Thank you for your interest

Vlad Estivill-Castro

School of Information and Communication Technology

Institute for Intelligent and Integrated Systems (many collaborators)

Modelling Behaviour Requirements for Automatic Interpretation, Simulation and Deployment

David Billington,
Vladimir Estivill-Castro,
René Hexel, and
Andrew Rock

Robots in Human Environments

Implications for:

Software Engineering for Robots

Reasoning

Human Computer Interaction

Agent technology / Game Theory

How do you describe the behavior as an everyday person? (to your robot / companion)

- There is a declarative part
  - a context, a description
    - ontology (?) knowledge representation?
  - If formal (unambiguous), needs a logic
- There is a state - transition - action part
  - Formally, an algorithm in a formal model of computation

Specifying a behavior

- It should be natural to the human
  - For the declarative parts, mechanisms used by humans should be provided
    - common sense reasoning
    - non-monotonic logic
- Mechanism should be simple to learn
- Formal to remove ambiguity
- Implementable (interpreter/compiler)

Illustration

- Naturally to develop rules systems where the new rules redefine exception to the previous ones
  - 3 laws of robotics
    1. A robot may not harm a human
    2. A robot must obey a human unless it contradict law 1
    3. A robot must protect itself unless contradicts rule 1 or 2
  - Ripple down rules
    - Rules are defined and new rules are subsequently added to revise the cases not covered by the more general rules
    - A tree that is a hierarchy of rules
      - No formal reasoning

Proposal for engineering the behavior

- Using visual descriptions of the behaviour that incorporate formal logic
- Engineers use diagrams to model artefacts.
- Software Engineering has traditionally used diagrams to convey characteristics and descriptions of software

- High-level tools
- Observations:
  - Specifying behaviour unambiguously is difficult
  - Interpret human descriptions of behaviour is also difficult

Requirement Engineering

- CASE (Computer Assisted Software Engineering)
  - graphical models
  - code generation
- Bottom-up approach
- Elude the very large syntax and semantics of OMG modeling (standard) languages
  - for example : UML [2.0]

Requirement Engineering

- Minimize software faults
  - disambiguate requirements
    - completeness
    - consistency
  - validate requirements
    - correctness
  - model / simulate requirements
    - platform independence
  - traceability of evolution / change in requirements
  - communicate requirements
  - implement requirements (automation)

Modelling behaviours

- Computer Assisted Software Engineering enables the manipulation of modelling diagrams and the generation of code from the models.
- We introduce diagrams that use logic to describe behaviour.
- Our proposal extends techniques like Finite State Machines, Petri Nets Object Models for Object Orientation, and Behaviour Trees.
- We model the relationship between several inputs as asserted conditions about the environment that an agent can reason about (using logics) and resolve with respect to knowledge of the environment.

Formal Logics (declarative)

For the description of the behaviour

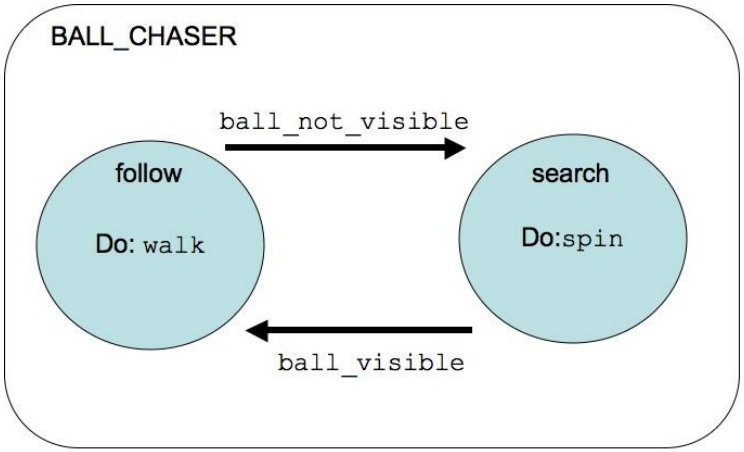
Advantages

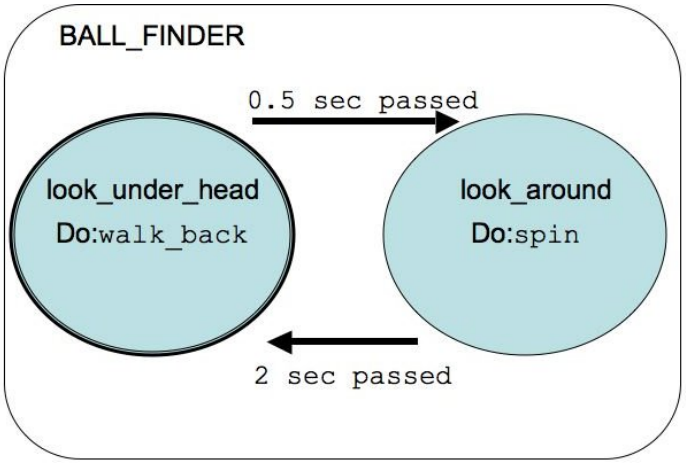
1. Descriptions are unambiguous
   - Descriptions have specific meanings.
2. Ease of description - descriptive
   - Focus is on what the behaviour does, not how it happens
3. Can be translated to implementations in imperative languages like C++, Java
4. Understandable by humans
   - Can be the result of a knowledge engineering exercise
   - Usually humans describe exceptions and laws governing many situations in this way

Disadvantages

1. Can lead to undecidable settings or other difficulties for implementation, like very large and/or inefficient programs

Illustrating state diagrams





- Exclusivity

$c_i \wedge c_j$ = **false** $\forall\ i \neq j$

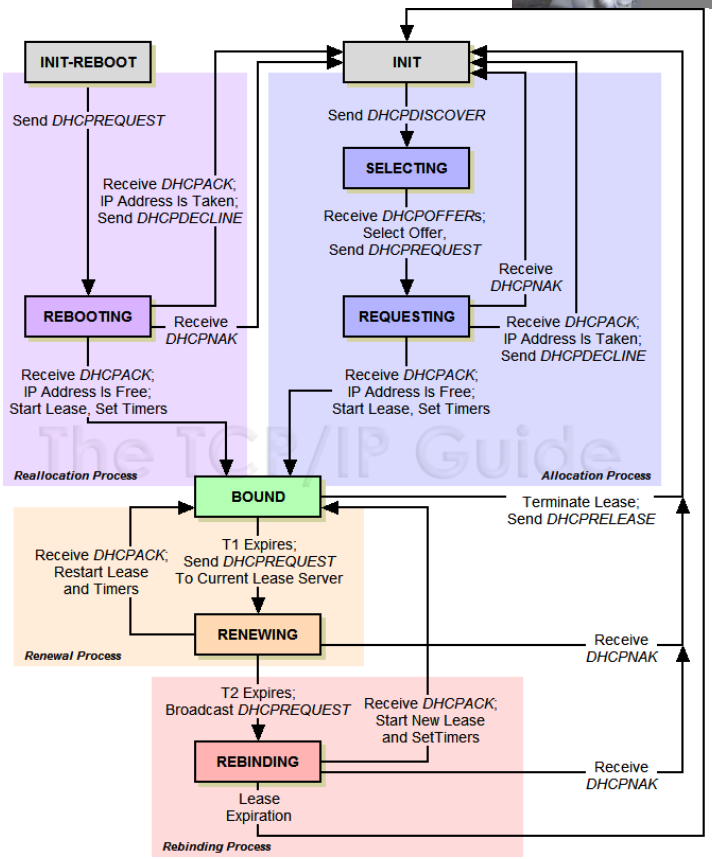
- Exhaustivity

$\bigvee_{i=1}^{n} c_i$ = **true**

s_1	$c_1 = event_u$	s_i
s_1	$c_2 = event_v$	s_j
s_i	$c_t = event_x$	s_p

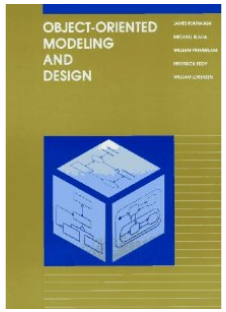
State diagrams (action)

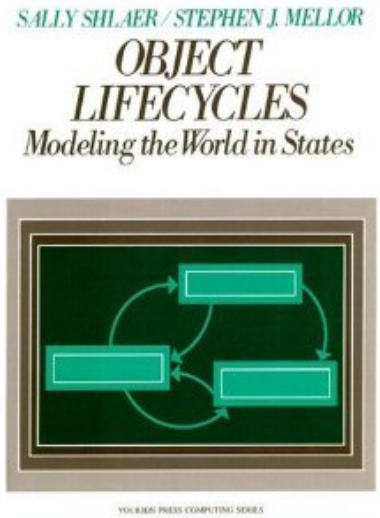
- Correspond naturally to the notion of state machine
- Already very common in many human-computer interfaces
  - elevators/mobile phones/ washing machines
- Formal semantics (formal mathematical object)



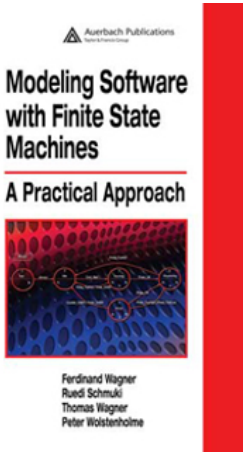
State diagrams (action)

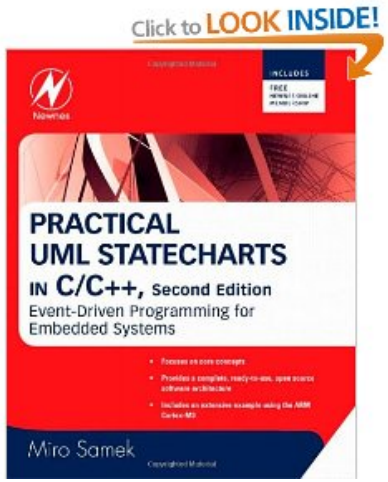
- Widely used in Software Engineering
  - OMT, then UML, Shlaer-Mellor

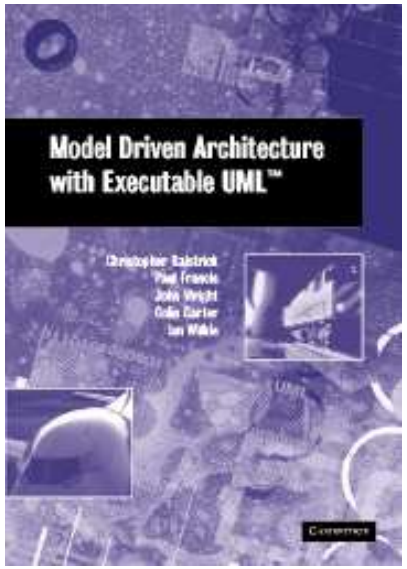




- Widely successful tool in industry
  - StateWorks, executableUML



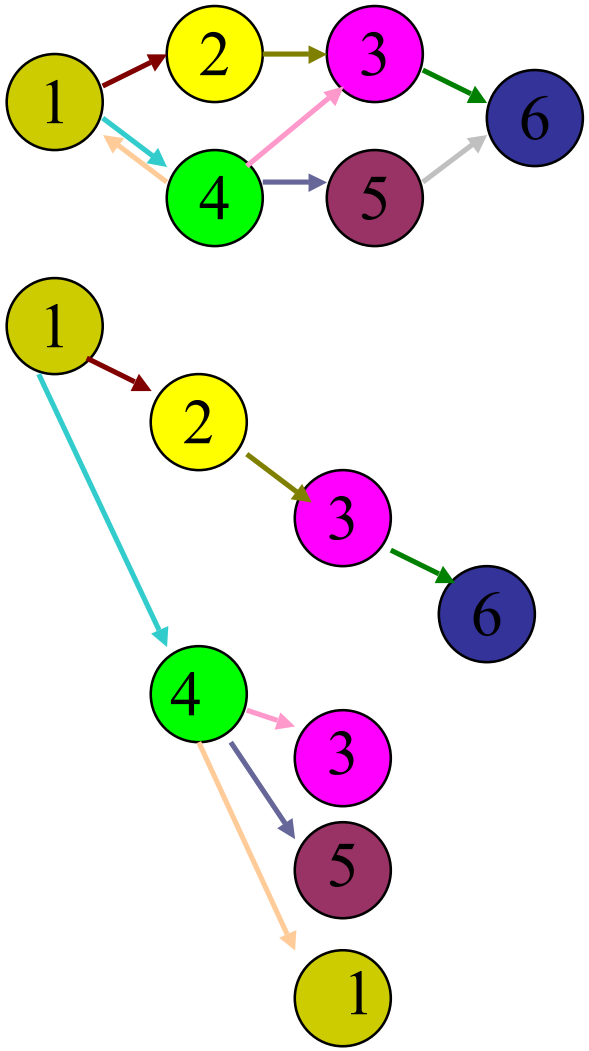




State Machines

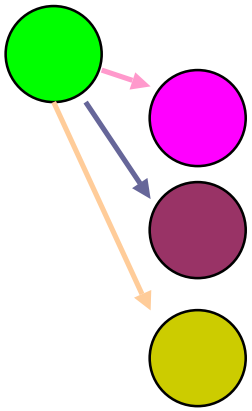
- Some extension and equivalences to other formal models
- Multi-threaded State Machines
- Petri Nets
- Distributed computation
- Team automata
- Security formalisms (verification)

Convert State Diagram into Behaviour Tree



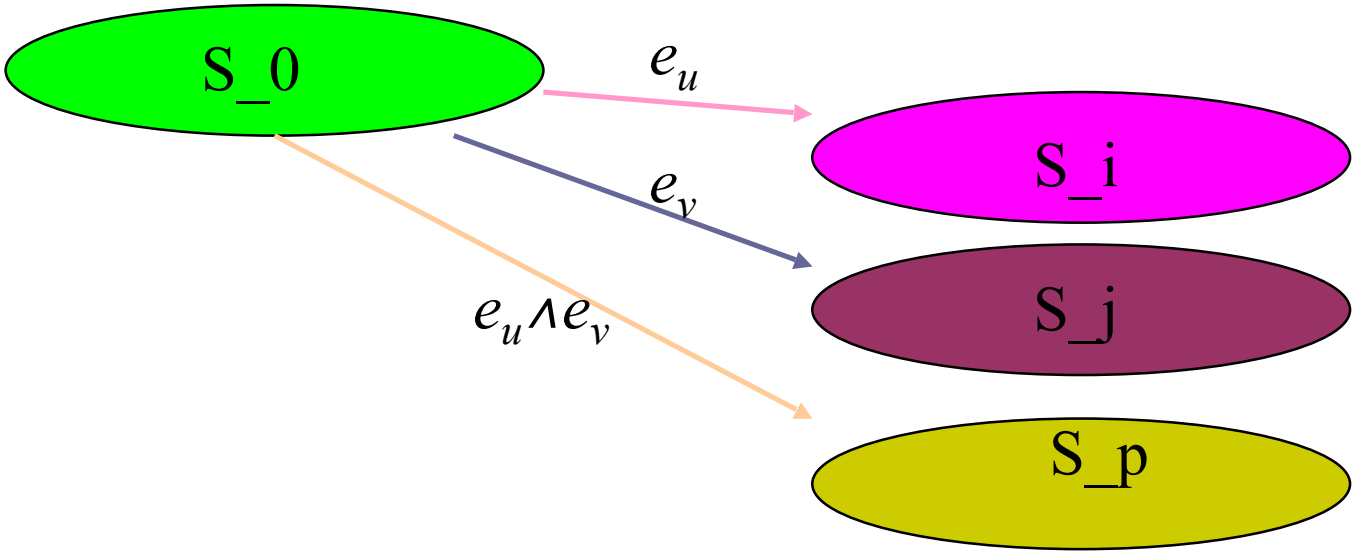
- Draw down by breadth-first search
- Already visited nodes are cloned but not explored again

Convert a node in the tree to a module in Plausible Logic



1. name(Node).
2. type State_Type(S_0, S_1,...,S_k).
3. v{State(S_0),…,State(S_k)}.
4. v{¬State(S_i),¬State(S_j)}. (∀ i ≠ j)
5. input{“e_i”}. (for i=1,…,k}
6. Default: ⇒ State(S_0).
7. Switch_S_0_S_i:{“e_i”} ⇒ State(S_i). (for i=1,…,k)
8. Switch_S_0_S_i > Default. (for i=1,…,k)

Using the priority relation

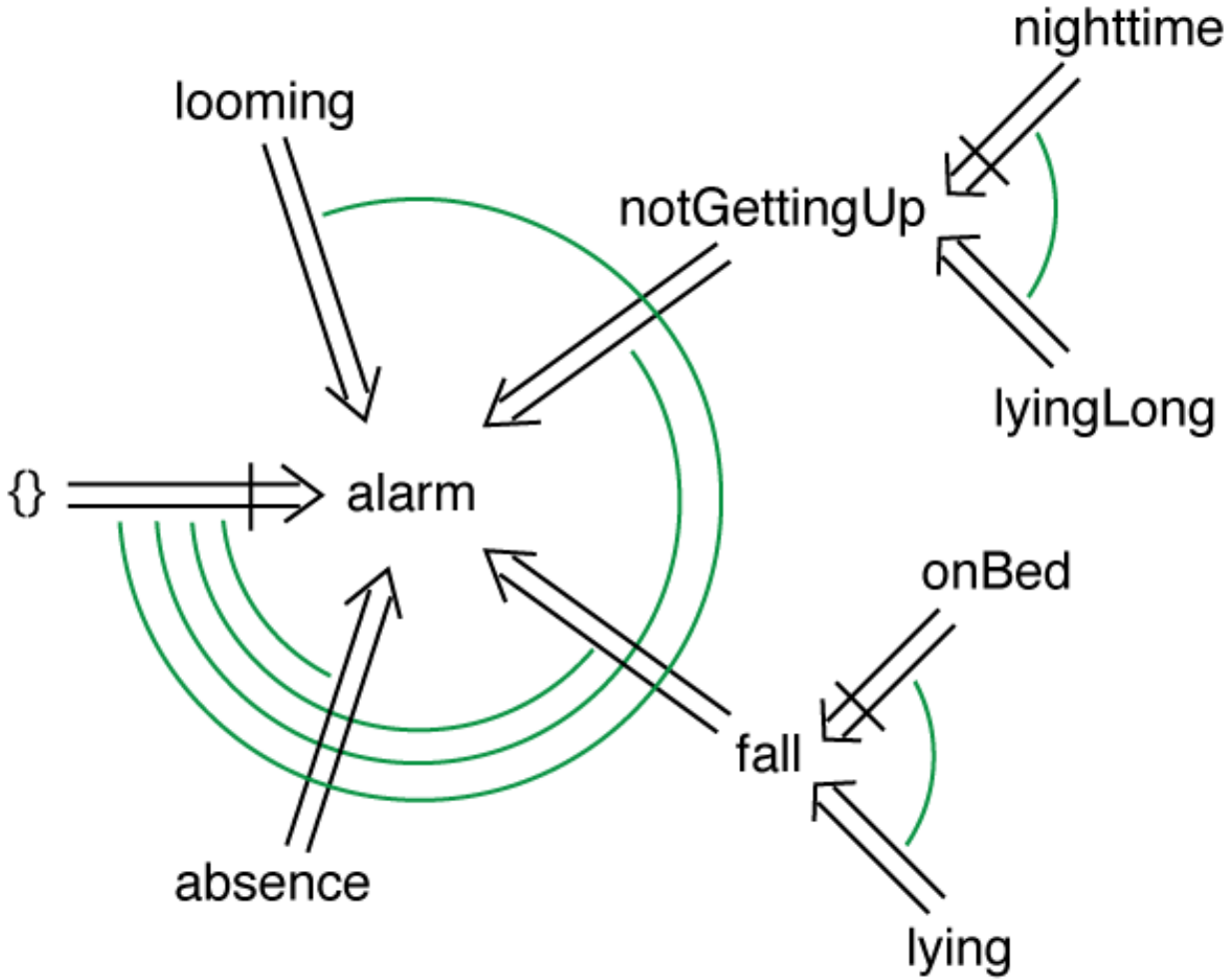


```
1.  Switch_S_0_S_i:{"e_u"} ⇒ State(S_i).
2.  Switch_S_0_S_i > Default.
3.  Switch_S_0_S_j:{"e_v"} ⇒ State(S_j).
4.  Switch_S_0_S_j > Default.
5.  Switch_S_0_S_p:{"e_v∧e_u"} ⇒ State(S_p).
6.  Switch_S_0_S_p > Default.
7.  Switch_S_0_S_p > Switch_S_0_S_i.
8.  Switch_S_0_S_p > Switch_S_0_S_i.
```

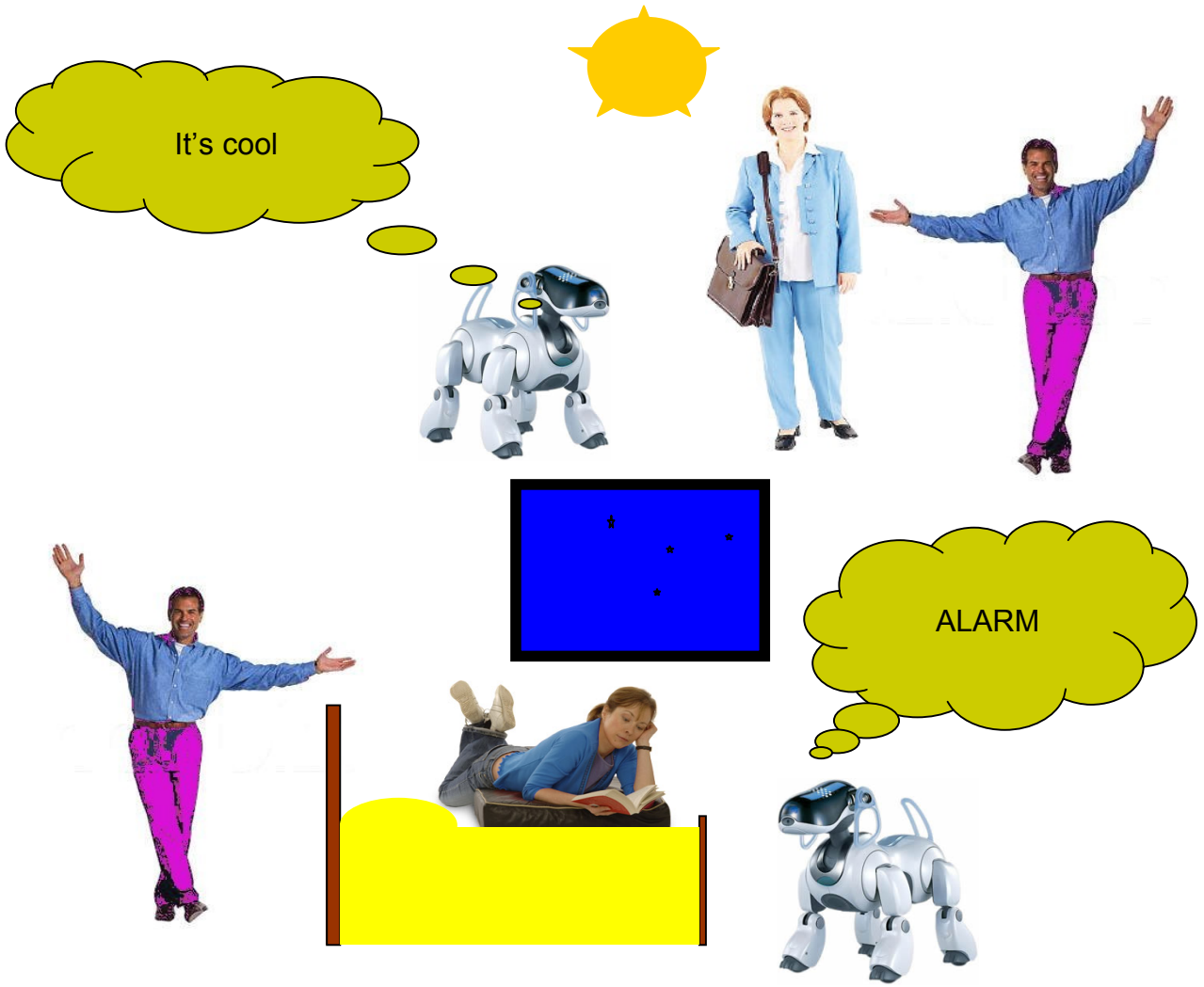
A logic for looking after the lady

1. Usually there is no reason for alarm
2. The absence of owner for a long time is reason for alarm (this takes precedence over rule 1)
3. Lying usually results from a fall
4. A fall is usually a reason for alarm (this takes precedence over rule 1)
5. Being on bed is not a fall (this takes precedence over rule 4)
6. Lying for a long time means owner is not getting up.
7. Not getting up is a reason for alarm (this takes precedence over rule 1)
8. If it is night, it is fine not to get up (this takes precedence over rule 7)
9. If there is a stranger looming over the lady, it is reason for an alarm (takes precedence over rule 1)
10. Owner can't be absent while on bed, or lying or lying for a long time.
11. Owner can't be lying for a long time without lying for a short time.

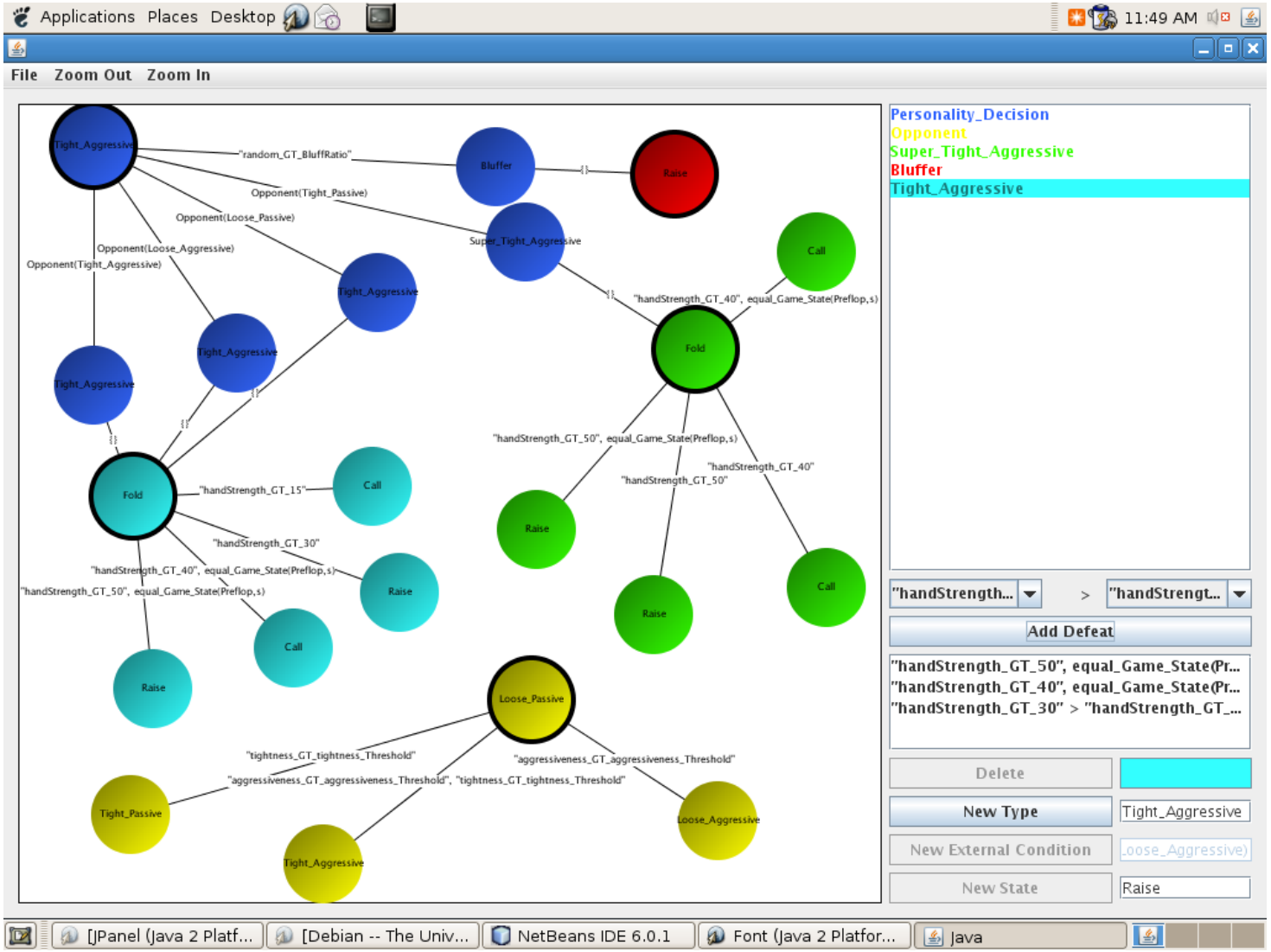
Diagrams to illustrate rule relations



Prototype demonstrated at RoboCup@Home 2007



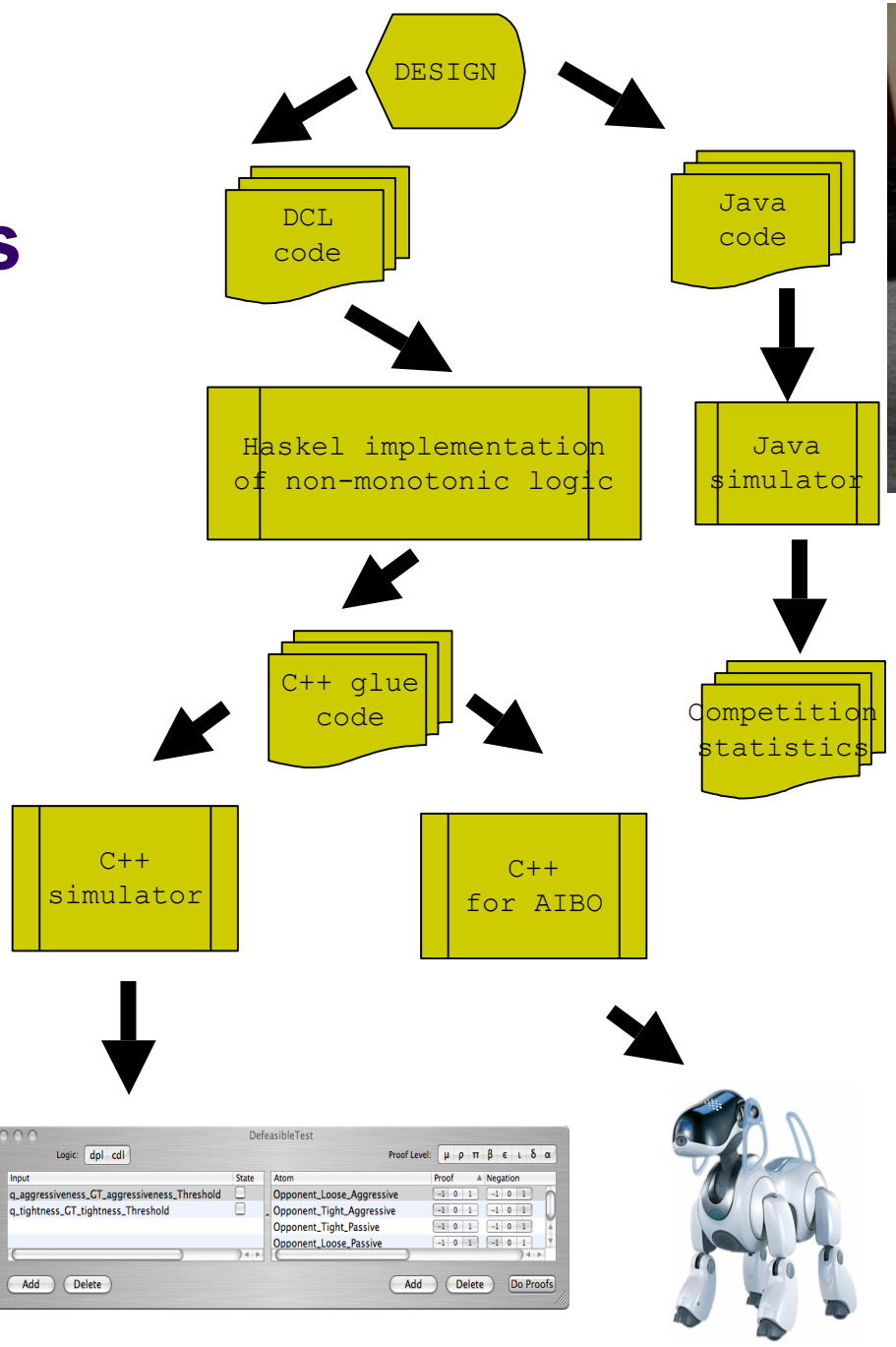
A diagram for a poker player



Code generated (example)

```
/* This is code Generated by the DPLGenerator
** This program was made by Mark Johnson 2008 (MiPAL)
** File Opponent.d
*/

name{Opponent}.

type Opponent(x<-Opponent_Type).

type Opponent_Type = {Loose_Passive, Loose_Aggressive, Tight_Passive, Tight_Aggressive}.

\/{Opponent(Loose_Passive), Opponent(Loose_Aggressive), Opponent(Tight_Passive), Opponent(Tight_Aggressive)}.

\/{~Opponent(Loose_Passive),~Opponent(Loose_Aggressive)}.
\/{~Opponent(Loose_Passive),~Opponent(Tight_Passive)}.
\/{~Opponent(Loose_Passive),~Opponent(Tight_Aggressive)}.
\/{~Opponent(Loose_Aggressive),~Opponent(Tight_Passive)}.
\/{~Opponent(Loose_Aggressive),~Opponent(Tight_Aggressive)}.
\/{~Opponent(Tight_Passive),~Opponent(Tight_Aggressive)}.

input{"aggressiveness_GT_aggressiveness_Threshold"}.
input{"tightness_GT_tightness_Threshold"}.

Default_Opponent: {}=>Opponent(Loose_Passive).

Switch_aggressiveness_GT_aggressiveness_Threshold: {"aggressiveness_GT_aggressiveness_Threshold"} => Opponent(Loose_Aggressive).
Switch_aggressiveness_GT_aggressiveness_Threshold > Default_Opponent.

Switch_tightness_GT_tightness_Threshold: {"tightness_GT_tightness_Threshold"} => Opponent(Tight_Passive).
Switch_tightness_GT_tightness_Threshold > Default_Opponent.

Switch_aggressiveness_GT_aggressiveness_Threshold_n_tightness_GT_tightness_Threshold: {"aggressiveness_GT_aggressiveness_Threshold",
        "tightness_GT_tightness_Threshold"} => Opponent(Tight_Aggressive).
Switch_aggressiveness_GT_aggressiveness_Threshold_n_tightness_GT_tightness_Threshold > Default_Opponent.

Switch_aggressiveness_GT_aggressiveness_Threshold_n_tightness_GT_tightness_Threshold > Switch_tightness_GT_tightness_Threshold.
Switch_aggressiveness_GT_aggressiveness_Threshold_n_tightness_GT_tightness_Threshold > Switch_aggressiveness_GT_aggressiveness_Threshold.
```

Earlier Process to Embed Design into the AIBO Robot



Systems interacting with humans

A classical example

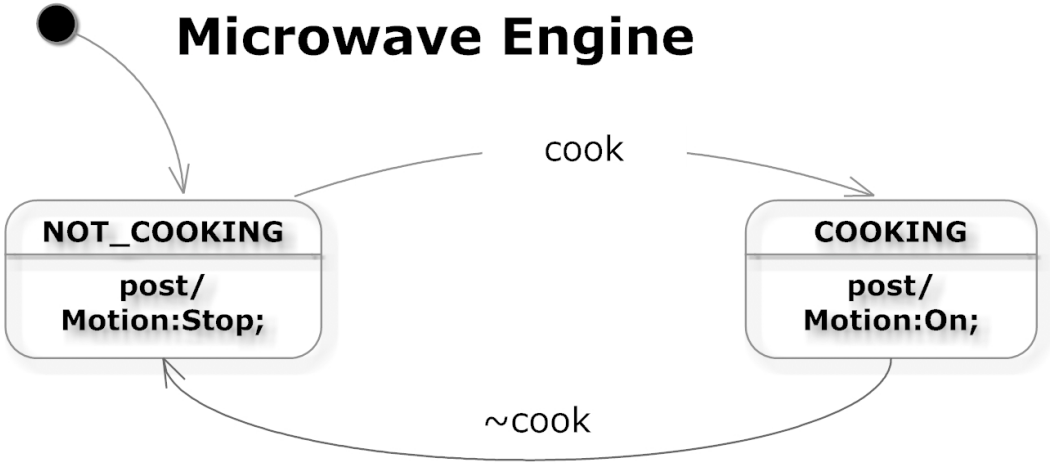
- The One-Minute Microwave Oven
  - literature approach
  - behavior specification of all objects of a class
    - Shlaer-Mellor
  - StateWorks
  - Behavior Trees
  - PetriNets
  - SCXML - State Chart XML: State Machine Notation for Control Abstraction

Requirements (One-Minute Microwave Oven)

Requirements	Description
R1	There is a single control button available for the use of the oven. If the oven is closed and you push the button, the oven will start cooking (that is, energize the power-tube) for one minute
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3	Pushing the button when the door is open has no effect.
R4	Whenever the oven is cooking or the door is open, the light in the oven will be on.
R5	Opening the door stops the cooking. and stops the timer and does not clear the timer
R6	Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
R7	If the oven times out, the light and the power-tube are turned off and then a beeper emits a warning beep to indicate that the cooking has finished.

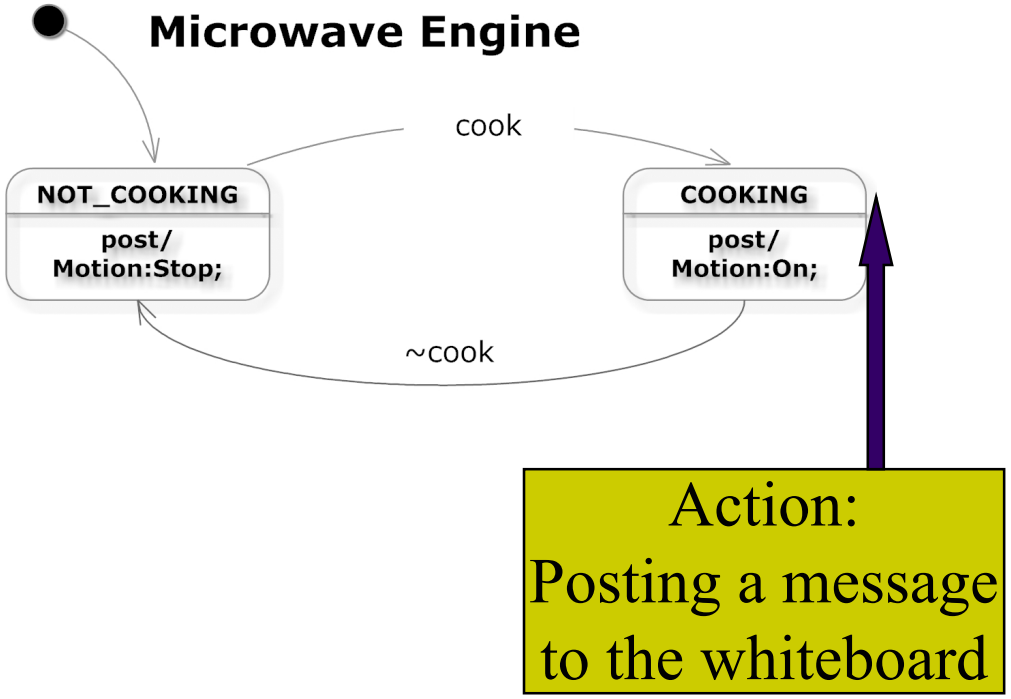
The DPL+State_Machine approach

- Step 1: Consider writing the script of music for an orchestra. Write individual scripts and place together all actuators that behave with the same actions for the same cues
- Example: The control of the tube (energizing), the fan and the spinning plate



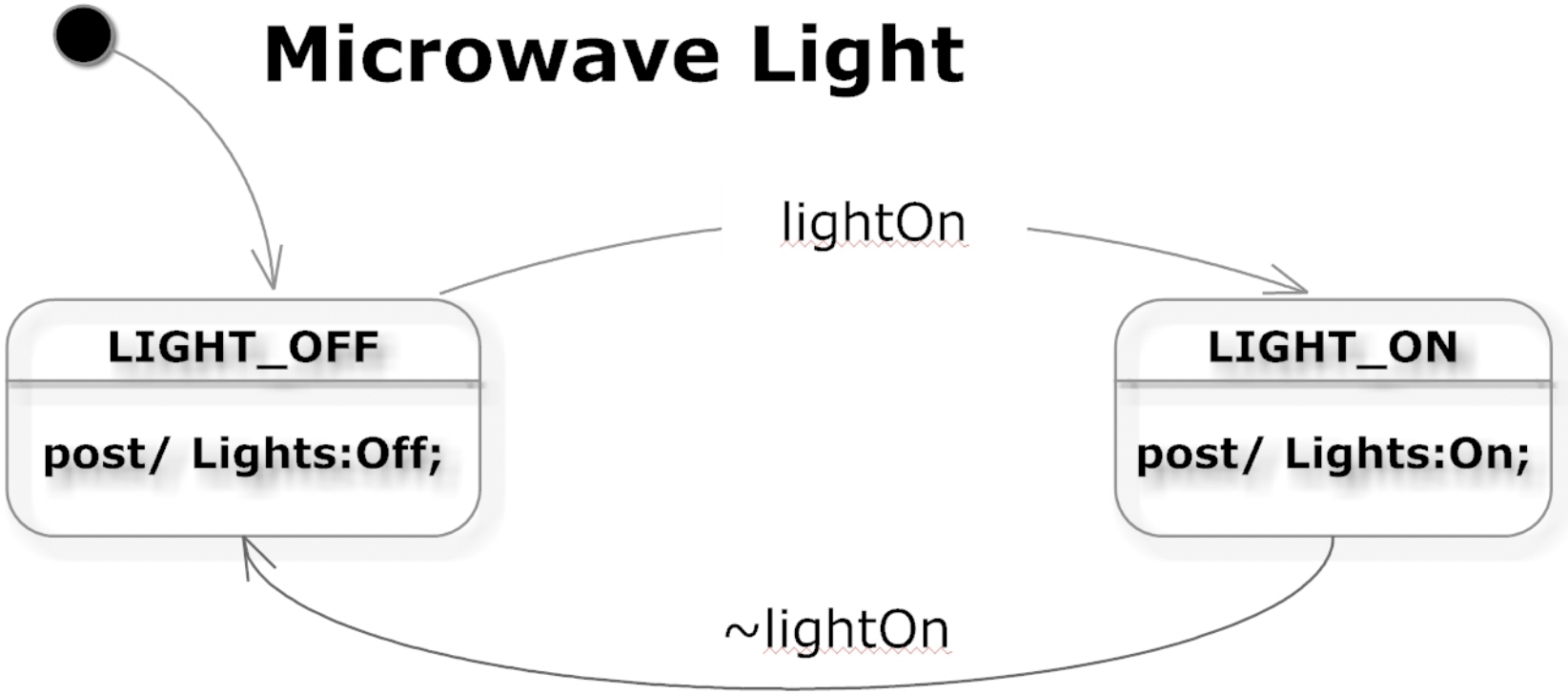
Step 2: Describe the conditions that result in the need to change state

```
% MicrowaveCook.d

name{MicrowaveCook}.

input{timeLeft}.
input{doorOpen}.

C0: {}         => ~cook.
C1: timeLeft =>  cook. C1 > C0.
C2: doorOpen => ~cook. C2 > C1.

output{b cook, "cook"}.
```



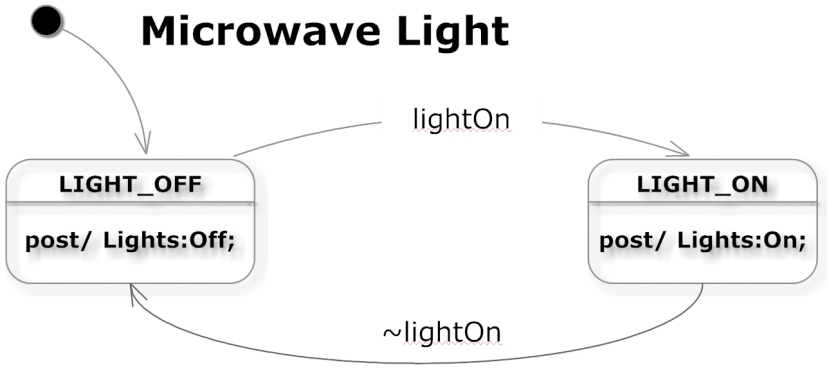
Step 1 (again): Analyze another actuator

- Illustration: The light



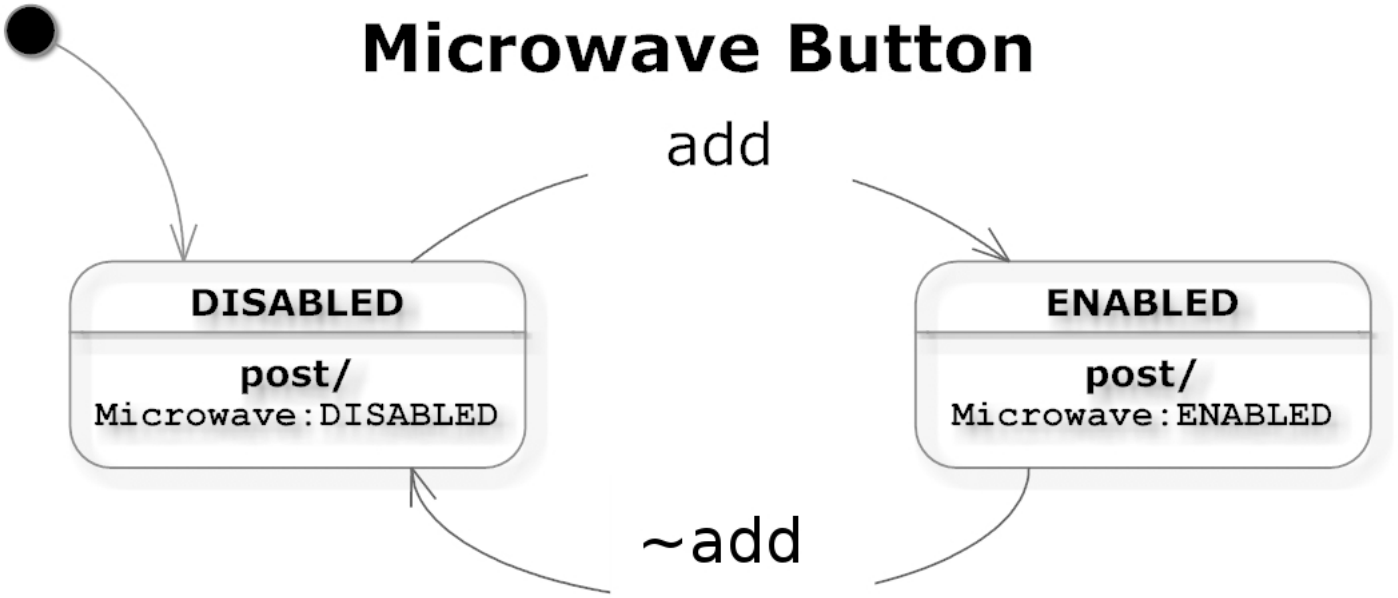
Step 2 (again): Describe the conditions that result in the need to change state

```
% MicrowaveLight.d

name{MicrowaveLight}.

input{timeLeft}.
input{doorOpen}.

L0: {}         => ~lightOn.
L1: timeLeft =>  lightOn. L1 > L0.
L2: doorOpen =>  lightOn. L2 > L0.

output{b lightOn, "lightOn"}.
```



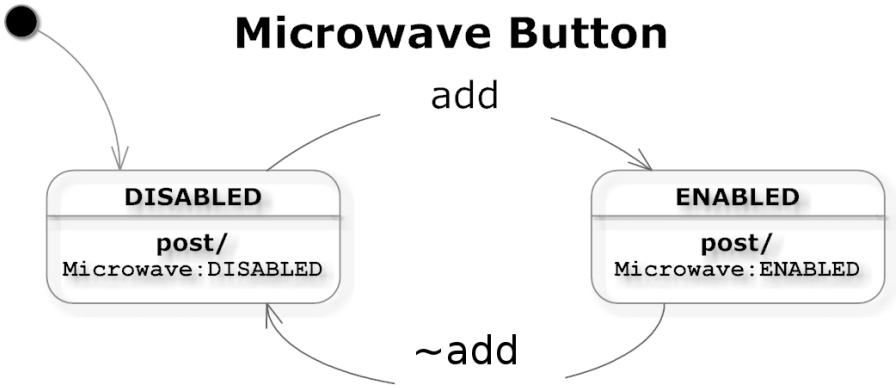
Step 1 (again): Analyze another actuator

- Illustration: The button



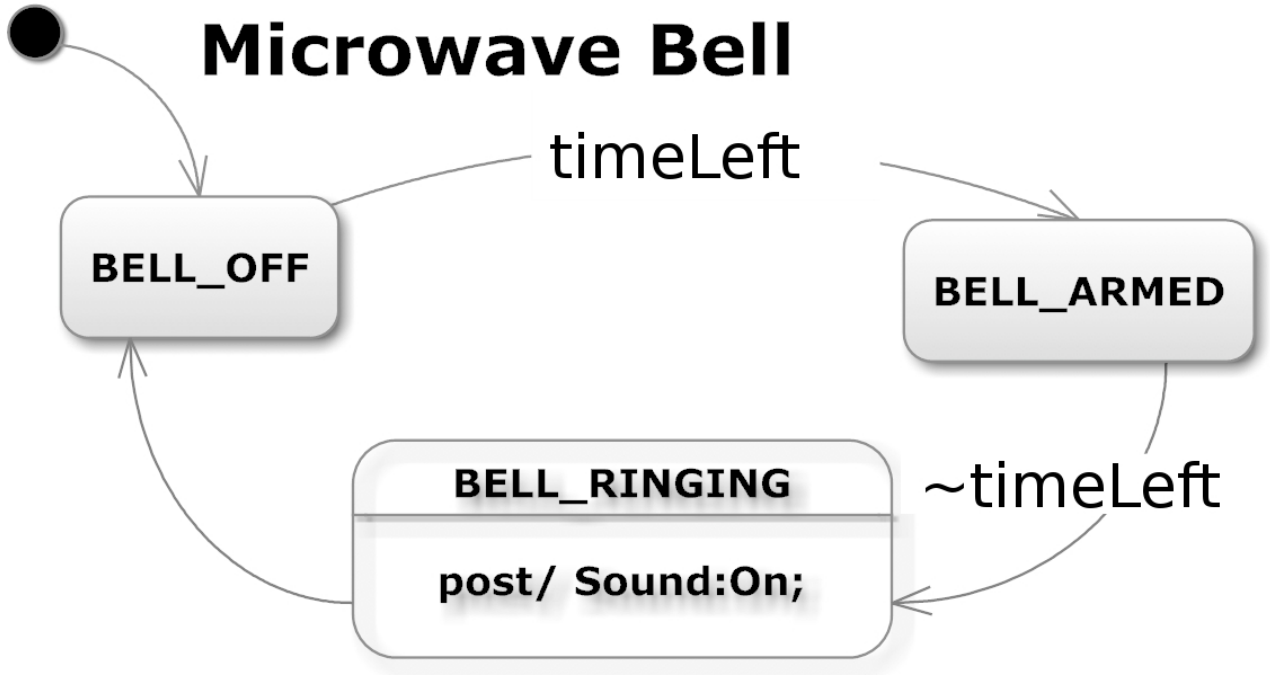
Step 2 (again): Describe the conditions that result in the need to change state

```
% MicrowaveButton.d

name{MicrowaveButton}.

input{doorOpen}.
input{buttonPushed}.


CB0: {}            => ~add.
CB1: buttonPushed =>  add. CB1 > CB0.
CB2: doorOpen     => ~add. CB2 > CB1.


output{b add, "add"}.
```



Step 1 (again): Analyze another actuator

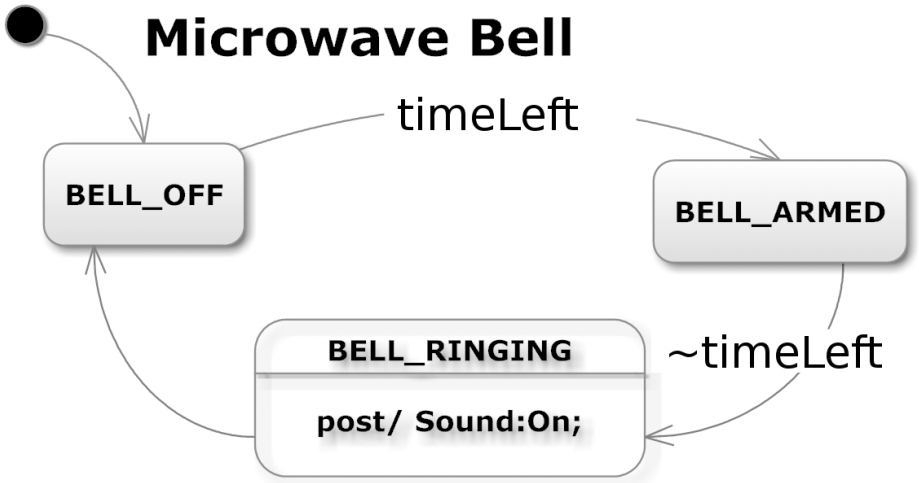
- Illustration: The bell



Step 2 (again): Describe the conditions that result in the need to change state

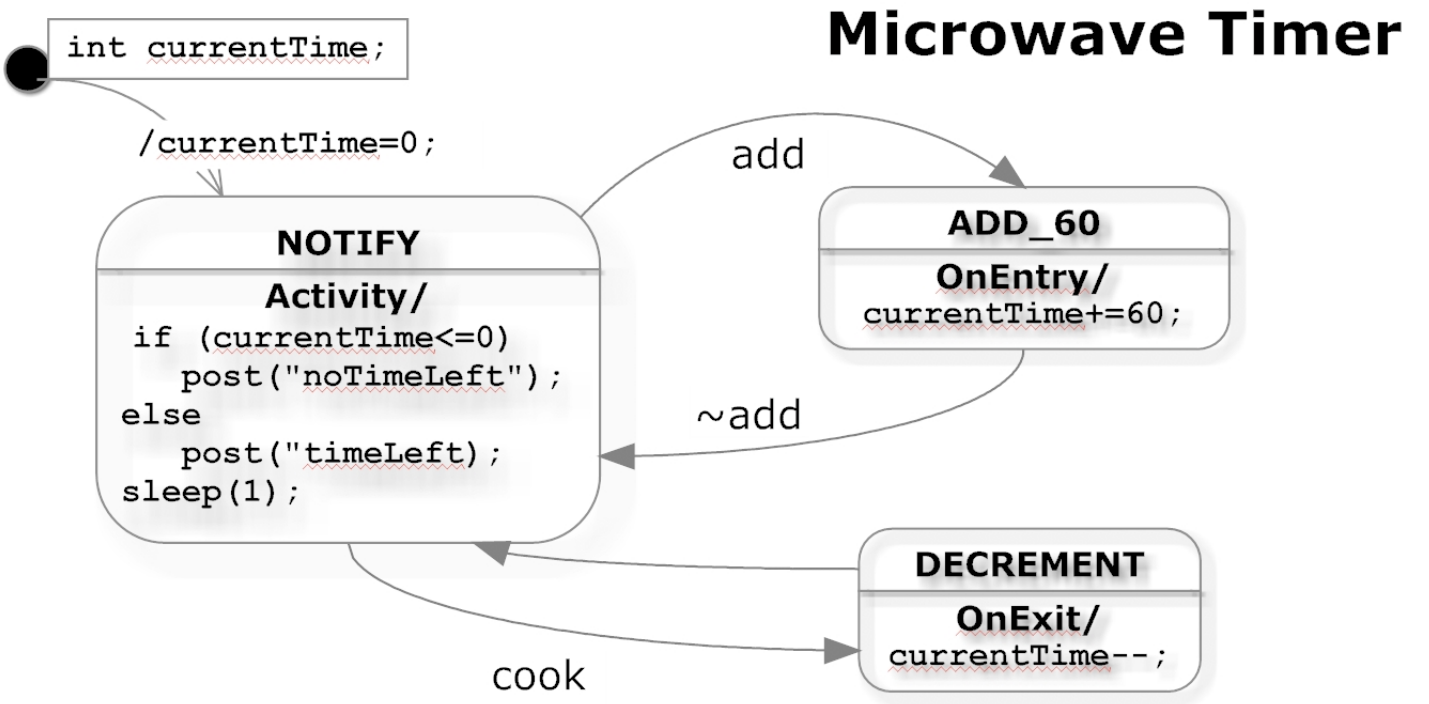
No need for a logic: `timeLeft`

- posted by another module
- do not require a proof



Step 1 (again): Analyze another actuator

- Illustration: The timer



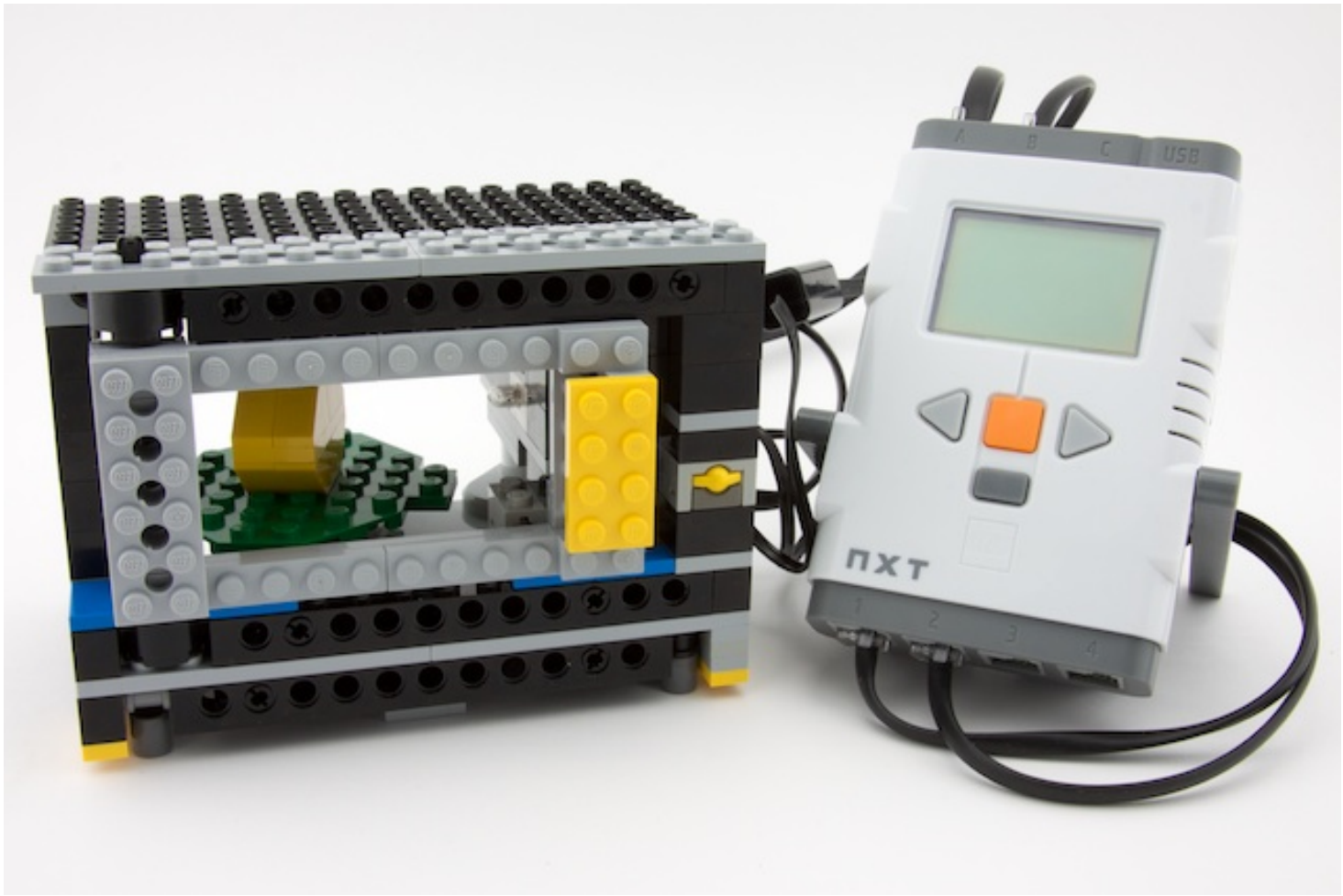
The simple C++ code

```
incrementTimer()
  currentCookTime+=60;

decrementTimer()
  If (currentCookTime>0)
     currentCookTime--;

postTimeValue()
  if (currentCookTime<=0)
     post "~timeLeft";
  else
     post "timeLeft";
  sleep(1);
```

That is all folks!

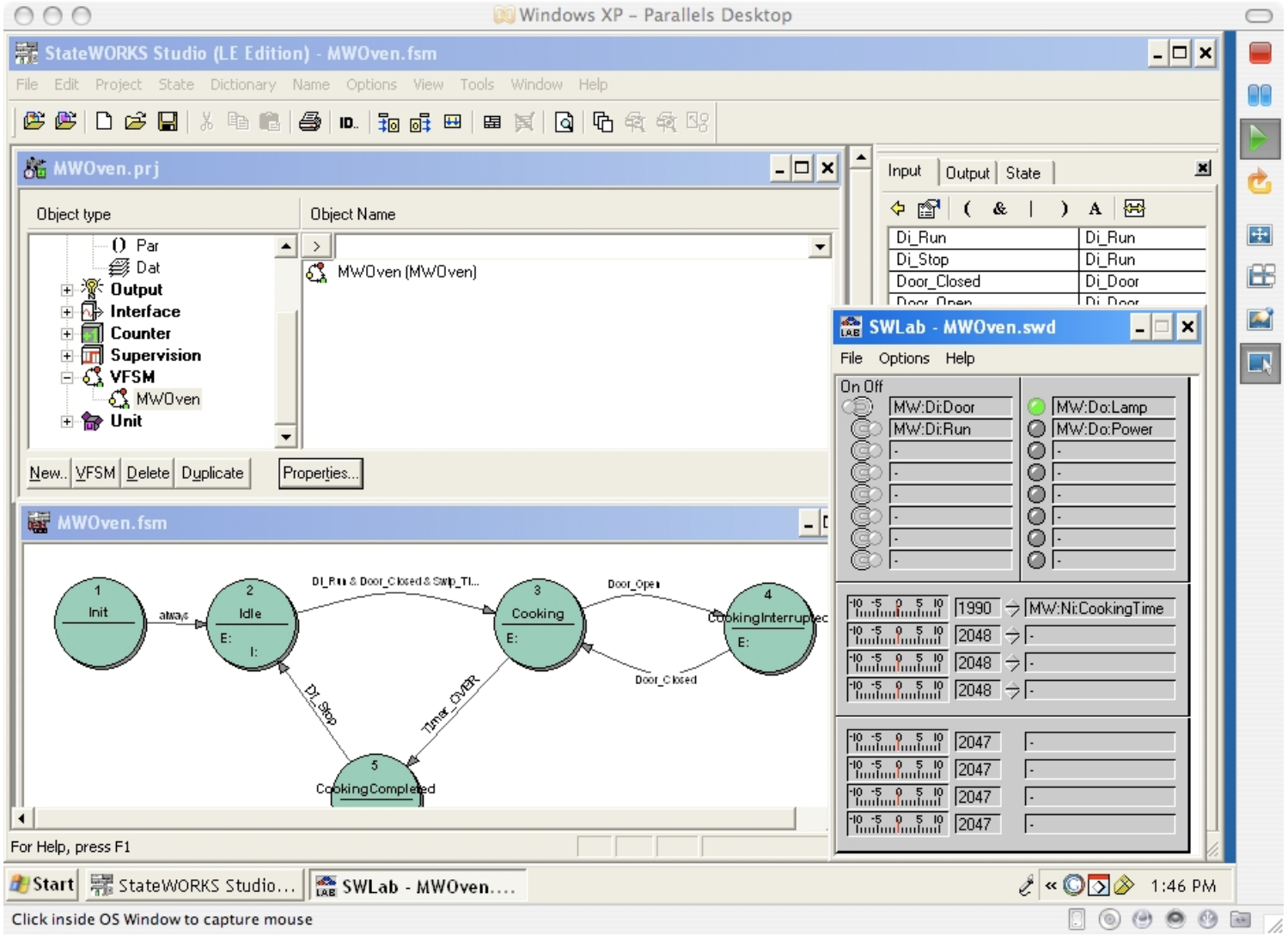
- I
  U

www.youtube.com/watch?v=iEkCHqSfMco

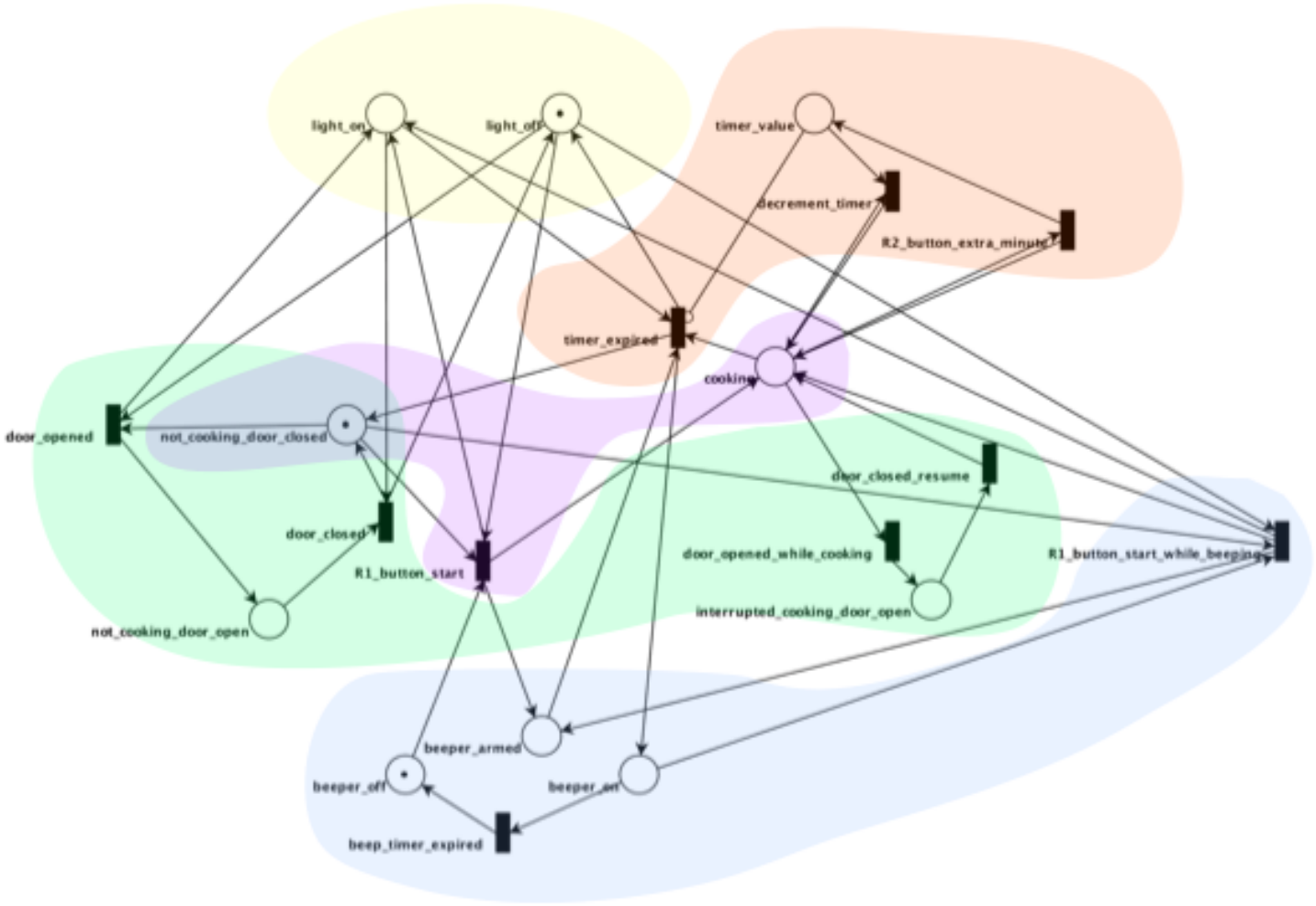


http://www.youtube.com/watch?v=Dm3SP3q9_VE

StateWorks


Windows XP – Parallels Desktop
StateWORKS Studio (LE Edition) - MWOven.fsm
File Edit Project State Dictionary Name Options View Tools Window Help
MWOven.prj
Object type
Object Name
Par
Dat
Output
Interface
Counter
Supervision
VFSM
MWOven
Unit
MWOven (MWOven)
New.. VFSM Delete Duplicate
Properties...
Input Output State
Di_Run Di_Run
Di_Stop Di_Run
Door_Closed Di_Door
SWLab - MWOven.swd
File Options Help
On Off
MW:Di:Door
MW:Di:Run
MW:Do:Lamp
MW:Do:Power
MW:Ni:CookingTime
MWOven.fsm
1 Init
always
2 Idle
E:
I:
Di_Run & Door_Closed & Swip_TI...
3 Cooking
E:
Door_Open
4 CookingInterrupted
E:
Door_Closed
Di_Stop
Timer_OVER
5 CookingCompleted
For Help, press F1
Start
StateWORKS Studio...
SWLab - MWOven....
1:46 PM
Click inside OS Window to capture mouse

Petri Nets


light_on
light_off
timer_value
decrement_timer
R2_button_extra_minute
timer_expired
cooking
door_opened
not_cooking_door_closed
door_closed_resume
door_closed
R1_button_start
door_opened_while_cooking
R1_button_start_while_beeping
not_cooking_door_open
interrupted_cooking_door_open
beeper_armed
beeper_off
beeper_on
beep_timer_expired

Behavior Trees

- Model Behavior Tree

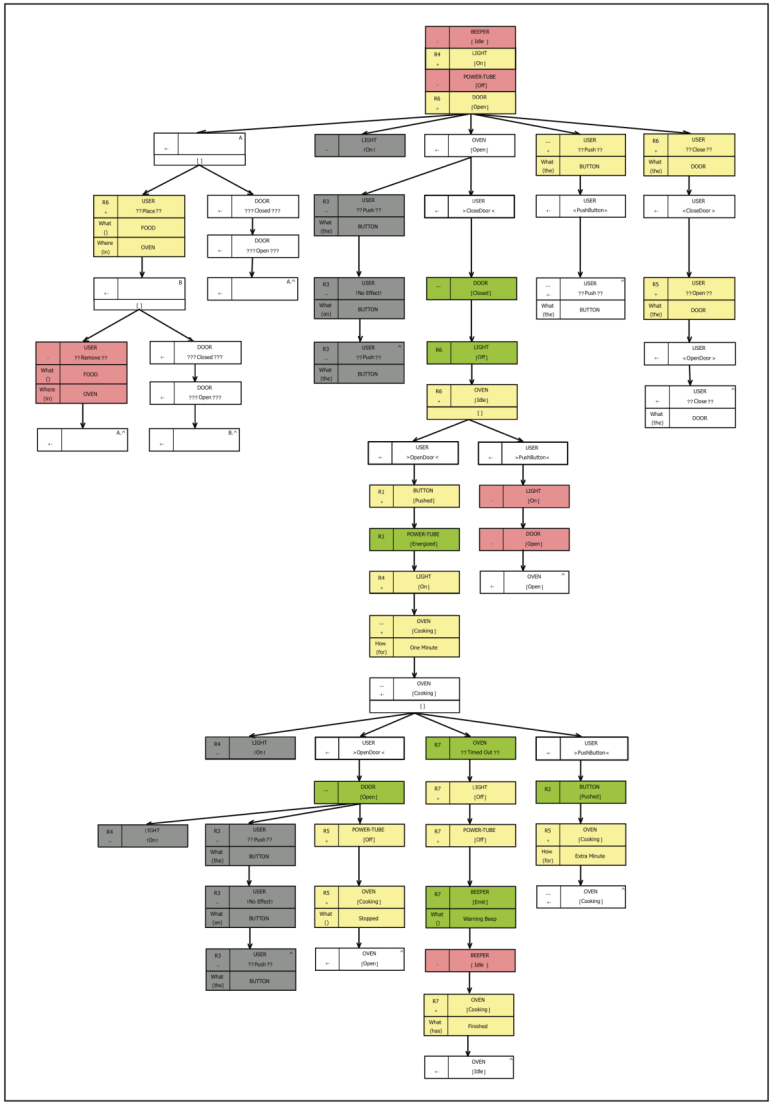


Figure 6. The Model Behavior Tree of the Microwave Oven

Behavior Trees

- Design Behavior Tree

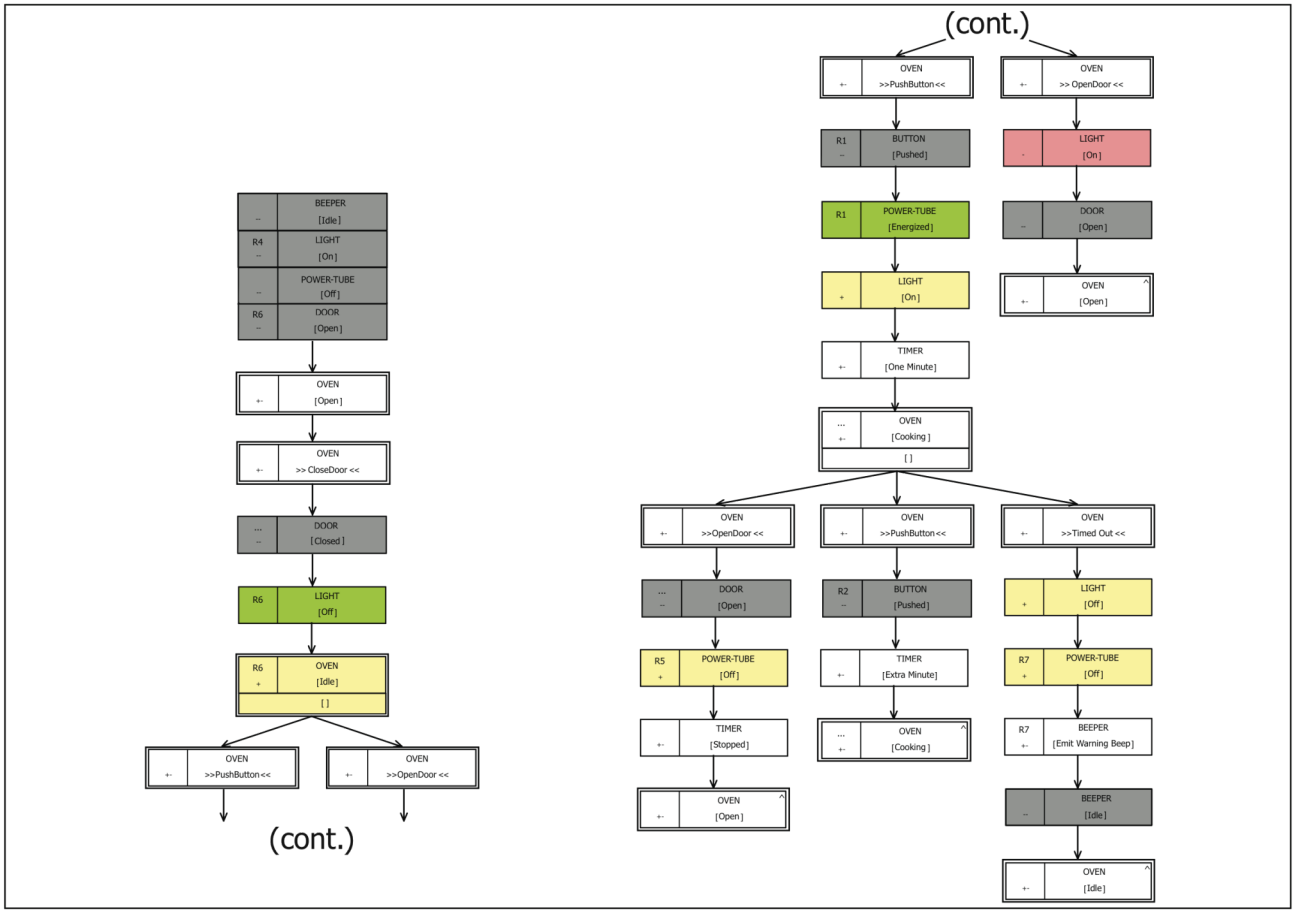


Figure 8. The Design Behavior Tree of the Microwave Oven

Authorized licensed use limited to: GRIFFITH UNIVERSITY. Downloaded on October 25, 2009 at 22:28 from IEEE Xplore. Restrictions apply.

Comparison

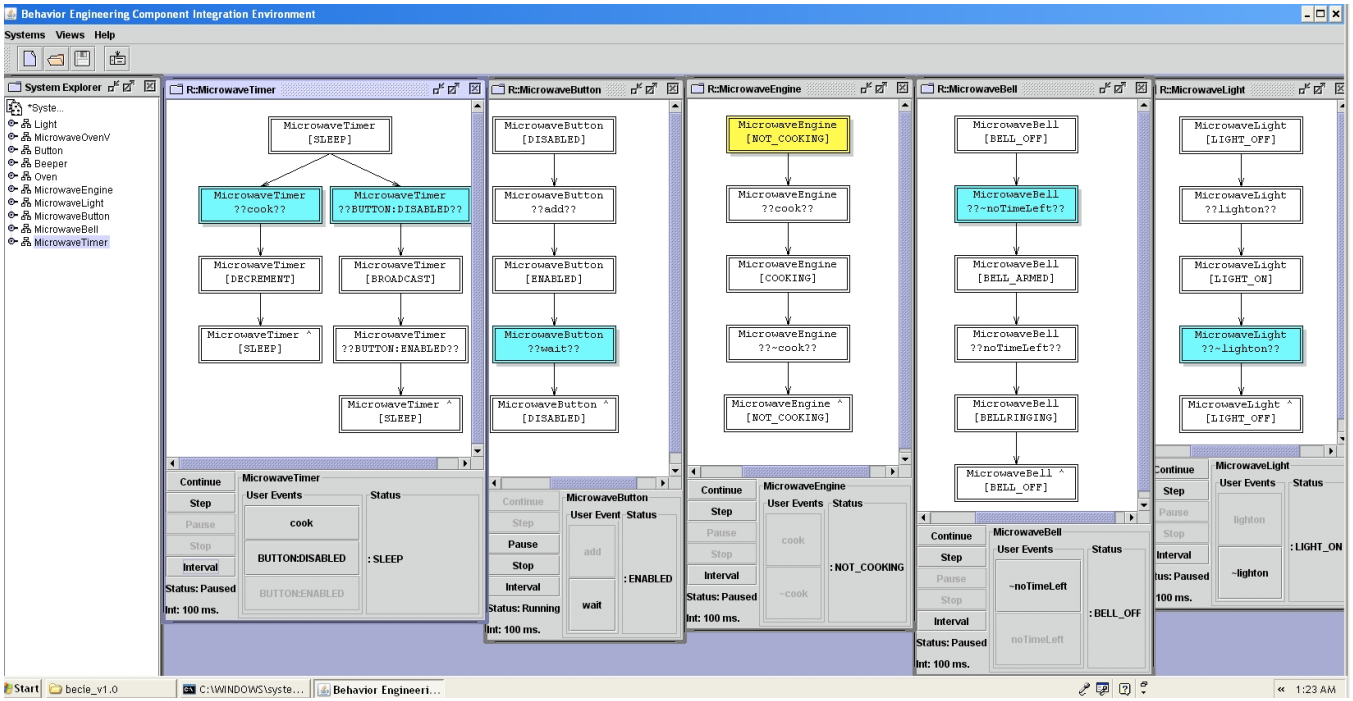
- Far simpler
  - Less states that
    - StateWorks,
    - Behavior Trees

      (less boxes and arrows)
  - Far less crossings that Petri nets

- Behavior Trees miss the alarm (beeper).

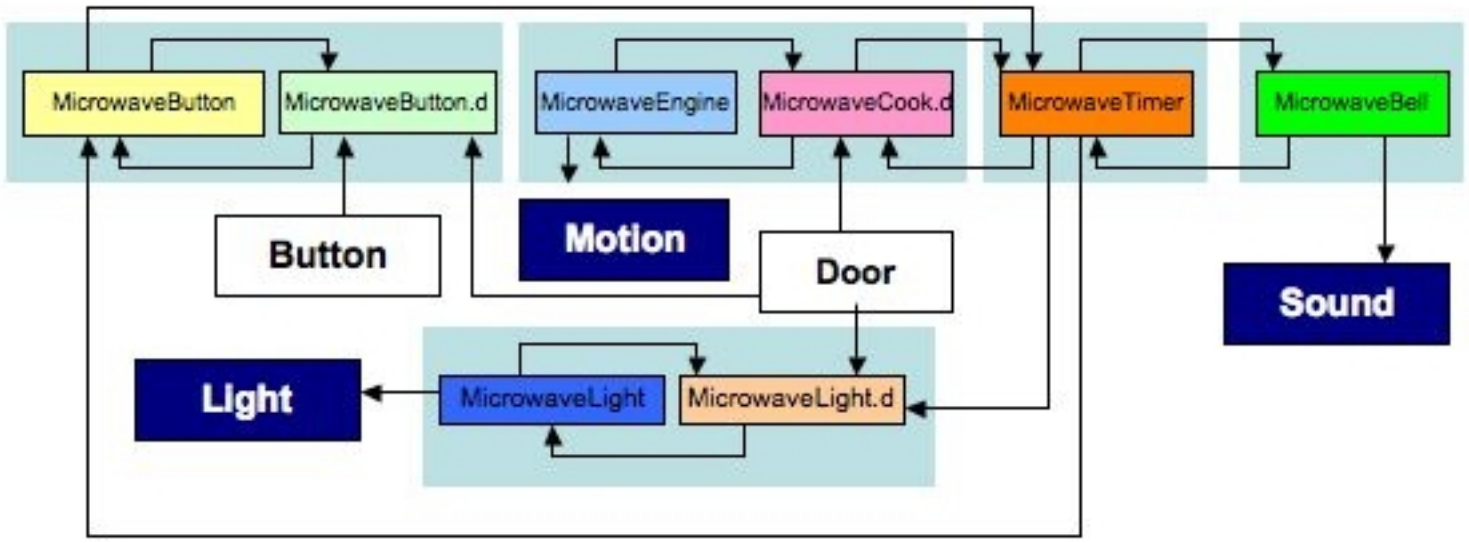
The interaction between modules

- Shows up in the behavior tree.
- But does not happen in BECCIE



Module interaction diagrams

- Perhaps of a global behavior tree



The Software Architecture

For implementation

Software Architecture

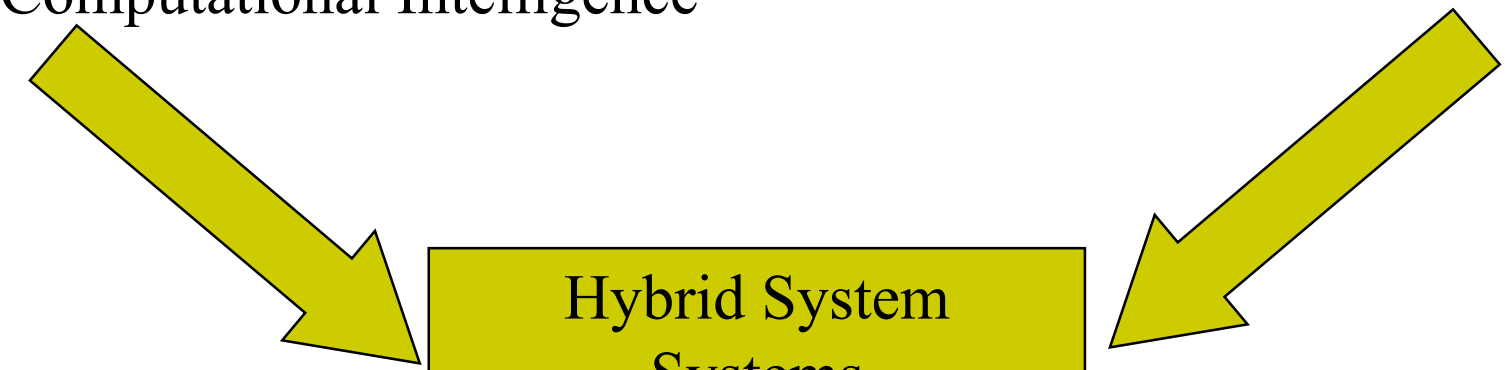
- Agents / Robots

Reactive Systems

Reasoning/ Planning Systems

"Soft-Computing/ Computational Intelligence"

Symbolic AI



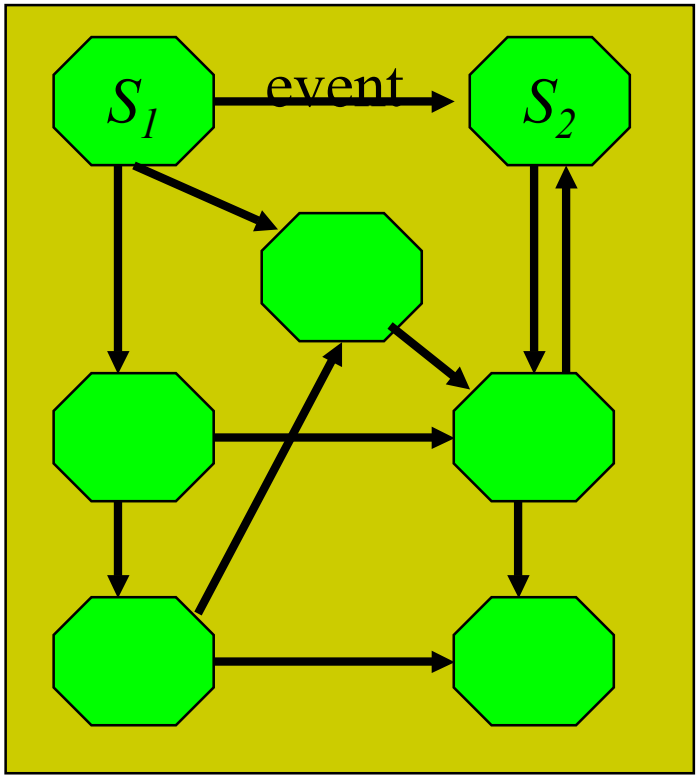
How to integrate?

A hybrid system

- The initial progress on logic and reasoning within AI has largely been discarded from mobile robotics in favour of reactive architectures
- We demonstrate the use of non-monotonic reasoning in the challenging application of RoboCup
- Plausible logic is the only non-monotonic logic with an algorithm that detects loops

Hybrid System for Intelligent and Integrated System

- Reactive System
  - State Machine
- Reasoning
  - Non-Monotonic Logic




```
1.  name(Node).
2.  type State_Type(S_0,..,S_k).
3.  ∨{State(S_0),…,State(S_k)}.
4.  ∨{¬State(S_i), ¬State(S_j)}.
    (∀ i ≠ j)
5.  input{"e_i"}. (for i=1,…,k}
6.  Default: ⇒ State(S_0).
7.  Switch_S_0_S_i:{"e_i"} ⇒
    State(S_i). (for i=1,…,k)
8.  Switch_S_0_S_i > Default.
```

Reasoning

- Deriving conclusions from facts
  - Apparently, a fundamental characteristic of intelligence
- An expected aspect of intelligent systems
- Withdrawing conclusions in the light of new evidence is a capability usually referred to as non-monotonic reasoning

Non-Monotonic Reasoning

- **A form of Common Sense**
- **Retract previous conclusions in the light of new evidence**

1. Planes usually leave on time.
2. My flight leaves at 11:00 am.
3. Therefore, I should be at the airport at 9:00am.
4. My flight is cancelled.
5. Makes no sense to take actions for going to the airport early.

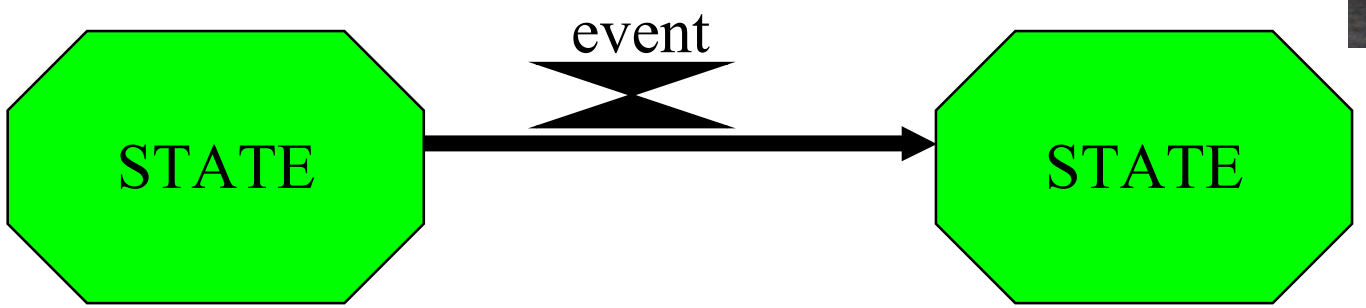
Result: Robotic Poker Player

- Integrate
  - Vision
  - Sound recognition
  - Motion Control

- Environment
  - Complex
  - Interactive
  - Unpredictable
  - Competitive
  - Incomplete Information

Behaviour Design

- Software Engineering
  - visual models of behaviour



statement from non-monotonic logic

- Behaviour Specification
  - by humans
- Human-Robot Interaction

→ Human-Robot Collaboration

Previous Work

--- Software architectures for robotics

- Action - Sensor Model [Wooldridge 2002]
  - Solution for control problem
- Golog [Vassos et al 2007]
  - Aim for “Cognitive Robotics”
- Knowledge Middleware [Heintz et al 2007]
  - Bridge low level sensor knowledge
- Robotic Architectures [Liu 2004]
  - Generic Robot [Kim et al 2005]
    - Solution to platform dependence

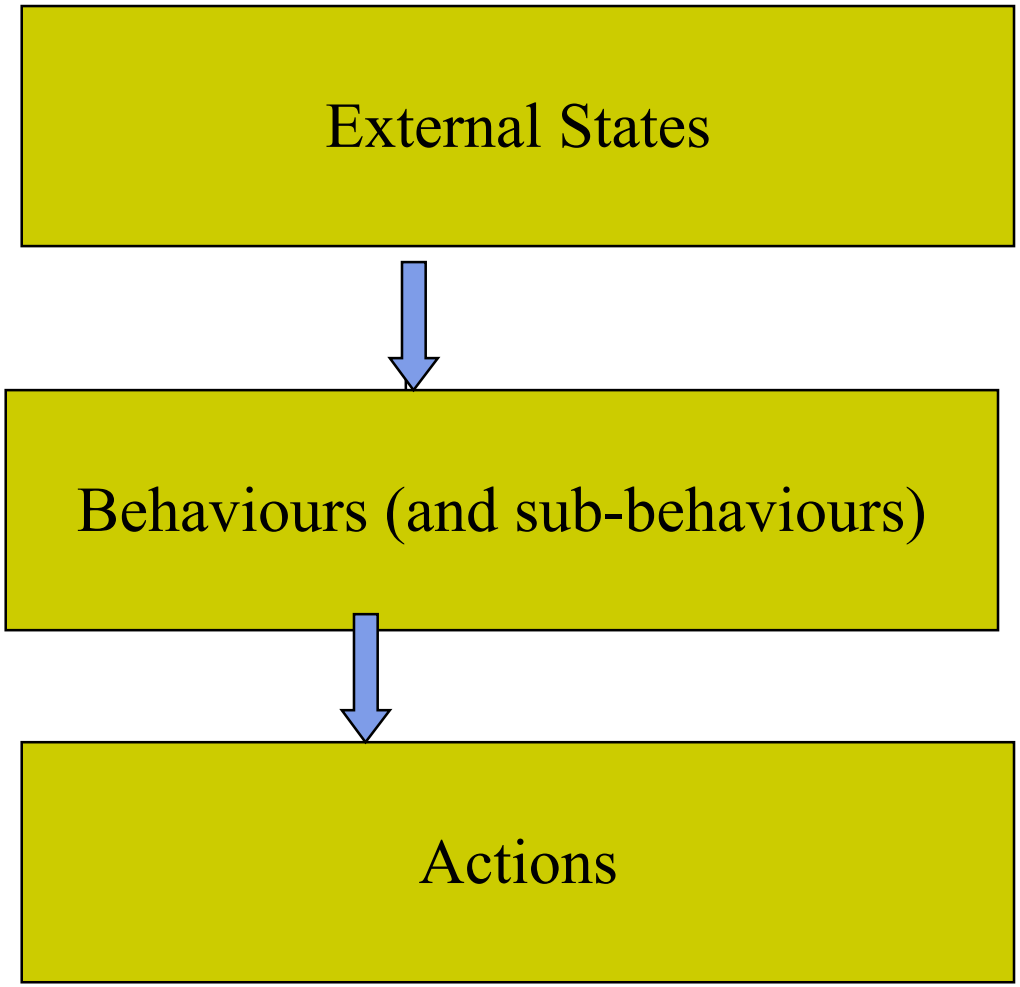
Global Architecture

- Framework = Software Engineering
  - Solves
    - Module Production / Workload problems
    - Software Development Methodology Problem
- Whiteboard (Blackboard [Hayes-Roth 1988])
  - Solves
    - Knowledge representation problem
      - (facts with timestamp and author)
    - Module Interaction Problem
- Domain Knowledge
  - Logics
    - Belief revision / knowledge elicitation
  - Solves
    - Validation / verification /specification

Our Architecture

- Solution to Control Problem

exclusive

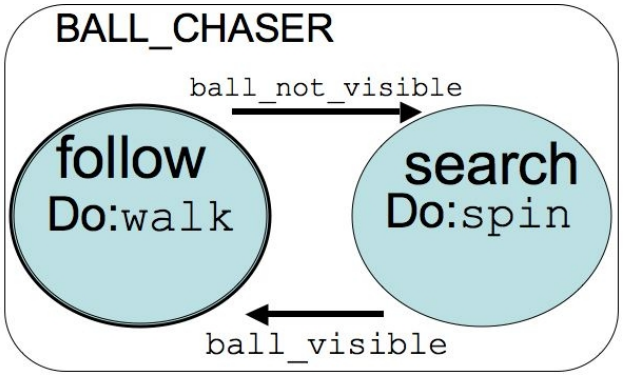


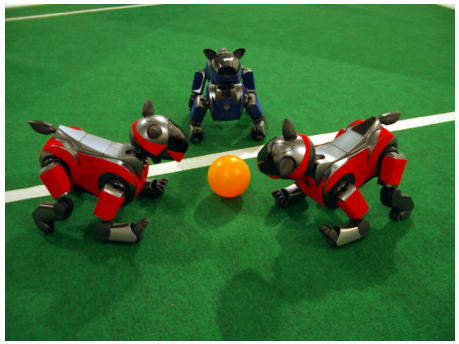
decomposable

priorities
asynchronous
associated with
actuators

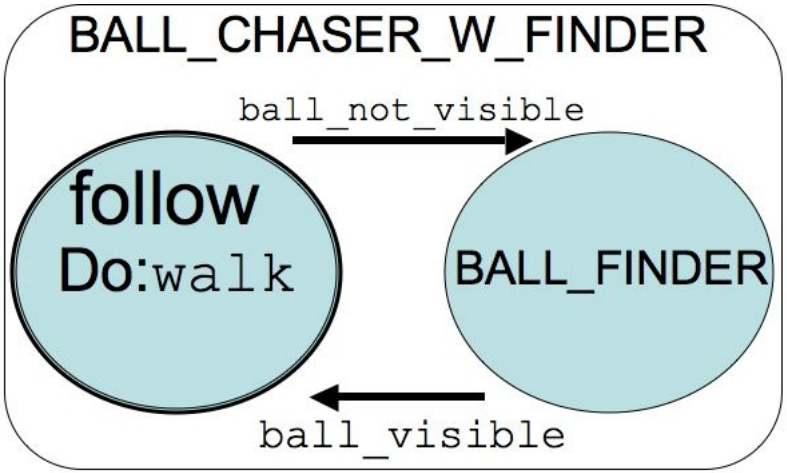
Behaviour Illustration

- Robotic Soccer
  - Simple Behaviour

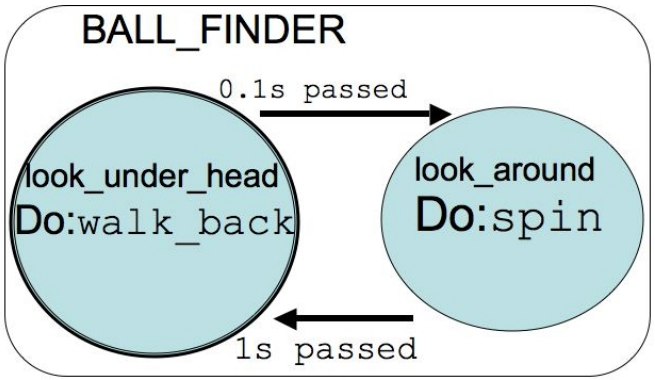




- Robotic Soccer
  - Complex behaviour

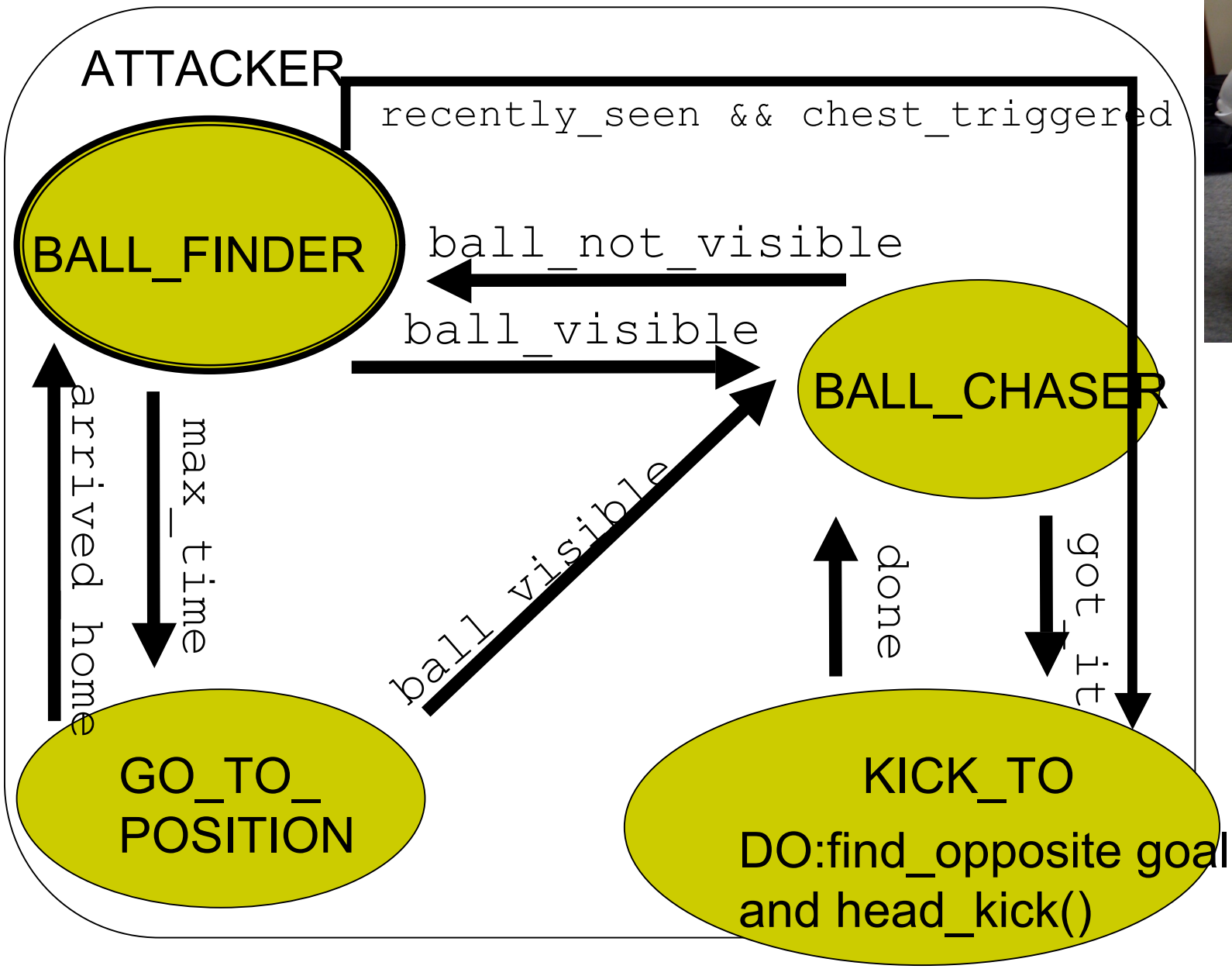


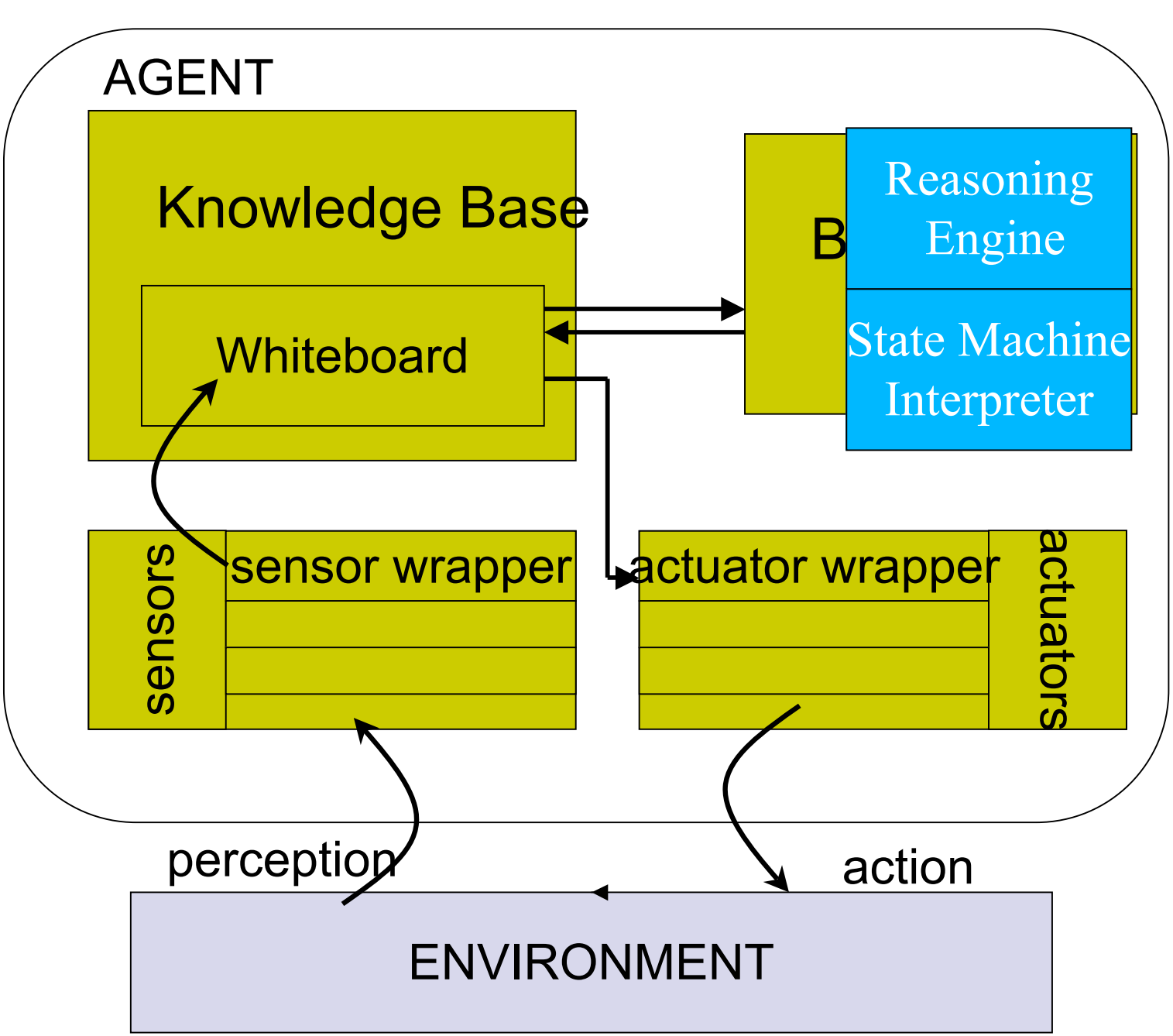
  - Sub-behavior



Engineering the behavior

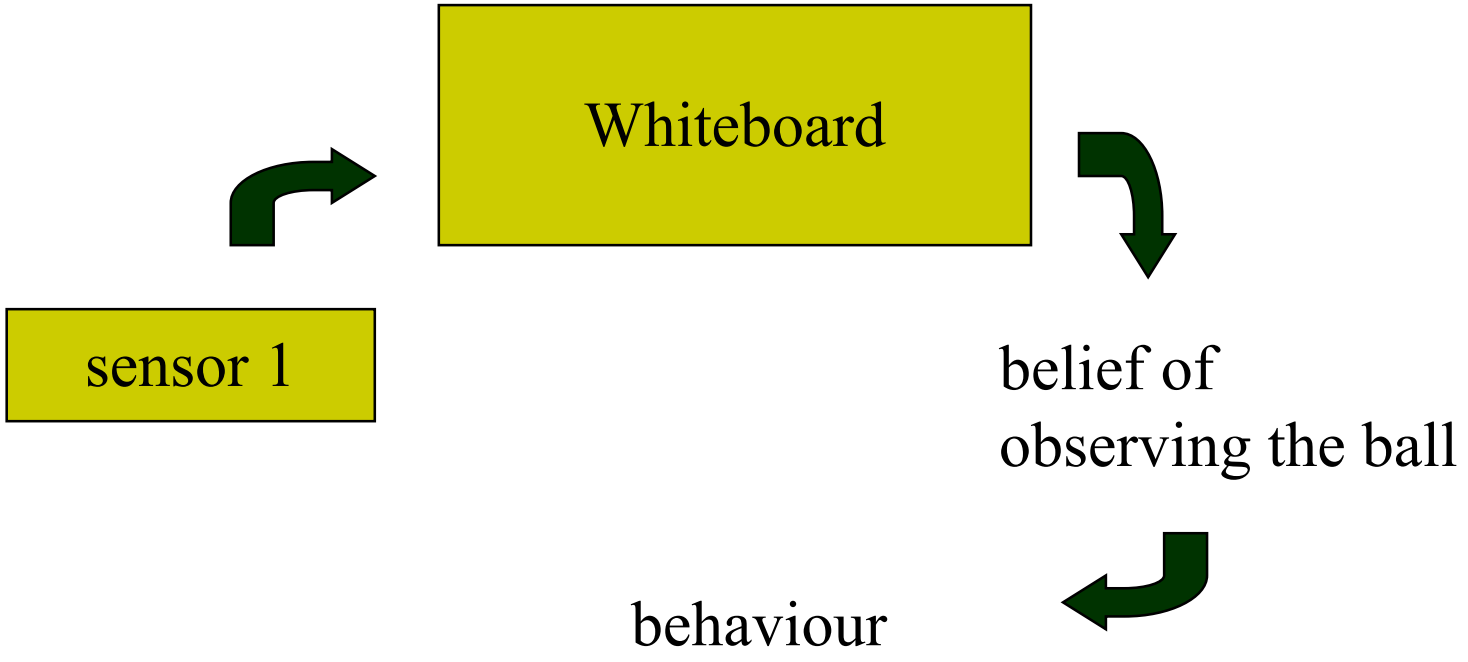
- Using visual descriptions of the behaviour that incorporate formal logic
- Engineers use diagrams to model artefacts.
- Software Engineering has traditionally used diagrams to convey characteristics and descriptions of software


ATTACKER
recently_seen && chest_triggered
BALL_FINDER
ball_not_visible
ball_visible
BALL_CHASER
arrived home
max_time
ball visible
done
got it
GO_TO_
POSITION
KICK_TO
DO:find_opposite goal
and head_kick()


AGENT
Knowledge Base
Whiteboard
B
Reasoning Engine
State Machine Interpreter
sensors
sensor wrapper
actuator wrapper
actuators
perception
action
ENVIRONMENT

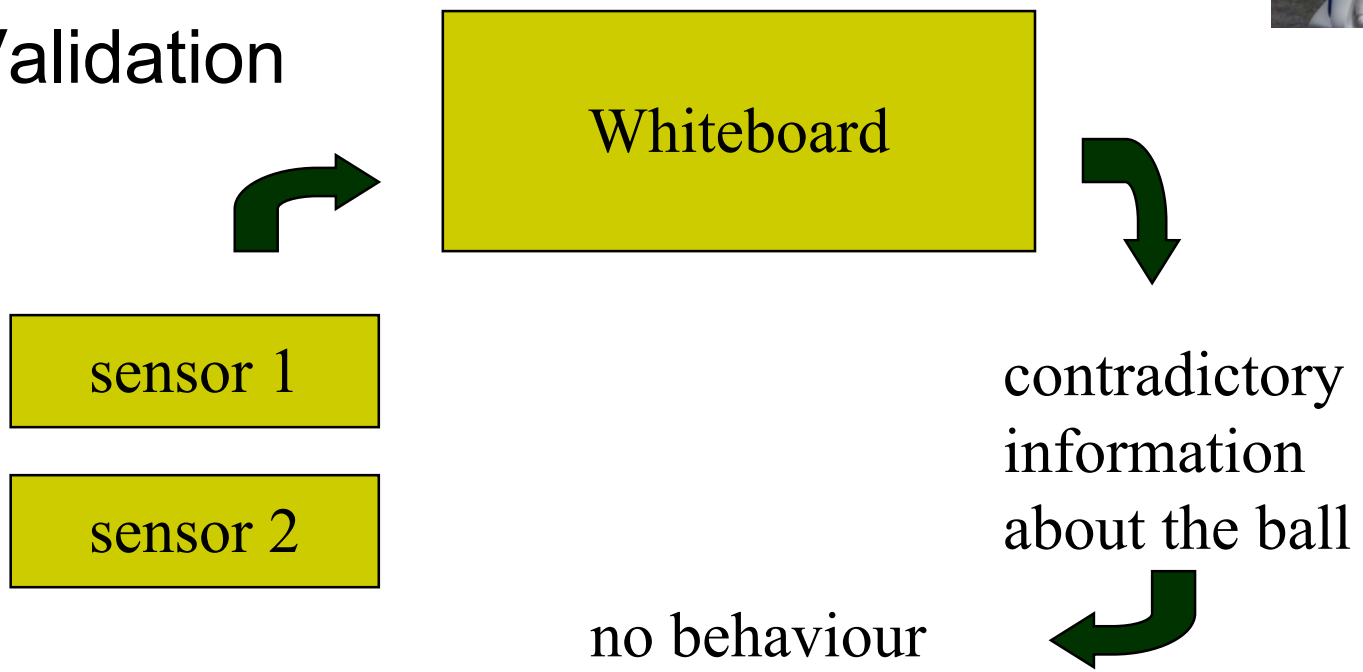
Wrapping Sensors and Actuators

- Portability
- Simulation / Virtualisation
- Validation



Wrapping Sensors and Actuators

- Portability
- Simulation / Virtualisation
- Validation



Our approach

Vision and Object Recognition

Non-monotonic reasoning

Consistency Module

Sensor fusion

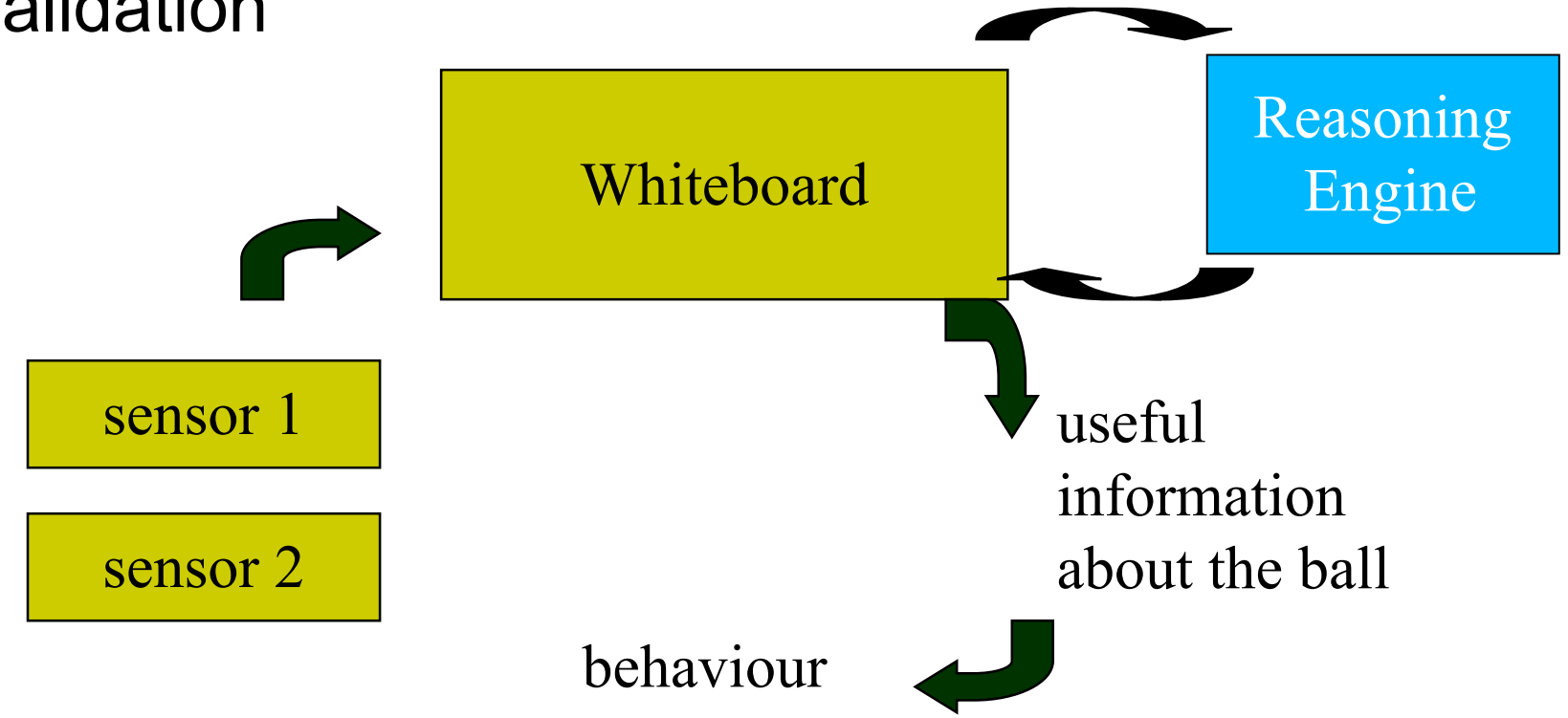
Our approach

Consistency
Module

Non-monotonic logic that combines facts known about the environment with what is reported by the sensors

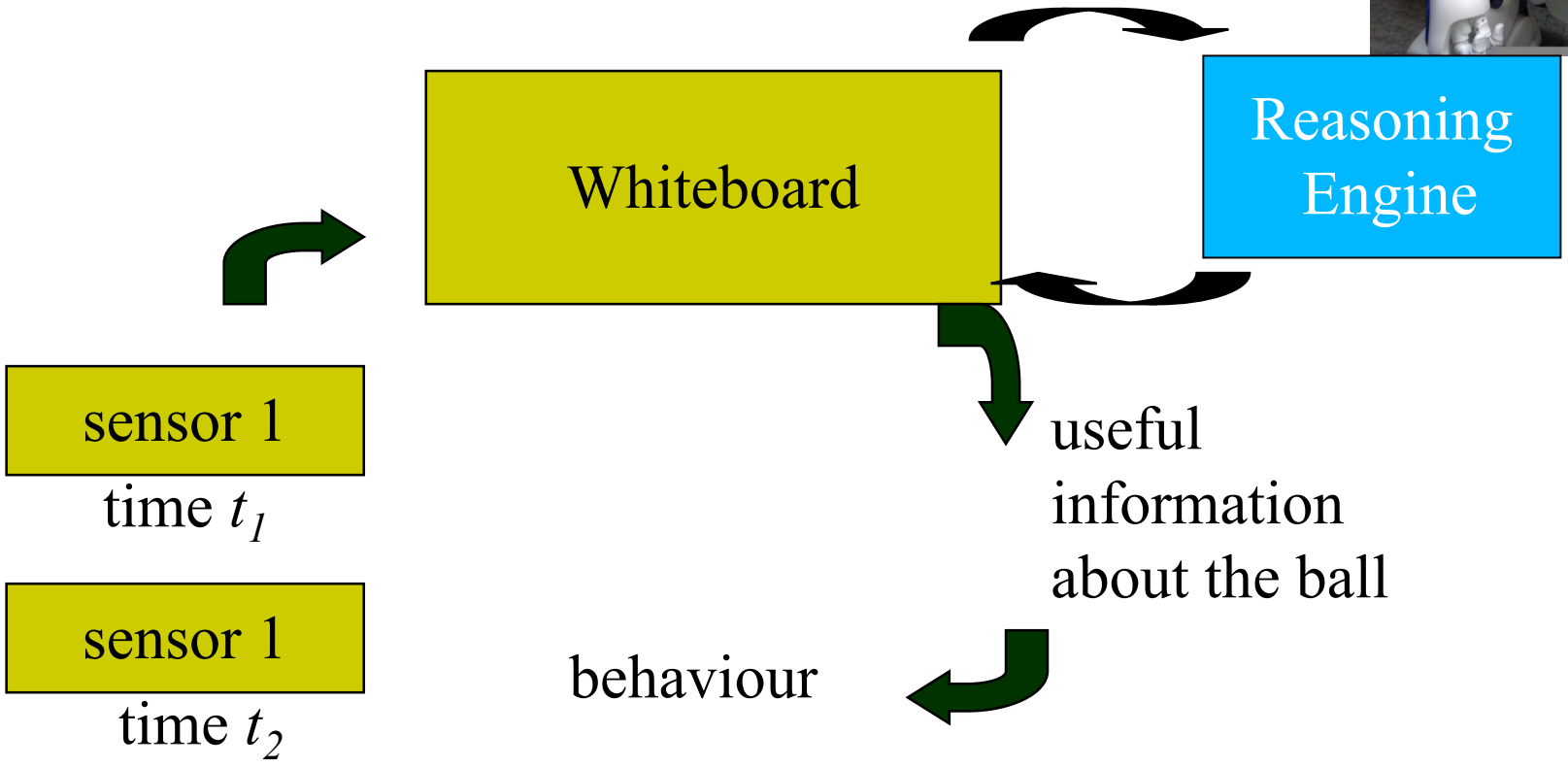
Wrapping Sensors and Actuators

- Portability
- Simulation / Virtualisation
- Validation



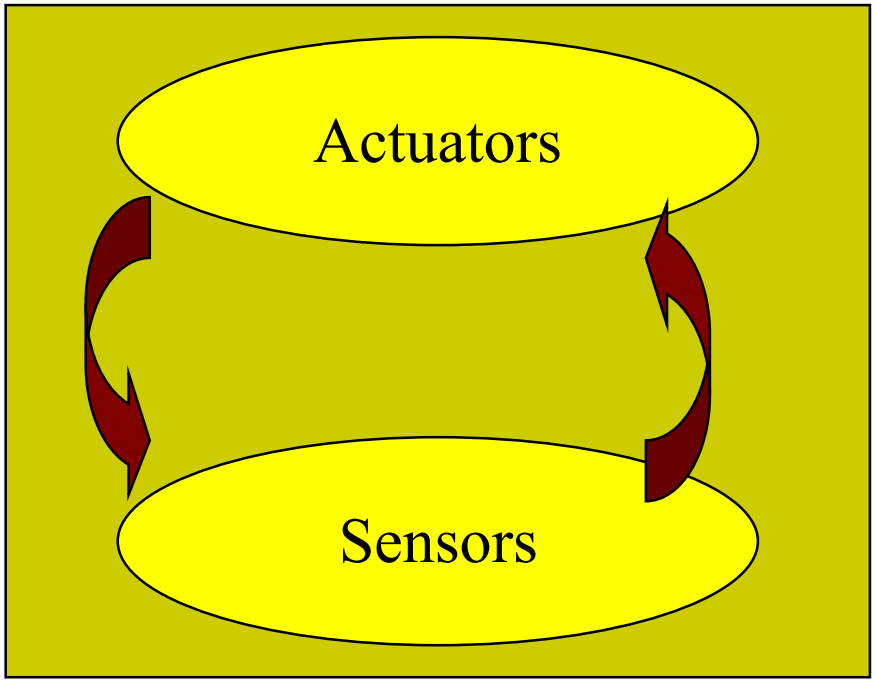
Wrapping Sensors and Actuators

- Fusion in time

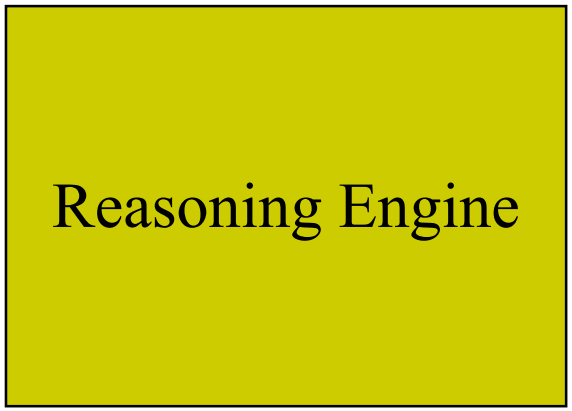


Independent and Asynchronous

- Reasoning Engine



Control

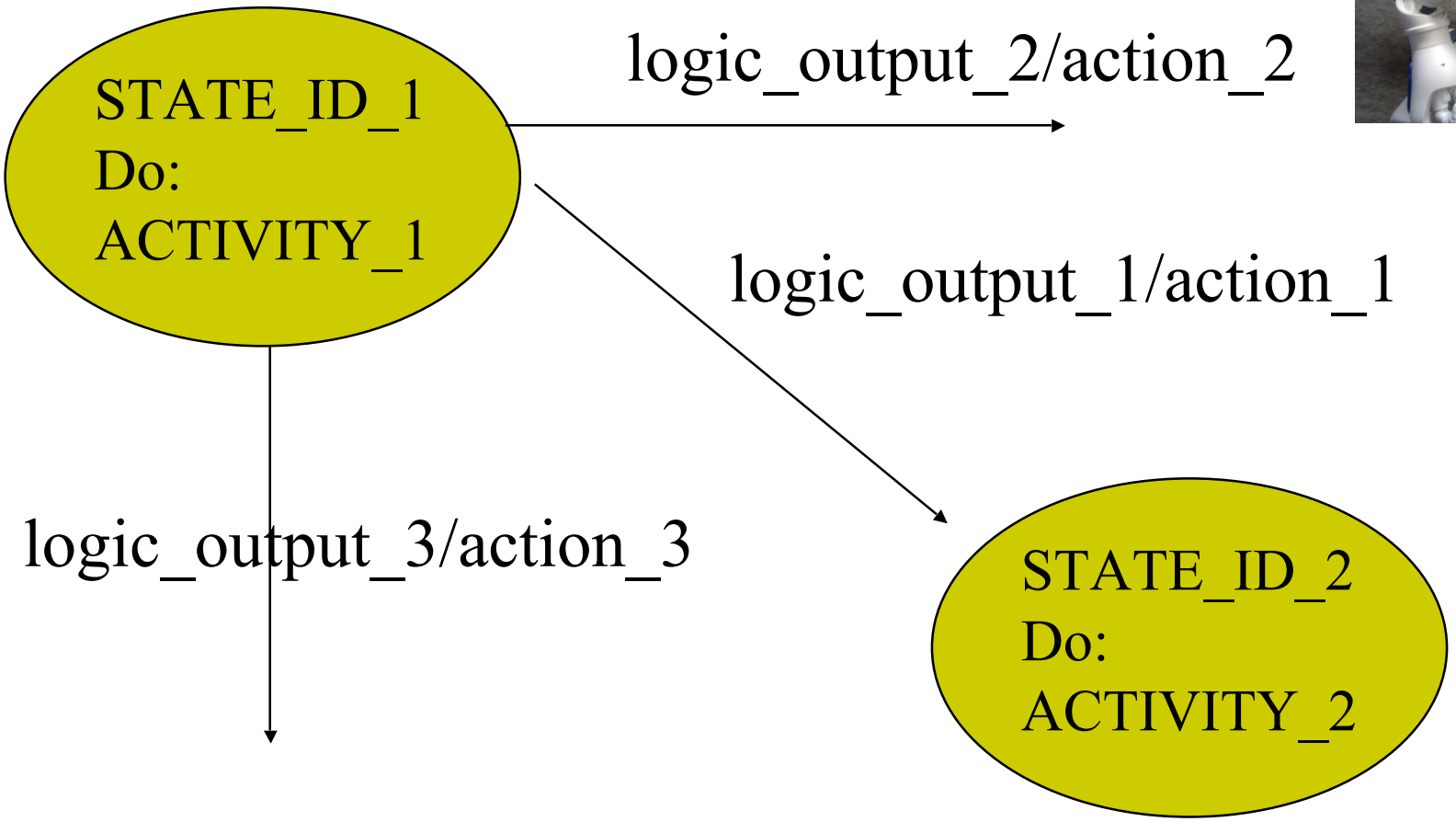


Reasoning Engine

- Template Method

1. All facts are labelled unknown
2. Extract facts from whiteboard
3. Execute predicates that are more efficient in imperative languages
4. Run the necessary queries /proofs on DPL

Interpret a behavior



Behavior Interpreter (version 1)

```
void fsmMachine :: execute ()
{
    vector <fsmState*>::iterator it;
    it=theStates.begin();
    fsmState* current = (*it);
    int currentID = current -> getID();
    cerr << Initial State is State Number " << current->getID()<< "\n";

    while (1) // run for ever
    {// Evaluate labels of transitions going out of current state
     // and may change state
    p_fsmTransition p_itTransitions;
    p_itTransitions = current->theFirstTransition();
    bool transitionFired = false;
     while ((!transitionFired) &&  (NULL!= p_itTransitions))
              {cout << "Evaluate : " <<( (p_itTransitions)->getExpression() ) -> getWhatToEvaluate() << "\n";
               cout << "Does this expression evaluate to true (Y/N)?\n";
               char response;     cin >> response;
               if ('Y'== response) // we need to execute the transition
                {            current= p_itTransitions->getTarget();
                             currentID=current->getID();
                             // break out
                             transitionFired = true;
                 }
                else // advance to next transition
                  {  p_itTransitions = current ->theNextTransition();
                  }
             } // or != NULL
    // send message to Actuators of My Activity
    // by posting to whitebaord
    cout  << " After evalaution the state is : " <<find(current->getID())->getID() << "\n";
    cout << " We are " << ( current->getActivity() )->getWhatToDo() << "\n";
    }
}
```

Get initial state

Always

Get first transition

Evaluate

Move to new state and break if true

Do activity

Summary

- I hope to collaborate with your expertise
- A focused project
  - Humanoid that interactively plays team games of incomplete information with humans
- Enables research on intelligent/smart devices
- We can postulate the use of intelligent capabilities to enhance the life of humans
  - care / assistance / education / tele-presence
- Keep in mind it is more important to improve the condition of human living than to imitate it.

THANK YOU