Multiagent Systems I

Prof. Dr. Jürgen Dix

Department of Informatics
Clausthal University of Technology
WS 08/09

Time: Tuesday, Wednesday 10–12
Place: Am Regenbogen, IfI (Lab)
Labs: From 19. November on.

Website

http://www.in.tu-clausthal.de/abteilungen/cig/cigroot/

Visit regularly!

There you will find important information about the lecture, documents, excercises et cetera.

Lecture: Prof. Dix
Labs: T. Behrens, M. Köster
Exam: Oral exams on demand.

About this lecture

This course gives a first introduction to multi-agent systems for Bachelor students. Emphasis is put on applications and programming MAS, not on theory. After some general introduction of agent systems, we consider one programming language together with a platform for developing agents: 2APL. Students are grouped into teams and implement agent teams for solving a task on our agent contest platform. These teams fight against each other. The winning team will be determined in a competition and get a price.

My thanks go to Tristan Behrens, Michael Köster and our students who prepared the lab work and also some of the slides of this course. In addition, Mehdi Dastani provided me with some slides.

[Bordini et al. 2005] Bordini, R., Dastani, M., Dix, J., and El Fallah Segrouchni, A. (2005).
Programming Multi Agent Systems: Languages, Platforms and Applications.
Springer.

[Wooldridge 2002] Wooldridge, M. (2002).
An Introduction to Multi Agent Systems.
John Wiley & Sons.

Lecture Overview

1. Week: **Chapter 1, Introduction**
2. Week: **Chapter 2, Basic Notions**
3. Week: **Chapter 3, Scenarios + Chapter 4, 2APL**
4. Week: **Chapter 4, 2APL**

4.-14. Week: **Labs.**

Chapter 1. Introduction

Content of this chapter:

We are setting the stage for a precise discussion of agency. From **informal concepts** to (more or less) mathematical definitions.

1. MAS versus **Distributed AI (DAI)**,
2. Environment of agents,
3. Agents and other frameworks,
4. Runs as characteristic behaviour,
5. state-based versus standard agents.

1.1 Why Agents?

Three Important Questions

(Q1) What is a (software) agent?

(Q2) If some program P is not an agent, how can it be transformed into an agent?

(Q3) If (Q1) is clear, what kind of Software Infrastructure is needed for the interaction of agents? What services are necessary?

Definition 1.1 (Distributed Artificial Intelligence (DAI))

The area investigating systems, where several autonomous acting entities work together to reach a given goal.

The entities are called Agents, the area Multiagent Systems.

AAMAS: several conferences joined in 2002 to form the main annual event. Bologna (2002), Melbourne (2003), New York (2004), Utrecht (2005), Hakodate (2006), Hawaii (2007), Lisbon (2008), Budapest (2009).

Example 1.2 (RoboCup)

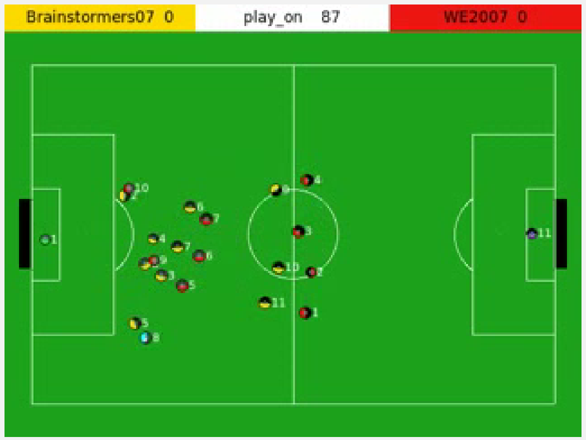


Figure 1: 2D-Simulation league: RoboCup 2007 Final

Example 1.3 (RoboCup)

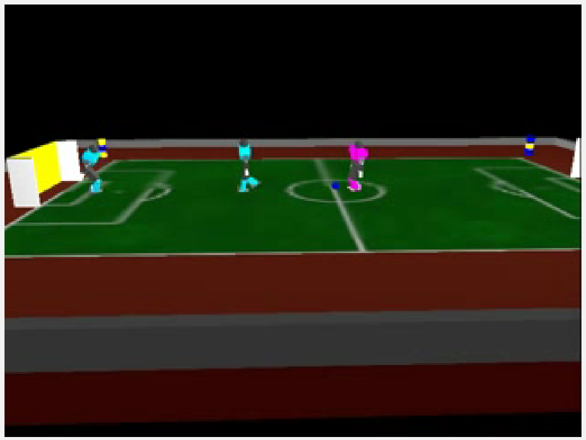


Figure 2: 3D-Simulation league: RoboCup 2007 Final

Example 1.4 (RoboCup)

Figure 3: Small size league

Example 1.5 (RoboCup)

Figure 4: Middle size league

Example 1.6 (RoboCup)

Figure 5: Standard platform

Example 1.7 (RoboCup)

Figure 6: Humanoid league

Example 1.8 (RoboCup)

Figure 7: Rescue league

Example 1.9 (Grand Challenge 2004 (1))

Grand Challenge: Organised by DARPA since 2004.
First try: **Huge Failure**.

Figure 8: Grand Challenge 2004

Example 1.10 (Grand Challenge 2004 (2))

- Prize money: 1 million Dollars
- Race course: 241 km in the Mojave desert
- 10 hours pure driving time
- More than 100 registered participants, 15 of them were chosen
- No one reached the end of the course
- The favourite “Sandstorm” of Carnegie Mellon in Pittsburgh managed 5% of the distance

Example 1.11 (Grand Challenge 2005 (1))

Second try: **Big Success**:
Stanley (Sebastian Thrun) won in 2005.

Figure 9: VW Touareg coached by Stanford University

Example 1.12 (Grand Challenge 2005 (2))

- Prize money: 2 million Dollars
- Race course: 212,76 km in the Mojave desert
- 10 hours pure driving time
- 195 registered participants, 23 were qualified
- 5 teams reached the end of the course (4 teams in time)
- Stanley finished the race in 6 hours and 53 minutes (30,7 km/h)
- Sandstorm achieved the second place

Example 1.13 (Urban Challenge (1))

Urban Challenge: Organised by DARPA since 2007.

Figure 10: Urban Challenge 2007

Example 1.14 (Urban Challenge (2))

- No straight-line course but real streets covered with buildings.
- 60 miles
- Prize money: 3,5 million Dollars
- Tartan Racing won, Stanford Racing Team second, VictorTango third place.
- Some teams like Stanford Racing Team and VictorTango as well as Tartan Racing were sponsored by DARPA with 1 million Dollar beforehand.

Example 1.15 (CLIMA Contest: Gold Mining (1))

First try: A simple grid where agents are supposed to collect gold. Different roles of agents: scouts, collectors.

- Old site: `http://cig.in.tu-clausthal.de/agentcontest2008`
- New site: `http://multiagentcontest.org`

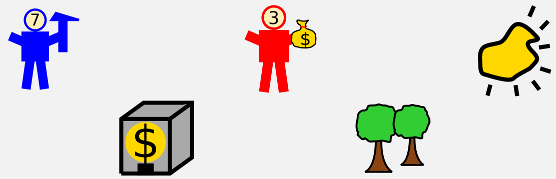


Figure 11: Gold mining elements

Example 1.16 (CLIMA Contest: Gold Mining (2))

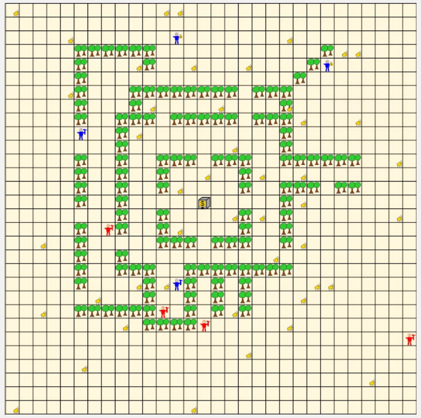


Figure 12: Gold Mining 2006: CLIMABot (blue) vs. brazil (red)

Example 1.17 (Agent Contest: Chasing Cows (1))

Second try: Push cows in a corral.

- Old site: `http://cig.in.tu-clausthal.de/agentcontest2008`
- New site: `http://multiagentcontest.org`

Example 1.18 (Agent Contest: Chasing Cows (2))

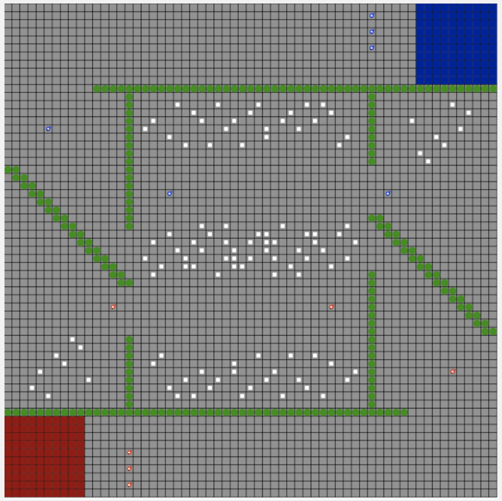


Figure 13: Chasing Cows 2008

Why do we need them?

Information systems are **distributed**, **open**, **heterogeneous**.
We therefore need intelligent, interactive agents, that act autonomously.

(Software) Agent: Programs that are implemented on a platform and have **sensors** and **effectors** to read from and make changes to the environment, respectively.

Intelligent: Performance measures, to evaluate the success. **Rational** vs. **omniscient**, **decision making**

Interactive: with other agents (software or humans) by observing the environment.

Coordination: **Cooperation** vs. **Competition**

MAS versus Classical DAI

MAS: Several Agents coordinate their knowledge and actions (semantics describes this).

DAI: Particular problem is divided into smaller problems (nodes). These nodes have common knowledge. **The solution method is given.**

Attention:

Today DAI is used synonymously with MAS.

AI	DAI
Agent	**Multiple** Agents
Intelligence: Property of a **single** Agent	Intelligence: Property of several Agents
Cognitive Processes of a **single** Agent	Social Processes of several Agents

10 Desiderata

1. Agents are for everyone! We need a method to agentise given programs.
2. Take into account that data is stored in a wide variety of data structures, and data is manipulated by an existing corpus of algorithms.
3. A theory of agents must *not* depend upon the set of actions that the agent performs. Rather, the set of actions that the agent performs must be a *parameter* that is taken into account in the semantics.

10 Desiderata

4. Every (software) agent should execute actions based on some *clearly articulated* decision policy. A declarative framework for articulating decision policies of agents is imperative.
5. Any agent construction framework must allow agents to reason:
   - **Reasoning about its beliefs** about other agents.
   - **Reasoning about uncertainty** in its beliefs about the world and about its beliefs about other agents.
   - **Reasoning about time**.

   These capabilities should be viewed as *extensions* to a core agent action language.

1. Introduction 1. Why Agents?

10 Desiderata

6. Any infrastructure to support multiagent interactions *must* provide security.
7. While the efficiency of the code underlying a software agent cannot be guaranteed (as it will vary from one application to another), guarantees are needed that provide information on the performance of an agent relative to an oracle that supports calls to underlying software code.

1. Introduction 1. Why Agents?

10 Desiderata

8. We must identify efficiently computable fragments of the general hierarchy of languages alluded to above, and our implementations must take advantage of the specific structure of such language fragments.
9. A critical point is reliability—there is no point in a highly efficient implementation, if all agents deployed in the implementation come to a grinding halt when the agent “infrastructure” crashes.

1. Introduction 1. Why Agents?

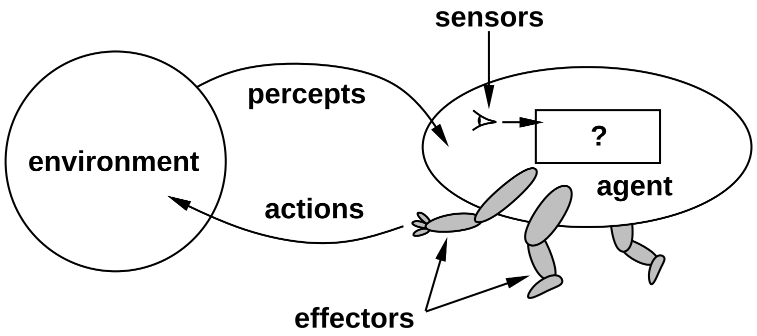
10 Desiderata

10. The only way of testing the applicability of any theory is to build a software system based on the theory, to deploy a set of applications based on the theory, and to report on experiments based on those applications.

1.2 Intelligent Agents

Definition 1.19 (Agent a)

An agent a is anything that can be viewed as **perceiving** its environment through **sensor** and **acting** upon that environment through **effectors**.



Definition 1.20 (Rational Agent, Omniscient Agent)

A rational agent is one that does the right thing (**Performance measure** determines how successful an agent is).

A omniscient agent knows the actual outcome of his actions and can act accordingly.

Attention:

A rational agent is in general not omniscient!

Question

What is the right thing and what does it depend on?

1. Performance measure (as objective as possible).
2. Percept sequence (everything the agent has received so far).
3. The agent's knowledge about the environment.
4. How the agent can act.

Definition 1.21 (Ideal Rational Agent)

For each possible percept-sequence an ideal rational agent should do whatever action is expected to maximize its performance measure (based on the evidence provided by the percepts and built-in knowledge).

Mappings:

set of percept sequences $\mapsto$ set of actions

can be used to describe agents in a mathematical way.

Hint:

Internally an agent is

agent = architecture + program

AI is engaged in designing agent programs

Agent Type	Perform. Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs	Patient, hospital, staff	Display questions, tests, diagnoses, treatments	Entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Table 1: Examples of agents types and their **PEAS** descriptions.

Question:

How do environment properties influence agent design?

Definition 1.22 (Properties of the Environment)

Accessible/Inaccessible: If not completely accessible, one needs internal states.

Deterministic/Indeterministic: An inaccessible environment might seem indeterministic, even if it is not.

Episodic/Nonepisodic: Percept-Action-Sequences are independent from each other. Closed episodes.

Static/Dynamic: While the agent is thinking, the world is the same/changing. **Semi-dynamic**: The world does not change, but the performance measure.

Discrete/Continous: Density of observations and actions. Relevant: Level of granularity.

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

xbiff, software demons are agents (not intelligent).

Definition 1.23 (Intelligent Agent)

An intelligent agent is an agent with the following properties:

1. Autonomous: Operates without direct intervention of others, has some kind of control over its actions and internal state.
2. Reactive: Reaction to changes in the environment at certain times to reach its goals.
3. Pro-active: Taking the initiative, being goal-directed.
4. Social: Interaction with others to reach the goals.

Pro-active alone is not sufficient
(C-Programs): The environment can change during execution.

Socialisation: coordination, communication, (negotiation) skills.

Difficulty: right balance between pro-active and reactive!

Agents vs. Object Orientation

Objects have

1. a **state** (encapsulated): control over internal state
2. message passing capabilities

Java: private and public methods.

- Objects have control over their state, but **not over their behaviour**.
- An object can not prevent others to use its public methods.

Agents call other agents and request them to execute actions.

- Objects do it for free, agents do it for money.
- No analoga to **reactive**, **pro-active**, **social** in OO.
- MAS are multi-threaded or even multi-processed: each agent has a control thread or is a new process. (In OO only the system as a whole possesses one.)

A simple agent program:

function SKELETON-AGENT(*percept*) **returns** action
  static: *memory*, the agent's memory of the world

  memory ← UPDATE-MEMORY(*memory, percept*)
  action ← CHOOSE-BEST-ACTION(*memory*)
  memory ← UPDATE-MEMORY(*memory, action*)
  return *action*

In theory everything is trivial:

function TABLE-DRIVEN-AGENT(*percept*) **returns** *action*
  static: *percepts*, a sequence, initially empty
    table, a table, indexed by percept sequences, initially fully specified

  append *percept* to the end of *percepts*
  action ← LOOKUP(*percepts, table*)
  return *action*

An agent example – taxi driver:

Agent Type	Perform. Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn	Cameras, sonar, GPS odometer, engine sensors

Table 2: **PEAS** description of the task environment for an automated taxi

Some examples:

1. **Production rules:** If the driver in front hits the breaks, then hit the breaks too.

function SIMPLE-REFLEX-AGENT(*percept*) **returns** *action*
  static: *rules*, a set of condition-action rules

  state ← INTERPRET-INPUT(*percept*)
  rule ← RULE-MATCH(*state, rules*)
  action ← RULE-ACTION[*rule*]
  return *action*

Agents as Intentional Systems

Intentions: Agents are endowed with **mental states**.

Matthias took his umbrella because he **believed** it was going to rain.
Kjeld attended the MAS course because he **wanted** to learn about agents.

An intentional system describes entities whose behaviour can be predicted by the method of attributing beliefs, desires and rational acumen.

1.3 Formal Description

A first mathematical description

At first, we want to keep everything as simple as possible.

Agents and environments

An agent is situated in an environment and can perform actions

$$A := \{a_1, \dots, a_n\} \qquad \text{(set of actions)}$$

and change the state of the environment

$$S := \{s_1, s_2, \dots, s_n\} \qquad \text{(set of states)}.$$

How does the environment (the state s) develop when an action a is executed?

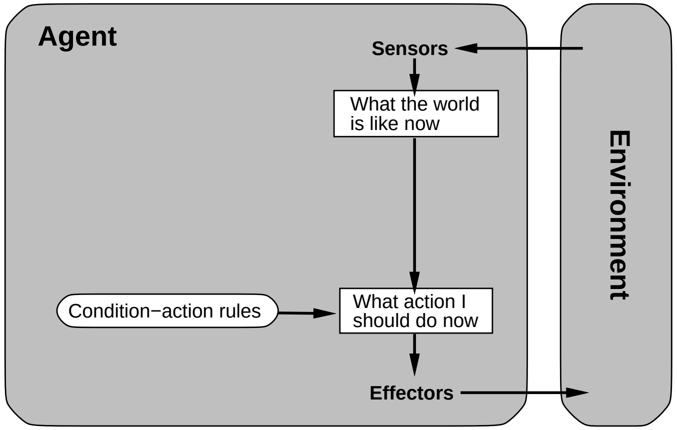
We describe this with a function

$$\mathbf{env} : \mathbf{S} \times \mathbf{A} \longrightarrow 2^{\mathbf{S}}.$$

This includes non-deterministic environments.

How do we describe agents?

We could take a function $\mathbf{action} : \mathbf{S} \longrightarrow \mathbf{A}.$



Question:

How can we describe an agent, now?

Definition 1.24 (Purely Reactive Agent)

An agent is called purely reactive, if its function is given by

$$\mathbf{action} : \mathbf{S} \longrightarrow \mathbf{A}.$$

This is too weak!

Take the whole history (of the environment) into account: $\mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots \mathbf{s_n} \rightarrow_{\mathbf{a_n}} \ldots.$

The same should be done for $\mathbf{env}$!

This leads to agents that take the **whole sequence of states** into account, i.e.

$$\text{action} : \mathbf{S}^* \longrightarrow \mathbf{A}.$$

We also want to consider the actions **performed by an agent**. This requires the notion of a run (next slide).

We define the run of an agent in an environment as a sequence of interleaved states and actions:

Definition 1.25 (Run r, $R = R^{act} \cup R^{state}$)

A run r over $\mathbf{A}$ and $\mathbf{S}$ is a finite sequence

$$\mathbf{r} : \mathbf{s}_0 \longrightarrow_{\mathbf{a}_0} \mathbf{s}_1 \longrightarrow_{\mathbf{a}_1} \ldots \mathbf{s}_n \longrightarrow_{\mathbf{a}_n} \ldots$$

Such a sequence may end with a state $\mathbf{s}_n$ or with an action $\mathbf{a}_n$: we denote by R^{act} the set of **runs ending with an action** and by R^{state} the set of **runs ending with a state**.

Definition 1.26 (Environment, 2. version)

An environment Env is a triple $\langle \mathbf{S}, \mathbf{s}_0, \boldsymbol{\tau} \rangle$ consisting of

1. the set $\mathbf{S}$ of states,
2. the initial state $\mathbf{s}_0 \in \mathbf{S}$,
3. a function $\boldsymbol{\tau} : R^{act} \longrightarrow 2^{\mathbf{S}}$, which describes how the environment changes when an action is performed (given the whole history).

Definition 1.27 (Agent $\mathfrak{a}$)

An agent $\mathfrak{a}$ is determined by a function

$$\text{action} : R^{state} \longrightarrow \mathbf{A},$$

describing which action the agent performs, given its current history.

Important:

An agent system is then a pair $\mathfrak{a} = \langle \text{action}, Env \rangle$ consisting of an agent and an environment.
We denote by $R(\mathfrak{a}, Env)$ the set of runs of agent $\mathfrak{a}$ in environment Env.

Definition 1.28 (Characteristic Behaviour)

The characteristic behaviour of an agent $\mathfrak{a}$ in an environment Env is the set R of all possible runs $\mathbf{r} : \mathbf{s_0} \rightarrow_{a_0} \mathbf{s_1} \rightarrow_{a_1} \ldots \mathbf{s_n} \rightarrow_{a_n} \ldots$ with:

1. for all n: $\mathbf{a_n} = \mathbf{action}(\langle \mathbf{s_0}, \mathbf{a_0} \ldots, \mathbf{a_{n-1}}, \mathbf{s_n} \rangle)$,
2. for all $n > 0$:
   $\mathbf{s_n} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{n-1}}, a_{n-1})$.

For deterministic τ, the relation "$\in$" can be replaced by "=".

Important:

The formalization of the characteristic behaviour is dependent of the concrete agent type. Later we will introduce further behaviours (and corresponding agent designs).

Equivalence

Two agents $\mathfrak{a}$, $\mathfrak{b}$ are called **behaviourally equivalent wrt. environment** Env, if $\mathrm{R}(\mathfrak{a}, Env) = \mathrm{R}(\mathfrak{b}, Env)$.
Two agents $\mathfrak{a}$, $\mathfrak{b}$ are called behaviourally equivalent, if they are behaviourally equivalent wrt. all possible environments Env.

So far so good, but...

What is the problem with all these agents and this framework in general?

Problem

All agents have perfect information about the environment!

(Of course, it can also be seen as feature!)

We need more realistic agents!

Note

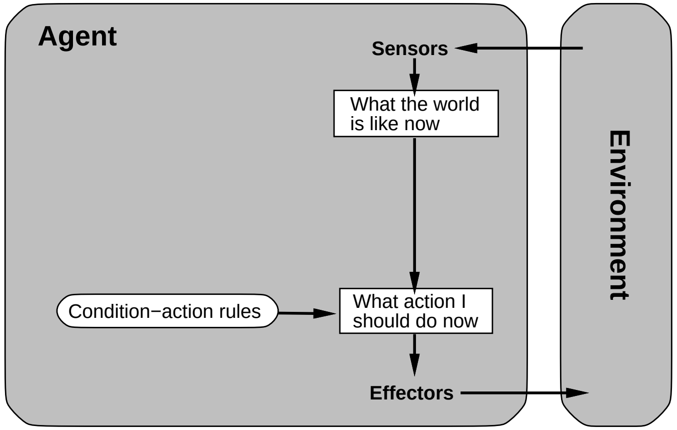
In general, agents only have incomplete/uncertain information about the environment!

We extend our framework by perceptions:

Definition 1.29 (Actions A, Percepts P, States S)

$A := \{a_1, a_2, \ldots, a_n\}$ is the set of *actions*.
$P := \{p_1, p_2, \ldots, p_m\}$ is the set of percepts.
$S := \{s_1, s_2, \ldots, s_l\}$ is the set of *states*

Sensors don't need to provide perfect information!



Question:

How can agent programs be designed?

There are four types of agent programs:

- Simple **reflex agents**
- **Agents that keep track of the world**
- Goal-based agents
- Utility-based agents

First try

We consider a purely reactive agent and just replace states by perceptions.

Definition 1.30 (Simple Reflex Agent)

An agent is called simple reflex agent, if its function is given by

$$\mathrm{action} : P \longrightarrow A.$$

A very simple reflex agent

function SIMPLE-REFLEX-AGENT(*percept*) **returns** *action*
  static: *rules*, a set of condition-action rules

  state ← INTERPRET-INPUT(*percept*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← RULE-ACTION[*rule*]
  return *action*

A simple reflex agent with memory

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** *action*
  static: *state*, a description of the current world state
      rules, a set of condition-action rules

  state ← UPDATE-STATE(*state*, *percept*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← RULE-ACTION[*rule*]
  state ← UPDATE-STATE(*state*, *action*)
  return *action*

As before, let us now consider sequences of percepts:

Definition 1.31 (Standard Agent $\mathfrak{a}$)

A standard agent $\mathfrak{a}$ is given by a function

$$\mathbf{action} : \mathbf{P}^* \longrightarrow A$$

together with

$$\mathbf{see} : \mathbf{S} \longrightarrow \mathbf{P}.$$

An agent is thus a pair $\langle \mathbf{see}, \mathbf{action} \rangle$.

Definition 1.32 (Indistinguishable)

Two different states s, s' are indistinguishable for an agent $\mathfrak{a}$, if $\mathbf{see}(s) = \mathbf{see}(s')$.

The relation “indistinguishable” on $\mathbf{S} \times \mathbf{S}$ is an **equivalence** relation.
What does $|\sim| = |\mathbf{S}|$ mean?
And what $|\sim| = 1$?

As mentioned before, the characteristic behaviour has to match with the agent design!

Definition 1.33 (Characteristic Behaviour)

The characteristic behaviour of a standard agent $\langle \mathbf{see}, \mathbf{action} \rangle$ in an environment Env is the set of all finite sequences

$$\mathbf{p_0} \rightarrow_{a_0} \mathbf{p_1} \rightarrow_{a_1} \cdots \mathbf{p_n} \rightarrow_{a_n} \cdots$$

where

$\mathbf{p_0} = \mathbf{see}(\mathbf{s_0})$,
$\mathbf{a_i} = \mathbf{action}(\langle \mathbf{p_0}, \ldots, \mathbf{p_i} \rangle)$,
$\mathbf{p_i} = \mathbf{see}(\mathbf{s_i})$, where $\mathbf{s_i} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{i-1}}, a_{i-1})$.

Such a sequence, even if deterministic from the agent's viewpoint, may cover different environmental behaviours (runs):
$\mathbf{s_0} \rightarrow_{a_0} \mathbf{s_1} \rightarrow_{a_1} \cdots \mathbf{s_n} \rightarrow_{a_n} \cdots$

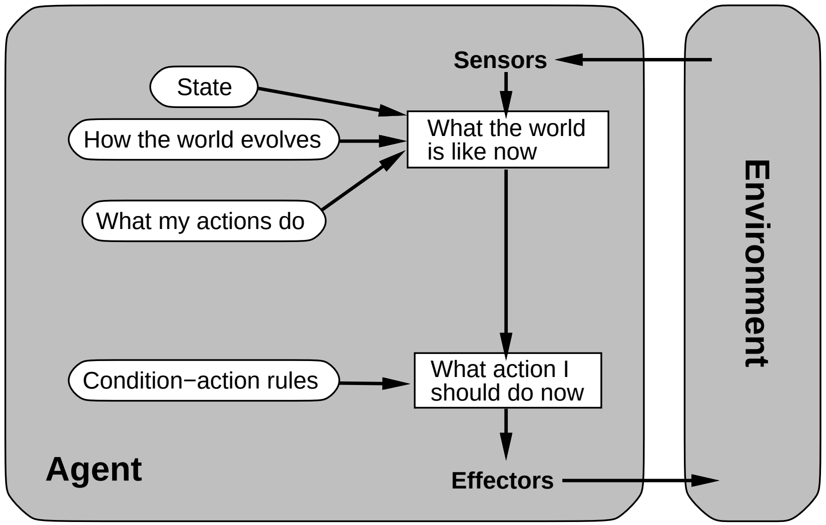
Instead of using the whole history, resp. $\mathbf{P}^*$, one can also use **internal states** $\mathbf{I} := \{\mathbf{i_1}, \mathbf{i_2}, \ldots, \mathbf{i_n}, \mathbf{i_{n+1}}, \ldots\}$.

Definition 1.34 (State-based Agent $\mathfrak{a}_{state}$)

A state-based agent $\mathfrak{a}_{state}$ is given by a function $\mathbf{action} : I \longrightarrow A$ together with

$$\mathbf{see} : \mathbf{S} \longrightarrow \mathbf{P},$$
$$\text{and} \quad \mathbf{next} : \mathbf{I} \times \mathbf{P} \longrightarrow \mathbf{I}.$$

Here $\mathbf{next}(\mathbf{i}, \mathbf{p})$ is the successor state of $\mathbf{i}$ if $\mathbf{p}$ is observed.



Definition 1.35 (Characteristic Behaviour)

The characteristic behaviour of a state-based agent $\mathfrak{a}_{state}$ in an environment Env is the set of all finite sequences

$$(\mathbf{i}_0, \mathbf{p_0}) \rightarrow_{a_0} (\mathbf{i}_1, \mathbf{p_1}) \rightarrow_{a_1} \ldots \rightarrow_{a_{n-1}} (\mathbf{i}_n, \mathbf{p_n}), \ldots$$

with

$\mathbf{p_0} = \mathbf{see}(\mathbf{s_0})$,
$\mathbf{p_i} = \mathbf{see}(\mathbf{s_i})$, where $\mathbf{s_i} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{i-1}}, a_{i-1})$,
$\mathbf{a_n} = \mathbf{action}(\mathbf{i_{n+1}})$,
$\mathbf{next}(\mathbf{i_n}, \mathbf{p_n}) = \mathbf{i_{n+1}}$.

Sequence covers the runs $\mathbf{r} : \mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots$ where

$\mathbf{a_j} = \mathbf{action}(\mathbf{i_{j+1}})$,
$\mathbf{s_j} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{j-1}}, a_{j-1})$,
$\mathbf{p_j} = \mathbf{see}(\mathbf{s_j})$

Are state-based agents more expressive than standard agents? How to measure?

Definition 1.36 (Environmental Behaviour of a_{state})

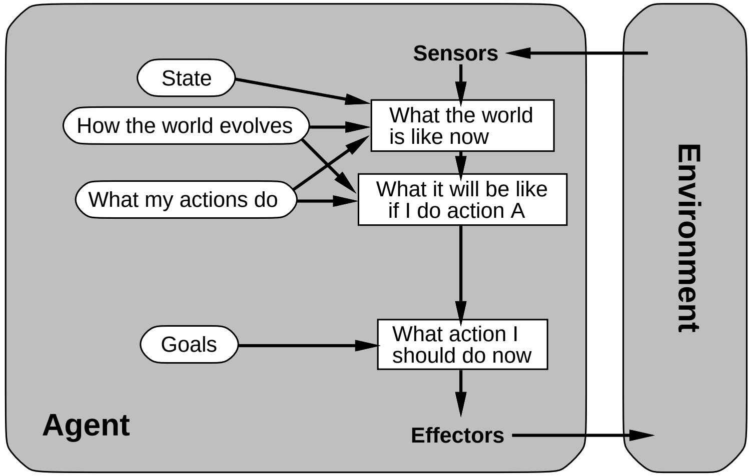
The environmental behaviour of an agent a_{state} is the set of possible runs covered by the characteristic behaviour of the agent.

Theorem 1.37 (Equivalence)

Standard agents and state-based agents are equivalent with respect to their environmental behaviour.

More precisely: For each state-based agent a_{state} and $\mathbf{next}$ storage function there exists a standard agent a which has the same environmental behaviour, and vice versa.

3. **Goal based agents**:



This leads to **Planning**.

Chapter 2. Basic Notions

Content of this chapter:

In this chapter we present some important techniques that will be used later for programming agents.

- An architecture for reactive agents, based on a subsumption.
- The BDI/Agent oriented programming-, architecture. While 2APL is not exactly based on this version of BDI, it is very similar in spirit.
- We introduce some PROLOG technology: terms, facts and rules. These are the basic ingredients for writing agents in the labs.

2.1 Reactive Agents

Idea:

Intelligent behaviour is Interaction of the agents with their environment.

It emerges through splitting in simpler interactions.

Subsumption-Architectures:

- Decision making is realized through **goal-directed behaviours**: each behaviour is an individual action.
  nonsymbolic implementation.
- Many behaviours can be applied **concurrently**. How to select between them?
  Implementation through Subsumption-Hierarchies, Layers.
  Upper layers represent abstract behaviour.

Formal Model:

- **see**: as up to now, but close relation between observation and action: **no transformation of the input**.
- **action**: Set of behaviors and inhibition relation.

$$Beh := \{\langle \mathbf{c}, \mathbf{a} \rangle : \mathbf{c} \subseteq \mathbf{P}, \mathbf{a} \in \mathbf{A}\}.$$

$\langle \mathbf{c}, \mathbf{a} \rangle$ "fires" if
$\mathbf{see}(\mathbf{s}) \in \mathbf{c}$ ($\mathbf{c}$ stands for "condition").

$$\prec \subseteq Ag_{rules} \times Ag_{rules}$$

is called inhibition-relation, $Ag_{rules} \subseteq Beh$.
We require $\prec$ to be a total ordering.

$\mathbf{b_1} \prec \mathbf{b_2}$ means: b_1 inhibits b_2,

$\mathbf{b_1}$ has priority over $\mathbf{b_2}$.


```
Function: Action Selection in the Subsumption Architecture
function action(p : P) : A
var fired : ℘(R)
var selected : A
begin
    fired ← {(c,a) | (c,a) ∈ R and p ∈ c}
    for each (c,a) ∈ fired do
        if ¬(∃(c',a') ∈ fired such that (c',a') ≺ (c,a)) then
            return a
        end-if
    end-for
    return null
end function action
```

Figure 5.1 Action Selection in the subsumption architecture.

Example 2.1 (Exploring a Planet)

A distant planet (asteroid) is assumed to contain gold. Samples should be brought to a spaceship landed on the planet. It is not known where the gold is. Several autonomous vehicles are available. Due to the topography of the planet there is no connection between the vehicles.

The spaceship sends off radio signals: **gradient field**.

Low Level Behaviour:
(1) **If** detect an obstacle **then** change direction.

2. Layer:
(2) **If** Samples on board **and** at base **then** drop off.
(3) **If** Samples on board **and** not at base **then** follow gradient field.

3. Layer:
(4) **If** Samples found **then** pick them up.

4. Layer:
(5) **If** true **then** take a random walk.

With the following ordering

$$(1) \prec (2) \prec (3) \prec (4) \prec (5).$$

Under which assumptions (on the distribution of the gold) does this work perfectly?

- Vehicles can **communicate indirectly** with each other:
  - they put off, and
  - pick up

  radiactive samples that can be sensed.

Low Level Behaviour:
: (1) **If** detect an obstacle **then** change direction.

2. Layer:
: (2) **If** Samples on board **and** at base **then** drop off.
: (3) **If** Samples on board **and** not at base **then** drop off two radioactive crumbs and follow gradient field.

3. Layer:
: (4) **If** Samples found **then** pick them up.
: (5) **If** radioactive crumbs found **then** take one and follow the gradient field (away from the spaceship).

4. Layer:
: (6) **If** true **then** take a random walk.

With the ordering $(1) \prec (2) \prec (3) \prec (4) \prec (5) \prec (6)$.

Pro: Simple, economic, efficient, robust, elegant.

Contra:
- Without knowledge about the environment agents need to know about the own local environment.
- Decisions only based on local information.
- How about bringing in **learning**?
- Relation between agents, environment and behaviours is not clear.
- Agents with ≤ 10 behaviours are doable. But the more layers the more complicated to understand what is going on.

2.2 BDI-Architecture

Belief, Desire, Intention.

```
Agent Control Loop Version 1
1. while true
2.     observe the world;
3.     update internal world model;
4.     deliberate about what intention to achieve next;
5.     use means-ends reasoning to get a plan for the intention;
6.     execute the plan
7. end while
```


Agent Control Loop Version 2
1. $B := B_0$; /* initial beliefs */
2. while true do
3. get next percept ρ;
4. $B := brf(B, \rho)$;
5. $I := deliberate(B)$;
6. $\pi := plan(B, I)$;
7. $execute(\pi)$
8. end while

Three main questions:

Deliberation: How to **deliberate**?

Planning: Once committed to something, how to **reach the goal**?

Replanning: What if during execution of the plan, things are running out of control and the **original plan fails**?

Belief 1:	Making money is important.
Belief 2:	I like computing.
Desire 1:	Graduate in Computer Science.
Desire 2 (Int.):	Pass the BSc.
Desire 3:	Graduate in time, marks are unimportant.
New Belief:	Money is not so important after all.
New Belief:	Working scientifically is fun.
Desire 4:	Pursue an academic career.
Desire 5 (Int.):	Make sure to graduate with honours.
Desire 6 (Int.):	Study abroad.

- Intentions are the most important thing.
- **Beliefs** and **intentions** generate **desires**.
- **Desires** can be inconsistent with each other.
- Intentions are recomputed based on the current intentions, **desires** and beliefs.
- Intentions should persist, normally.
- **Beliefs are constantly updated and thus generate new desires.**
- From time to time intentions need to be re-examined.

Agent Control Loop Version 3

1.
2. $B := B_0$;
3. $I := I_0$;
4. while true do
5. get next percept ρ;
6. $B := brf(B, \rho)$;
7. $D := options(B, I)$;
8. $I := filter(B, D, I)$;
9. $\pi := plan(B, I)$;
10. $execute(\pi)$
11. end while

Deliberation has been split into two components:

1. Generate options (desires).
2. Filter the right intentions.

(B, D, I) where $B \subseteq$ **Bel**, $D \subseteq$ **Des**, $I \subseteq \mathrm{Int}$

I can be represented as a **stack** (priorities are available)

An agent has commitments both to

end: the wishes to bring about,

means: the mechanism to achieve a certain state of affairs.

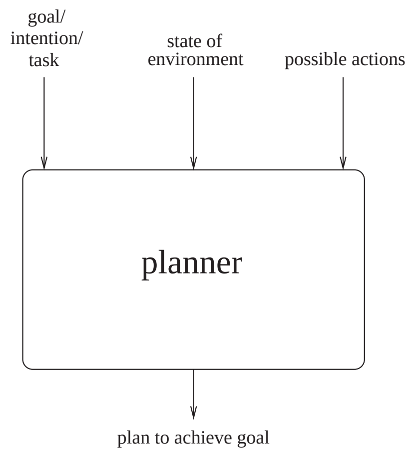
$\rightsquigarrow$ Means-end reasoning.

What is wrong with our current control loop?

It is overcommitted to both means and end.

No way to replan if something goes wrong.

```
Agent Control Loop Version 4

B := B_0;
I := I_0;
while true do
    get next percept ρ;
    B := brf(B, ρ);
    D := options(B, I);
    I := filter(B, D, I);
    π := plan(B, I);
    while not empty(π) do
        α := hd(π);
        execute(α);
        π := tail(π);
        get next percept ρ;
        B := brf(B, ρ);
        if not sound(π, I, B) then
            π := plan(B, I)
        end-if
    end-while
end-while
```

What is a plan, a planning algorithm?

Definition 2.2 (Plan)

A plan π is a list of primitive actions. They lead, by applying them successively, from the **initial state** to the **goal state**.

Still overcommitted to intentions!


```
Agent Control Loop Version 5
B := B_0;
I := I_0;
while true do
    get next percept ρ;
    B := brf(B, ρ);
    D := options(B, I);
    I := filter(B, D, I);
    π := plan(B, I);
    while not empty(π)
          or succeeded(I, B)
          or impossible(I, B)) do
        α := hd(π);
        execute(α);
        π := tail(π);
        get next percept ρ;
        B := brf(B, ρ);
        if not sound(π, I, B) then
            π := plan(B, I)
        end-if
    end-while
end-while
```

Still limited in the way the agent can reconsider its intentions.


```
Agent Control Loop Version 6

B := B0;
I := I0;
while true do
    get next percept ρ;
    B := brf(B, ρ);
    D := options(B, I);
    I := filter(B, D, I);
    π := plan(B, I);
    while not (empty(π)
            or succeeded(I, B)
            or impossible(I, B)) do
        α := hd(π);
        execute(α);
        π := tail(π);
        get next percept ρ;
        B := brf(B, ρ);
        D := options(B, I);
        I := filter(B, D, I);
        if not sound(π, I, B) then
            π := plan(B, I)
        end-if
    end-while
end-while
```


But reconsidering intentions is **costly**.

Pro-active vs. reactive

Extreme: *stubborn agents*, *unsure agents*.

What is better? Depends on the environment.

Let γ the **rate of world change**.

1. γ small: **stubbornness pays off**.
2. γ big: **unsureness pays off**.

What to do?

Meta-level control


```
Agent Control Loop Version 7

B := B0;
I := I0;
while true do
    get next percept ρ;
    B := brf(B, ρ);
    D := options(B, I);
    I := filter(B, D, I);
    π := plan(B, I);
    while not (empty(π)
            or succeeded(I, B)
            or impossible(I, B)) do
        α := hd(π);
        execute(α);
        π := tail(π);
        get next percept ρ;
        B := brf(B, ρ);
        if reconsider(I, B) then
            D := options(B, I);
            I := filter(B, D, I);
        end-if
        if not sound(π, I, B) then
            π := plan(B, I)
        end-if
    end-while
end-while
```

2.3 PROLOG

Prolog

Prolog = programmation en logique

is a logic programming language that is based on **Horn clauses** and **resolution**. We also use negation as failure to deal with incomplete information.

Programming constructs of Prolog that are important for our course are:

- terms,
- facts (also called atoms), and
- rules.

Other important notions are **queries**, and predefined constructs like **arithmetical expressions**, and lists.

Terms

- **Constants** starting with a digit or a lower-case letter:
  $abraham, lot, milcah, 1, 2, 3, \ldots$
- **Variables** starting with an upper-case letter:
  $X, Y, List, Family, \ldots$
- **(Compound) Terms** $f(t_1, t_2, \ldots, t_n)$ composed using constants, variables and functors:
  $s(0), s(s(0)), f(c_1, f(c_1, f(s(0), c_2))),$
  $first\_name\_of(einstein), father\_of(X), \ldots$
- **Ground Terms** are terms without variables. They are also called **fully instantiated**.

Facts (Atoms)

They express that a **relation holds between objects**: They can be true or not.

mother$(sarah, isaac)$.	*father*$(terach, abraham)$.
mother$(lea, dina)$.	*father*$(abraham, isaac)$.
mother$(sarah, ismael)$.	*father*$(abraham, ismael)$.
male$(esau)$.	*father*$(isaac, esau)$.
female$(dina)$.	*father*$(isaac, jakob)$.
	father$(jakob, dina)$.

father is also called a **binary predicate**. Similarly, facts are sometimes called predicates.

Facts (Atoms) (2)

- *father*$(Y, \textit{father}(Y,X))$ is **meaningless and not well-formed**.
- One could consider $plus(X,Y)$ as a binary function.
- Then $plus(1, plus(1,1))$ would make sense (and evaluate to something like 3, if this were available in the language).

Facts (Atoms) (3)

- **A belief base always consists of facts.**
- If a belief base does not contain a particular atom, say *father*$(isaac, terach)$, then we can also say that "**not** *father*$(isaac, terach)$" is true.
- Such negated facts are also called negated atoms. We use the notion literal, to denote an atom or its negation.
- Therefore a belief base is always consistent: It can not contain any contradictory information.

Rules, Clauses (1)

To define **new predicates**:

$$
\begin{array}{lcl}
\textit{son}(X,Y) & :- & \textit{father}(Y,X), \textit{male}(X). \\
\textit{daughter}(X,Y) & :- & \textit{parent}(Y,X), \textit{female}(X). \\
\textit{grandfather}(X,Y) & :- & \textit{father}(X,Z), \textit{parent}(Z,Y). \\
\textit{grandmother}(X,Y) & :- & \textit{mother}(X,Z), \textit{parent}(Z,Y). \\
\textit{parent}(X,Y) & :- & \textit{father}(X,Y). \\
\textit{parent}(X,Y) & :- & \textit{mother}(X,Y). \\
\textit{sibling}(X,Y) & :- & \textit{parent}(Z,X), \textit{parent}(Z,Y).
\end{array}
$$

Rules, Clauses (2)

They are also used to state important properties and **relations between predicates**:

$$
\begin{array}{rcl}
\textit{male}(Y) & :- & \textit{father}(Y,X). \\
\textit{female}(Y) & :- & \textit{mother}(Y,X).
\end{array}
$$

Queries

Given a set of rules and some facts (in a **belief base**), it is interesting to know whether something can be deduced from that (see the Wumpus example).
We can ask queries: They can be true, they can fail, or, if they contain variables, they can result in an instantiation of the variables.

son$(isaac, abraham)$?	true
plus$(1, 1, 2)$?	true
daughter(X, lea)?	true, X=dina
grandmother$(X, esau)$?	true, X=sarah
siblings$(esau, jakob)$?	true
mother$(terach, Y)$?	false?
plus$(1, 1, Y)$?	true, Y=2
plus(X, X, Y)?	true, X=0, Y=0

How to interpret the rules?

Example 2.3 (SLD-Resolution)

Let a program consist of the following rules

$$\begin{array}{lll} (1) & p(X,Z) & :- \; q(X,Y), p(Y,Z) \\ (2) & p(X,X). & \\ (3) & q(a,b). & \end{array}$$

The query Q we are interested in is "$p(X, b)$".
I.e. we are looking for all instances (terms) t for X such that $p(t, b)$ follows from the program.

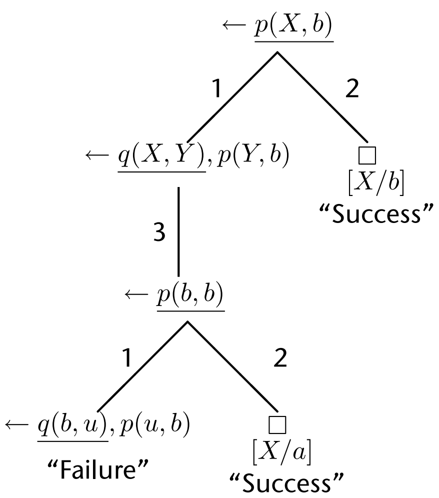


Figure 14: A finite SLD-Tree

Lists

We often use lists and consider $[\cdot]$ as a **function symbol**, written in **infix notation**.

$[]$	empty list
$[a]$	list with one element
$[a, b]$	list with two elements
$[a, b, c]$	list with three elements
$[a, [b, c]]$	list of lists

Predicates for lists

We assume we have a list of built-in predicates:

member$(a, [a, b, c])$	membership
member$(X, [a, b, c])$	membership
prefix$([a, b], [a, b, c])$	prefix
suffix$([b, c], [a, b, c])$	suffix
sublist$([b], [a, b, c])$	sublist
append$([a, b], [c, d], [a, b, c, d])$	appending two lists

Append

Suppose for a moment we do not have the **append** predicate available.

How can we define it using rules?

$$\textit{append}([], X, X)\ :-$$
$$\textit{append}([X|Y], Z, [X|T])\ :-\ \textit{append}(Y, Z, T)$$

Order of atoms

How can we define the **reverse** of a list?

Example 2.4 (Termination depends on the order)

Consider the following two programs:

$$(1)\ \textit{reverse}([X|Y], Z)\ :-\ \textit{append}(U, [X], Z), \textit{reverse}(Y, U)$$

$$(2)\ \textit{reverse}([X|Y], Z)\ :-\ \textit{reverse}(Y, U), \textit{append}(U, [X], Z)$$

together with the above definition for *append* and the query "$Q : \textit{reverse}([a|X], [b, c, d, b])$".

Order of atoms (cont.)

- The first program (1) leads to:
  $Q^1 :\ \textit{append}(U, [a], [b, c, d, b]),\ \textit{reverse}(X, U)$
- The second program (2) leads to:
  $Q^2 :\ \textit{reverse}(X, U),\ \textit{append}(U, [a], [b, c, d, b])$
- **We get different results using a naive execution!**
- This problem has been solved: Just do not care too much about the ordering!

not: negation-as-failure (1)

not has a very special meaning.

$$\begin{aligned} \mathit{reachable}(X) &:- \mathit{edge}(X,Y), \mathit{reachable}(Y). \\ \mathit{edge}(a,b). & \\ \mathit{edge}(b,a). & \\ \mathit{edge}(c,d). & \\ \mathit{out\_of\_reach}(X) &:- \textbf{not } \mathit{reachable}(X). \end{aligned}$$

What about the query *out_of_reach*(c)?

not: negation-as-failure (2)

Remember *female*(X), *male*(X). These predicates exclude each other. **How to express this with rules?**

$$\begin{aligned} \mathit{female}(X) &:- \textbf{not } \mathit{male}(X). \\ \mathit{male}(X) &:- \textbf{not } \mathit{female}(X). \end{aligned}$$

This ensures that we always have *male*(c) or *female*(c) in a belief base, but never both (unless explicitly stated).

System Functions

These (and other) functions are pre-defined:

Sum:	$1+1$	Subtract:	$2-3$
Quotient:	$5/8$	Multiply:	$13*21$
Minus:	-34	Absolute:	$\mathit{abs}(2)$
Square root:	$\mathit{sqrt}(16)$	Pi:	pi
Integer:	$\mathit{int}(2.1)$	Random:	$\mathit{random}(16)$
Time:	$\mathit{cputime}$	Ceiling:	$\mathit{ceil}(2.5)$
Floor:	$\mathit{floor}(2.5)$	Assign:	$\mathit{is}(X,3)$

Downloads

You can download SWI-Prolog here:

`http://www.swi-prolog.org/`

And you can download the relatives-example from our homepage.

Chapter 3. Some Scenarios

Content of this chapter:

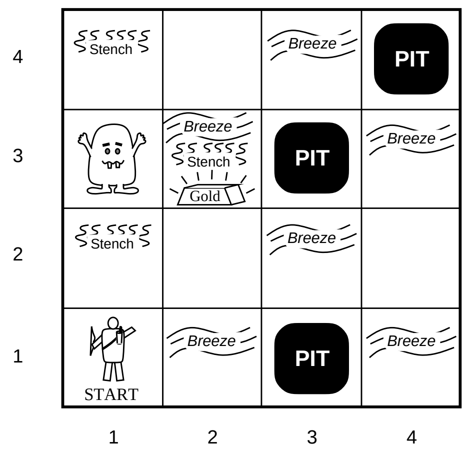
We present two interesting scenarios and our agent contest.

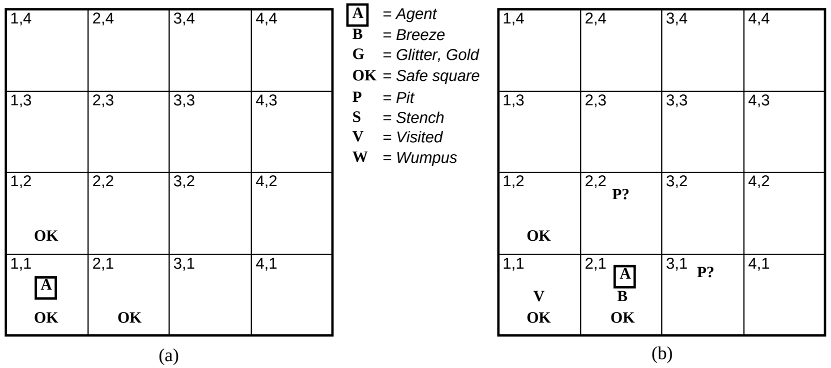
Wumpus: a simple yet difficult to solve deterministic (but incomplete) environment.

Harry and Sally: a simple test case for two agents that **communicate**.

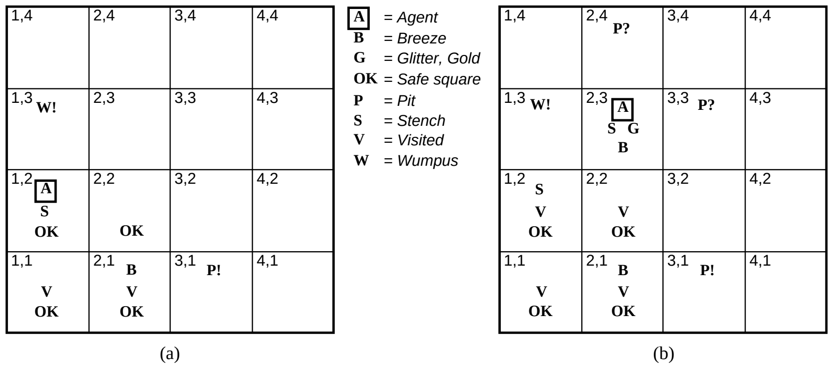
Agent Contest: where agents need to **collaborate** together to achieve a goal in an indeterministic environment.

3.1 Wumpus





(a) (b)



(a) (b)

Definition of suitable predicates

$S(i,j)$	cell (i,j) stenches
$B(i,j)$	cell (i,j) breezes
$Gl(i,j)$	cell (i,j) glitters
$Pit(i,j)$	cell (i,j) is a pit
$W(i,j)$	cell (i,j) contains a Wumpus

The first three predicates correspond to percepts of the agent.

The last two predicates can be determined based on the observations of the agent and the path it has taken.

General background knowledge

$$\begin{array}{lll}
\neg S(1,1) & \longrightarrow & (\neg W(1,1) \wedge \neg W(1,2) \wedge \neg W(2,1)) \\
\neg S(2,1) & \longrightarrow & (\neg W(1,1) \wedge \neg W(2,1) \wedge \neg W(2,2) \wedge \neg W(3,1)) \\
\neg S(1,2) & \longrightarrow & (\neg W(1,1) \wedge \neg W(1,2) \wedge \neg W(2,2) \wedge \neg W(1,3)) \\
 & & \\
S(1,2) & \longrightarrow & (W(1,3) \vee W(1,2) \vee W(2,2) \vee W(1,1))
\end{array}$$

+ many more!!

These have to be rewritten in the form of rules $a :- b$.

Belief base in initial state:

$\neg W(1,1), \neg S(1,1), \neg Pit(1,1), \neg B(1,1)$

Belief base after first move:

$\neg W(1,1), \neg S(1,1), \neg Pit(1,1), \neg B(1,1), \neg S(2,1), B(2,1)$

Belief base after second move:

$\neg W(1,1), \neg S(1,1), \neg Pit(1,1), \neg B(1,1), \neg S(2,1), B(2,1)$

Belief base after the 3rd move:

$\neg W(1,1), \neg S(1,1), \neg Pit(1,1), \neg B(1,1), \neg S(2,1), B(2,1),$
$S(1,2), B(2,1), \neg B(1,2)$

Question:
Can we deduce that the wumpus is located at (1,3)?

Answer:
Yes. This can be done automatically using built-in features of the programming language: By querying $W(1,3)$ against the belief base.

Lab excercise

We will implement agents that are situated in the wumpos world in one of the lab excercises.

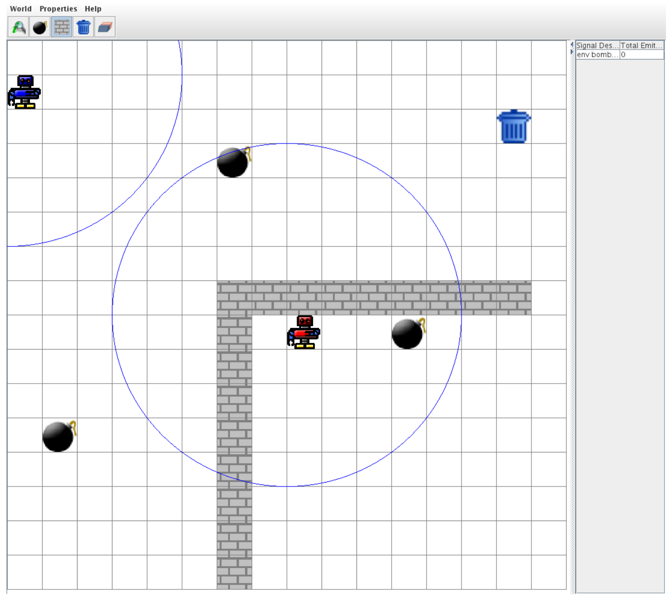
3.2 Harry and Sally

Harry and Sally

- This example is about two Agents living in a $n \times n$ grid.
- The world can contain bombs, walls and dustbins.

Sally: Searching for bombs, notifying Harry when a bomb has been found

Harry: Cleaning the grid by picking up the bombs and throwing it in a dustbin.



Details

Percepts:

- actual Position *pos*$(1,1)$
- visible bombs *bomb*$(1,2)$, *bomb*$(3,4), \ldots$

Actions:

- movement *north*$()$, *west*$()$, *south*$()$, *east*$()$
- bomb manipulation *pickup*$()$, *drop*$()$
- entering the environment physically *enter*$(1,1,blue)$
- send a message

Roles:

- Sally explores the environment (random), looks for bombs, and informs Harry about detected bombs.
- Harry waits until Sally sends bomb positions. Once Harry becomes aware of a bomb he moves to it and picks it up.

Sally

Initial beliefs: $\emptyset$

Initial goals:

- *search*$(blockWorld)$

Initial plans:

- enter the environment at the position $[8,8]$

Behavior: as long as Sally has the goal *search*$(blockWorld)$

1. go to a random position
2. sense visible bombs
3. if bombs are visible tell Harry about the position of the bomb

Harry

Initial beliefs: $\emptyset$

Initial goals:

- *clean*$(blockWorld)$

Initial plans:

- enter the environment at the position $[0, 1]$

Behavior: when Harry has the goal *search*$(blockWorld)$ and beliefs *bomb*(X, Y)

1. go to $[X, Y]$
2. pick up bomb
3. go to $[0, 0]$
4. drop bomb

Lab excercise

We will have a closer look at Harry and Sally in a lab excercise.

3.3 Agent Contest

Scenario: Gold Miners

Task: Implement a team of agents that collects more gold than the opponent.

Aim: Agents should cooperate and coordinate their actions. Agents can take on roles and split into subgroups to solve the overall task more efficiently.

Emerging behaviour instead of a hard-wired solution.

Environment: Can be quite indeterministic: percepts can be blurred, actions could fail with certain probability, ...

Environment

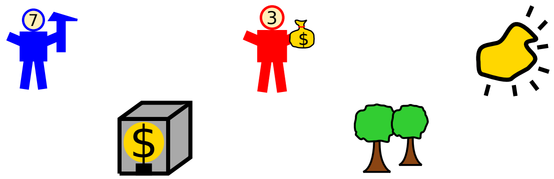


Figure 15: Elements in the environment

- Agents
- Gold
- Obstacles
- Depot

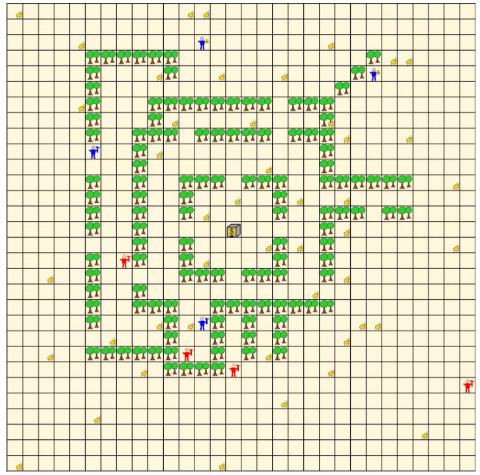


Figure 16: Gold Miners 2006: CLIMABot (blue) vs. brazil (red)

Details

Discrete Simulation: in each step do

- send perceptions to agents
- wait for agents' actions or timeout
- let agents act

Tournament Structure:

- maximum step duration around 4 seconds
- approx. 1000 steps per simulation
- 3 simulations = 1 match
- each team plays against all others, 1 match per pair

Technical details

- Grid size: 30×30
- Perception failure: 1%
- Action failure: 2%
- Occupying the depot leads to teleporting the agent

Lab excercise

We will implement agent teams in the lab excercises and let the teams compete against each other. The better one wins!

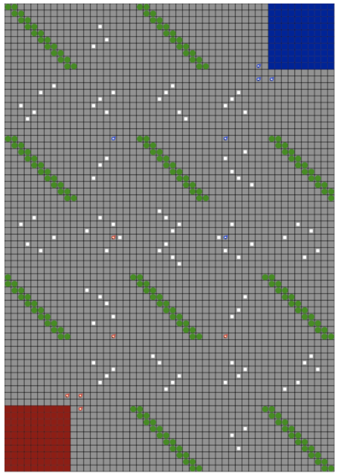
Scenario: Cows and Cowboys

Task: Implement a team of agents that collects more cows than the opponent.

Aim: Agents have to cooperate and coordinate their actions. Agents can take on roles and split into subgroups to solve the overall task more efficiently.
Emerging behaviour instead of a hard-wired solution.

Environment: Can be quite indeterministic: behaviour of cows, percepts can be blurred, actions could fail with certain probability, ...

Environment



- Cows
- Cowboys
- Corrals
- Obstacles

What is the optimal solution?

We do not know!

Details

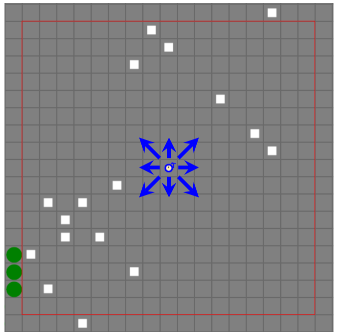
Discrete Simulation: in each step do

- send perceptions to agents
- wait for agents' actions or timeout
- let agents act and move cows

Tournament Structure:

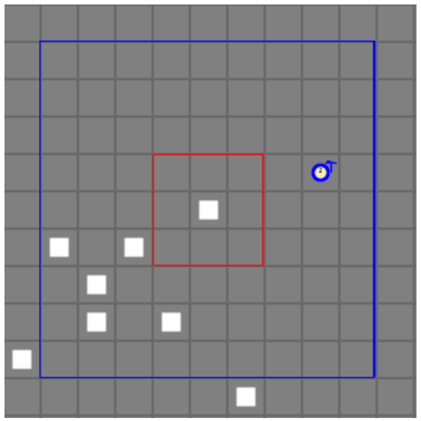
- maximum step duration around 4 seconds
- approx. 1000 steps per simulation
- 3 simulations = 1 match
- each team plays against all others, 1 match per pair

Agents



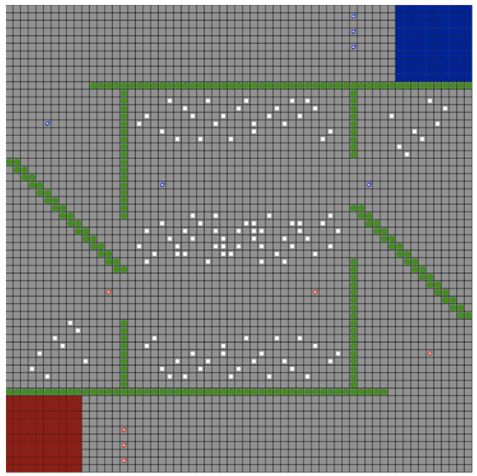
- fixed visibility range (square)
- actions: move to one of eight directions

Cows

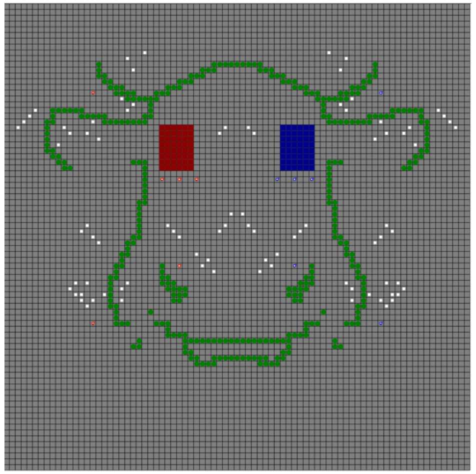


- visibility range (square)
- afraid of: agents, obstacles
- feel good: near other cows and empty spaces
- actions: move to one of eight directions
- slower than agents

Map: Razoredge



Map: Cowskullmountain



Chapter 4. 2APL

Content of this chapter:

We introduce 2APL and illustrate how to construct agents using the provided syntactical constructs.

1. Abstractions: IDE, deliberation cycle, recursive plans.
2. Abstraction levels.
3. 2APL programming constructs: Bases, rule bases, operations on them.

4.1 Abstractions in MAS

Languages for Cognitive Agents (1)

Programming Languages for MAS
=
Data Structures + **Programming Instructions**
E.g., 2APL, 3APL, Jason, Jadex, Jack

- Data Structures to represent mental state
  - **Beliefs** : General and specific Information about environment
  - **Goals** : Objectives that agent want to reach
  - **Events** : Observations of (environmental) changes
  - **Capabilities** : Actions that agent can perform
  - **Plans** : Procedures to achieve objectives
  - **Reasoning rules** : Reason about goals, events and plans
    - planning rules (goal $\longrightarrow$ plan)
    - event rules (event $\longrightarrow$ plan)
    - plan revision rules (plan $\longrightarrow$ plan)

Languages for Cognitive Agents (2)

- Programming Instructions to process mental states
  - Select Event
  - Plan Goals
  - Select Plans
  - Execute Plans
  - Select Rules
  - Apply Rules

Agent Interpreter or Agent Deliberation is a loop consisting of such instructions. The loop determines the behavior of the agent.

Motivation (1)

- Languages for implementing MAS: **Jack**, **Jadex**, **Jason**, **3APL**, **ConGoLog**, **MetateM**, **IMPACT**, **CLAIM**, **MINERVA**, **Go!**
- Efficient implementation of MAS architectures: Individual Cognitive Agents, Shared Environment, Multi-Agent Organisation

Motivation (2)

- Support for Programming Principles: Recursion, Compositionality, Abstraction, Exception Handling, Encapsulation, Autonomy, Reusability, Heterogeneity, Legacy Systems
- Integrated Development Environment (IDE): Editor, Debugging and Monitoring Facility, Support the Development of Individual Agents, Multi-Agent Organisation, and Shared Environment

Features of 2APL (1)

Programming Constructs:

- Multi-Agent System: Which and how many agents to create? Which environments? Which agent can access which environment?
- Individual Agent: Beliefs, Goals, Plans, Events, Messages

Features of 2APL (2)

Programming Principles and Techniques:

- Abstraction: Procedures/Recursion in Plans
- Error Handling: Plan Failure and their revision by Internal Events, Execution of Critical Region of Plans
- Legacy Systems: Environment and External Actions
- Encapsulation: Including 2APL files in other 2APL files
- Autonomy: Adjustable Deliberation Process

Features of 2APL (3)

Integrated Development Environment:

- 2APL platform is built on JADE and uses related tools
- Editor with High-Lighting Syntax
- Monitoring mental attitudes of individual agents, their reasoning and communications
- Executing in one step or continuous mode
- Visual Programming of the Deliberation Process

4.2 Programming in 2APL

Interlude – Grammar Notation

TMB: Hier brauchen wir ein paar Beispiele an der Tafel.

- Non-terminals: $\langle zero \rangle$, $\langle one \rangle$, $\langle two \rangle$...
- Terminals: ''0'', ''1'', ''2'', ...
- Rules: $\langle zeroOneZero \rangle$:= ''0'' ''1'' ''0'' ;
- Choice: $\langle digit \rangle$:= ''0'' | ''1'' | ''2'' | ''3'' | ...;
- Ommission or repetition:
  $\langle anystring \rangle$:= { ''0'' | ''1'' };
- Repetition: $\langle anystring2 \rangle$:= (''0'' | ''1'')$^+$;
- Option: $\langle oneOrOneOne \rangle$:= [''1''] ''1'';

2APL Platform

2APL = MAS Progr. + Agent Progr.

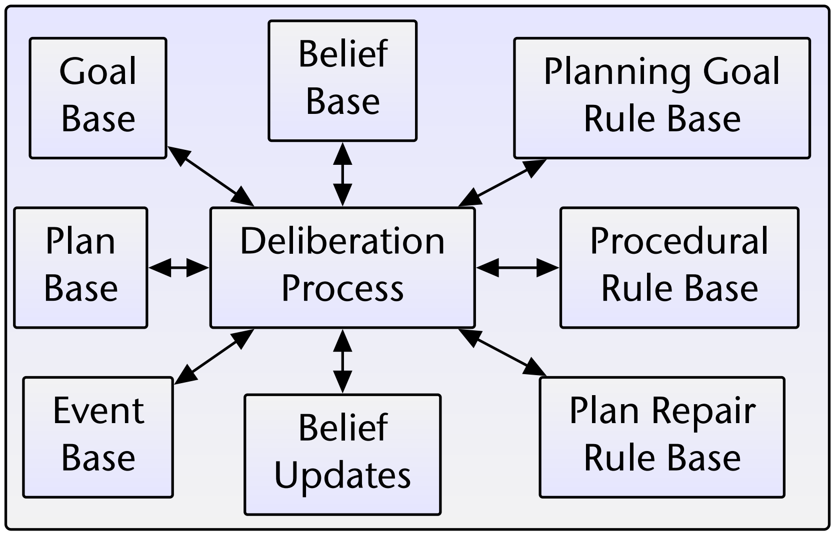
Abstraction Levels

Individual Agent: Autonomy, Situatedness, Proactivity
- Cognitive concepts:
  beliefs, **goals**, **plans**, **actions**
- Deliberation and control:
  sense/reason/act, **reactive/pro-active**

Multi-Agent: Social and Organizational Structures
- **Roles**: functionalities, activities, and responsibilities
- **Organizational Rules**: constraints on roles and their interactions
- **Organizational Structures**: topology of interaction patterns, control of activities

Environment: Resources and Services that MAS can access and control

Data Structures



Data Structures

- Beliefs: what the agent knows about the world
- Belief Updates: how the agent updates its beliefs
- Goals: states that the agent wants to achieve
- Plans: how to act
- Events: messages, external events from external environments
- Planning Goal Rules: which plan to instantiate in respect to the beliefs and goals
- Procedural Rules: which plans to instantiate in reaction to messages, events and abstract actions
- Plan Repair Rules: how to react to a failed action

4.3 Syntax

MAS Definition

$$
\begin{array}{lcl}
\langle MAS\_Prog \rangle & = & (\,\langle agentname\rangle \text{ '':'' } \langle filename\rangle\ [\langle int\rangle] \\
 & & [\langle environments\rangle]\,)^+ ; \\
\langle agentname\rangle & = & \langle ident\rangle ; \\
\langle filename\rangle & = & \langle ident\rangle \text{''}\texttt{.2apl}\text{''} ; \\
\langle environments\rangle & = & \text{''@''}\langle ident\rangle\ \{\text{'',''}\langle ident\rangle\} ;
\end{array}
$$

Examples:

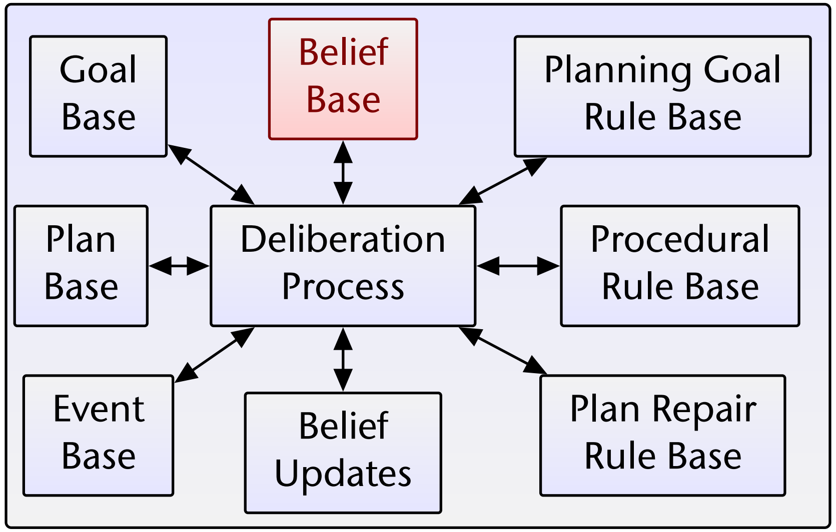
```
harry : harry.2apl @blockworld
sally : sally.2apl @blockworld

unit : unit.2apl 3 @env1,env2
```


Agent Programs

$$
\begin{array}{lcl}
\langle AgentProg\rangle & = & \{\ \text{"Include:"}\langle ident\rangle \\
 & | & \text{"Beliefupdates:"}\langle BelUpSpec\rangle \\
 & | & \text{"Beliefs:"}\langle beliefs\rangle \\
 & | & \text{"Goals:"}\langle goals\rangle \\
 & | & \text{"Plans:"}\langle plans\rangle \\
 & | & \text{"PG rules:"}\langle pgrules\rangle \\
 & | & \text{"PC rules:"}\langle pcrules\rangle \\
 & | & \text{"PR rules:"}\langle prrules\rangle\ \} ;
\end{array}
$$

This is how the initial state of an agent is defined.

Data Structures



Beliefs

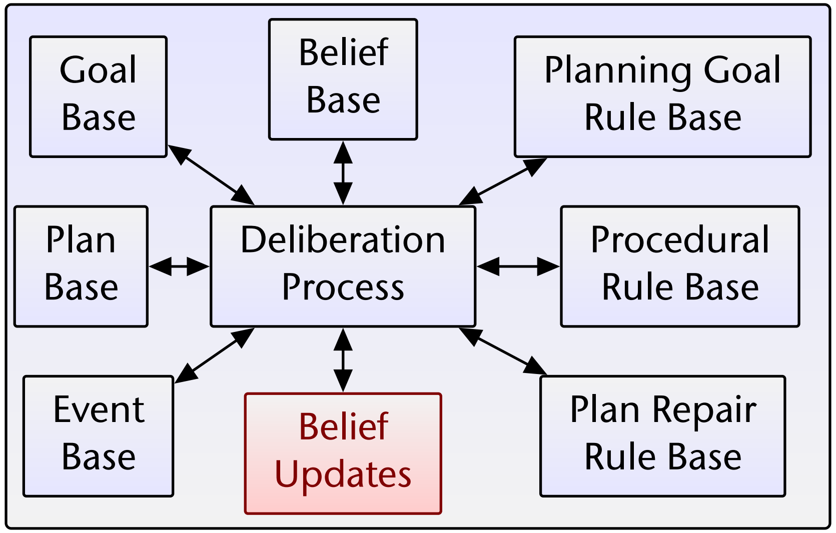
$$\langle belief\rangle = (\ \langle ground\_atom\rangle\ "."\ |\ \langle atom\rangle\ ":-"\ \langle literals\rangle\ "."\ )^{+}\ ;$$

Example:

```
Beliefs:
  start(0,1).
  bomb(3,3).
  clean( blockWorld ) :- not bomb(X,Y),
                         not carry(bomb).
```

⇝ Prolog facts and rules

Data Structures



Belief Updates

$$\langle BelUpSpec\rangle = (\ "\{"\langle belquery\rangle"\}"\ \langle beliefupdate\rangle\ "\{"\langle literals\rangle"\}"\ )^{+}\ ;$$

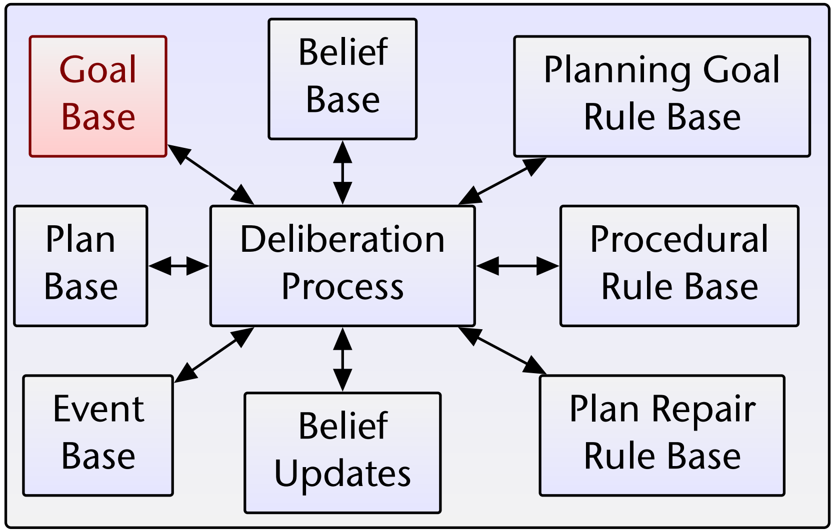
Structure:

```
{ pre } BeliefUpdateAction { post }
```

Belief Updates Example

```
BeliefUpdates:
  { bomb(X,Y) }
    RemoveBomb(X,Y)
      { not bomb(X,Y) }
  { true }
    AddBomb(X,Y)
      { bomb(X,Y) }
  { carry(bomb) }
    Drop( )
      { not carry(bomb) }
  { not carry(bomb) }
    PickUp( )
      { carry(bomb) }
```


Data Structures



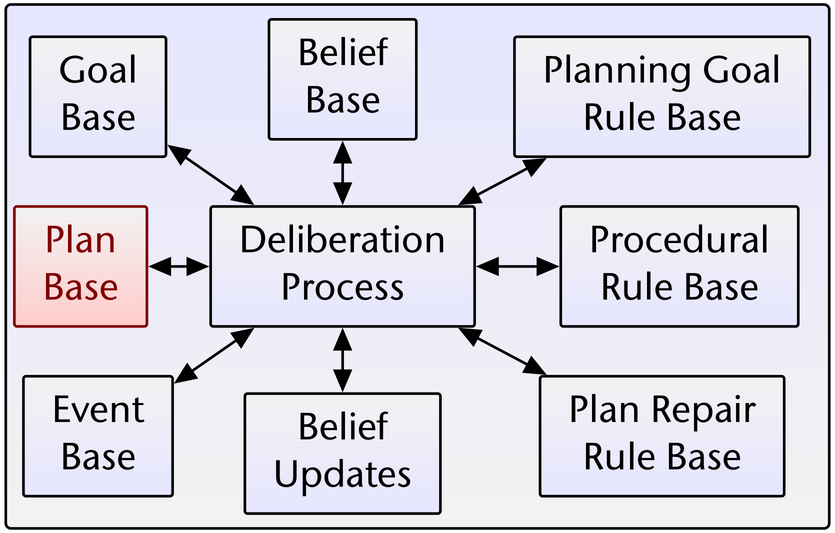
Goals

$\langle goals \rangle$ = $\langle goal \rangle$ { "," $\langle goal \rangle$ } ;
$\langle goal \rangle$ = $\langle ground\_atom \rangle$ { "and" $\langle ground\_atom \rangle$ } ;

Example:

```
Goals:
  clean( blockWorld )
```


Data Structures



Plans

Plans = Basic Actions + Plan Operators

Basic Actions

An agent can

- do nothing,
- update its beliefs,
- send a message,
- act in the external environment(s),
- execute a plan,
- test its beliefs/goals,
- adopt a goal, and
- drop a goal.

Basic Actions

$$
\begin{aligned}
\langle baction \rangle = & \text{''}\texttt{skip}\text{''} \\
& \mid \langle beliefupdate \rangle \\
& \mid \langle sendaction \rangle \\
& \mid \langle externalaction \rangle \\
& \mid \langle abstractaction \rangle \\
& \mid \langle test \rangle \\
& \mid \langle adoptgoal \rangle \\
& \mid \langle dropgoal \rangle \ ;
\end{aligned}
$$

Belief Update Action

- Let T be the function that takes a belief update action and a belief base, and returns the modified belief base if the pre-condition of the action is entailed by the agent's belief base.
- This function can be defined based on the specification of the belief update actions.
- If the belief update action cannot be applied, the action fails.

Belief Update Action

$\langle beliefupdate \rangle = \langle Atom \rangle$;

Examples:

`PickUp( )`

RemoveBomb(X, Y)

Applies the `BeliefUpdates`.

Send Action

$\langle sendaction \rangle$ = "send(" $\langle iv \rangle$ "," $\langle iv \rangle$ "," $\langle atom \rangle$ ")" ;
| "send(" $\langle iv \rangle$ "," $\langle iv \rangle$ "," $\langle iv \rangle$ "," $\langle iv \rangle$ ","
$\langle atom \rangle$ ")" ;

Example:

`send( harry , inform , bombAt( X1, Y1 ) )`

Informs `harry` that there is a bomb at a specific position.

Send Action

- An agent can send a message to another agent by means of the send(j, p, l, o, ϕ) action.
- An agent is assumed to be able to receive a message that is sent to it at any time. The received message is added to the event base of the agent.
- **Synchronized Communication:** The execution of the send action will broadcast a message which will be received by the receiving agent and added in its event base.

External Action

$\langle externalaction \rangle$ = "@"$\langle ident \rangle$"("$\langle atom \rangle$ "," $\langle Var \rangle$ ")" ;

Example:

`@blockworld( pickup( ), L1 )`

Execute the action `pickup()` in the environment `blockworld`. The return value is stored in `L1`.

Abstract Action

$\langle abstractaction \rangle = \langle atom \rangle$;

Example:

`goto( X, Y )`

Executes the plan `goto( X, Y )`.

Test Action

$\langle test \rangle =$ ''B('' $\langle belquery \rangle$ '')''
| ''G''($\langle goalquery \rangle$ '')''
| $\langle test \rangle$ ''&'' $\langle test \rangle$;

Example:

B(bomb (3,3))

G(clean(blockworld))

B(POS = [X , Y]);

$\rightsquigarrow$ Prolog queries to belief- and goal-base

Adopt Goal Action

$\langle adoptgoal \rangle =$ ''adopta('' $\langle goalvar \rangle$ '')''
| ''adoptz('' $\langle goalvar \rangle$ '')'' ;

Example:

adopta(clean(blockWorld))

adoptz(clean(blockWorld))

Drop Goal Action

$\langle dropgoal \rangle =$ ''dropgoal('' $\langle goalvar \rangle$ '')''
| ''dropsubgoals('' $\langle goalvar \rangle$ '')''

Example:

dropgoal(clean(blockWorld))

About Goal Actions

- As the goal base of an agent is defined as a list, `adopta(`ϕ`)` action adds goal ϕ to the begin of goal base and `adoptz(`ϕ`)` adds it to the end of the goal base.
- If the goal is already believed, the adopt goal action fails.
- Goals can be dropped and removed from the goal base by means of `dropgoal(`ϕ`)`, `dropsubgoals(`ϕ`)`, and `dropsupergoals(`ϕ`)` actions.
- The first action removes the goal ϕ from the goal base, the second action removes all subgoals of the goals ϕ from the goal base, and the third action removes all goals that entail the goal ϕ from the goal base. These actions succeeds always.

Plan Operators

Basic actions can be composed using operators for

- sequence,
- conditional choice,
- conditional iteration, and
- atomic plans.

Plan Operators

$$
\begin{aligned}
\langle plans \rangle &= \langle plan \rangle \; \{ \text{'',''} \; \langle plan \rangle \} \; ; \\
\langle plan \rangle &= \langle baction \rangle \mid \langle sequenceplan \rangle \mid \langle ifplan \rangle \mid \langle whileplan \rangle \\
&\quad \mid \langle atomicplan \rangle \; ;
\end{aligned}
$$

Sequence

$$\langle sequenceplan \rangle = \langle plan \rangle \; \text{'';''} \; \langle plan \rangle \; ;$$

Example:

```
goto( 0, 0 );
@blockworld( drop( ), L2 );
Drop( )
```

Conditional Choice

$\langle ifplan\rangle$ = ''`if`'' $\langle test\rangle$ ''`then`'' $\langle scopeplan\rangle$ [''`else`'' $\langle scopeplan\rangle$] ;

Example:

```
if B( not bomb( A, B ) ) then
{
  AddBomb( X, Y );
  adoptz( clean( blockWorld ) )
}
else
{
  AddBomb( X, Y )
}
```


Conditional Iteration

$\langle whileplan\rangle$ = ''`while`'' $\langle test\rangle$ ''`do`'' $\langle scopeplan\rangle$;

Example:

```
while G( clean(blockworld) ) do
{
  skip
}
```


Atomic Plan

$\langle atomicplan\rangle$ = ''['' $\langle plan\rangle$ '']'' ;

Example:

```
[AddBomb( X, Y );
adoptz( clean( blockWorld ) )]
```


Plans Example 1

```
{
  goto( X, Y );
  @blockworld( pickup( ), L1 );
  PickUp( );
  RemoveBomb( X, Y );
  goto( 0, 0 );
  @blockworld( drop( ), L2 );
  Drop( )
}
```

Plans Example 2

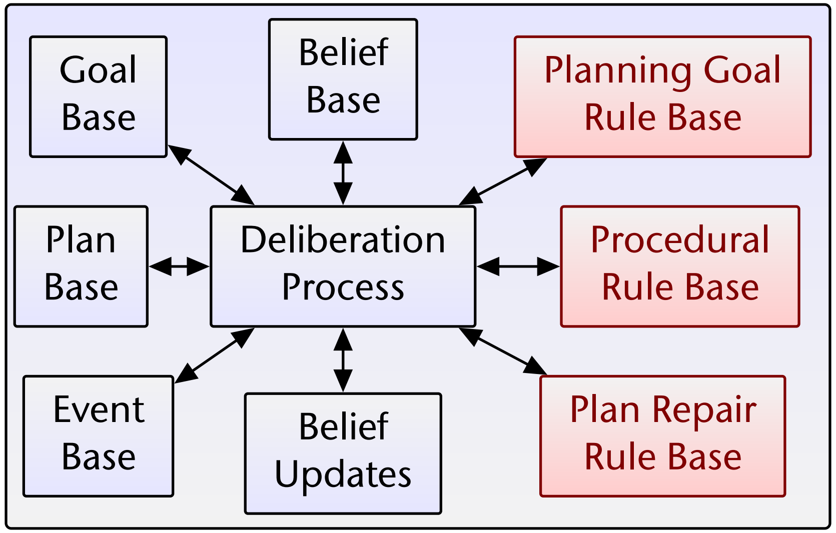
```
{
  if B( not bomb( A, B ) ) then
  {
    AddBomb( X, Y );
    adoptz( clean( blockWorld ) )
  }
  else
    { AddBomb( X, Y )}
}
```


Initial Plans vs Reasoning Rule Plans

An initial plan:

```
Plans:
{
  B(start(X,Y)) ;
  @blockworld( enter( X, Y, blue ), L )
}
```


Data Structures



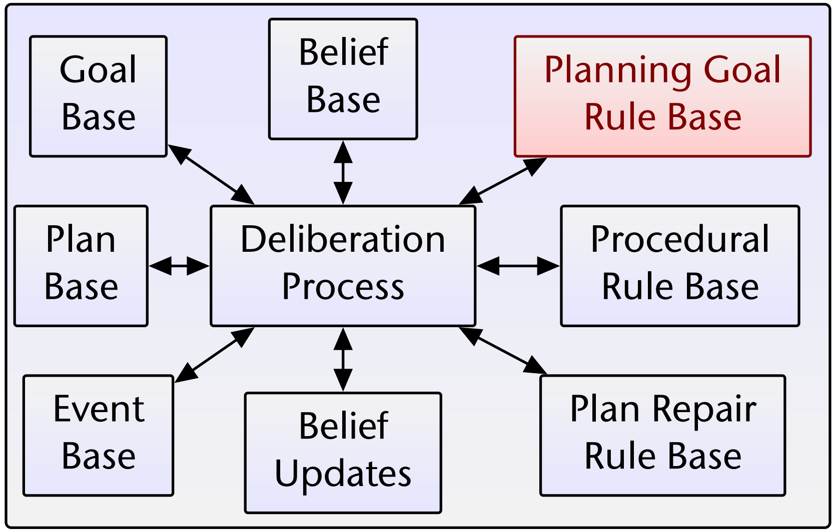
Reasoning Rules

Planning Goal Rules (PG Rules): generate plans if an agent has certain goals and beliefs.

Procedural Rules (PC Rules): generate plans as a response to the reception of a message, events generated by the external environment(s), and the execution of abstract actions.

Plan Repair Rules (PR Rules): generate plans if an agent's actions fail.

Data Structures



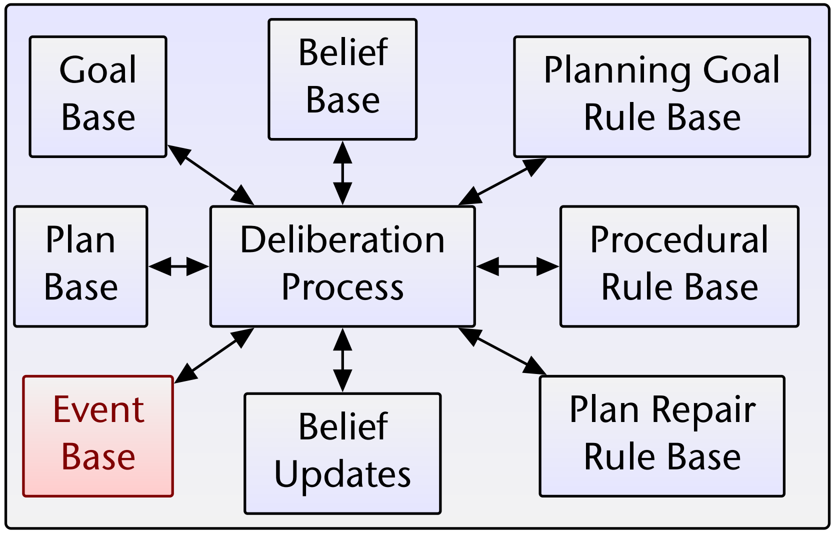
Planning Goal Rules (PG Rules)

$\langle pgrules \rangle$ = $\langle pgrule \rangle$+ ;
$\langle pgrule \rangle$ = [$\langle goalquery \rangle$] "<-" $\langle belquery \rangle$ "|" $\langle plan \rangle$;

PG Rules Example

```
PG-rules:
  clean( blockWorld ) <- bomb( X, Y ) |
  {
    goto( X, Y );
    @blockworld( pickup( ), L1 );
    PickUp( );
    RemoveBomb( X, Y );
    goto( 0, 0 );
    @blockworld( drop( ), L2 );
    Drop( )
    }
```


Data Structures

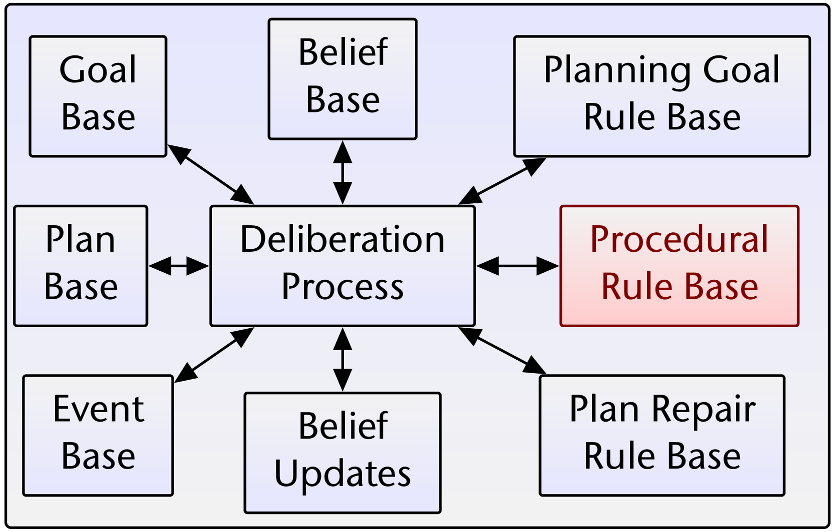


Events

There are three kinds of events:

- incoming messages,
- events from the external environment, and
- abstract action execution.

Data Structures



Procedural Rules (PC Rules)

$\langle pcrules \rangle$ = $\langle pcrule \rangle$+ ;
$\langle pcrule \rangle$ = $\langle atom \rangle$ "<-" $\langle belquery \rangle$ "|" $\langle plan \rangle$;

PC Rules Example (received message)

```
PC-rules:
  message(sally,inform,La,On,bombAt( X, Y ))
  <- true | {
     if B( not bomb( A, B ) ) then
     {
       AddBomb( X, Y );
       adoptz( clean( blockWorld ) )
     }
     else
     {
       AddBomb( X, Y )
     }
  }
```

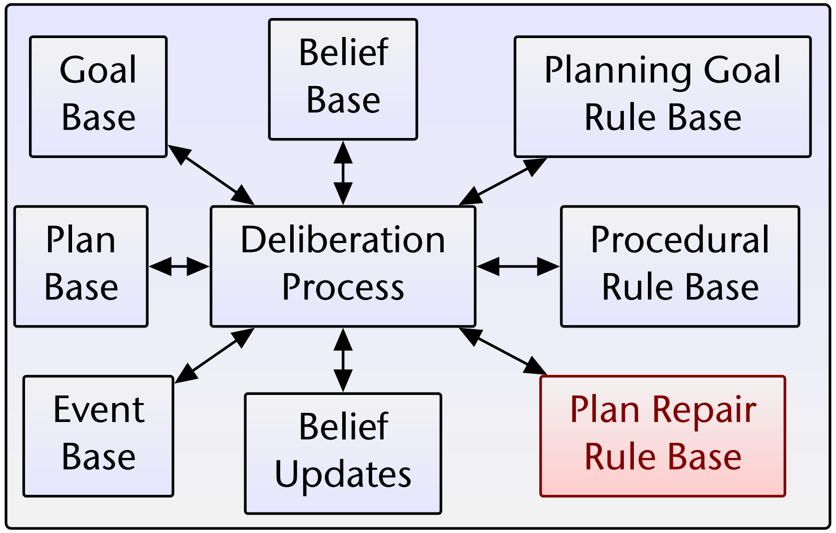
PC Rules Example (external event)

```
PC-rules:
  event( simResult(R), Env ) <- true | {
    print("Simulation is over.");
    print(R)
  }
```


PC Rules Example (abstract action)

```
PC-rules:
  goto( X, Y ) <- true |
  {
    @blockworld( sensePosition(), POS );
    B(POS = [A,B]);
    if B(A > X) then
    { @blockworld( west(), L );
      goto( X, Y )
    }
    else if B(A < X) then
    { @blockworld( east(), L );
      goto( X, Y )
    }
   ...
```


Data Structures



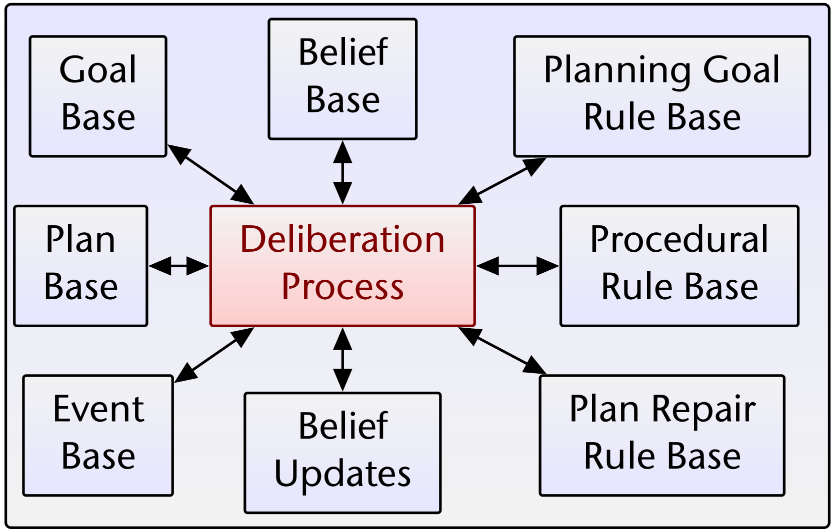
Plan Repair Rules (PR Rules)

$$\langle prrules\rangle = \langle prrule\rangle + ;$$
$$\langle prrule\rangle = \langle planvar\rangle \text{ "<-" } \langle belquery\rangle \text{ "|" } \langle planvar\rangle ;$$

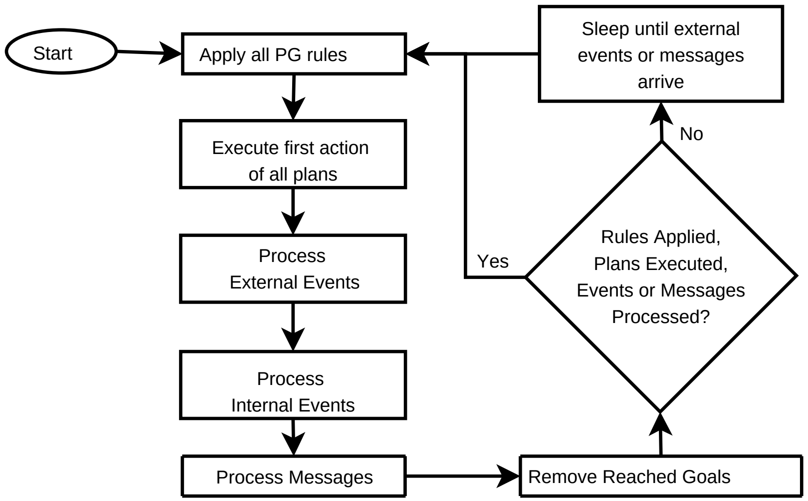
PR Rules Example

```
PR-rules:
  @blockworld( pickup( ) , L ) ; REST
  <- true | {
    @blockworld( sensePosition( ) , POS ) ;
    B( POS = [ X , Y ] );
    RemoveBomb( X , Y )
  }
```


Data Structures



Deliberation Process



Downloads

The complete grammar is contained in the 2APL user guide.

You can download the 2APL IDE and the user guide here:
`http://www.cs.uu.nl/2apl/`

2APL Programming Example

Harry and Sally will be explained in detail in the next lab excercise.