristika verschiedener Abstraktionen

Modelle	Kommunikationszeit	Berechnungszeit	Kommunikationsschema	Schnittstelle zu Prozessorel.
<b>Spezifikationsmodell</b>	nein	nein	Variabel	(keine PE)
<b>Komponentenmodell</b>	nein	ungefähr	Variabler Kanal	abstrakt
<b>Busarbitrierungsmodell</b>	ungefähr	ungefähr	abstrakter Bus Channel	abstrakt
<b>Funktionales Busmodell</b>	Zeit-/zyklusgenau	ungefähr	Protokollbus Channel	abstrakt
<b>Zyklengenaues Berechnungsm.</b>	ungefähr	zyklengenau	abstrakter Bus Channel	pingenau
<b>Implementierungsmodell</b>	zyklengenau	zyklengenau	Bus (Leitung)	pingenau

# Modellalgebra

- Algebra =  $\langle \{\text{Objekte}\}, \{\text{Operationen}\} \rangle$  [z.B.:  $a * (b + c)$ ]
- Modell =  $\langle \{\text{Objekte}\}, \{\text{Kompositionen}\} \rangle$  [z.B.:  ]
- Transformation  $t(model)$  ist eine Änderung der Objekte oder Kompositionen
- Modellverfeinerung: geordnete Menge von Transformationen  $\langle t_m, \dots, t_2, t_1 \rangle$ .  
Damit ist das Modell  $B = t_m( \dots ( t_2( t_1( model A ) ) ) \dots )$
- Modellalgebra =  $\langle \{\text{Modelle}\}, \{\text{Verfeinerungen}\} \rangle$
- Vorgehensweise: Abfolge von Modellen und zugehörigen Verfeinerungen

# Modell-Definition

---

- Modell =  $\langle \{\text{Objekte}\}, \{\text{Kompositionsregeln}\} \rangle$
- Objekte
  - Verhalten (stellen Tasks/Berechnungen/Funktionen dar)
  - Channels (stellen Kommunikation zwischen Verhaltenselementen dar)
- Kompositionsregeln
  - Sequentiell, parallel, pipelined, FSM
  - Verhaltenskomposition erzeugt Hierarchie
  - Verhaltenskomposition erzeugt Ausführungsreihenfolge
    - Zusammenhang zwischen Verhalten und Kanälen
  - Datenübertragung zwischen Channels
  - Schnittstelle zwischen Verhalten und Channels

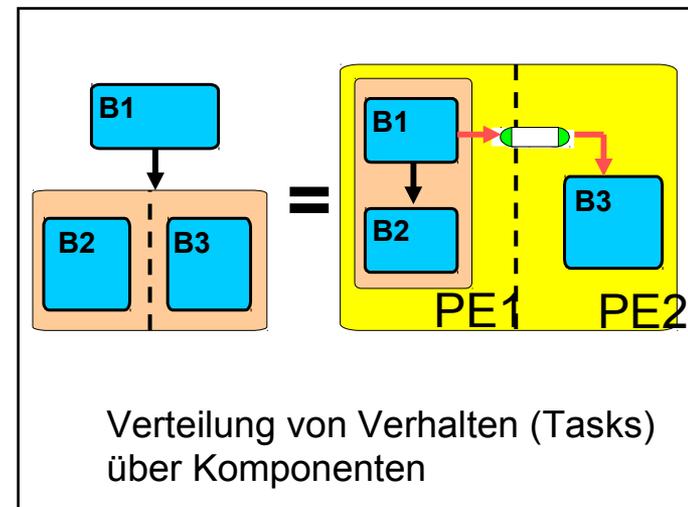
# Modelltransformationen (Umbauen und Ersetzen)

- Objektkomposition umbauen
  - Um Berechnungen auf Komponenten zu verteilen
- Objekte ersetzen
  - Import von Librarykomponenten
- Synchronisation hinzufügen/entfernen
  - Um sequentielle Komposition korrekt in parallele zu transformieren und umgekehrt
- Dekomposition abstrakter Datenstrukturen
  - Implementierung von Daten-Transaktionen über einen Bus
- Andere Transformationen

$$a*(b+c) = a*b + a*c$$

Distributivität der Multiplikation über der Addition

analog zu.....

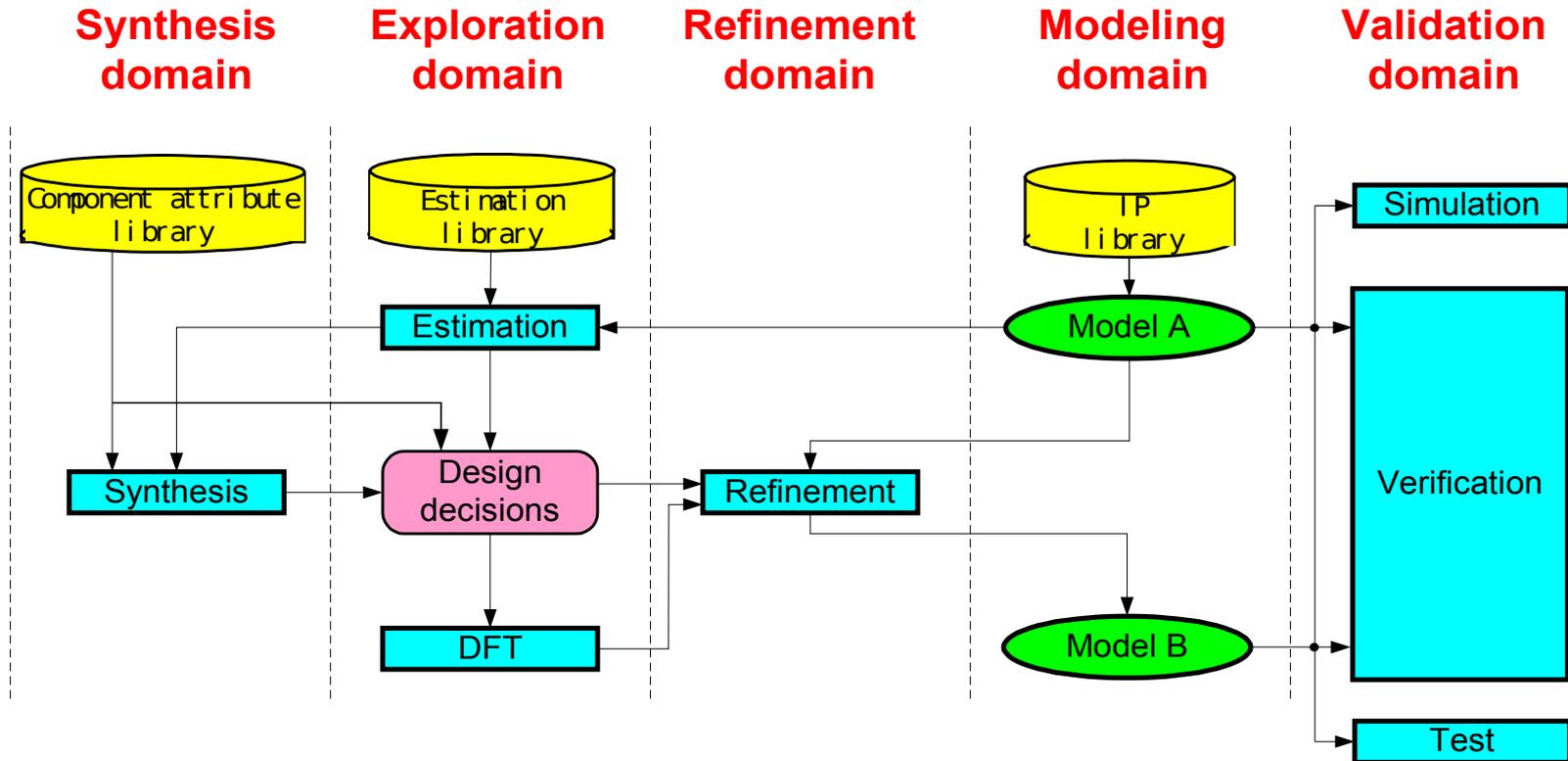


# Verfeinerung des Modells

---

- Definition
  - Verfeinerung = geordnete Menge von Transformationen  $\langle t_m, \dots, t_2, t_1 \rangle$
  - *Modell  $B = t_m( \dots ( t_2( t_1( model A ) ) ) \dots )$*
- Leitet detailliertes Modell von einem abstrakteren ab
  - Spezifische Abfolge für jede Modell-Verfeinerung
  - Nicht alle Folgen sind relevant
- Verifikation der Äquivalenz
  - Jede Transformation erhält die funktionale Äquivalenz
  - Die Verfeinerung ist damit durch Konstruktion korrekt
  - Methodik auf Systemebene basiert auf Verfeinerung
  - Methodik besteht aus Abfolge von Modellen und Verfeinerungen

# Entwurfsdomänen



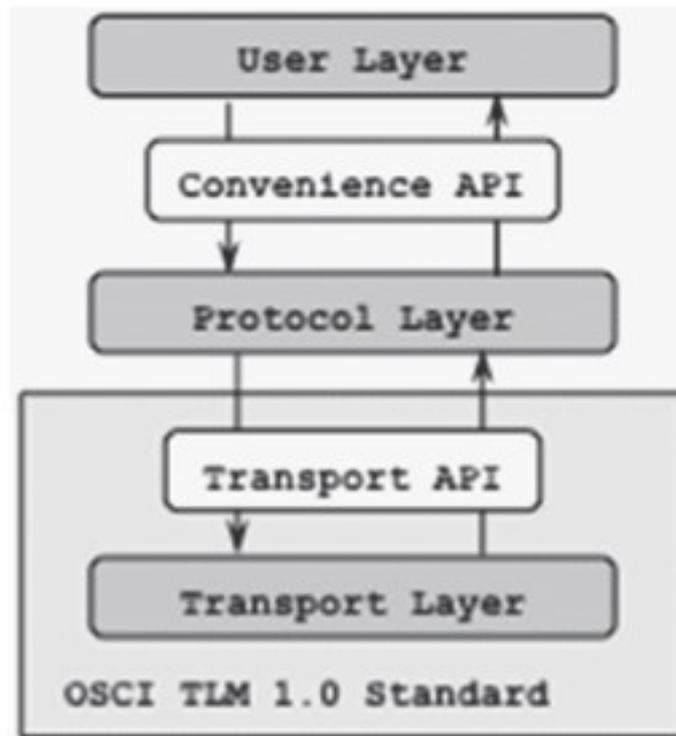
# Synthese

---

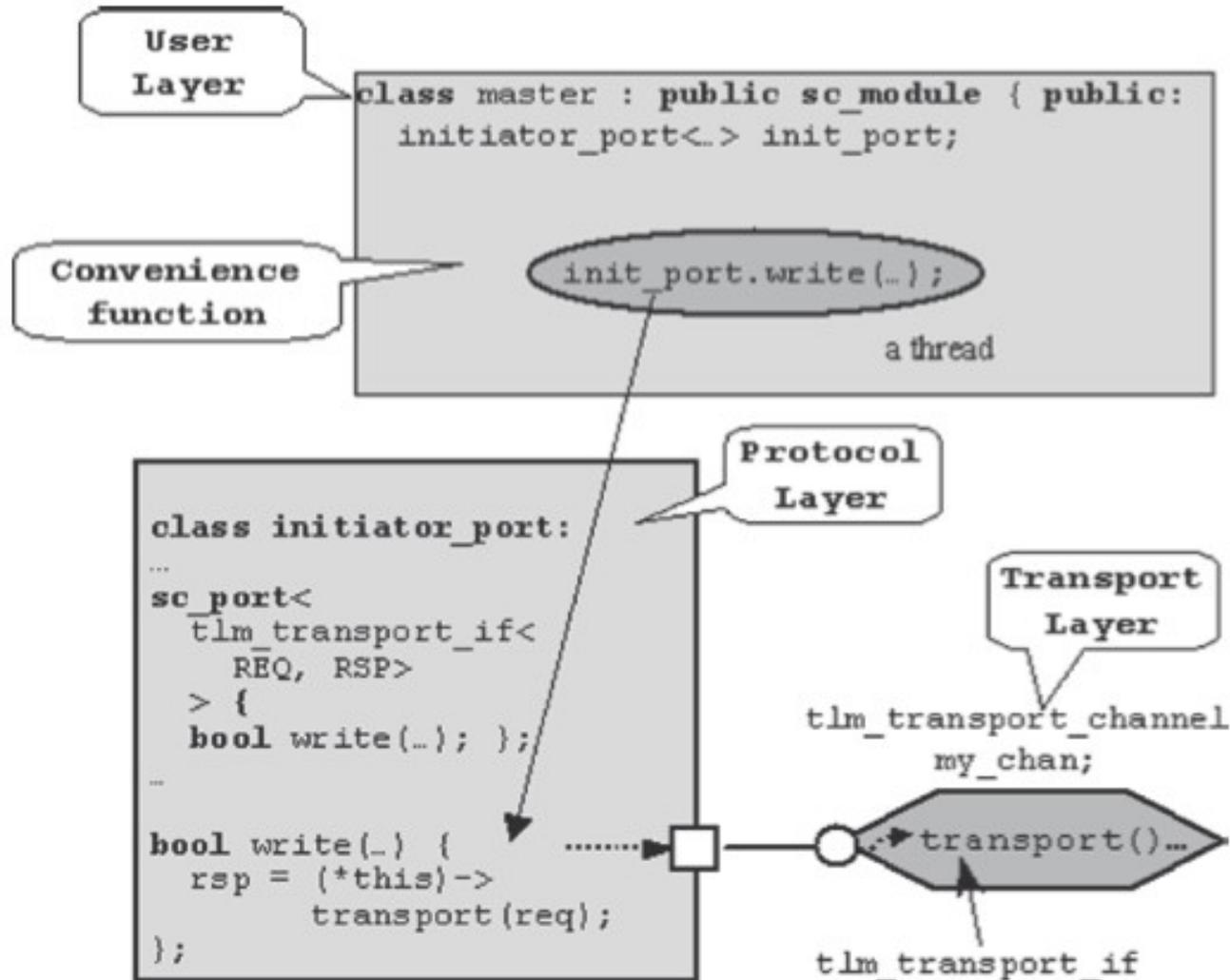
- Menge von Modellen
- Menge von Entwurfsaufgaben
  - Profile
  - Erkunden
  - Auswahl von Komponenten / Verbindungen
  - Abbildung von Verhalten / Kanälen
  - Scheduling von Verhalten / Kanälen
- Jede Entwurfsentscheidung => Modelltransformation
- Detaillierung ist Folge von Entwurfsentscheidungen
- Verfeinerung ist Folge von Transformationen
- Synthese = Detaillierung + Verfeinerung
- Aufgabe: Definition der Folge von Entwurfsentscheidungen und Transformationen
- **Synthesedetails in den folgenden Vorlesungen!**

# TLM in SystemC

- Architektur



# TLM in SystemC

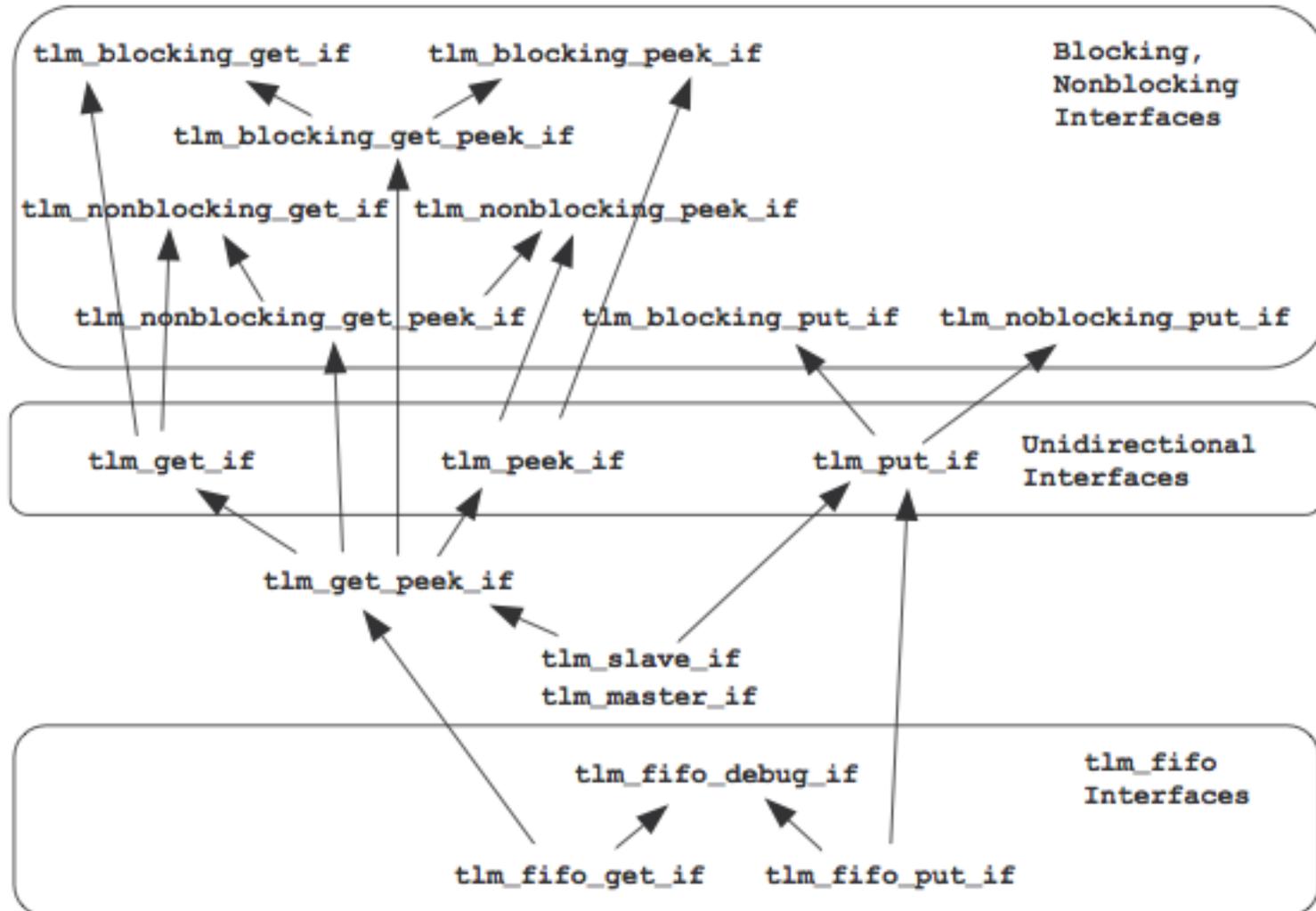


# TLM in SystemC

---

- Drei Schnittstellenkategorien
  - Unidirektional, blockierend
  - Unidirektional, nicht blockierend
  - Bidirektional, blockierend

# TLM-Klassenhierarchie



# Unidirektionale blockierende Schnittstellen

---

```
tlm_uni_channel<T> instance("instance");
instance.get(result_ptr);
result = get();
instance.put(value);
```

- Die Funktionen **get()** und **put()** entsprechen *read()* und *write()* bei **sc\_fifo<T>**
- Offensichtlich können die Funktionen **get()** und **put()** **waits** enthalten und erst zurückkehren, wenn Daten übertragen wurden
- Daher lassen sich diese Funktionen nur aus einem **SC\_THREAD**-Prozess heraus aufrufen
  - **SC\_METHOD**-Prozesse dürfen nicht blockieren!

# Unidirektionale blockierende Schnittstellen

---

```
tlm_uni_channel<T> instance("instance");
instance.peek(result_ptr);
instance.peek(result_ptr, offset);
result = peek();
result = peek(offset);
instance.poke(value);
instance.poke(value, offset);
```

- Die blockierenden Schnittstellen besitzen auch Klassen zum Prüfen auf Verfügbarkeit von Daten, ohne diese selbst zu lesen
  - Nützlich für verteilte Decodierung
  - Benötigt, wenn ein Ziel herausfinden muss, ob die Daten für diese Ziel bestimmt sind, bevor die Daten gelesen werden

# Unidirektionale nichtblockierende Schnittstellen

```
tlm_uni_channel<T> instance("instance");
if (not instance.nb_get(variable) {
    next_trigger(instance.ok_to_get());
}
if (instance.nb_can_get()) {
    cout << "instance available for get" << endl;
}
if (not instance.nb_put(value) {
    next_trigger(instance.ok_to_put());
}
if (instance.nb_can_put()) {
    cout << "instance available for put" << endl;
}
```

- Wie bei den blockierenden Schnittstellen auch beinhalten diese APIs **nb\_get()** und **nb\_put()**-Funktionen
- Benutzer kann testen, ob Datentransfer erfolgreich sein wird, bevor er Übertragungsfunktionen direkt aufruft
  - Auch bei Funktionen mit Rückgabewerten oder Events, die anzeigen, dass Transfer fortfahren kann

# Unidirektionale nichtblockierende Schnittstellen

---

- Keine Garantie, ob Datentransfer erfolgreich sein wird
  - Auch nicht nach Warten auf **ok\_to\_put()** und **ok\_to\_get()**-Events
  - Abhängig von FIFOs und der Anzahl an Requestern
  - Erfolg einer Übertragung kontrollierbar anhand des Rückgabewertes (fail/success) der Übertragungsfunktionen
- Nichtblockierende Funktionen geben **bool** zurück
  - Gibt an, ob Datentransfer tatsächlich stattgefunden hat
  - Nichtblockierende Schnittstellen können aus **SC\_METHOD** und **SC\_THREAD**-Prozessen heraus aufgerufen werden

# Unidirektionale nichtblockierende Schnittstellen

---

- Nichtblockierende Debug-Schnittstellenmethoden existieren
  - Liefern zusätzliche Eigenschaften, hauptsächlich für Verifikation verwendet
  - Debug-Schnittstellen simulieren keine Zeit, da sie nur für Debugging gedacht sind
- Offset in *peek*- und *poke*-Funktionen stellt die “Tiefe” der Schnittstelle dar (z.B. bei einer FIFO)
  - Offset 0 spricht die “oberste” Information an (der von **get()** gelieferte Wert)
  - Andere Werte schauen tiefer in die Schnittstelle
  - Für eine FIFO ist der Offset einleuchtend
  - Bei anderen Konstrukten evtl. nicht => Dokumentation des Channel-Entwicklers notwendig

# Unidirektionale nichtblockierende Schnittstellen

---

- Beispiel für Debug-Schnittstellenaufrufe:

```
tlm_uni_channel<T> instance("instance");
if (!instance.nb_peek(variable, offset) {
    next_trigger(instance.ok_to_peek())&
}
if (instance.nb_can_peek(offset)) {
    cout << "instance available for get" << endl;
}
if (!instance.nb_poke(value) {
    next_trigger(instance.ok_to_poke())&
}
if (instance.nb_can_poke()) {
    cout << "instance available for poke" << endl;
}
```

# Bidirektionale blockierende Schnittstellen

```
tlm_transport<REQ,RSP> instance;  
result = instance.transport(request);
```

- Für Fälle, in denen eine 1:1-Beziehung zwischen Anfrage und Antwort existiert
- Schnittstelle macht das Leben leichter...
  - Basisimplementierung kann blockierende unidirektionale **put()** und **get()** –Funktionen nutzen
  - Der OSCI TLM-Standard implementiert dies so
  - Anfrage und Antwort müssen unterschiedliche Klassen sein
    - Vermeidung von Mehrdeutigkeiten innerhalb der Klasse
    - Erreichbar durch Ableitung einer Funktion von der anderen
    - Vermeidet C++-Mehrdeutigkeit zur Unterscheidung der Signaturen zweier **get()**- und **put()**-Funktionen in der Klassenimplementierung

# TLM Channels

---

- Channels implementieren APIs der Schnittstellen, die sie erben
- Drei Beispiel-Channels im OSCI TLM-Standard:
  - **tlm\_fifo<T>** - *implementiert unidirektionale Schnittst.*
  - **tlm\_req\_rsp\_channel<Req,Rsp>** - *implementiert zwei unidirektionale Schnittstellen*
  - **tlm\_transport\_channel<Req,Rsp>** - *implementiert bidirektionale Schnittstelle*
- Diese Channels sind **nicht** Teil der TLM-Spezifikation!
  - Unterstützende Kommunikationskomponenten
  - In einigen Fällen ist die Entwicklung protokoll-spezifischer Channels notwendig
    - Wenn der Entwurf das erfordert

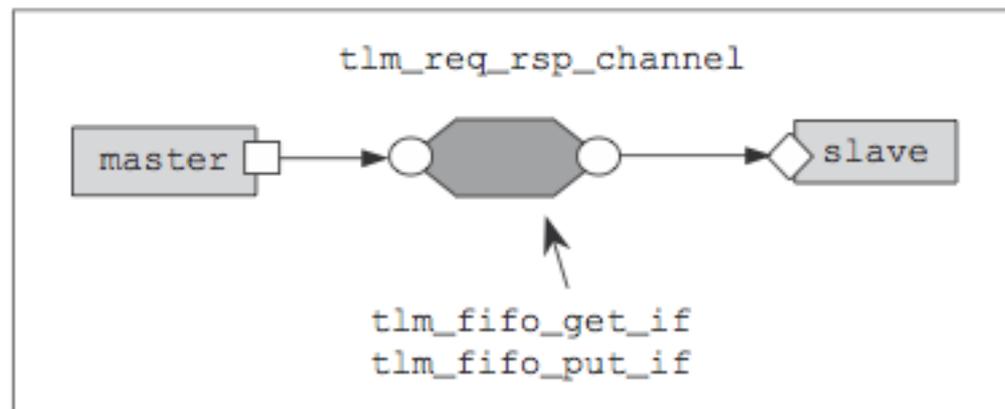
# TLM Channels

---

- Der **tlm\_fifo**<*T*> -Kanal ist in Anlehnung an die SystemC FIFO-Klasse modelliert
  - Implementiert die unidirektionalen Schnittstellen
  - blockierend und nichtblockierend
  - FIFO-Tiefe wählbar von 0 bis unendlich

# TLM Channels

- Der Channel `tlm_req_rsp_channel<Req,Rsp>` implementiert zwei unidirektionale Schnittstellen mit TLM FIFOs
  - Eine FIFO für Anfragen und die andere für Antworten
  - Aus Sicht der Verbindung exportiert der `tlm_req_rsp_channel<Req,Rsp>` das “put request” FIFO interface und “get response” FIFO interface zum Master hin



# TLM Channels

---

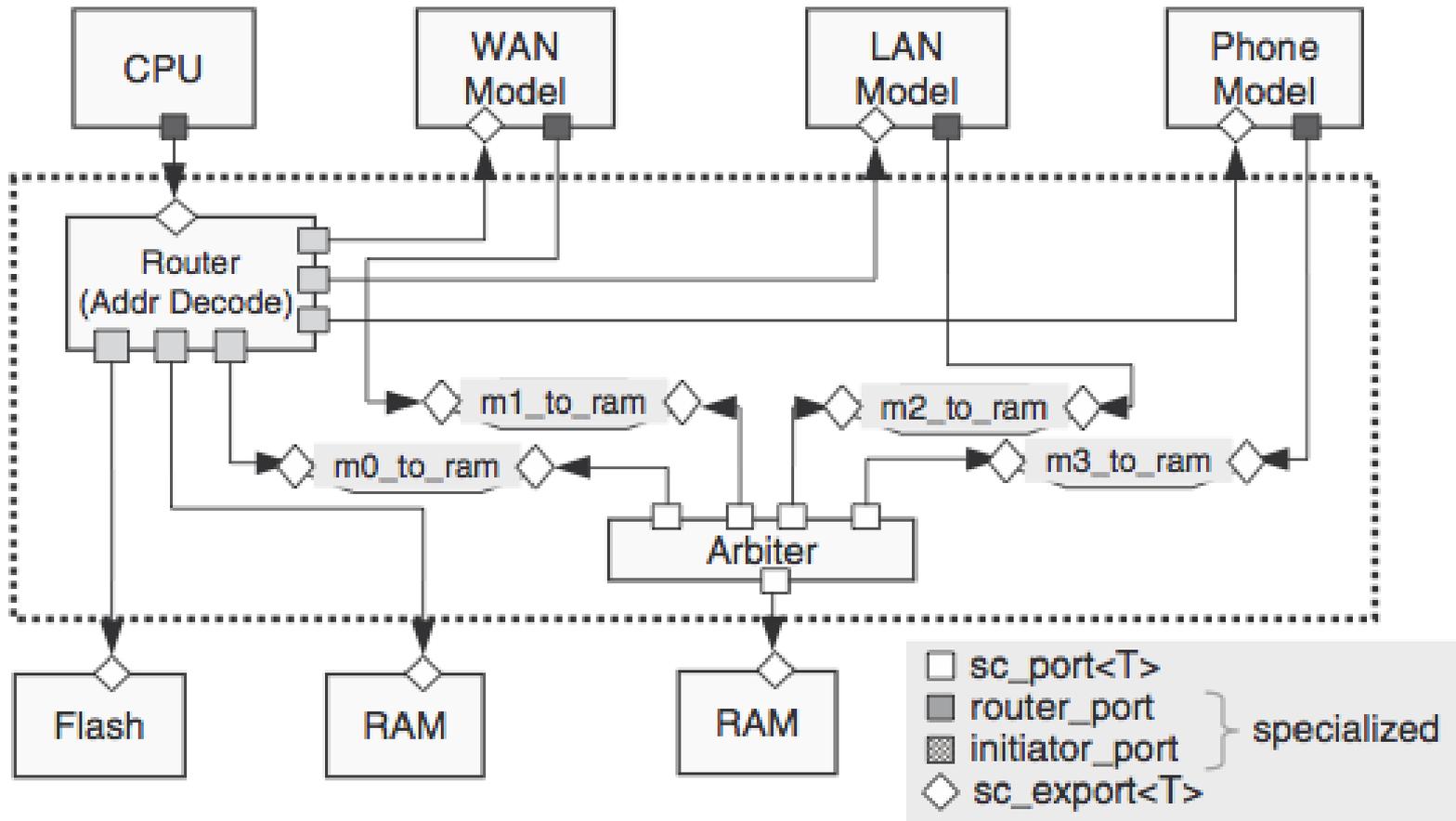
- Entsprechend ist der Slave mit dem **get request** FIFO interface und dem **put response** interface verbunden
- Zudem existiert noch der **tlm\_transport\_channel<Req,Rsp>**.
  - Intern wird dieser channel implementiert durch einen **tlm\_req\_rsp\_channel<Req,Rsp>** mit einer FIFO-Größe von 1
  - Der Master ist angebunden über ein **tlm\_transport\_if<Req,Rsp>sc\_export<T>**,
  - Der Slave über die unidirektionalen Interfaces angebunden

# TLM: Beispiel

---

- Entwurf eines realitätsnahen Systems
  - Erste Frage beim Entwurf auf Systemebene: “Was ist der Zweck des Modells?”
  - TLM-Modelle können vielseitig verwendet werden
    - Hilfe bei Architekturerkundung
    - Verifikation der Systemperformance
    - Unterstützung bei frühen Stufen der Softwareentwicklung
    - Hilfestellung bei der funktionalen Verifikation
- Das folgende Beispiel zeigt ein abstraktes high-level Systemmodell, das für Architekturerkundung und Softwareentwicklung verwendet werden kann

# TLM: Beispiel VoIP-System



# TLM: Beispiel VoIP-System

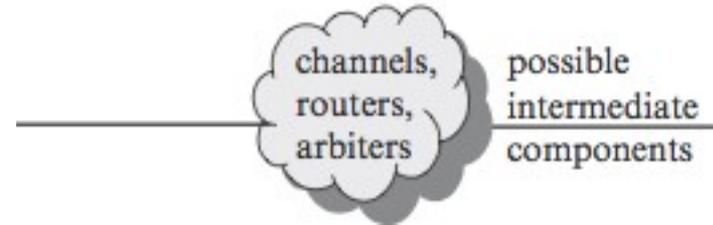
---

- System-Blockdiagramm für einen Voice-Over-IP (VoIP)-Entwurf
- Systemmodell simuliert ein Heimnetzwerk mit mehreren PCs und Telefonen
- Einsatz vieler typischer SoC-Komponenten
  - Prozessor, Flash-Speicher,
  - RAM, I/O-Geräte
- Alle Systemkomponenten sind mit TLM verbunden
- WAN-Modell erzeugt einkommende Ethernet-Daten,
- LAN-Modell erzeugt ausgehende Ethernet-Daten,
- Telefonmodell erzeugt ein/ausgehende Telefondaten
- **Ziel:** Simulation eines 20-Sekunden Telefonanrufs bei gleichzeitigem Netzwerkverkehr

# TLM: Beispiel VoIP-System

```
class master:public sc_module{
public:
    sc_port<...> init_port;
    mem_write(...) {
        rsp = init_port-
>transport(req);
    }
};
```

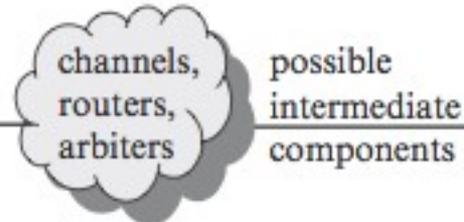
- Target bearbeitet Initiator-Anfrage mit **tlm\_transport\_if<Req,Rsp>**
- SystemC ruft zuerst Prozess-initiator auf, um Speicher-schreibzugriff zu erzeugen
- Initiator ruft **transport()**-Funktion über TLM Interfaceport **init\_port** auf



```
class slave : public sc_module,
    public virtual tlm_transport_if<...>
{
public:
    sc_export<...>target_port;
    ...
    // implemented TLM API
    basic_response<DATA> transport(...) {
        switch(req) {
            case WRITE:
                mem_array[req.ADDR] =
req.DATA;
                wait(write_delay);
            ...
        }
    }
private:
    unsigned mem_array[1024*1024];
};
```

# TLM: Beispiel VoIP-System

```
class master:public sc_module{
public:
  sc_port<...> init_port;
  mem_write(...) {
    rsp = init_port-
>transport(req);
  }
};
```



- Anfrage läuft evtl. durch andere Komponenten, bevor sie beim Speicher ankommt
- Ziel “empfängt” Schreibanfrage
  - Als Funktionsaufruf seiner **transport()**-Funktion
  - Bearbeitet Anfrage entsprechend
  - **tlm\_transport\_if<Req,Rsp>** ist ein blockierendes Interface

```
class slave : public sc_module,
  public virtual tlm_transport_if<...>
{
public:
  sc_export<...>target_port;
  ...
  // implemented TLM API
  basic_response<DATA> transport(...) {
    switch(req) {
      case WRITE:
        mem_array[req.ADDR] =
req.DATA;
        wait(write_delay);
        ...
    }
  }
private:
  unsigned mem_array[1024*1024];
};
```

# TLM: Beispiel VoIP-System

---

- Mit diesem Systemmodell lassen sich verschiedene Eigenschaften des Entwurfs validieren und analysieren:
  - Busparameter
  - Single-cycle vs. Blocktransfers vs. Busbreiten
  - Blockdiagramm—Identifikation aller benötigten Blöcke und I/O-Komponenten
  - HW/SW-Partitionierung
  - Speicherpartitionierung und –performance—  
Trennung von Instruktions- und Datenspeicher
  - Speicherzugriffskonflikte bei Benutzung der Arbitrierung  
Größe der Warteschlangen für Übertragungen von und zum Speicher
  - HW/SW-Schnittstelle

# Zusammenfassung

---

- Berechnungs- und Kommunikationsobjekte von TLM werden durch abstrakte Datentypen verbunden
- TLM ermöglicht unabhängige Modellierung jeder Komponente auf unterschiedlichen Abstraktionsebenen
  - Basis für Verfeinerung von Entwürfen
- Größte Herausforderung: Definition einer notwendigen und hinreichenden Menge von Modellen für einen Entwurfsfluss

# Literatur

---

- F. Ghenassia (Ed.):  
*Transaction-Level Modeling with SystemC – TLM Concepts and Applications for Embedded Systems*  
Springer, 2005, ISBN 978-0-387-26232-1
- <http://www.doulos.com/knowhow/systemc/tlm2/>
- Daniel Gajski, Lukai Cai:  
Transaction Level Modeling: An Overview  
Proc. of CODES+ISSS'03