Chapter 1. Once upon a time.....
Distribution is inherent in the world

objects, individuals, 

Interaction is inevitable with distribution.

computer communication, speech, 

Interacting software components
### Distributed software engineering?

<table>
<thead>
<tr>
<th>Models</th>
<th>Mathematical Abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- reasoning and property checking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prototypes</th>
<th>Simplified implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- property checking in practice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Systems</th>
<th>Compositions of subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>built from proven components.</td>
</tr>
</tbody>
</table>
Chapter 2. Early experience.....

“configuration programming”

CONIC
The first generation of distributed software engineers
British Coal - automation and remote monitoring

Mine Drainage Pump:
The installation is used to pump to the surface the water that collects in a sump at the bottom of a mine shaft.

Diagram:
- To surface
- Methane sensor
- Airflow sensor
- Carbon Monoxide sensor
- To surface control room
- Environment Monitoring Station
- Pump Control Station
- Pump
- Sump
- High water level detector
- Low water level detector
CONIC - a basis for “configuration programming”

Separation of concerns - separate structure from component programming

- Configuration Language
to express structure

- Dynamic configuration

- Software tools -
  compilers, checkers,
  run-time environment, graphical display
Structural view - for construction and evolution

components + configuration description

Construction/implementation

system

evolved configuration description

change script

changes

evolved system
CONIC was widely distributed to academic and industrial research establishments.

Universities in UK, Germany, France, Belgium, Sweden, Canada, Japan, Korea.

Industries such as British Coal, British Petroleum, British Telecom, GEC.
Experience

... and used for a wide range of applications ....

- Underground monitoring and communications
- Multi-loop self-tuning adaptive controllers
- Process control plant automation
- Parallel algorithms - FFT
- Image processing
- Programming environments
- Teaching distributed programming
Distributed software engineering?

<table>
<thead>
<tr>
<th>Models</th>
<th>Mathematical Abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- reasoning and property checking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prototypes</th>
<th>Simplified implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- property checking in practice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Systems</th>
<th>Compositions of subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>built from proven components.</td>
</tr>
</tbody>
</table>
## Main Principle of Configuration Programming

| Description and Construction | “The *configuration language* used for structural description should be *separate* from the *programming language* used for programming components.” |
| Can we apply this experience to .... |

| Modelling and Analysis? | “The *configuration language* used for structural description should be *separate* from the *specification language* used for *modelling* component behaviour.” |
Chapter 3. A simple problem
Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.
Dining Philosophers implementation
Deadlock

The Dining Philosophers program deadlocks when every philosopher has obtained his right fork. No philosopher can obtain his left fork and so no progress can be made.

They are waiting for a condition that will never become true (i.e. left fork becoming free).
Could we have modelled this and predicted that deadlock was possible?

If we propose a way to correct it, can we be sure that we have corrected it for all possible situations?
Chapter 4. Darwin, an architecture description language
Darwin support for multiple views

- Structural View
- Behavioural View
- Service View

Analysis

Construction/implementation
A component in Darwin can have one or more interfaces. At this abstract level, an interface is simply a set of names. These refer to actions in a specification or functions in an implementation.

```
component PHIL {
  portal right:I ;
  left:I ;
}

interface I {
  get ;
  put ;
}
```
Composite components are constructed from more primitive components using inst - instantiation and bind - binding.

```plaintext
component PAIR {
    portal right; left;
    inst plato : PHIL;
    fork : FORK;
    bind plato.right -- right;
    fork.left -- left;
    plato.left -- fork.right;
}
```
Darwin architecture description language (ADL)

- Darwin describes *structure*.
- Darwin architecture specification is *independent* of component behaviour and component interaction.
- Darwin provides a *framework* for describing system *construction* and *behaviour*. 
Component: PHIL

Process specification:

PHIL = (think
-> right.get -> left.get
-> eat
-> right.put -> left.put
-> PHIL ).

Action prefix: ->
PHIL Labelled Transition System - LTS

0 ➔ think
1 ➔ right.get
2 ➔ left.get
3 ➔ eat
4 ➔ right.put
5 ➔ left.put
philosopher component - animation, trace and LTS
Component:

\[
\text{FORK} = ( \text{left.get} \to \text{left.put} \to \text{FORK} \\
\quad \mid \text{right.get} \to \text{right.put} \to \text{FORK} )
\]
FORK Labelled Transition System - LTS

Diagram:

- Node 0
  - Transitions:
    - right.get
    - right.put
    - left.put

- Node 1
  - Transitions:
    - left.get

- Node 2
  - Transitions:
    - right.get
fork component - LTS animation, trace and LTS
Primitive Component - summary

- **Component** behaviour is modelled by a finite state process (LTS) using:
  - action prefix
  - choice
  - guarded recursion

- **Portal** interface represents an action (or set of actions) in which the component can engage.
composite component behaviour

parallel composition: ||

relabel: /

portal interface alphabet: @

PAIR = (plato:PHIL || fork:FORK)

/{right/plato.right,
left/fork.left,
plato.left/fork.right}

{@{right,left}.}
Composite Component - summary

- Composition in Darwin is modelled as parallel composition $|$ $|$.  
  (Interleaving of all the actions)

- Binding in Darwin is modelled by relabelling $\backslash$ $\slash$.  
  (Processes synchronise on actions that they have in common)

- Composition expressions are direct translations from Darwin architecture descriptions.
Darwin composition of the dining philosophers

```plaintext
component  DINERS(int N=5){
    forall i=0 to N-1 {
        inst
        phil[i] : PHIL;
        fork[i] : FORK;

        bind
        phil[i].left -- fork[i].right;
        phil[i].right --
        fork[((i-1)+N)%N].left;
    }
}
```
DINERS Specification

||DINERS (N=5) =

(phil[0..N-1]:PHIL || fork[0..N-1]:FORK )

/{phil[i:0..N-1].left / fork[i].right,

phil[i:0..N-1].right /

fork[((i-1)+N)%N].left

}.

State Space:

6 * 6 * 6 * 6 * 6 * 3 * 3 * 3 * 3 * 3 = 1889568
Chapter 6. Behaviour analysis
Reachability analysis

Searches the system state space for deadlock states and error states arising from property violations.

A deadlock occurs when the system enters a state with no outgoing transitions:
Dining philosophers - analysis

Composing potential DEADLOCK..

States Composed: 2163 Transitions: 8770 in 1760ms

Trace to DEADLOCK:

phil.0.think
phil.0.right.get
phil.1.think
phil.1.right.get
phil.2.think
phil.2.right.get
phil.3.think
phil.3.right.get
phil.4.think
phil.4.right.get
Deadlock Avoidance

Perhaps deadlock could be avoided in the Dining Philosophers system by making one of the philosophers pick up his forks in the reverse order?

```
PHIL(I=0) = (think \rightarrow PHILcheck),
PHILcheck =
    ( when I==0  left.get \rightarrow right.get \rightarrow EAT
    | when !I==0  right.get \rightarrow left.get \rightarrow EAT),
EAT =(eat \rightarrow right.put \rightarrow left.put \rightarrow PHIL).
```

(i.e. phil.0 gets left before right).
analysis

||DINERS (N=5) =
  (phil[i:0..N−1]:PHIL(i) || fork[0..N−1]:FORK)
/

States Composed: 2163 Transitions: 8770 in 1870ms
No deadlocks/errors
Specifying properties

Safety properties are specified by deterministic finite state processes called property automata.

```
property NOGLUTTONY =
  (phil[i:0..4].eat ->
   (when i>0 phil[j:0..i-1].eat -> NOGLUTTONY
    |when i<4 phil[j:i+1..4].eat -> NOGLUTTONY)
  ).
```

The property NOGLUTTONY asserts that if a philosopher \( i \) eats, then one of the other philosophers \( j \) eats next (philosopher \( i \) does not eat twice in succession).
checking properties

Composing property NOGLUTTONY violation..........
States Composed: 9768 Transitions: 39703 in 17740ms
Trace to property violation in NOGLUTTONY:

phil.0.think
phil.0.left.get
phil.0.right.get
phil.0.eat
phil.0.right.put
phil.0.left.put
phil.0.think
phil.0.left.get
phil.0.right.get
phil.0.eat
Ring of Dining Philosophers using PAIRs?

\[
||PAIR = (plato:PHIL || fork:FORK) \\
/\{right/plato.right, \\
    left/fork.left, \\
    plato.left/fork.right\} \\
@\{right, left\}.
\]
Ring of PAIRs

pair[0] PHIL FORK PHIL

pair[1] FORK PHIL

pair[2] PHIL FORK

pair[3] FORK PHIL

pair[4] PHIL FORK
DINERSPairs Specification

```
| | | DINERSPairs(N=5) = (pair[0..N-1]:PAIR) / {pair[i:0..N-1].right / pair[((i-1)+N)%N].left }.
```

Composing
potential DEADLOCK
States Composed: 82 Transitions: 265 in 50ms
Trace to DEADLOCK:

```
pair.0.right.get
pair.1.right.get
pair.2.right.get
pair.3.right.get
pair.4.right.get
```
Chapter 7. Another example....
For safety reasons, the pump must not be started or continue running when the percentage of methane in the atmosphere exceeds a safe limit.
pump control system

\[ \text{PUMPSTATION} = (\text{PUMP} \ | \ \text{LEVEL} \ | \ \text{PUMPCONTROL}) \]

\[ \text{MINESYSTEM} = (\text{PUMPSTATION} \ | \ \text{OP} \ | \ \text{METH}) \]
Methane safety property

Test that the pump is stopped if the methane level reaches the limit when the pump is running (started).
Property analysis

Composition:
\[
PUMPSTATION = PUMP \parallel LEVEL \parallel PUMPCONTROL
\]
Composition:
\[
MINESYSTEM = PUMPSTATION \parallel OP \parallel METH
\]
Composition:
\[
MINESYSTEMtest = MINESYSTEM \parallel METHANE
\]
State Space:
\[
112 \times 4 = 448
\]
Composing
States Composed: 160 Transitions: 640 in 160ms
No deadlocks/errors
Action hiding and minimisation

\[ \text{MINESYSTEM}_{\text{hide}} = \text{MINESYSTEM} \]
\[ @ \{\text{safe, limit, high, low, enable, disable, start, stop}\}. \]

Result.....

Composing
States Composed: 112 Transitions: 424 in 110ms
minimised in 550ms
Minimised States: 9
Scalability

The problem with reachability analysis is that the state space “explodes” exponentially with increasing problem size.

How do we hope to alleviate this problem?

Compositional Reachability Analysis:
We construct the system incrementally from subcomponents, based on the software architecture. State reduction is achieved by hiding actions not in their interfaces and minimising.
### Distributed software engineering?

<table>
<thead>
<tr>
<th>Models</th>
<th>Mathematical Abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- reasoning and property checking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prototypes</th>
<th>Simplified implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- property checking in practice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Systems</th>
<th>Compositions of subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>built from proven components.</td>
</tr>
</tbody>
</table>
Chapter 8. Some Lessons
Automated software tools are essential to support software engineers in the design process. Techniques which are not amenable to automation are unlikely to survive in practice.
Software Technology - the need for teams

Software technology research necessarily involves both theory and practice, in the form of experimental implementations.

This is best conducted by small teams of researchers with a shared vision.
The next generation of distributed software engineers
The need to strive for Clarity and Simplicity

“It has been my experience with literary critics and academics in this country, that clarity looks a lot like laziness and ignorance and childishness and cheapness to them. Any idea which can be grasped immediately is for them, by definition, something they knew all the time.”

Kurt Vonnegut
What is the skeleton in the software cupboard?

Software architecture is the overall structure of a system in terms of its constituent components and their interconnections. It can be used to provide the skeleton upon which to flesh out the particular details of concern.
Software Architecture

For system **construction**, we can associate implementations with the components of the architecture.

For **analysis**, we can associate behavioural descriptions with the components and reason about the behaviour of systems composed of these components according to the architecture.
View consistency

Systems developed in this way have an explicit structural skeleton which, being shared, helps to maintain consistency between the system and the various elaborated views.
Time to come out of the software cupboard!