Distributed Software

Distribution is inherent in the world
objects, individuals, ....

Interaction is inevitable with distribution.
computer communication, speech, ....

Interacting software components

Engineering distributed software?

- **Structure**
  Programming-in-the-small Vs Programming-in-the-large
deRemer and Kron, TSE 1975

- **Composition**
  “Having divided to conquer, we must reunite to rule”
  Jackson, CompEuro 1990

Our underlying philosophy

System structure as interacting components is a blessing. It directs software engineers towards compositional techniques which offer the best hope for constructing scalable and evolvable systems in an incremental manner.
Three Phases

- Explicit Structure
- Modelling
- Dynamic Structure

Phase 1. Explicit Structure

The National Coal Board project

The investigators:

The Research Assistant:

The mission:

Communications for computer control & monitoring of underground coalmining.

Coalmines

Underground coalmines consist of a number of interacting subsystems:

- coal cutting
- coal transport
- ventilation
- drainage ...

Model...
The research results

The mission:
- Communications for computer control & monitoring of underground coalmining.

The result:
- Software Architecture for control applications running on a distributed computing platform.

The solution had three major parts ...

Part I - components

Key property of context independence simplified reuse in the same system e.g. multiple pumps, and in different systems e.g. other mines.

- parameterised component types
- input and output ports

Part II - architecture description

Explicit description of the structure of the system in terms of the composition of component instances and connections.

Part III - “configuration programming”

Toolset and runtime platform support for:

- **Construction**
  Build system from software architecture description.

- **Modification/Evolution**
  On-line change to the system by changing this description.

We return to this later...
Benefits

- **Reusable components**
  The control software for a particular coalmine could easily and quickly be assembled from a set of components.

- **On-line change**
  Once installed, the software could be modified without stopping the entire system to deal with change - the development of new coalfaces.

---

Outcome I - coalmining

Underground coalmines consisted of deep shafts and long tunnels, for some of which there was to be no light at the end.

End of National Coal Board 1994

---

Outcome II - the Mine Pump example

Became a standard real-time systems example.

---

Outcome III - the CONIC system

- Wider application than coalmining.
- Distributed worldwide to academic and industrial research institutions.
- Conceptual basis lives on...

Research team:

Kevin Twidle
Naranker Dulay
Keng Ng

TSE 1989
Software Architecture

The fundamental architectural principles embodied in CONIC evolved through a set of systems and applications:

- **GIN & TONIC**
- **REX**
  - Reconfigurable & Extensible Distributed Systems
- **REGIS**
  - Distributed Services
- **Steve Crane** Location Services
- **Christos Karamanolis** Highly Available Services

Darwin - A general purpose ADL

- **Component types** have one or more interfaces. An interface is simply a set of names referring to actions in a specification or services in an implementation, **provided** or **required** by the component.

- **Systems / composite component types** are composed hierarchically by component **instantiation** and interface **binding**.

Koala

In the ARES project, **Rob van Ommering** saw potential of Darwin in specifying television product architectures and developed **Koala**, based on Darwin, for Philips.

First large-scale industrial application of an ADL.

Darwin applicability...

- Darwin enforces a strict separation between architecture and components.

- Build the software for each product variant from the architectural description of that product.

- Variation supported by both different Darwin descriptions and parameterisation.

- Variants can be constructed at compile-time or later at system start-time.
What we could not do...

In advance of system deployment, answer the question:

Will it work?

When faced with this question engineers in other disciplines build models.

Engineering Models

- Abstract
- Complexity Model \ll \text{System}
- Amenable to Analysis
Architecture & Models

Modelling technique should exploit structural information from S/W architecture.

Use process calculus FSP in which static combinators capture structure and dynamic combinators component behaviour.

<table>
<thead>
<tr>
<th>Darwin</th>
<th>FSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>• instantiation inst</td>
<td>• instantiation ;</td>
</tr>
<tr>
<td>• composition</td>
<td>• parallel composition</td>
</tr>
<tr>
<td>• binding bind</td>
<td>• relabelling /</td>
</tr>
<tr>
<td>• interfaces</td>
<td>• sets and hiding @</td>
</tr>
</tbody>
</table>

FTDCS 1997, WICSA 1999

Process Calculus - FSP

\[
\text{PUMP} = \text{STOPPED}, \\
\text{STOPPED} = (\text{cmd}.\text{start} \rightarrow \text{STARTED}), \\
\text{STARTED} = (\text{pump} \rightarrow \text{STARTED} \\
| \text{cmd}.\text{stop} \rightarrow \text{STOPPED}) .
\]

\[
| | \text{P} _C = (\text{CONTROL} | | \text{PUMP}) @((\text{level}, \text{pump}).)
\]

Analysis - LTSA

What questions can we ask of the behaviour model?

- fluent \( \text{RUNNING} = <\text{start},\text{stop}> \)
- fluent \( \text{METHANE} = <\text{methane.high}, \text{methane.low}> \)

- assert \( \text{SAFE} = [\square](\text{METHANE} \rightarrow !\text{RUNNING}) \)

Model...

- assert \( \text{SAFE} = [\square](\text{tick} \rightarrow (\text{METHANE} \rightarrow !\text{RUNNING})) \)

Contributors...

- Shing-Chi Cheung - Safety
- Dimitra Giannakopoulou - Progress & Fluent LTL
- Nat Pryce - Animation

Engineering distributed software

Models
- Mathematical Abstractions
  - reasoning and property checking

Systems
- Compositions of subsystems
  - built from proven components.

S/W Tools
- Automated techniques and tools
  - construction and analysis

Phase 3. Dynamic Structure

"dynamic structure"

Managed Structural Change

Structural change
- **load**
  - component type
- **create/delete**
  - component instances
- **bind/unbind**
  - component services

But how can we do this safely?
Can we maintain consistency of the application during and after change?

TSE 1985
General Change Model

**Principle:**
Separate the specification of structural change from the component application contribution.

A Passive component
- is consistent with its environment, and
- services interactions, but does not initiate them.

**Component States**

PASSIVE

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pre-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>- component is quiescent and isolated</td>
</tr>
<tr>
<td>bind/unbind</td>
<td>- connected component is quiescent</td>
</tr>
<tr>
<td>create</td>
<td>- true</td>
</tr>
</tbody>
</table>

**Change Rules**

Quiescent - passive and no transactions are in progress or will be initiated.

**Required Properties (1)**

// node is PASSIVE if passive signalled and not yet changing or deleted
fluent PASSIVE[i:Nodes]
= <node[i].passive,
  node[i].{change[Value],delete}>

// node is CREATED after create until delete
fluent CREATED[i:Nodes]
= <node[i].create, node[i].delete>

// system is QUIESCENT if all CREATED nodes are PASSIVE
assert QUIESCENT
= forall[i:Nodes] (CREATED[i]->PASSIVE[i])
Required Properties (2)

// value for a node i with color c
fluent VALUE[i:Nodes][c:Value]
  = <node[i].change[c], ...>

// state is consistent if all created nodes have the same value
assert CONSISTENT
  = exists[c:Value] forall[i:Nodes] (CREATED[i] -> VALUE[i][c])

// safe if the system is consistent when quiescent
assert SAFE = [] (QUIESCENT -> CONSISTENT)

// live if quiescence is always eventually achieved
assert LIVE = [] <> QUIESCENT

Self Organising Architecture

A self-organising architecture is both self-assembling and self-healing.

Self-assembling - initially, a set of component instances organise their interaction to satisfy architectural specification.

Self-healing - components collaborate to satisfy required architectural properties after failure/change in the environment.

Objective is to minimise explicit management

Self Assembling

Ordered Ring Architecture

Self Assembling

Ordered Ring Architecture
Self Healing

Ordered Ring Architecture

Component Model

Attributes

Provided Services (ports)

Required Services (ports)

Self Organising Architecture Specification

Architecture is specified by a set of constraints on structure and attribute values.

A new component must ensure that constraints remain satisfied.

Ordered Ring Constraints
- a rcv service has exactly one snd service bound to it.
- all ring components form a single chain
- component id < successor id unless max(id).

Research Challenges

We have some of the pieces, but need ...

- Scalable decentralised implementation.
- Analysis tools
- Capability to update constraints for operational system
In conclusion...

Architecture as a structural skeleton ....

...so that the same simple architectural description can be used as the framework to compose behaviours for analysis, to compose component implementations for systems, ....

Darwin support for multiple views

Structural View

Behavioural

Performance View

Service View

Analysis

Construction/implementation

Research into practice...

- Application
- Education...
- Further research...
Education...

Further research...

- Model synthesis from scenarios
- Model synthesis from goals
- Probabilistic performance models
- Self-organising Architectures

Research voyage of discovery

*Has been a lot of fun and is far from over :-)*

Sebastian Uchitel
Emmanuel Letier