Overview

- Some more software architectures
- Motivation for ADL
- Basic Concepts of Darwin
- Case study - Cruise Control system
- Property verification and other ADL issues

Acknowledgements: These slides were adopted from the tutorial given at ICSE by Jeff Kramer and Jeff Magee, Imperial College, London and from a number of researchers at CMU.

What are software architectures?

<table>
<thead>
<tr>
<th>Software Architectures</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>interactions among parts structural properties declarative mostly static system-level performance outside component boundary</td>
<td>implementations of parts computational properties operational mostly dynamic algorithmic performance inside component boundary</td>
</tr>
</tbody>
</table>

(from Garlan/Shaw)
Explicit architectural descriptions

“architecture” ➔ “system product”

*Why tolerate a loss of explicit architectural information?*

The architectural description should be specified explicitly and precisely using a special purpose language - an Architectural Description Language (ADL). An ADL specification should be usable both for design analysis and to generate and maintain the system.

---

*What structures should software architectures describe?*

There are many possible architectural models.

Selection is determined by problem domain and implementation environment

- components?
- processes?
- objects?
- interactions?
- interconnections?
- bindings?
- connectors?
- control (invocation)?
- streams (data flow)?
Elements of Architectural Descriptions

- The architecture of a system includes
  - Components: define the locus of computation
    Examples: filters, databases, objects, ADTs
  - Connectors: define the interactions between components
    Examples: procedure call, pipes, event announce
- An architectural style defines a family of architectures constrained by
  - Component/connector vocabulary
  - Topology
  - Semantic constraints

Example: Alternating Characters

Produce alternating case of characters in a stream

Definition/Use Modularization
Components

A component has **state**, exhibits some well-defined **behaviour**, and has a unique **identity**.

The **identity** of an component distinguishes it from all other components.

The **behaviour** of a component represents its outwardly visible and testable activity.

The **state** of a component represents the cumulative results of its behaviour.

A component/object is a member (instance) of a **type** or **class**.
**Issues Addressed by an Architectural Design**

- Gross decomposition of a system into interacting components
  - typically hierarchical
  - often using common design idioms/styles
- Emergent system properties
  - performance, throughput, latencies
  - reliability, security, fault tolerance, evolvability
- Rationale
  - relates requirements and implementations
- Envelope of allowed change
  - “load-bearing walls”
  - design idioms and styles

---

**Desirable Properties: Software development lifecycle ....**

**How stable should a software architecture be?**

*Spiral model for software engineering....*

The architectural description should not change dramatically between releases of the same basic system.
Desirable Properties: *Consistency*

Consistency between the architectural description and the code?

For the architectural description to be useful, it is essential that there is a means to check and preserve consistency with the corresponding system(s).

Properties: *Variations and families*

Can architectures support replication, variation and product families?

Architectural descriptions should provide a generic structure from which various features and variations can be selectively included into particular system(s).
### Common Architectural Idioms

- **Data flow systems**
  - Batch/sequential
  - Pipes and filters
- **Call-and-return systems**
  - Main program & subroutines
  - Object-oriented systems
  - Hierarchical layers
- **Independent components**
  - Communicating processes
  - Event systems
- **Virtual machines**
  - Interpreters
  - Rule-based systems
- **Data-centered systems (repositories)**
  - Databases
  - Blackboards

... and more ...

### Architectural Description Languages (ADLs)

- More sound basis for describing and reasoning about software architecture.
- ADLs provide **constructs** for specifying architectural abstractions in a **descriptive notation**.
- Give mechanisms for decomposing a system into components and connectors and specifying how these elements are combined.
The State of Architectural Description

- Emerging realization of ADL value
- Proliferation of Architectural Description Languages
  - Acme: Generic ADL to support interchange
  - Aesop: style-specific environments
  - UniCon: architectural compilation
  - Wright: protocol analysis
  - CHAM: rewrite rule semantics
  - SADL: refinement patterns
- C-2: arch style using implicit invocation
- Darwin: distributed systems structure
- Meta-H: real-time, fault-tolerant avionics
- Rapide: event patterns, arch simulation

Darwin as an Architecture Description Language

- **Darwin** describes *structure*.
- **Darwin** architecture specification *independent* of component behaviour and component interaction.
- **Framework** for describing component behaviour, resource requirement, interaction type etc.
- **Darwin** used for *specification, construction* and *management*. 
**separation of concerns**

Separate:

- **Configuration**
  hierarchic structure of system from component instances & interconnections

- **Communication**
  component interaction mechanisms

- **Computation**
  component behaviour

---

**multi-view**

- **Structural View**

  ![Diagram of Structural View](image)

- **Behavioural View**

  ![Diagram of Behavioural View](image)

- **Performance View**

  ![Diagram of Performance View](image)

- **Service View**

  ![Diagram of Service View](image)
**Structural View - Components & Interfaces**

A component in Darwin can have one or more interfaces.

```
component C {
    portal a:1;
    b:1;
}
```

At this abstract level, an interface is simply a set of names:

```
interface I {
    x;
    y;
    z;
}
```

These will refer to actions in a specification or functions in an implementation.

**Structural View - Composites & Binding**

Composite components are constructed from more primitive components using `inst` - instantiation & `bind` - binding.

Portal types are inferred where they are not directly specified.

```
component COMP {
    portal a; b;
    inst p:C;
    q:C;
    bind p.a -- a;
    q.b -- b;
    p.b -- q.a;
}
```
**structural view - connectors**

Darwin, in contrast to Wright & Unicon, does not have additional syntax to denote connectors. A connector is simply a type of component:

```
component PIPE {
    portal put;
    get;
}
```

**service view - provide & require**

The service view refines a portal into either a service provided by a component or a service required by a component.

```
component FILTER {
    provide prev : IntStream;
    require next : IntStream;
    output : IntStream;
}
```
service view - towards implementation

In a distributed system, interfaces can be specified in Corba IDL and primitive components implemented as CORBA objects:

CORBA IDL

```java
interface IntStream {
    void put(in long x);
}
```

FILTER

service view - binding patterns

**many-to-one** (e.g. client - server)

- For replicated services only:

```java
inst c1.C;
c2.C;
S;
bind
c1.r -- S.p;
c2.r -- S.p;
```
replicators (forall) and guards (when)

Component PIPELINE(int n) {
  provide input: InStream;
  require output: OutStream;
  for all i = 0 to n-1 {
    inst [i].FILTER;
    bind [i].output -- output;
    when i < n-1
      bind [i].next -- [i+1];
  }
  bind input -- [0].prev;
}

Dimensioning & Variants

recursion

Component PIPELINE(int n) {
  provide input;
  require output;
  inst head: FILTER;
  bind
    input -- head.prev;
    head.output -- output;
  when n>1 {
    inst tail: PIPELINE(n-1);
    bind
      head.next -- tail.input;
      tail.output -- output;
  }
}
**generic components**

```cpp
component ELEM {
  provide prev;
  require next; output;
}

component PIPELINE(int n, <ELEM>) {
  provide input;
  require output;
  forall i = 0 to n-1 {
    inst f[i]:<ELEM>;
    bind f[i].output -- output;
    when i < n-1
      bind f[i].next -- f[i+1];
  }
  bind input -- f[0].prev;
}
```

**binding components**

Darwin components may contain only bindings. This is used to encapsulate complex interconnection patterns such as the perfect shuffle pattern shown below.

```cpp
component SHUFFLE(int n) {
  portal
    input[n];
    output[n];
  forall k: 0 to (n / 2)-1 bind
    input[k] -- output[k*2];
    input[k+(n/2)] -- output[k*2 + 1];
}
```
hierarchic bindings

A Darwin binding may result in many implementation bindings

Interfaces to composite components have no representation at runtime and consequently no runtime overhead. Elaboration of the Darwin program results in a flat structure of interconnected primitive component instances

Darwin - summary

Main Constructs

- `component` - declares a primitive or composite type.
- `interface` - declares an interface type
- `portal` - declares an interface instance
- `provide` - declares a service provided by a component.
- `require` - declares a service required by a component.
- `inst` - declares an instance of a component.
- `bind` - declares a binding from a requirement to a provision
**Darwin - summary**

**Additional Constructs**

- `forall` - replicates structure.
- `when` - guards structure.
- `dyn` - declares a set of dynamical created instances.
- `export` - exports a service into a namespace.
- `import` - imports a service from a namespace.

---

**Car Cruise Control - Example**

- An automobile cruise control system is controlled by the three buttons depicted above.
- When the engine is running and **on** is pressed, the cruise control system records the current speed and maintains the car at this speed.
- When the accelerator, brake or **off** is pressed, the cruise control system disengages but retains the speed setting.
- If **resume** is pressed, the system accelerates or deaccelerates the car back to the previously recorded speed.
Car Cruise Control - Example

**Design:**
1. the Design Model for the cruise control system.
2. the State Chart to describe the main control process for the cruise controller.
3. partition into concurrent interacting processes, specified in Darwin.

**Analysis:**
1. specify the behaviour of each of the processes which comprise the cruise control system.
2. compose the processes and analyse the behaviour of the system to check for validity.

*Will not do that yet - see later in the course!*

Cruise Control - Environment Context Diagram
Cruise Control - Partitioning into processes

Cruise Control - Structural View
Cruise Control - Process Architecture

```
component CruiseControlSystem {
  inst
    SensorScan;
    CruiseController;
    InputSpeed;
    SpeedControl;
    Throttle;
  bind
    SensorScan.sensors -- CruiseController.sensors;
    SensorScan.engine -- InputSpeed.engine;
    SpeedControl.throttle -- Throttle.setting;
    CruiseController.prompts -- SpeedControl.prompts;
    InputSpeed.speed -- SpeedControl.speed;
}
```

Beyond Architectural Structure

- What can be represented?
  - Behavior
    - computations of components
    - protocols of connectors
  - Performance
    - timing, computational demands, latencies, etc.
  - Reliability
- How can it be represented?
  - As associated properties
  - Architectural structure forms the skeleton; semantics forms the flesh
  - nMany possible notations
Analysis

We can check the system for deadlock and for different properties. *How do we know which properties to verify? ... what test cases to check for?*

**Requirements specification** may indicate desired and undesired properties and scenarios.

**Animation and experimentation** may indicate problems and properties to be preserved.

A Cruise Control problem?

We can start by composing the cruise control system and testing for **deadlock** (*we will learn how to do this later in the course*)

We can experiment by stepping through the system using different test scenarios...

- does it enable the system after engineOn and on is pressed?
- does it disable the system when the brake is pressed?
- does it enable the system when resume is pressed?
- does it disable the system when the engine is switched off?

**Safety:**

*If the cruise control system is in operation, then it is always disabled if the brake or accelerator is pressed, or if it is switched off, or if the engine is switched off.*
...and implementation

**Design analysis:** This was the process of gaining confidence in the particular Darwin structural architecture and component behaviour. Desired properties verified and the system exercised to ensure that it provided the required behaviour.

![Diagram showing the relationship between Sensor, Cruise Controller, Speed Control, and Throttle]

**Implementation:** The Darwin structural architecture can now be refined to provide the service view necessary for implementation and construction, possibly as a distributed program with components allocated to machines as desired. Primitive components would be provided with implementations, possibly as CORBA objects.

Experience - ARES (EU project)

**Architectural Reasoning for Embedded Systems**
- architectures to support families of systems

<table>
<thead>
<tr>
<th>Industrial partners</th>
<th>University partners</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nokia</td>
<td>Imperial College, London</td>
</tr>
<tr>
<td>Philips</td>
<td>T.U. Vienna</td>
</tr>
<tr>
<td>ABB</td>
<td>Polytechnic U. Madrid</td>
</tr>
</tbody>
</table>

Currently working closely with Nokia to restructure the software for GSM mobile telephones, and with Philips on adapting Darwin for design construction and evolution of software for televisions.
Architectural Languages... some other approaches

- **UniCon**  Components and connectors with a variety of built-in types & roles; static structures. Mary Shaw (CMU).
- **Wright**  Formal specification of components and connectors in CSP-like notation for interaction analysis. David Garlan (CMU).
- **ROOM**  Actors, message ports and bindings for real-time OO modelling; tool support for code generation. Bran Selic (ObjectTime™)
- **Rapide**  Interconnected components (interfaces) with behaviour specified as event sets for simulation and prototyping. David Luckham (Stanford)
- **...**  AT&T use architectural validation (checklists and formal reviews) to identify and resolve potential problems in the component and interaction structures. Joe Maranzano (AT&T).
- **Meta-H, C2, Polylith**  ... and others.....

What can we do today?  
(In Research Prototypes)

- Describe architectures precisely
  - about a dozen Arch Description Languages (ADLs)
  - usually with supporting toolkits
  - emphasis on describing structure
- Analyze architectural descriptions
  - instance-level analyses
  - style-wide properties
- Generate implementations
  - for highly constrained domains
  - usually by providing the "glue" code and handling packaging
Analyzing Architectural Instances

- Consistency
  - Do the parts fit together?
- Completeness
  - Are parts missing?
- System-wide behavior, performance, reliability, etc.
  - What is the aggregate behavior of a system, given the behaviors of the parts?
- Refinement and verification
  - Can one architecture be substituted for another?
  - Does an implementation conform to the architecture?
- Evaluating Design Choices

Summary

- **Software architecture** defines the essential structure of a software system.
- **Darwin** and other ADLs are used to specify a software architecture in terms of components and connectors.
- **Composition** of components, behaviors, … provides the means for design, analysis, …
- Architectures should strive to provide the essential stable description of system structure.
- Architectures can remain **consistent** with the actual system structure by construction.
- ADLs (and in particular, Darwin) are capable of supporting multiple views and variation.