Introdução ao Desenvolvimento de Sistemas Multiagentes com *Jason*

Jomi Fred Hübner[1], Rafael Heitor Bordini[2], Renata Vieira[3]

[1] Grupo de Inteligência Artificial
Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB)
Campus IV, rua Braz Wanka, 238, Vila Nova
89035-160, Blumenau, SC

[2]Department of Computer Science
University of Durham
Durham DH1 3LE, U.K.

[3]Centro de Ciências Exatas e Tecnológicas,
Centro de Ciências da Comunicação
Universidade do Vale do Rio dos Sinos (UNISINOS)
Av. Unisinos, 950 – 93022-000 São Leopoldo, RS

jomi@inf.furb.br, R.Bordini@durham.ac.uk
renata@exatas.unisinos.br

Resumo. *Este texto apresenta uma introdução à concepção de sistemas computacionais como sendo formado por vários agentes autônomos. O desenvolvimento destes agentes passa pelo estudo de arquiteturas que determinam seu funcionamento e de linguagem e ferramentas que viabilizam sua utilização. Nesta direção, é apresentada uma arquitetura baseada nas noções de crenças, desejos e intenções e uma linguagem de programação de agentes, chamada AgentSpeak, que segue estas noções. Alguns exemplos são desenvolvidos utilizando uma ferramenta que implementa a linguagem AgentSpeak.*

1. Introdução

Tanto a Ciência da Computação quanto a Inteligência Artificial (IA) têm buscado formas de conceber sistemas que se aproximam da realidade considerando, em geral, as visões que outras áreas do conhecimento têm da realidade. Assim surgiram a orientação a objetos (da Matemática), a representação de conhecimento e raciocínio (da Psicologia e da Lógica), as redes neurais (da Biologia), etc. De forma análoga, a área de Sistema Multi-Agentes (SMA) é influenciada pela Sociologia e pela Etologia e, portanto, tem vislumbrado uma concepção de sistema com propriedades que até então somente sociedades possuíam. O estudo de SMA, ao contrário dos paradigmas tradicionais da IA, têm como objeto de estudo a *coletividade* e não um único indivíduo. Desta forma, deixam de ter atenção as iniciativas de compreender e simular o comportamento humano isoladamente, seja mental (IA simbolista) ou neural (IA conexionista), passando o foco da atenção para a forma de interação entre as entidades que formam o sistema (chamadas de agentes) e sua organização.

Este paradigma é motivado pela observação de alguns sistemas naturais, nos quais se percebe o surgimento de um comportamento inteligente a partir da interação de seus elementos (Johnson, 2001). Por exemplo, apesar de uma colônia de formigas ser formada por seres simples, pode-se dizer que o formigueiro como um todo é um sistema complexo cujo comportamento é mais inteligente do que os das formigas que o formam; os neurônios são células simples, mas de sua interação e organização emerge um comportamento complexo e inteligente. Estes dois exemplos mostram que a coletividade possui características que não podem ser reduzidas aos componentes que a formam, mas que são essenciais para o comportamento bem adaptado que tais sistemas apresentam. Entre as características coletivas do SMA incluem-se a *interação* entre os agentes (linguagens e protocolos), o *ambiente* e a *organização*.

A partir desta motivação, a área de SMA estuda o comportamento de um grupo *organizado* de agentes *autônomos* que cooperam na resolução de problemas que estão além das capacidades de resolução de cada um individualmente. Duas propriedades, aparentemente contraditórias, são fundamentais para os SMA: a autonomia dos agentes e sua organização (Briot e Demazeau, 2002). O atributo autônomo significa aqui o fato de que um agente tem sua existência independente dos demais e mesmo do problema sendo solucionado (Weiß, 1999b, p. 548). No caso, trata-se de uma autonomia de existência. Para funcionar, um agente não precisa de outros agentes, mesmo que para alcançar seus objetivos ele eventualmente precisará da ajuda de outros.[1] Por outro lado, a organização estabelece restrições aos comportamentos dos agentes procurando estabelecer um comportamento grupal coeso. Muitas das propriedades desejadas nos SMA advém do equilíbrio destes dois opostos, portanto, compreender como estas duas propriedades interagem é uma questão importante (e interessante) no contexto dos SMA.

No processo de desenvolvimento de SMA duas abordagens são comuns na tentativa de conciliar autonomia e controle:

- *Top-down*: inicia-se definindo os aspectos coletivos, como organização e comunicação, que são refinados até a definição dos agentes. Esta é, normalmente, a ênfase dada pelas metodologias de Agent Oriented Software Engineering (AOSE) (Garijo et al., 2001; Iglesias et al., 1999a; Odell et al., 2000; Wooldridge et al., 1999).
- *Bottom-up*: inicia-se definindo os aspectos individuais, relacionados aos agentes, de tal forma que ocorra a emergência dos aspectos coletivos. A interação e a organização são definidos do ponto de vista dos agentes.

Em outras palavras, na abordagem *top-down*, o projetista olha o sistema como um todo e na abordagem *bottom-up* ele vê o sistema como se estivesse "dentro dos agentes".

Este texto apresenta uma introdução ao desenvolvimento de SMA adotando a segunda abordagem e está organizado da seguinte forma (a figura 1 ilustra a relação entre os tópicos abordados neste texto): as próximas duas seções fazem uma introdução geral à área de SMA. Na seção 4, uma arquitetura para determinar o funcionamento dos agentes é apresentada – a arquitetura BDI. Uma linguagem que segue a arquitetura BDI é descrita na seção 5 – linguagem AgentSpeak. Por fim, uma ferramenta que implementa a linguagem AgentSpeak é utilizada na construção de alguns exemplos de SMA na seção 6. Algumas conclusões e problemas ainda não resolvidos são apresentados na seção 7.

[1] Existem outras formas de autonomia, Castelfranchi (1990), por exemplo, define um agente autônomo como aquele que decide quais objetivos adotar.

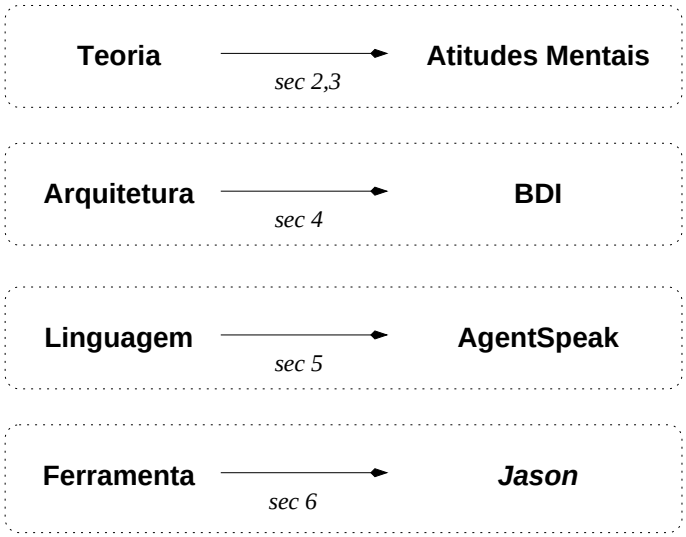


Figura 1: Visão geral do texto

2. Sistemas Multiagentes

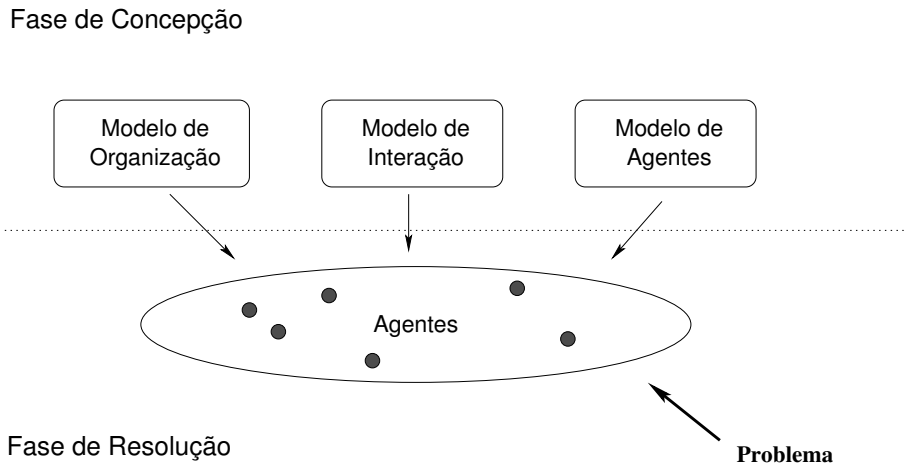
Tomando um ponto de vista *bottom-up* no desenvolvimento de sistemas computacionais, o objetivo da área de SMA passa a ser a definição de modelos genéricos de agentes, interações e organizações que possam ser instanciados dinamicamente dado um problema (estas etapas são brevemente descritas na figura 2)[2]. Dado este ideal de metodologia de desenvolvimento de sistema, esta abordagem apresenta as seguintes características (Alvares e Sichman, 1997):

- os agentes são concebidos independentemente de um problema particular;
- a interação entre os agentes não é projetada anteriormente, busca-se definir protocolos que possam ser utilizados em situações genéricas;
- a decomposição de tarefas para solucionar um dado problema pode ser feita pelos próprios agentes;
- não existe um controle centralizado da resolução do problema.

Das quais decorrem algumas vantagens:

- Viabilizam sistemas *adaptativos* e *evolutivos*: o SMA tem capacidade de adaptação a novas situações, tanto pela eliminação e/ou inclusão de novos agentes ao sistema quanto pela mudança da sua organização.
- É uma *metáfora natural* para a modelagem de sistemas complexos e distribuídos: em muitas situações o conhecimento está distribuído, o controle é distribuído, os recursos estão distribuídos. E, quanto à modelagem do sistema, a decomposição de um problema e a atribuição dos sub-problemas a agentes permite um alto nível de abstração e independência entre as partes do sistema (Jennings e Wooldridge, 1998).
- Toma proveito de ambientes *heterogêneos* e *distribuídos*: agentes com arquiteturas diferentes, que funcionam em plataformas diferentes, distribuídas em uma rede de computadores, podem cooperar na resolução de problemas. Isto permite o uso das potencialidades particulares de cada arquitetura e, pela distribuição, melhora o desempenho do sistema.
- Permite conceber *sistemas abertos*: os agentes podem migrar entre sociedades, isto é, agentes podem sair e entrar em sociedades, mesmo que desenvolvidos por projetistas e objetivos distintos. Tal abertura permite a evolução

[2]A definição mais detalhada de SMA, seus problemas e aplicações podem ser encontradas nas seguintes referências Alvares e Sichman (1997); Bordini et al. (2001); Demazeau e Müller (1990); Ferber (1999a); Jennings e Wooldridge (1998); Weiß (1999b); Wooldridge (2002).



De forma geral, o ciclo de vida de um SMA passa por duas etapas: concepção e resolução (Sichman, 1995). Na concepção são definidos modelos de propósito geral para os agentes, para suas interações e para suas formas de organização. Na resolução, um grupo de agentes adota estes modelos para resolver os problemas que lhe são apresentados. Diferentes tipos de problemas demandam dos agentes diferentes escolhas de modelos. A principal característica é a independência entre a concepção dos modelos e o problema, isto é, os modelos não são desenvolvidos para solucionar um problema particular. Por exemplo, os protocolos contratuais de Smith (1980) são um modelo de interação aplicável em vários tipos de problemas.

Figura 2: Desenvolvimento ideal de Sistemas Multiagentes.

e a adaptabilidade do sistema. Esta noção de sistema aberto (baseada, entre outros, em (Bordini, 1994, 1999; da Rocha Costa et al., 1994; Hübner, 1995)) difere da noção adotada, por exemplo, na área de engenharia de software, onde a principal característica de um sistema é aberto é seguir determinados padrões.

2.1. Aplicações de SMA

Não serão fornecidas aqui as referências para trabalhos específicos nas área de aplicação mencionadas abaixo. Em Jennings et al. (1998), Wooldridge (1998) pode-se encontrar algumas referências para trabalhos nestas áreas de aplicação. Uma apresentação mais completa de aplicações industriais de sistemas multiagentes é dada em Parunak (1999).

Algumas das principais áreas de aplicação que tem sido abordadas com o uso de sistemas multiagentes são:

Controle de Tráfego Aéreo: É uma aplicação reconhecidamente complexa. A aplicação de sistemas multiagentes mais comentada é um sistema para controle de tráfego aéreo que está sendo testado para uso no aeroporto de Sydney na Austrália. Este sistema foi implementado usando o PRS (veja seção 4). O PRS foi utilizado também para implementar outras importantes aplicações, como alguns sistemas para controle de espaçonaves e robôs.

Indústria: Modelar uma linha de produção através de agentes cooperantes é uma área de aplicação bastante grande.

Gerência de Negócios: Agentes podem ser empregados em sistemas de *workflow* (i.e., sistemas que visam garantir que as tarefas necessárias serão feitas pelas pessoas certas nas horas certas).

Interação Humano-Computador: Esta área utiliza agentes *credíveis*[3] e improvisacionais Bates (1994), Hayes-Roth et al. (1995) para melhorar a qualidade das interfaces de software.

Ambientes de Aprendizagem: Estes sistemas podem empregar agentes tanto com o mesmo objetivo mencionado acima como para a modelagem dos alunos em tutores inteligentes.

Entretenimento: Agentes podem ser utilizados para aumentar o realismo de personagens de jogos e sistemas para a indústria de entretenimento em geral.

Aplicações Distribuídas: Para domínios naturalmente distribuídos, a metáfora de *agente* é bastante adequada; telecomunicações, transportes e sistemas para a área de saúde são alguns exemplos.

Aplicações para a Internet: Esta é uma área muito comum para aplicações multiagentes. Existem várias classes de sistemas que se enquadram neste item, como sistemas para gerência de informação (em particular agentes conhecido como *personal digital assistants* ou assistentes digitais pessoais, que auxiliam na gerência da sobrecarga de informação recebida pelas pessoas via Internet), sistemas de comércio eletrônico e sistemas para busca de informações na Internet.

Simulação Social: Uma área de aplicação que também representa um desafio bastante grande para a área é a simulação social (Gilbert e Conte, 1995; Gilbert e Doran, 1994). O objetivo destas simulações multiagentes é auxiliar cientistas sociais em seus estudos. Para isto é preciso representar aspectos cognitivos e sociais de comunidades de pessoas, vindo ao encontro dos objetivos dos sistemas multiagentes. Este tipo de simulação pode auxiliar na compreensão de questões importantes para as ciências sociais, como o problema do elo micro-macro (Castelfranchi, 1997).

2.2. Sugestões de Leituras em SMA

Optou-se por apresentar aqui um texto resumido, provendo uma visão geral da área de SMA. Felizmente, devido à ampla divulgação que os SMA obtiveram no final da década de 90, existe uma vasta literatura disponível sobre o assunto, para aqueles interessados em aprofundar seus conhecimentos desta área.

Exemplo de artigos do tipo *survey* podem ser encontrados em (Green et al., 1997; Wooldridge e Jennings, 1995). Artigos introdutórios com muitas referências são, por exemplo, (Jennings et al., 1998; Sycara, 1998; Wooldridge, 1998). Existe também uma extensiva bibliografia da área de IAD em Wooldridge et al. (1996). Outro tipo importante de material bibliográfico para a área são livros com seleções de artigos previamente publicados; veja, por exemplo, (Bond e Gasser, 1988; Huhns e Singh, 1997). Para um estudo mais aprofundado da área recomenda-se livros editados com artigos de conferencias importantes, como por exemplo (Gasser e Huhns, 1989; Huhns, 1987) e os anais do *Workshop on Agent Theories Architectures and Languages* (ATAL) publicado pela Springer-Verlag em sua série *Lecture Notes in Computer Science*, e principalmente os anais da *International Conference on Multi-Agent Systems* (ICMAS) (Demazeau, 1998; Durfee, 1996, 2000; Lesser e Gasser, 1995). Recentemente, estas duas últimas conferências uniram-se no evento *International Joint Conference on Autonomous Agents and Multi-Agent Systems* (AAMAS) (Castelfranchi e Johnson, 2002; Rosenschein et al., 2003).

[3]Agentes credíveis são agentes capazes de provocar "suspenção de descrença" no fato de que eles não são seres reais, como personagens de desenhos animados. Isto pode ser obtido fazendo os agentes exibirem algo parecido com personalidades, por exemplo.

O primeiro livro-texto para a área foi lançado somente no final da década de 90 (Weiß, 1999a). Deste livro, pelo menos três capítulos são fortemente recomendados: o capítulo introdutório escrito por Wooldridge (1999), o capítulo que revisa os principais aspectos de métodos formais usados em IAD (Singh et al., 1999), e o capítulo sobre aplicações industriais de IAD (Parunak, 1999). Um segundo livro-texto na área é (Wooldridge, 2002).

Existem também *surveys* especializados, como por exemplo (Müller, 1999) sobre arquiteturas de agentes, e (Iglesias et al., 1999b) sobre métodos de engenharia de software orientados a agentes. Na área de *Simulação Social* e *Sociedades Artificiais*, as principais coleções de artigos são (Gilbert e Conte, 1995; Gilbert e Doran, 1994; Moss e Davidsson, 2001; Sichman et al., 1998); e uma coleção de artigos na área de simulações de instituições, organizações e grupos é encontrada em (Prietula et al., 1998).

Atualmente muitos recursos relacionados a agentes encontram-se disponíveis na Internet. Um repositório eletrônico destes recursos encontra-se no *web site* do AgentLink[4], URL `http://www.AgentLink.org/`. Um dos primeiros grandes *web sites* dedicados a agentes foi o "*AgentWeb*" da *University of Maryland Baltimore County*, que possui um repositório de materiais relacionados a agentes, e encontra-se na URL `http://agents.umbc.edu/`. Outro *web site* que possui uma lista extensiva de plataformas para desenvolvimento de sistemas multiagentes encontra-se na URL `http://www.agentbuilder.com/AgentTools/`. Uma bibliografia sobre agentes de software pode ser encontrada na URL `http://www.cs.helsinki.fi/~hhelin/agents/biblio.html`.

3. Agentes

Existem dois grandes tipos de arquitetura para os agentes de um SMA: os *reativos* (Ferber, 1999b) e os *cognitivos* (como os que serão abordados ao longo deste texto). SMA reativos seguem a idéia de que um comportamento inteligente em um sistema emerge da interação entre um grande número de agentes muito simples; a principal influência nesse tipo de trabalho vem da entomologia (estudo dos insetos). Nesses sistemas, os agentes não possuem uma representação explícita do estado do ambiente e dos outros agentes, nem da história de suas ações anteriores; eles têm comportamentos que podem ser descritos como autômatos finitos simples, e possuem um conjunto de regras que mapeiam percepções do ambiente diretamente em ações sobre este (os agentes agem sob um esquema estímulo-resposta). Já os SMA cognitivos em geral possuem tipicamente poucos agentes, dado que cada agente é um sistema sofisticado e computacionalmente complexo.

É preciso explicar logo de início que os pesquisadores da área não possuem uma definição de consenso para o que é um agente. Algumas definições ficaram bem conhecidas, como a de Wooldridge e Jennings (1995) e a de Franklin e Graesser (1997). Ao invés de citar essas definições que são as mais aceitas, preferiu-se aqui mencionar algumas características que são importantes (do ponto de vista dos autores) para o conceito de agente na concepção de SMA e que estão relacionadas com a linguagem AgentSpeak(L) apresentada a seguir. Note que, como uma área da inteligência artificial distribuída, trata-se aqui de sistemas que usam técnicas de inteligência artificial.[5] Portanto, um agente, nesta concepção, é um sistema compu-

[4]AgentLink é uma rede de excelência para computação baseada em agentes, financiada pela Comissão Européia.

[5]Devido ao sucesso e grande atenção dada a essa área na segunda metade da década de 90, o

tacional que é capaz de:

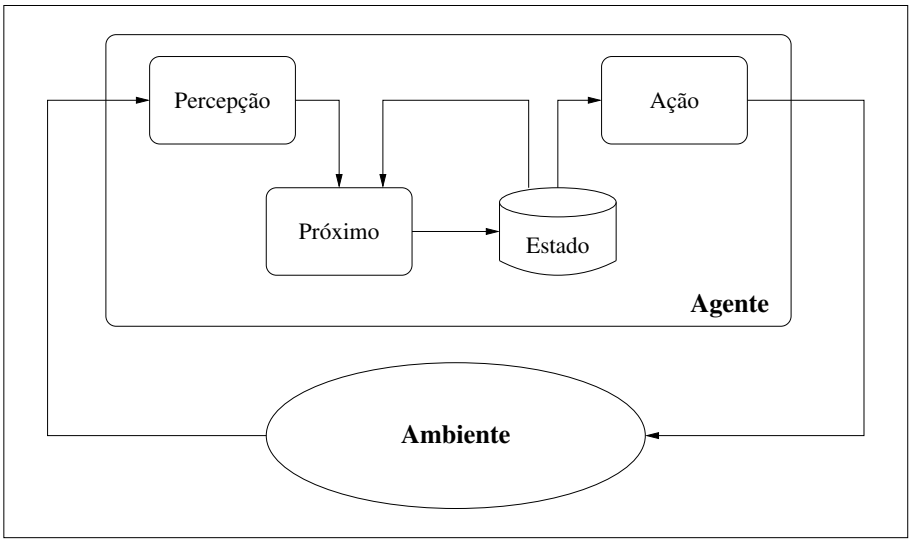


Figura 3: Modelo Geral de Agente (traduzido de Wooldridge (1999))

percepção: o agente é capaz de perceber alterações no ambiente (conforme ilustrado na figura 3);

ação: as alterações no ambiente são provenientes das ações que os agentes realizam constantemente no ambiente; um agente age sempre com o intuito de atingir seus objetivos (motivação), ou seja, com o intuito de transformar o ambiente de seu estado atual em um outro estado desejado pelo agente (veja *motivação* abaixo);

comunicação: umas das ações possíveis de um agente é comunicar-se com outros agentes da sociedade (i.e., que compartilham o mesmo ambiente); como os agentes precisam coordenar suas ações, a comunicação entre eles é essencial;

representação: o agente possui uma representação simbólica explícita daquilo que acredita ser verdade em relação ao ambiente e aos outros agentes que compartilham aquele ambiente;

motivação: como em SMA os agentes são (ou podem ser) autônomos, é essencial que exista não só uma representação do conhecimento do agente, mas também uma representação dos desejos ou objetivos (i.e., aspectos motivacionais) daquele agente; em termos práticos, isto significa ter uma representação de estados do ambiente que o agente almeja alcançar; como conseqüência, o agente age sobre o ambiente por iniciativa própria para satisfazer esses objetivos;

deliberação: dada uma motivação e uma representação do estado atual do ambiente em que se encontra o agente, esse tem que ser capaz de decidir, dentre os estados de ambiente possíveis de ocorrerem no futuro, quais de fato serão os objetivos a serem seguidos por ele;

raciocínio e aprendizagem: técnicas de inteligência artificial clássica para, por exemplo, raciocínio e aprendizagem podem ser estendidas para múltiplos agentes, aumentando significativamente o desempenho desses, por exemplo no aspecto de deliberação; note que nem sempre se esperam essas características de raciocínio e aprendizagem de quaisquer agentes; a criação de mecanismos de aprendizagem específicos para ambientes multiagente é uma área

termo *agente* se difundiu amplamente em diversas áreas da Ciência da Computação. Nessa perspectiva, criou-se o termo *agentes de software* (Genesereth e Ketchpel, 1994), em que praticamente qualquer processo comunicante passa a ser denominado agente.

de pesquisa que ainda requer bastante investigação (existem, contudo, vários trabalhos que aplicam aprendizagem por reforço em SMA (Weiß, 1997)).

4. A arquitetura BDI

As mais importantes arquiteturas de agentes deliberativos são baseadas em um modelo de cognição fundamentado em três principais atitudes mentais que são as crenças, os desejos, e as intenções (abreviadas por BDI, *beliefs*, *desires* e *intentions*). A fundamentação filosófica para esta concepção de agentes vem do trabalho de Dennett (1987) sobre sistemas intencionais e de Bratman (1987) sobre raciocínio prático. A primeira implementação de um sistema baseado nessas idéias foi o IRMA (*Intelligent Resource-bounded Machine Architecture*) (Bratman et al., 1988). Com base nessa experiência, foi criado outro sistema utilizando a arquitetura BDI, chamado PRS (*Procedural Reasoning System*) (Georgeff e Lansky, 1987); o PRS, por sua vez, tem como sucessor um sistema chamado dMARS (d'Inverno et al., 1998; Kinny, 1993). Baseando-se nesses trabalhos, Rao criou a linguagem de programação AgentSpeak(L). Georgeff e Rao foram os principais autores na elaboração de arquiteturas abstratas de agente baseadas no modelo BDI (Rao e Georgeff, 1992, 1995), bem como na concepção de lógicas BDI (Rao, 1996b; Rao e Georgeff, 1998). Lógicas BDI são lógicas multimodais com um operador modal para cada uma das atitudes mentais do modelo BDI, além dos operadores usuais da lógica temporal de tempo ramificado e da lógica de ação. Para uma apresentação sucinta de uma lógica BDI com referências para as lógicas modais nas quais ela se baseia, bem como para a semântica de mundos possíveis normalmente utilizada para elas, veja (Singh et al., 1999).

De forma esquemática, a arquitetura BDI genérica está apresentada na figura 4, conforme proposto em Wooldridge (1999).

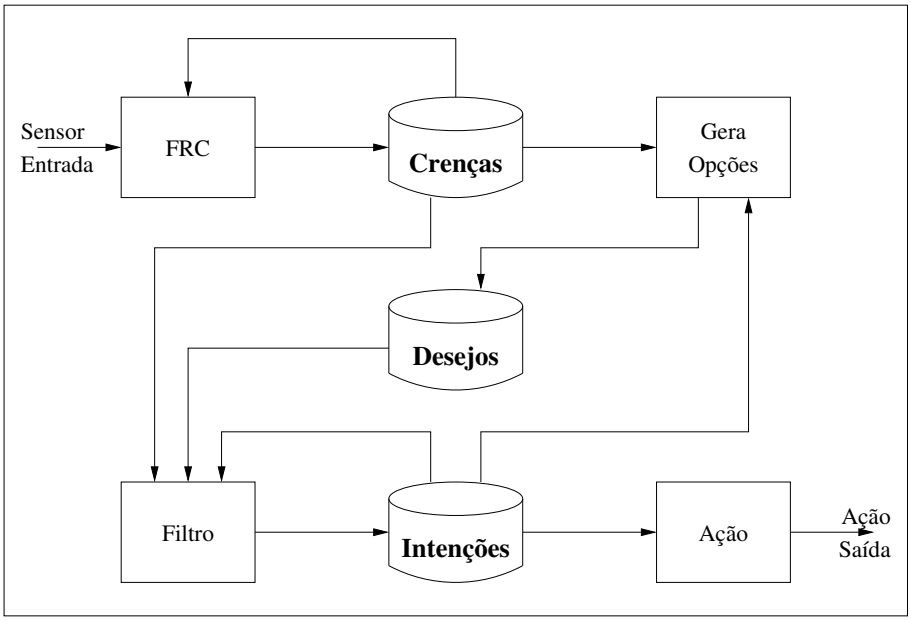


Figura 4: Arquitetura BDI Genérica (adaptado de Wooldridge (1999))

Resumidamente, esta arquitetura de agente está estruturada da seguinte forma. As *crenças* representam aquilo que o agente sabe sobre o estado do ambiente e dos agentes naquele ambiente (inclusive sobre si mesmo). Os *desejos* representam estados do mundo que o agente quer atingir (dito de outra forma, são

representações daquilo que ele quer que passe a ser verdadeiro no ambiente). Em tese, desejos podem ser contraditórios, ou seja, pode-se desejar coisas que são mutuamente exclusivas do ponto de vista de ação prática. Normalmente se refere a *objetivos* como um subconjunto dos desejos que são todos compatíveis entre si. As *intenções* representam seqüências de ações específicas que um agente se compromete a executar para atingir determinados objetivos.

A *função de revisão de crenças* (referenciada por *FRC* na figura) recebe a informação sensória (i.e., percebe propriedades do ambiente) e, consultando as crenças anteriores do agente, atualiza essas crenças para que elas reflitam o novo estado do ambiente[6]. Com essa nova representação do estado do ambiente, é possível que novas opções fiquem disponíveis (opções de estados a serem atingidos). A função denominada na figura como *Gera Opções* verifica quais as novas alternativas de estados a serem atingidos, que são relevantes para os interesses particulares daquele agente. Isto deve ser feito com base no estado atual do mundo (conforme as crenças do agente) e nas intenções com que o agente já está comprometido. A atualização dos objetivos se dá, então, de duas formas: as observações do ambiente possivelmente determinam novos objetivos do agente, e a execução de intenções de mais alto nível pode gerar a necessidade de que objetivos mais específicos sejam atingidos.

Uma vez atualizado o conhecimento e a motivação do agente, é preciso, em seguida, decidir que curso de ações específico será usado para alcançar os objetivos atuais do agente (para isso é preciso levar em conta os outros cursos de ações com os quais o agente já se comprometeu, para evitar ações incoerentes, bem como eliminar intenções que já foram atingidas ou que se tornaram impossíveis de ser atingidas). Esse é o papel da função *Filtro*, que atualiza o conjunto de intenções do agente, com base nas crenças e desejos atualizados e nas intenções já existentes. Esse processo realizado pela função Filtro para determinar como atualizar o conjunto de intenções do agente é normalmente chamado de *deliberação*[7]. Com o conjunto de intenções já atualizado, a escolha de qual ação específica, entre aquelas pretendidas, será a próxima a ser realizada pelo agente no ambiente é feita pela função *Ação*. Em certos casos, em que não é necessário priorização entre múltiplas intenções, a escolha pode ser simples; ou seja, basta escolher qualquer uma entre as intenções ativas, desde que se garanta que todas as intenções terão, em algum momento, a oportunidade de serem escolhidas para execução. Porém, alguns agentes podem precisar usar escolha de intenções baseadas em critérios mais sofisticados para garantir que certas intenções sejam priorizadas em relação a outras em certas circunstâncias.

5. A Linguagem AgentSpeak(L)

A linguagem AgentSpeak(L) foi projetada para a programação de agentes BDI na forma de sistemas de planejamento reativos (*reactive planning systems*). Sistemas de planejamento reativos são sistemas que estão permanentemente em execução, reagindo a eventos que acontecem no ambiente em que estão situados através da execução de planos que se encontram em uma biblioteca de planos parcialmente instanciados.[8]

[6]Note que os mecanismos reais de percepção normalmente são imprecisos e/ou incompletos; ou seja, a percepção pode ser falha, no sentido de não refletir a realidade do ambiente, ou pode não ser total, perdendo alguma informação sobre o ambiente.

[7]Essa é a origem do termo *deliberativo* muitas vezes utilizado para se referir a agentes cognitivos, já que a habilidade de realizar essa deliberação é uma característica muito importante para esse tipo de agente.

[8]O texto desta seção foi originalmente publicado em (Bordini e Vieira, 2003)

A linguagem de programação AgentSpeak(L) foi primeiramente apresentada em (Rao, 1996a). A linguagem é uma extensão natural e elegante de programação em lógica para a arquitetura de agentes BDI, que representa um modelo abstrato para a programação de agentes e tem sido a abordagem predominante na implementação de agentes inteligentes ou "racionais" (Wooldridge, 2000). Um agente AgentSpeak(L) corresponde à especificação de um conjunto de crenças que formarão a base de crenças inicial e um conjunto de planos. Um *átomo de crença* é um predicado de primeira ordem na notação lógica usual, e *literais de crença* são átomos de crenças ou suas negações. A *base de crenças* de um agente é uma coleção de átomos de crença.

AgentSpeak(L) distingue dois tipos de objetivos: *objetivos de realização* (*achievement goals*) e *objetivos de teste* (*test goals*). Objetivos de realização e teste são predicados, tais como crenças, porém com operadores prefixados '`!`' e '`?`', respectivamente. Objetivos de realização expressam que o agente quer alcançar um estado no ambiente onde o predicado associado ao objetivo é verdadeiro. Na prática, esses objetivos iniciam a execução de *subplanos*. Um objetivo de teste retorna a unificação do predicado de teste com uma crença do agente, ou falha caso não seja possível a unificação com nenhuma crença do agente. Um *evento ativador* (*triggering event*) define quais eventos podem iniciar a execução de um plano. Um *evento* pode ser interno, quando gerado pela execução de um plano em que um subobjetivo precisa ser alcançado, ou externo, quando gerado pelas atualizações de crenças que resultam da percepção do ambiente. Eventos ativadores são relacionados com a *adição* e *remoção* de atitudes mentais (crenças ou objetivos). Adição e remoção de atitudes mentais são representadas pelos operadores prefixados ('`+`') e ('`-`').

Planos fazem referência a *ações básicas* que um agente é capaz de executar em seu ambiente. Essas ações são definidas por predicados com símbolos predicativos especiais (chamados símbolos de ação) usados para distinguir ações de outros predicados. Um plano é formado por um evento ativador (denotando o propósito do plano), seguido de uma conjunção de literais de crença representando um *contexto*. O contexto deve ser conseqüência lógica do conjunto de crenças do agente no momento em que o evento é selecionado pelo agente para o plano ser considerado *aplicável*. O resto do plano é uma seqüência de ações básicas ou subobjetivos que o agente deve atingir ou testar quando uma instância do plano é selecionada para execução.

```
+concert(A,V) : likes(A)
   ← !book_tickets(A,V).

+!book_tickets(A,V) : ¬busy(phone)
   ← call(V);
      ...;
      !choose_seats(A,V).
```

Figura 5: Exemplos de planos AgentSpeak(L)

A figura 5 apresenta exemplos de planos AgentSpeak(L). O primeiro plano especifica que ao anúncio de um concerto a ser realizado pelo artista `A` no local `V` (do Inglês *venue*), correspondendo a adição de uma crença `concert(A,V)` como conseqüência da percepção do ambiente. Se for o caso de o agente gostar do artista

A, então o agente terá como objetivo a reserva dos ingressos para esse concerto. O segundo plano especifica que ao adotar o objetivo de reservar ingressos, se for o caso de a linha telefônica não estar ocupada, então o agente pode executar o plano que consiste de: executar a ação básica de fazer contato telefônico com o local do concerto V, seguido de um determinado protocolo de reserva de ingressos (indicado por '...'), e que termina com a execução de um subplano para a escolha de acentos em eventos desse tipo naquele local. (Assume-se que fazer contato telefônico é uma ação básica que o agente é capaz de executar no ambiente em questão).

5.1. Sintaxe Abstrata

A especificação de um agente *ag* em AgentSpeak(L) deve ser feita de acordo com a gramática apresentada na figura 6 (adaptada de (Moreira e Bordini, 2002)). Em AgentSpeak(L), um agente é especificado por um conjunto de crenças *bs* (*beliefs*) correspondendo à base de crenças inicial do agente, e um conjunto de planos *ps* que forma a biblioteca de planos do agente.

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

Figura 6: Sintaxe AgentSpeak(L)

As fórmulas atômicas *at* da linguagem são predicados onde P é um símbolo predicativo e $t_1, \ldots, t_n$ são termos padrão da lógica de primeira ordem. Chamamos de *crença* uma fórmula atômica *at* sem variáveis e *b* é meta-variável para crenças. O conjunto inicial de crenças de um programa AgentSpeak(L) é uma seqüência de crenças *bs*.

Um plano em AgentSpeak(L) é dado por *p* acima, onde *te* (*triggering event*) é o evento ativador, *ct* é o contexto do plano (uma conjunção de literais de crença) e *h* é uma seqüência de ações, objetivos ou atualizações de crenças. A construção $te : ct$ é dita a *cabeça* do plano, e *h* o *corpo* do plano. O conjunto de planos de um agente é dado por *ps* como uma lista de planos.

Um evento ativador *te* corresponde à adição/remoção de crenças da base de crenças do agente ($+at$ e $-at$, respectivamente), ou à adição/remoção de objetivos ($+g$ e $-g$, respectivamente). O agente possui um conjunto de *ações básicas* que utiliza para atuar sobre o ambiente. Ações são referidas por predicados usuais com a exceção de que um símbolo de ação A é usado no lugar do símbolo predicativo. Objetivos *g* podem ser objetivos de realização ($!at$) ou de teste ($?at$). Finalmente, $+at$ e $-at$ (no corpo de um plano) representam operações de atualização (*update*) da base de crença *u*, através da adição ou remoção de crenças respectivamente.

Note que uma fórmula $!g$ no corpo de um plano gera um evento cujo evento ativador é $+!g$ (este assunto será tratado em maiores detalhes na próxima seção). Portanto, planos escritos pelo programador que tenha um evento ativador que possa ser unificado com $+!g$ representam alternativas de planos que devem ser considerados no tratamento de tal evento. Planos com evento ativador do tipo $+at$ e $-at$ são utilizados no tratamento de eventos que são gerados quando crenças são adicionadas ou removidas (tanto como conseqüência da percepção do ambiente, como devido a alterações de crenças explicitamente requisitadas no corpo de um plano). Eventos ativadores do tipo $-!g$ são usados para o tratamento de falhas de planos (mas não serão abordados neste texto), e eventos ativadores do tipo $+?g$ e $-?g$ não são utilizados na implementação atual da linguagem. A seção a seguir fornece mais detalhes sobre a geração de eventos, e outros aspectos importantes da interpretação de programas AgentSpeak(L).

5.2. Semântica Informal

Um interpretador abstrato para a linguagem AgentSpeak(L) precisa ter acesso à base de crenças e à biblioteca de planos, e gerenciar um conjunto de *eventos* e um conjunto de *intenções*. Seu funcionamento requer três *funções de seleção*: a função de seleção de eventos (S_E) seleciona um único evento do conjunto de eventos; uma outra função (S_{Ap}) seleciona uma "opção" (um plano aplicável) entre o conjunto de planos aplicáveis para um evento selecionado; e a terceira função (S_I) seleciona uma intenção do conjunto de intenções. As funções de seleção são específicas para cada agente, sendo responsáveis por parte significativa do comportamento do agente. Essas funções, apesar de fazerem parte do interpretador, devem ser fornecidas pelo projetista do agente juntamente com o programa AgentSpeak(L), exceto nas situações em que funções de seleção *default*[9], que são muito simples, possam ser utilizadas.

Os trabalhos anteriores sobre AgentSpeak(L) não discutem como projetistas podem especificar essas funções. Em princípio linguagens de programação tradicionais devem ser usadas para a definição de funções de seleção *ad hoc*. A extensão de AgentSpeak(L) apresentada em (Bordini et al., 2002) visava justamente esse problema, propondo a geração automática e eficiente de funções de seleção de intenções. A linguagem estendida permite expressar relações entre planos, bem como critérios quantitativos para sua execução. Usa-se, então, escalonamento de tarefas baseado em teoria de decisão para guiar as escolhas feitas pela função de seleção de intenções.

Como dito acima, duas estruturas importantes para o interpretador abstrato são o conjunto de eventos e o conjunto de intenções. *Intenções* são cursos de ações com os quais um agente se compromete para tratar certos eventos. Cada intenção é uma pilha de planos parcialmente instanciados.

Eventos causam o início da execução de planos que tem eventos ativadores relevantes. Eventos podem ser *externos*, quando originados da percepção do ambiente (i.e., adição ou remoção de crenças resultantes do processo de revisão de crenças); ou *internos*, quando gerados pela execução de planos do agente (um subobjetivo em um plano gera um evento do tipo "adição de objetivo de realização"). No último caso, o evento é acompanhado da intenção que o gerou (o plano escolhido para aquele evento

[9]As funções de seleção *default* são as seguintes: S_E seleciona eventos sem nenhuma prioridade, utilizando a ordem em que eles são inseridos no conjunto de eventos; a função S_{Ap} utiliza a ordem em que os planos foram escritos no programa fonte AgentSpeak(L) para a escolha entre diversos planos aplicáveis; e a função S_I funciona, por assim dizer, como um escalonador *round-robin*, executando uma ação de cada uma das intenções ativas por vez.

será colocado no topo daquela intenção). Eventos externos criam novas intenções representando diferentes focos de atenção na atuação do agente no ambiente.

A seguir o funcionamento de um interpretador AgentSpeak(L) é apresentado em detalhe, com o auxílio da figura 7 (reproduzida de (Machado e Bordini, 2002)). Na figura, conjuntos de crenças, eventos, planos e intenções são representados por retângulos. Losangos representam a seleção de um elemento de um conjunto. Círculos representam alguns dos processos envolvidos na interpretação de programas AgentSpeak(L).

A cada ciclo de interpretação de um programa AgentSpeak(L), a lista de eventos é atualizada com o resultado do processo de revisão de crenças. Assume-se que as crenças são atualizadas pela percepção e que sempre que houverem mudanças na base de crenças do agente, isso implica na inserção de um evento no conjunto de eventos. Essa função de revisão de crenças não é parte de um interpretador AgentSpeak(L), mas é um componente que deve estar presente na arquitetura geral do agente (implementações de interpretadores AgentSpeak(L) tipicamente fornecem uma função de revisão de crenças simples utilizada como *default*).

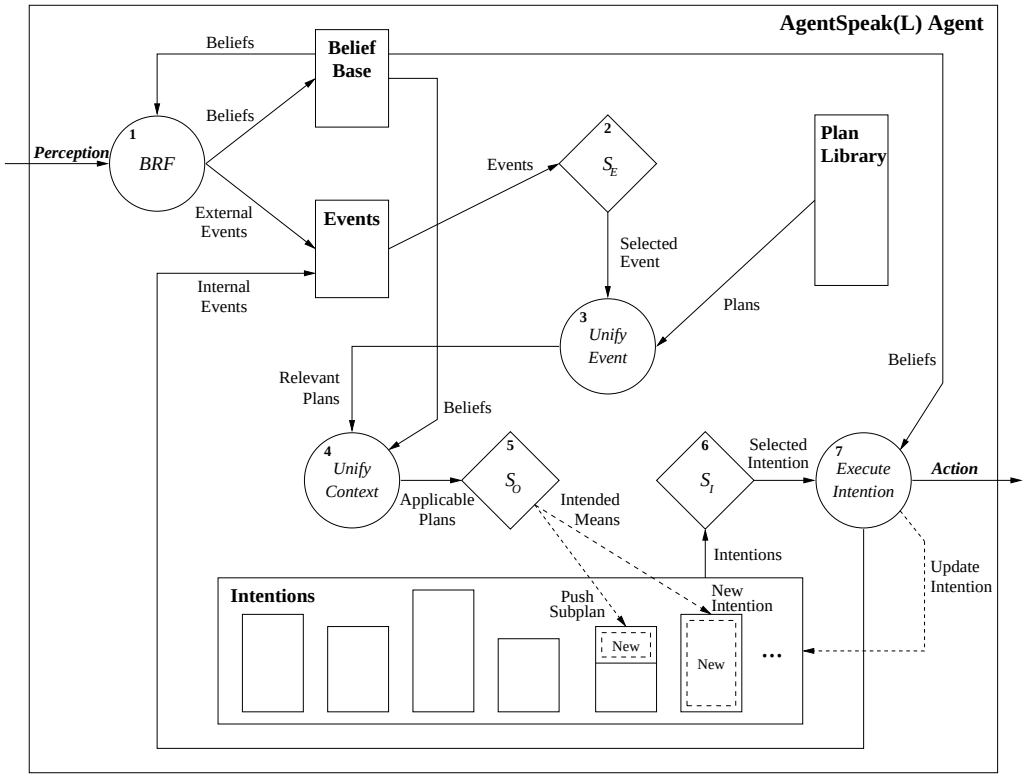


Figura 7: Ciclo de interpretação de um programa AgentSpeak(L) (Machado e Bordini, 2002).

Depois de S_E selecionar um evento, o interpretador AgentSpeak(L) tem que unificar aquele evento com eventos ativadores nas cabeças dos planos presentes na biblioteca de planos. Isso gera um conjunto de todos os *planos relevantes* para o evento escolhido. Pela verificação dos contextos de planos que seguem logicamente das crenças do agente, AgentSpeak(L) determina o conjunto de *planos aplicáveis* (planos que podem ser usados, na situação presente, para tratar o evento selecionado naquele ciclo). Depois, S_{Ap} escolhe, entre os planos do conjunto de planos aplicáveis, um único plano aplicável que se torna o *meio pretendido* para o tratamento daquele evento, e coloca o plano no topo de uma intenção existente (se o evento for interno), ou cria uma nova intenção no conjunto de intenções (se o evento for externo, i.e.,

gerado por percepção do ambiente), definindo um novo "foco de atenção" do agente.

Nesse estágio, resta apenas a seleção de uma única intenção para ser executada no ciclo. A função S_I seleciona uma intenção do agente (i.e. uma das pilhas de planos parcialmente instanciados que se encontram dentro do conjunto de intenções, cada uma representando um dos focos de atenção do agente). No topo dessa intenção existe uma instância de um plano da biblioteca de planos, e a fórmula no início do corpo do plano é executada. Isso implica em uma ação básica a ser realizada pelo agente no ambiente, na geração de um evento interno (se a fórmula selecionada for um objetivo de realização) ou na execução de um objetivo de teste (através do acesso à base de crenças). Caso a fórmula seja um objetivo de realização, simplesmente um evento do tipo "adição de objetivo de realização" é adicionado ao conjunto de eventos, acompanhado da intenção que gerou o evento. Essa intenção tem que ser removida do conjunto de intenções, pois ela fica suspensa até que o evento interno seja escolhido pela função S_E. Quando uma instância de plano for escolhida como meio pretendido para tratar este evento, o plano é colocado no topo da pilha de planos daquela intenção, e ela é retornada para o conjunto de intenções (podendo novamente ser selecionada por S_I).

Se a fórmula a ser executada é a realização de uma ação básica ou a execução de um objetivo de teste, a fórmula deve ser removida do corpo da instância de plano que se encontra no topo da intenção selecionada. No caso da execução de um objetivo de teste, a base de crenças será inspecionada para encontrar um átomo de crença que unifica com o predicado de teste. Se uma unificação for possível, instanciações de variáveis podem ocorrer no plano parcialmente instanciado; após isto, o objetivo de teste pode ser removido do conjunto de intenções, pois já foi realizado. No caso de uma ação básica a ser executada, o interpretador simplesmente informa ao componente da arquitetura do agente que é responsável pela atuação sobre o ambiente qual ação é requerida, podendo também remover a ação[10] do conjunto de intenções. Quando todas as fórmulas no corpo de um plano forem removidas (i.e., tiverem sido executadas), o plano é removido da intenção, tal como o objetivo de realização que o gerou, se esse for o caso, é removido do início do corpo do plano abaixo daquele na pilha de planos daquele foco de atenção. O ciclo de execução termina com a execução de uma fórmula do corpo de um plano pretendido, e AgentSpeak(L) começa um novo ciclo, com a verificação do estado do ambiente após a ação do agente sobre ele, a geração dos eventos adequados, e continuando a execução de um ciclo de raciocínio do agente como descrito acima.

O objetivo desta seção foi apresentar informalmente alguns aspectos importantes da semântica da linguagem AgentSpeak(L). A semântica formal da linguagem pode ser encontrada nos seguintes artigos (Bordini e Moreira, 2004; Moreira e Bordini, 2002; Moreira et al., 2003) (o último deles trata especificamente de aspectos de comunicação entre agentes).

5.3. Estudos de Caso

Esta seção mostra dois exemplos bem simples de programas escritos em AgentSpeak(L). O fato de os programas serem simples é uma vantagem para a assimilação dos conceitos básicos da linguagem, porém dificultam a compreensão da importância das abstrações da arquitetura BDI em sistemas mais complexos. O objetivo aqui

[10]Em algumas implementações, e.g. (Bordini et al., 2002), o interpretador espera um retorno do ambiente simulado para saber se a ação pode ser executada ou não. O plano falha caso o retorno do ambiente não seja positivo.

não é mostrar o tipo de sistema em que este paradigma é particularmente apropriado, mas especificamente mostrar exemplos das construções básicas da linguagem AgentSpeak(L). Ambos exemplos foram utilizados em trabalhos que propõem o uso da técnica de *model checking* (verificação de modelos) para a verificação formal de sistemas programados em AgentSpeak(L).

5.3.1. Robôs Coletores de Lixo em Marte

O cenário utilizado nesta seção foi apresentado em (Campbell e d'Inverno, 1990) e lembra também o cenário utilizado em (Rao, 1996a); o código apresentado aqui para este cenário foi introduzido anteriormente em (Bordini et al., 2003b). Dois robôs estão coletando lixo no planeta Marte. O robô `r1` procura por lixos depositados no solo do planeta e quando algum lixo é encontrado, o robô coleta o lixo e o leva para o local onde `r2` encontra-se, larga o lixo lá e retorna ao local onde o lixo foi encontrado para continuar sua busca a partir daquela posição anterior. O robô `r2` está posicionado ao lado de um incinerador; todo o lixo levado por `r1` é colocado no incinerador. Os pedaços de lixo são colocados randomicamente em uma grade que define o território do planeta[11]. Outra fonte de não-determinismo é a imprecisão do braço do robô em pegar os pedaços de lixo. A ação de pegar o lixo pode falhar, mas o mecanismo é bom o suficiente para não falhar mais que duas vezes seguidas; ou seja, no pior caso `r1` irá tentar apanhar o lixo três vezes.

O código AgentSpeak(L) para `r1` é dado abaixo. No código, cada plano é anotado com um *label* para que se possa referir ao plano no texto que segue. As ações que possuem um ponto ('.') em seu nome, denotam ações internas, uma noção introduzida em (Bordini et al., 2002). Essas ação são executadas internamente pelo agente, e não afetam o ambiente como as ações básicas que o agente executa.

Agent `r1`

Beliefs

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

Plans

```
+pos(r1,X1,Y1) : checking(slots) & not(garbage(r1))                (p1)
   <- next(slot).

+garbage(r1) : checking(slots)                                     (p2)
   <- !stop(check);
      !take(garb,r2);
      !continue(check).

+!stop(check) : true                                               (p3)
   <- ?pos(r1,X1,Y1);
      +pos(back,X1,Y1);
      -checking(slots).

+!take(S,L) : true                                                 (p4)
   <- !ensure_pick(S);
      !go(L);
```

[11] Cada posição da grade representa uma área de território de tamanho limitado pelo alcance dos sensores do robô. Qualquer lixo dentro da área de território representado por uma posição na grade é instantaneamente percebido pelo agente.

```
      drop(S).

+!ensure_pick(S) : garbage(r1)                                    (p5)
   <- pick(garb);
      !ensure_pick(S).

+!ensure_pick(S) : true <- true.                                  (p6)

+!continue(check) : true                                          (p7)
   <- !go(back);
      -pos(back,X1,Y1);
      +checking(slots);
      next(slot).

+!go(L) : pos(L,Xl,Yl) & pos(r1,Xl,Yl)                            (p8)
   <- true.
+!go(L) : true                                                    (p9)
   <- ?pos(L,Xl,Yl);
      moveTowards(Xl,Yl);
      !go(L).
```

A crença inicial do agente é sobre a posição do agente `r2` na grade que define o território do planeta em que `r1` deve procurar lixo, e que sua tarefa inicial será verificar os diversos pontos do território, procurando lixo. Todos os planos são explicados abaixo.

O plano `p1` é usado quando o agente percebe que ele está numa nova posição e está procurando por lixo. Se não há lixo percebido naquele ponto, só o que deve ser feito é a ação básica `next(slot)` que move o robô para o próximo ponto na grade (com exceção do local do incinerador, pois o lixo nessa posição é tratado por `r2`). Note que essa é uma ação básica do ponto de vista do agente: assume-se que o robô possui os mecanismos físicos necessários para se mover a uma *próxima* posição no território, seguindo algum caminho pré-determinado de tal forma que todas as posições sejam ordenadas e o mecanismo pode então detectar quando não há mais nenhuma posição seguinte a ser verificada.

O ambiente simulado provê a informação sobre a existência de lixo nas posições dos robôs `r1` e `r2`, quando o agente realiza percepção do ambiente. Quando `r1` percebe lixo em sua posição, a crença `garbage(r1)` é adicionada à base de crenças de tal maneira que o plano `p2` pode ser então usado. A tarefa de lidar com um pedaço de lixo percebido é decomposta em três partes, envolvendo subobjetivos de realização que garantem que: (i) o robô irá parar de procurar por lixo de maneira consistente (lembrando seu último ponto de procura para que a tarefa possa ser continuada daquele ponto); (ii) o lixo seja levado até a posição de `r2`; e (iii) a tarefa de procurar por lixo será, então, retomada. Cada um desses objetivos são realizados respectivamente pela execução dos três planos que seguem.

Quando o agente pretende atingir o subobjetivo (i) acima, o plano `p3` é sua única opção, e ele é sempre aplicável (já que seu contexto é vazio). O agente recupera da sua base de crenças sua posição atual (a posição é inserida na base de crenças através da percepção do ambiente). O agente então toma nota da informação de para onde deve retornar para a continuação da sua busca por lixos. Isso é feito pela adição de uma crença na base de crenças: `+pos(back,X1,Y1)`. O agente remove da sua base de crenças a informação de que está procurando por lixo, e após isso o agente terá o objetivo de levar o lixo até a posição de `r2` e retornar.

O subobjetivo (ii) é tratado pelo plano `p4`, que diz que para o robô levar o lixo até a posição de `r2`, ele deve coletar o lixo e atingir o subobjetivo de ir para a posição de `r2` e, quando chegar lá, ele poderá finalmente largar o lixo. Note que `pick(garb)` e `drop(garb)` são ações básicas, ou seja, são coisas que o robô pode realizar fisicamente no ambiente por meio de seu hardware.

Os planos `p5` e `p6` juntos asseguram que o robô irá continuar tentando pegar um pedaço de lixo até que ele não possa mais perceber o lixo na grade (i.e., até que a ação de coleta seja realizada com sucesso). Lembre que o mecanismo de coleta é impreciso, e o robô pode ter que tentar algumas vezes até obter êxito na execução da ação.

O plano `p7` é usado para que o agente continue a tarefa de verificar a existência de lixo na grade que representa o espaço de território a ser limpo. O agente precisa atingir o subobjetivo de retornar para sua posição anterior (`!go(back)`), e uma vez lá ele poderá remover a nota que fez sobre aquela posição, lembrando que voltou ao estado de procura por lixo e, assim, procedendo para o próximo ponto na grade.

Os últimos dois planos são usados para atingir o objetivo de ir para uma posição específica na grade (representada pela constante com a qual a variável `L` está instanciada). O plano `p9` recupera a crença que o agente possui sobre a posição na grade em que fica a localidade `L`, move-se uma posição na grade em direção ao ponto referente às coordenadas em questão `moveTowards(Xl,Yl)`, e volta a ter o objetivo de mover-se (continuar movendo-se) em direção a `L`; note que o plano é recursivo. O plano `p8` estabelece o fim da recursão dizendo que não há mais o que fazer para atingir o objetivo de ir em direção a `L` se o agente já está naquela posição (o plano anterior não mais será aplicável neste caso).

O agente `r2` é definido pelo código AgentSpeak(L) abaixo. Tudo o que ele faz é queimar os pedaços de lixo (`burn(garb)`) quando ele percebe que há lixo nessa posição (`+garbage(r2)`).

Agent `r2`

```
+garbage(r2) : true
   <- burn(garb).
```

Crenças `garbage(r2)` são adicionadas à base de crenças do agente a partir da percepção do ambiente, sempre que existir lixo no ponto da grade onde `r2` se encontra.

5.3.2. Um Modelo Abstrato de Leilões

Nessa seção será descrito um cenário simplificado de leilões, utilizado anteriormente em (Bordini et al., 2003a). O ambiente anuncia 10 leilões e designa quem é o vencedor em cada um (aquele com o lance mais alto). Existem três agentes participando desses leilões com estratégias de lance simplificadas.

Agent `ag1`

```
+auction(N) : true
   <- place_bid(N,6).
```

O agente `ag1` é um agente que oferece 6 como lance sempre que o ambiente anuncia um novo leilão.

Agent `ag2`

```
myself(ag2).
bid(ag2,4).
ally(ag3).

+auction(N) : myself(I) & ally(A) & not(alliance(A,I))
   <- ?bid(I,B); place_bid(N,B).

+auction(N) : alliance(A,I)
   <- place_bid(N,0).

+alliance(A,I) : myself(I) & ally(A)
   <- ?bid(I,B);
      .send(A,tell,bid(I,B));
      .send(A,tell,alliance(A,I)).
```

O agente ag2 oferece 4, caso não tenha combinado uma aliança com ag3, caso em que ele oferece 0 (já que ag3 fará um lance em nome da aliança entre eles). Quando ag2 recebe uma mensagem de ag3 propondo uma aliança, a crença alliance(ag3,ag2) é adicionada à base de crenças de ag2; uma função de confiança (*trust*) *default* é utilizada. Isso constitui um evento ativador para um último plano que informa ao ag3 o quanto ag2 costuma oferecer como lance e confirma que ag2 concorda em formar uma aliança com ag3.

Agent ag3

```
myself(ag3).
bid(ag3,3).
ally(ag2).
threshold(3).

+auction(N) : threshold(T) & .gte(T,N)
   <- !bid_normally(N).
+auction(N) : myself(I) & winner(I)
              & ally(A) & not(alliance(I,A))
   <- !bid_normally(N).

+auction(N) : myself(I) & not(winner(I))
              & ally(A) & not(alliance(I,A))
   <- !alliance(I,A);
      !bid_normally(N).

+auction(N) : alliance(I,A)
   <- ?bid(I,B); ?bid(A,C);
      .plus(B,C,D); place_bid(N,D).

+!bid_normally(N) : true
   <- ?bid(I,B); place_bid(N,B).

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

O agente ag3 tenta ganhar os primeiros T leilões, onde T é um limiar (*threshold*) armazenado em sua base de crenças. Se ele não ganha nenhum leilão até aquele limiar, ele tenta então fazer uma aliança com ag2 (através de um envio de mensagem). Quando ag2 confirma a aliança, então ag3 apresenta em nome dos dois um lance com a soma de suas lances usuais.

5.4. Trabalhos Relacionados

A partir da proposta original apresentada por Rao (1996a), vários outros autores têm levado adiante a investigação de diferentes aspectos de AgentSpeak(L). Em (d'Inverno e Luck, 1998) um interpretador abstrato para essa linguagem foi formalmente especificado através da linguagem Z. Muitos dos elementos nessa formalização foram primeiramente apresentados em (d'Inverno et al., 1998), que apresenta uma especificação formal para dMARS. Isso deve-se ao fato de AgentSpeak(L) ser fortemente baseada nas experiências com o sistema dMARS (Kinny, 1993), sucessor do PRS (Georgeff e Lansky, 1987), ambos sistemas que utilizam a arquitetura BDI.

Algumas extensões de AgentSpeak(L) foram propostas em (Bordini et al., 2002), e um interpretador para a versão estendida foi apresentado. As extensões tinham por objetivo a obtenção de uma linguagem de programação que fosse mais prática. A linguagem estendida permite a especificação de relações entre planos e critérios quantitativos para sua execução. O interpretador usa, então, um escalonador de tarefas baseado em teoria de decisão para guiar, automaticamente, as escolhas relacionadas à função de seleção de intenções de um agente.

Em (Moreira e Bordini, 2002), uma semântica operacional para AgentSpeak(L) foi dada com base na abordagem estrutural de Plotkin (1981), uma notação mais usual do que Z para dar semântica a linguagens de programação. Posteriormente, essa semântica operacional foi utilizada na especificação de um *framework* para a construção de provas de propriedades BDI da linguagem AgentSpeak(L) (Bordini e Moreira, 2002). A combinação dos princípios da "tese da assimetria" (*assymetry thesis*) satisfeitos por um agente AgentSpeak(L) foi apresentado inicialmente em (Bordini e Moreira, 2002); as provas detalhadas dessas propriedades foram apresentadas em (Bordini e Moreira, 2004). Esses princípios são relevantes para se assegurar a racionalidade dos agentes programados em AgentSpeak(L) com base nos princípios de racionalidade segundo a teoria BDI. Em (Moreira et al., 2003), a semântica operacional de AgentSpeak(L) foi estendida para dar semântica à comunicação entre agentes baseada na teoria dos atos de fala.

Em (Bordini et al., 2003c) foi apresentado o uso de técnicas de *model checking* para a verificação formal de programas AgentSpeak(L) através de um conjunto de ferramentas chamado CASP. CASP traduz uma versão simplificada de AgentSpeak(L) em linguagens que podem ser utilizadas em verificadores de modelos para a lógica temporal linear (Allen Emerson, 1990). A tradução para Promela foi apresentada em (Bordini et al., 2003a), e a tradução para Java em (Bordini et al., 2003b). A verificação de modelo é então feita com o uso de Spin (Holzmann, 1991) no primeiro caso, e com JPF2 (Visser et al., 2000) no segundo. Esse trabalho em verificação de modelos para programas AgentSpeak(L) usa as definições de modalidades BDI em termos da semântica operacional de AgentSpeak(L) apresentadas em (Bordini e Moreira, 2004) para mostrar de forma precisa como as especificações BDI (a serem verificadas para sistemas com múltiplos agentes AgentSpeak(L)) são interpretadas.

Poucas aplicações foram desenvolvidas com AgentSpeak(L) até o momento, dado que a sua implementação prática é muito recente e ainda requer um trabalho mais aprofundado de experimentação para que se torne uma linguagem de programação poderosa. O mesmo se aplica a outras linguagens baseadas em agentes (veja a próxima seção). Apesar do trabalho na área datar do início dos anos 90, um grande número de questões está ainda em aberto. Entre as aplicações existentes, desenvolvidas em AgentSpeak(L), podemos mencionar uma simulação de aspectos

sociais do crescimento urbano (Bordini et al., 2004) e um robô carregador encarregado do armazém de uma fábrica, que funciona em um ambiente de realidade virtual (Torres et al., 2003). O primeiro trabalho foi desenvolvido como um estudo de caso para a plataforma MAS-SOC[12] de simulação social baseada em agentes em que cada agente de uma simulação é implementado em AgentSpeak(L). O segundo trabalho apresenta uma arquitetura de duas camadas com o interpretador AgentSpeak(L) em um nível, e, em outro nível, um sistema articulado para a modelagem de personagens 3D em ambientes virtuais. Tal arquitetura tem por objetivo permitir o uso de agentes autônomos sofisticados em sistemas de realidade virtual ou outras aplicações baseadas em animações[13] tridimensionais.

6. Programação de agentes BDI com *Jason*

Jason é um interpretador para um extensão de AgentSpeak incluindo comunicação entre agentes baseada na teoria de atos de fala (Austin, 1975; Searle e Vanderveken, 1985). Utilizando o SACI[14] (Hübner e Sichman, 2000), um SMA desenvolvido com o ***Jason*** pode ser distribuído em uma rede de computadores sem muito esforço. Existem muitas implementações *ad hoc* de sistemas BDI, contudo uma característica importante do AgentSpeak é sua fundamentação teórica (trabalhos sobre a verificação formal de sistemas AgentSpeak estão em andamento — referências serão dadas ao longo do texto). Outra característica importante do ***Jason*** em comparação como outros sistemas BDI é que ele é implementado em Java (portanto multi-plataforma) e é disponível como *Open Source* sob a licença GNU LGPL[15].

Além de interpretar a linguagem AgentSpeak(L) original, o ***Jason*** possui os seguintes recursos:

- negação forte (*strong negation)*, portanto tanto sistemas que consideram mundo-fechado (*closed-world*) quanto mundo-aberto (*open-world*) são possíveis;
- tratamento de falhas em planos;
- comunicação baseada em atos de fala (incluindo informações de fontes como anotações de crenças);
- anotações em identificadores de planos, que podem ser utilizadas na elaboração de funções personalizadas para seleção de planos;
- suporte para o desenvolvimento de ambientes (que normalmente não é programada em AgentSpeak; no ***Jason*** o ambiente é programado em Java);
- a possibilidade de executar o SMA distribuidamente em uma rede (usando o SACI);
- possibilidade de especializar (em Java) as funções de seleção de planos, as funções de confiança e toda a arquitetura do agente (percepção, revisão de crenças, comunicação e atuação);
- possuir uma biblioteca básica de "ações internas";
- possibilitar a extensão da biblioteca de ações internas.

[12]A URL do projeto MAS-SOC é <`http://www.inf.ufrgs.br/~massoc/`>. Nesta página também podem ser encontrados *links* para outros recursos relacionados à linguagem AgentSpeak(L).

[13]Exemplos de animações realizadas com esta arquitetura podem ser encontrados na URL <`http://www.inf.ufrgs.br/cg/ras/`>.

[14]Disponível em `http://www.lti.pcs.usp.br/saci/`.

[15]Disponível em `http://jason.sourceforge.net`

6.1. Sintaxe para definição dos agentes

A gramática BNF abaixo especifica a sintaxe AgentSpeak que é aceita pelo ***Jason***. Nesta BNF, <ATOM> é um identificador que inicia com letra minúscula ou '.', <VAR> (uma variável) é um identificador que inicia com letra maiúscula, <NUMBER> é qualquer inteiro ou real e <STRING> é uma string delimitada por aspas duplas.

```
agent                a  beliefs plans
beliefs              a  ( literal "." )*
                        N.B.: um erro semântico é gerado se o
                        literal tiver variáveis não instanciadas.
plans                a  ( plan )+
plan                 a  [ atomic_formula "->" ]
                        triggering_event ":" context "<-" body "."
triggering_event     a  "+"     literal
                     |  "-"     literal
                     |  "+" "!" literal
                     |  "-" "!" literal
                     |  "+" "?" literal
                     |  "-" "?" literal
literal              a  "~" atomic_formula
                     |  "~" "(" atomic_formula ")"
                     |  atomic_formula
default_literal      a  "not" literal
                     |  "not" "(" literal ")"
                     |  literal
context              a  "true"
                     |  default_literal ( "&" default_literal )*
body                 a  "true"
                     |  body_formula ( ";" body_formula )*
body_formula         a  literal
                     |  "!" literal
                     |  "?" literal
                     |  "+" literal
                     |  "-" literal
atomic_formula       a  <ATOM> [ "(" list_of_terms ")" ]
                        [ "[" list_of_annotations "]" ]
structure            a  <ATOM> "(" list_of_terms ")"
list_of_terms        a  term ( "," term )*
list_of_annotations  a  as list_of_terms, but generating a
                        semantic error if not ground;
list                 a  "["
                        [ term ( ( "," term )*
                               | "|" ( list | <VAR> )
                               )
                        ] "]"
term                 a  structure
                     |  list
                     |  <ATOM>
                     |  <VAR>
                     |  <NUMBER>
                     |  <STRING>
```

As principais diferenças em relação a linguagem AgentSpeak(L) original são as seguintes: enquanto que um predicado era aceito na linguagem original, no ***Jason*** é utilizado um literal. Um literal é uma fórmula atômica `p(`t_1`,...,`t_n`)`, $n \geq 0$, ou `~p(`t_1`,...,`t_n`)`, onde '`~`' denota negação forte. A negação *default* é usada no contexto de um plano e é denotada por '`not`' precedendo o literal. O contexto é portanto uma conjunção de literais.[16] Cada termo t_i pode ser um átomo, uma estrutura, uma variável ou uma lista como no Prolog, ou ainda um inteiro, um real ou uma string.

Um diferença maior é que os predicados (i.e., fórmulas atômicas) podem ter anotações. Uma anotação é uma lista de termos delimitadas por colchetes que segue imediatamente o predicado. Na base de crenças, anotações são usadas para registrar as fontes das informações (ambiente, interna, outros agentes, ...). Dois átomos especiais, `percept` e `self`, são usados para denotar que uma crença tem sua origem em percepções do ambiente ou a partir de inclusão explícita na base de crenças feita durante a execução de um plano. As crenças iniciais que fazem parte do código fonte de um agente são assumidas como crenças internas (i.e., elas têm uma anotação `[self]`), a menos que o predicado tenha outra anotação explicitamente dado pelo programador (isso pode ser útil se o programador deseja que o agente tenha uma crença inicial sobre o ambiente ou sobre informações enviadas por outros agentes). Mais informações sobre anotações em crenças podem ser obtidas em (Moreira et al., 2003).

Finalmente, como já apresentado em (Bordini et al., 2002), *ações internas* podem ser usadas tanto no contexto como no corpo de um plano. Qualquer símbolo iniciando com '`.`', ou tendo um '`.`' em algum lugar, denota uma ação interna. Ações internas são ações definidas pelo usuário que executam internamente no agente. São chamadas internas para ditingui-las das ações que estão no corpo de um plano e denotam ações que um agente deseja executar para alterar o ambiente (chamadas "ações básicas" que representam a atuação do agente). No ***Jason***, ações internas são definidas pelo usuário na linguagem Java. Algumas das ações internas que fazem parte da biblioteca padrão são[17]:

`.send:` esta ação interna é usada por um agente para enviar uma mensagem. O primeiro argumento (`receiver`) é simplesmente o nome do receptor da mensagem; os valores disponíveis para o segundo argumento (`illocutionary_force`) são `tell`, `untell`, `achieve`, `unachieve`, `tellHow` e `untellHow`. O efeito de receber uma mensagem com estas forças ilocucionárias estão formalmente definidas em (Moreira et al., 2003). Finalmente, `propositional_content` é um literal representado o conteúdo da mensagem.

`.print:` é usado para imprimir uma mensagem no console. Um número qualquer de termos pode ser informado como parâmetro.

`.desire:` esta ação recebe um literal como argumento e tem sucesso se o literal é um dos desejos do agente (mais informações no manual do ***Jason***);

`.sum:` esta ação recebe termos como argumentos e tem sucesso se a soma dos 2 primeiros for igual ao terceiro, caso o terceiro argumento seja uma variável livre, esta receberá o valor da soma.

6.2. Definição e Execução de um SMA no *Jason*

Nesta seção é descrito como um SMA é definido para ser executado no ***Jason***. A definição de um SMA deve incluir um conjunto de agentes AgentSpeak e um

[16] Mais informação sobre os conceitos de negação forte e *default* podem ser encontradas em (Leite, 2003).

[17] A lista completa das ações internas e a forma de estender a biblioteca pode pode ser obtida no manual do ***Jason*** disponível em `http://jason.sourceforge.net`.

ambiente onde todos estes agentes estarão situados. Estas definições devem ser informadas em um arquivo texto com extensão `.mas2j` e em conformidade com a sintaxe definida pela gramática abaixo. Nesta gramática, `<NUMBER>` é um número inteiro, `<ASID>` são identificados AgentSpeak, que devem iniciar com letra minúscula, `<ID>` é qualquer identificados e `<PATH>` é um nome de arquivo com caminho.

```
mas          a  "MAS" <ID> "{"
                   [ "architecture" ":" <ID> ]
                   environment
                   agents
                "}"
environment  a  "environment" ":" <ID> [ "at" <ID> ]
agents       a  "agents" ":" ( agent )+
agent        a  <ASID>
                [ filename ]
                [ options ]
                [ "agentArchClass" <ID> ]
                [ "agentClass" <ID> ]
                [ "#" <NUMBER> ]
                [ "at" <ID> ]
                ";"
filename     a  [ <PATH> ] <ID>
options      a  "[" option ( "," option )* "]"
option       a  "events" "=" ( "discard" | "requeue" )
             |  "intBels" "=" ( "sameFocus" | "newFocus" )
             |  "verbose" "=" <NUMBER>
```

O `<ID>` que segue a palavra reservada `MAS` é o nome da sociedade. A palavra reservada `architecture` é usada para especificar qual a arquitetura utilizada na criação dos agentes. As opções atuais são "`Centralised`" ou "`Saci`"; a última é a opção *default* e permite que os agentes executem em computadores distribuídos em rede.

O valor que segue `environment` é o nome da classe java que implementa o ambiente. Note que opcionalmente pode-se informar o nome do computador onde o ambiente irá executar. As próximas seções irão explicar como implementar um ambiente.

A palavra reservada `agents` é usada para definir o conjunto de agentes que irão fazer parte do SMA. Um agente é definido inicialmente por seu nome simbólico, seguido, opcionalmente, do nome do arquivo com o código AgentSpeak deste agente; por *default*, o ***Jason*** assume que estes códigos estão em arquivos com o nome `<name>.asl`, onde `<name>` é o nome simbólico do agente. Um número, precedido de `#`, pode ser utilizado para indicar o número de instâncias que deverão ser criadas para o agente. Neste caso os nomes dos agentes terão como sufixo um número sequencial iniciando em 1. Como no `environment`, a definição de um agente pode incluir o nome do computador onde ele irá executar por meio da palavra reservada "`at`". Há também opções que determinam detalhes do funcionamento da arquitetura dada, como o que fazer caso não exista um plano para um evento, e como personalizar a arquitetura dada escrevendo classes Java. Estas opções podem ser consultadas no manual do ***Jason***.

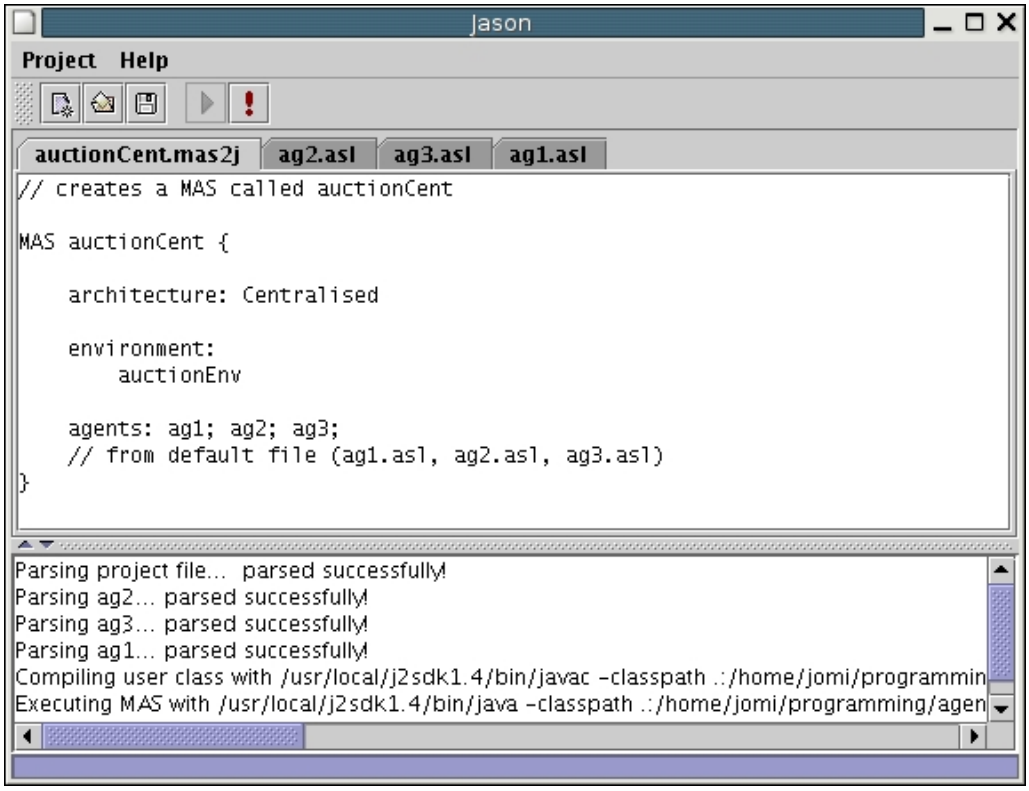


Figura 8: Tela principal do Jason.

Por exemplo, o arquivo que descreve o SMA para o exemplo dos leilões apresentado na seção 5.3.2 é o seguinte[18]:

```
MAS auctionCent {

    architecture: Centralised

    environment:
        auctionEnv

    agents: ag1; ag2; ag3;
}
```

Portanto, o ambiente deve estar definido em uma classe Java chamada `auctionEnv` e o código AgentSpeak dos robôs estão nos arquivos `ag1.asl`, `ag2.asl` e `ag3.asl`. As figuras 8 e 9 mostram a tela principal do ambiente ***Jason*** e o resultado da execução do SMA para este exemplo.

6.3. Definição da Classe do Ambiente

Além de programar o código AgentSpeak dos agentes e prover o arquivo de configuração do SMA, para viabilizar a execução do SMA, é necessário criar o ambiente que os agentes perceberão e atuarão. Isto é feito na forma de um classe Java. Esta classe deve ter pelo menos o seguinte código:

```
import java.util.*;
import jason.*;
```

[18]Outros exemplos, como o dos robôs coletores, são disponíveis na distribuição do ***Jason***.

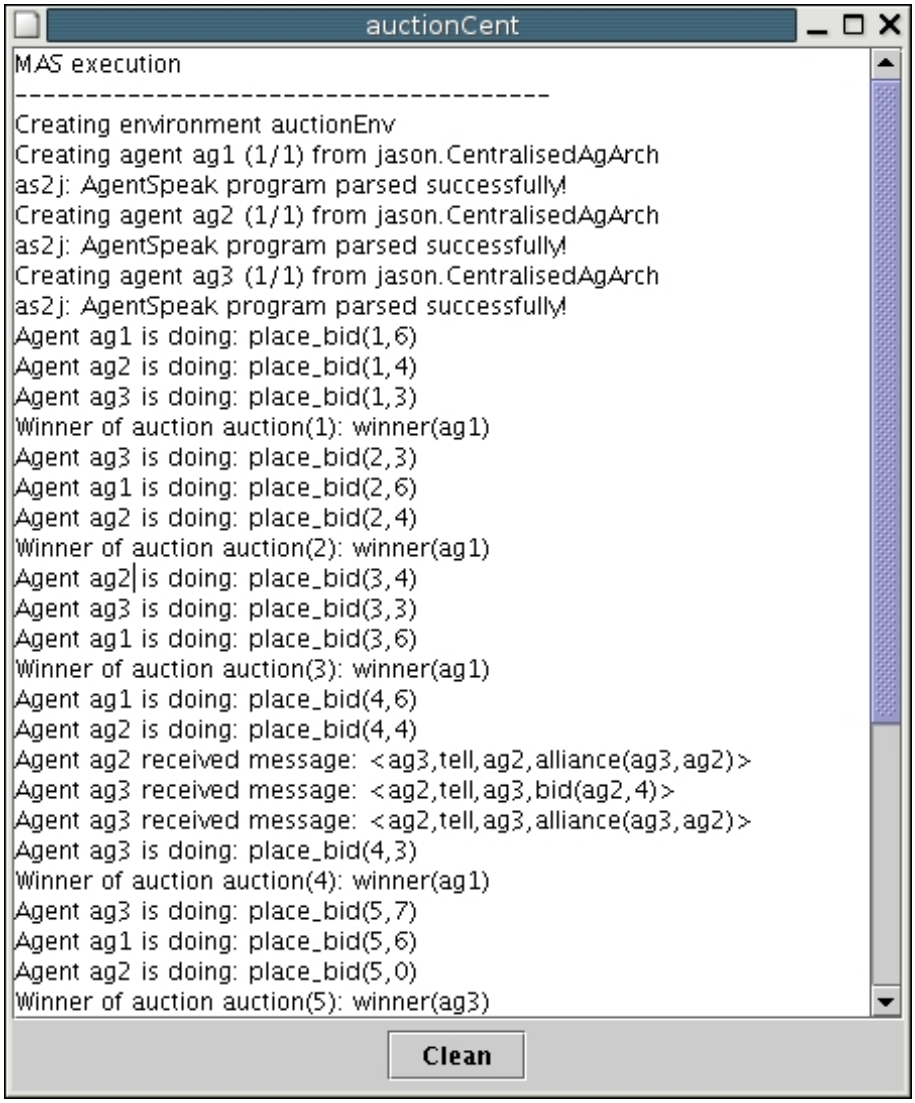


Figura 9: Resultado da execução do SMA no Jason.

```
public class <EnvironmentName> extends Environment {
    public boolean executeAction(String ag, Term act) {
        ...
    }
}
```

onde <EnvironmentName> é o nome da classe ambiente que será coloca no arquivo de configuração do SMA.

A super classe Environment possui o método getPercepts que retornam uma lista onde o programador pode incluir e remover as percepções que os agentes terão. Por exemplo, no caso do exemplo do Leilão descrito na seção 5.3.2, para a percepção inicial dos agentes ser de que existe um leilão, pode-se escrever o seguinte código no construtor da classe:

```
getPercepts().clear();
Term auction = Term.parse("auction(1)");
getPercepts().add(auction);
```

Normalmente, a maior parte do código do ambiente é escrita no método executeAction. Sempre que um agente tenta executar uma ação básica, o nome do agente e um objeto Term representando a ação escolhida pelo agente são passados para este método. Portanto, o código deste método deve verificar se a ação é válida e então realizar o que for necessário para que a ação seja de fato executada. Possivelmente a execução da ação irá alterar as percepções dos agentes. Se este método retornar `true` significa que a ação executou com sucesso.

Segue um exemplo do código que implementa o ambiente dos leilões:

```
import java.util.*;
import jason.*;

public class auctionEnv extends Environment {
  Integer NoBid = new Integer(-1);
  byte    LastAuc  = 8;
  byte    NAg      = 3;
  Term    winner1   = Term.parse("winner(ag1)");
  Term    winner2   = Term.parse("winner(ag2)");
  Term    winner3   = Term.parse("winner(ag3)");
  int     nauc;
  HashMap bid = new HashMap();
  Term    auction, winner;

  public auctionEnv() {
    nauc=1;
    bid.put("ag1",NoBid);
    bid.put("ag2",NoBid);
    bid.put("ag3",NoBid);

    // Percepção inicial
    getPercepts().clear();
    auction = Term.parse("auction("+nauc+")");
    getPercepts().add(auction);
  }

  public boolean executeAction(String ag, Term action) {
    if (action.hasActionSymb("place_bid")) {
```

```
      Integer x = new Integer(action.parameter(2).toString());
      bid.put(ag,x);
    }

    // se todos mandaram ofertas
    if ( ((Integer)bid.get("ag1")).intValue()!=NoBid.intValue() &&
       ((Integer)bid.get("ag2")).intValue()!=NoBid.intValue() &&
       ((Integer)bid.get("ag3")).intValue()!=NoBid.intValue() &&
        nauc<=LastAuc) {
      if (nauc>1)
        getPercepts().remove(winner);
      getPercepts().remove(auction);
      if ( ((Integer)bid.get("ag1")).intValue() >=
           ((Integer)bid.get("ag2")).intValue() &&
           ((Integer)bid.get("ag1")).intValue() >=
           ((Integer)bid.get("ag3")).intValue() ) {
         winner = winner1;
      } else if (((Integer)bid.get("ag2")).intValue() >=
                 ((Integer)bid.get("ag1")).intValue() &&
                 ((Integer)bid.get("ag2")).intValue() >=
                 ((Integer)bid.get("ag3")).intValue() ) {
         winner = winner2;
      } else if (((Integer)bid.get("ag3")).intValue() >=
                 ((Integer)bid.get("ag1")).intValue() &&
                 ((Integer)bid.get("ag3")).intValue() >=
                 ((Integer)bid.get("ag2")).intValue() ) {
         winner = winner3;
      }
      getPercepts().add(winner);
      bid.put("ag1",NoBid);
      bid.put("ag2",NoBid);
      bid.put("ag3",NoBid);
      nauc++;
      System.out.println("Vencedor do leilão "+auction+" é "+winner);
      auction = Term.parse("auction("+nauc+")");
      if (nauc<=LastAuc)
        getPercepts().add(auction);
    }
    return true;
  }
}
```

6.4. Outras Linguagens de Programação Orientadas a Agentes

Desde o artigo seminal em programação orientada a agentes de Shoham (1993), várias outras linguagens nesse paradigma foram propostas, seguindo diferentes abordagens e influências. Naquele artigo, foi apresentada a linguagem Agent0, que é inspirada na arquitetura BDI com sintaxe baseada em LISP. A seguir, serão mencionadas algumas das outras linguagens orientada a agentes que apareceram na literatura de sistemas multiagente desde então.

Concurrent MetateM (Fisher, 1994) é baseada em lógica temporal executável (essa linguagem, de fato, é anterior ao trabalho de Shoham). Originalmente, a idéia era de que cada agente fosse executado diretamente de uma especificação em lógica temporal de tempo linear. Trabalhos mais recentes adicionam outras modalidades à linguagem, permitindo aos desenvolvedores projetar agentes baseados em

abstrações antropomórficas, de maneira similar a agentes BDI (Fisher e Ghidini, 2002). Outros trabalhos em Concurrent METATEM têm permitido aos usuários a especificação de grupos (aninhados) de agentes, com características interessantes de comunicação e acesso para os agentes em diferentes grupos (Fisher et al., 2002).

Uma linguagem recentemente introduzida na literatura da área chama-se STAPLE (Kumar et al., 2002). Essa linguagem é baseada na teoria de intenções conjuntas de Cohen e Levesque, que é formalizada na lógica apresentada em (Cohen e Levesque, 1990). A plataforma IMPACT (Subrahmanian et al., 2000) tem como objetivo permitir a "agentificação" de código legado, e a semântica de agentes IMPACT é baseada em lógica deontológica (Åqvist, 1984). Algumas linguagens de programação apresentadas na literatura estendem sistemas de programação que tem demonstrado grande aplicação em inteligência artificial de modo geral ao conceito de sistemas de múltiplos agentes situados. Um exemplo é a linguagem ConGolog (de Giacomo et al., 2000), uma linguagem de programação concorrente baseada em cálculo de situações. Outro exemplo é $\mathcal{MINERVA}$, uma arquitetura de agente (Leite, 2003; Leite et al., 2002) baseada em programação em lógica dinâmica.

Outras linguagens BDI, tais como 3APL (Hindriks e de Boer, 1998), foram inspiradas em AgentSpeak(L) modificando-a de alguma maneira. 3APL, por exemplo, não inclui o conceito de evento ou de uma estrutura intencional (como o conjunto de intenções em AgentSpeak(L)); isso faz com que 3APL se distancie um pouco da concepção original da arquitetura BDI. Por outro lado, a linguagem tem outras características, tais como regras especiais que operam nos objetivos do agente durante sua execução, que são úteis, por exemplo, no tratamento de planos que falham. Em trabalho recente, a linguagem Dribble (van Riemsdijk et al., 2003) foi proposta, entendendo 3APL com a noção de objetivos declarativos. Outra abordagem a agentes BDI é o cálculo Ψ (Kinny, 2002a), baseado no cálculo π (Milner et al., 1992), com o objetivo de dar suporte teórico para linguagens de programação visuais (Kinny, 2002a,b).

Uma linguagem fortemente baseada em AgentSpeak(L), chamada AgentTalk, foi recentemente disponibilizada pela Internet[19]. Também encontram-se disponíveis na Internet, plataformas JAVA para o desenvolvimento de agentes BDI, tais como JAM[20] (Huber, 1999) e JACK[21] (Howden et al., 2001). A grande diferença entre essas ferramentas e linguagens BDI como AgentSpeak(L) e 3APL é que essas últimas têm semântica formal, o que possibilita a verificação formal de sistemas programados com estas linguagens; além disto, pode-se estabelecer, com rigorismo ainda maior, uma relação formal com a teoria BDI original, vinda da literatura de filosofia sobre raciocínio prático e que influencia todo o trabalho em agentes racionais (através de trabalhos teóricos que estão em desenvolvimento).

7. Conclusão

Após uma década desde a primeira publicação utilizando o termo "programação orientada a agentes" (Shoham, 1993), pode-se dizer que as linguagens de programação orientadas a agentes encontram-se ainda em sua infância. Muita pesquisa em aspectos formais dessas linguagens já foi desenvolvida, mas ainda há muitas questões em aberto. Além disto, só recentemente estas linguagem com base formal e realmente

[19]Essa linguagem tem um interpretador escrito na linguagem funcional Scheme. A única referência a esse trabalho é a URL: <http://www.cs.rmit.edu.au/~winikoff/agenttalk/>.

[20]URL: <http://www.marcush.net>.

[21]URL: <http://www.agent-software.com>.

substanciadas nas idéias da área de sistemas multiagente tiveram interpretadores disponibilizados publicamente. Com isto, pouca experimentação prática, com sistemas de grande porte, foi desenvolvida até o momento. As relações com os demais níveis do processo de engenharia de software orientada a agente também não estão bem consolidadas ainda.

De toda a forma, dado o grande sucesso recente da área de sistemas multiagente, influenciando o trabalho nas mais diversas área da computação, é muito provável que o próximo grande paradigma de programação seja o paradigma de orientação a agentes, alterando substancialmente a forma como sistemas computacionais serão concebidos. As linguagens de programação com esta orientação certamente terão um papel importante nesse processo, mas exatamente a forma como isto acontecerá é difícil prever. O objetivo deste texto foi dar uma visão geral de alguns conceitos que serão importantes nesse paradigma, e fornecer referências para os trabalhos na área, esperando que as idéias apresentadas aqui ajudem também na compreensão da significativa diferença desta forma de concepção de sistemas computacionais na concepção de entidades de software auto-motivadas e autônomas.

Referências

Allen Emerson, E. (1990). Temporal and modal logic. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier Science, Amsterdam.

Alvares, L. O. e Sichman, J. S. (1997). Introdução aos sistemas multiagentes. In Medeiros, C. M. B., editor, *Jornada de Atualização em Informática (JAI'97)*, chapter 1, pages 1–38. UnB, Brasília.

Åqvist, L. (1984). Deontic logic. In Gabbay, D. M. e Günthner, F., editors, *Handbook of Philosophical Logic: Volume II: Extensions of Classical Logic*, chapter II.11, pages 605–714. D. Reidel Publishing Company.

Austin, J. L. (1975). *How to do things with Words*. Harvard University Press, Cambridge, 2 edition.

Bates, J. (1994). The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125.

Bond, A. H. e Gasser, L., editors (1988). *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA.

Bordini, R. H. (1994). Suporte lingüístico para migração de agentes. Dissertação de mestrado, Instituto de Informática/UFRGS, Porto Alegre.

Bordini, R. H. (1999). *Contributions to an Anthropological Approach to the Cultural Adaptation of Migrant Agents*. These (ph.d.), University College London, London.

Bordini, R. H., Bazzan, A. L. C., Jannone, R. O., Basso, D. M., Vicari, R. M., e Lesser, V. R. (2002). AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In Castelfranchi, C. e Johnson, W. L., editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15–19 July, Bologna, Italy*, pages 1294–1302, New York, NY. ACM Press.

Bordini, R. H., Fisher, M., Pardavila, C., e Wooldridge, M. (2003a). Model checking AgentSpeak. In Rosenschein, J. S., Sandholm, T., Wooldridge, M., e Yokoo, M., editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, 14–18 July*, pages 409–416, New York, NY. ACM Press.

Bordini, R. H., Fisher, M., Visser, W., e Wooldridge, M. (2003b). Verifiable multi-agent programs. In *Proceedings of the First International Workshop on Program-*

ming Multiagent Systems: languages, frameworks, techniques and tools (ProMAS-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia. To appear as a volume in Springer's LNAI series.

Bordini, R. H. e Moreira, Á. F. (2002). Proving the asymmetry thesis principles for a BDI agent-oriented programming language. In Dix, J., Leite, J. A., e Satoh, K., editors, *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02), 1st August, Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science 70(5). Elsevier. URL: <http://www.elsevier.nl/locate/entcs/volume70.html>. CLIMA-02 was held as part of FLoC-02. This paper was originally published in Datalogiske Skrifter number 93, Roskilde University, Denmark, pages 94–108.

Bordini, R. H. e Moreira, Á. F. (2004). Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3). Special Issue on Computational Logic in Multi-Agent Systems, to appear.

Bordini, R. H., Okuyama, F. Y., de Oliveira, D., Drehmer, G., e Krafta, R. C. (2004). The MAS-SOC approach to multi-agent based simulation. In Lindemann, G., Moldt, D., e Paolucci, M., editors, *Proceedings of the First International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02), 16 July, 2002, Bologna, Italy (held with AAMAS02) — Revised Selected and Invited Papers*, number 2934 in Lecture Notes in Artificial Intelligence, pages 70–91, Berlin. Springer-Verlag.

Bordini, R. H. e Vieira, R. (2003). Linguagens de programação orientadas a agentes: uma introdução baseada em AgentSpeak(L). *Revista de Informática Teórica e Aplicada*, X(1):7–38. Instituto de Informática da UFRGS, Brazil.

Bordini, R. H., Vieira, R., e Moreira, Á. F. (2001). Fundamentos de sistemas multiagentes. In Ferreira, C. E., editor, *Jornada de Atualização em Informática (JAI'01)*, volume 2, chapter 1, pages 3–44. SBC, Fortaleza, Brasil.

Bordini, R. H., Visser, W., Fisher, M., Pardavila, C., e Wooldridge, M. (2003c). Model checking multi-agent programs with CASP. In Hunt Jr., W. A. e Somenzi, F., editors, *Proceedgins of the Fifteenth Conference on Computer-Aided Verification (CAV-2003), Boulder, CO, 8–12 July*, number 2725 in Lecture Notes in Computer Science, pages 110–113, Berlin. Springer-Verlag. Tool description.

Bratman, M. E. (1987). *Intentions, Plans and Practical Reason*. Harvard University Press, Cambridge, MA.

Bratman, M. E., Israel, D. J., e Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355.

Briot, J.-P. e Demazeau, Y., editors (2002). *Principes et architecture des systèmes multi-agents*. Hermes, Paris.

Campbell, J. A. e d'Inverno, M. (1990). Knowledge interchange protocols. In Demazeau, Y. e Müller, J.-P., editors, *Decentralized A.I.—Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'89), 16–18 August, Cambridge, 1989*, pages 63–80. Elsevier Science B.V., Amsterdam.

Castelfranchi, C. (1990). Social power: A point missed in multi-agent, DAI and HCI. In Demazeau, Y. e Müller, J.-P., editors, *Decentralized Artificial Intelligence*. Elsevier, Amsterdam.

Castelfranchi, C. (1997). The theory of social functions — challenges for agent-based social simulation. In *Workshop on Simulating Societies, held as part of the International Conference on Computer Simulation and the Social Sciences (ICCS&SS)*, Cortona, Italy. Draft paper from an Invited Talk.

Castelfranchi, C. e Johnson, W. L., editors (2002). *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2002)*. First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2002), ACM Press.

Cohen, P. R. e Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261.

da Rocha Costa, A. C., Hübner, J. F., e Bordini, R. H. (1994). On entering an open society. In *Anais do XI Simpósio Brasileiro de Inteligência Artificial*, pages 535–546, Fortaleza. `http://www.inf.furb.br/~jomi/pubs/1994/sbia/Rocha-sbia94.ps`.

de Giacomo, G., Lespérance, Y., e Levesque, H. J. (2000). ConGolog: A concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169.

Demazeau, Y., editor (1998). *Third International Conference on Multi-Agent Systems (ICMAS'98), Agents' World, 4–7 July, Paris*, Washington. IEEE Computer Society Press.

Demazeau, Y. e Müller, J.-P., editors (1990). *Decentralized Artificial Intelligence*. Elsevier, Amsterdam.

Dennett, D. C. (1987). *The Intentional Stance*. The MIT Press, Cambridge, MA.

d'Inverno, M., Kinny, D., Luck, M., e Wooldridge, M. (1998). A formal specification of dMARS. In Singh, M. P., Rao, A. S., e Wooldridge, M., editors, *Intelligent Agents IV—Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97), Providence, RI, 24–26 July, 1997*, number 1365 in Lecture Notes in Artificial Intelligence, pages 155–176. Springer-Verlag, Berlin.

d'Inverno, M. e Luck, M. (1998). Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):1–27.

Durfee, E., editor (1996). *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS'96), 11–13 December, Kyoto, Japan*, Menlo Park, CA. AAAI Press.

Durfee, E., editor (2000). *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000), 10–12 July, Boston, MA*, Los Alamitos, CA. IEEE Computer Society.

Ferber, J. (1999a). *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, London.

Ferber, J. (1999b). *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley.

Fisher, M. (1994). A survey of concurrent METATEM—the language and its applications. In Gabbay, D. M. e Ohlbach, H. J., editors, *Temporal Logics—Proceedings of the First International Conference*, number 827 in Lecture Notes in Artificial Intelligence, pages 480–505. Springer-Verlag, Berlin.

Fisher, M. e Ghidini, C. (2002). The ABC of rational agent modelling. In Castelfranchi, C. e Johnson, W. L., editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15–19 July, Bologna, Italy*, pages 849–856, New York, NY. ACM Press.

Fisher, M., Ghidini, C., e Hirsch, B. (2002). Organising logic-based agents. In *Proceedings of the Second NASA/IEEE Goddard Workshop on Formal Approaches to Agent Based Systems (FAABS 2002), Greenbelt, MD, October 29-31*.

Franklin, S. e Graesser, A. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In Müller, J. P., Wooldridge, M. J., e Jennings, N. R., editors, *Intelligent Agents III—Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-*

96), ECAI'96 Workshop, Budapest, Hungary, number 1193 in Lecture Notes in Artificial Intelligence, pages 21–35. Springer-Verlag, Berlin. URL: `http://www.msci.memphis.edu/~franklin/aagents.html`.

Garijo, F., Gómes-Sanz, J. J., Pavón, J., e Massonet, P. (2001). Multi-agent system organization: An engineering prespective. In *Pre-Proceeding of the 10th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW'2001)*.

Gasser, L. e Huhns, M. N., editors (1989). *Distributed Artificial Intelligence*, volume II of *Research Notes in Artificial Intelligence*. Pitman / Morgan Kaufmann, London / San Mateo, CA.

Genesereth, M. R. e Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, 37(7):48–53.

Georgeff, M. P. e Lansky, A. L. (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87), 13–17 July, 1987, Seattle, WA*, pages 677–682, Manlo Park, CA. AAAI Press / MIT Press.

Gilbert, N. e Conte, R., editors (1995). *Artificial Societies: The Computer Simulation of Social Life*. UCL Press, London.

Gilbert, N. e Doran, J., editors (1994). *Simulating Society: The Computer Simulation of Social Phenomena*. UCL Press, London.

Green, S., Hurst, L., Nangle, B., Cunningham, P., Somers, F., e Evans, R. (1997). Software agents: A review. Technical Report TCD-CS-1997-06, Trinity College, University of Dublin, Dublin.

Hayes-Roth, B., Brownston, L., e van Gent, R. (1995). Multiagent collaboration in directed improvisation. In Lesser, V. e Gasser, L., editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), 12–14 June, San Francisco, CA*, pages 148–154, Menlo Park, CA. AAAI Press / MIT Press.

Hindriks, K. V. e de Boer, F. S. a. (1998). Formal semantics for an abstract agent programming language. In Singh, M. P., Rao, A. S., e Wooldridge, M., editors, *Intelligent Agents IV—Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97), Providence, RI, 24–26 July, 1997*, number 1365 in Lecture Notes in Artificial Intelligence, pages 215–229, Berlin. Springer-Verlag.

Holzmann, G. J. (1991). *Design and Validation of Computer Protocols*. Prentice Hall.

Howden, N., Rönnquist, R., Hodgson, A., e Lucas, A. (2001). JACK intelligent agents™ — summary of an agent infrastructure. In *Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, held with the Fifth International Conference on Autonomous Agents (Agents 2001), 28 May – 1 June, Montreal, Canada*.

Huber, M. J. (1999). JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents-99), 1–5 May, Seattle, WA*, pages 236–243. ACM Press.

Hübner, J. F. (1995). Migração de agentes em sistemas multi-agentes abertos. Dissertação de mestrado, CPGCC da UFRGS, Porto Alegre. `http://www.inf.furb.br/~jomi/pubs/dissertacao/dissertacao.pdf`.

Hübner, J. F. e Sichman, J. S. (2000). SACI: Uma ferramenta para implementação e monitoração da comunicação entre agentes. In Monard, M. C. e Sichman, J. S., editors, *Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (Open Discussion Track)*, pages 47–56, São Carlos. ICMC/USP. `http://www.inf.furb.br/~jomi/pubs/2000/Hubner-iberamia2000.pdf`.

Huhns, M. N., editor (1987). *Distributed Artificial Intelligence*. Research Notes in Artificial Intelligence. Pitman / Morgan Kaufmann, London / San Mateo, CA.

Huhns, M. N. e Singh, M., editors (1997). *Readings in Agents*. Morgan Kaufmann, San Mateo, CA.

Iglesias, C., Garrijo, M., e Gonzalez, J. (1999a). A survey of agent-oriented methodologies. In *"Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories*, pages 317–330. "Springer-Verlag: Heidelberg.

Iglesias, C. A., Garijo, M., e González, J. C. (1999b). A survey of agent-oriented methodologies. In Müller, J. P., Singh, M. P., e Rao, A. S., editors, *Intelligent Agents V—Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98), held as part of the Agents' World, Paris, 4–7 July, 1998*, number 1555 in Lecture Notes in Artificial Intelligence, pages 317–330, Heidelberg. Springer-Verlag.

Jennings, N. R., Sycara, K. P., e Wooldridge, M. J. (1998). A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38.

Jennings, N. R. e Wooldridge, M. J. (1998). *Agent Technology: foundations, applications, and markets*. Springer Verlag, London.

Johnson, S. (2001). *Emergence: the connected lives of ants, brains, cities, and software*. Touchstone, New York.

Kinny, D. (1993). The distributed multi-agent reasoning system architecture and language specification. Technical report, Australian Artificial Intelligence Institute, Melbourne, Australia.

Kinny, D. (2002a). The Ψ calculus: An algebraic agent language. In Meyer, J.-J. e Tambe, M., editors, *Intelligent Agents VIII – Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1–3, 2001, Seattle, WA*, number 2333 in Lecture Notes in Artificial Intelligence, pages 32–50, Berlin. Springer-Verlag.

Kinny, D. (2002b). ViP: A visual programming language for plan execution systems. In Castelfranchi, C. e Johnson, W. L., editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002, featuring 6th AGENTS, 5th ICMAS and 9th ATAL), 15–19 July, Bologna, Italy*, pages 721–728, New York, NY. ACM Press.

Kumar, S., Cohen, P. R., e Huber, M. J. (2002). Direct execution of team specifications in STAPLE. In Castelfranchi, C. e Johnson, W. L., editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15–19 July, Bologna, Italy*, pages 567–568, New York, NY. ACM Press. Short paper.

Leite, J. A. (2003). *Evolving Knowledge Bases: Specification and Semantics*, volume 81 of *Frontiers in Artificial Intelligence and Applications, Dissertations in Artificial Intelligence*. IOS Press/Ohmsha, Amsterdam.

Leite, J. A., Alferes, J. J., e Pereira, L. M. (2002). $\mathcal{MINERVA}$—a dynamic logic programming agent architecture. In Meyer, J.-J. e Tambe, M., editors, *Intelligent Agents VIII – Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1–3, 2001, Seattle, WA*, number 2333 in Lecture Notes in Artificial Intelligence, pages 141–157, Berlin. Springer-Verlag.

Lesser, V. e Gasser, L., editors (1995). *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), 12–14 June, San Francisco, CA*, Menlo Park, CA. AAAI Press / MIT Press.

Machado, R. e Bordini, R. H. (2002). Running AgentSpeak(L) agents on SIM_AGENT. In Meyer, J.-J. e Tambe, M., editors, *Intelligent Agents VIII –*

Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1–3, 2001, Seattle, WA, number 2333 in Lecture Notes in Artificial Intelligence, pages 158–174, Berlin. Springer-Verlag.

Milner, R., Parrow, J., e Walker, D. (1992). A calculus for mobile processes (parts I and II). *Information and Computation*, 100(1):1–40 and 41–77.

Moreira, Á. F. e Bordini, R. H. (2002). An operational semantics for a BDI agent-oriented programming language. In Meyer, J.-J. C. e Wooldridge, M. J., editors, *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), April 22–25, Toulouse, France*, pages 45–59.

Moreira, Á. F., Vieira, R., e Bordini, R. H. (2003). Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia*. To appear as a volume in Springer's LNAI series.

Moss, S. J. e Davidsson, P., editors (2001). *Second International Workshop on Multi-Agent-Based Simulation (MABS 2000), 8–9 July, Boston, MA*, number 1979 in Lecture Notes in Computer Science, Berlin. Springer-Verlag. Revised Papers.

Müller, J. P. (1999). The right agent (architecture) to do the right thing. In Müller, J. P., Singh, M. P., e Rao, A. S., editors, *Intelligent Agents V—Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98), held as part of the Agents' World, Paris, 4–7 July, 1998*, number 1555 in Lecture Notes in Artificial Intelligence, pages 211–225, Heidelberg. Springer-Verlag.

Odell, J., Parunak, H. V. D., e Bauer, B. (2000). Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*.

Parunak, V. (1999). Industrial and practical applications of DAI. In Weiß, G., editor, *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*, chapter 9. MIT Press, Cambridge, MA.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, Aarhus.

Prietula, M., Carley, K., e Gasser, L., editors (1998). *Simulating Organizations: Computational Models of Institutions and Groups*. AAAI Press / MIT Press, Menlo Park, CA.

Rao, A. S. (1996a). AgentSpeak(L): BDI agents speak out in a logical computable language. In Van de Velde, W. e Perram, J., editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in Lecture Notes in Artificial Intelligence, pages 42–55, London. Springer-Verlag.

Rao, A. S. (1996b). Decision procedures for propositional linear-time belief-desire-intention logics. In Wooldridge, M., Müller, J. P., e Tambe, M., editors, *Intelligent Agents II—Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages (ATAL'95), held as part of IJCAI'95, Montréal, Canada, August 1995*, number 1037 in Lecture Notes in Artificial Intelligence, pages 33–48, Berlin. Springer-Verlag.

Rao, A. S. e Georgeff, M. P. (1992). An abstract architecture for rational agents. In Rich, C., Swartout, W. R., e Nebel, B., editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92), 25–29 October, Cambridge, MA*, pages 439–449, San Mateo, CA. Morgan Kaufmann.

Rao, A. S. e Georgeff, M. P. (1995). BDI agents: From theory to practice. In Lesser, V. e Gasser, L., editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), 12–14 June, San Francisco, CA*, pages 312–319, Menlo Park, CA. AAAI Press / MIT Press.

Rao, A. S. e Georgeff, M. P. (1998). Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343.

Rosenschein, J. S., Sandholm, T., Michael, W., e Yokoo, M., editors (2003). *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.

Searle, J. R. e Vanderveken, D. (1985). *Foundations of illocutionary logic*. Cambridge University Press, Cambridge.

Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60:51–92.

Sichman, J. S. (1995). *Du Raisonnment Social Chez les Agents: Une Approche Fondeé sur la Théorie de la Dépendance*. Thèse (doctorat), Institut National Polytechnique de Grenoble.

Sichman, J. S., Conte, R., e Gilbert, N., editors (1998). *Multi-Agent Systems and Agent-Based Simulation, Proceedings of the International Workshop held as part of the Agents' World, Paris, 4–7 July*, number 1534 in Lecture Notes in Artificial Intelligence, Berlin. Springer-Verlag. To appear in December 1998.

Singh, M. P., Rao, A. S., e Georgeff, M. P. (1999). Formal methods in DAI: Logic-based representation and reasoning. In Weiß, G., editor, *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*, chapter 8, pages 331–376. MIT Press, Cambridge, MA.

Smith, R. G. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transaction on Computers*, 29(12):1104–1113.

Subrahmanian, V. S., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Ozcan, F., e Ross, R. (2000). *Heterogeneous Agent Systems*. The MIT Press.

Sycara, K. P. (1998). Multiagent systems. *AI Magazine*, 19(2):79–92.

Torres, J. A. R., Nedel, L. P., e Bordini, R. H. (2003). Using the BDI architecture to produce autonomous characters in virtual worlds. In Rist, T., Aylett, R., Ballin, D., e Rickel, J., editors, *Proceedings of the Fourth International Conference on Interactive Virtual Agents (IVA 2003), Irsee, Germany, 15–17 September*, number 2792 in Lecture Notes in Artificial Intelligence, pages 197–201, Heidelberg. Springer-Verlag. Short paper.

van Riemsdijk, B., van der Hoek, W., e Meyer, J.-J. C. (2003). Agent programming in dribble: from beliefs to goals using plans. In Rosenschein, J. S., Sandholm, T., Michael, W., e Yokoo, M., editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, 14–18 July*, pages 393–400, New York, NY. ACM Press.

Visser, W., Havelund, K., Brat, G., e Park, S. (2000). Model checking programs. In *Proceedings of the Fifteenth International Conference on Automated Software Engineering (ASE'00), 11-15 September, Grenoble, France*, pages 3–12. IEEE Computer Society.

Weiß, G., editor (1997). *Distributed Artificial Intelligence Meets Machine Learning—Learning in Multiagent Environments*. Number 1221 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin.

Weiß, G., editor (1999a). *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA.

Weiß, G., editor (1999b). *Multiagent Systems: A modern approach to distributed artificial intelligence*. MIT Press, London.

Wooldridge, M. (1998). Agent-based computing. *Interoperable Communication Networks*, 1(1):71–97.

Wooldridge, M. (1999). Intelligent agents. In Weiß, G., editor, *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 27–77. MIT Press, Cambridge, MA.

Wooldridge, M. (2000). *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley and Sons.

Wooldridge, M. e Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152. URL: `http://www.elec.qmw.ac.uk/dai/pubs`.

Wooldridge, M., Jennings, N. R., e david Kinny (1999). A methodology for agent-oriented analysis and design. In *Proceedings of the Third International Conference on Autonomous Agentes (Agent's 99)*. ACM.

Wooldridge, M., Müller, J. P., e Tambe, M. (1996). Agent theories, architectures, and languages: A bibliography. In Wooldridge, M., Müller, J. P., e Tambe, M., editors, *Intelligent Agents II—Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages (ATAL'95), held as part of IJCAI'95, Montréal, Canada, August 1995*, number 1037 in Lecture Notes in Artificial Intelligence, pages 408–431, Berlin. Springer-Verlag.