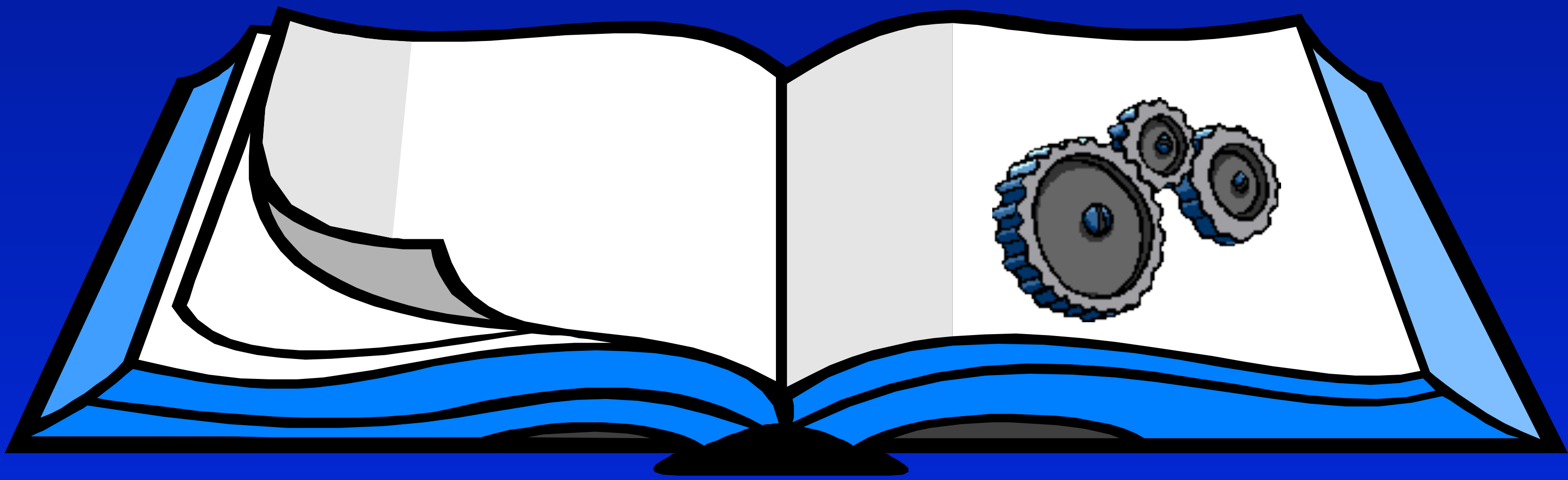
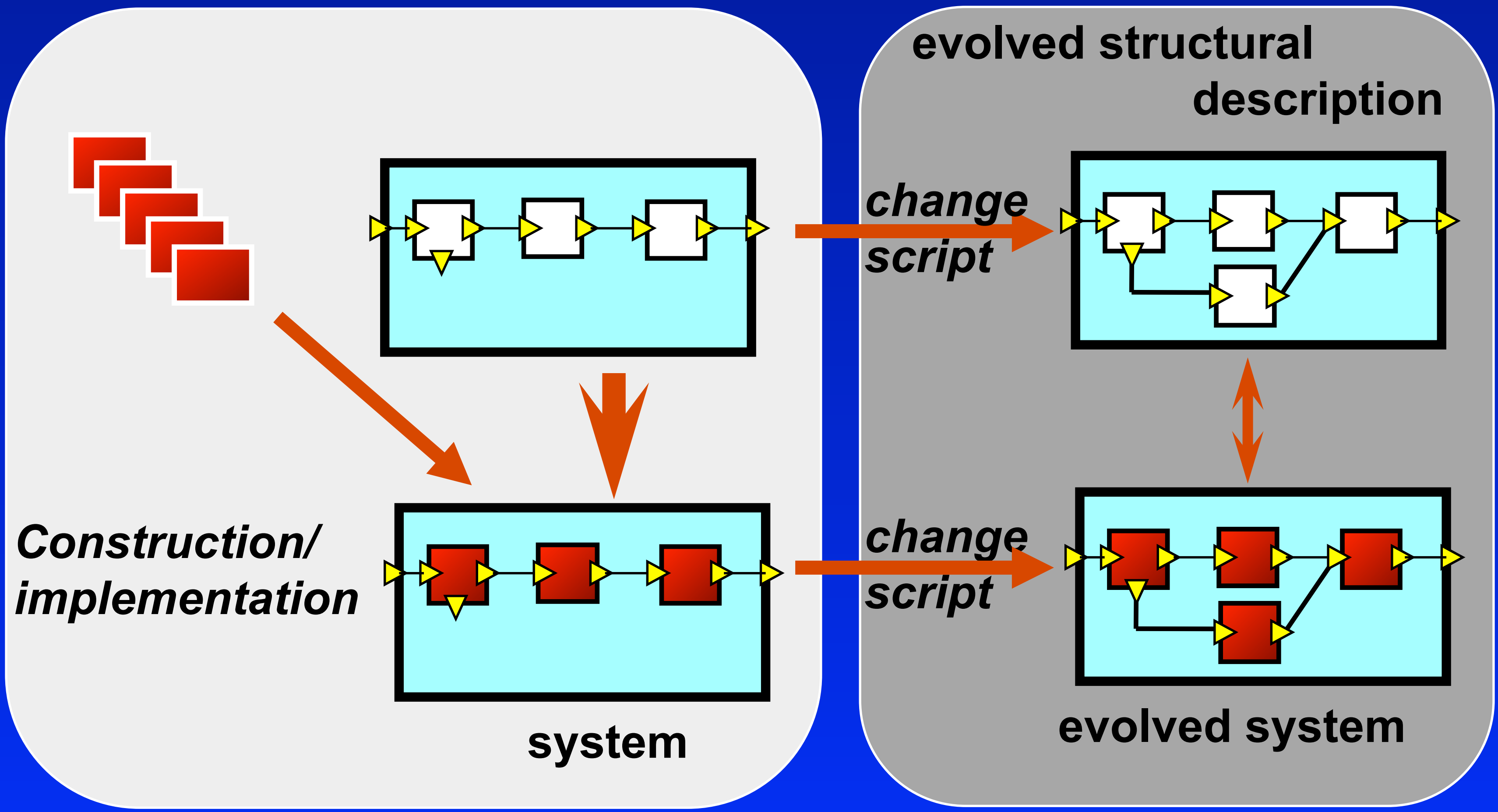
Chapter 8. Dynamic and Adaptive Systems



Managed Structural Change

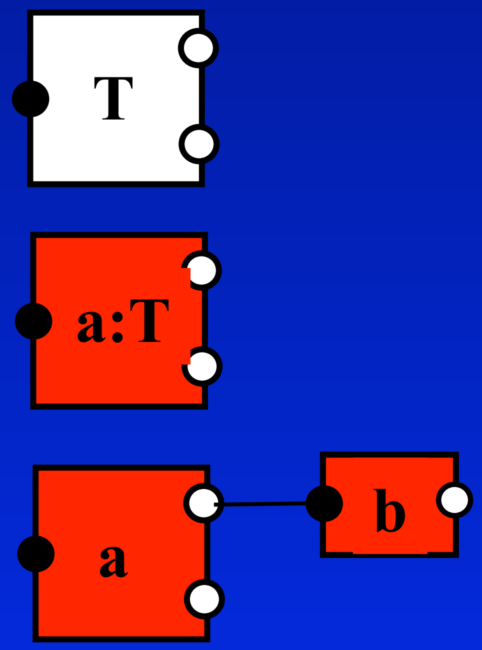


e.g. Conic, Regis

TSE 1985

Structural change

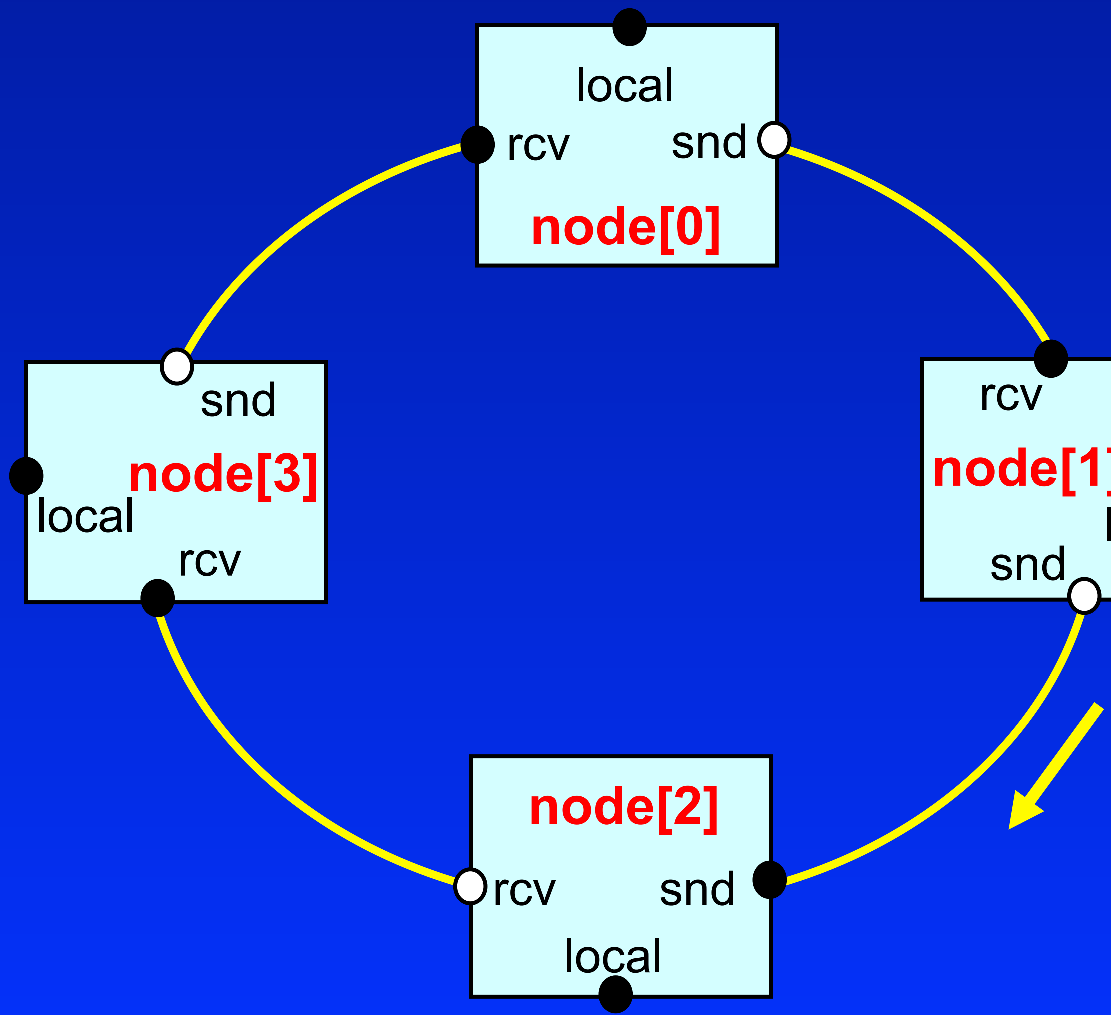
- **load**
  component type
- **create/delete**
  component instances
- **bind/unbind**
  component services



But how can we do this safely?

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

Example - a simplified RING Database

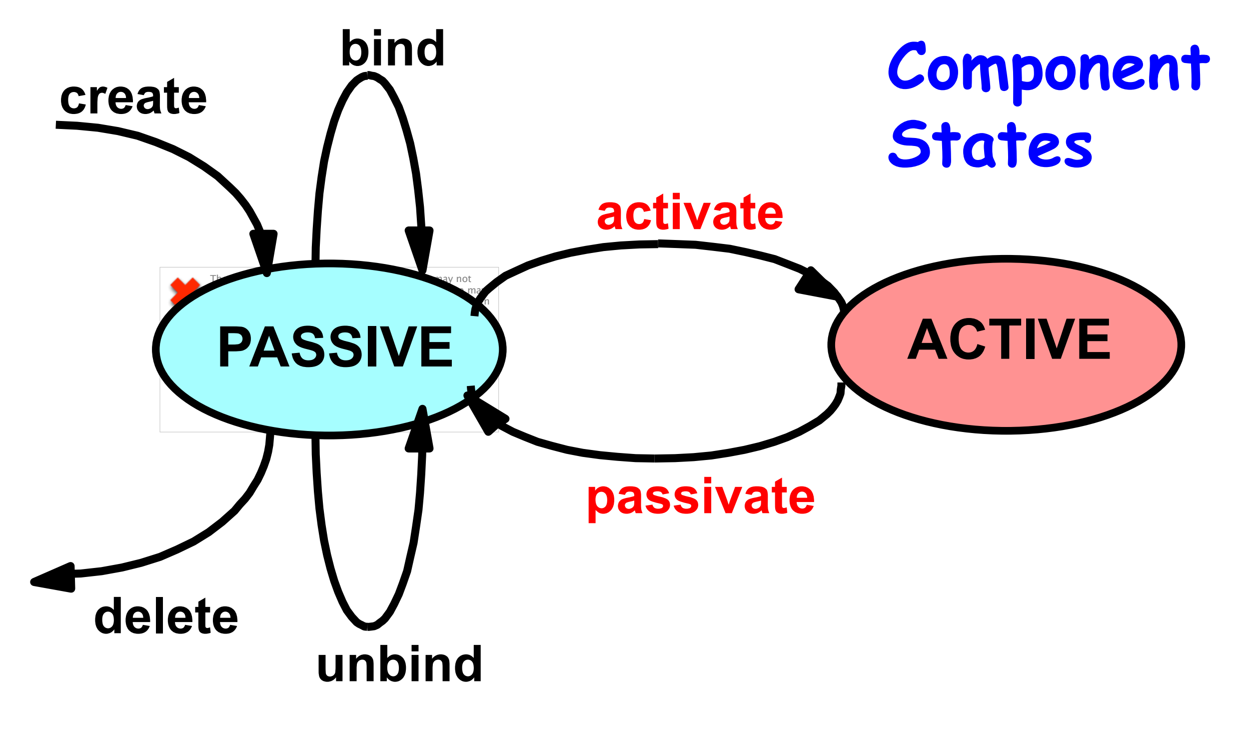


Nodes perform autonomous updates

Updates propagate round the ring via channels

CDS 1998, IEE Proc 1998

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

RING 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])
```

RING Required Properties (2)

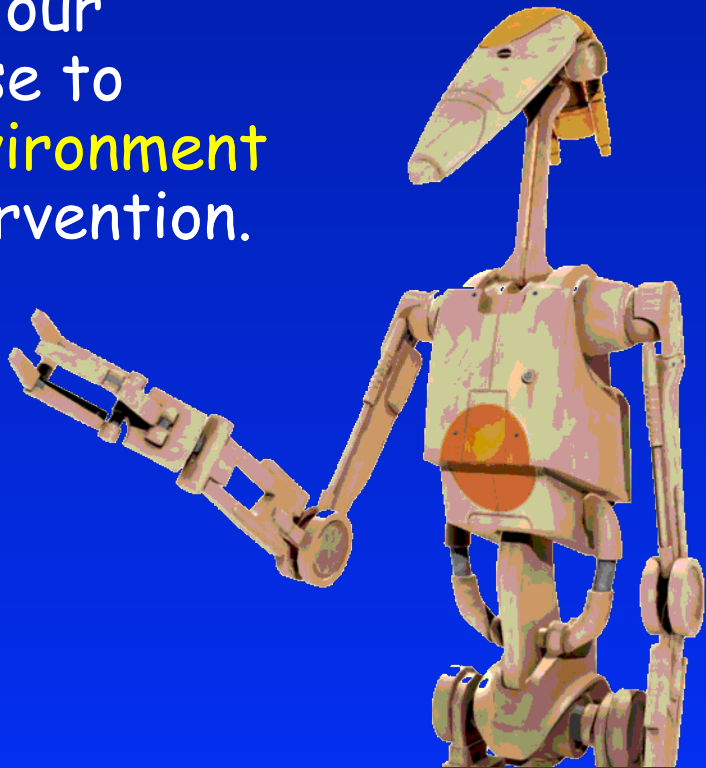
```
// value for a node i with color c
fluent VALUE[i:Nodes][c:Value]
   = <node[i].change[c], ...>

// 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
```

Current Research ...

Self-Managed Adaptive Systems

- Autonomous Adaptation
  - Change/update behaviour dynamically in response to changes in goals & environment without operator intervention.

- Self
  - - Configuring
  - - Healing
  - - Tuning



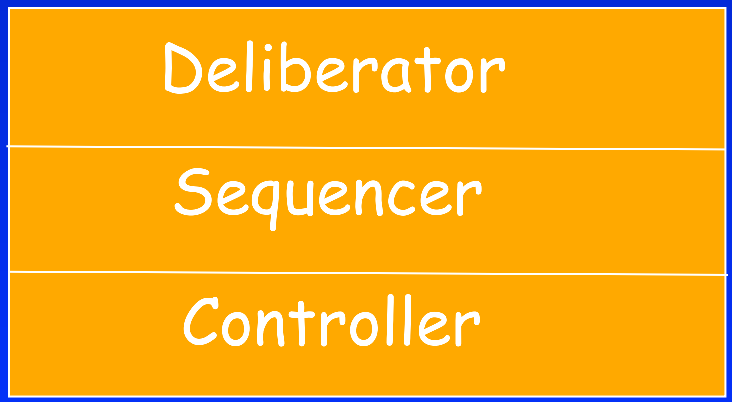
Example Scenario: robotics

S/W Architecture in Robotics

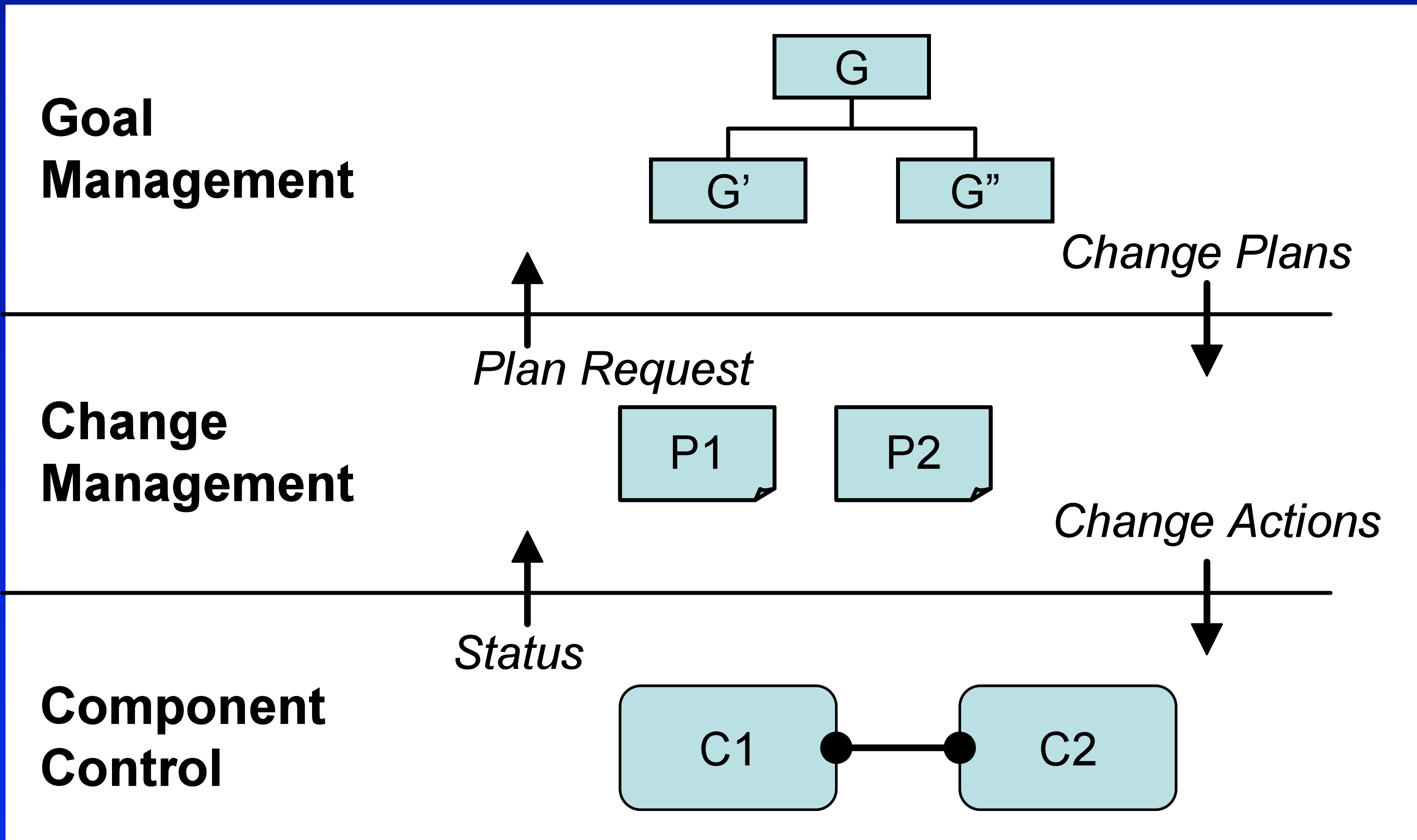
- **SPA 1970's**

- ....

- Three-Layer Architecture (Gat 98)



A Three-Layer Architecture Model



Planning over abstract domain

Assembly of software components to execute plans

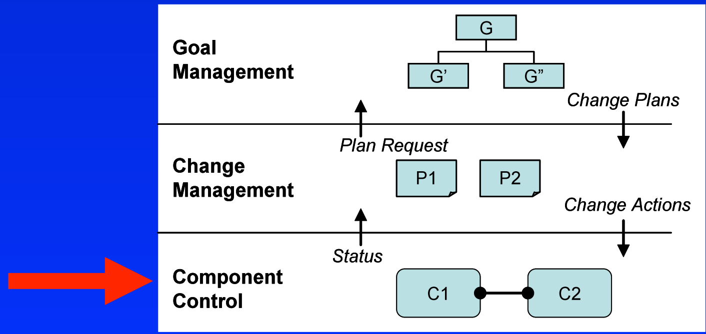
Plan execution using components

- separation of concerns
- layering according to required response times

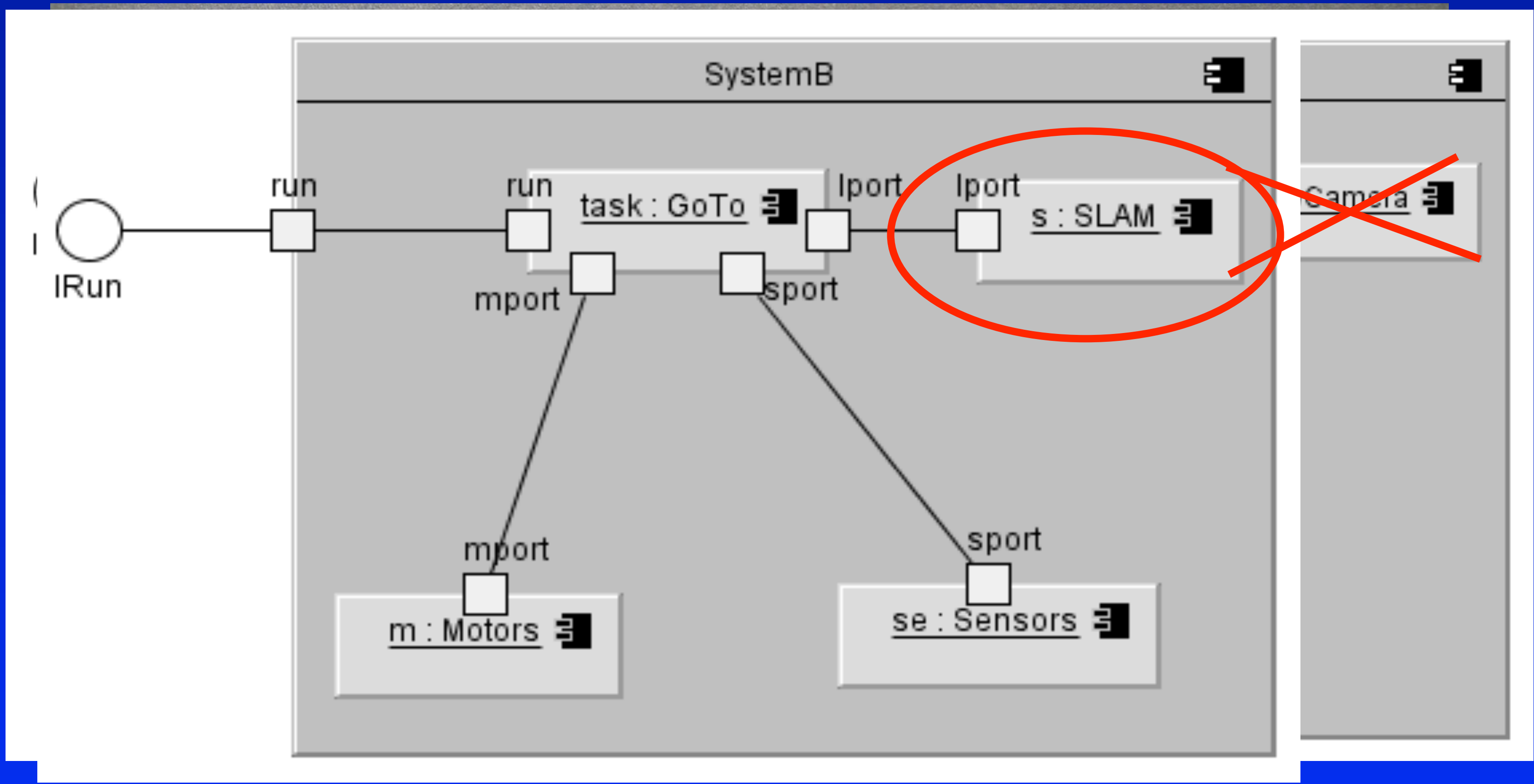
1. Component Control Layer

Layer supports

- Dynamic configuration
  - component creation, deletion and binding
  - Event/status reporting during change
  - Probes & Effectors
- Component execution
- Component self-tuning
  - e.g. TCP timeouts,
    collision avoidance



Component Control - implementation



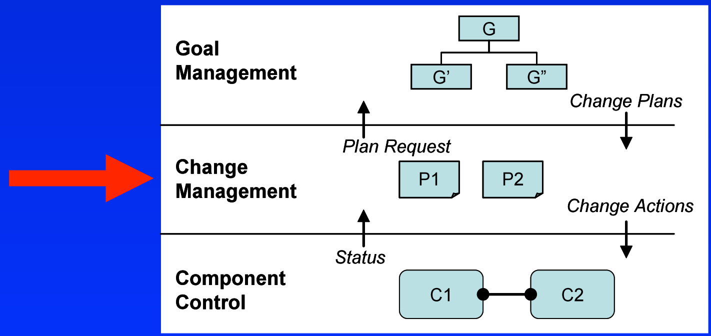
Component Control - Research Challenges

- Safe operation during change
  - stable conditions (quiescence)
    - Kramer & Magee 1986
  - Tranquility
    - Vandewoude et al 2006
  - avoid control transients
    - Schaefer & Wehrheim 2007
- Verification of safety properties during change
  - Zhang & Cheng 2006

2. Change Management Layer

Layer supports

- Plan execution
  - in response to predicted class of events/ state changes in the underlying layer e.g. component failure, mode change.
- Component selection and configuration management
- Plan update
  - in response to unpredicted change (eg. goals)

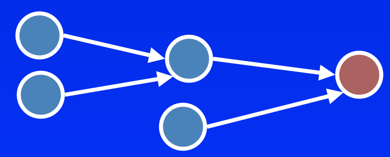


Plan execution

- Reactive Plans are described in terms of condition-action rules over an alphabet of plan actions

```
...
AT.loc1  && !LOADED
      -> pickup
AT.loc1  && LOADED
      -> moveto.loc2
AT.loc2  && LOADED
      -> putdown
AT.loc2  && !LOADED
      -> moveto.loc1
...
```

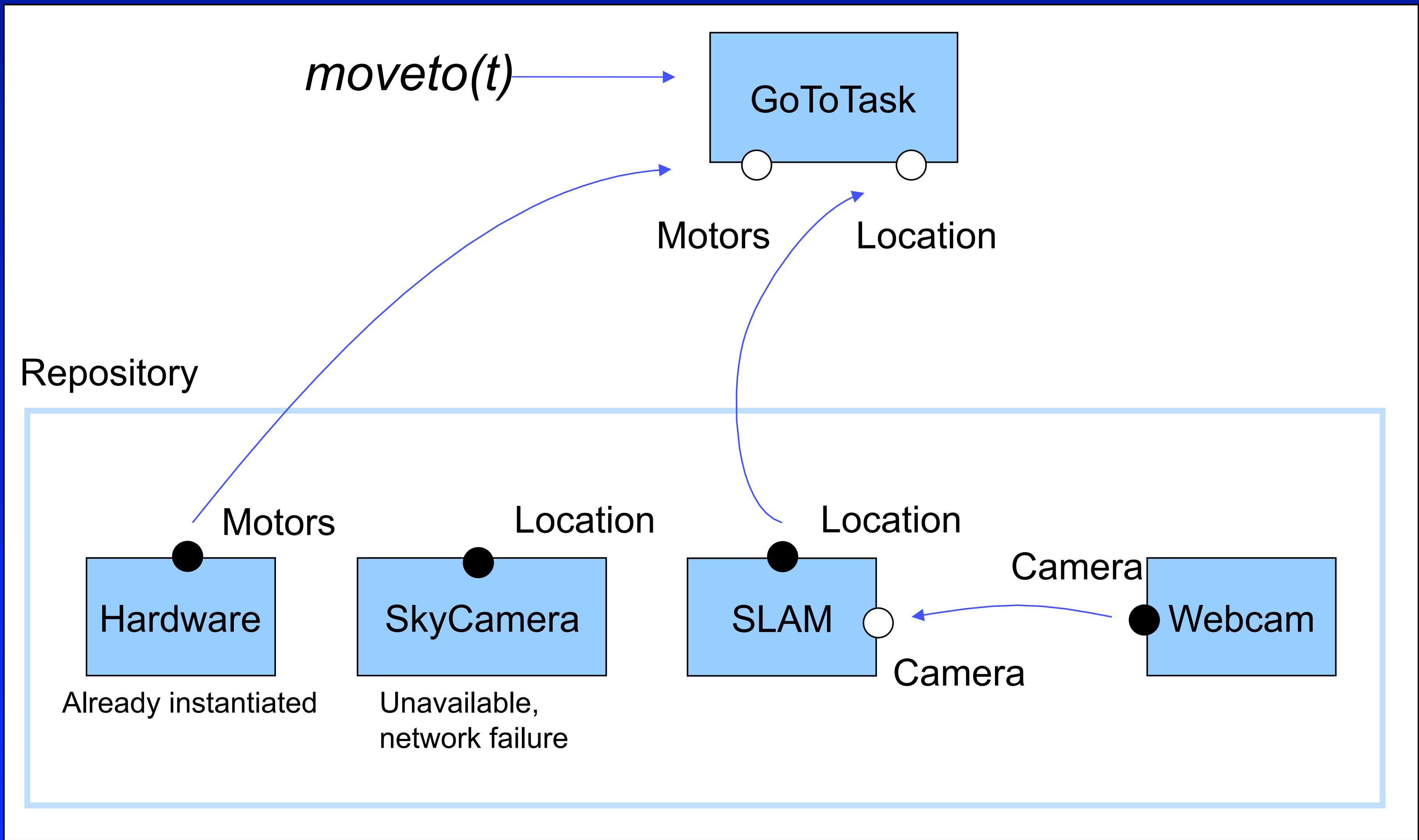
Includes alternative paths to the goals should the environment change in an unpredictable manner.



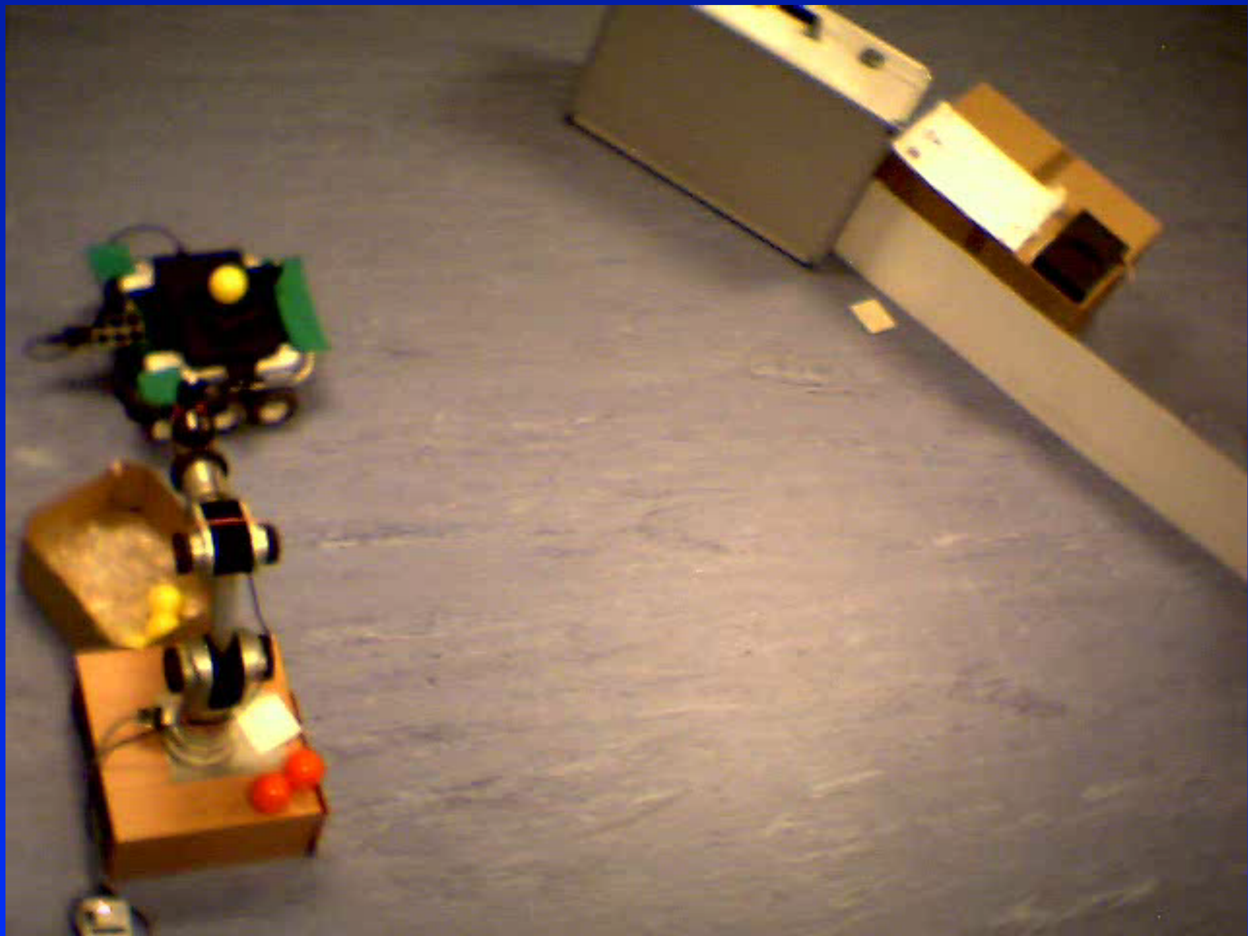
Deriving configurations

- Plan actions (*pickup*, *moveto*, …) do not refer to component configurations explicitly
- Primitive actions associated with interfaces which the interpreter can call (pickup, moveto, …)
- Hence, need a set of components which implement every interface required by the plan, elaborated using dependencies
- Components to interfaces is a many to many relationship, providing alternatives

Component selection



Adaptation Demonstration



Adaptation may require component reselection OR replanning

Change Management – Research Challenges

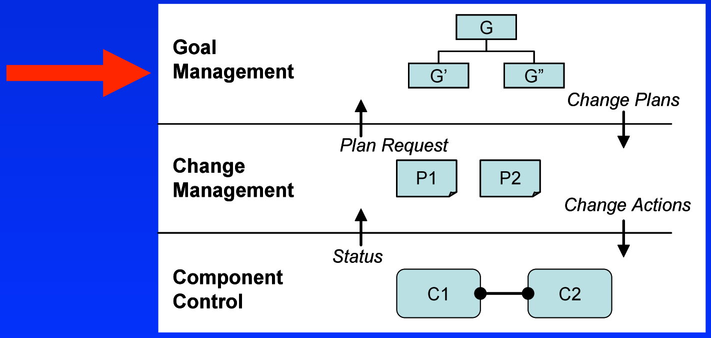
- Scalability -> Distribution & Decentralisation

Georgiadis 2002 – Imposed total ordering Decentralized but not Scalable

Daniel Sykes 2010 – gossip algorithm with convergence

3. Goal Management Layer

- Layer supports plan generation in response to
  - addition/removal of goals
  - requests from below, due to plan failure



Goal Management

Synthesis

“For reactive systems, (systems that maintain an ongoing interaction with a dynamic environment) the synthesis problem has been posed as early as 1957 by Church in the context of digital circuits ... but apart from some impressive theoretical results ... the work on synthesis remained marginal compared to the vast literature on verification ...”

Symbolic Controller Synthesis for Discrete and Timed Systems, Asarin, Maler & Pnueli, LNCS 999, 1995.

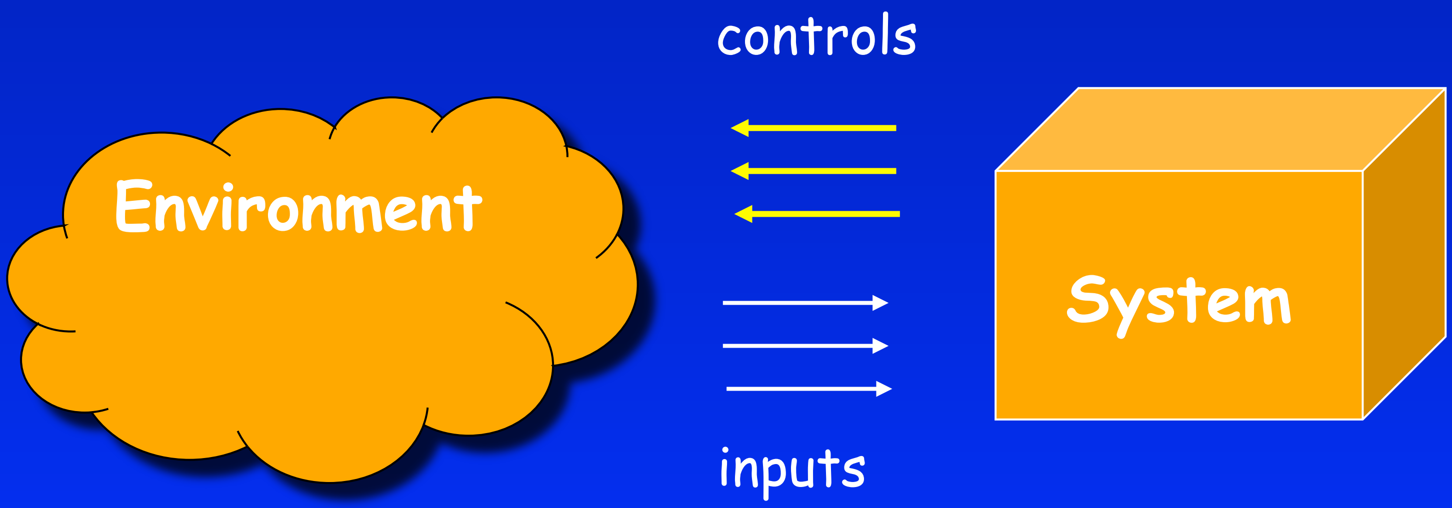
Goal Management – our approach

- Use synthesis to facilitate automated response to changes in goals & environment.
- Translate existing state-based synthesis work into event-based framework – facilitated by our work on Fluent LTL.
- Validate in Koala robot test-bed.

Goal Management

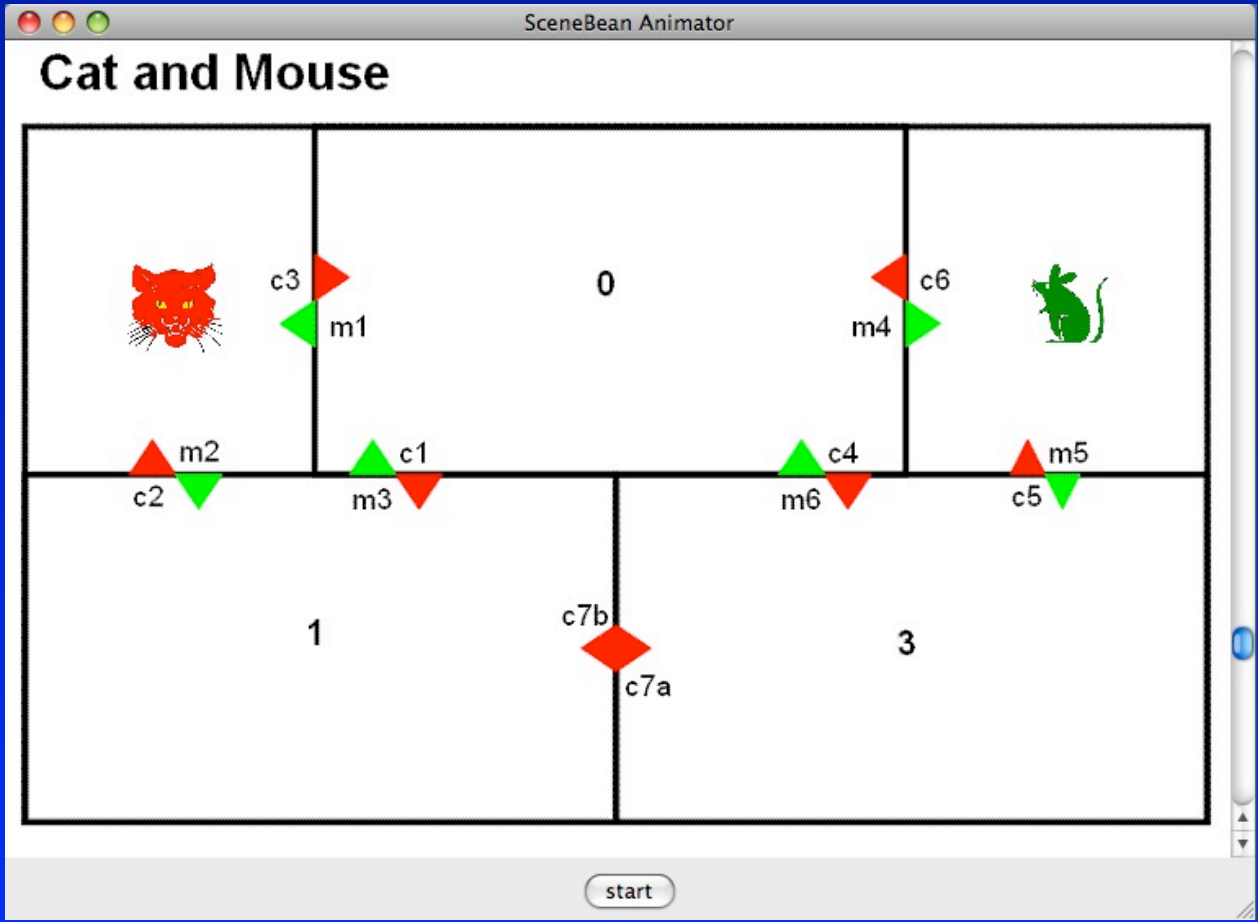
Plan Synthesis

Consider plan as a winning strategy in an infinite two player game between the environment and the system such that goal **G** is always satisfied no matter what order of inputs from environment.



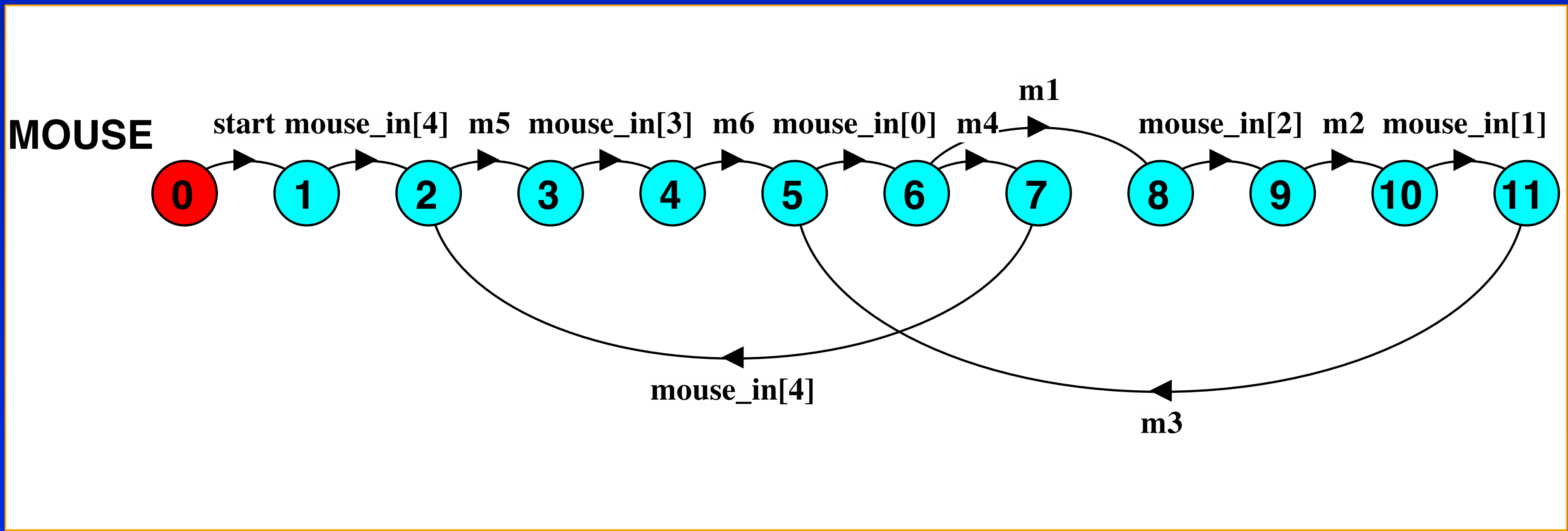
Example

Goal:
Controller of the cat and mouse flaps such that ensure cat and mouse are never in the same room.



Environment Representation

Environment: || composition of LTS



Goal Representation

Goal: Linear Temporal Logic property

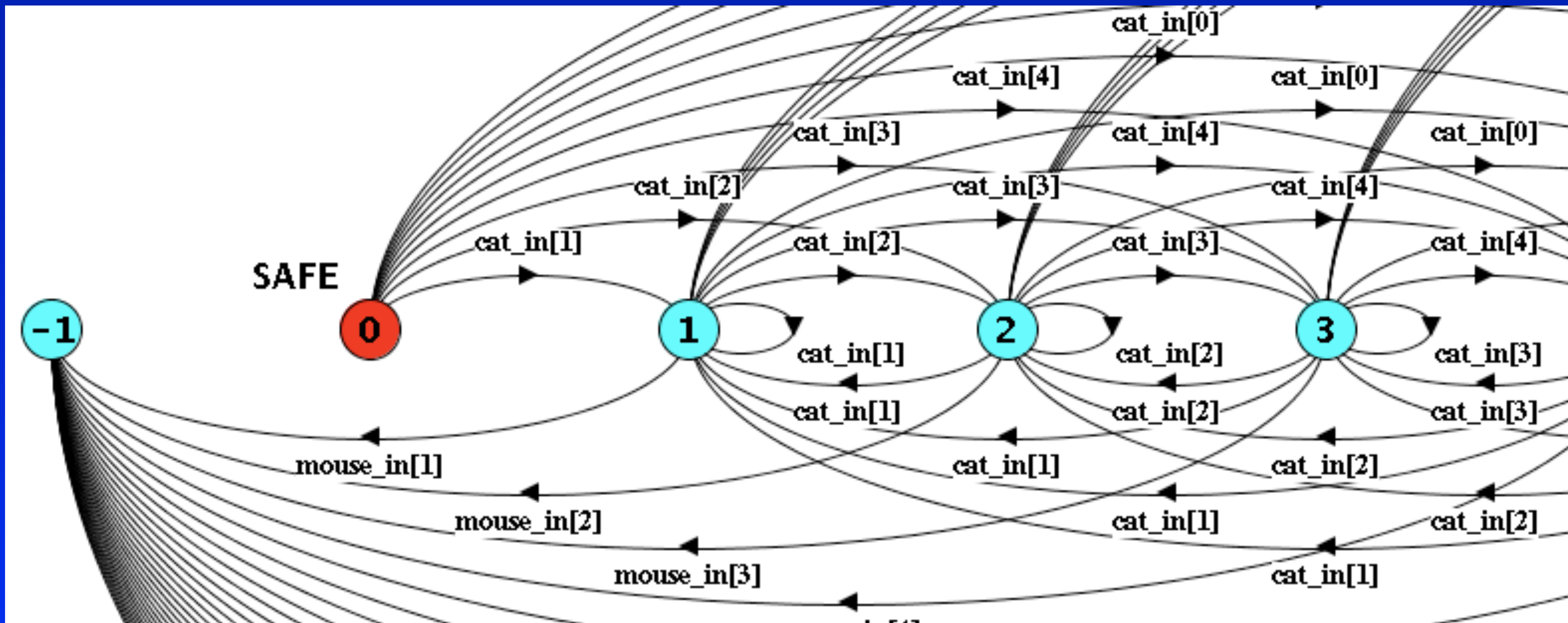
```
ltl_property SAFE =
      []( !exists[i:0..4]
      (CATROOM[i] && MOUSEROOM[i]))
```

Fluents:

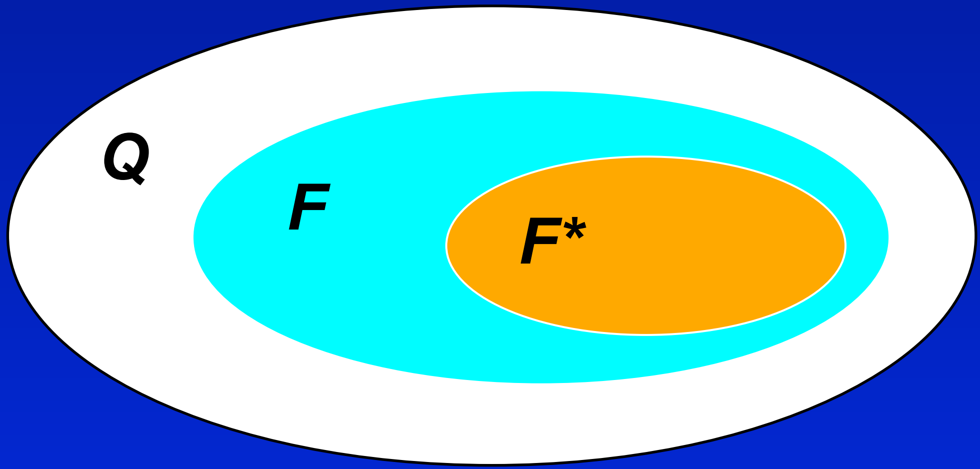
```
fluent CATROOM[room:0..4] =
      <cat_in[room],
      {cat_in[0..4]}\{cat_in[room]}>
fluent  MOUSEROOM[room:0..4] =
    <mouse_in[room],
    {mouse_in[0..4]}\{mouse_in[room]}>
```

Goal Representation - LTS

- Safety Property Automata



Plan Synthesis*



safety game

Q = set of states
F = set of accepting states (G holds)
F* = set of winning states found iteratively such that transition out of ***F**** is via a controlled action.

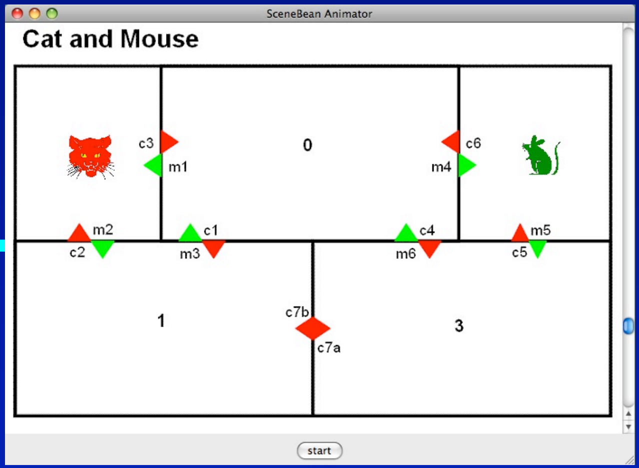
* *Symbolic Controller Synthesis..*, Asarin, Maler, Pneuli, 1989

Computing *F**

- Q = `(CAT || MOUSE || SAFE)`
- Compute F^* by backward propagation of error state:

finally

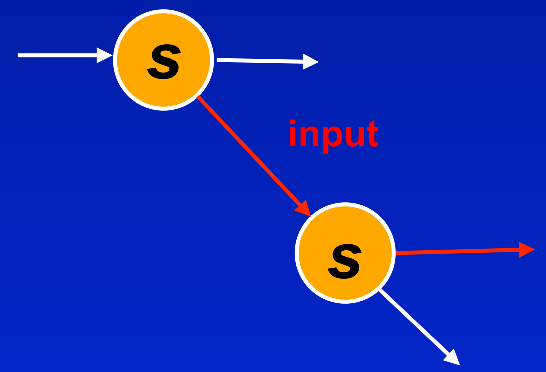
Reactive Plan




```
controller:-
 CATROOM.0 MOUSEROOM.1 -> c4
 CATROOM.0 MOUSEROOM.2 -> {c1, c4, m2}
 CATROOM.0 MOUSEROOM.3 -> c1
 CATROOM.0 MOUSEROOM.4 -> {c1, c4, m5}
 CATROOM.1 MOUSEROOM.0 -> {c2, c7b, m1, m4}
 CATROOM.1 MOUSEROOM.2 -> c7b
 CATROOM.1 MOUSEROOM.3 -> {c2, m6}
 CATROOM.1 MOUSEROOM.4 -> {c2, c7b, m5}
 CATROOM.2 MOUSEROOM.0 -> m4
 CATROOM.2 MOUSEROOM.3 -> {c3, m6}
 CATROOM.2 MOUSEROOM.4 -> {c3, m5}
 CATROOM.3 MOUSEROOM.0 -> {c5, c7a, m1, m4}
 CATROOM.3 MOUSEROOM.1 -> {c5, m3}
 CATROOM.3 MOUSEROOM.2 -> {c5, c7a, m2}
 CATROOM.3 MOUSEROOM.4 -> c7a
 CATROOM.4 MOUSEROOM.0 -> m1
 CATROOM.4 MOUSEROOM.1 -> {c6, m3}
 CATROOM.4 MOUSEROOM.2 -> {c6, m2}
```

Plan extraction

- Label states in ***F**** with fluent values
- Reactive Plan computed from set of *control* states ***S***.
- Control state – has outgoing transition labelled with *control*.
- Stable state – all outgoing transitions are controls – environment can make no moves – quiescent.



Adaptation

- Additional Goals (safety)
  - `[]!(MOUSEROOM[0] && CATROOM[2])`
- Changing Environment
  - doors c7a and c7b not controllable

```
controller:-
 CATROOM.0 MOUSEROOM.3 -> {}
 CATROOM.0 MOUSEROOM.4 -> {c1, c4, m5}
 CATROOM.2 MOUSEROOM.0 -> m4
 CATROOM.2 MOUSEROOM.3 -> {c3, m6}
 CATROOM.2 MOUSEROOM.4 -> {c3, m5}
```

General Goals

- General synthesis problem is 2EXPTIME in length of LTL formula.
- For Generalised Reactivity*, problem can be solved in N^3, where N is state space size.
- Large state spaces can be represented symbolically using BDDs

***Synthesis of Reactive(1) Designs*, Piterman, Pnueli and Sa'ar, 2004

Generalized Reactivity

$$\left(\bigwedge_{i=1}^{k} \Box S_i\right) \wedge \left(\bigwedge_{j=1}^{m} \Box \Diamond J_j^2 \rightarrow \bigwedge_{l=1}^{n} \Box \Diamond J_l^1\right)$$

No Safety Violations!

Using Safety Game algorithm

$$\left(\bigwedge_{j=1}^{m} \Box \Diamond J_j^2 \rightarrow \bigwedge_{l=1}^{n} \Box \Diamond J_l^1\right)$$

Liveness Assumptions

Liveness Guarantees

Example

- Cat & Mouse repeatedly visit room 2 & room 4

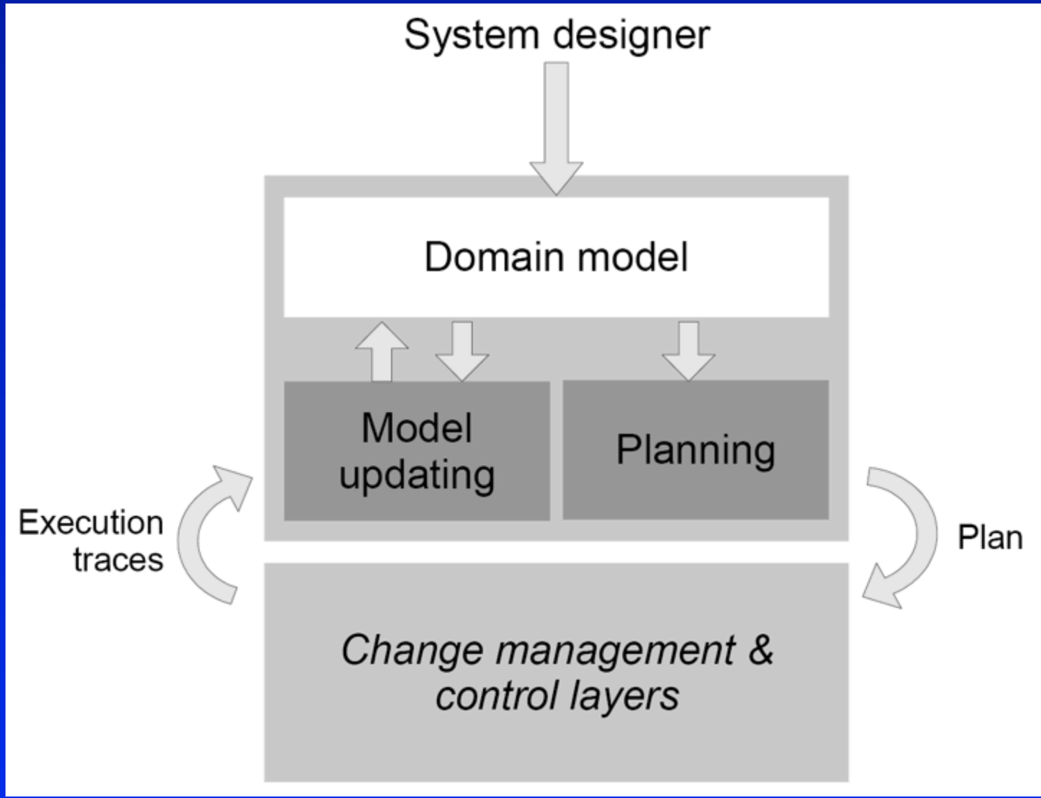
```
assert A1 = MOUSEROOM[2]
assert A2 = MOUSEROOM[4]
assert A3 = CATROOM[2]
assert A4 = CATROOM[4]

goal G1 =
    safety    { SAFE }
    assume    {}
    guarantee {A1, A2, A3, A4}
    controls  { Controllable }
```

Goal Management – Research Challenges

- Specification of domain model and goals
  - application goals
  - system goals
  - covering structure, behaviour, performance …
  - partial knowledge
- Goal refinement
- Runtime Goal & Constraint Checking
- Planning
  - Liveness goals
  - Scalability → Hierarchical Decomposition

Generating Revised Plans

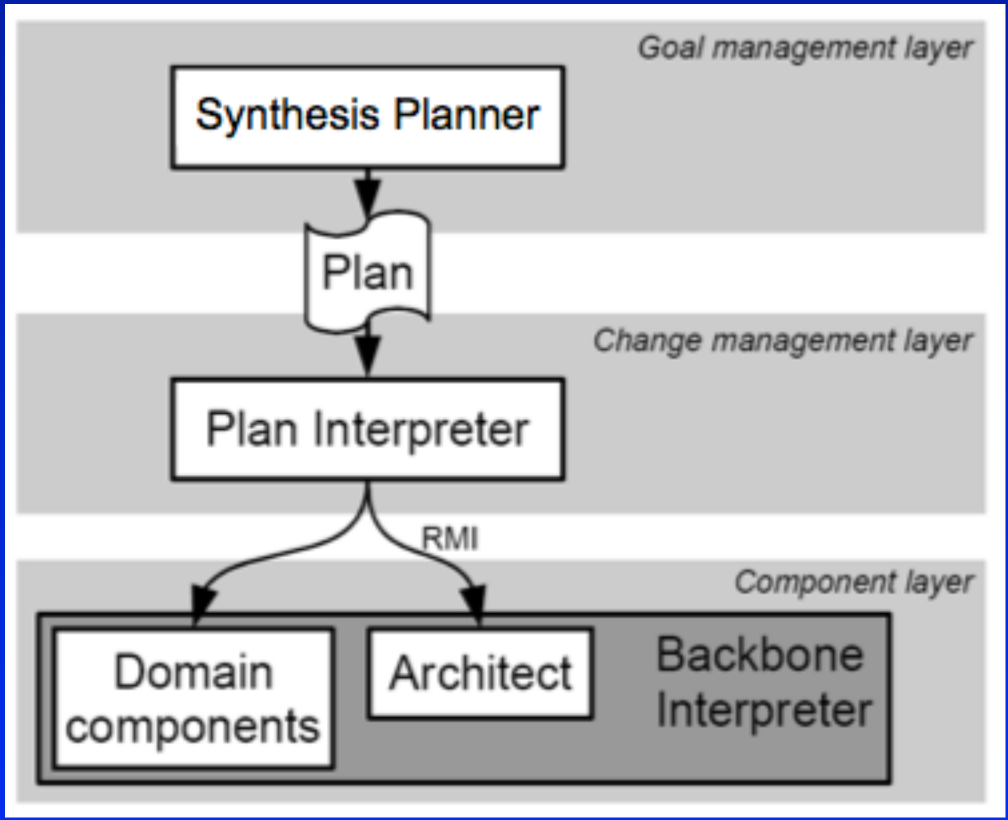
- Plan revision through model revision using observations and probabilistic machine learning



with Daniel Sykes, Alessandra Russo, Katsumi Inoue and Dominico Corapi

Implementation - Status

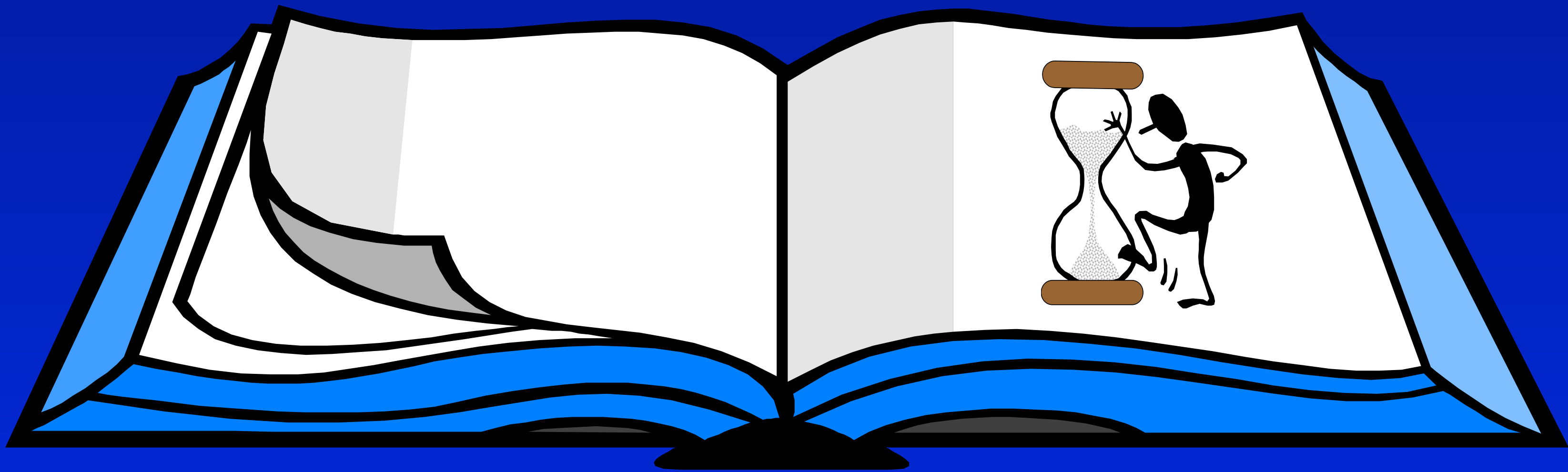
- Plan interpreter
  - Currently runs on a desktop machine
- Component selection
  - Selection not yet fully integrated with plan interpreter
- Components
  - implemented in Java, running on top of the Backbone system, directly on the Koala robots



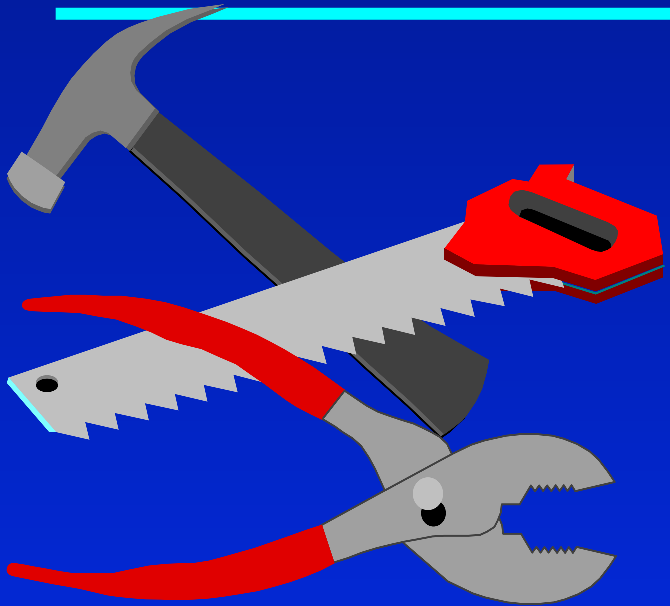
Overall SE Research Challenges

- Challenge is to automate and run on-line what are currently off-line RE/design processes e.g. goal-refinement….
- Need to decide for a given application the requirement for adaptability etc. and the level of automation needed.
- Need to cope with incomplete information about the environment.

Chapter 9. In conclusion... Model Based Design



Software tools



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.

Extensive experience in teaching the approach to both undergraduates and postgraduates in courses on Concurrency.

Experience with R&D teams in industry (BT, Philips, NATS)

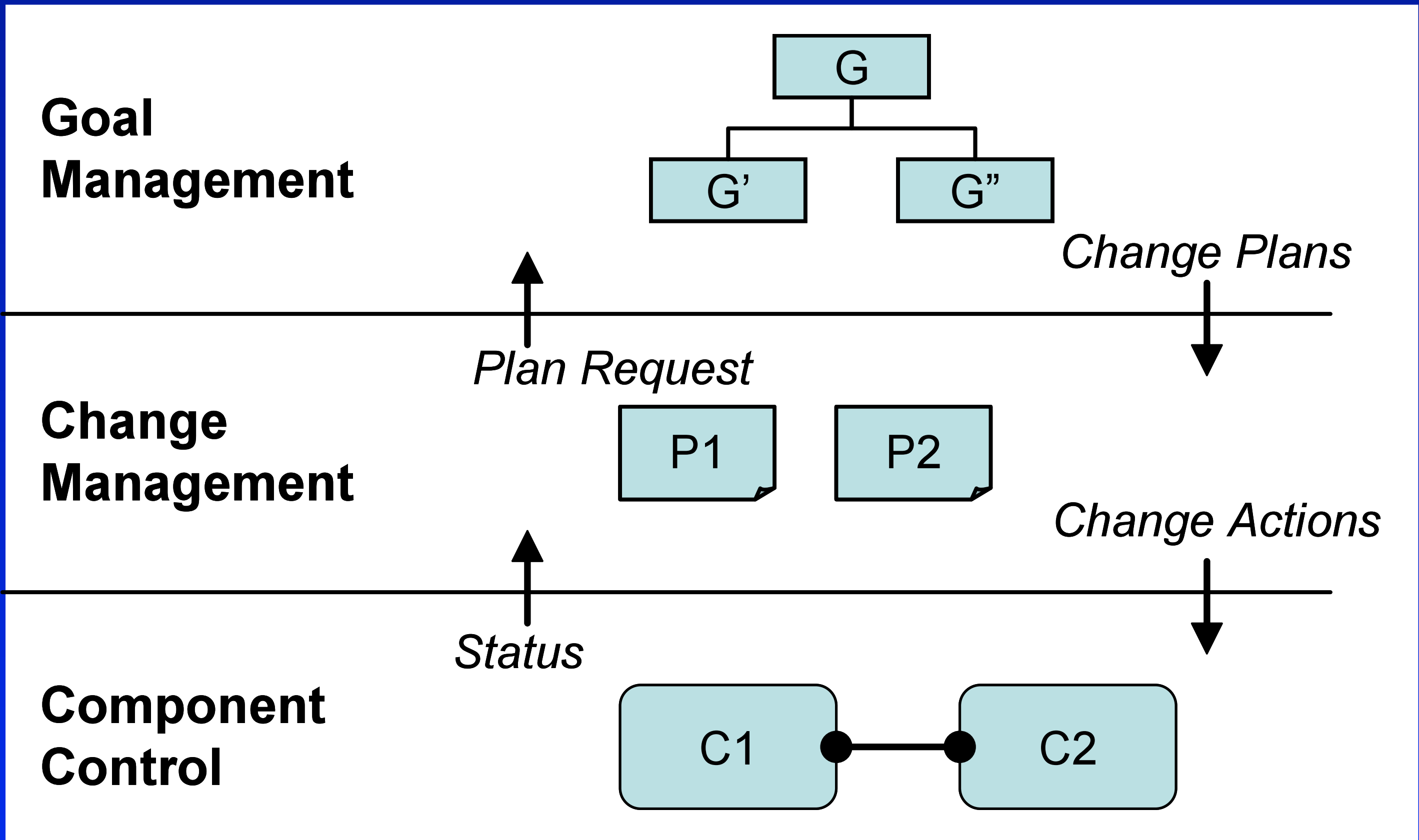
Software Tools - Lightweight vs. Heavyweight

Short learning curve.
Immediate benefits.
Supports incremental model construction. Facilitates interactive experimentation.

vs.

Traditional verification and analysis tools tend to require considerable expertise and have as their goal the ability to target large problems rather than ease of use.

A Three-Layer Architecture Model



Planning

Assembly of software components

Plan execution

- separation of concerns
- layering according to response times

"Self-Managed Systems: An Architectural Challenge", Jeff Kramer & Jeff Magee ICSE FOSE'07

Related Work –

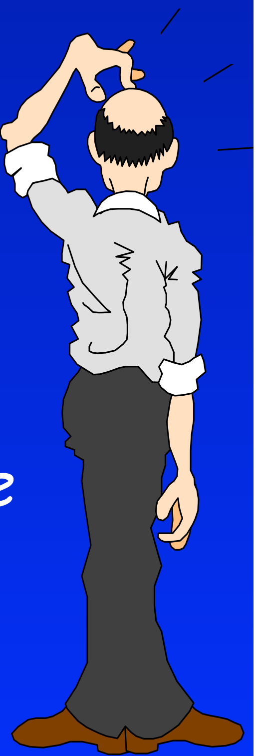
- Lots and lots and lots......

Current work

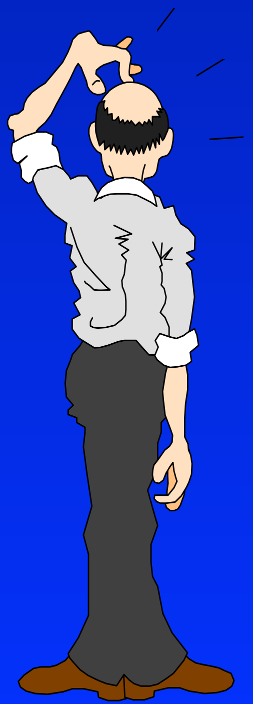
- Modal transition systems (MTS) for partial models
- Adaptive autonomous systems
- Model Checking & Machine Learning for requirements elaboration
- Model revision using observations and probabilistic machine learning

*Emphasis on lightweight, accessible and interactive tools tailored for **engineers**.*

LTSA available from: http://www.doc.ic.ac.uk/~jnm/book/



Model-based design and analysis of concurrent and adaptive software



Jeff Kramer

Imperial College
London

Microsoft Research Summer School, 2012