
Desenvolvimento de Sistemas Multiagentes

Escola Regional de Informática – PR

Jomi Fred Hübner
(FURB / DSC)

Rafael Heitor Bordini
(Univ. of Durham)

Renata Vieira
(UNISINOS)

agosto de 2004

Objetivo

- Apresentar uma visão geral do que é e para que serve a **abordagem** de Sistemas Multiagentes
- Introdução a uma arquitetura para desenvolvimento dos agentes (BDI)
- Mostrar algumas ferramentas

Roteiro

- **Sistemas Multiagentes**

- ★ Motivação
- ★ Vantagens
- ★ Aplicações

- **Agentes**

- ★ Tipos
- ★ Teorias
- ★ Arquiteturas
- ★ Linguagens
- ★ Ferramentas
- ★ Comunicação

Sistemas Multiagentes

Motivações para SMA: novas **Fontes de Inspiração**

- Fontes de “inspiração” para a Computação
 - ★ Filosofia: Orientação a Objetos
 - ★ Psicologia: Inteligência Artificial
 - ★ Lógica: Sistemas de raciocínio
 - ★ Biologia: Redes Neurais, Algoritmos genéticos
- Sociologia e Etologia: ?

Motivações para SMA: **Coletividade**

- IA × SMA
 - ★ IA: “construir **uma** entidade artificial apresentar propriedades inteligentes”. (psicologia + engenharia)
 - ★ Abordagens
 - * Simbolista (mente)
 - * Conexionista (cérebro)
 - “Inteligência” como processo **emergente**
 - ★ Formigueiro
 - ★ Cérebro
 - ★ Cidade
- O todo é mais que a soma das partes**
- O que tem no “todo” que não tem nas partes?

Motivações para SMA: desenvolvimento de Sistemas

Ciclo “clássico” de desenvolvimento de sistemas (distribuído):

- Identificação de requisitos (problema)
- Análise (do problema)
- Projeto (de solução para o problema)
- Implementação
- Teste

Se o problema muda, a solução tem que mudar! (forte acoplamento entre os módulos)

Propostas de solução:

- objetos, componentes, webservices, **agentes**

Motivações para SMA: **autonomia**

“Devido a uma falha inesperada, uma espaçonave que se aproximava de Saturno perde contato com sua base na Terra, ficando desorientada. Ao invés de desaparecer no espaço, a nave percebe a falha, faz um diagnóstico do problema e o corrige procurando restabelecer o contato com a base.”

(Wooldridge, 2002)

“Depois de uma semana com um frio ‘de rachar’, você decide, na sexta-feira a tarde, que gostaria de passar o fim de semana em um lugar quente e agradável. Depois de passar suas preferências para seu celular, ele ‘conversa’ com vários web sites que vendem passagens, fazem reservas, alugam carros, etc. Após uma dura negociação, o celular apresenta o pacote completo para o fim de semana.”

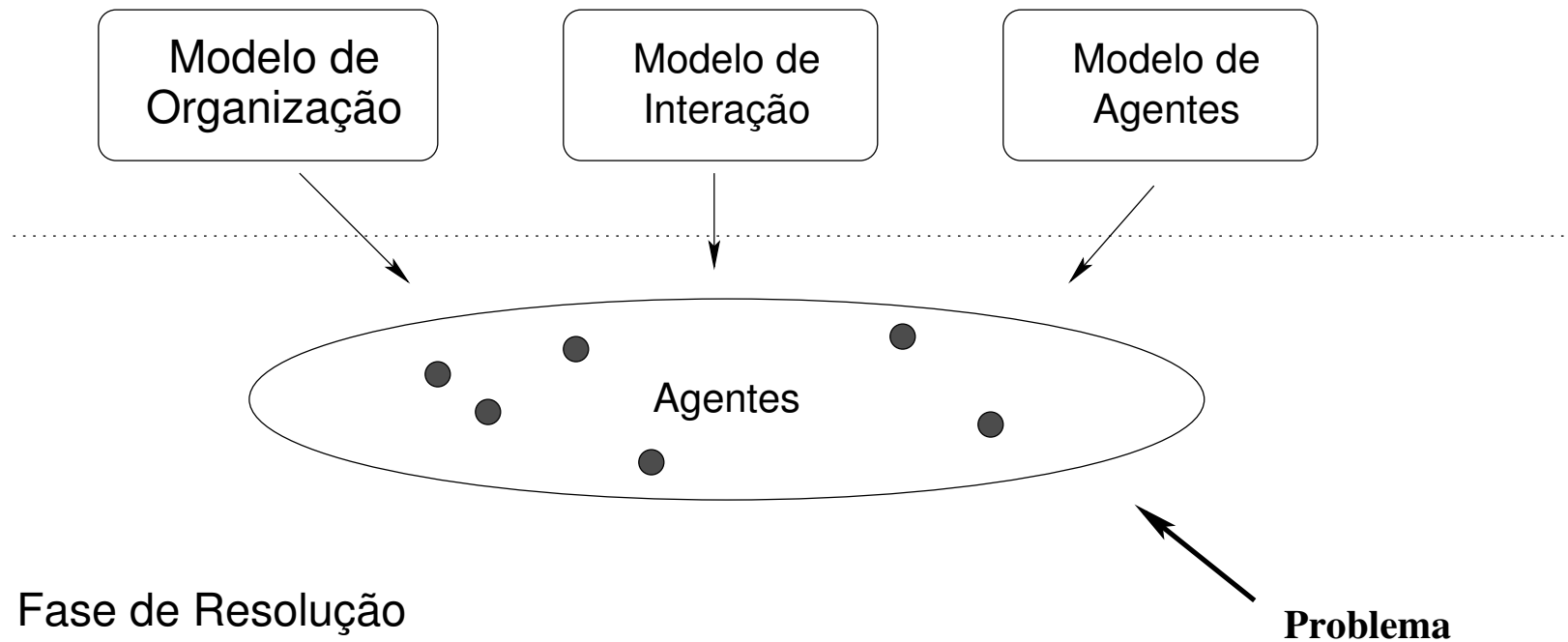
(Wooldridge, 2002)

Características dos SMA

- Os agentes são concebidos independentemente de um problema particular (exemplo: sistema operacional, WebServices).
- A interação entre os agentes não é projetada anteriormente, busca-se definir protocolos que possam ser utilizados em situações genéricas (exemplo: ContractNet).
- A decomposição de tarefas para solucionar um dado problema pode ser feita pelos próprios agentes (exemplo: sistema operacional orientado a agentes).
- Não existe um controle centralizado da resolução do problema (exemplo: controle de aeroporto, Internet × Matrix).
- Os agentes são **Autônomos**.

Ciclo (**ideal**) proposto pela área de SMA

Fase de Concepção



Vantagens dos SMA

- Permite conceber **sistemas abertos**.
- Viabilizam sistemas **adaptativos** e **evolutivos**.
- É uma **metáfora natural** para a modelagem de sistemas complexos e distribuídos.
- Toma proveito de ambientes **heterogêneos** e **distribuídos**.

Exemplos de aplicação

- Controle de Tráfego Aéreo
- Gerência de Negócios (B2B)
- Interação Humano-Computador
- Ambientes de Aprendizagem
- Entretenimento e Jogos
- Telecomunicações, transportes e sistemas para a área de saúde
- Simulação Social

Exemplo:

Futebol do Robôs

Por que times?

- Pelas mesmas razões que motivaram os **SMA**
 - ★ alta complexidade e
 - ★ limitações temporais, espaciais e funcionais.
- Exemplos:
 - ★ Exploração planetária / subaquática
 - ★ Combate a incêndios em florestas
 - ★ Busca e resgate
 - ★ Remoção de minas terrestres
 - ★ Limpeza de grandes áreas
 - ★ Jogar bola (!)

Um time é um SMA

Vantagens dos times

- **Tempo** de execução: um time cobre uma grande área num tempo menor.
- **Custo**: um único robô exigiria maior capacidade de processamento, maior autonomia, maior robustez para executar uma tarefa complexa.
- **Redundância**: caso um robô falhe ou seja destruído, o restante do time pode continuar a tarefa.

Dificuldades em times

- Como comunicar adequadamente (como + quanto)?
- Como decompor uma tarefa e alocar as sub-tarefas?
- Como garantir que o time agir  coerentemente?
- Como os rob s reconhecer o e resolver o conflitos?
- Como dotar um time com capacidade de adapta o?

Mudar seu comportamento em resposta  s mudan as din micas do ambiente,  s mudan as de metas,  s mudan as de capacidades e composi o do time, etc.

Características dos times

- **Granularidade**: tamanho = número de robôs
- **Heterogeneidade**: diversidade sw/hw entre robôs
- **Comunicação**
 - ★ explícita: informações são intencionalmente trocadas
 - ★ implícita: informações são adquiridas por observação das ações dos outros robôs ou por rastros deixados
- Estrutura de **controle**
 - ★ centralizado: robôs se comunicam com um computador central, que distribui atribuições, metas, informações (robôs semi-autônomos, dependentes do computador central para decidir suas ações)
 - ★ distribuído: robôs tomam suas próprias decisões e agem de modo independente.

- **Cooperação**

- ★ não-ativa: robôs não compartilham explicitamente uma meta comum (possuem sub-metas individuais)
- ★ acidental (não intencional) e não-ativa. Ex: dois robôs demolindo uma parede
- ★ ativa: quando pelo menos alguns dos membros se reconhecem e trabalham conjuntamente para atingir meta comum explícita, caracterizando intencionalidade na cooperação (pode exigir sensores mais complexos). Ex: passe em futebol.

Exemplo de configurações de times

- Aplicações que **não** impõem **restrições drásticas no tempo** de execução e que requerem várias **repetições**: limpeza (praias, grandes estacionamentos), coleta (missões espaciais), busca e resgate
 - ★ time: homogêneo, alta granularidade, controle descentralizado, cooperação não-ativa, sem comunicação explícita
 - ★ robôs: paradigma reativo (poucos e simples comportamentos), concorrentes e independentes

- Aplicações com restrições drásticas na **eficiência** de sua execução
 - ★ time: heterogêneo, baixa granularidade, controle distribuído, cooperação ativa, comunicação explícita
 - ★ robôs: diferentes capacidades
 - ★ pontos importantes de projeto: mapeamento apropriado das sub aos robôs (em função das suas capacidades);

RoboCup

“By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.”

- Um novo desafio para a IA!

Categoria small size



Categoria **middle size**



Categoria “quatro pernas”



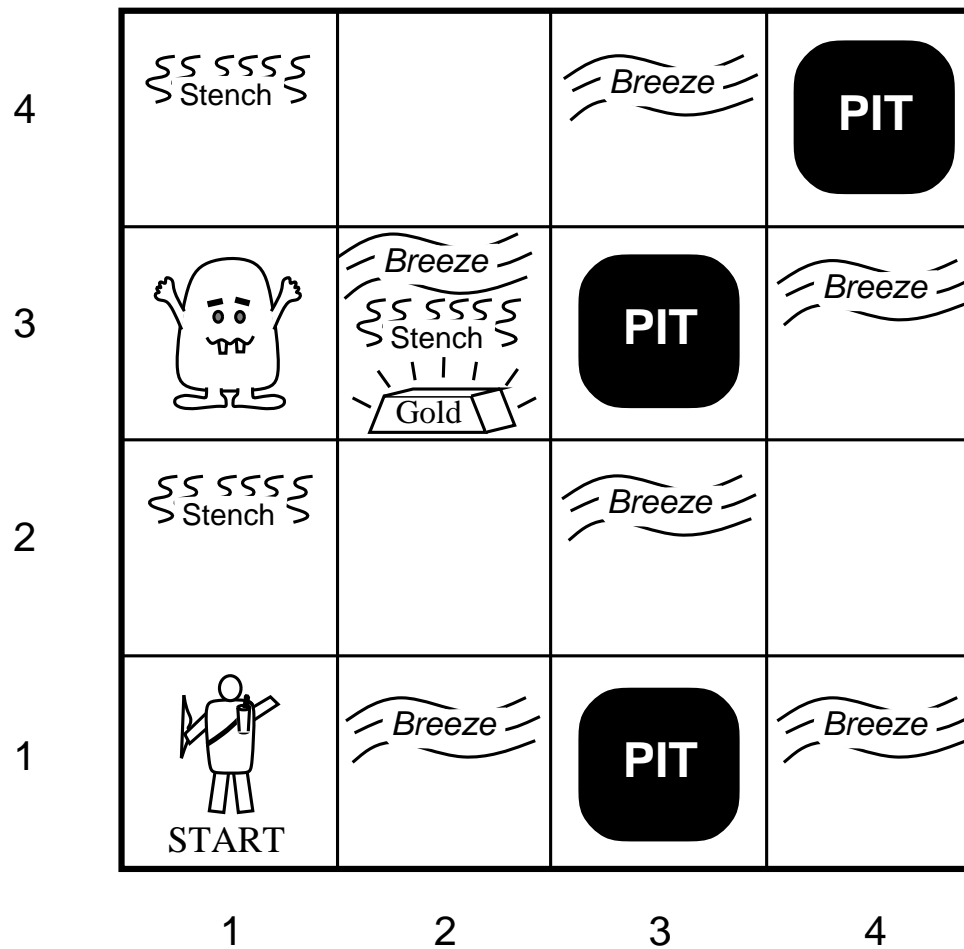
Categoria **Humanóides**



Categoria Simulador



Wumpus World (Russel & Norvig)



Ambiente do Wumpus

- Se o agente estiver em uma sala diretamente (não diagonalmente) ao lado da sala do Wumpus, perceberá um fedor (Stench).
- Em salas ao lado de uma sala com precipício (Pit), passa uma brisa (Breeze).
- Na sala com ouro, o agente percebe um brilho.
- O jogador tem apenas um tiro para tentar matar o Wumpus.
- Se o Wumpus for morto, dará um berro que será escutado em toda a caverna.
- O Jogador morre miseravelmente se ficar em uma sala com o Wumpus vivo ou entrar em uma sala com precipício.

O Agente Wumpus

- **Objetivo**: entrar na caverna, pegar o ouro e sair o mais rápido possível.
- **Percebe** em cada sala: fedor, brisa, brilho do ouro, se esta batendo em uma parede e o berro da morte do Wumpus.
- Pode **agir** da seguinte forma: virar 90 graus para direita ou esquerda, ir em frente, atirar no Wumpus, sair da caverna (só funciona na posição 1,1).

Inferências no Wumpus World

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 OK	2,2	3,2	4,2
1,1 A OK	2,1 OK	3,1	4,1

(a)

- A** = Agent
- B** = Breeze
- G** = Glitter, Gold
- OK** = Safe square
- P** = Pit
- S** = Stench
- V** = Visited
- W** = Wumpus

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 OK	2,2 P?	3,2	4,2
1,1 V OK	2,1 A B OK	3,1 P?	4,1

(b)

1,4	2,4	3,4	4,4
1,3 W!	2,3	3,3	4,3
1,2 A S OK	2,2 OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

(a)

- A** = Agent
- B** = Breeze
- G** = Glitter, Gold
- OK** = Safe square
- P** = Pit
- S** = Stench
- V** = Visited
- W** = Wumpus

1,4	2,4 P?	3,4	4,4
1,3 W!	2,3 A S G B	3,3 P?	4,3
1,2 S V OK	2,2 V OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

(b)

Agentes

SMA são sistemas compostos por vários **agentes** que interagem entre si.

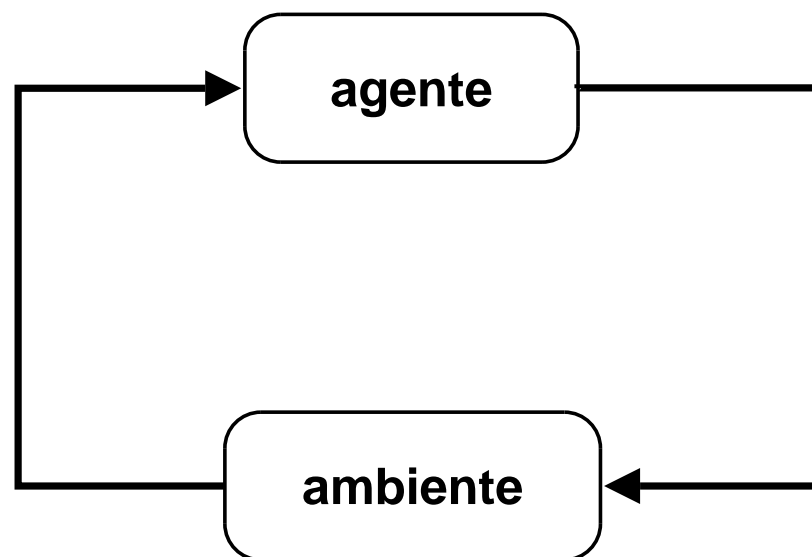
Definição de agente

“Um agente é um sistema computacional que está situado em um ambiente e que é capaz de agir autonomamente neste ambiente para atingir seus objetivos de projeto”

Existem muitos **tipos** de agentes. Estamos interessados naqueles que apresentam as seguintes propriedades:

- **Autônomo** (independente para com seus objetivos)

- **pro-ativo** (tem meta)
- **reativo** (reage rapidamente ao ambiente)
- **social** (coletivo, cooperativo)



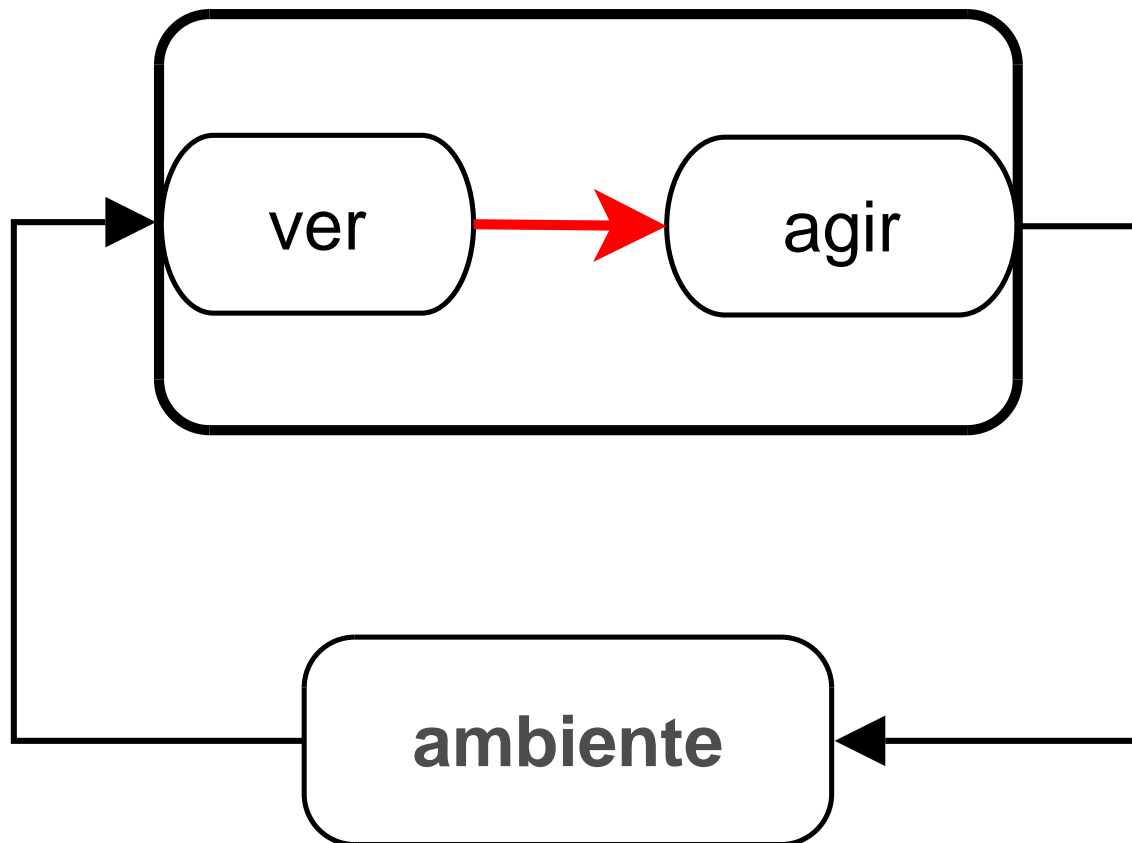
Agentes × objetos

- Objetos encapsulam um estado (atributos), realizam ações (métodos) e enviam mensagens.
- Agentes e objetos têm coisas em comum, mas possuem algumas diferenças
 - ★ **Autonomia**
 - * Objetos têm controle sobre seu estado, mas não sobre seu comportamento.
 - * Invocação de um método × requisição de um serviço.
 - ★ **Controle de execução**
 - * Um agente tem sua própria *thread* (pro-atividade).

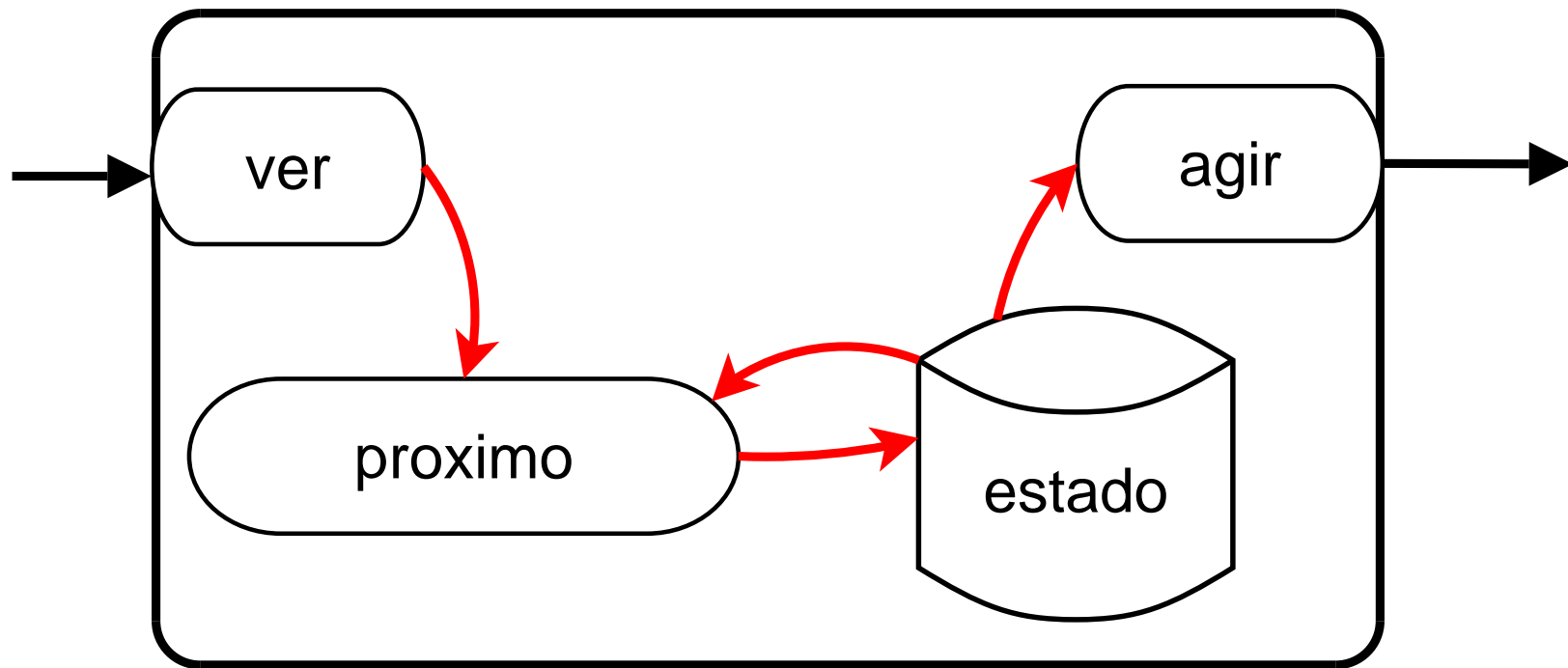
Arquiteturas para construção de agentes

- Existem várias formas de implementar um sistema que apresente os **comportamentos** desejados para um agente.
- Arquiteturas são **padrões** de funcionamento (como ligar as percepções do agente a suas ações).
 - ★ Padrões simplificam a construção de um software por já terem sido avaliados.
 - ★ Permitem um nível de abstração maior (Factories na OO, BDI para agentes).
 - ★ Cada arquitetura é adequada para um tipo de problema.
 - ★ Existem n arquiteturas propostas.

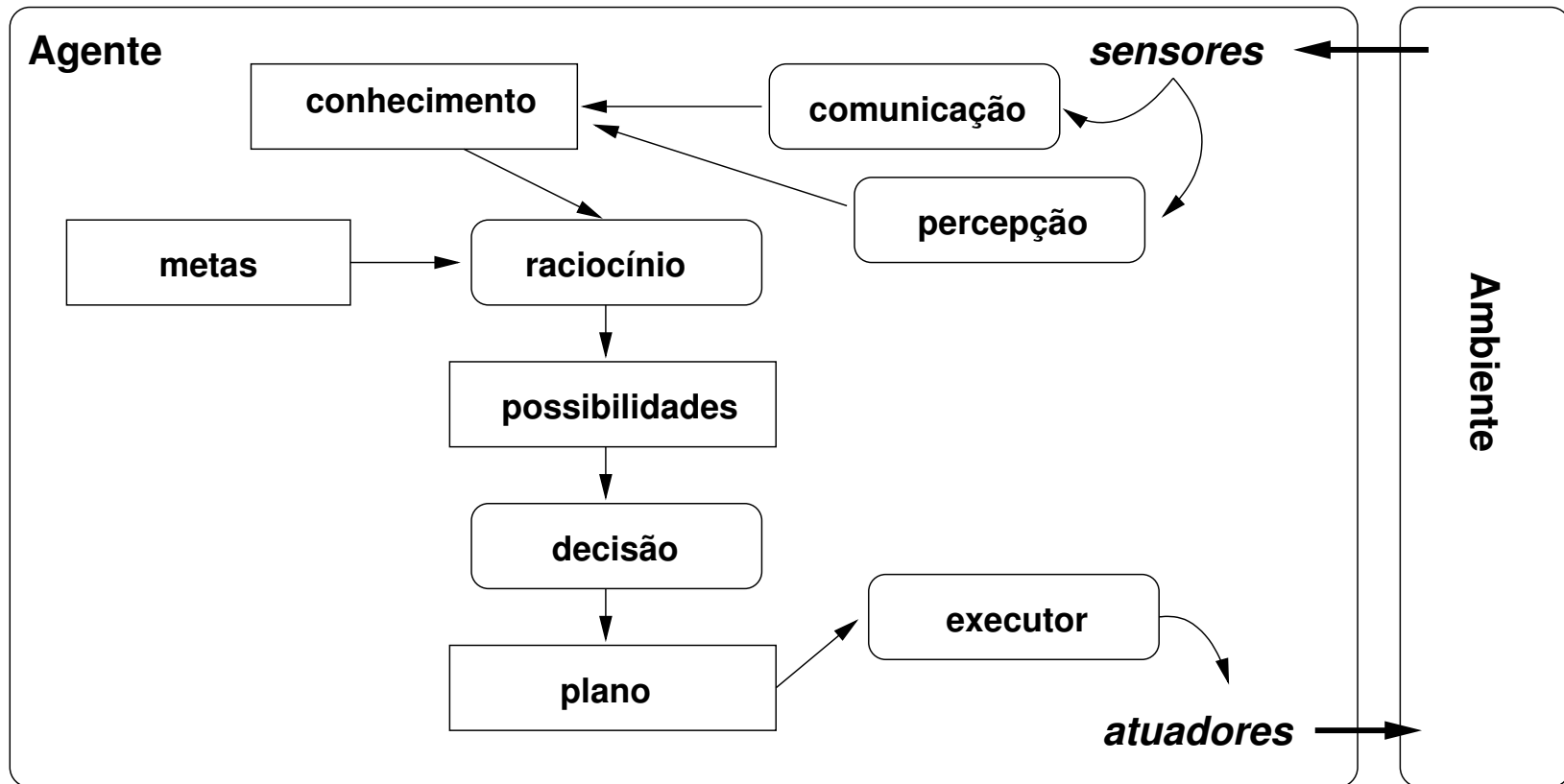
Arquitetura para agentes puramente **reativos**



Arquitetura para agentes reativos com **estado**



Arquitetura para agentes **cognitivos**



(Demazeau, 1990)

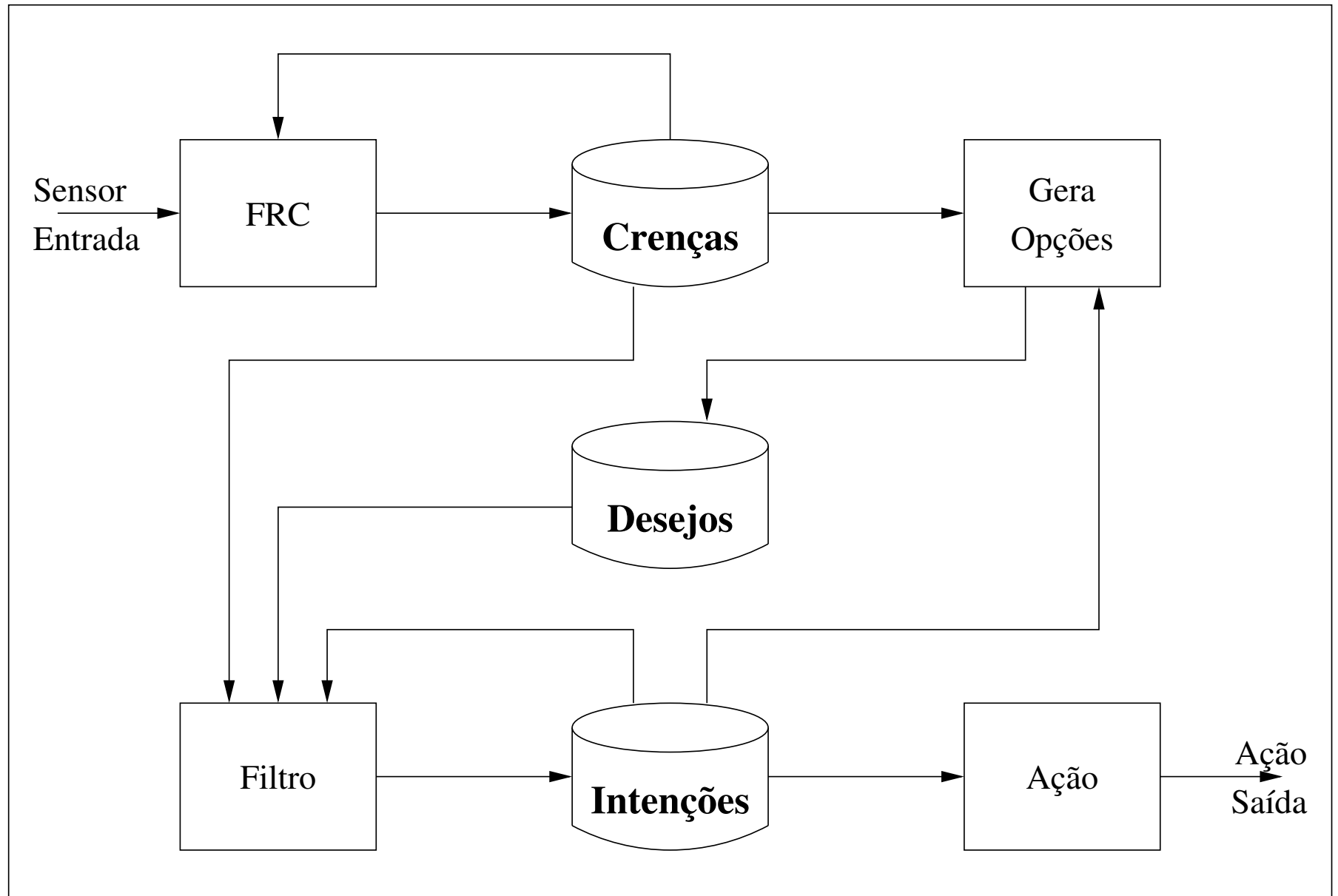
Arquiteturas híbridas

- Em arquiteturas reativas é difícil de implementar comportamento direcionado a uma meta.
- Em arquiteturas cognitivas é difícil de implementar comportamento reativo.
- Possível solução:
 - ★ Arquiteturas híbridas
 - ★ Arquiteturas baseadas em **raciocínio prático**

Arquitetura **BDI**

Para agentes com **raciocínio prático**, uma possível arquitetura é a BDI (**B**elief, **D**esire, **I**ntentions).

- Fundamentação filosófica (Bratman)
- Tem várias implementações (IRMA, PRS, Jack, Jason, ...)



- **Crenças**: o que o agente sabe do mundo e sobre si mesmo
 - ★ Gerado pela função de revisão de crenças
- **Desejos**: o que o agente quer (pode ser contraditório)
 - ★ Estados do mundo que são desejados
 - ★ Consideram as crenças para verificar sua viabilidade
 - ★ Permitem a criação de um sub-conjunto chamado **objetivos** (desejos consistente)
- **Intenções**: seqüência de ações que o agente se compromete a executar para atingir seus objetivos.
 - ★ Gerado por um processo de **deliberação** (que ações devem ser executadas, considerando as crenças, desejos e intenções atuais).
 - ★ O agente pode possuir várias intenções simultâneas. O componente **ação** escolhe uma para executar baseado, por exemplo, em prioridades. (**reatividade**)

AgentSpeak(L)

Visão geral



A linguagem AgentSpeak(L)

- A linguagem é uma extensão natural e elegante de programação em lógica para a arquitetura de agentes BDI (Rao, 1996).
- Um agente AgentSpeak(L) corresponde à especificação de um conjunto de **crenças** e um conjunto de **planos**.

- Tipos de **eventos**:
 - ★ evento **externo**: gerados pelo ambiente
 - +concerto(A,L)
 - ★ evento **interno**: gerados pelo próprio agente
 - gosta(A)
- Tipos de **objetivos**:
 - ★ objetivos de **teste**: ?gosta(A)
 - ★ objetivos de **realização**: !reserva_tickets(A,L)
- **Ações básicas**: liga(L)

- **Planos**: evento ativador + contexto + seqüência de ações básicas ou subobjetivos

```
+concerto(A,L) : gosta(A)
    ← !reserva_tickets(A,L).

+!reserva_tickets(A,L) : ¬ocupado(phone)
    ← liga(L);
    ...;
    !escolhe_lugar(A,L).
```

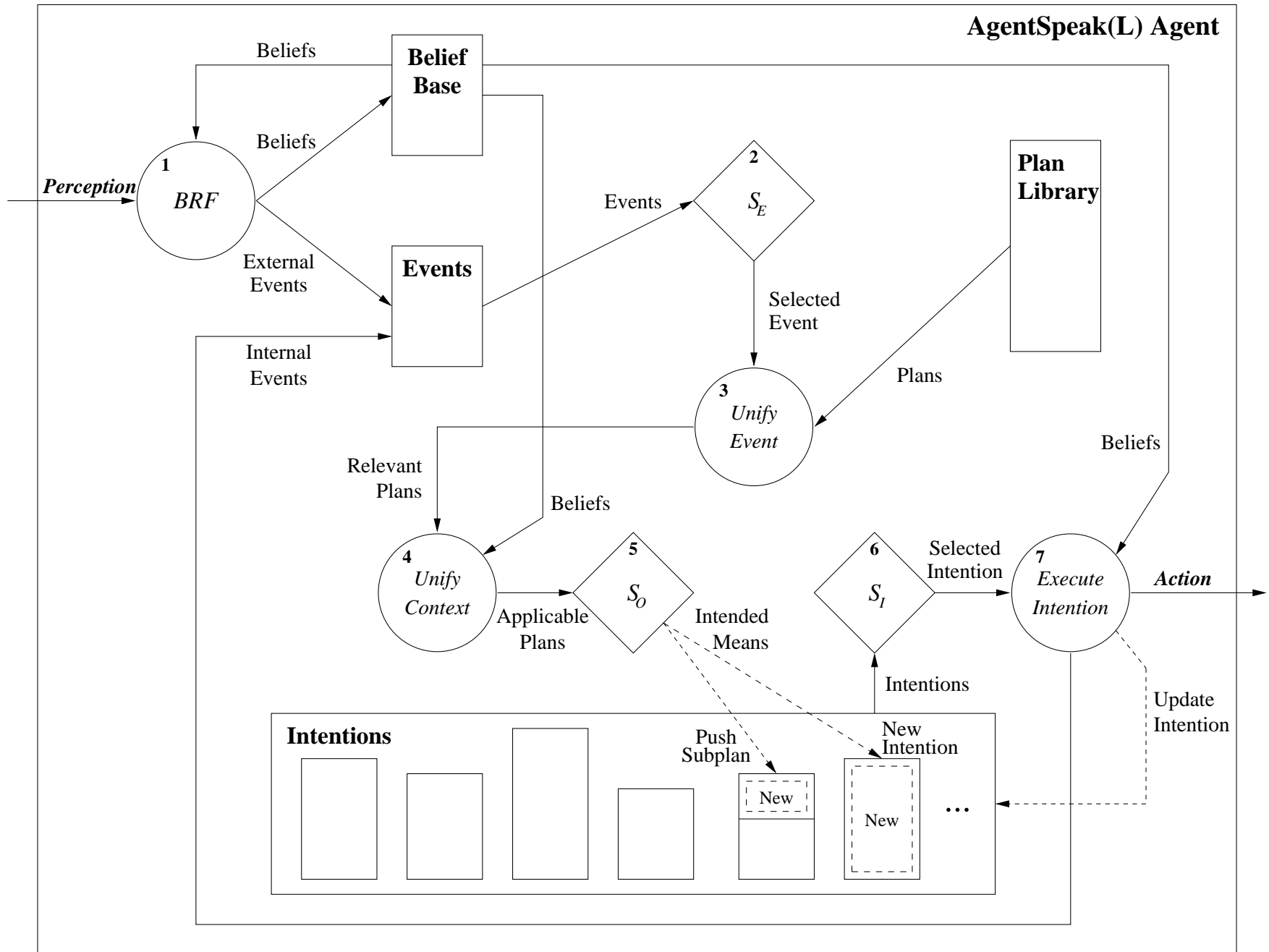
Sintaxe de um agente AgentSpeak(L)

$$\begin{array}{ll}
 ag & ::= \quad bs \ ps \\
 bs & ::= \quad b_1 \dots b_n \quad (n \geq 0) \\
 at & ::= \quad P(t_1, \dots, t_n) \quad (n \geq 0) \\
 ps & ::= \quad p_1 \dots p_n \quad (n \geq 1) \\
 p & ::= \quad te : ct \leftarrow h \\
 te & ::= \quad +at \quad | \quad -at \quad | \quad +g \quad \quad | \quad -g \\
 ct & ::= \quad at \quad \quad | \quad \neg at \quad | \quad ct \wedge ct \quad | \quad \top \\
 h & ::= \quad a \quad \quad | \quad g \quad \quad | \quad u \quad \quad \quad | \quad h; h \\
 a & ::= \quad A(t_1, \dots, t_n) \quad (n \geq 0) \\
 g & ::= \quad !at \quad \quad | \quad ?at \\
 u & ::= \quad +at \quad \quad | \quad -at
 \end{array}$$

Arquitetura para AgentSpeak(L)

Uma arquitetura para o funcionamento de um agente segundo o AgentSpeak(L) deve considerar três funções de personalização

- Seleção de eventos
- Seleção de planos
- Seleção de intenções



Exemplo: Robôs Mineradores

- **Robô 1**

Crenças

```
pos(r2,2,2).
```

```
procurando(slots).
```

Planos

```
+pos(r1,X,Y) : procurando(slots) & not( lixo(r1) )  
  <- move_proximo(slot).
```

```
+lixo(r1) : procurando(slots)  
  <- !parar(check);  
      !entregar(garb,r2);  
      !continuar(check).
```

```
+!parar(check) : true
  <- ?pos(r1,X1,Y1);
  +pos(back,X1,Y1);
  -procurando(slots).

+!entregar(S,L) : true
  <- !verifica_pegou(S);
  !vai(L);
  solta(S).

+!verifica_pegou(S) : lixo(r1)
  <- pega(garb);
  !verifica_pegou(S).

+!verifica_pegou(S) : true <- true.
```

```
+!continuar(check) : true
  <- !vai(back);
    -pos(back,X,Y);
    +procurando(slots);
    move_proximo(slot).
```

```
+!vai(L) : pos(L,X,Y) & pos(r1,X,Y)
  <- true.
```

```
+!vai(L) : true
  <- ?pos(L,X,Y);
    move_pata(X,Y);
    !vai(L).
```

- **Robô 2**

Planos

```
+lixo(r2) : true  
  <- queima(garb).
```

Exemplo: **Leilão**

- **Agente 1**

Planos

```
+leilao(N) : true  
  <- faz_oferta(N,6).
```

- **Agente 2**

Crenças

```
myself(ag2). oferta(ag2,4). aliado(ag3).
```

Planos

```
+leilao(N) : myself(I) & aliado(A) & not( alianca(A,I) )  
    <- ?oferta(I,B);  
        faz_oferta(N,B).
```

```
+leilao(N) : alianca(A,I)  
    <- faz_oferta(N,0).
```

```
+alianca(A,I) : myself(I) & aliado(A)  
    <- ?oferta(I,B);  
        .send(A,tell,oferta(I,B));  
        .send(A,tell,alianca(A,I)).
```

- **Agente 3**

Crenças

```
myself(ag3).      oferta(ag3,3).  
aliado(ag2).      limite_tentativas(3).
```

Planos

```
+leilao(N) : limite_tentativas(T) & .gte(T,N)  
    <- !oferta_normal(N).
```

```
+leilao(N) : myself(I) & ganhador(I)  
    & aliado(A) & not(alianca(I,A))  
    <- !oferta_normal(N).
```

```
+!oferta_normal(N) : true  
    <- ?oferta(I,B);  
    faz_oferta(N,B).
```

```
+leilao(N) : myself(I) & not(ganhador(I))
           & aliado(A) & not(alianca(I,A))
  <- !propoe_alianca(I,A);
     !oferta_normal(N).
```

```
+leilao(N) : alianca(I,A)
  <- ?oferta(I,B);
     ?oferta(A,C);
     .plus(B,C,D);
     faz_oferta(N,D).
```

```
+!propoe_alianca(I,A) : true
  <- .send(A,tell,alianca(I,A)).
```

Jason

Principais características do Jason

- Implementação de um **interpretador** de AgentSpeak(L)
- Feito em **Java**
- Suporte para **comunicação** entre agentes distribuídos utilizando atos de fala
- Suporte para desenvolvimento de **ambientes** para os agentes
- Possui uma biblioteca extensível de **ações internas**.

Arquivo de **configuração** do SMA

```
MAS leilao {  
  
    architecture: Saci  
  
    environment: leilaoEnv  
  
    agents: ag1; ag2; ag3;  
  
}
```

- Opções de **arquitetura**: Centralised ou Saci
- Pode-se dizer em que **máquinas** os agentes e o ambiente irão executar

```
agents:
```

```
ag1 at host1.acme.br;
```

- Pode-se dizer o nome do arquivo onde está o **fonte** do agente

```
agents:
```

```
ag1 file1;
```

- Pode-se indicar a **quantidade** de agentes que deve ser criado

```
agents:
```

```
ag1 #10;
```

- Pode-se definir uma **arquitetura** “casca” específica para o agente
 - ★ Como o agente faz percepção, recebe mensagens, revisa crenças e age

Exemplo no arquivo de configuração:

```
agents: ag1 agentArchitecture MyArch;
```

Exemplo de classe:

```
import jason.*;
public class MyAgArch extends CentralisedAgArch {

    public void perceive() {
        System.out.println("Getting percepts!");
        super.perceive();
    }
}
```

- Pode-se personalizar as **funções de seleção** da arquitetura
 - ★ Função de seleção de eventos, planos e intenções

Exemplo no arquivo de configuração:

```
agents: ag1 agentClass MyAgClass;
```

Exemplo de classe:

```
import jason.*;
import java.util.*;
public class MyAgClass extends Agent {

    public Event selectEvent(List evList) {
        System.out.println("Selecting an event from "+evList);
        return((Event)evList.remove(0));
    }
}
```

Exemplo de classe ambiente

```
import java.util.*;
import jason.*;
public class leilaoEnv extends Environment {

    public leilaoEnv() {
        getPercepts().add(.....);
    }

    public boolean executeAction(String ag, Term action) {
        if (action.hasActionSymb("faz_oferta")) {
            Integer x = new Integer(action.parameter(2).toString());
            oferta.put(ag,x);
        }

        ... // verify winner
        getPercepts().add(winner);
        return true;
    } }
}
```

Resumo

- Arquiteturas
- BDI
- Agentes
 - ★ Crenças
 - ★ Planos
- **Jason**

Interação

Contexto



(Demazeau, 95)

O que uma ferramenta para interação deve ter:

- Envio e recebimento de mensagens utilizando uma linguagem de comunicação
- Uso de protocolos de comunicação especificados pelo usuário
- Acompanhamento do funcionamento do sistema
- Facilidades: páginas amarelas, execução remota, interface, ...

Interação

- Teorias (atos de fala)
- Arquiteturas (KQML, FIPA ACL)
- Linguagens (FIPA-OS, JADE, SACI)

Exemplo de comunicação com **KQML**

KQML (Knowledge Query and Manipulation Language) é uma especificação de linguagem de comunicação entre agentes (Finin, 1997).

- Qualquer linguagem pode ser usada para escrever o conteúdo da mensagem
- A informação necessária para a interpretação da mensagem está na própria mensagem
- Os agentes podem ignorar o mecanismos de transporte (TCP/IP, RMI, IIOP, ...)
- O formato é simples, fácil de ler a verificar

Exemplo de mensagem KQML:

(*ask-one*

:language	<i>SQL</i>	}	nível de mensagem
:ontology	<i>SOSTore</i>		
:sender	<i>jomi</i>	}	nível de comunicação
:receiver	<i>ricardo</i>		
:reply-with	<i>id1</i>		
:content	<i>“select price from stocktable where ent = Conectiva”</i>	}	nível de conteúdo

```
(tell
  :language      prolog
  :ontology      SOSStore
  :receiver      jomi
  :sender        ricardo
  :in-reply-to   id1
  :reply-with    id45
  :content       "[price(10.0)]" )
```

Embora a KQML tenha um conjunto pré-definido de performativas e palavras-chave, este conjunto não é nem mínimo nem fechado. Contudo, os agentes que usam uma das performativas reservadas devem usá-las da forma padrão.

Saci

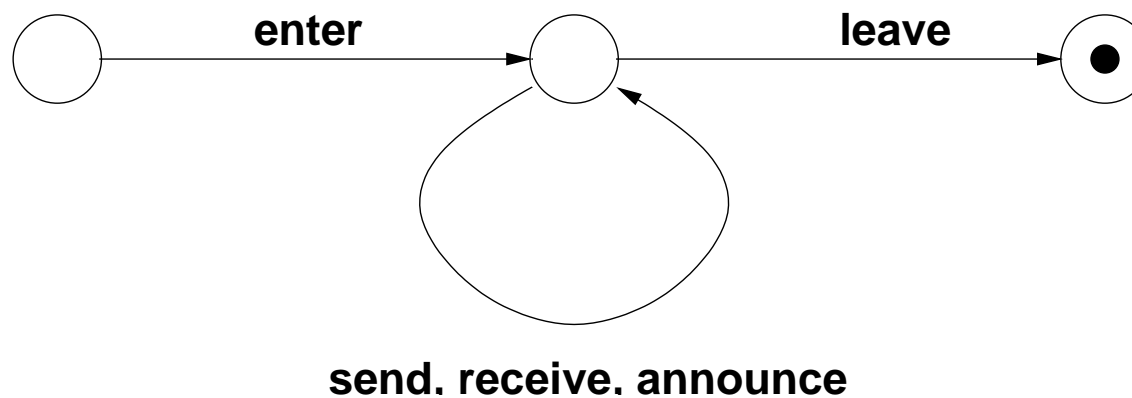
Ferramenta que torna a programação da **comunicação** entre agentes distribuídos mais fácil, em conformidade com um padrão, rápida e robusta.

- Possui uma API para compor (parser), enviar (as/sincronamente) e receber (mail box) mensagens KQML
- Um conjunto de facilidades:
 - ★ páginas brancas (nomes e localização dos agentes)
 - ★ páginas amarelas (serviços disponibilizados pelos agentes)
 - ★ controle e execução remota de agentes
 - ★ monitoramento da sociedade (eventos sociais podem ser obtidos)
 - ★ mobilidade de agentes

Agentes Saci

Para o Saci, os **agentes** possuem as seguintes propriedades:

- estão agrupados em sociedades
- possuem uma identificação única
- interagem com os demais agentes utilizando uma linguagem comum
- oferecem serviços aos demais agentes da sua sociedade
- tem o seguinte ciclo de vida:



Sociedade Saci

A estrutura de uma **sociedade** é especificada pela tupla

$$Soc = \langle \mathcal{A}, \mathcal{S}, l, \delta \rangle$$

tal que

$\mathcal{A} = \{ \alpha \mid \alpha \text{ é uma identificação de agente} \},$

$\mathcal{S} = \{ \sigma \mid \sigma \text{ é uma habilidade disponível} \},$

l é a linguagem de comunicação da sociedade

$\delta : \mathcal{A} \rightarrow \mathbb{P}(\mathcal{S})$ mapeamento das habilidades dos agentes, tal que

$\delta(\alpha) = \{ \sigma \mid \sigma \text{ é uma habilidade de } \alpha \}.$

por exemplo:

$$\begin{aligned} \text{Iti} = & \langle \{ \text{Jomi, Jaime, Julio, Jose} \}, \\ & \{ \text{Java, C, Prolog, Teach} \}, \text{Portuguese}, \\ & \{ \text{Jomi} \mapsto \{ \text{Java} \}, \text{Jaime} \mapsto \{ \text{C, Teach} \}, \text{Julio} \mapsto \{ \text{Java} \} \} \rangle \end{aligned}$$

Eventos Sociais

- Um agente **entra** na sociedade

$$\langle \mathcal{A}, \mathcal{S}, l, \delta \rangle_i \Rightarrow \langle \mathcal{A}', \mathcal{S}, l, \delta \rangle_{i+1} \mid \mathcal{A}' = \mathcal{A} \cup \{\alpha\}$$

- Um agente **anuncia** uma habilidade

$$\langle \mathcal{A}, \mathcal{S}, l, \delta \rangle_i \Rightarrow \langle \mathcal{A}, \mathcal{S}', l, \delta' \rangle_{i+1} \mid \mathcal{S}' = \mathcal{S} \cup \{\sigma\}$$

$$\delta'(x) = \begin{cases} \delta(x) & \text{if } x \neq \alpha \\ \delta(x) \cup \{\sigma\} & \text{otherwise} \end{cases}$$

- Os agentes **enviam/recebem** mensagens

- Um agente **deixa** a sociedade

$$\langle \mathcal{A}, \mathcal{S}, l, \delta \rangle_i \Rightarrow \langle \mathcal{A}', \mathcal{S}', l, \delta' \rangle_{i+1} \mid \mathcal{A}' = \mathcal{A} - \{\alpha\}$$

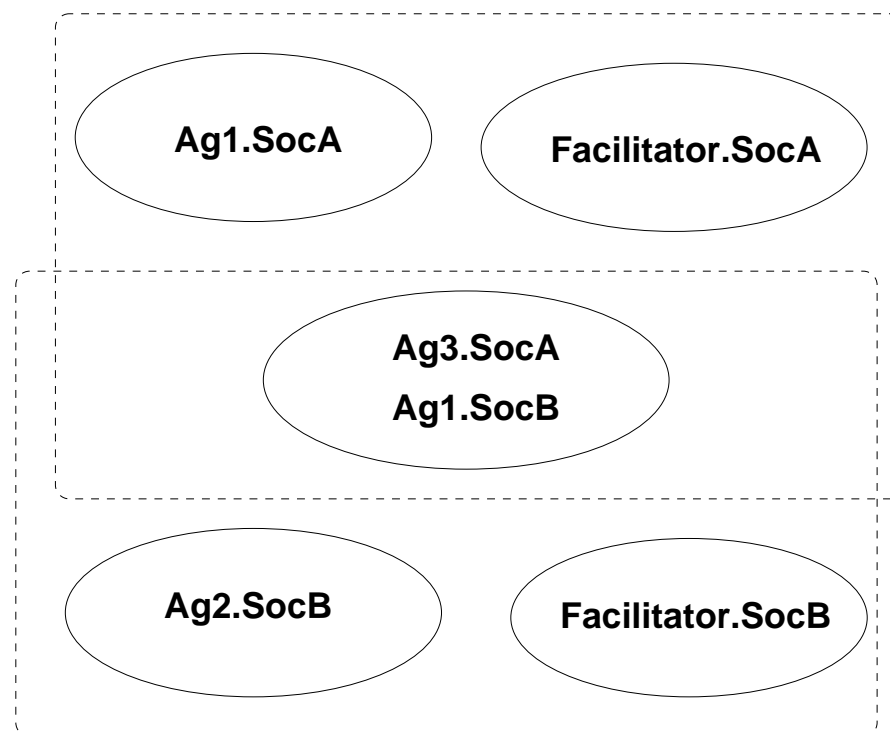
$$\delta'(x) = \begin{cases} \delta(x) & \text{if } x \in \mathcal{A}' \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathcal{S}' = \{\sigma \mid \sigma \in \delta'(x)\}$$

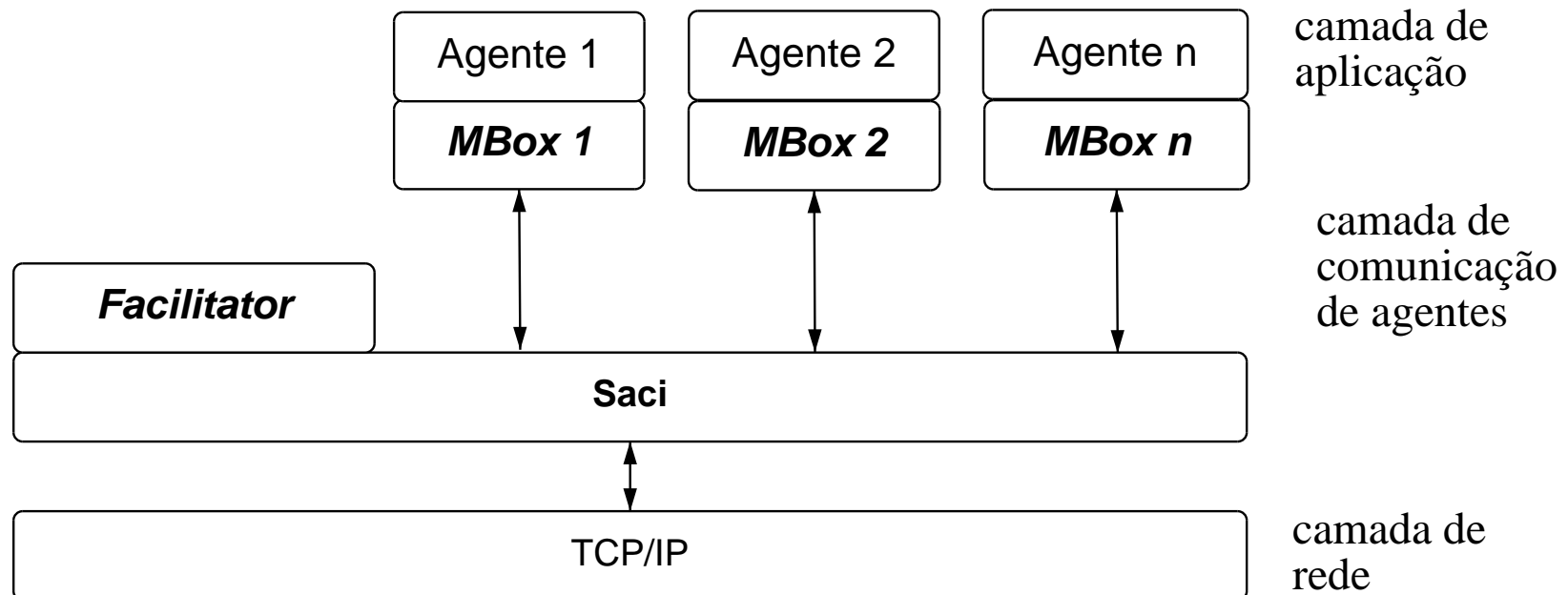
Entrada e saída da sociedade

Cada sociedade possui um **agente facilitador** que mantém sua estrutura no decorrer da sua história (sequência de eventos sociais).

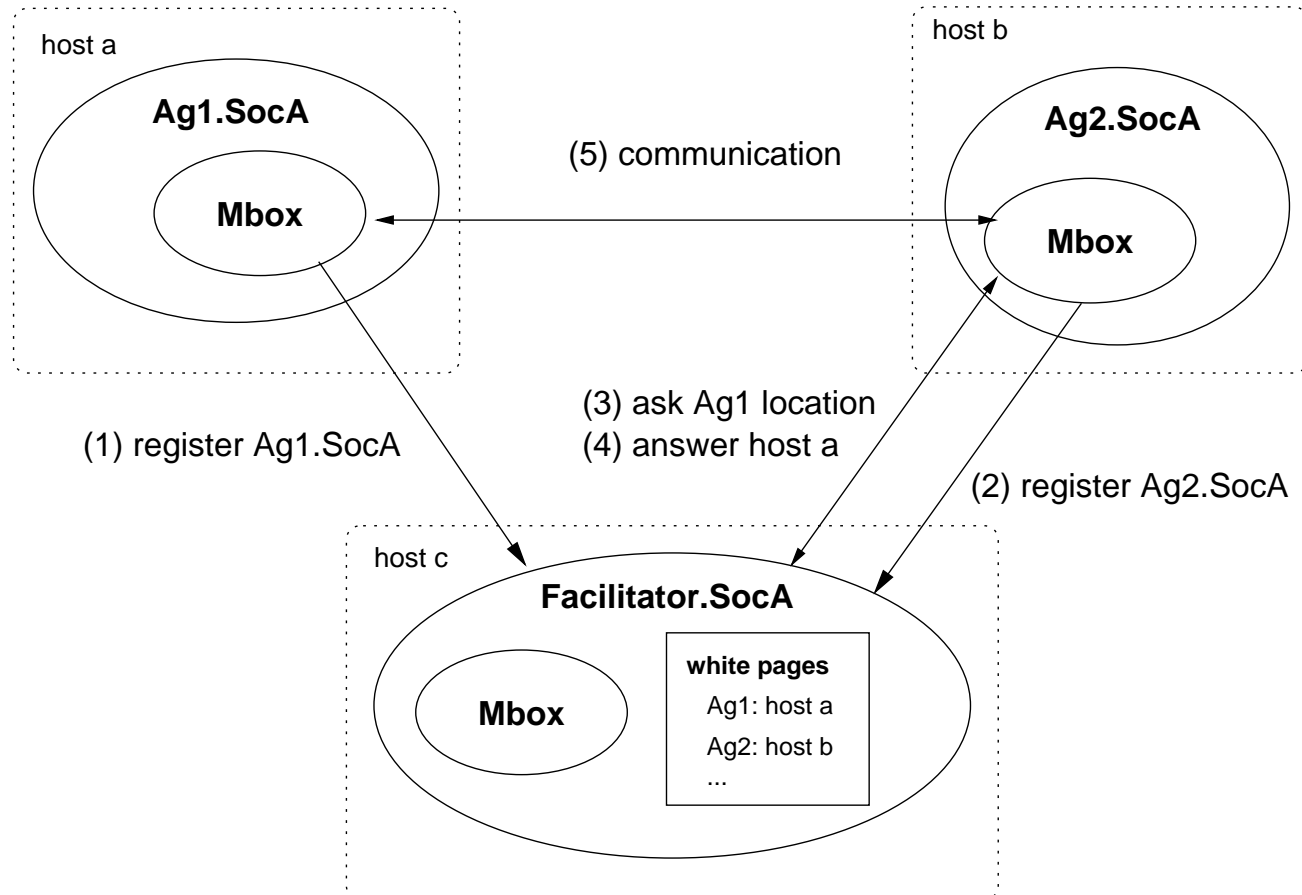
- Para entrar em uma sociedade um agente tem que registrar seu nome no facilitador e, antes de sair, deve avisar o facilitador.
- Os serviços que um agente deseja disponibilizar à sociedade devem ser anunciados ao facilitador.



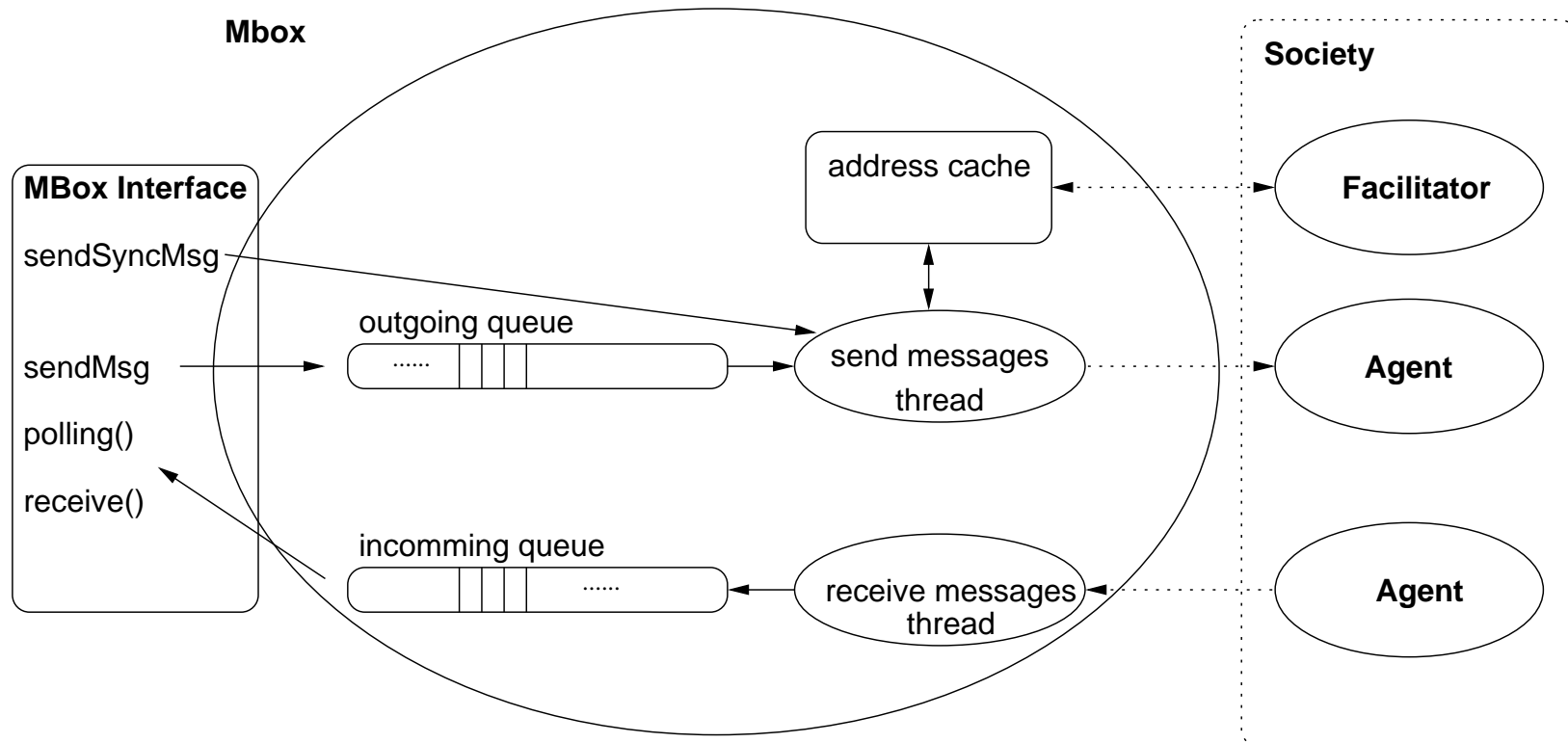
Arquitetura de comunicação



Envio e recebimento de mensagens



Arquitetura do MBox



API do MBox para envio e recebimento de mensagens

- `sendMsg(Message)`
- `sendSyncMsg(Message)`
- `broadcast(Message)`
- `receive()`
- `receive(Pattern)`
- `polling([timeout])`
- `polling(Pattern, timeout)`
- `getMessages(Pattern [, quantity, timeout, remove])`

API do MBox para protocolos KQML

- `ask(Message [, timeout])`
- `forward(Message)`

Exemplo: agente receptor

```
1 import saci.*;
2 class Receptor {
3     public static void main(String[] args) throws Exception {
4         MBoxSAg mbox = new MBoxSAg("receptor");
5         mbox.init();
6         while (true) {
7             Message m = mbox.polling();
8             if (m != null) {
9                 System.out.println("recebi "+m);
10            }
11        }
12    }
13 }
```

Exemplo: agente emissor

```
1 import saci.*;
2 class Emissor {
3     public static void main(String[] args) throws Exception {
4         MBoxSAg mbox = new MBoxSAg("emissor");
5         mbox.init();
6         mbox.sendMsg(new
6             Message("(tell :receiver receptor :content oi)"));
7     }
8 }
```

Exemplo de comunicação com ask

```
1 import saci.*;
2 class Emissor {
3     public static void main(String[] args) throws Exception {
4         MBoxSAg mbox = new MBoxSAg("emissor");
5         mbox.init();
6         Message resposta = mbox.ask(new
7             Message("(ask :receiver receptor :content oi)"));
8         System.out.println("resposta é "+resposta);
9     }
10 }
```

(Receptor)

```
4     MBoxSag mbox = new MBoxSag("receptor");
5     mbox.init();
6     System.out.println("Meu nome é "+mbox.getName());
7     Message pattern = new Message();
8     pattern.put("content" , "oi");
9     while (true) {
10        Message m = mbox.polling(pattern);
11        if (m != null) {
12            System.out.println("recebi "+m);
13            Message resposta = new Message("(tell :content bemVindo)");
14            resposta.put("receiver", m.get("sender"));
15            resposta.put("in-reply-to", m.get("reply-with"));
16            mbox.sendMsg(resposta);
17            System.out.println("respondi "+resposta);
18            ...
```

Manipuladores de mensagens

- Um manipulador de mensagem é um objeto que é “avisado” quando chega uma mensagem KQML de certo tipo.
- Cada mbox pode ter vários manipuladores, inclusive para o mesmo tipo de mensagem.
- O manipulador considera como “filtro” os seguintes valores:
 - ★ o conteúdo da mensagem,
 - ★ a linguagem do conteúdo,
 - ★ a ontologia e
 - ★ a performativa

Exemplo:

```
8     mbox.addMessageHandler("oi", "ask", null, "apresentacao",
9         new MessageHandler() {
10             public boolean processMessage(Message m) {
11                 System.out.println("recebi "+m);
12                 Message resposta =
13                     new Message("(tell :content bemVindo)");
14                 resposta.put("receiver", m.get("sender"));
15                 resposta.put("in-reply-to", m.get("reply-with"));
16                 mbox.sendMsg(resposta);
17                 System.out.println("respondi "+resposta);
18                 return true; // do not add m in mbox
19             }
20         }
21     );
```

A classe **Agent**

- A classe `agent` permite utilizar alguns serviços de controle sobre o ciclo de vida do agente.
 - ★ Criar os agentes via interface do `saci`
 - ★ Matá-los
 - ★ Movê-los
- O desenvolvedor pode sobre-escrever os seguintes métodos da classe
 - ★ `initAg`: método que é chamado uma única vez quando o agente é criado.
 - ★ `run`: método que é chamado para o agente iniciar sua execução.
 - ★ `stopAg`: método que é chamado para o agente terminar sua execução.

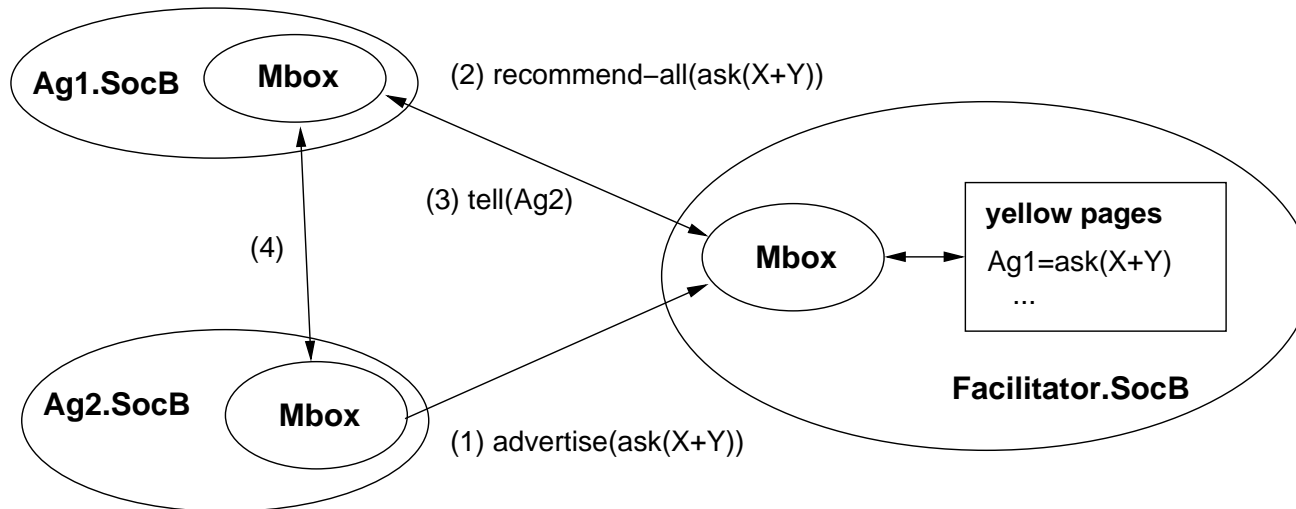
Exemplo:

```
1 import saci.*;
2
3 class Receptor extends Agent {
4
5     public static void main(String[] args) {
6         Receptor r = new Receptor();
7         if (r.enterSoc("receptor")) {
8             r.initAg(null);
9             r.run();
10        }
11    }
12
```

```
13  public void initAg(String[] a) {
14      try {
15          System.out.println("Meu nome é "+mbox.getName());
16
17          mbox.addMessageHandler("oi", "ask", null, "apresentacao",
18              new MessageHandler() {
19                  public boolean processMessage(Message m) {
20                      Message resposta =
21                          new Message("(tell :content bemVindo)");
22                      resposta.put("receiver", m.get("sender"));
23                      resposta.put("in-reply-to", m.get("reply-with"));
24                      try {
25                          mbox.sendMsg(resposta);
26                      } catch (Exception e) { System.err.println("Erro msg");}
27                      System.out.println("respondi "+resposta);
28                      return true; // do not m add in mbox
```

```
29         }
30     }
31 );
32 } catch (Exception e) { System.err.println("Erro "+e); }
33 }
34
35 public void run() {
36     System.out.println("rodando....");
37 }
38
39 public void stopAg() {
40     System.out.println("terminando....");
41 }
42
43 }
```

Páginas Amarelas: **Anúncio** de habilidades



(advertise :receiver	Facilitator			
:sender	Ag2			
:language	KQML			
:ontology	yp			
:content	(ask-one :receiver	Ag2	(tell :receiver	Ag1
	:language	alg	:sender	Facilitator
	:ontology	math	:in-reply-to	id1
	:content	"X + Y"))	:language	KQML
			:ontology	yp
			:content	(Ag2))

API do MBox para uso de **YP**

- boolean **advertise**(*performative, language, ontology, content*)
- String **recommendOne**(*performative, language, ontology, content*)
- List **recommendAll**(*performative, language, ontology, content*)
- Message **brokerOne**(*message [, timeout]*)

API da classe agent para mobilidade

- Callbacks

- ★ `onMoving(host)`: este método do agente é chamado pelo saci antes do agente deixar um host.
- ★ `onMoved()`: este método do agente é chamado pelo saci assim que o agente chega em um host e antes de iniciar sua execução (método `run`).

O desenvolvedor pode sobre-escrever estes dois métodos para realizar alguma operação antes de migrar.

- `move(host)`: move o agente para um host (o saci deve estar rodando em *host*)

Os seguintes passos são executados na migração de um agente:

- ★ o método `onMoving` é chamado
- ★ copia o **estado** do agente para o host remoto (inclusive o mailbox)
- ★ o método `onMoved` é chamado
- ★ inicia a thread/processo do agente no host remoto
- ★ termina a thread/processo do agente no host local

Um agente móvel

```
import saci.*;  
public class SampleMoveAg extends Agent {  
    boolean go = true;  
    public static void main(String[] args) {  
        SampleMoveAg a = new SampleMoveAg();  
        if (a.enterSoc(" SampleMoveAg" )) {  
            a.run();  
        }  
    }  
    public void run() {  
        if (go) {  
            go = false;  
            move("dijon.pcs.usp.br"); // and do something there  
        } else {  
            move("annecy.pcs.usp.br"); // and show the results  
        }  
    }  
}
```

Resumo

- Linguagem de comunicação (KQML, FIPA-ACL)
- Atos de fala (tell, ask, ...)
- Serviços (WP, YP, mobilidade, ...)
- SACI

Conclusões

- SMA e programação orientada a agentes
 - ★ Principais aspectos: Agentes, Interação, Organização e Ambiente
 - ★ Oferecem uma **abordagem** interessante para vários problemas
 - ★ Não é uma “solução geral”
 - ★ É mais um item na nossa “caixa de ferramentas”
 - ★ Está no início
- Existem vários trabalhos para ser desenvolvido na área
 - ★ Ferramentas
 - ★ Avaliações

Bibliografia básica

- [1] Jeffrey S. Rosenschein, Tuomas Sandholm, Wooldridge Michael, and Makoto Yokoo, editors. *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2003)*. International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2003), ACM Press, 2003.
- [2] Gerhard Weiß, editor. *Multiagent Systems: A modern approach to distributed artificial intelligence*. MIT Press, London, 1999.
- [3] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley and Sons, 2002.