„Architektur-beschreibungs-sprachen“ (ADL)

Ausarbeitung zum
Vortrag im Seminar

„Model-based Software Engineering“

im WS 2005/06
am 24. Januar 2006
von Dominik Schindler

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Moderne Softwaresysteme werden zunehmend größer und komplexer. Durch Abstraktion und die Verwendung einer Software-Architektur wird versucht, diese Komplexität zu bewältigen.

1.1. Definition: Software-Architektur

Da eine einheitliche Definition für den Begriff „Software-Architektur“ noch nicht gefunden wurde, werden hier zwei Definitionen exemplarisch wiedergegeben.

Definition nach Balzert: *„Eine Software-Architektur ist eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen.“ [Balz03]*

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 versteht man unter Software-Architektur die Beschreibung der Struktur eines Systems durch Komponenten und deren Beziehungen (beispielweise Kommunikation) untereinander. Dazu existieren die folgenden grundlegenden Elemente. [ICGNSS04]

- **Komponente**: Stellt eine grundlegende „Rechen“-Einheit dar, z.B. Datenspeicher, Klienten, Server, Filter, Datenbanken, usw.
- **Konnektor**: 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**: Ist ein Punkt, an dem die Interaktionen zwischen einem Konnektor und einer Komponente stattfinden; außerdem beschreibt eine Rolle, wie sich die verbundenen Komponenten zu verhalten haben

Hinweis: In dieser Ausarbeitung wird im Gegensatz zu anderen Veröffentlichungen explizit zwischen einem Typ und einer Instanz des Typs unterschieden (z.B. Komponententyp/Komponente).

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

1.2. Konformitätskriterien

Um festzustellen ob ein System eine bestimmte Architektur hat, existieren die folgenden drei Konformitätskriterien [vgl. LVM95]:

1.2.1. Dekomposition

Für jede Schnittstelle in einer Architektur gibt es genau eine entsprechende Komponente im System (z.B. eine Komponente, die diese Schnittstelle implementiert).

1.2.2. Schnittstellenkonformität

Jede Komponente des Systems ist konform zu seiner Schnittstelle. In dieser Schnittstelle können außerdem Bedingungen definiert werden, die das Verhalten genauer spezifizieren.

1.2.3. Kommunikationsintegrität

Die Komponenten des Systems interagieren nur so, wie es in der System-Architektur spezifiziert ist.

1.3. Architekturkonzepte

Im Software-Engineering gibt es zwei grundlegende Architekturkonzepte: die Objektverbindungsarchitektur und die Schnittstellenverbindungsarchitektur.

1.3.1. Objektverbindungsarchitektur

Bei der Objektverbindungsarchitektur erfolgt eine Verbindung (vergleichbar mit dem Aufruf einer Methode) von Objekt zu Objekt. Eine solche Software-Architektur ist in Abbildung 1.1 dargestellt. In dieser Abbildung sind die Schnittstellen durch schattierte Flächen dargestellt und die darunterliegenden Boxen repräsentieren Module, die konform zu dieser Schnittstelle sind. Diese Konformität entspricht der in Kapitel 1.2.2 genannten Schnittstellenkonformität.

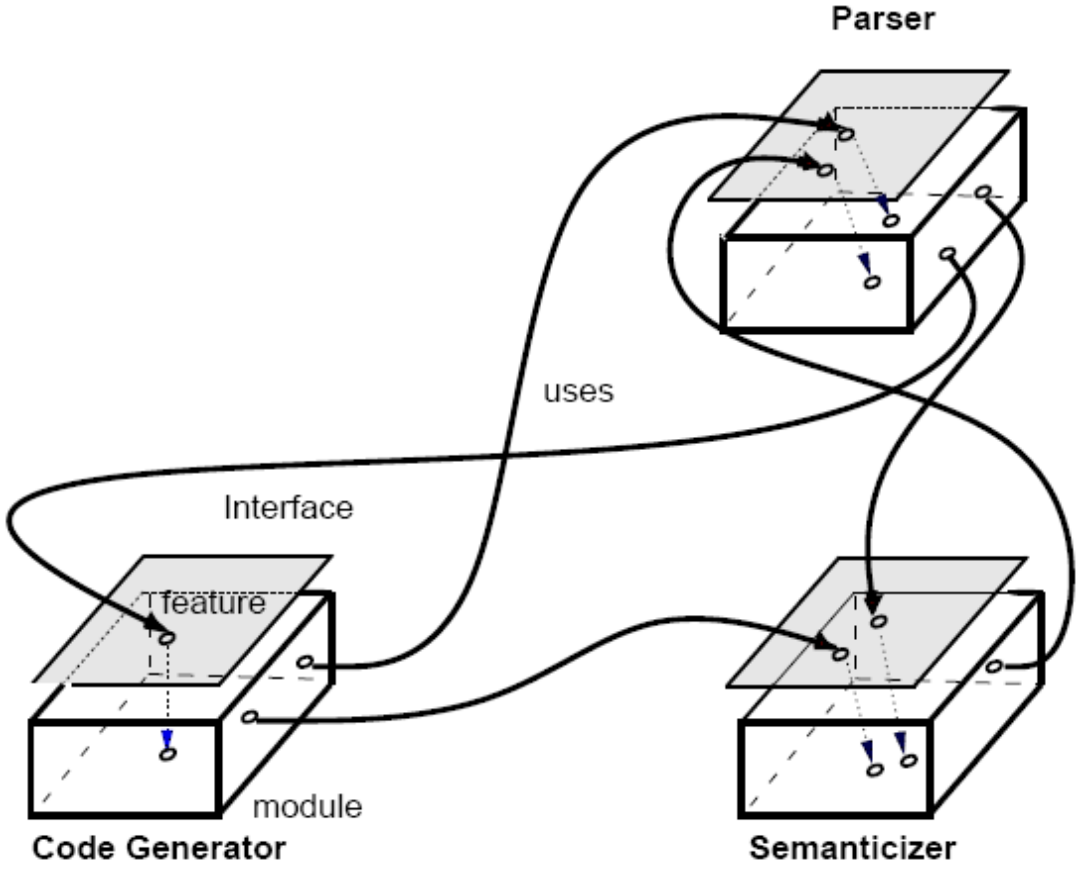

Abb. 1.1: Objektverbindungsarchitektur

Eine Verbindung zwischen einem Modul und einem Feature (vergleichbar mit einer Methode/Funktion) wird als gerichtete Kante vom Modul zum verwendeten Feature in der spezifizierenden Schnittstelle dargestellt. Eine gestrichelte Kante zeigt eine Verbindung zwischen dem Feature, spezifiziert in der Schnittstelle, und der Implementierung im Modul.

Diese Architektur ist typisch für objekt-orientierte Programmiersprachen. Überträgt man beispielsweise die hier genannten Konzepte auf C++, entsprechen die Features den Methoden, Schnittstellen den öffentlichen Teil einer Klasse und Verbindungen den Aufruf einer öffentlichen Methode eines Objekts.

Problem 1: Architektur nur deskriptiv
Da der Aufruf einer Methode von einem Modul aus direkt an ein Objekt gesendet wird, müssen die Module **vor** der Architektur spezifiziert werden. Dies ist notwendig, damit alle in den Modulen existierenden Aufrufe (= Verbindungen) im System gefunden und dargestellt werden können. Das bedeutet aber auch, dass diese Architektur nicht zum Planen des Systems verwendet werden kann sondern rein deskriptiv ist.

Problem 2: Änderungen der Schnittstelle
Ein weiterer Nachteil dieser Architektur zeigt sich, wenn eine Schnittstelle geändert wird. So muss bei einer Änderung einer Schnittstelle jedes Modul inspiziert und diejenigen Module angepasst werden, die diese Schnittstelle verwenden.

Problem 3: Austausch eines Moduls
Das größte Problem dieser Architektur tritt aber auf, wenn Module ausgetauscht werden sollen. So können bei dieser Software-Architektur zwei Module konform zur gleichen Schnittstelle sein, aber ein komplett anderes Verhalten aufweisen. Zum Beispiel wenn das in der Abbildung dargestellte Modul `Code_Generator` am Anfang eine Funktion `Initialize_Parser` des Moduls Parser aufrufen muss, damit das ganze System korrekt funktioniert, das ausgetauschte Modul diese notwendige Initialisierung aber nicht vornimmt, sind schwere Funktionsstörungen vorprogrammiert.

1.3.2. Schnittstellenverbindungsarchitektur

Bei der Schnittstellenverbindungsarchitektur erfolgt eine Verbindung von Schnittstelle zu Schnittselle und nicht wie in Kapitel 1.3.1 von Objekt zu Objekt. Die Abbildung 1.2 zeigt eine solche Architektur. In dieser Abbildung ist ebenfalls zu erkennen, dass die Schnittstelle in zwei Bereiche unterteilt ist: einen Bereich mit den bereitgestellten Features und einen anderen Bereich mit den benötigten Features. Die Features der einzelnen Bereiche sind durch eine gerichtete Kante, jeweils vom benötigten zum bereitgestellten Feature, verbunden.

Eine Architekturbeschreibungssprache, die diese Architektur unterstützt, muss also die folgenden zwei Mechanismen bereitstellen:

- Einen Mechanismus, um Verbindungen zwischen benötigten und bereitgestellten Feature in den Schnittstellen definieren zu können. Dargestellt in Abb. 1.2 als Kante zwischen den Features des jeweiligen Bereichs.

- Einen Mechanismus, um einem Modul das Benutzen eines Features der eigenen Schnittstelle zu erlauben. Dargestellt in Abbildung 1.2 durch eine gestrichelte Kante zwischen einem Modul und einem Feature der Modul-Schnittstelle.

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

Diese Art von Architektur erfüllt alle drei oben genannten Konformitätskriterien. Außerdem besitzt sie nicht die Probleme der Objektverbindungsarchitektur in Kapitel 1.3.1., d.h., dass

diese Architektur zum Planen des Systems verwendet werden kann. Alle hier vorgestellten Architekturbeschreibungssprachen benutzen dieses Architekturkonzept.

Anmerkung: Dieses Konzept wurde ursprünglich für die Entwicklung von Kommunikationsprotokollen entworfen und wird noch von wenigen Programmiersprachen unterstützt.

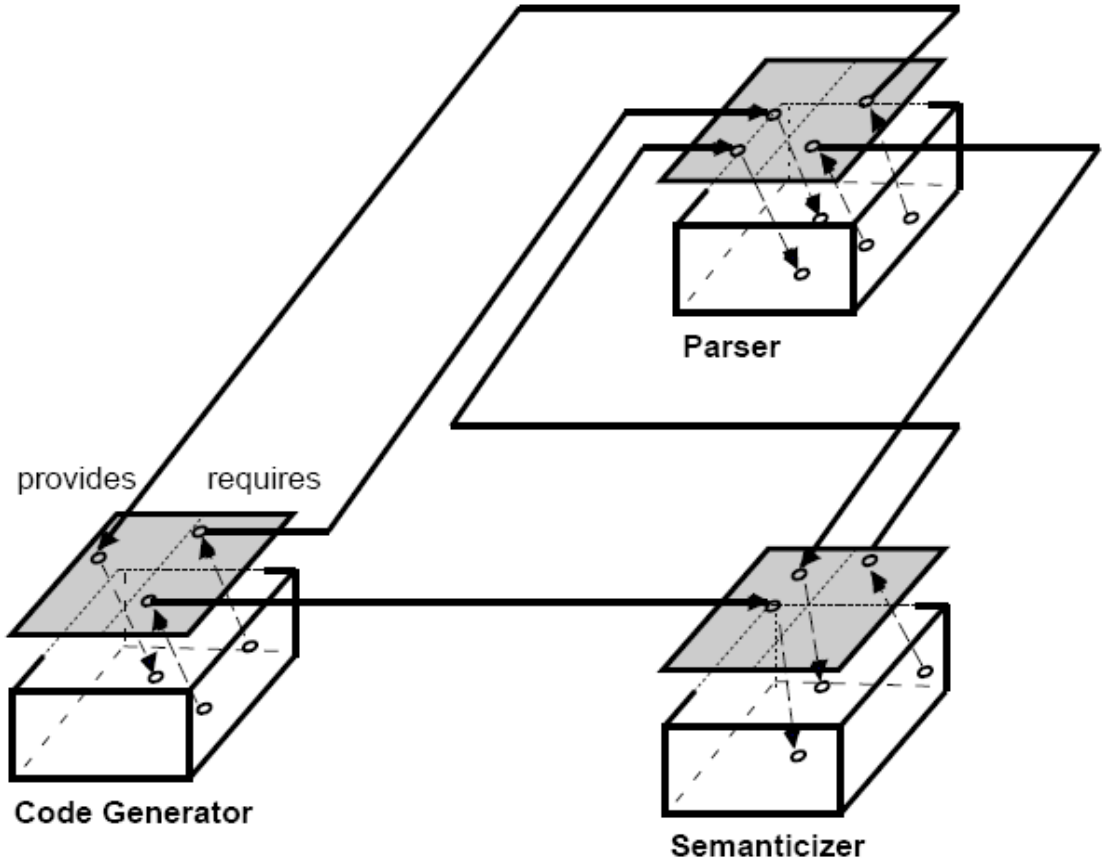

Abb. 1.2: Schnittstellenverbindungsarchitektur

2. Architekturbeschreibungssprachen

„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.“ [Koch04]

2.1. Motivation

Eine Software-Architektur muss in irgendeiner Form dargestellt werden. Gängige Praxis im Software-Engineering ist die informelle Darstellung durch Kästchen und Pfeile. Ein solches Diagramm ist zwar schnell erstellt und schafft einen ersten Überblick über das System, aber bestimmte Fragen der Semantik (z.B. „Was bedeutet der Pfeil“) oder dem Verhalten (z.B. „Was macht der Kasten“) können durch solch ein Diagramm nicht beantwortet werden. Um diese Diagramme näher zu beschreiben kann beispielsweise die natürliche Sprache verwendet werden. Ein Problem der natürlichen Sprache sind die möglicherweise auftretenden Mehrdeutigkeiten, die bei einer formalen Beschreibung nicht vorkommen

können. Außerdem bieten fast alle Architekturbeschreibungssprachen weitere Vorteile [vgl. Cook99]:

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

Im den folgenden Abschnitten werden drei Architekturbeschreibungssprachen vorgestellt.

2.2. ADL Rapide

Rapide ist eine ereignisbasierte, nebenläufige, objekt-orientierte Simulationssprache, die speziell zum Bau von Prototypen verteilter Systeme entwickelt wurde. Rapide wurde von Dr. David Luckham an der Universität Stanford entwickelt und ist ein Framework, das aus fünf Teilsprachen besteht [vgl. LKAVBM95]:

- die „Type Language“ zum Beschreiben der Komponentenschnittstellen (und indirekt der Komponenten)
- die „Architecture Language“, um den Ereignisfluss zwischen den Komponenten zu modellieren
- die „Specification Lanugage“, 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

Im Folgenden wird auf die Typsprache und Architektursprache näher eingegangen. Für einen detaillierteren Einblick in Rapide und weitere Informationen bezüglich der hier nicht erwähnten Sprachen des Frameworks verweise ich auf [LKAVBM95].

Das folgende Abbildung 2.2.1 zeigt eine Reader/Writer-Architektur spezifiziert in 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;
```

Abb. 2.2.1: Reader/Writer-Architektur spezifiziert in Rapide

Typsprache

Im Abbildung 2.2.1 werden zwei Schnittstellentypen `Producer` und `Consumer` spezifiziert, die die Schnittstellen für `Producer`- und `Consumer`-Objekte definieren. Diese Schnittstellentypen besitzen bestimmte Aktionen („actions“) für die asynchrone Kommunikation. Zum Beispiel besitzt die `Producer`-Schnittstelle zwei Aktionen, eine ausgehende Aktion `Send(N: Integer)` und eine eingehende Aktion `Reply(N: Integer)`, die jeweils einen Integer als Parameter erwarten. Die Objekte dieses Schnittstellentyps können also `Send`-Ereignisse produzieren und `Reply`-Ereignisse empfangen.

Die `Producer`-Schnittstelle definiert außerdem Verhaltensregeln im Abschnitt `behavior`. In der ersten Regel wird spezifiziert, dass durch die `Start`-Aktion (die alle Objekte erhalten, wenn sie aktiviert werden) die Ausgabeaktion `Send(0)` ausgelöst wird. Die zweite Regel spezifiziert, dass beim Empfang einer `Reply()`-Aktion eine ausgehende `Send()`-Aktion mit dem nächsten Wert ausgelöst wird, sofern der übergebene Wert `X` kleiner als Max ist.

Die Spezifikation der `Consumer`-Schnittstelle ist ähnlich. Das Verhalten der Schnittstelle ist durch eine Regel definiert die besagt, dass beim Empfang einer `Receive()`-Aktion mit einer `Ack()`-Aktion und dem erhaltenen Wert reagiert wird.

Architektursprache

Die Architektur definiert die Kommunikation zwischen den Komponenten durch das Verbinden der in den Schnittstellen definierten Aktionen. Im Beispiel besitzt die `ProdCon`-Architektur die beiden Komponenten `Prod` vom Typ `Producer` und `Con` vom Typ `Consumer`. Die Verbindung zwischen den beiden Komponenten wird durch zwei reaktive Verbindungsregeln im Abschnitt `connect` definiert. Die erste Regel spezifiziert, dass ein `Cons.Receive(n)` mit dem Wert `n` immer dann ausgelöst wird, wenn die Komponente `Prod` ein `Send(n)` ausgelöst hat. Dadurch wird die `out`-Aktion `Send()` der `Prod` Komponente mit der `in`-Aktion `Receive()` der `Cons` Komponente in Beziehung gebracht. Die zweite Regel verbindet die `out`-Aktion `Ack()` der `Cons` Komponente mit der `in`-Aktion `Reply()` der `Prod` Komponente.

Simulation der Architektur

Eines der interessanten Features von Rapide ist, dass die Architektur im Vorfeld simuliert werden kann. Das Ergebnis einer solchen Simulation ist eine Menge von Ereignissen („events“) zusammen mit den Abhängigkeiten und dem Zeitpunkt. Diese Menge wird in Rapide als „posets“ („partially ordered sets of events“) bezeichnet. Solche „posets“ liefern einen Überblick über das Verhalten der Architektur und es können Fehler wie Deadlocks, Synchronisationsprobleme, usw. erkannt werden. Die Abbildung 2.2.2 zeigt ein „posets“ für die `ProdCon`-Architektur. Dargestellt werden die Abhängigkeiten der Ereignisse `Send`, `Receive`, `Ack` und `Reply`, und der zeitliche Verlauf dieser Ereignisse.

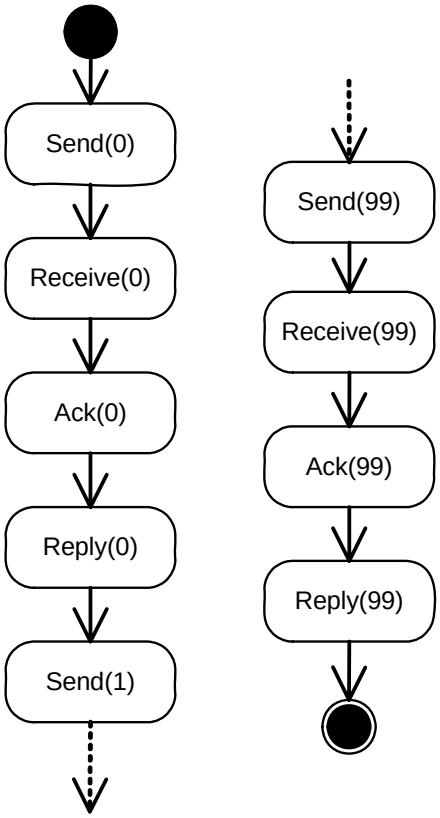

Abb. 2.2.2: „posets“ der ProdCon-Architektur

2.3. ADL Wright

Wright ist eine universelle Architekturbeschreibungssprache die von Robert Allen an der Carnegie Mellon University entwickelt wurde. Sie beschreibt die Komponenten und Verbindungen einer Architektur, sowie deren Verhalten in einem erweiterten CSP Dialekt [vgl. AllGar98]. Die zentralen Begriffe in Wright sind Komponente, Port, Konnektor und Glue, die anhand des folgenden einfachen Beispiels genauer erläutert werden.

Anmerkung: CSP („Communicating Sequential Processes“) ist eine von Tony Hoare an der Universität Oxford entwickelte Prozessalgebra zur Beschreibung von parallelen Prozessen die auch miteinander kommunizieren können.

Die Abbildung 2.3.1 zeigt die Spezifikation einer einfachen Client-Server-Architektur in 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
```

Abb. 2.3.1: Client/Server-Architektur spezifiziert in Wright

Komponente(-ntyp)

Der erste Abschnitt definiert die Komponenten- und Verbindungstypen. Ein Komponententyp besitzt eine Menge von Ports und eine Komponentenspezifikation, die das (abstrakte) Verhalten der Komponente festlegt. In Abbildung 2.3.1 hat die Client und Server-Komponente jeweils einen einzigen Port, allgemein kann aber eine Komponente mehrere Ports besitzen. Solche Ports beschreiben die Schnittstellen zu den anderen Teilen des Systems.

Konnektor(-typ)

Die Komponenten kommunizieren nicht direkt miteinander, sondern indirekt über Konnektoren. Ein solcher Konnektortyp besitzt eine Menge von Rollen und eine Glue-Spezifikation. Eine Rolle beschreibt, wie sich eine Komponente zu verhalten hat, wenn sie an der Interaktion mit dem Konnektor teilnimmt.

Die „Glue“-Spezifikation spezifiziert, wie die Aktivitäten der Client- und Server-Rolle aufeinander abgestimmt werden. Sie stellt sozusagen das Protokoll der beiden Rollen dar. Im Beispiel oben schreibt die „Glue“-Spezifikation vor, dass die Aktivitäten des Clients und Servers wie folgt verzahnt sind: Client fragt Dienst an, Server behandelt die Anfrage, Server stellt Ergebnis zu, Client erhält Ergebnis.

Obwohl der Konnektortyp in diesem Beispiel nur zwei Rollen hat, kann ein Konnektortyp mehrere Rollen definieren.

Instanzen

Der zweite Abschnitt besteht aus einer Menge von Komponenten- und Konnektorinstanzen. Diese Instanzen definieren welche tatsächlichen Einheiten in der Konfiguration existieren. Im Beispiel sind das ein einzelner Server (`s`), ein einzelner Client (`c`) und eine einziger Konnektor (`cs`). Die Anzahl der Instanzen muss aber bei Wright nicht statisch sein. So können weitere Instanzen dynamisch zur Laufzeit erzeugt werden.

Bindungen („attachements“)

Im dritten Abschnitt der Spezifikation werden die Ports der Komponenten mit den Konnektor-Rollen verknüpft. Im Beispiel sind der Client-`request`-Port und der Server-`provide`-Port mit der Client-Rolle bzw. Server-Rolle verbunden, so dass der Konnektor `cs` das Verhalten der beiden Ports `c.request` und `s.provide` koordinieren kann.

Spezifikation des Verhaltens mittels CSP [Hoare04, All97]

Um das Verbindungsprotokoll der Konnektoren zu beschreiben, wird für jede Rolle und jedem Glue ein Prozess in der Sprache CSP spezifiziert. Es folgt eine kurze Einführung in CSP, mit den ansprechenden Anpassungen von Wright:

- **e → P**: Beschreibt den Prozess, der auf das Ereignis („event“) 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 das Datum x bzw. y 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.
- **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.

Die Server-Rolle im Beispiel beschreibt das kommunikative Verhalten des Diensterbringers. Dieses Verhalten ist definiert durch einen rekursiven Prozess, der auf eine Anfrage einen Wert zurückliefert, oder erfolgreich beendet werden kann. Da der externe Auswahloperator gewählt wurde, wird die Entscheidung, welcher der beiden Alternativen gewählt wird, von der Umgebung (=anderen Rolle und Glue) der Rolle getroffen. Das bedeutet also, dass der Server nicht terminieren darf, bis die Umgebung dies erlaubt.

Die Client-Rolle beschreibt das kommunikative Verhalten des Dienstnutzers. Wie beim Server ist das Verhalten ein Prozess, der wiederholt Anforderungen eines Dienstes absetzten kann und dann das Ergebnis erhält, oder erfolgreich terminiert. Die Wahl der Alternativen wird aufgrund des internen Auswahloperators diesmal nicht von der Umgebung, sondern vom Client-Prozess selbst nichtdeterministisch getroffen.

Der „Glue"-Prozess koordiniert das Verhalten der beiden Rollen indem er spezifiziert, wie die Ereignisse der Rollen zusammenarbeiten. Dazu kommuniziert der „Glue" mit den Rollen über die spezifizierten Ereignisse. Im Beispiel erzeugt die Client-Rolle das Ausgabeereignis `request!x` mit dem Datum x, das vom „Glue" empfangen wird (`Client.request?x`). Anschließend löst der „Glue"-Prozess das Ausgabeereignis `Server.invoke?y` aus, das wiederum vom der Server-Rolle (`invoke?y`) empfangen wird, usw.

Anmerkung: Der „Glue" verwendet Eingabeereignisse, wenn eine Rolle Ausgabeereignisse erzeugt und umgekehrt. Dadurch wird verdeutlicht, dass die Rollen und der „Glue" durch die gesendeten und empfangenen Ereignisse miteinander kommunizieren.

2.4. 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 wird nicht direkt vorgeschrieben, wie die grundlegenden Elemente einer Architektur dargestellt werden sollen. Deshalb wurden von [ICGNSS04] mehrere Möglichkeiten aufgezeigt, die bestehenden Elemente der UML 2.0 auf Architekturebene zu übertragen. Im Folgenden werden unterschiedliche Strategien gezeigt, wie die Elemente Komponente, Port, Konnektor, Schnittselle und System/Architektur in der UML dargestellt werden können.

Komponententyp
[ICGNSS04] beschreibt zwei Strategien, wie ein Komponententyp durch bestehende UML Elemente dargestellt werden kann.

1. Mittels einer Klasse:
Die Abbildung 2.4.1 zeigt die Darstellung mittels einer Klasse und annotierter Ports (siehe später). Die Darstellung eines Komponententyps durch eine Klasse ist zweckmäßig, da das Verhältnis zwischen Klasse und Objekt genau dem Verhältnis zwischen Komponententyp und Komponenteninstanz entspricht.

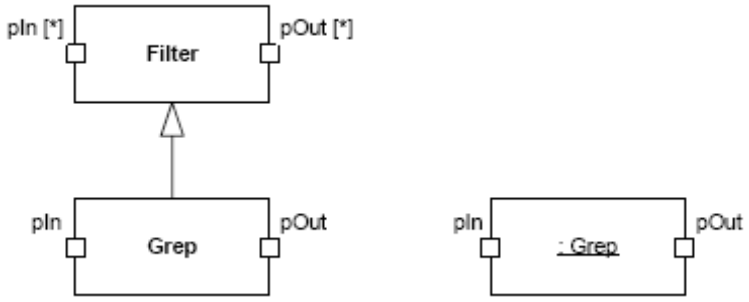

Abb. 2.4.1: Darstellung einer Komponente mittels einer UML Klasse

Bei dieser Strategie können strukturelle Eigenschaften der Komponenten durch Klassenattribute oder Assoziationen dargestellt werden. Ebenso kann das Verhalten durch UML Zustandsdiagramme oder UML Aktivitätsdiagramme modelliert und das Konzept der Generalisierung/Spezialisierung benutzt werden, um die Komponententypen in Beziehung zu setzen. Außerdem ist diese Darstellung eine gute Wahl beim Anwenden UML-basierterter Tools.

Obwohl eine Klasse zur Repräsentation einer Komponente benutzt werden kann, hat diese Strategie zwei Nachteile: Das Verhältnis Klasse/-Instanz und Komponententyp/-Instanz einer Architektur ist zwar ähnlich, aber nicht identisch. So kann eine Komponenteninstanz in einer Architektur die Anzahl der Ports redefinieren, im Gegensatz dazu kann in der UML aber ein Objekt nur die Teile enthalten, die in seiner Klasse definiert wurden. Außerdem besteht bei der gleichzeitigen Darstellung der Komponenten- und Konnektortypen mittels Klassen Verwechslungsgefahr.

2. Mittels einer Komponente:
Im UML 2.0 Metamodell ist die Komponente jetzt ein Subtyp einer Klasse, hat also die gleiche Ausdrucksstärke und die gleichen Eigenschaften wie eine Klasse. Deshalb können die Klassen aus Abbildung 2.4.1 einfach durch UML 2.0 Komponenten ersetzt werden (siehe Abbildung 2.4.2).

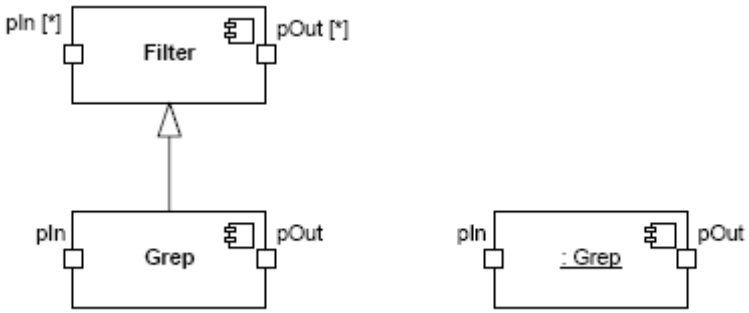

Abb. 2.4.2: Darstellung einer Komponente mittels einer UML 2.0 Komponente

Der Hauptunterschied zwischen einer Klasse und einem Komponententyp ist, das ein Komponententyp zusätzliche Elemente beinhalten kann (Stichwort: Kompositionsstruktur).

Die Wahl der richtigen Strategie hängt davon ab, ob das Modellierungswerkzeug das neue Komponentenkonzept unterstützt.

Port
Das Konzept der Port ist neu in der UML 2.0. Ein Port stellt einen Punkt dar, über den mit der Außenwelt (also mit anderen Komponenten bzw. Konnektoren) interagiert werden kann. Dazu kann ein Port mit mehreren bereitstellenden und benötigten Schnittstellen verbunden werden. Diese Schnittstellen beschreiben die Interaktionen der Komponente mit der Außenwelt, die an diesem Port möglich sind.

Abb. 2.4.3: Komponententyp mit zwei Ports

Die Abbildung 2.4.3 zeigt einen Komponententyp mit den beiden Ports `pIn` und `pOut`. Der Port `pIn` ist mit einer benötigten Schnittstelle verbunden, dargestellt durch eine Buchse. Der Port `pOut` ist mit einer bereitgestellten Schnittstelle verbunden, dargestellt als „Lollipop“. Die gesamte Kommunikation der Komponenten erfolgt nur über Ports, so dass die Komponenten leicht ausgetauscht werden können

Konnektortyp
[ICGNSS04] beschreibt, dass das Konzept des Konnektortyps in der UML 2.0 zu wenig Ausdruckkraft besitzt, um damit einen Konnektortyp im Sinne einer Architektur zu beschreiben. So ist es nicht möglich, semantische Informationen (z.B. die Beschreibung des Verhaltens) oder Rollen mit einem Konnektor zu verbinden. Aufgrund dieser Mängel wurden von [ICGNSS04] drei Konzepte entwickelt, wie Konnektoren besser dargestellt werden können.

1. Mittels einer Assoziation:
Die erste Möglichkeit ist die Darstellung mittels einer Assoziation. Die Abbildung 2.4.4 zeigt zwei Komponenten (hier dargestellt als Klasseninstanzen), die über eine Verbindung (= eine Instanz einer Assoziation) verbunden sind. Durch einen Stereotyp (hier `<<pipe>>`) kann die Art der Verbindung näher spezifiziert werden.

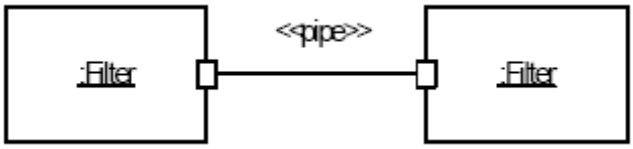

Abb. 2.4.4: Darstellung eines Konnektors mittels einer Verbindung
(= Instanz einer Assoziation)

Durch eine Assoziation kann ein einfacher Konnektortyp dargestellt werden. Soll dieser aber näher spezifiziert werden, ist eine Assoziation nicht mehr ausreichend. So kann ein Konnektortyp das Protokoll zwischen den Komponenten beschreiben und/oder selbst eine Semantik haben. Außerdem können mit dieser Darstellung keine Rollen definiert werden, da eine Assoziation keine Schnittstellen bzw. Ports haben darf.

2. Mittels einer Assoziationsklasse:
Diese Probleme können durch die Benutzung einer Assoziationsklasse umgangen werden. Die Abbildung 2.4.5 zeigt das oben genannte Beispiel verfeinert mit einer Assoziationsklasse (genau genommen einer Instanz einer Assoziationsklasse).

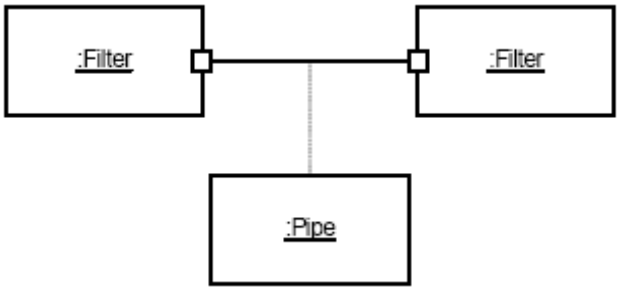

Abb. 2.4.5: Darstellung eines Konnektors mittels einer Assoziationsklasseninstanz

Eine Assoziationsklasse erlaubt es, zusätzlich die Semantik und das Verhalten einer Assoziation zu beschreiben. Außerdem ist es dadurch möglich, eine Rolle durch einen Port darzustellen. Die Abbildungen 2.4.6 bis 2.4.8 zeigen verschiedene Möglichkeiten, wie eine Verknüpfung zwischen Rolle und Port dargestellt werden kann.

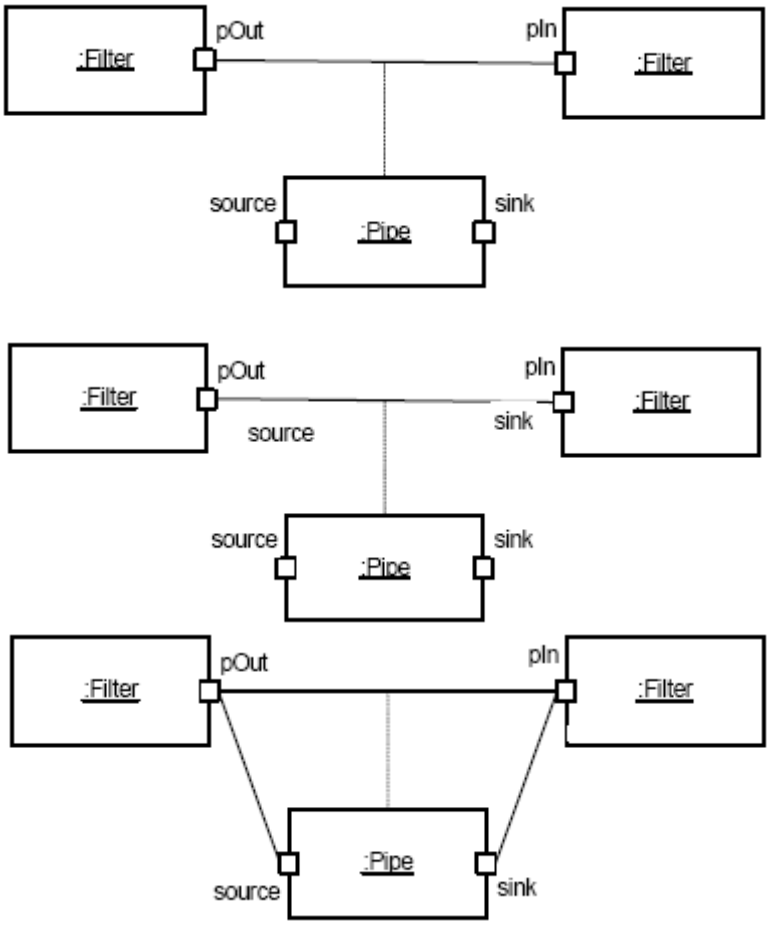

Abb. 2.4.6-2.4.8: Darstellung von Konnektoren mittels Assoziationsklasseninstanzen und der Rollen durch Ports

In Abbildung 2.4.6 ist nicht klar, ob die Rolle `source` dem Port `pIn` oder `pOut` zugeordnet ist. Diese Mehrdeutigkeit lässt sich wie in den beiden Abbildungen 2.4.7 und 2.4.8 gezeigt lösen. Die Abbildung 2.4.8 hat aber wiederum das Problem, dass bei großen Systemen diese Art der Darstellung unübersichtlich wird.

3. Mittels einer Klasse:
Die letzte Möglichkeit, einen Konnektortyp in UML darzustellen ist mittels einer Klasse. Die Abbildung 2.4.9 zeigt ein Beispiel, in dem ein Konnektor als Klasseninstanz zwischen zwei Komponenten dargestellt wird.

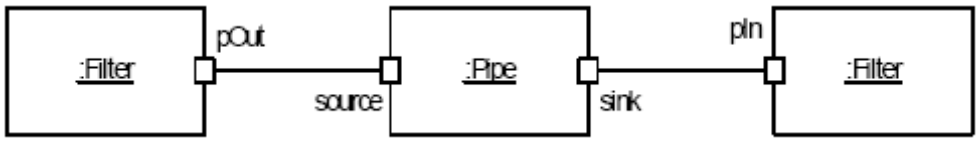

Abb. 2.4.9: Darstellung eines Konnektors mittels einer Klasseninstanz

Diese Darstellung löst die Probleme der zweiten Strategie, ohne die Ausdrucksstärke zu beeinflussen. Außerdem gibt es bei dieser Darstellung keine Mehrdeutigkeiten wie in Abbildung 2.4.7 und das System bleibt auch bei steigender Komplexität übersichtlich, im Gegensatz zu Abbildung 2.4.8. Leider kann bei dieser Darstellung schwer zwischen Konnektor und Komponente unterschieden werden. Eine Möglichkeit dieses Problem zu lösen ist, die Komponenten durch UML Komponenten und die Konnektoren durch Klasseninstanzen darzustellen, wie in Abbildung 2.4.10 gezeigt.

Abb. 2.4.10: Klassen durch Komponenten ausgetauscht

System/Architektur
Die Abbildung 2.4.11 zeigt, wie die genannten Konzepte benutzt werden können, um ein konkretes System zu beschreiben.

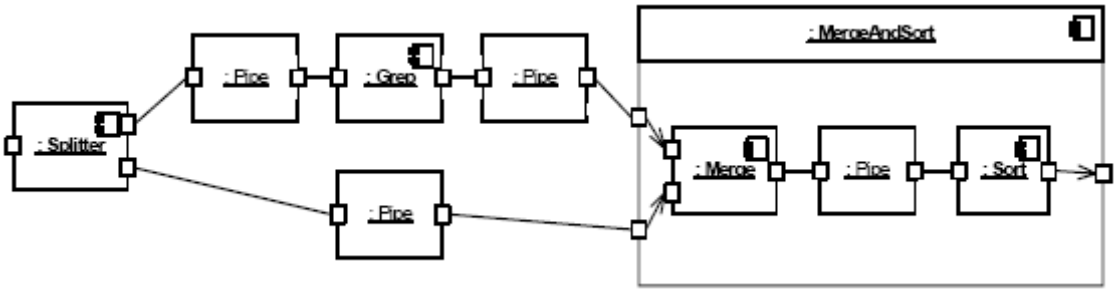

Abb. 2.4.11: Dokumentation eines Systems

In dieser Abbildung werden Komponenten durch UML Komponenten und Konnektoren durch Klasseninstanzen dargestellt. Außerdem wird hier gezeigt, dass in der UML 2.0 durch das Konzept der Kompositionsstruktur eine Komponente (z.B. `MergeAndSort`) weitere Elemente beinhalten kann. Wie die Ports einer Komponente mit den Ports der Unterelemente (den sogenannten Parts) verbunden sind, wird durch Delegations-konnektoren ebenfalls gezeigt.

3. Bewertung und Fazit

Im Folgenden werden die Eigenschaften der oben genannten Architekturbeschreibungs-sprachen kurz erläutert.

Rapide

- Erzeugt einen Prototypen der Architektur, der simuliert werden kann
- Es existieren viele Simulationswerkzeuge für Rapide
- Der Sprachstandard ist sehr stabil (seit 1998 erfolgte keine Änderung)
- Kann Quelltext-Frameworks erstellen
- **Problem:** Sprache muss erlernt werden

Rapide ist eine ADL die nicht nur akademischen Charakter besitzt, sondern auch praktisch Anwendung findet. Zum Beispiel wurde die Architektur des SPARC-V9-Prozessors in Rapide realisiert [IntStan].

Wright

- Wird von wenige Werkzeugen unterstützt, aber Werkzeuge für CSP können verwendet werden
- Momentan können die wenigen Werkzeuge kein Quelltext-Framework erstellen
- **Problem:** Sprache muss erlernt werden

UML

- Es gibt verschiedene Strategien, wie Software-Architekturen/Systeme beschrieben werden können (wie gezeigt)
- Es ist nicht möglich, den integrierten Konnektor mit semantischen Apsekten zu versehen
- **Gut:** UML-Diagramme sind intuitiv und damit leicht zu verstehen

Eine Besonderheit der UML ist das Einsatzdiagramm, das die Hardware-Konfiguration zeigt, auf der das laufende Zielsystem zum Einsatz kommt. Außerdem stellt es die Verteilung der (Software)-Komponenten auf den einzelnen Knoten (= eine physische Speicher- und Verarbeitungseinheit, die zur Laufzeit existiert) der (Hardware-)Konfiguration dar. In der UML 2.0 hat sich gegenüber seinen Vorgängern in Richtung ADL viel getan. Die Einführung der Kompositionsstruktur zur Darstellung von Hierarchien und Ports als Interaktionspunkt hat die Situation wesentlich verbessert.

Zum Schluss wäre noch zu erwähnen, dass es bis jetzt noch keine Standard-ADL gibt. Da die UML aber bereits die dominierende Modellierungssprache im Software-Engineering ist, kristallisiert sie sich langsam als ADL-Standard heraus. [vgl. IntWiki]

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“, 1999

[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