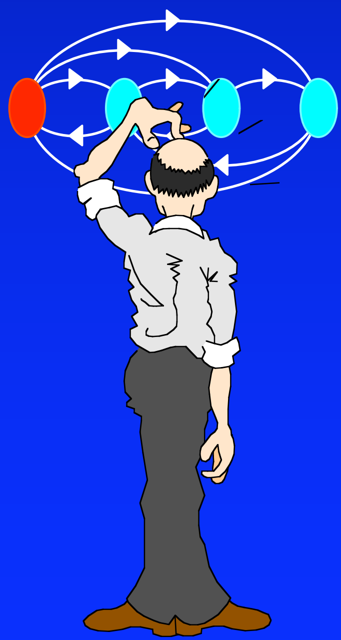
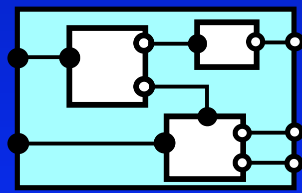
# Distributed Software Engineering: an Architectural Approach




**Jeff Magee**

Distributed Software Engineering
Department of Computing
Imperial College
London

Work conducted with
my close colleague,
Jeff Kramer

# 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

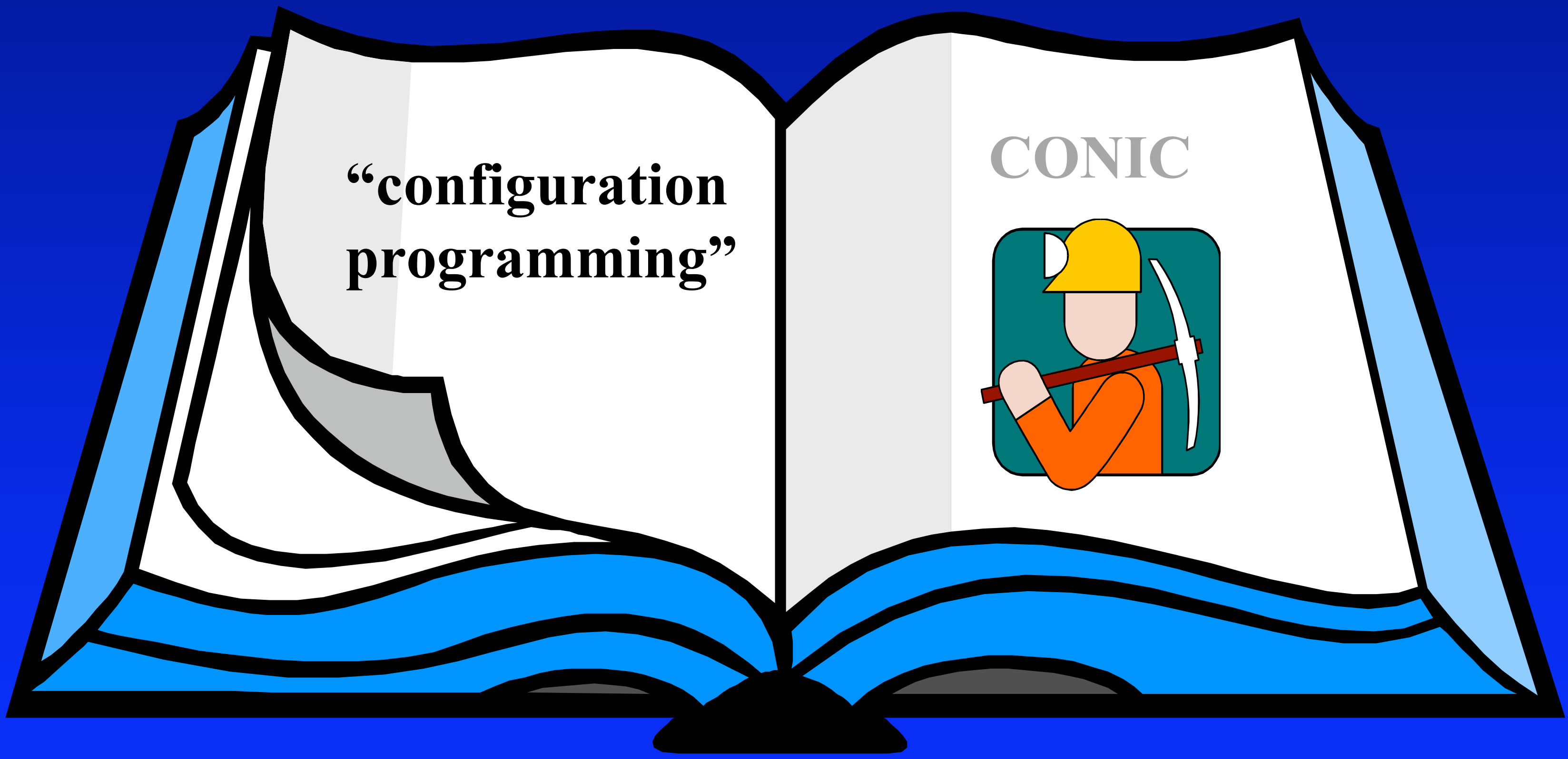
## Our underlying philosophy

*A focus on system structure as interacting components is essential for all complex systems. 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:

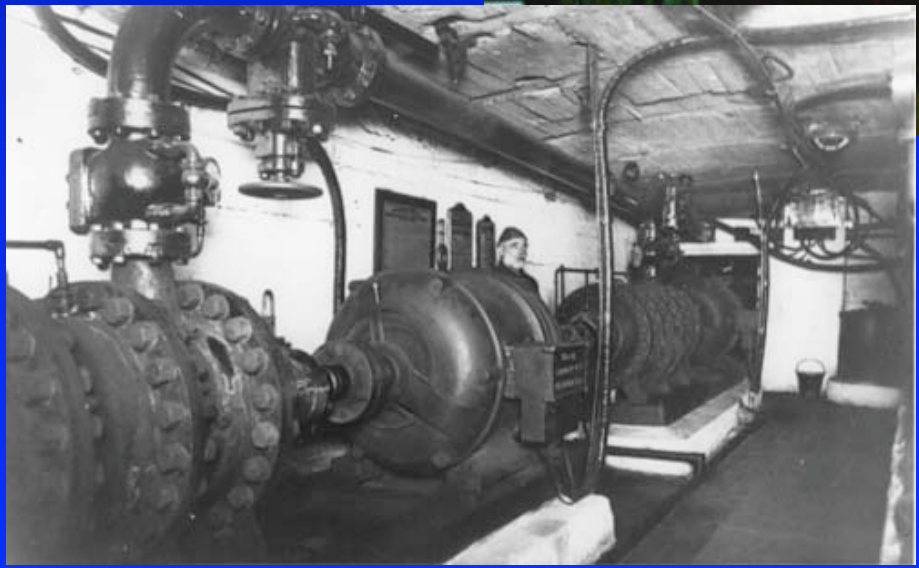
The mission:

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:

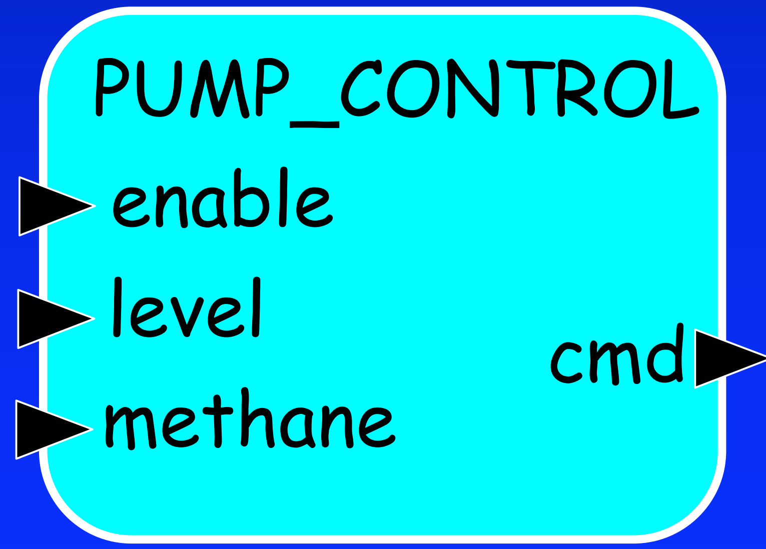
- Software Architecture for control applications running on a distributed computing platform.

*The solution had three major parts …*

# Part I - components

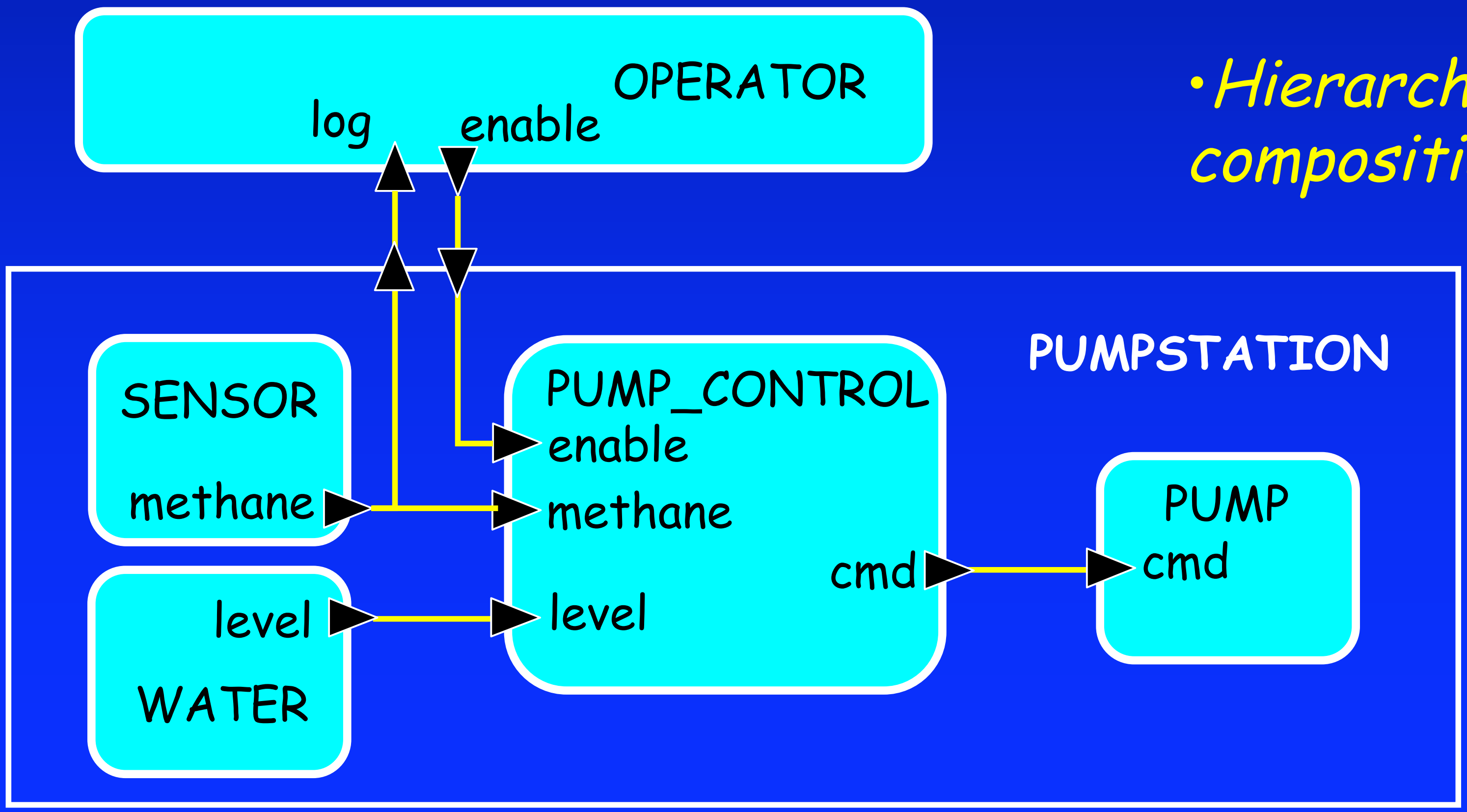
Key property of context independence simplified reuse in the same system e.g. multiple pumps, and in different systems e.g. other mines.

- *parameterised component types*
- *input and output ports*



# Part II - architecture description

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




- *Hierarchical composition*

## 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…*

TSE 1985, CompEuro 1990

## 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.

*Final outcome…*

## Outcome - 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

# Software Architecture

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

REGIS
Distributed Services

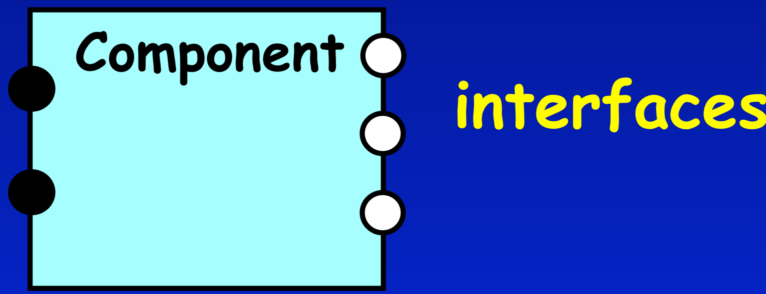
Steve Crane
Ulf Leonhardt
Location Services

Christos Karamanolis
Highly Available Services

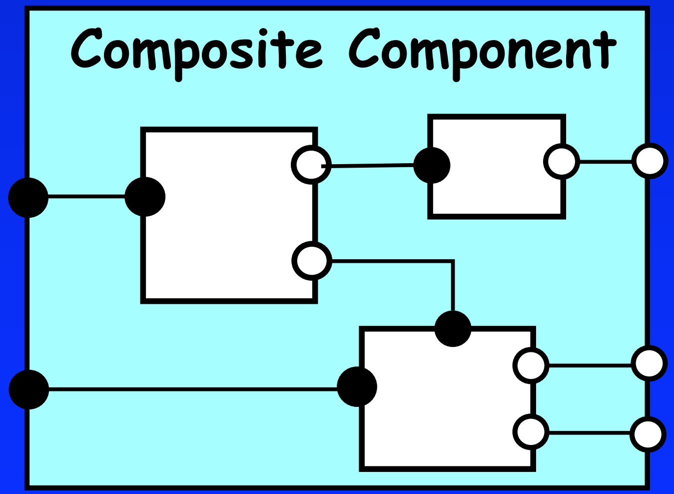
Parle 1991, SEJ 1992, DSEJ 1994

# 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

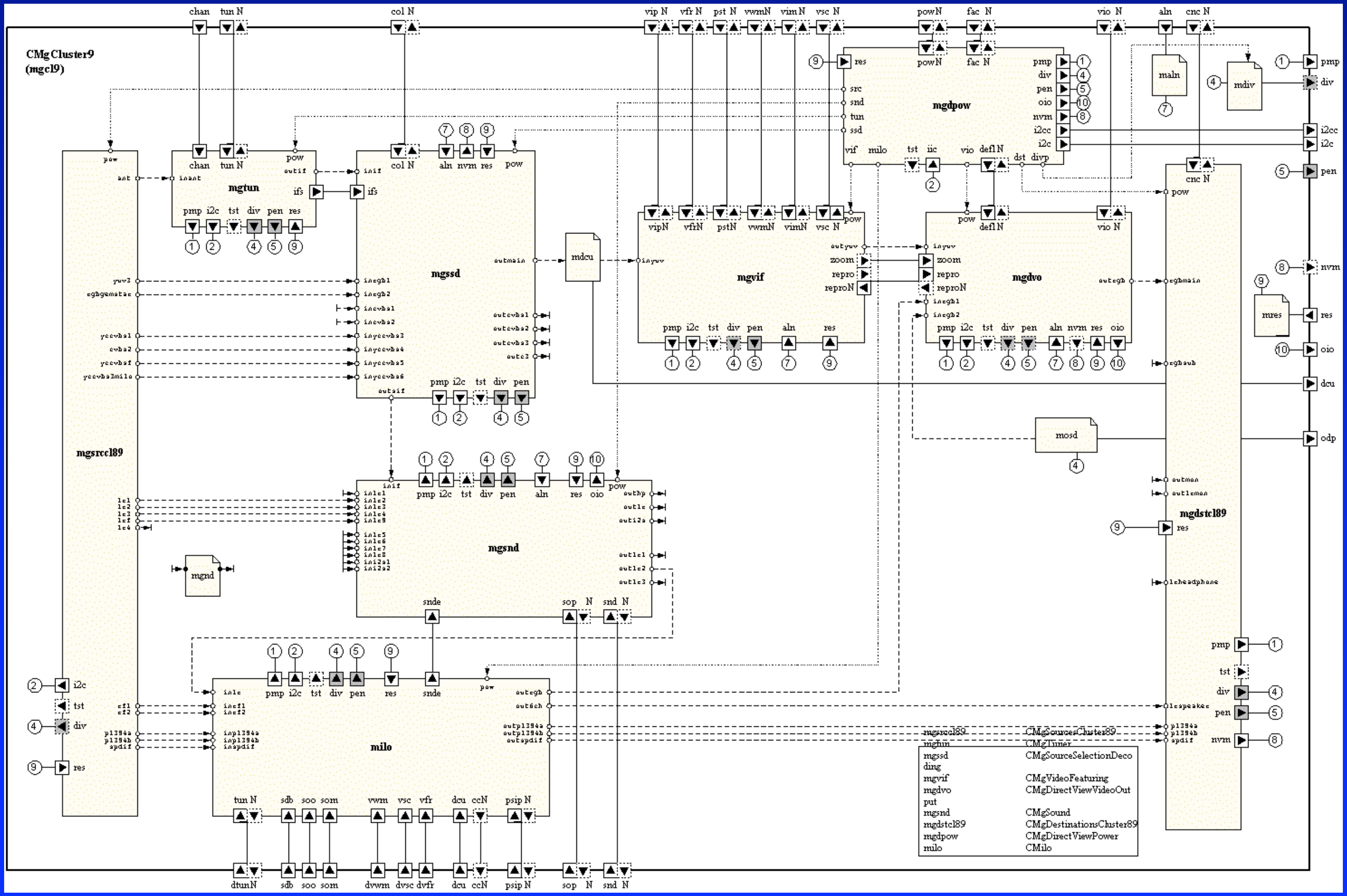
In the ARES project Rob van Ommering saw potential of Darwin in specifying television product architectures and developed Koala, based on Darwin, for Philips.

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.

# Koala - example



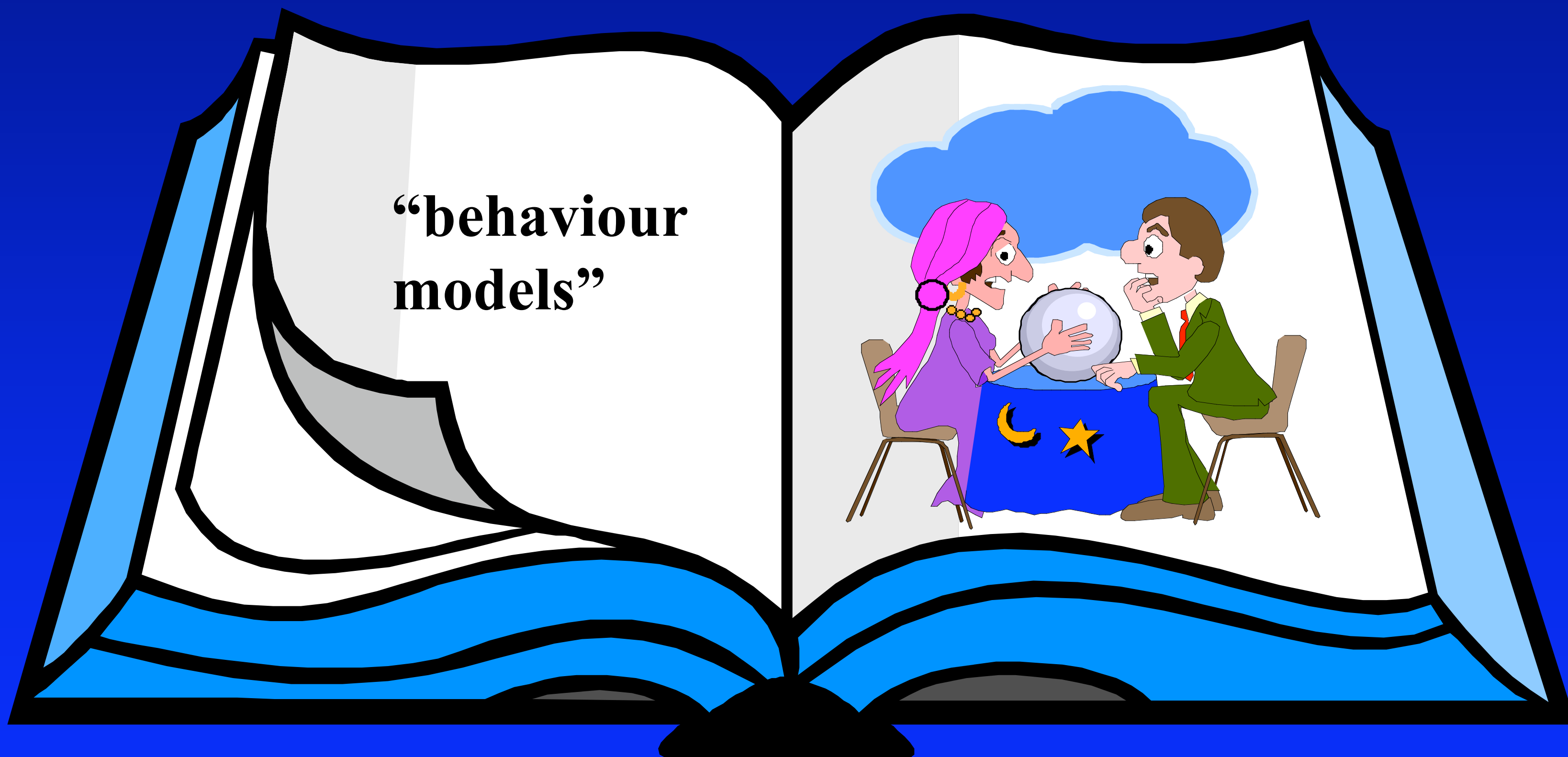
## What we could not do…

In advance of system deployment, answer the question:

**Will it work?**

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

# Phase 2. Modelling



# Engineering Models

- Abstract
- Complexity
  Model << System
- Amenable to Analysis

# Architecture & Models

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

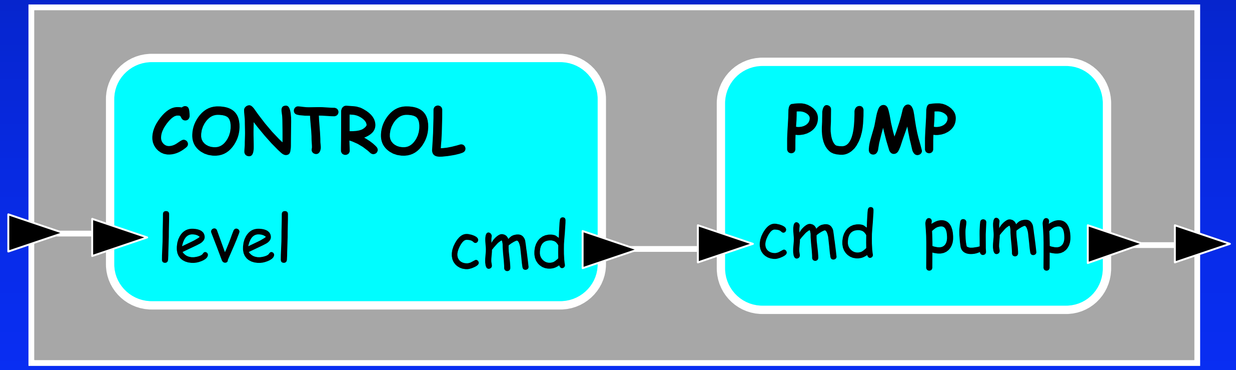
→ Use process calculus FSP in which static combinators capture **structure** and dynamic combinators **component** behaviour.

| **Darwin** | | **FSP** | |
|---|---|---|---|
| instantiation | **inst** | instantiation | **:** |
| composition | | parallel composition | **\|\|** |
| binding | **bind** | relabelling | **/** |
| interfaces | | sets and hiding | **@** |

FTDCS 1997, WICSA 1999

# Process Calculus - FSP

```
PUMP = STOPPED,
STOPPED = ( cmd.start -> STARTED),
STARTED = ( pump -> STARTED
        | cmd.stop -> STOPPED
        ).
```




```
||P_C = (CONTROL || PUMP)@{level,pump}.
```

# Analysis - LTSA

What questions can we ask of the behaviour model?

```
fluent RUNNING = <start,stop>
fluent METHANE = <methane.high, methane.low>

assert SAFE = [](tick->(METHANE -> !RUNNING))
```

Model…

# Contributors...

Shing-Chi Cheung
- LTS, CRA & Safety

Dimitra Giannakopoulou
- Progress & Fluent LTL

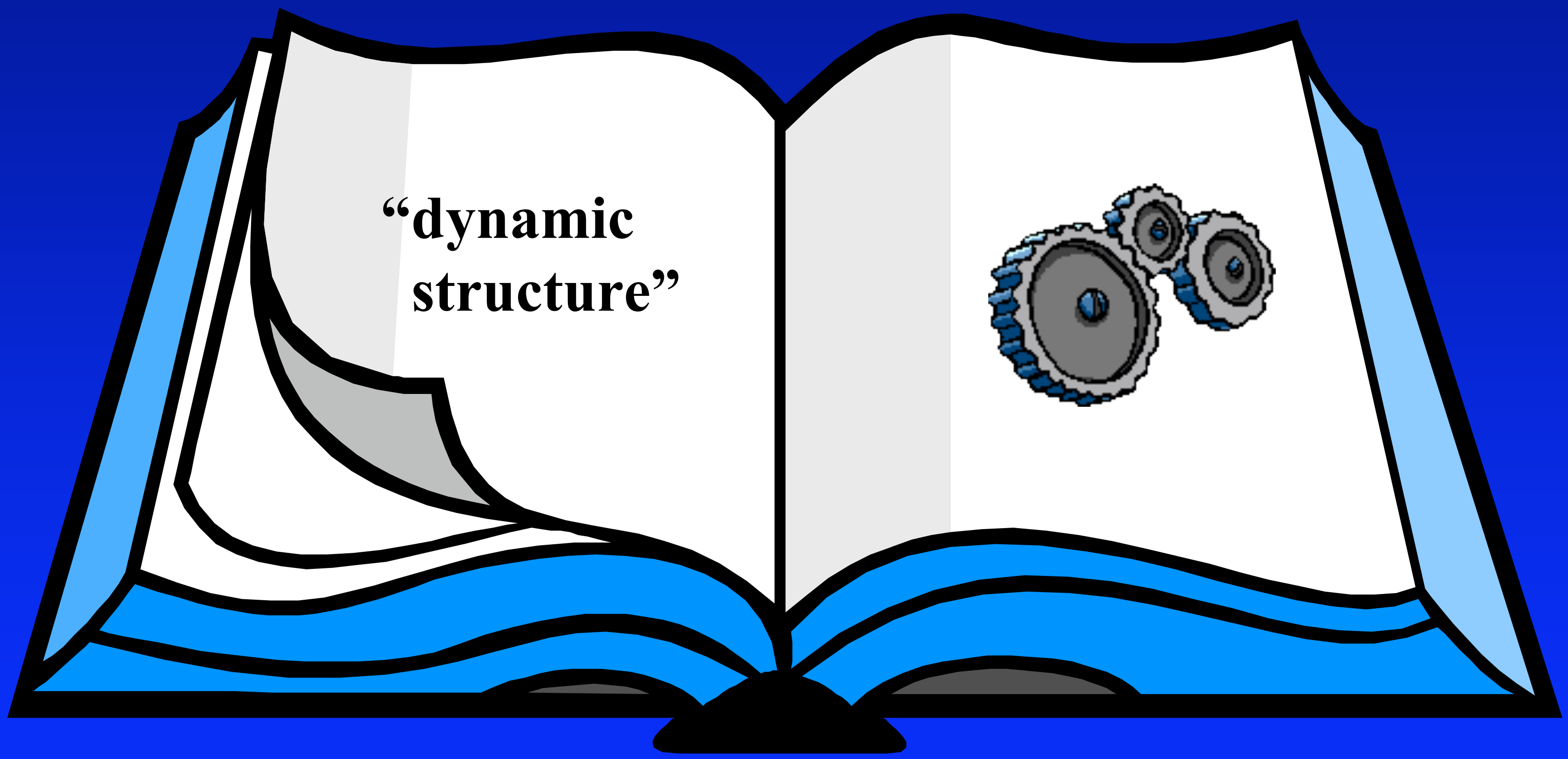
Nat Pryce
- Animation

ICSE 1996, FSE 1999, ICSE 2000, ESEC/FSE 2003

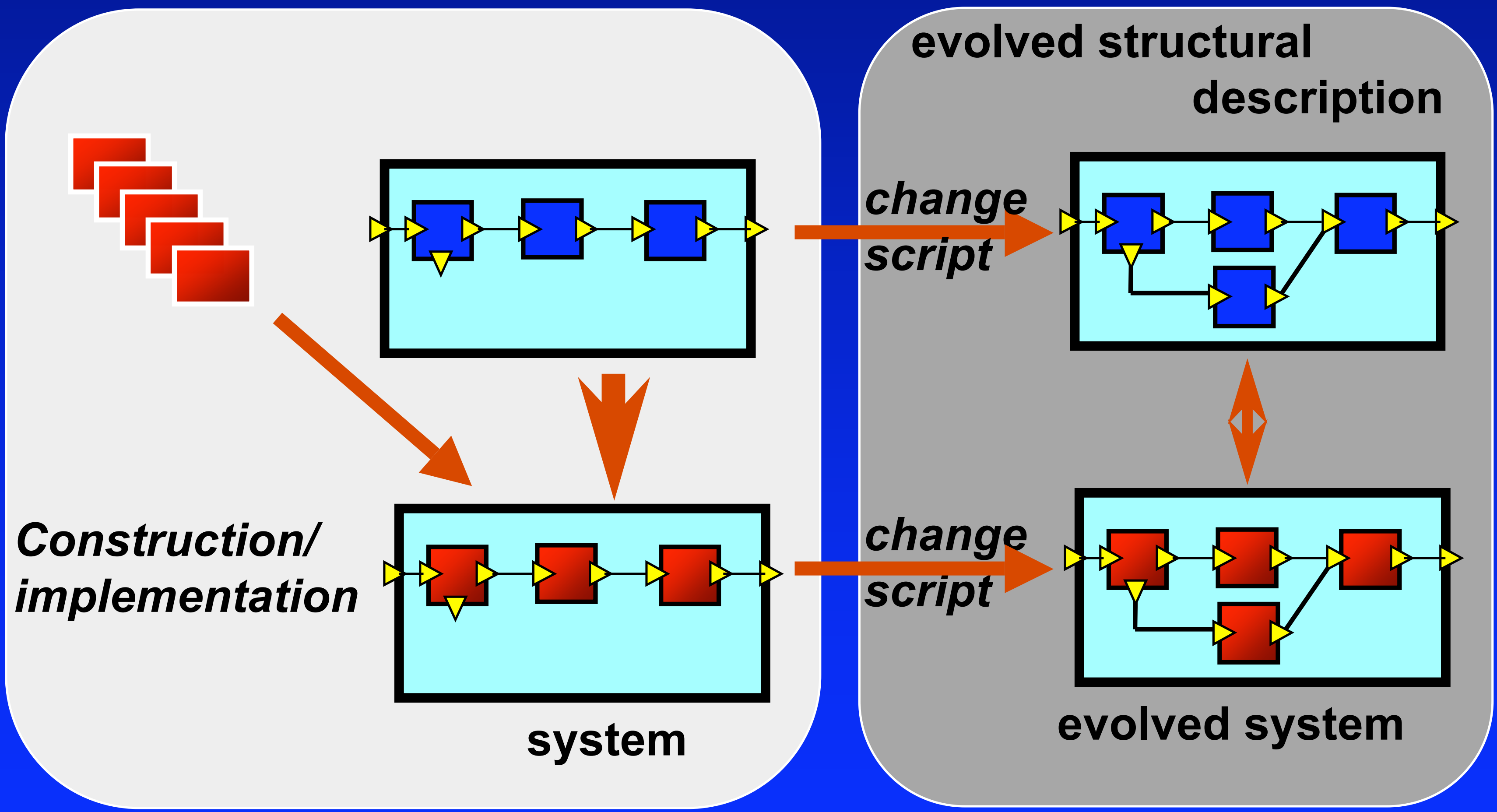
# Engineering distributed software

| | |
|---|---|
| **Models** | Mathematical Abstractions<br>- reasoning and property checking |
| **Systems** | Compositions of subsystems<br>- built from proven components. |
| **S/W Tools** | Automated techniques and tools<br>- construction and analysis |

# Phase 3. Dynamic Structure



# Managed Structural Change

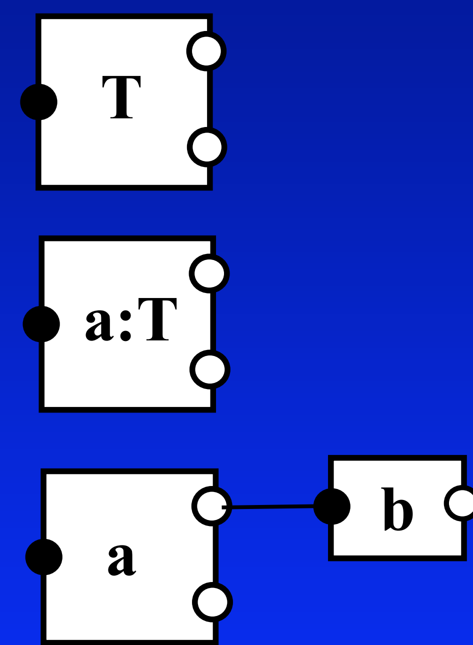



*e.g. Conic, Regis*

TSE 1985

# Structural change

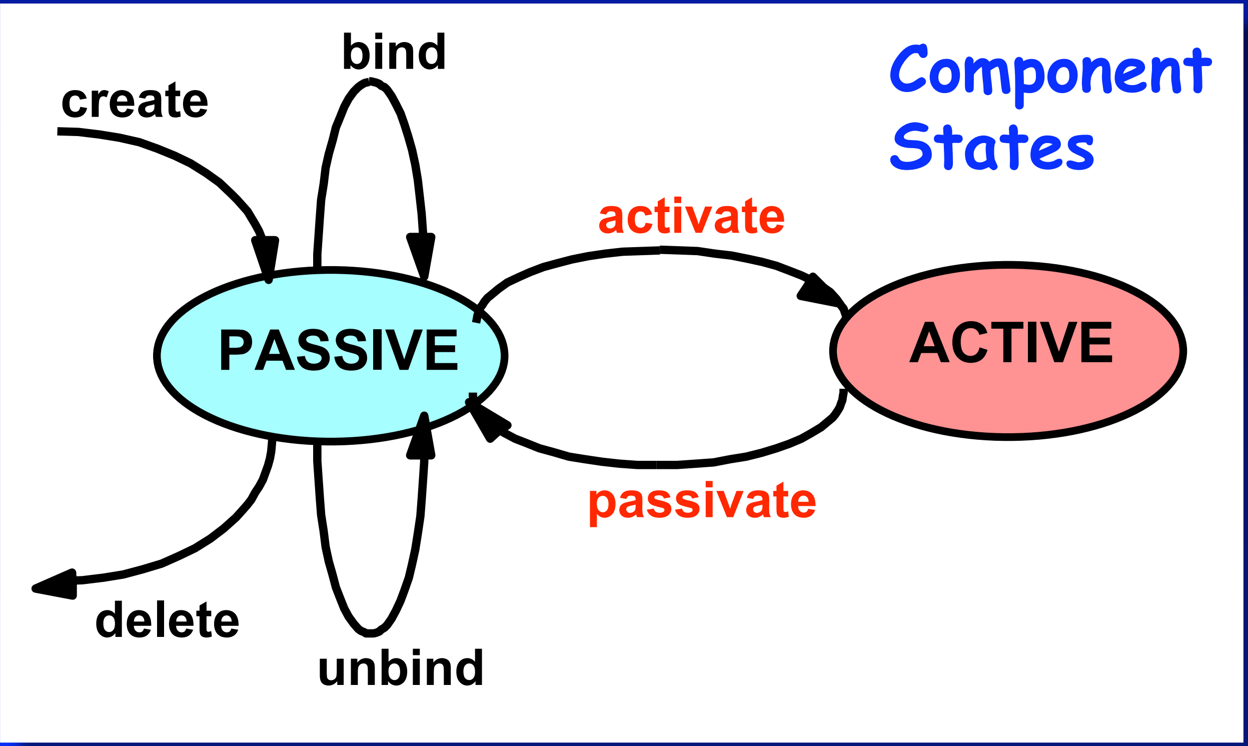
- **load**
  component type
- **create/delete**
  component instances
- **bind/unbind**
  component services




But how can we do this safely?

Can we maintain consistency of the application during and after change?

# General Change Model




**Principle**:

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

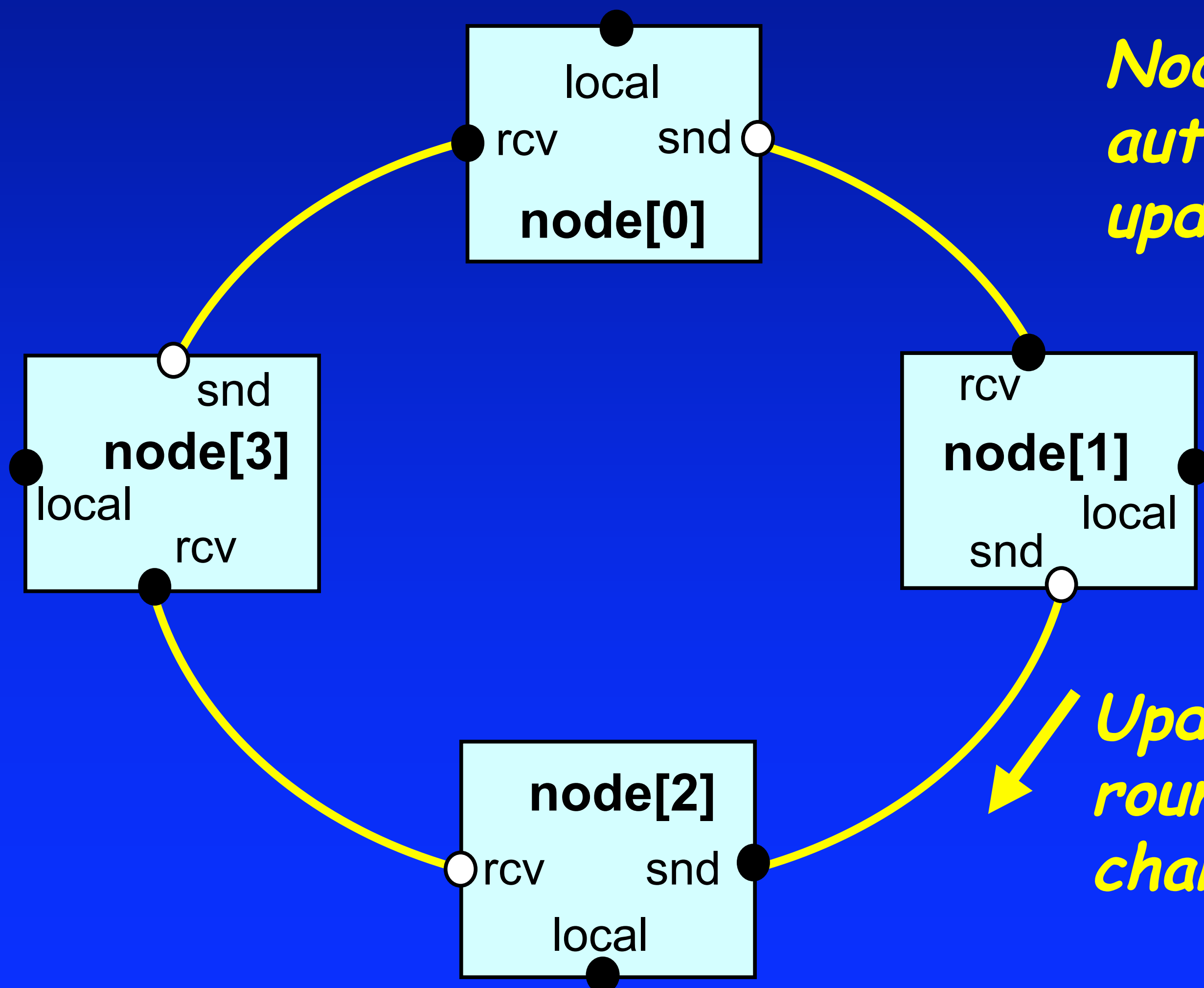
A **Passive** component
- is consistent with its environment, and
- services interactions, but does not initiate them.

## Change Rules

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

| Operation | Pre-condition |
| --- | --- |
| **delete** | – component is quiescent and isolated |
| **bind/unbind** | – connected component is quiescent |
| **create** | - true |

# Example - a simplified RING Database




***Nodes perform autonomous updates***

***Updates propagate round the ring via channels***

CDS 1998, IEE Proc 1998

# 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*

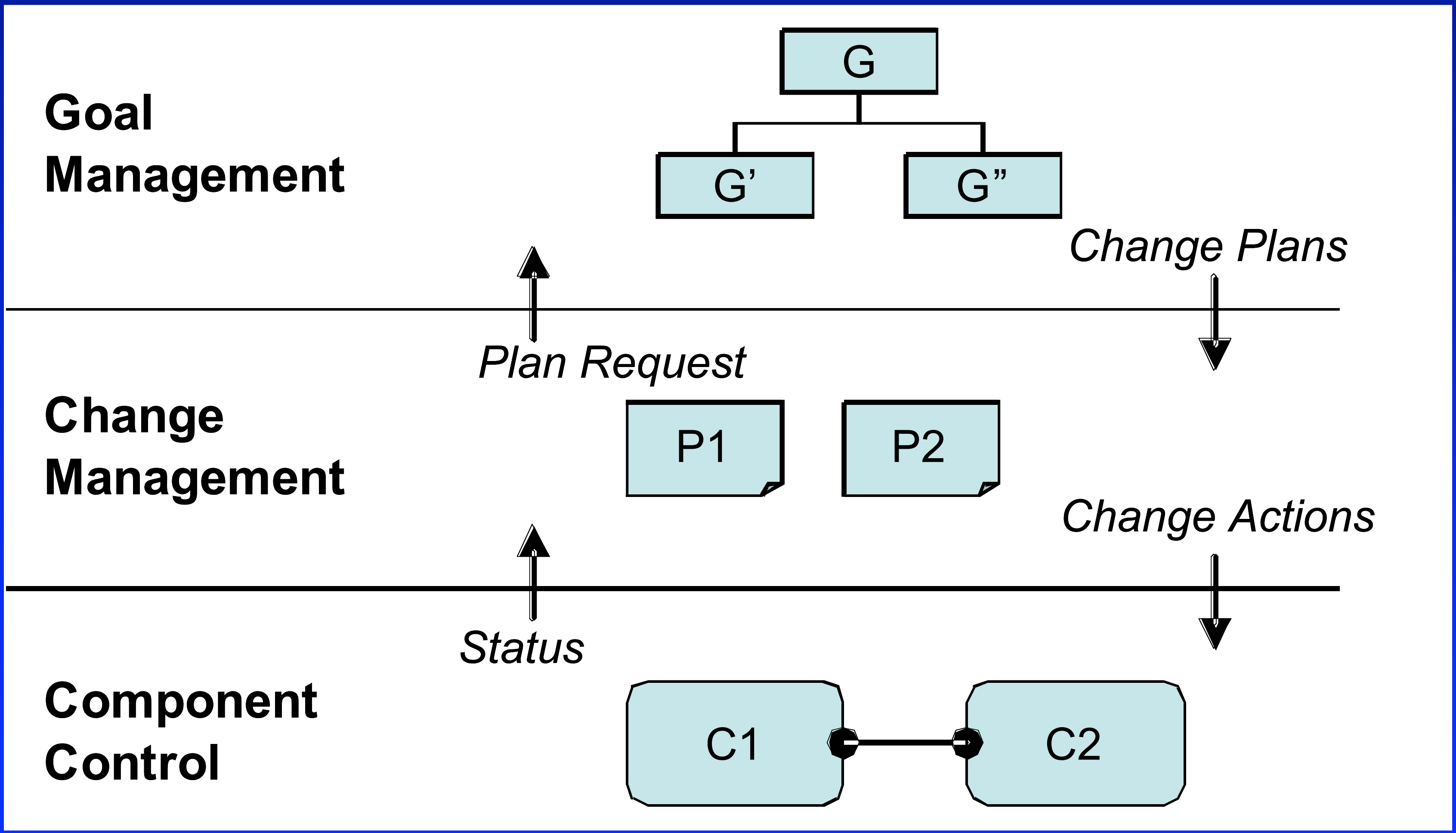
```
assert LIVE = []<> QUIESCENT
```

# Software Architecture for Self-Managed Systems

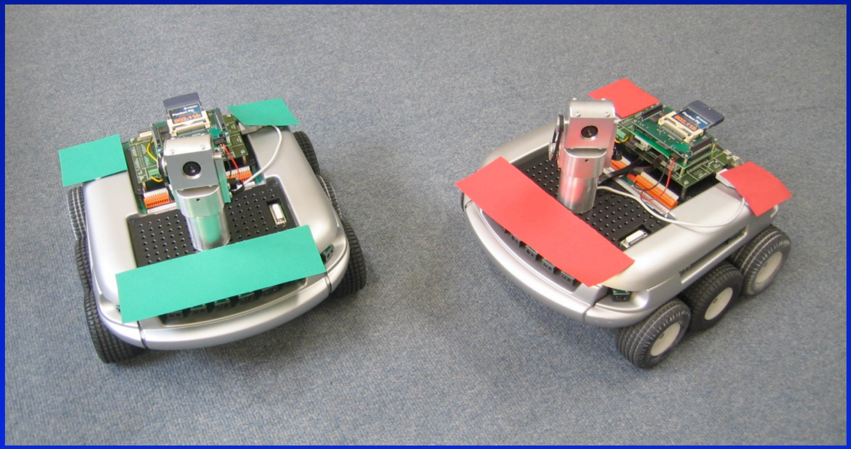
- Autonomous adaptation in response to change of goals and operating environment.

- Self
  - - Configuring
  - - Healing
  - - Tuning

# Three-level architecture (from Gat)

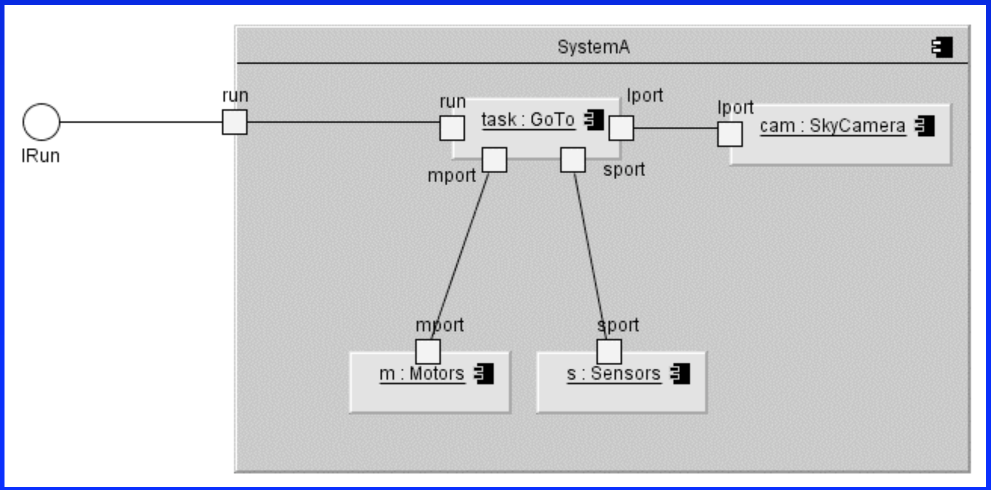


# Test-bed



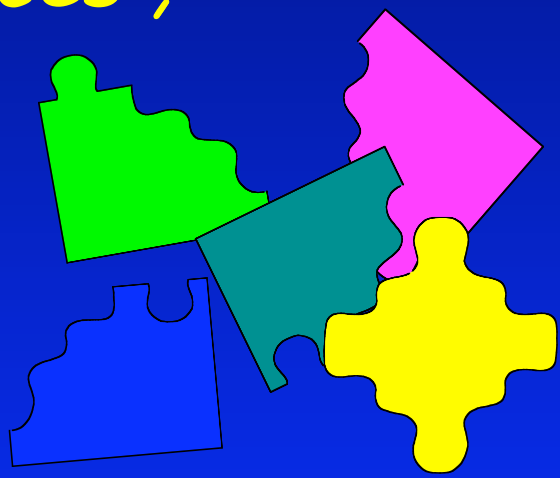
Koala
Robots

Backbone ADL
(UML 2 compatible)

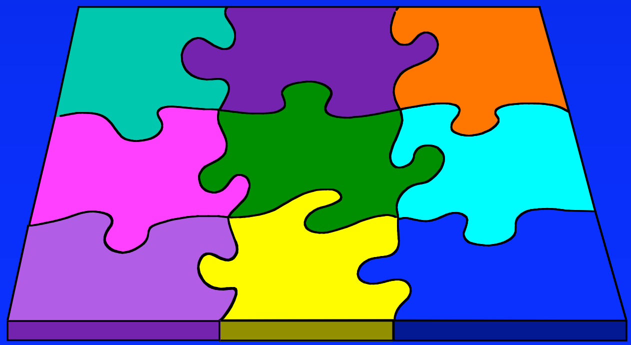


# Research Challenges

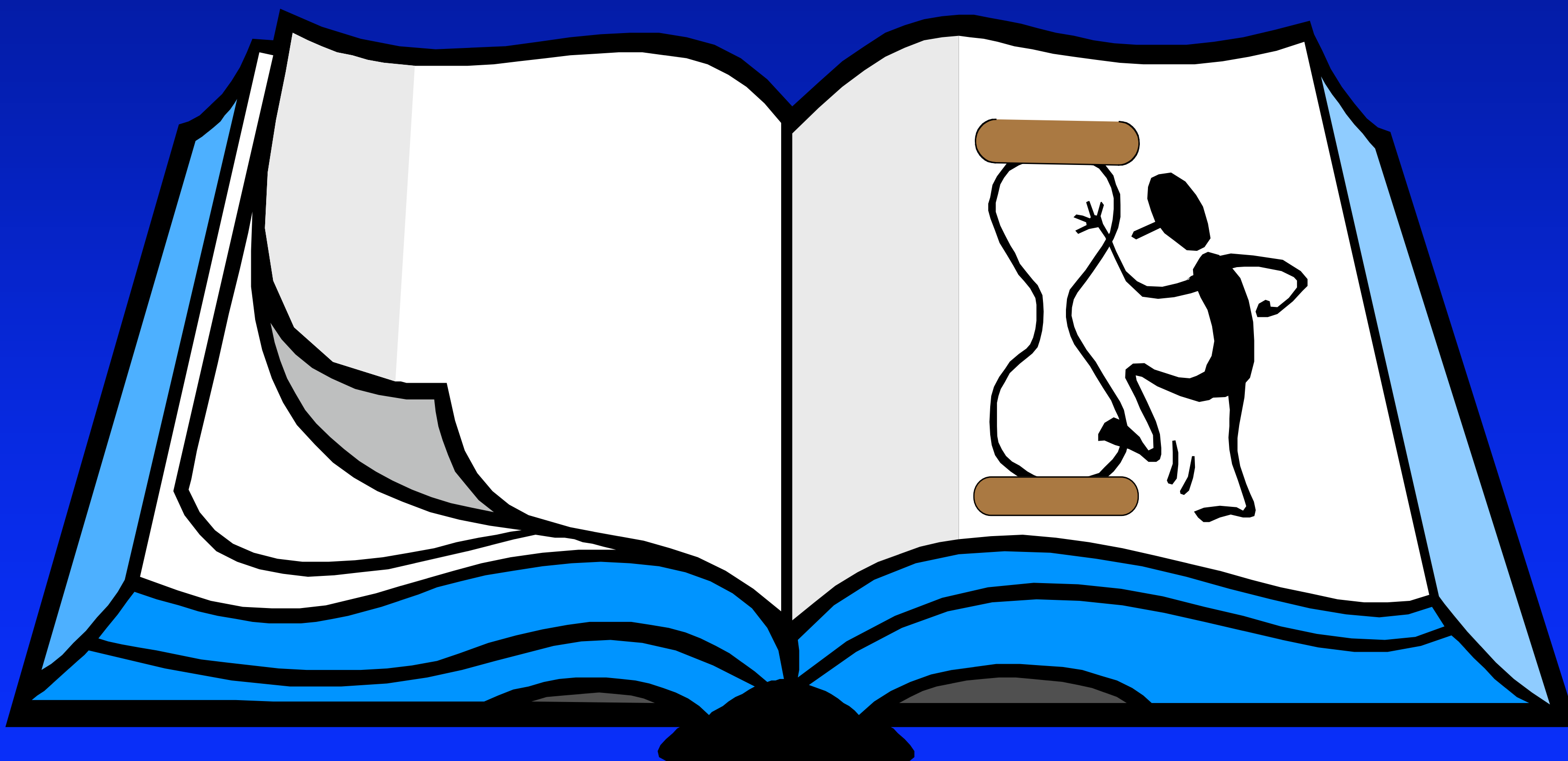
*We have some of the pieces ,*
*but need …*



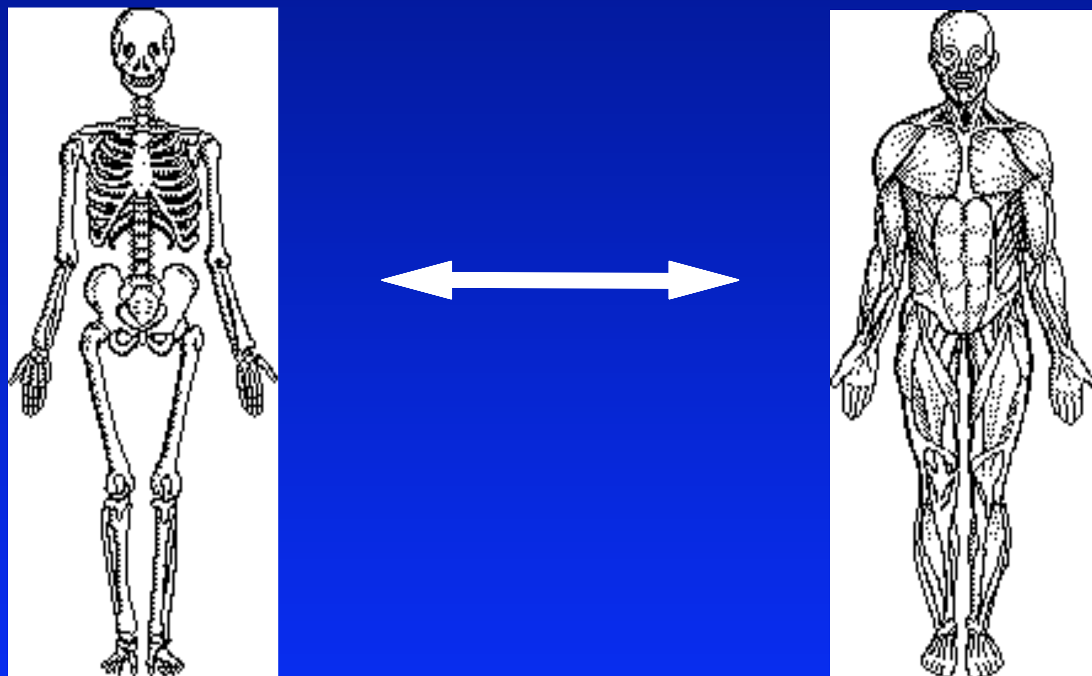
- Scalable decentralised implementation.
- Analysis tools
- Capability to update goals & constraints for operational system



# In conclusion...

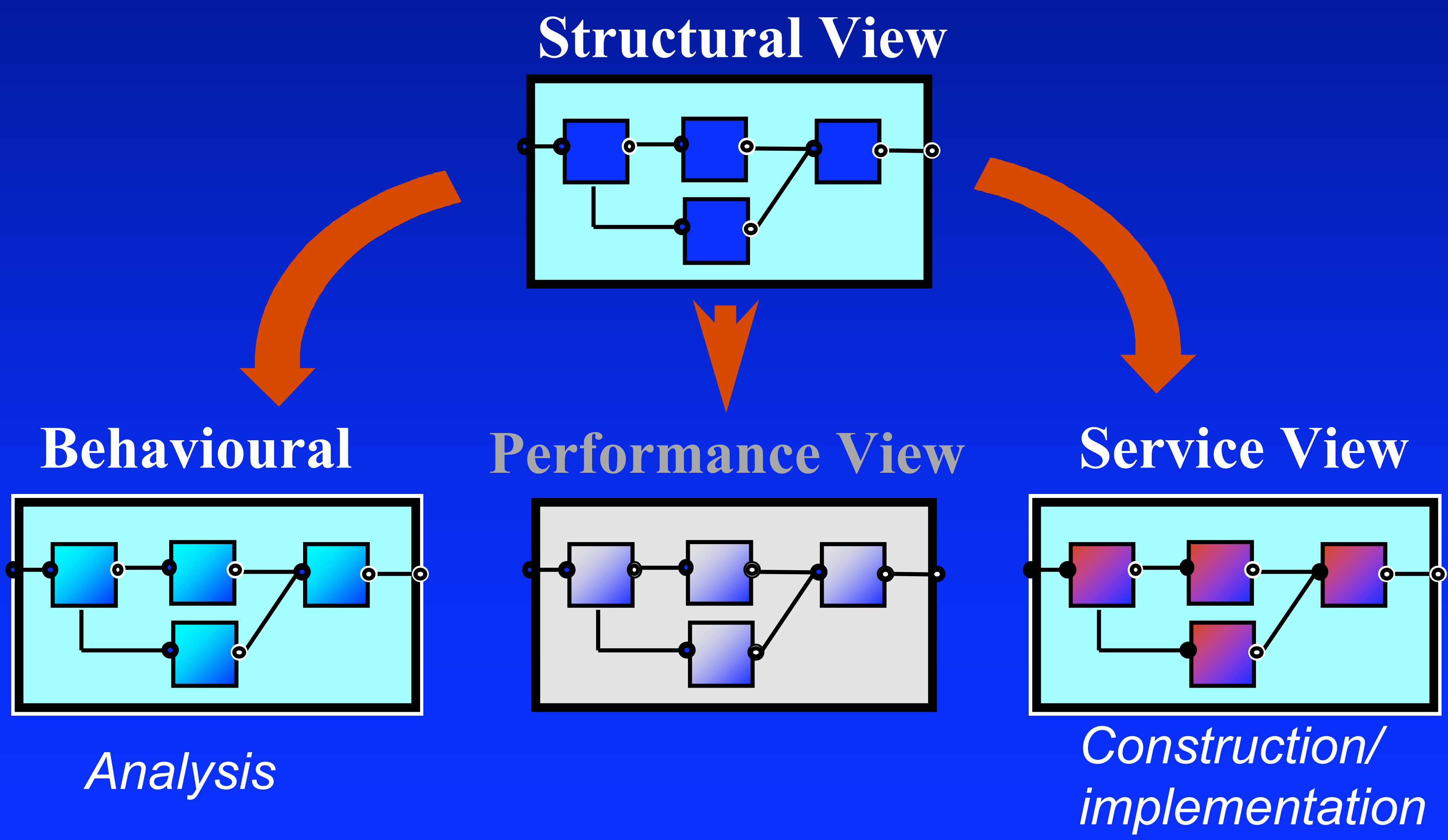


# Architecture as a structural skeleton ….

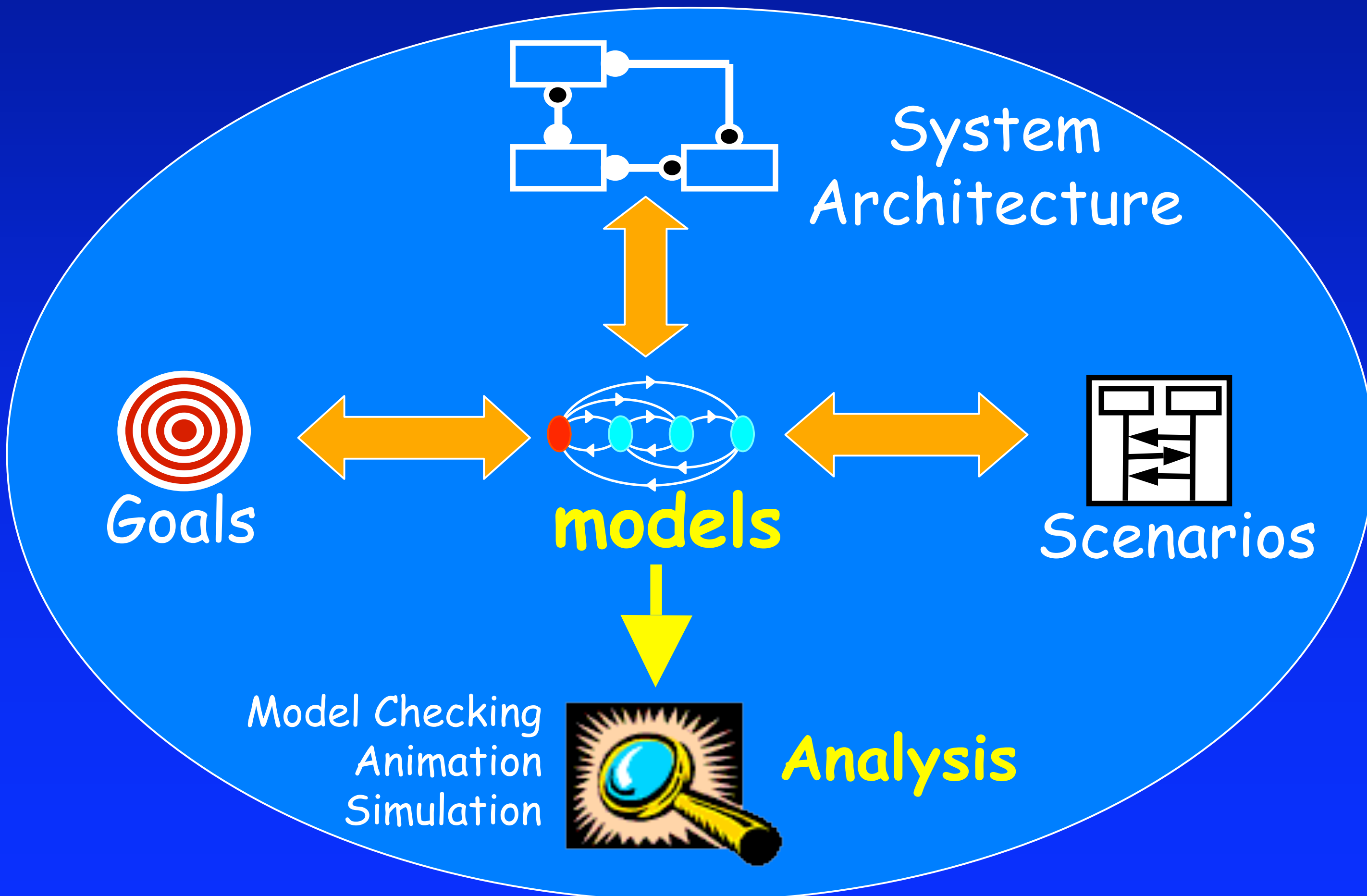


…so that the same simple architectural description can be used as the framework to compose behaviours for analysis, to compose component implementations for systems, ….

# Darwin support for multiple views



# Model-centric approach

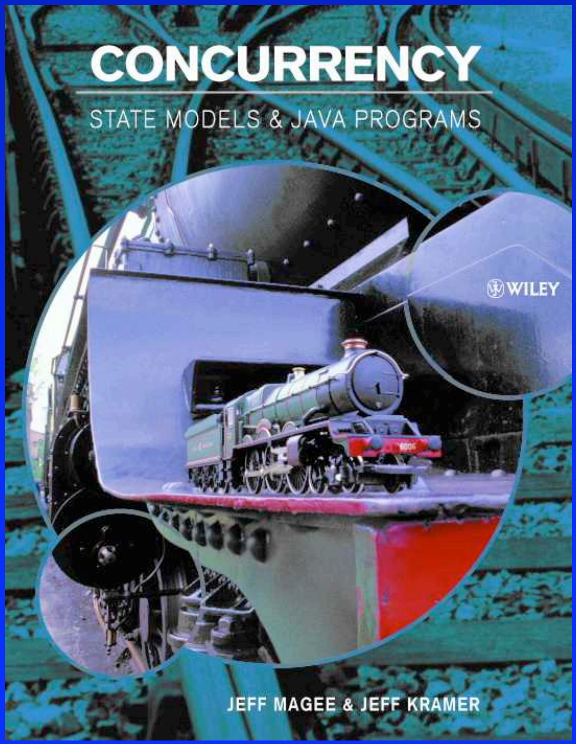


# Research into practice…

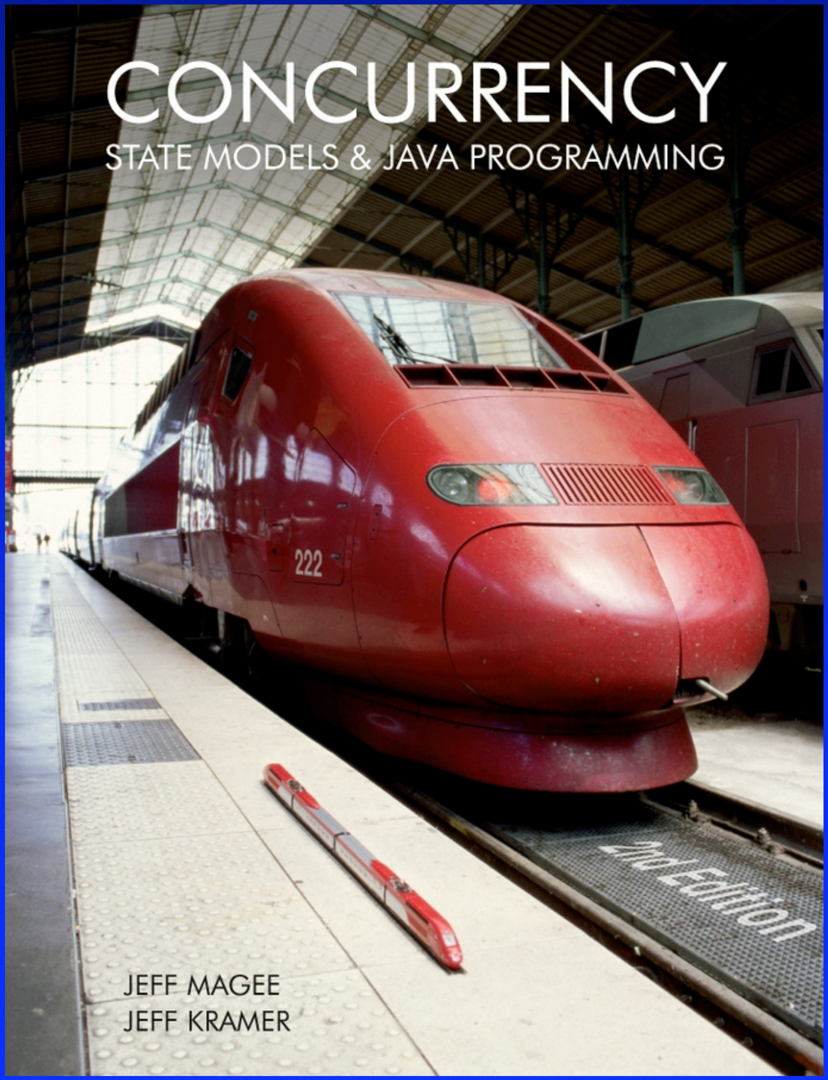
- Application
- Education…
- Further research…

# Education...

1999




2006



# Further research…

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


Sebastian Uchitel

Emmanuel Letier

# Research voyage of discovery

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