Plan-Directed Architectural Change for Autonomous Systems

Daniel Sykes, William Heaven, Jeff Magee, Jeff Kramer

das05@doc.ic.ac.uk

Imperial College London

September 3rd 2007

A linear plan

- Motivation for adaptation
- Generating reactive plans
- Deriving configurations from plans
- Ongoing work and conclusion

Coping with reality

- Autonomous systems need to cope with the real world
- The real world is unpredictable
- Autonomy implies minimal contact with programmer
- Thus, need to adapt to changing circumstances and potentially changing goals

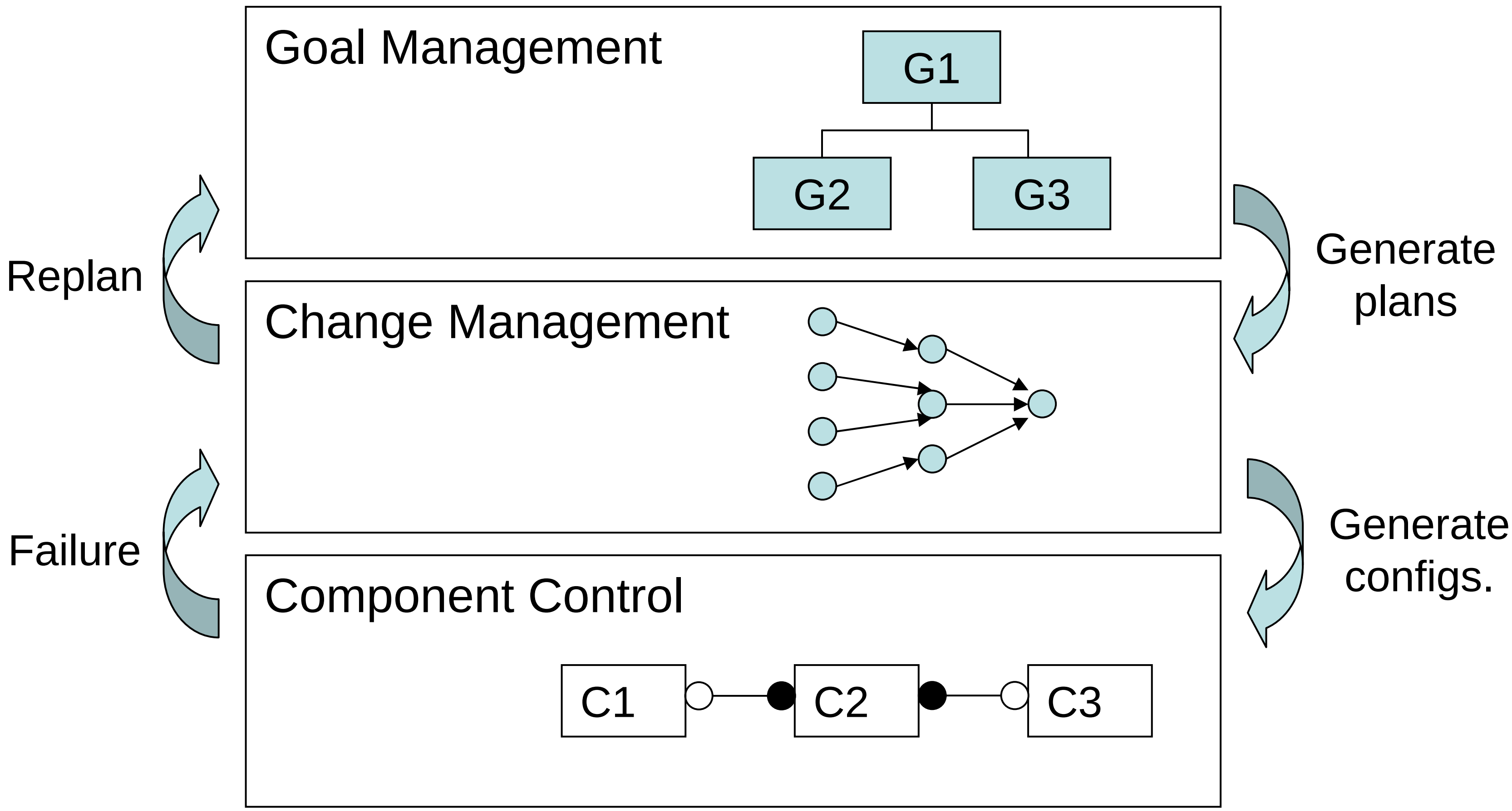
Architectural adaptation

- Adaptations can range from small (continuous) parameter adjustments to complete change of software
- Focus on architectural reconfiguration
  - Wide scope from ‘medium’ to ‘total’ change
  - Can reason about adaptation independent of domain specifics (components are black boxes)
- Much previous work is too rigid
  - Programmer specifies what to change in what circumstances (can he predict all combinations of circumstances?)

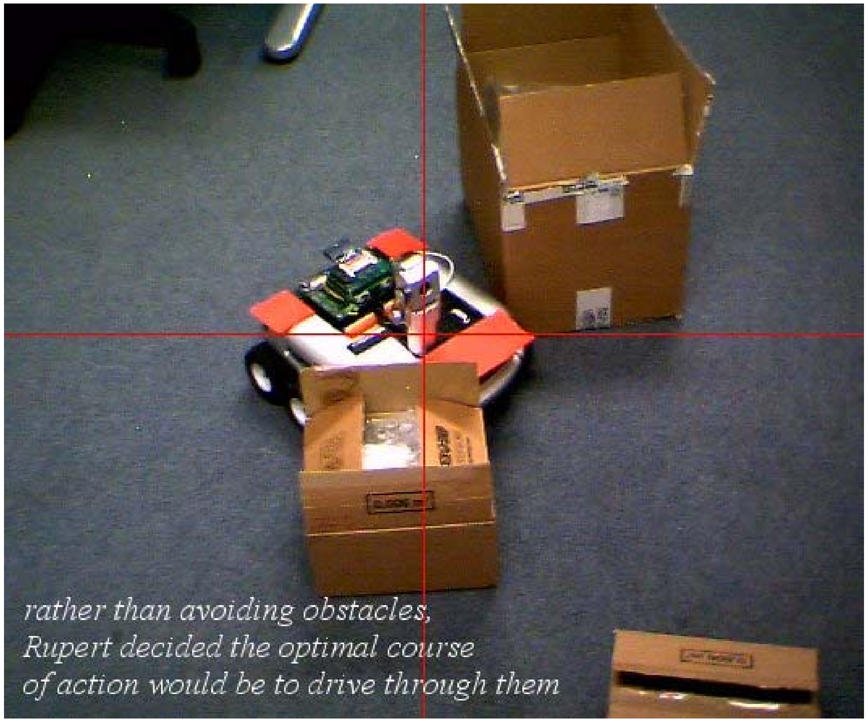
Changing with intent

- Want to allow arbitrary change, but change that serves our goals
- Use the system's plan as a functional specification
- If a component fails during operation we need to find an alternative

Overview

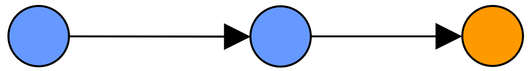

Goal Management
G1
G2
G3
Change Management
Component Control
C1
C2
C3
Replan
Failure
Generate plans
Generate configs.

- ‘Failure’ may be implementation error, environment problem (network connections, unexpected obstacles)
- Hopefully find alternative component(s) and continue same plan
- Otherwise, replan (not currently addressed)

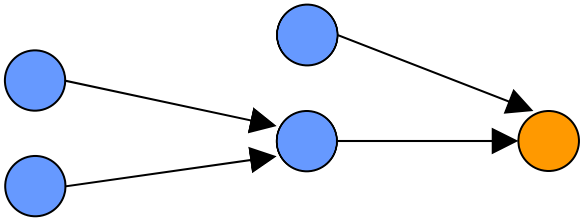


Reactive plans

- Desired behaviour of the system given as CTL goals, over some domain description
- Planner (MBP) uses model-checking to generate a *reactive* plan (as opposed to a linear plan)
- The plan contains all (world) states from which goal is reachable
  - handles non-determinism in environment – actual next state may not be the expected result of an action



Linear plan

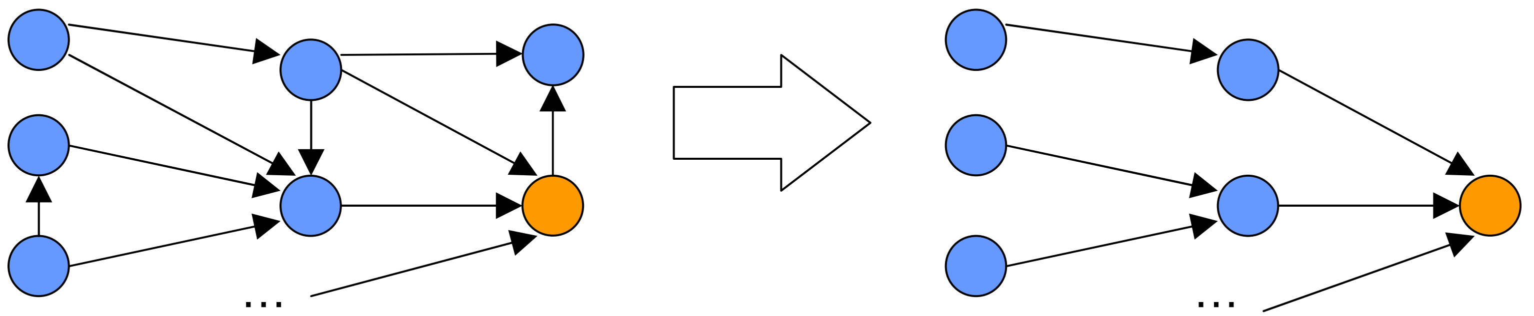


Reactive plan

das05@doc.ic.ac.uk

Domain description

- Domain description contains a set of actions, with their pre and post conditions
  - Pre: ball_at(loc1), robot_at(loc1)
  - Action: pickup
  - Post: robot_has(ball)
- Can be regarded as an LTS where states are conjunctions of predicates, which the planner prunes to generate a plan



Domain description

Reactive plan

Plans

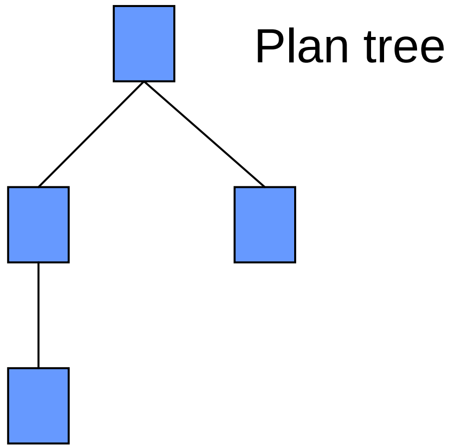
- Generated plans are sets of condition-action rules
- Interpreter checks actual world state after every action

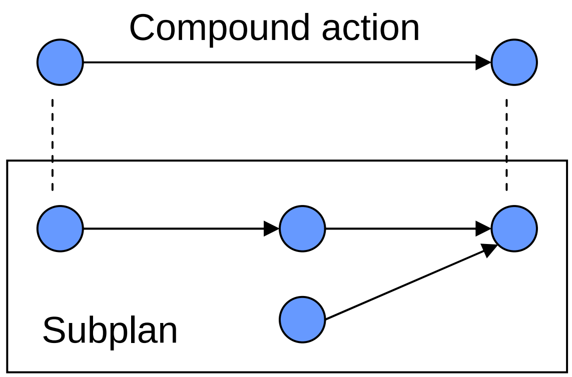
```
S1  (case (and (= photographed target1))
              (done))
S2  (case (and (= photographed 0) (= koala1_location loc1) (= target1_location loc1))
              (action koala1_photograph_target1))
S3  (case (and (= photographed 0) (= koala1_location loc1) (= target1_location loc2))
              (action koala1_goto_loc2))

    …

    …

Sn  (case (and (= photographed 0) (= koala1_location loc3) (= target1_location loc3))
              (action koala1_photograph_target1))
    (else (fail))
```

(ordering of states is arbitrary)

Managing state space

- State explosion a problem for non-trivial domains
- Use a hierarchy of partial descriptions, and generate a hierarchy of plans
- Root plan contains only ‘abstract’ or ‘compound’ actions
- Subplans contain ‘primitive’ actions which elaborate or *refine* the compound actions
- Subplans are generated at runtime from the current state

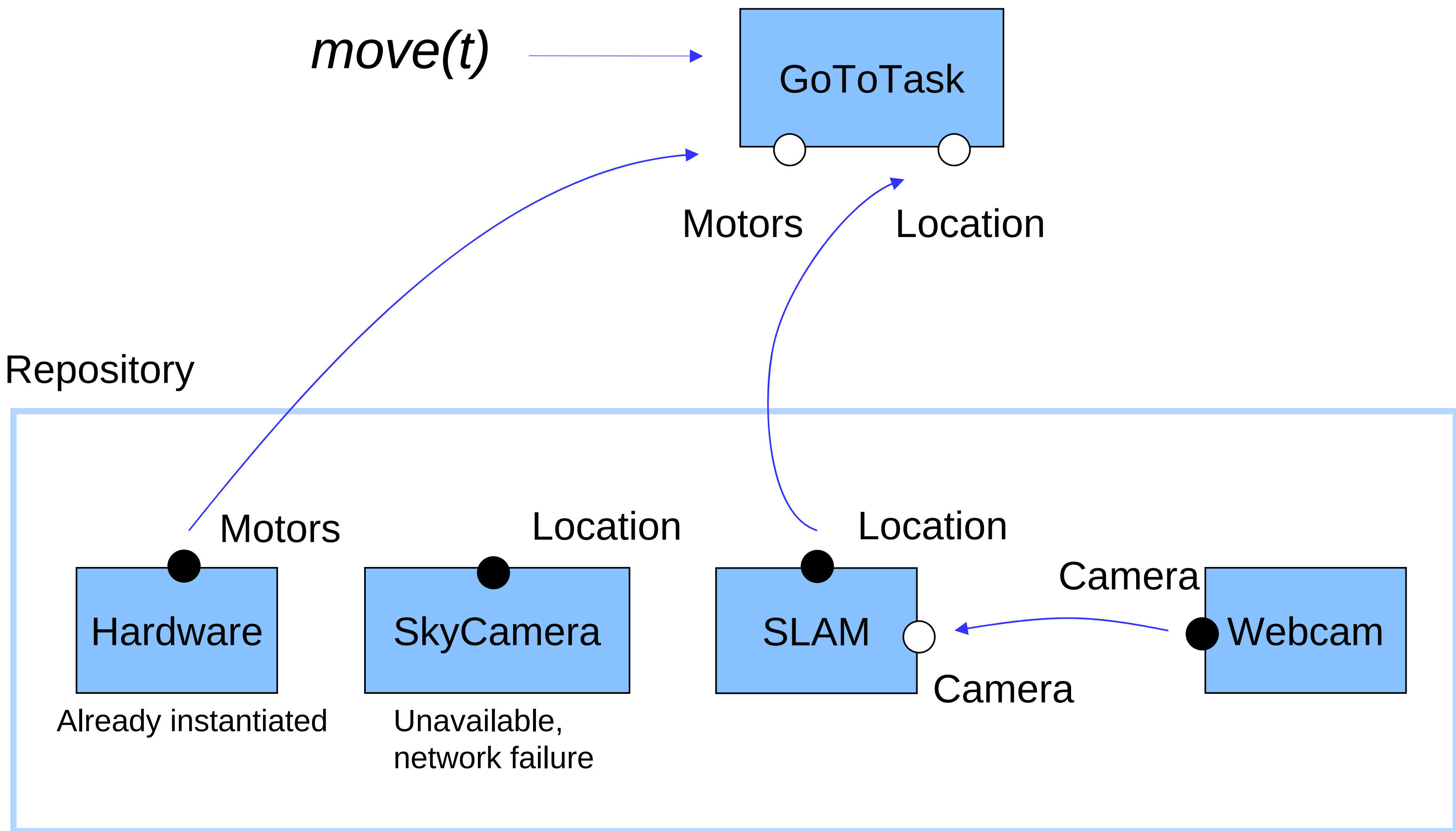




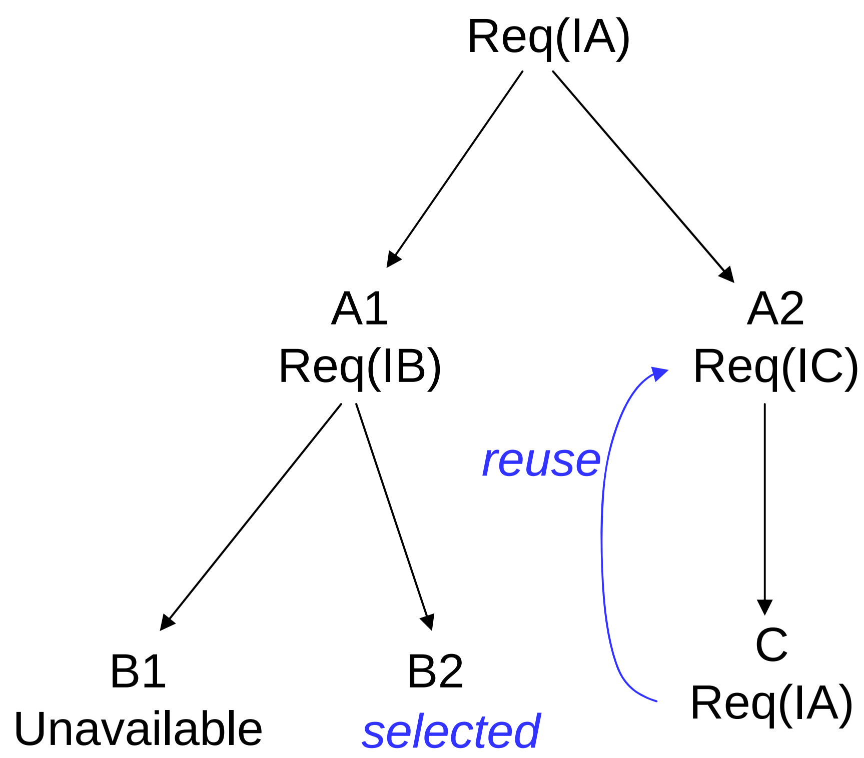
Deriving configurations

- Plan describes functional requirements in terms of actions
  - They do not refer to configurations explicitly
- Primitive actions associated with interfaces which the interpreter can call
- Hence, need a set of components which implement every interface required by the plan
- Components to interfaces is a many to many relationship, providing alternatives

Component selection



- Components already instantiated or already selected are reused
  - Assumes one instance providing each interface
- Components marked as unavailable (or have unsatisfiable requirements) are not selected
- Here, 2 solutions – {A1,B2} or {A2,C} – which is better?

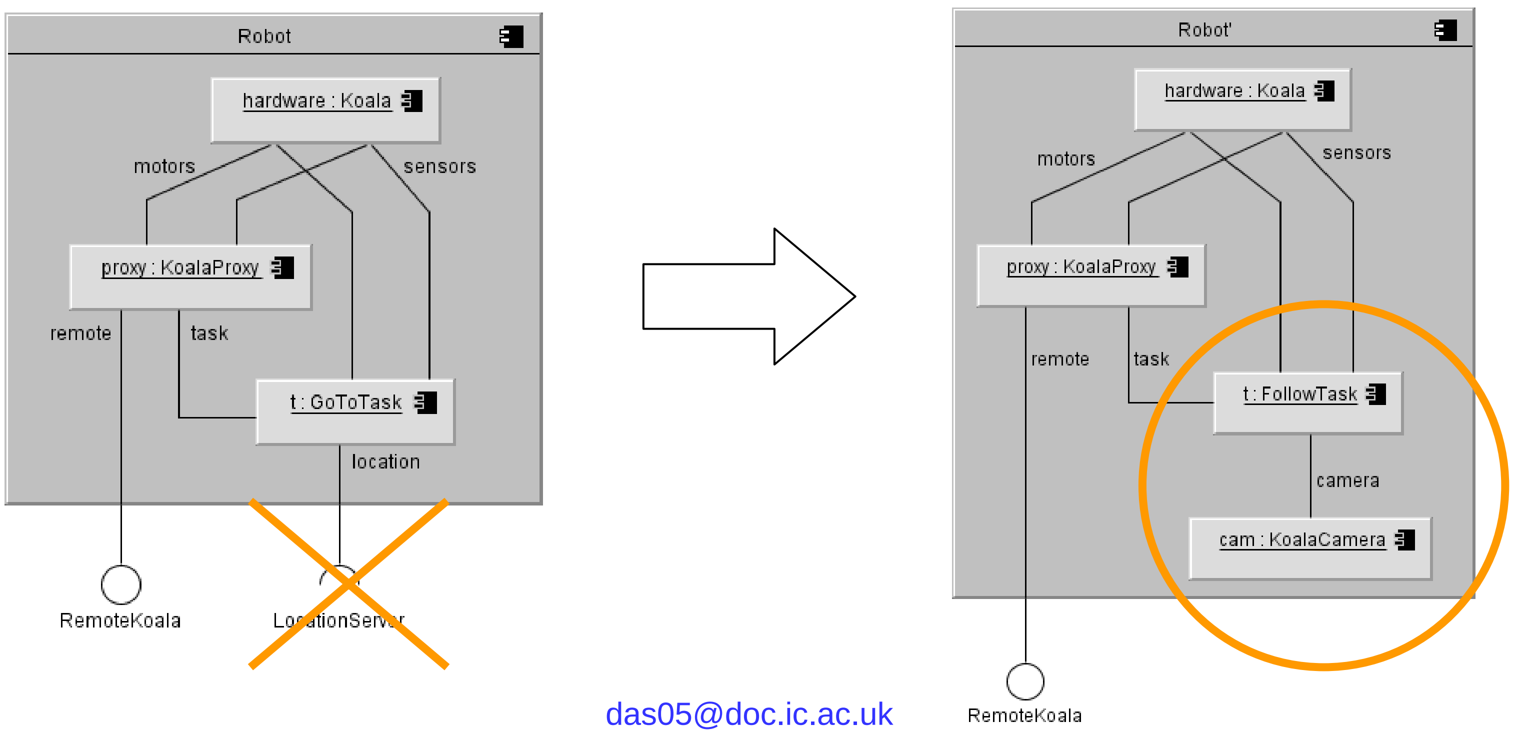


Component properties

- {A1,B2} and {A2,C} may have very different characteristics
  - Power usage, reliability, CPU use, network use, number of changes to existing configuration
  - Further structural constraints
- Ideal selection would account for these non-functional attributes
- Suppose A1 has low reliability, but low CPU use; A2 has high reliability, but high CPU use
- Need to prioritise CPU use versus reliability to make a choice

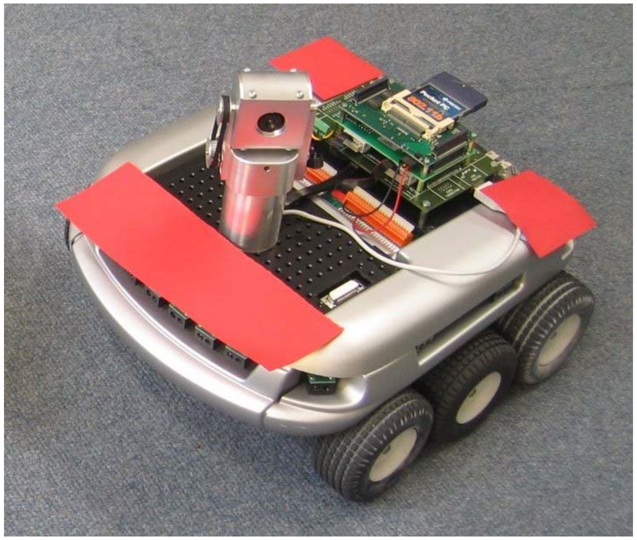
Adaptation

- Components that ‘fail’ at runtime invoke the selection process
- ‘Failed’ component marked as unavailable
- If no alternatives can be used, replanning may be necessary

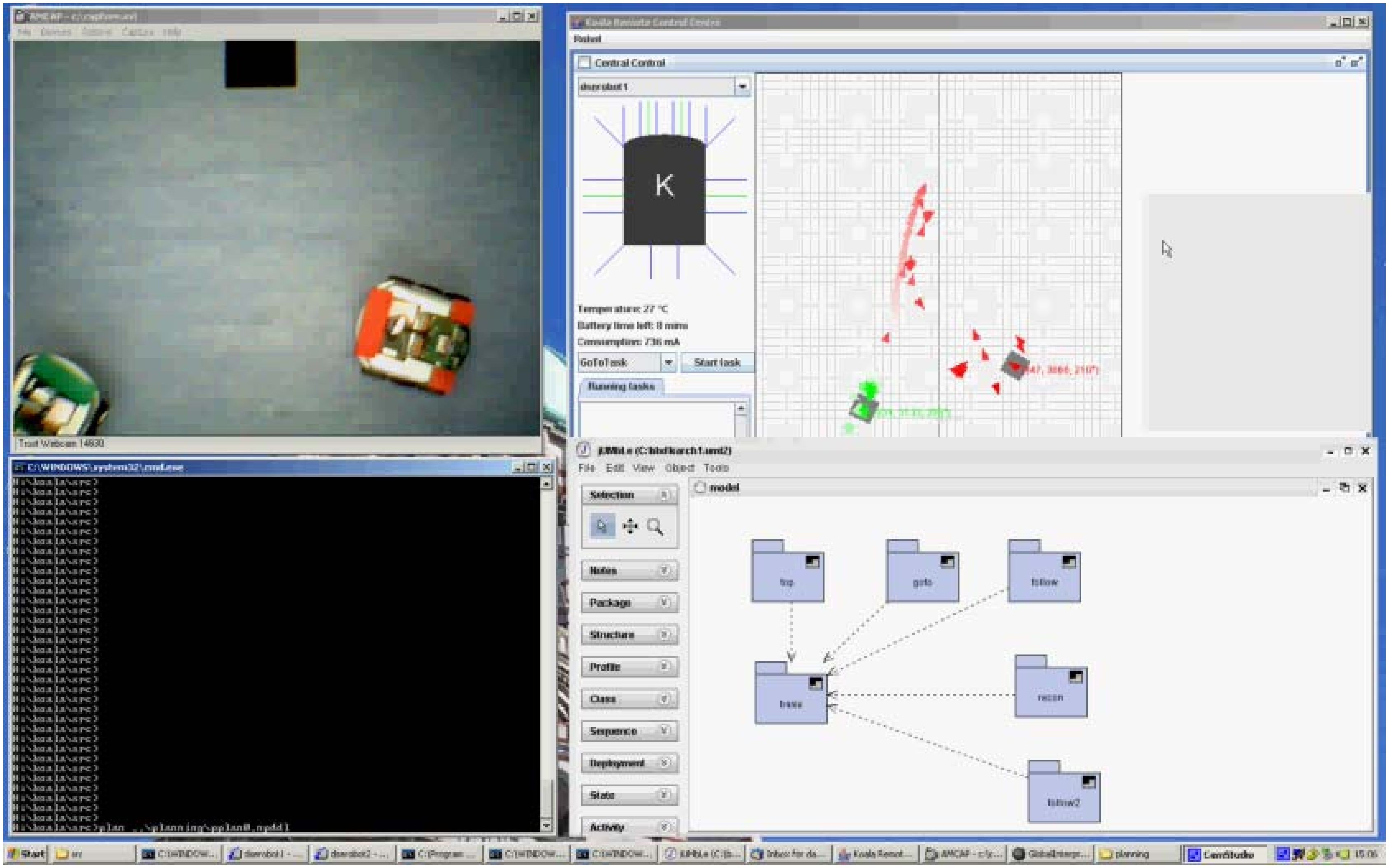


Implementation

- Implemented component selection from NPDDL plans generated from goals on Koala robotic platform
- Components implemented in Java, using the Backbone system
- Goals such as “ensure the ball is in location 1”
- Plans involve moving around, picking up, and so on



Videos at
www.doc.ic.ac.uk/~das05/



Central Control
K
Temperature: 27 °C
GoToTask
Start task
Running tasks
File Edit View Object Tools
model
Selection
Notes
Package
Structure
Profile
Class
Sequence
Deployment
State
Activity
top
goto
follow
recon
follow2
Start

Ongoing work

- Replanning when necessary
- Dynamic modification of goals and domain
- Incorporate non-functional properties into selection process
- Address safety issues in changing components at runtime – quiescence

Conclusions

- Plans provide a convenient source of functional requirements
- Reactive plans cope with non-determinism in environment
- Components selected at runtime based on mapping from action to interface and on availability
- Adaptation achieved by selecting alternatives after a fault
- Working towards ‘safer’ dynamic adaptation