“Architektur--beschreibungs sprachen” (ADL)

Vortrag im Seminar

Model-based Software Engineering

von Dominik Schindler

am 24.1.2006

Übersicht

1. Einleitung

1. Einleitung

- Moderne Softwaresysteme werden zunehmend größer und komplexer
- Durch eine Software-Architektur wird versucht, diese Komplexität zu bewältigen
- **Definition „Software-Architektur“**
  - Es gibt viele unterschiedliche Definitionen; eine einheitliche Definition existiert noch nicht
  - **Definition nach Balzert:** *„Eine Software-Architektur ist eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen.“ [Balz03]*

1.1. Einleitung

- **Definition nach Bass, Clements und Kazman:** *„The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them.” [BCK98]*

- **Allgemein:** Software-Architektur ist die Beschreibung der Komponenten eines Systems und deren Beziehungen (beispielsweise Kommunikation) untereinander.

- **Definition „System“:** Ein System ist eine Instanz einer Architektur. Eine Architektur dient also als Vorlage für ein System.

1.1. Einleitung

- **Grundlegende Elemente eines Systems/Architektur**
  - **Komponente/-ntyp**: Stellt eine grundlegende „Rechen“-Einheit dar, z.B. Datenspeicher, Klienten, Server, Filter, Datenbanken, usw.
  - **Konnektor/-typ**: Beschreibt eine Verbindung zwischen Komponenten; kann ein eigenes Verhalten (= Protokoll) aufweisen, z.B. Pipes, Bus, usw.
  - **Port/Komponentenschnittstelle**: Ist ein Punkt, an dem die Interaktionen zwischen einer Komponente und der Außenwelt stattfinden; diese Interaktionen werden durch Schnittstellen spezifiziert
  - **Rolle/Konnektorschnittstelle**: Stellt einen Punkt dar, an dem die Interaktionen zwischen einem Konnektor und einer Komponente stattfinden; eine Konnektor-Rolle beschreibt, wie sich die verbundenen Komponenten zu verhalten haben

1.2. Konformitätskriterien

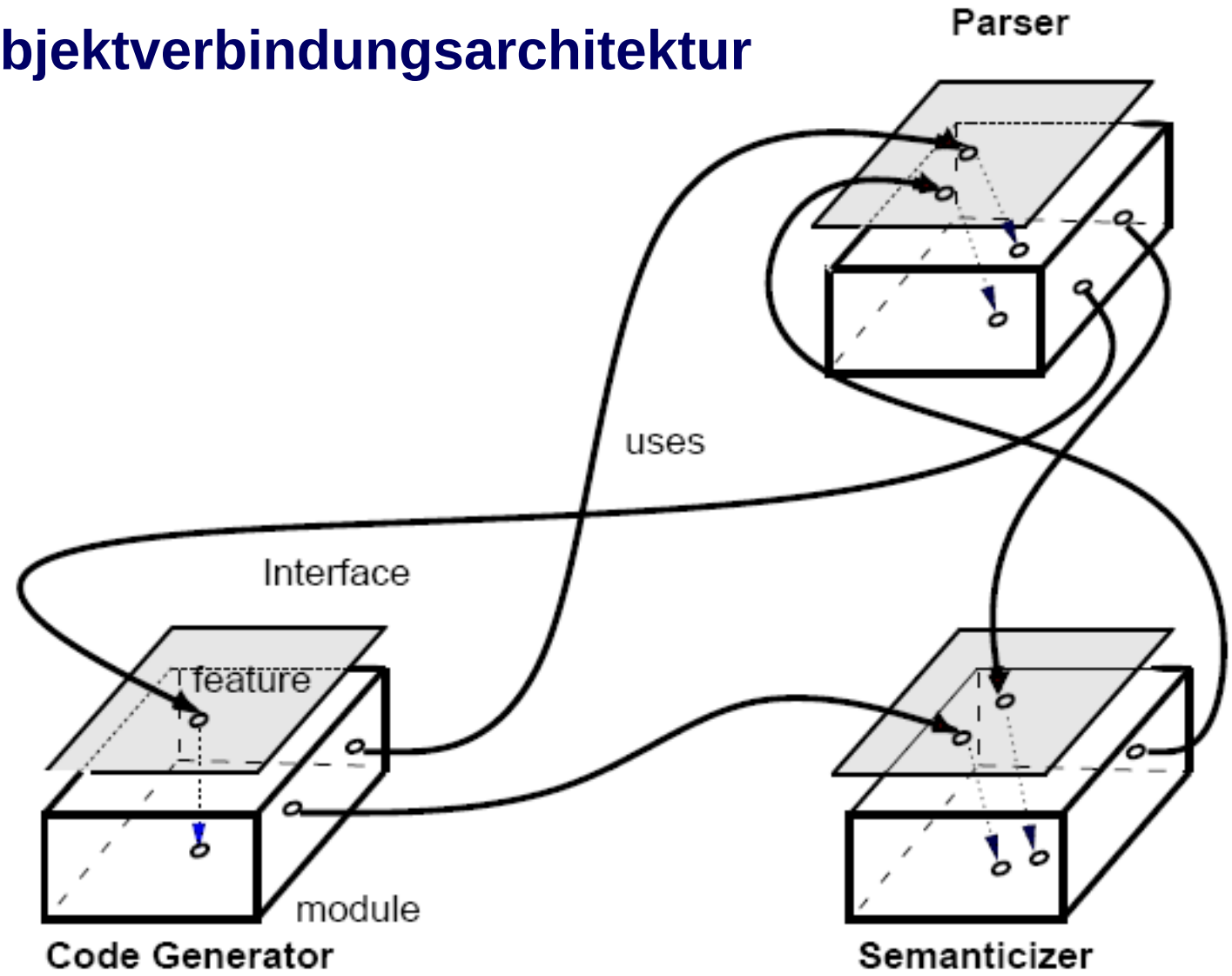
- Um festzustellen ob ein System eine bestimmte Architektur hat, existieren die folgenden Konformitätskriterien:
  - **Dekomposition:** Für jede Schnittstelle in der Architektur gibt es genau eine entsprechende Komponente im System (z.B. eine Komponente, die diese Schnittstelle implementiert).
  - **Schnittstellenkonformität:** Jede Komponente des Systems ist konform zu seiner Schnittstelle. In dieser Schnittstelle können Bedingungen definiert werden, die das Verhalten genauer spezifizieren.
  - **Kommunikationsintegrität:** Die Komponenten des Systems interagieren nur so, wie es in der System-Architektur spezifiziert ist.

2. Architekturkonzepte

Objektverbindungsarchitektur,
Schnittstellenverbindungsarchitektur

2. Architekturkonzepte

1. Objektverbindungsarchitektur



2. Architekturkonzepte

- **Nachteile:**

  1. Architektur nur deskriptiv, da Module vor der Architektur definiert werden müssen

  2. Bei einer Änderung an einer Schnittstelle muss jedes Modul inspiziert werden

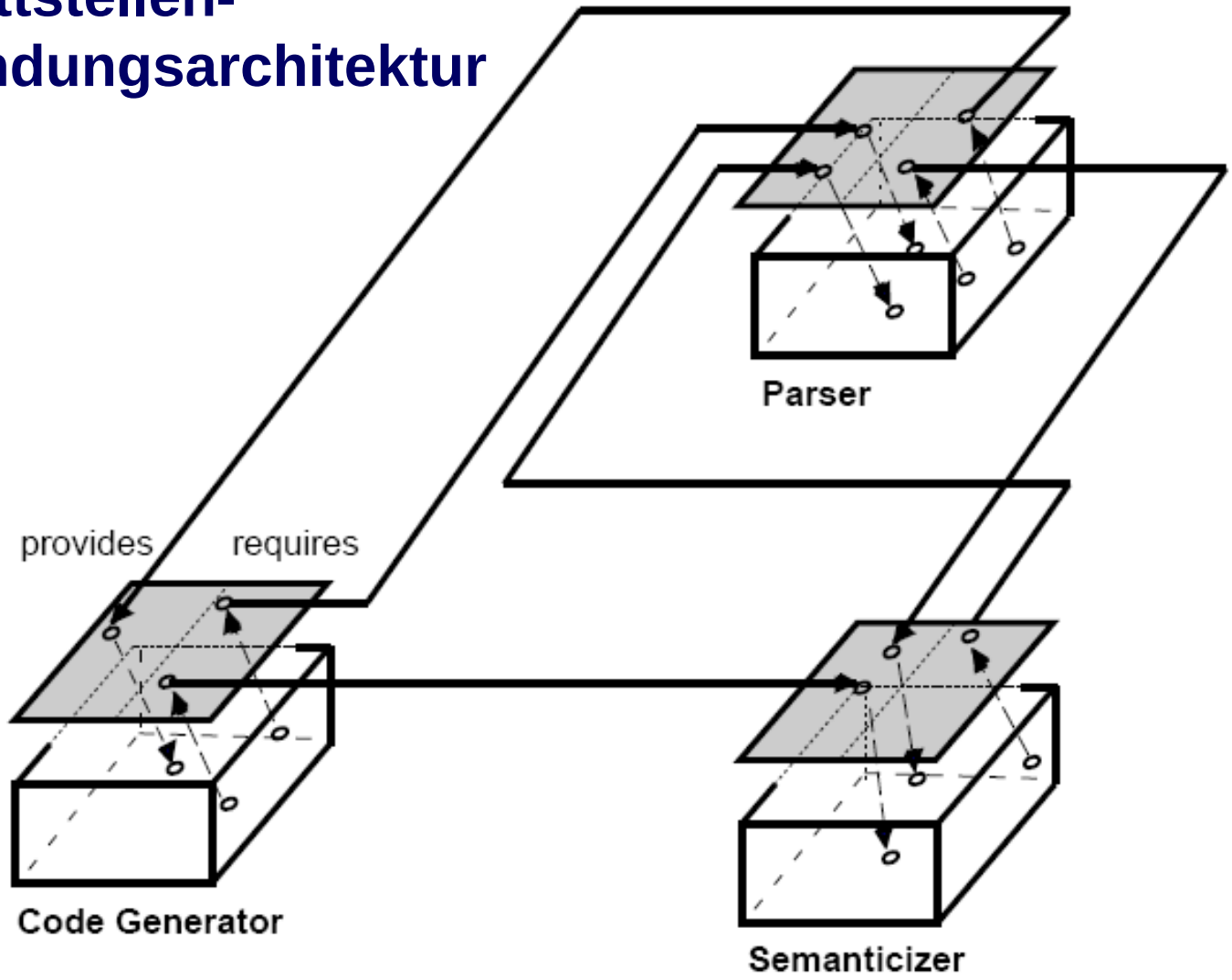
  3. Module mit gleicher Schnittstelle können zwar ausgetauscht werden, aber ein ganz anderes Verhalten aufweisen

- Architektur typisch für objekt-orientierte Programmiersprachen

- **Beispiel (C++):** Feature = Methode, Schnittstelle = öffentlicher Teil einer Klasse, Verbindung = Aufruf einer Methode

2. Architekturkonzepte

2. Schnittstellen-verbindungsarchitektur



1.3 Architekturkonzepte

- ADLs für diese Architektur müssen zwei Mechanismen unterstützen

1. Mechanismus, um Verbindungen zwischen benötigten und breitgestellten Features in den Schnittstellen definieren zu können (Kanten zwischen den Features)

2. Mechanismus, um einem Modul das Benutzen von Features der eigenen Schnittstelle zu erlauben (gestrichelte Kante zwischen Modul und einem Feature seiner Schnittstelle)

- Der Aufruf eines nicht lokalen Features führt vom aufrufenden Modul über dessen Schnittstelle und der Schnittstelle des bereitstellen Moduls zur Implementierung.

1.3 Architekturkonzepte

- Kann zum Planen des Systems verwendet werden
- Ursprünglich entworfen für die Entwicklung von Kommunikationsprotokollen
- Wird noch von wenigen Programmiersprachen unterstützt
- Alle hier vorgestellten ADLs verwenden diese Software-Architektur

3. Architekturbeschreibungssprachen

3. Architekturbeschreibungssprachen

- **Definition nach [Koch04]:**

  „Eine Architektur-Beschreibungssprache beschreibt die Struktur eines Systems auf einer hohen Abstraktionsebene um sie mit formalen Methoden quantitativ oder qualitativ zu untersuchen, sowie zu simulieren oder Code zu erzeugen, und so Aussagen über ein existierendes, oder Vorhersagen über zu bauende Systeme, zu liefern. Dabei werden insbesondere die Bausteine/Subsysteme, ihr Verbindungen und ihr Verhalten betrachtet.“

- **Motivation:** Üblich ist die Darstellung einer Architektur im SWE durch Kästchen und Pfeile
  - **Problem:** Keine Beschreibung der Semantik (z.B. „Was beutet der Pfeil“) oder dem Verhalten (z.B. „Was macht der Kasten“)
  - Informelle Beschreibung der Semantik durch die natürliche Sprache kann zu Mehrdeutigkeiten führen

3. Architekturbeschreibungssprachen

Vorteile von ADLs:

- Keine Mehrdeutigkeiten möglich
- Sie besitzen eine wohldefinierte Semantik und Syntax
- Sind vom Mensch und Maschine gleichermaßen lesbar
- Ermöglichen die Analyse auf Vollständigkeit, Konsistenz, Mehrdeutigkeit und der Performanz von Software-Architekturen
- Sie können evtl. automatisch ein Quellcode-Framework aus Software-Systemen generieren

3.2. ADL Rapide

3.2. ADL Rapide

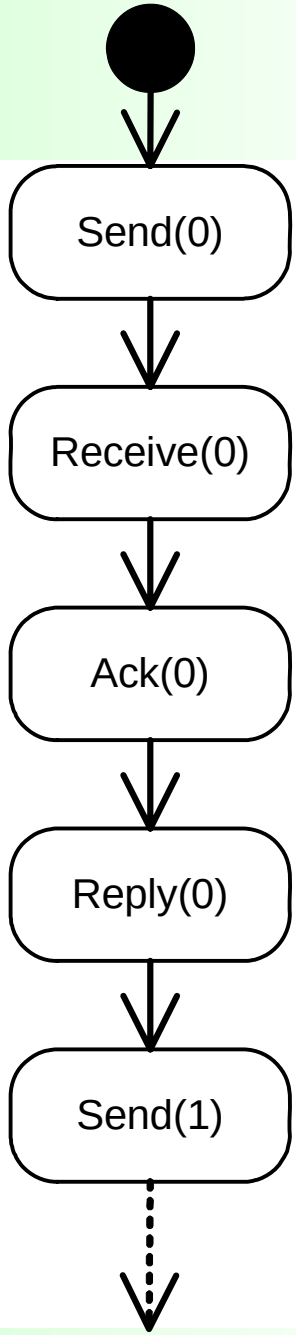
- Entwickelt von Dr. David Luckham an der Universität Stanford
- Rapide ist eine ereignisbasierte, nebenläufige, objekt-orientierte Simulationssprache
- Wurde speziell zum Bau von Prototypen verteilter Systeme entwickelt
- Framework, das aus fünf Teilsprachen besteht [vgl. LKAVBM95]:
  - die „**Type Language**“ zum Beschreiben der Komponentenschnittstellen
  - die „**Architecture Language**“, um den Ereignisfluss zwischen den Komponenten zu modellieren
  - die „Specification Language“, um Bedingungen/Einschränkungen für das Komponentenverhalten zu formulieren
  - die „Execution Language“ zum Schreiben von ausführbaren Modulen für die Simulation
  - die „Pattern Language“ zur Beschreibung von Ereignismustern

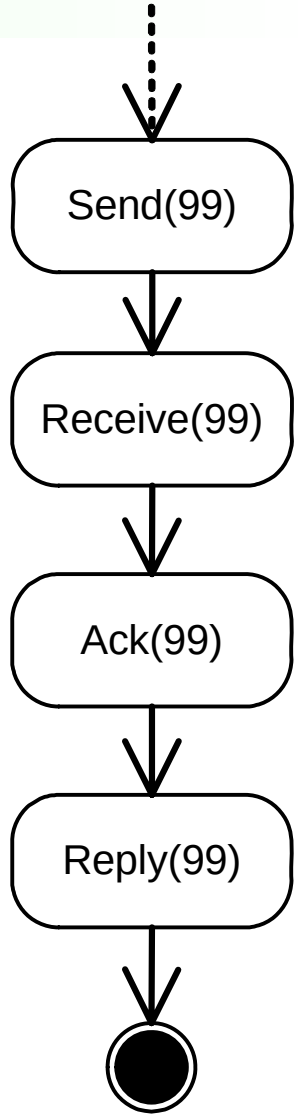
3.2. ADL Rapide

```
type Producer(Max: Positive) is interface
   action out Send(N: Integer);
   action in Reply(N: Integer);
behavior
   Start => Send (0);
   (?X in Integer) Reply(?X) where ?X < Max => Send(?X+1);
end Producer;

type Consumer is interface
   action in Receive(N :Integer);
   action out Ack(N : Integer);
behavior
   (?X in Integer) Receive(?X) => Ack(?X);
end Consumer;

architecture ProdCon() return SomeType is
   Prod: Producer(100);
   Cons: Consumer;
connect
   (?n in Integer)
   Prod.Send(?n) => Cons.Receive(?n);
   Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCon;
```

3.2. ADL Rapide

- Das interessanteste Feature von Rapide ist, dass die Architektur bereits im Vorfeld simuliert werden kann
- Das Ergebnis einer solchen Simulation ist eine Menge von Ereignissen zusammen mit den Abhängigkeiten und dem Zeitpunkt
- „posets“ („partially ordered sets of events“)





3.3. ADL Wright

3.3. ADL Wright

- Wright ist eine universelle Architekturbeschreibungssprache die von Robert Allen an der Carnegie Mellon University entwickelt wurde
- Beschreibt die Komponenten und Verbindungen einer Architektur, sowie deren Verhalten in einem erweiterten CSP Dialekt
- Zentrale Begriffe sind Komponente, Port, Konnektor/Rolle und „Glue“ (= Protokoll)
- **Anmerkung:** CSP („Communicating Sequential Processes“) ist eine von Tony Hoare an der Universität Oxford entwickelte Prozessalgebra zur Beschreibung von parallelen Prozessen die miteinander kommunizieren können.

3.3. ADL Wright

```
System SimpleExample
   component Server =
        port provide [provide protocol]
        spec [Server specification]
   component Client =
        port request [request protocol]
        spec [Client specification]
   connector C-S-connector =
        role Client = (request!x → result?y → Client) ◊ §
        role Server = (invoke?x → return!y → Server) □ §
        glue = (Client.request?x → Server.invoke!x → Server.return?y
                → Client.result!y → glue) □ §
   instances
        s: Server
        c: Client
        cs: C-S-connector
   attachments
        s.provide as cs.server
        c.request as cs.client
end SimpleExample
```

3.3. ADL Wright

Spezifikation des Verhaltens mittels CSP (1)

- **`e → P`**

  Beschreibt den Prozess, der auf das Ereignis e wartet und dann in den Prozess P übergeht. Ist das Ereignis unterstrichen (e) bedeutet das, dass der Prozess das Ereignis selbst erzeugt

- **`e?x, e!y`**

  Sind Ereignisse, die ein Datum x tragen. „?“ beschreibt ein Eingabedatum, „!“ beschreibt ein Ausgabedatum

- **`P □ Q`**

  Externe Auswahl – beschreibt den Prozess, der sich entweder wie P oder wie Q verhält; die Auswahl erfolgt extern durch die Umgebung

3.3. ADL Wright

Spezifikation des Verhaltens mittels CSP (2)

- **P ◊ Q**

  Interne Auswahl – s.o., nur wird die Wahl intern vom Prozess selbst getroffen

- **§**

  Erfolgreiche Terminierung – bezeichnet das erfolgreiche Ende eines Prozesses

- **P = e → P ◊ §**

  Rekursiver Prozess, der eine Folge von Ereignissen e erzeugt, bis er irgendwann selbst entscheidet, erfolgreich zu terminieren

3.4. ADL UML

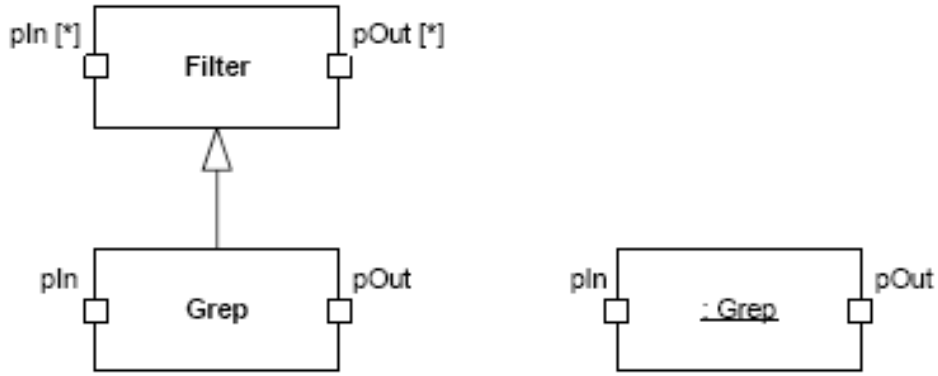
3.4. ADL UML

- Die Unified Modeling Language (UML) ist eine von der OMG entwickelte, universelle, standardisierte, semi-formale Sprache zur Modellierung von Software und anderen Systemen

- In der UML gibt es mehrere Möglichkeiten, wie die Architekturelemente „Komponententyp“ und „Konnektortyp“ spezifiziert werden können.

- Die Spezifikation des Verhaltens (z.B. das Verbindungs-protokoll) kann durch UML Zustandsdiagramme oder Aktivitätsdiagramme erfolgen.

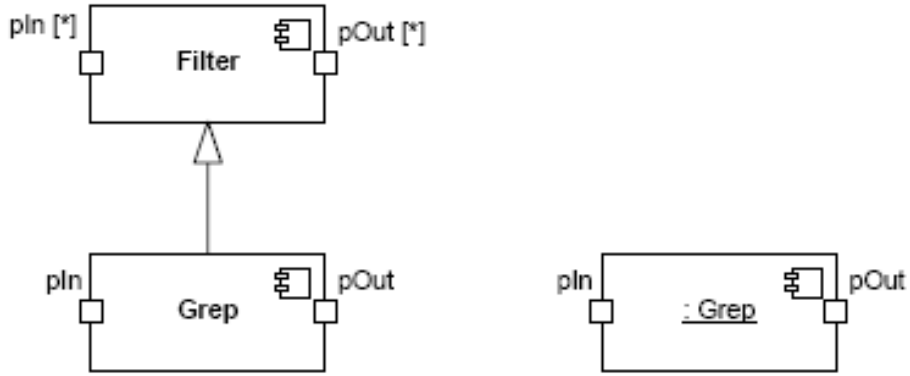
3.4. ADL UML



Komponententyp (1)

- Darstellung mittels einer Klasse und Ports
- Zweckmäßig, da das Verhältnis Klasse/Objekt genau dem Verhältnis Komponententyp/–instanz entspricht
- Das Verhalten kann durch UML Zustandsdiagramme oder Aktivitätsdiagramme modelliert werden
- Das Konzept der Generalisierung/Spezialisierung kann benutzt werden, um Komponententypen in Beziehung zu setzen
- Kann bei UML-basierten Tools verwendet werden
- **Problem:** Werden Konnektoren ebenfalls durch Klassen dargestellt, besteht Verwechslungsgefahr

3.4. ADL UML



Komponententyp (2)

- Darstellung mittels UML 2.0 Komponente und Ports
- In der UML 2.0 ist jetzt eine Komponente ein Subtyp einer Klasse, d.h., sie hat die gleichen Eigenschaften wie eine Klasse
- Eine Komponente kann im Gegensatz zu einer Klasse weitere Unterelemente enthalten
- Werden Konnektoren mittels Klassen dargestellt, besteht keine Verwechslungsgefahr mehr
- **Problem:** Diese Art der Darstellung muss von einem Tool explizit unterstützt werden.

3.4. ADL UML

Port

- Ports sind neue Elemente der UML 2.0
- Stellt einen Punkt dar, über den mit der Außenwelt (also mit anderen Komponenten bzw. Konnektoren) interagiert werden kann
- Ein Port kann mit mehreren bereitstellenden und benötigten Schnittstellen verbunden werden
- Diese Schnittstellen beschreiben die Interaktionen der Komponente, die an diesem Port möglich sind
- **Es gilt:** Damit Komponenten austauschbar sind, muss die gesamte Kommunikation über Ports erfolgen.

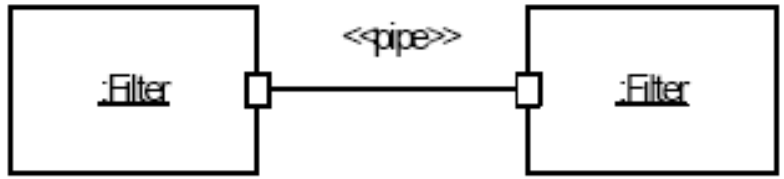
3.4. ADL UML

Konnektortyp (1)

- Der in der UML 2.0 integrierte Konnektortyp hat zu wenig Ausdruckskraft, um einen Konnektor im Sinne einer Architektur zu beschreiben. [ICGNSS04]
- So können semantischen Informationen und/oder eine Beschreibung des Verhaltens nicht mit einem Konnektor verbunden werden.
- Außerdem werden vom integrierten Konnektor keine Rollen unterstützt

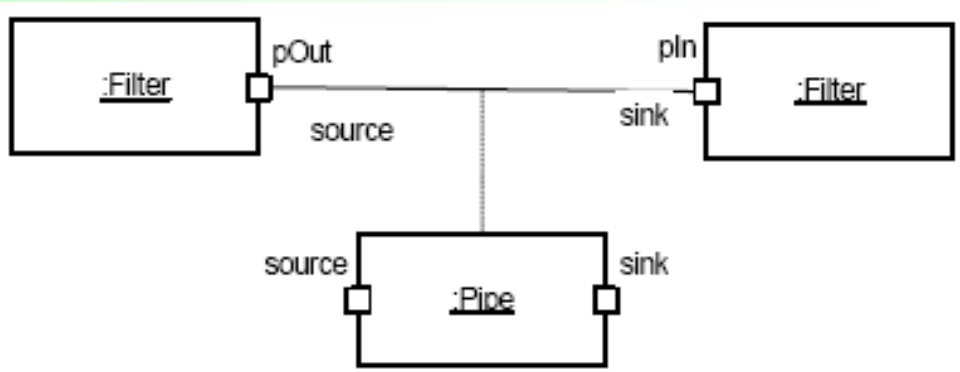
3.4. ADL UML

Konnektortyp (2)



- Darstellung mittels einer Assoziation
- Durch einen Stereotyp (hier `<<Pipe>>`) kann die Art der Verbindung näher spezifiziert werden
- Die einfachste Art der Darstellung, die keine genauere Spezifikation erlaubt
- **Problem 1:** Falls ein Konnektor ein Protokoll zwischen den Komponenten beschreibt und/oder selbst eine Semantik hat, ist dieser Darstellung nicht mehr ausreichend
- **Problem 2:** Rollen können nicht dargestellt werden

3.4. ADL UML



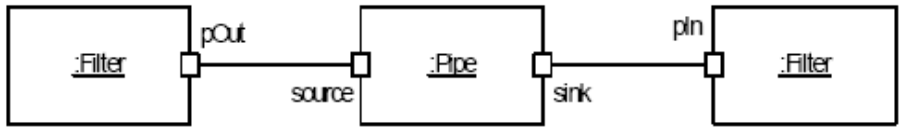
Konnektortyp (3)

- Darstellung mittels einer Assoziationsklasse
- Erlaubt zusätzlich die Semantik und das Verhalten einer Assoziation zu beschreiben
- Eine Rolle kann durch einen Port dargestellt werden
- Die Zuordnung einer Rolle zu einem Port wird durch einen gleichen Bezeichner erreicht
- **Problem:** Zuordnung zwischen Rolle und Port wird bei größeren Systemen unübersichtlich

3.4. ADL UML

Konnektortyp (3)

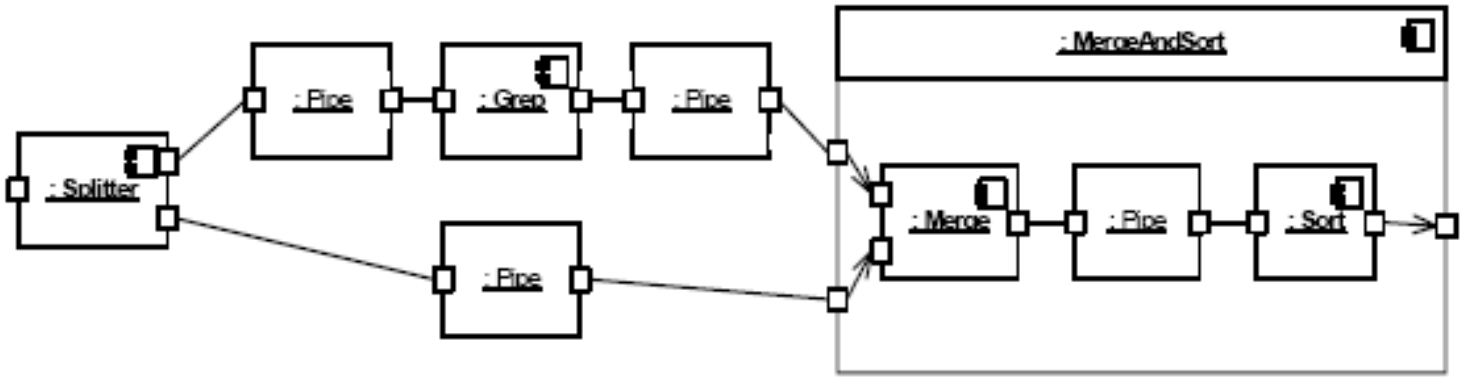
- Darstellung eines Konnektors mittels einer Klasse
- Bei größeren System ist diese Darstellung übersichtlicher als mittels Assoziationsklassen und ausdrucksstärker als mittel Assoziationen
- Hat die gleichen Eigenschaften wie eine Assoziationsklasse
- **Problem:** Bei der Darstellung eines Konnektors und einer Komponente mittels einer Klasse besteht Verwechslungsgefahr



3.4. ADL UML

Beispielsystem

- Komponenten werden durch UML Komponenten dargestellt und Konnektoren durch Klassen
- Das UML 2.0 Konzept der Kompositionsstruktur erlaubt es, dass eine Komponente (`MergeAndSort`) weitere Elemente beinhalten kann
- Außerdem wird gezeigt, wie die Ports der Komponente mit den Ports der Unterelemente (den sogenannten Parts) durch Delegationskonnektoren verbunden sind



4. Bewertung und Fazit

4. Bewertung und Fazit

Rapide

- Erzeugt einen Prototypen der Architektur, der simuliert werden kann
- Es existieren viele Simulationswerkzeuge für Rapide
- Kann Quelltext-Frameworks erstellen
- **Problem:** Sprache muss erlernt werden
- Rapide besitzt nicht nur akademischen Charakter, so wurde die Architektur des SPARC-V9-Prozessors in Rapide realisiert [IntStan].

4. Bewertung und Fazit

UML

- Es gibt verschiedene Strategien, wie Architekturen beschrieben werden können
- Besonderheit der UML ist das Einsatzdiagramm
  - Zeigt die Hardware-Konfiguration, auf der das laufende Zielsystem zum Einsatz kommt
  - Stellt die Verteilung der (Software)-Komponenten auf den einzelnen Knoten (= eine physische Speicher- und Verarbeitungseinheit) der (Hardware)-Konfiguration dar
- **Gut:** UML-Diagramme sind intuitiv und damit leicht zu verstehen
- In der UML 2.0 hat sich gegenüber seinen Vorgängern in Richtung ADL viel getan. Die Einführung der strukturierten Klassifizierer zur Darstellung von Hierarchien und Ports als Interaktionspunkt hat die Situation wesentlich verbessert.

4. Bewertung und Fazit

Wright

- Werkzeuge für CSP können verwendet werden
- Momentan können die wenigen Werkzeuge kein Quelltext-Framework erstellen
- **Problem:** Sprache muss erlernt werden

- Bis jetzt existiert noch keine Standard-Architektur-beschreibungssprache
- **Aber:** Es kristallisiert sich die UML langsam als ADL-Standard heraus. [IntWiki]

5. Literaturverzeichnis

[All97] R. Allen, „A Formal Approach to Software Architecture“, 1997

[AllGar98] Robert Allen, David Garlan, „A Formal Basis for Architectural Connection“, 1998

[Balz01] Helmut Balzert, „Lehrbuch der Software-Technik“, 2. Auflage, 2001

[BCK98] Len Bass, Paul Clements, Rick Kazman, „Software Architecture in Practice”, 1998

[Cook99] Tw Cook, „Architecture Description Languages: An Overview“,

[Hoare04] C. A. R. Hoare, „Communicating Sequential Processes“, 2004

[ICGNSS04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, Jaime Rodrigo Oviedo Silva, „Documenting Component and Connector Views with UML 2.0“, 2004

[IntStan] Alexandre Santoro, Woosang Park, David Luckham, „SPARC-V9 Architecture Specification With Rapide“, Stand 15.1.2006

[IntWiki] http://www.wikipedia.de, Stand 16.1.2006

[LKAVBM95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, Walter Mann, „Specification and Analysis of System Architecture Using Rapide“, 1995

[LVM95] David C. Luckham, James Vera, Siguard Meldal, „Three Concepts of System Architecture“, 1995

[KSW04] Koch, Störrle, Wirsing, „Methoden des Software Engineering“, 2004