Software Architecture Systems

0. Motivation: Separate architecture aspect from application
2. Case studies: Unicon, Modelica, CoSy
3. Other architecture systems (PhD course only)
4. Modeling Software Architecture with UML and UML 2.0
5. Application of UML connectors in design (PhD course only)
6. Summary

Additional Literature

http://www-2.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html

http://citeseer.ist.psu.edu/shaw96field.html


Additional Literature (cont.)


Examples of Architecture Systems

http://citeseer.ist.psu.edu/shaw95abstractions.html


- (Darwin) http://www-dse.doc.ic.ac.uk/Software/Darwin/


### The Ladder of Component and Composition Systems

<table>
<thead>
<tr>
<th>Classical Component Systems</th>
<th>Object-Oriented Systems</th>
<th>Modular Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Systems</td>
<td>Architecture as Aspect</td>
<td>Danin</td>
</tr>
<tr>
<td>Standard Components</td>
<td></td>
<td>ACME</td>
</tr>
<tr>
<td>Modular Systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Objects as Run-Time Components</td>
<td></td>
<td>C++ Java</td>
</tr>
<tr>
<td>Modules as Compile-Time Components</td>
<td></td>
<td>Module Ada85</td>
</tr>
</tbody>
</table>

### Separation of Concerns

- Different concerns should be separated
  - so that they can be specified independently

- Dimensional specifications

- Specify from different viewpoints

- But: different concerns are not always independent of each other
  - Interferences
  - Consistency issues
  - Ordering constraints on application

### A Basic Rule for Design ...

- ... is to focus on one problem at a time and to forget about others.

  - Abstraction is neglection of unnecessary detail
    - Display and consider only essential information

### An Example of Separation of Concerns: Separate Policy and Mechanism

- Mechanism:
  - The way to technically realize a solution

- Policy:
  - The way to parameterize the realization of a solution

Separate Policy from Mechanism!

Then they can be varied independently.
Aspects in Architecture

Structure

Light plan

Integrated house

Water pipe plan

Media plan

Software Architecture Systems as Composition Systems

- Component model
  - Binding points: Ports
  - Communication between component instances is split off in connectors: Transfer (carrier) of the communication is transparent

- Composition technique
  - Adaptation and glue code generated from connectors
  - Aspect separation: application and communication are separated
    - Topology (who communicates with whom?)
    - Carrier (how?)
    - When?
  - Scalability (distribution, binding time with dynamic architectures)

- Composition language:
  - An Architecture Description Language (ADL) is a simple composition language!

An Example of Separation of Concerns: Architectural Aspect in Software

Software Architecture (Connection of components)

Components

Software configuration with glue code for communication between components

Codegenerator

Component Model in Architecture Systems

- Ports = abstract interface points (events, methods)
- Ports specify the data-flow into and out of a component
  - in(data)
  - out(data)

- Connectors as special communication components
  - Connectors are attached to ports
  - Connectors are explicitly applied per communication
  - Components and connectors are bound together to form a configuration.
Abstract Binding Points: Ports

- Ports abstract from the concrete carrier, but indicate where data has to flow in and out of the component.
  - To fit to connectors, a legacy system must convert all procedure calls to ports, i.e., to abstract calls.
  - Ports have protocols.
- Connectors can be binary or n-ary.
  - Every end is called a role.
  - Roles fit only to certain types of ports = Typing of roles and ports.
- The interfaces remain at run time.

Ports In More Detail

- Input ports are synchronous or asynchronous:
  - `in(data)` (aka. `receive(data)`):
    - `testAndGet(data)`: Asynchronous in port, taking in one data if it is available.
  - Output ports are synchronous or asynchronous:
    - `out(data)`
      - `put(data)`: Asynchronous out port, putting out one data, not waiting until acknowledge.

A Simple Example

- A description of a small example architecture in the ADL Acme [Garlan et al., CMU, 2000]

```plaintext
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc = { Roles { caller, callee } }
  Attachments: {
    client.sendRequest to rpc.caller;
    server.receiveRequest to rpc.callee;
  }
}
```

Ports and Services

- Services are groups of ports.
- A data service is a tuple:
  - `[in(data), ..., in(data), out(data), ..., out(data)]`
- A special case is a call service with one return port:
  - `[in(data), ..., in(data), out(data)]`
- A property service is a service to access component attributes, i.e., a simple tuple:
  - `[in(data), out(data)]`
### Architectural Styles

- Frequently occurring connection topology patterns (Architectural Design Patterns)
  - Pipe-and-Filter
    - UNIX shells
    - Stream-parallel programming languages
  - Client-Server Architecture
    - CORBA RPC, Java RMI, ...
  - Layered Architecture (aka. Onion Architecture)
    - Layered operating systems (UNIX, Windows)
    - Multi-tier architectures (e.g. 3-tier: clients / server objects / DB)
  - Blackboard Architecture (aka Repository Architecture)
    - Linda [Carriero/Gelernter’96]
    - Service discovery repositories, e.g. Jini, CORBA repositories
    - CoSy CCMIR

*...and more, and combinations of these*

### Architecture can be Exchanged Independently of Components

- “Rewiring”
- Reuse of components and architectures is fundamentally improved

### Two Dimensions of Reuse

- Architecture and components can be reused independently of each other

### Architecture Descriptions are Reducible

- Components are nested (fractal-like behavior)
- Ports of outer components are called players.
- This type of diagram is now supported in UML 2.0 as component diagram
Additionally, Connectors have Protocols

- A connector, since it is a precise concept to specify communication of components, must have a protocol.

Set/Get Connector Protocol

- on data services

Client
- get Field
- set Field(value)

Server
- set Field(value)

Call Connector Protocol

- on call services

Client
- initialize
- call(data)
- return value

Server
- initialize
- return value

RPC Connector

- on call services

Client
- call(data)
- return value

Server
- call(data)
- return value
Dynamic Call via CORBA DII - Protocol

Connectors Provide Glueing in Connections

From Connectors in ADL Specification Generate Architectural Glue Code

Connectors are Abstract Communication Buses
But we know that already from CORBA:

- CORBA is a simple architecture system with restricted connectors:

- CORBA Client and service provider
- ORB client side, server side
- Marshalling, Stub, Skeleton, Object Adapter
- Interfaces in IDL (not abstracted to data flow)
- Static call
- Dynamic call
- Connectors always binary
- Events, callbacks, persistence as services (cannot be exchanged to other communications)

Most Commercial Component Systems Provide Restricted Forms of Connectors

- It turns out that most commercial component systems do not offer connectors as explicit modelling concepts, but
- offer communication mechanisms that can be encapsulated into a connector component
- For instance, CORBA remote connections can be packed into connectors

Architecture Systems

- Examples
  - Unicon [Shaw 95]
  - Aesop [Garlan95]
  - Darwin [Kramer 92]
  - Rapide [Luckham95], C2 [Medvedovic]
  - Wright [Garlan/Allen]
  - ACME [Garlan 2000]
  - CoSy [Assmann/Alt/vanSomeren'94] [www.ace.nl]
    - Equation-based connectors

Example: The KWIC Problem in UNICON
[ISC pp. 74-76]

- Example from UniCon distribution
- "Keyword in Context" problem (KWIC)
  - The KWIC problem is one of the 10 model problems of architecture systems
  - Originally proposed by Parnas to illustrate advantages of different designs
    [Parnas'72]
  - For a text, a KWIC algorithm produces a permuted index
    - every sentence is replicated and permuted in its words, i.e., the words are shifted from left to right.
    - every first word of a permutation is entered into an alphabetical index, the permuted index.

The KWIC Problem in Unicon

- The components of KWIC work in a pipe-and-filter style
- KWIC has ports
  - stream input port \textit{input},
  - and two output ports \textit{output} and \textit{error}.
  They read text and spit out the permuted index
- KWIC is a compound component KWIC
  (Components in Unicon can be nested)
  - PLAYER definitions define ports of outer components.
  - BIND statements connect ports from outer components to ports of inner components.
- \textbf{USES definitions} create instances of components and connectors.
- CONNECT statements connect connectors to ports at their roles.

A KWIC Index

\begin{verbatim}
  every sentence is replicated
  and permuted
  every sentence is replicated and permuted
  every sentence is replicated and permuted
  every sentence is replicated and permuted
  every sentence is replicated and permuted
\end{verbatim}
The KWIC Problem in Unicon

- **Components**
  - The component `caps` converts the sentence to uppercase as necessary.
  - The `shifter` creates permutations of the sentence.
  - The `req-data` provides some data to the `merge` component which pipes the generated data to the component `sorter`.
  - `sorter` sorts the shifted sentences so that they form a keyword-in-context index.

- **Only connectors in the style of UNIX pipes are used**
- Other connection kinds can be introduced by only changing the type of connectors in a USES declaration.
- Hence, communication kinds can be exchanged easily, e.g. for Shared memory, Abstract data types, Message passing. [Garlan/Shaw'94]

- **Architecture systems allow for scalable communication**
  - binding procedures can be exchanged easily!

COMPONENT KWIC
/* This is the interface of KWIC with in- and output ports */ INTERFACE IS TYPE Filter        PLAYER input IS StreamIn SIGNATURE ("line") PORTBINDING (stdin)  END input        PLAYER output IS StreamOut SIGNATURE ("line") PORTBINDING (stdout) END output END INTERFACE IMPLEMENTATION IS        /* Here come the component definitions */ USES caps        INTERFACE upcase         END caps        USES shifter      INTERFACE cshift            END shifter USES req-data   INTERFACE const-data    END req-data USES merge      INTERFACE converge      END merge        USES sorter       INTERFACE sort               END sorter        /* Here come the connector definitions */ USES P PROTOCOL Unix-pipe END P USES Q PROTOCOL Unix-pipe END Q USES R PROTOCOL Unix-pipe END R ......
/* Here come the connections */ BIND          input                 TO caps.input       CONNECT caps.output      TO P.source       CONNECT shifter.input      TO P.sink       CONNECT shifter.output    TO Q.source CONNECT req-data.read   TO R.source CONNECT merge.in1         TO R.sink       CONNECT merge.in2         TO Q.sink       /* Syntactic sugar for anonymous connections */ ESTABLISH Unix-pipe WITH merge.output AS source sorter.input    AS sink       END Unix-pipe       BIND output TO sorter.output    END IMPLEMENTATIONEND KWIC

KWIC in Unicon

Modelica [Fritzson 2004]

- **Equation-based language for modeling and simulation of systems in physics and engineering**

- **Component model, ports, connectors**
- **Simple example:** Resistor component

```
R : model Resistor "Ideal resistor" extends OnePort;
    parameter Resistance R;
    equation
        R*i - v;
end Resistor;
```

Modelica (cont.)

- **Components are connected by connect statements**
- **Composition by equality of port variables** \(\rightarrow\) connects equations (realizes e.g. Kirchhoff's Node Law, Newton's First Law, ...)

```
R : model Resistor "Ideal resistor" extends OnePort;
    parameter Resistance R;
    equation
        R*i - v;
end Resistor;
```

- **Compiler builds a system of differential equations (ODE's, DAE's)**
  - Solving this equation system (numerically) = Simulation of the system.

```
R : model Resistor "Ideal resistor" extends OnePort;
    parameter Resistance R;
    equation
        R*i - v;
end Resistor;
```

**Example:** DC motor model
Graphical composition (attaching ports by drag-and-drop) creates connect statements in the model description code.

```model MotorDrive
  FID controller;  ------
  Motor motor;
  Gearbox gear (n=100);
  Inertia inertia(J=10);
  equation
  connectcontroller.outPort, motor.inPort);
  connectcontroller.inPort2, motor.outPort);
  connect(gear.flange_a , motor.flange_b);
  connect(gear.flange_b , inertia.flange_a);
end MotorDrive;
```

... Back to classical software architecture systems: The Composition Language: ADL

- Architecture language (architectural description language, ADL)
  - ADL-compiler
  - XML-Readers/Writer for ADL
- The reducibility of the architecture allows for simple overview, evolution, and documentation
  - The architecture is a reducible graph, with all its advantages
- Graphic editing of systems

ACME Studio
What ADL Offer for the Software Process

- Support when doing the requirements specification
  - Visualization for the customer: architecture graphics better to understand
  - Architecture styles classify the nature of a system in simple terms
- Design support
  - Simple specification by graphic editors
  - Stepwise design and refinement of architectures
  - Visual and textual views
- Design of product families is easy
  - A reference architecture fixes the commonalities of the product line
  - The components express the variability

Checking and Validating

- Checking, analysing
  - Test of (part of) an architecture with dummy components
  - Deadlock checking
  - Liveness checking
- Validation: Tools for consistency of architectures
  - Are all ports bound?
  - Do all protocols in the connectors fit?
  - Does the architecture correspond to a certain style?
  - Does the architecture fit to a reference architecture?
  - Parallelism features as deadlocks, fairness, liveness,
  - Dead parts of the system: Is everything reachable at run time?

What can be generated?

- Glue- and adapter code from connectors and ADL-specifications
  - Mapping of the protocols of the components to each other
  - Generation of glue code from the connectors
- Simulations of architectures (with dummy components):
  - The architecture can be created first
  - And tested stand-alone
  - Run time estimates are possible (if run times of components are known)
- Test cases for architectures
- Documentation (graphic structure diagrams)

CoSy

A commercial architecture system for compilers

[ISC 1.3]
www.ace.nl
Traditional Compiler Structure

- Traditional compiler model: sequential process
- Improvement: Pipelining (by files/modules, classes, functions)
- More modern compiler model with shared symbol table and IR

A CoSy Compiler with Repository-Architecture

- Engine
  - Modular compiler building block
  - Performs a well-defined task
  - Focus on algorithms, not compiler configuration
  - Parameters are handles on the underlying common IR repository
  - Execution may be in a separate process or as subroutine call - the engine writer does not know!
  - View of an engine class: the part of the common IR repository that it can access (scope set by access rights: read, write, create)

Examples: Analyzers, Lowerers, Optimizers, Translators, Support

Composite Engines in CoSy

- Built from simple engines or from other composite engines by combining engines in interaction schemes (Loop, Pipeline, Fork, Parallel, Speculative, ...)
- Described in EDL (Engine Description Language)
- View defined by the joint effect of constituent engines
- A compiler is nothing more than a large composite engine

```c
ENGINE CLASS compile (IN: mirUNIT) {
  PIPELINE
  frontend (u)
  optimizer (u)
  backend (u)
}
```
Hierarchical Components in the Repository Style (CoSy)

Example for CoSy EDL (Engine Description Language)

- Component classes (engine class)
- Component instances (engines)
- Basic components are implemented in C
- Interaction schemes (cf. skeletons)
  - SEQUENTIAL
  - PIPELINE
  - DATAPARALLEL
  - SPECULATIVE
- EDL can embed automatically
  - Single-call-components into pipes
  - p<> means a stream of p-items
  - EDL can map their protocols to each other (p y p<>)

EDL can map their protocols to each other (p y p<>)

CoSy generates for every component an adapter (envelope, container)
  - that maps the protocol of the component to that of the environment (all combinations of interaction schemes are possible)
  - Coordination, communication, encapsulation and access to the repository are generated.
Evaluation of CoSy

- CoSy is one of the single commercial architecture systems with professional support
- The outer call layers of the compiler are generated from the ADL
  - Adapter, coordination, communication, encapsulation
  - Sequential and parallel implementation can be exchanged (cf. skeletons)
- There is also a non-commercial prototype
- Access layer to the repository must be efficient (solved by generation of macros)
- Because of views, a CoSy-compiler is very simply extensible
  - That’s why it is expensive
  - Reconfiguration of a compiler within an hour

Survey of Other Architecture Systems

- For self-studies...
  - UniCon
  - RAPIDE
  - Aesop
  - Acme
  - Darwin

An Example System: UNICON

- UNICON supports
  - Components in C
  - Simple and user-defined connectors
- Design Goals
  - Practical tool for real problems
  - Uniform access to a large set of connections
  - Check of architectures (connections) should be possible
  - Analysis tools
  - Graphics and Text
  - Reuse of existing legacy components
  - Reduce additional run time costs

Description of Components and Connectors

- Name
- Interface (component) resp. protocol (connector)
- Type
  - component: modules, computation, SeqFile, Filter, process, general
  - connectors: Pipe, FileIO, procedureCall, DataAccess, PLBandier, RPC, RTScheduler
- Global assertions in form of a feature list (property list)
- Collection of
  - Players for components
    - for ports and port mappings for components of different nesting layers)
  - Roles for connectors
- The UNICON-compiler generates
  - Odin-Files from components and connectors. Odin is an extended Makefile
  - Connection code


**Supported Player Types per Component Type**

- Modules:
  - RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBandle, ReadFile, WriteFile
- Computation:
  - RoutineDef, RoutineCall, GlobalDataUse, PLBandle
- SharedData:
  - GlobalDataDef, GlobalDataUse, PLBandle
- SeqFile:
  - ReadNext, WriteNext
- Filter:
  - StreamIn, StreamOut
- Process:
  - RPCDef, RPCCall
- Schedprocess:
  - RPCDef, RPCCall, RTLoad
- General:
  - All

**Supported Role Types For Connector Types**

- Pipe:
  - Source fits to Filter.StreamOut, SeqFile.ReadNext
  - Sink fits to Filter.StreamIn, SeqFile.WriteNext
- Field:
  - Reader fits to modules.ReadFile
  - Reader fits to SeqFile.ReadNext
  - Writer fits to Modules.WriteFile
  - Writer fits to SeqFile.WriteNext
- ProcedureCall:
  - Definer fits to (Computation|Modules).RoutineDef
  - User fits to (SharedData|Computation|Modules).GlobalDataUse
- PLBandler:
  - Participant fits to PLBandle, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef
- RPC:
  - Definer fits to (Process|Schedprocess).RPCDef
  - User fits to (Process|Schedprocess).RPCCall
- RTScheduler:
  - Load fits to Schedprocess.RTLoad

**A Modules Component**

INTERFACE IS

TYPE modules

LIBRARY

PLAYER timeget IS RoutineDef
  SIGNATURE ("new_type"; "void")
END timeget

PLAYER timeshow IS RoutineDef
  SIGNATURE ("void")
END timeshow

END INTERFACE

**A Filter**

COMPONENT Reverse INTERFACE IS
  TYPE Filter
  PLAYER input IS StreamIn SIGNATURE ("line") PORTBINDING (stdin) END input
  PLAYER output IS StreamOut SIGNATURE ("line") PORTBINDING (stdout) END output PLAYER error IS StreamOut SIGNATURE ("line") PORTBINDING (stderr) END error
END INTERFACE

IMPLEMENTATION IS

/* Component instantiations are declared below */
USING reverse INTERFACE Reverse
USING stack INTERFACE Stack
USING libc INTERFACE Libc
USING datause protocol C-shared-data

/* We will use <establish> statements for the procedure call connections (next page) */

/* Now for the configuration of connectors to players */
CONNECT Reverse.bind ports to <define>
CONNECT Reverse乜 TO datause.definer
CONNECT libc.To TO datause.definer
END IMPLEMENTATION END Reverse
/* Establish connections */
ESTABLISH C-proc-call WITH reverse.stack_init AS caller stack.stack_init AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.stack_is_empty AS caller stack.stack_is_empty AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.push AS callr stack.push AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.pop AS callr stack.pop AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.exit AS callr libc.exit AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.fgets AS callr libc.fgets AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.fprintf AS callr libc.fprintf AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.malloc AS callr libc.malloc AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.strcpy AS callr libc.strcpy AS definer END C-proc-call
ESTABLISH C-proc-call WITH reverse.strlen AS callr libc.strlen AS definer END C-proc-call

/* Lastly, we bind the players in the interface to players in the implementation. Remember, it is okay to omit the bind of player “error.” */
BIND input TO ABSTRACTION MAPSTO (reverse.fgets) END input
BIND output TO ABSTRACTION MAPSTO (reverse.fprintf) END output END IMPLEMENTATION

In Version 4.0, connectors can be defined by users. However, the extension of the compilers is complex: a delegation class has to be developed, the semantic analysis, and the architecture analysis must be supported.
RAPIDE


- Central idea:
  Rapide leaves the **object connection architecture**, in which the objects are attached to each other directly, for an **interface connection architecture**, in which **required and provided interfaces** are related to each other
  - Specify in a interface not only the required methods, but also the offered ones (provided and required ports)
  - Connect the ports in a architecture description (separate)
  - Advantage: calls can be bound to other ports with different names
  - Generalizes ports to calls
- **Fundamentally more flexible concept for modules!**
- Rapide was marketed by a start-up company

Aesop

- Connectors are **first class language elements**
  - i.e., can be defined by users
  - Connectors are classes which can be refined by inheritance
- Users can derive their own connectors from system connectors
- Aesop supports the definition of **architectural styles with fables**
  - Architectural styles obey rules

Aesop Supports Architectural Styles (Fables)

- **Design Rule**
  - A design rule is an element of code with which a class extends a method of a super class. A design rule consists of the following:
    - A **pre-check** that helps control whether the method should be run or not.
    - A **post-action**
- **Environment**
  - A design environment tailored to a particular architectural style.
    - It includes a set of policies about the style, and a set of tools that work in harmony with the style, visualization information for tools
    - If something is part of the formal meaning, it should be part of a style
    - If it is part of the presentation to the user, it should be part of the environment.
ACME (CMU)

- ACME is an exchange language (exchange format) to which different ADL can be mapped (UNICON, Aesop, ...).
- It consists of abstract syntax specification
  - Similar to feature terms (terms with attributes).
  - With inheritance

```
Template SystemIO () : Connector {
  Roles (source = SystemIORole(); sink = SystemIORole();)
  properties (blockingtype = non-blocking; Aesop-style = subroutine-call)
}
```

Example ACME Pipe/Filter-Family

```
Example ACME Pipe/Filter-Family

We describe a simple pipe-filter family. This family definition demonstrates ACME's ability to specify an entire architecture as well as individual architectural instances.

An ACME family includes a set of component types, ports and role types that define the design vocabulary provided by the family.

```
Family PipeFilterFam = {
  // Declare component types.
  // A component type definition in ACME allows you to define the structure required by the type.
  // This structure is defined using the same syntax as an instance of a component.
  Component Type UnixFilterT extends FilterT with {
    property implementationFile : String;
  };
  Component Type SubsystemT = {
    property visualization : VisualizationT = [ x = 20; y = 30; width = 100; height = 75; shape = rect; color = black ];
  };
  // A component type definition in ACME allows you to define the structure required by the type.
  // This structure is defined using the same syntax as an instance of a component.
  Component Type ShowTracksSubsystem = {
    property visualization : VisualizationT = [ x = 200; y = 30; width = 100; height = 75; shape = rect; color = black ];
  };
}
```

ACME Studio as Graphic Environment

```
ACME Studio as Graphic Environment

Instance of an ACME System

We describe an instance of a system using the PipeFilterFam family.

```
System simplePF : PipeFilterFam = {
  property throughput : int;
  property bufferSize : int;
  property tasks : TasksT = { property time : int; property priority : int; property scheduling : int; property lock : boolean; property thread : threadID; property args : array(100); property env : environment; property resources : array(100); property status : status; }
};
```

```
Example ACME Pipe/Filter-Family

We describe a simple pipe-filter family. This family definition demonstrates ACME's ability to specify an entire architecture as well as individual architectural instances.

An ACME family includes a set of component types, ports and role types that define the design vocabulary provided by the family.

```
Family PipeFilterFam = {
  // Declare component types.
  // A component type definition in ACME allows you to define the structure required by the type.
  // This structure is defined using the same syntax as an instance of a component.
  Component Type UnixFilterT extends FilterT with {
    property implementationFile : String;
  };
  Component Type SubsystemT = {
    property visualization : VisualizationT = [ x = 20; y = 30; width = 100; height = 75; shape = rect; color = black ];
  };
  // A component type definition in ACME allows you to define the structure required by the type.
  // This structure is defined using the same syntax as an instance of a component.
  Component Type ShowTracksSubsystem = {
    property visualization : VisualizationT = [ x = 200; y = 30; width = 100; height = 75; shape = rect; color = black ];
  };
}
```

ACME Studio as Graphic Environment

```
ACME Studio as Graphic Environment

Instance of an ACME System

We describe an instance of a system using the PipeFilterFam family.

```
System simplePF : PipeFilterFam = {
  property throughput : int;
  property bufferSize : int;
  property tasks : TasksT = { property time : int; property priority : int; property scheduling : int; property lock : boolean; property thread : threadID; property args : array(100); property env : environment; property resources : array(100); property status : status; }
};
```

London Ambulance System in ACME

```c
// incidentInfoPath attachments
Attachments {  // calls to incident_manager
callInfo.sendCallMsg to callInfoChannel.fromRole;
incidentMgr.receiveCallMsg to callInfoChannel.toRole;
  // incident updates to resource_manager
incidentMgr.examineIncident to incidentNGP2.inIncident;
resourceMgr.receiveIncidentInfo to incidentNGP2.outIncident;
  // dispatch requests to dispatcher
resourceMgr.sendDispatchRequest to dispatcher.receiveDispatchRequest;
dispatcher.receiveDispatchRequest to dispatchRequestChannel.toRole;
  }

// rpcRequests attachments
Attachments {  // calls to map_server
incidentMgr.mapRequest to mapRequest1.clientEnd;
mapServer.requestPort1 to mapRequest1.serverEnd;
  // incident info from incident_manager
resourceMgr.incidentInfoRequest to incidentInfoRequest.clientEnd;
incidentMgr.incidentInfoRequests to incidentInfoRequest.serverEnd;
  }
```

London Ambulance System in ACME (cont.)

```
// incidentInfoPath attachments
Attachments {  // calls to incident_manager
callInfo.sendCallMsg to callInfoChannel.fromRole;
incidentMgr.receiveCallMsg to callInfoChannel.toRole;
  // incident updates to resource_manager
incidentMgr.examineIncident to incidentNGP2.inIncident;
resourceMgr.receiveIncidentInfo to incidentNGP2.outIncident;
  // dispatch requests to dispatcher
resourceMgr.sendDispatchRequest to dispatcher.receiveDispatchRequest;
dispatcher.receiveDispatchRequest to dispatchRequestChannel.toRole;
  }

// rpcRequests attachments
Attachments {  // calls to map_server
incidentMgr.mapRequest to mapRequest1.clientEnd;
mapServer.requestPort1 to mapRequest1.serverEnd;
  // incident info from incident_manager
resourceMgr.incidentInfoRequest to incidentInfoRequest.clientEnd;
incidentMgr.incidentInfoRequests to incidentInfoRequest.serverEnd;
  }
```

Darwin (Imperial College)

- **Components**
  - Primitive and composed
  - Components can be recursively specified or iterated by index range
  - Components can be parameterized
- **Ports**
  - In, out (required, provided)
  - Ports can be bound implicitly and in sets
- **Several versions available** (C++, Java)
- **Graphic or textual edits**
Simple Producer/Consumer

Architectural Languages in UML

Hofmeister, Nord, Søn: Describing Software Architecture with UML. 1999

Simple Producer/Consumer in Text

Architectural Languages versus UML

- So far, architecture systems and languages were research toys (except CoSy)
- "I have to learn UML anyway, should I also learn an ADL??"
  - Learning curve for the standard developer
  - Standard?
  - Development environments?
- This changes with UML 2.0
The Hofmeister Model of Architecture

- [Hofmeister/Nord/Soni’99] is the first article that has propagated the idea of specifying an architecture language with UML
  - Conceptual view: Functionality + interaction (components, ports, connectors)
  - Module view: Layering, modules and their interconnections
  - Execution view: runtime architecture (mapping modules to time and resources)
  - Code view: division of systems into files

- Describe these single views in UML
  - UML allows the definition of stereotypes
    - Model connectors and ports, modules, runtime components with stereotypes
    - Map them to icons, so that the UML specification looks similar to a specification in an architecture system

Background: Stereotypes in UML

- A stereotype is a UML modeling element introduced at modeling time. It represents a subclass of an existing modeling element (= metalevel) with the same form (attributes and relationships) but with a different intent, maybe special constraints.
  - To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype.
  - A mechanism for extending/customizing UML without changing it.

Example scenario: [Hofmeister/Nord/Soni’99]

- Digital camera produces sequence of image frames, flattened into a stream of pixel data
- Image acquisition system selects, starts, adjusts an image acquisition procedure
- Image processing pipeline
  - Framer: Restore complete image frames from pixel stream
  - Imager: One or more image transformation(s)
- Display images

UML model for image processing example
Modeling software architecture in UML

- For conceptual view: Class diagram
- Components, ports, connectors are a stereotype of Class: <<component>>, <<port>>, <<connector>>
- Use special symbols for ports and connectors
- Omit the stereotype for components and show their associations with their ports by nesting
- Roles are a stereotype of Association: <<role>>
  - shown as labels on port-connector associations
  - Default multiplicity is 1

For modeling protocols, use UML Sequence diagram or State diagram

Protocol for PacketIn port:

- Incoming messages
  - RequestDataPacket
  - incoming packet(pd)
- Outgoing messages
  - requestPacket
  - /subscribe

Components in UML 2.0

- Idea has been taken over by UML 2.0:
  - "A component is a self-contained unit that encapsulates the state and behavior of a number of classifiers.
  - ... A component specifies a formal contract of services ..."
  - Provided and required interfaces
  - Substitutable
  - Run-time representation of one or several classes
  - Source or binary code
- Difference to UML classes:
  - No inheritance
- New symbols
  - Components, component instances
  - New UML element, not a stereotype

Components can be nested
Ports in UML 2.0

- Ports in UML 2.0 are port objects (gates, interaction points) that govern the communication of a component
- Ports may be simple (only data-flow, data service)
  - in or out
- Ports may be complex services
  - Then, they implement a provided or required interface

Connectors in UML 2.0

- Connectors become special associations, marked up by stereotypes, that link ports

Simple Producer/Consumer in UML 2.0

- Ports can be grouped to Services
Exchangeability of Connectors

- The more complex the interface of the port, the more difficult it is to exchange the connectors.
- Data-flow ports and data services abstract from many details.
- Complex ports fix more details.
- Only with data services and property services, connectors have best exchangeability.

Rule of Thumb for Architectural Design with UML 2.0

- Start the design with data ports and services.
- Develop connectors.
- In a second step, fix control flow:
  - push-pull
  - Refine connectors.
- In a third step, introduce synchronization:
  - Parallel/sequential
  - Refine connectors.

Architecture Systems: Summary

- How to evaluate architecture systems as composition systems?
  - Component model
  - Composition technique
  - Composition language

Architecture Systems as Composition Systems

Component Model
Source or binary components
Binding points: ports

Composition Technique
Adaptation and glue code by connectors
Scaling by exchange of connectors

Architectural language
Composition Language
ADL: Mechanisms for Modularization

- **Component concepts**
  - Clean language-, interfaces and component concepts
  - New type of component: connectors
  - Secrets:
    - Connectors hide communication transfer
    - Partner of the communication
    - Distribution
  - Parameterisation: depends on language
  - Standardization: still pending

ADL: Mechanisms for Adaptation

- Connectors generate glue code: very good!
  - Many types of glue code possible
  - User definable connectors allow for specific glue
  - Tools analyze the interfaces and derive the necessary adaptation code automatically
- Mechanisms for aspect separation.
  - 2 major aspects are distinguished:
    - Architecture (sub-aspects: topology, hierarchy, communication carrier)
    - Application functionality
- An ADL-compiler is only a rudimentary weaver
  - Aspects are not weaved together but encapsulated in glue code

Architecture Systems - Component Model

- Development environments
- Secrets
- Types
- Distribution
- Location transparency
- Contracts
- Height
- Binding points
- Ports
- Parameterization
- UML genericity
- Business services

Architecture Systems – Composition Technique and Language

- Connection
- Adaptation
- Product quality
- Extensibility
- Software process
- Architecture language
- Architecture is separated
- Aspect Separation
- Metacomposition
- Scalability
- Product quality
What Have We Learned?

- Software architecture systems provide an important step forward in software engineering
  - For the first time, software architecture becomes visible
- Concepts can be applied in UML already today
- Architectural languages are the most advanced form of blackbox composition technology so far

How the Future Will Look Like

- Metamodels of architecture concepts (with MOF in UML) will replace architecture languages
  - The attempts to describe architecture concepts with UML are promising
- Model-driven architecture
  - Increasingly popular, also in embedded / realtime domain
- We should think more about general software composition mechanisms
  - Adaptation by glue is only a simple way of composing components (... see invasive composition)