PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TOLERÂNCIA A FALHAS COM UM MODELO DE AGENTES

JULIANA FONSECA ANTUNES

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Dr. Avelino Francisco Zorzo

Porto Alegre
Dezembro/2009

Dados Internacionais de Catalogação na Publicação (CIP)

Antunes, Juliana Fonseca

Tolerância a falhas com um modelo de agentes / Juliana Fonseca Antunes. – Porto Alegre, 2009.

82 f.

Diss. (Mestrado) – Fac. De Informática, PUCRS.
Orientador: Prof. Dr. Avelino Francisco Zorzo

1. Agentes BDI. 2. Sistemas Multiagentes.
2. Tolerância a Falhas

CDD 006.3

Ficha Catalográfica elaborada pelo Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "Tolerância à Falhas com um Modelo de agentes", apresentada por Juliana Fonseca Antunes, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Dstribuído, aprovada em 17/12/09 pela Comissão Examinadora:

Prof. Dr. Avelino Francisco Zorzo – Orientador — PPGCC/PUCRS

Prof. Dr. Eduardo Augusto Bezerra - — PPGCC/PUCRS

Prof. Dr. Fabian Luis Vargas – — PPGEE/PUCRS

Homologada em. 25./11...../10....., conforme Ata No. 23/10... pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

Dedico esta dissertação ao meu amor Rothschild, meu marido e grande amigo, a minha linda e amada filha Júlia, aos meus pais, Mananciel e Lêda, e a minha sogra, Cacilda, pela ajuda incondicional.

AGRADECIMENTOS

À Deus, por ter me mantido no meu caminho, demonstrando minha fé e perseverança em trilhar esse caminho.

Ao meu esposo Rothschild, que não poupou esforços para que eu conseguisse entrar e continuar no mestrado, não foi tarefa fácil, a distância e o tempo gerou uma grande saudade, mas enfim nós conseguimos superar todos obstáculos surgidos. A indiscutível força que sempre me passa, fez com que alcançasse mais essa conquista.

Ao Professor Dr. Avelino Zorzo pela paciência, orientação e pela grande oportunidade de aprendizado. Professor, obrigada pelo seu tempo de ensino que foi fundamental para eu conseguir terminar o mestrado. Por sempre me receber muito bem e com um sorriso sempre, por me orientar em caminhos desconhecidos, onde aprendi muito.

A meus pais, Mananciel e Lêda, que batalharam muito para dar, à mim e aos meus irmãos, um futuro promissor, em especial a minha mãe que me acompanhou em quase todas as viagens, para que pudesse ficar com minha filha por alguns minutos à noite depois de um longo dia de estudo.

A minha filha Júlia, que desde os três meses de idade me acompanha, desde a seleção no mestrado, a todas as viagens a Barra do Bugres-MT e Porto Alegre-RS, entendia quando eu falava que iria estudar e que não demoraria, me recebendo com uma alegria e um sorriso lindo que só uma filha pode dar a uma mãe depois de um dia de estudo. Ter você perto de mim em todos esses momentos foi muito importante para ter conseguido finalizar mais essa fase de minha vida.

Aos meus filhos, Anderson e Natália, que quando eu viajava, cuidavam muito do pai, com muito carinho, dedicação e amor, principalmente minha linda, Natália, que está se transformando em uma linda moça, responsável, compreensiva e inteligente.

A minha sogra Cacilda, que cuidava da minha caçulinha para que eu pudesse estudar, me acompanhando sempre quando foi preciso (enfrentando o frio de Porto Alegre! Mesmo não gostando do frio), com um apoio incondicional.

A Jomi Hubner pelo grande auxílio na ferramenta Jason, sempre respondendo prontamente minhas dúvidas e questionamentos.

Ao Professores e colegas do MINTER PUCRS/UNEMAT pela disponibilidade e auxílio, em especial a grande e querida amiga Estela.

TOLERÂNCIA A FALHAS COM UM MODELO DE AGENTES

RESUMO

Atualmente, os computadores são utilizados nas mais diversas áreas do conhecimento humano e são imprescindíveis em várias atividades fundamentais na sociedade. Particularmente em aplicações industriais, o sistema computacional têm que ser confiável e tolerante à falhas, ou seja, permitir que o sistema permaneça operando mesmo na presença de falhas. Desta forma, esta dissertação tem como objetivo descrever um modelo de agentes tolerantes a falhas. A fim de atingir tal objetivo foi feito um estudo de conceitos fundamentais de arquiteturas de agentes deliberativos baseados em estados mentais, esses descritos por crenças, desejos e intenções (modelo Belief Desire Intention), que podem ser implementados utilizando a linguagem de programação orientada a agentes AgentSpeak e o mecanismo de tolerância a falhas de interação multiparticipante confiável (Dependable Multiparty Interaction). Através destes conceitos é implementado um sistema composto por vários elementos computacionais interativos, denominados agentes, que interagem com outros agentes, formando um sistema multiagentes que são implementadas utilizando o interpretador Jason de linguagem AgentSpeak. Esse interpretador se comunica com o simulador da Célula de Produção FZI, escalonando o trabalho da célula de produção através das crenças e desejos, re-escalonando o trabalho se ocorrer alguma mudança do estado do sistema. A implementação de um agente tolerante a falhas permite que o sistema execute ações concorrentemente mesmo na presença de falhas, pois o mecanismo DMI gera uma interação multiparticipante entre diversos dispositivos que permite o tratamento de falhas concorrentes. A interação multiparticipante é criada pelos agentes conforme as percepções adquiridas no simulador da célula de produção, gerando ações que são enviadas ao simulador, conseqüentemente, alterando o estado dos dispositivos.

Palavras Chave: Tolerância a falhas, Interações Multiparticipantes Confiáveis, Sistemas Multiagentes, modelo BDI.

FAULT TOLERANCE TO A MODEL OF AGENTS

ABSTRACT

Nowadays, the computers have been used in most varied areas of the human knowledge and are indispensable in many essential activities in the society. Specifically in the context of the industrial applications the computational system must be dependable and fault-tolerance, that is, it must allow that the system remains keep on working when the faults occur. Based on this idea, this study aimed at describing a model for faults-tolerance agents. In order to achieve this objective, a study of essentials architecture concepts of deliberative agents based on mental states, described by beliefs, desires and intentions (Belief, Desire and Intention model) was carried out, these can be implemented using the agent-oriented programming language AgentSpeak and the mechanism of faults tolerance of dependable multiparty interaction (DMI). By means of these concepts, a system composed of various elements of interactive computer was implemented, these elements are called agents, because they interact with other agents, forming a multi-agent system that is implemented using the interpreter Jason of AgentSpeak language. This interpreter communicates with the simulator of FZI Production Cell, scheduling the cell production work through the beliefs and desires, re scheduling the task if occurs changes in the system state. The implementation of fault-tolerance agent allows the system executes concurrently actions even in the presence of faults because the mechanism DMI generates a multiparty interaction among the varied devices that permit the handling of concurrent faults. The multiparty interaction is created by the agents in conformity to perceptions acquired in the simulator of the cell production that is generating actions which are sent to the simulator and consequently are modifying the state of the devices.

Keywords: Fault Tolerance, Dependable Mutiparty Interaction, Multiagents System, BDI model.

LISTA DE ABREVIATURAS

CAA - Coordinated Atomic Actions

BDI - Belief, Desire, Intentions – Crença, Desejo, Intenção

CAA - Coordinated Atomic Actions – Ações Atômicas Coordenadas

DMI - Dependable Multiparty Interaction – Interação Multiparticipante Confiável

FZI - Forschungszentrum Informatik

IA - Inteligência Artificial

IDE - Integrated Development Environment – Ambiente de desenvolvimento integrado

IRMA - Intelligent Resource-bounded Machine Architecture – Arquitetura de máquina vinculado a Recursos inteligentes.

JASON - **J**ava-based **A**gentSpeak interpreter used with **S**ACI for multi-agent distribution **o**ver the **n**et Interpretador AgentSpeak baseado em Java e usado com SACI para sistemas multiagentes distribuídos em rede

PRS - Procedural Reasoning System – Sistema de Raciocínio Procedural

SACI - Simple Agent Communication Infrastructure – Infraestrutura de Comunicação entre agentes

SMA - Sistema Multiagente

X-BDI - EXecutable BDI – BDI Executável

LISTA DE FIGURAS

Figura 1.1: Características da dependabilidade....19
Figura 2.1: Agente em seu ambiente [WEI99]....26
Figura 2.2: Agente e ambiente [BOR07]....27
Figura 2.3: Estrutura de um Sistema Multiagentes [BOR07]....29
Figura 2.4: Arquitetura BDI [ZAM01]....32
Figura 2.5: Componentes de uma arquitetura BDI [WEI99]....34
Figura 3.1: Exemplo de plano em AgentSpeak....40
Figura 3.2: Tipos de termos AgentSpeak em Jason [BOR07]....42
Figura 3.3: Ciclo de interpretação de um programa em *AgentSpeak(L)* [MAC01]....45
Figura 3.4: Gramática de arquivo *.mas2j* [HÜB04]....47
Figura 3.5: Gramática aceita pelo Jason [HÜB04]....49
Figura 3.6: Arquivo de configuração MAS quarto....50
Figura 3.7: Planos dos agentes Porteiro, Claustrofóbico e Paranóico....51
Figura 3.8: Resultado da execução do SMA no Jason....52
Figura 3.9: Código do *ambienteQuarto.java*....53
Figura 3.10: Defeito de plano de **a**. (a) uma intenção antes do plano apresentar defeito; (b) a intenção depois do plano apresentar defeito [BOR07]....55
Figura 4.1: Interação multiparticipantes confiável (DMI) [ZOR99a]....59
Figura 4.2: Framework DMI [ZOR99a]....61
Figura 4.3: Criando Gerentes [ZOR99a]....62
Figura 5.1: Modelo de um agente BDI tolerante a falhas....65
Figura 5.2: Célula de produção tolerante a falhas [ZOR99a]....67
Figura 5.3: Célula de Produção [ZOR99a]....68
Figura 5.4: Modelo de agente para controle da Célula de Produção....69
Figura 5.5: Estrutura e descrição do Projeto ProjCP na Ferramenta Jason....69
Figura 5.6: (a) chama a ação no agente e (b)descrição e definição da ação no *ambCP*....70
Figura 5.7: Adição da Crença ao Agente Traffic Light1....71
Figura 5.8: AgEnvironment com suas crenças e objetivos....71
Figura 5.9: AgTrafficLight1 com suas crenças e objetivos....72
Figura 5.10: AgFeedBelt com suas crenças e objetivos....73
Figura 5.11: AgTable com suas crenças e objetivos....74
Figura 5.12: AgRobot com suas crenças e objetivos....75
Figura 5.13: AgPress1 com suas crenças e objetivos....76
Figura 5.14: AgDepositBelt com suas crenças e objetivos....77
Figura 5.15: Legenda dos diagramas para agente, crença, objetivos e ações....77
Figura 5.16: Agentes da DMI Load Cell....78

Figura 5.17: DMI Load Table.....80
Figura 5.18: Crenças que ativam a ação feedBeltLT no ambCP do agente FeedBelt 80
Figura 5.19: Crenças que ativam a ação tableLT no ambCP do agente Table.....81
Figura 5.20: Crenças que ativam a ação trafficLightLT no ambCP do agente TrafficLight1 81

Sumário

CAPÍTULO 1 INTRODUÇÃO 19

CAPÍTULO 2 MODELO BDI 25

2.1 – AGENTES INTELIGENTES 25
2.1.1 – Sistemas Multiagentes 28
2.1.2 – Ambientes 29
2.2 – ESTADOS MENTAIS 30
2.2.1 – Crenças 31
2.2.2 – Desejos 31
2.2.3 – Intenções 31
2.3 – ARQUITETURA BDI 31
2.4 – IMPLEMENTAÇÕES DA ARQUITETURA BDI 34
2.5 – FERRAMENTAS BDI 35
2.5.1 – X-BDI 35
2.5.2 – JASON 36
2.6 – CONSIDERAÇÕES FINAIS 38

CAPÍTULO 3 LINGUAGEM AGENTSPEAK(L) 39

3.1 – INTERPRETADOR JASON 40
3.1.1 – Crenças 41
3.1.2 – Objetivos 43
3.1.3 – Planos 43
3.1.4 – Ciclo de Raciocínio 44
3.1.5 – Sintaxe SMA 47
3.1.6 – Sintaxe de Agentes 48
3.1.6 – Exemplo de um Projeto utilizando a Ferramenta Jason 50
3.2 - DEFEITO DE PLANOS 53

CAPÍTULO 4 INTERAÇÕES MULTIPARTICIPANTES CONFIÁVEIS 57

4.1 – DESCRIÇÃO DO FRAMEWORK DMI 60
4.1.1 – Gerente 61
4.1.2 – Papéis 62
4.1.3 – Manipulação de exceções 63

CAPÍTULO 5 PROPOSTA DE UM MODELO DE AGENTE 65

5.1 – ESTUDO DE CASO 67
5.2 – AGENTES DESCRITOS NA CÉLULA DE PRODUÇÃO 71
5.2.1 - AGENTE ENVIRONMENT (agEnvironment) 71
5.2.2 - AGENTE TRAFFIC LIGHT (agTrafficLight1) 72
5.2.3 - AGENTE FEEDBELT (agFeedBelt) 72
5.2.4 - AGENTE TABLE (agTable) 73
5.2.5 - AGENTE ROBOT (agRobot) 74

5.2.6 - AGENTE PRESS (agPress1 e agPress2) *75*
5.2.7 - AGENTE DEPOSIT BELT (agDepositBelt) *76*
5.3 - IMPLEMENTAÇÃO DAS DMIS NOS AGENTES 77
CAPÍTULO 6 CONCLUSÃO 83
REFERÊNCIAS 85

Capítulo 1

Introdução

Existe uma crescente dependência da sociedade nos sistemas computacionais, evidenciando a necessidade de sistemas cada vez mais confiáveis, uma vez que à medida que mais pessoas são beneficiadas pelas máquinas, mais prejuízos são causados pelos problemas ocorridos nos funcionamentos destas. Dessa forma, torna-se necessário a utilização de mecanismos para lidar com os problemas que potencialmente possam afetar os sistemas. Esses mecanismos são tratados pela área de dependabilidade de sistemas.

A dependabilidade de sistemas é a habilidade de entregar um serviço que pode, justificadamente, ser confiado, evitando defeitos no serviço que são mais freqüentes e mais severos do que se considera aceitável. A dependabilidade consiste em três características principais (Figura 1.1): as ameaças, os atributos e os meios pelos quais ela pode ser atingida [AVI2004].

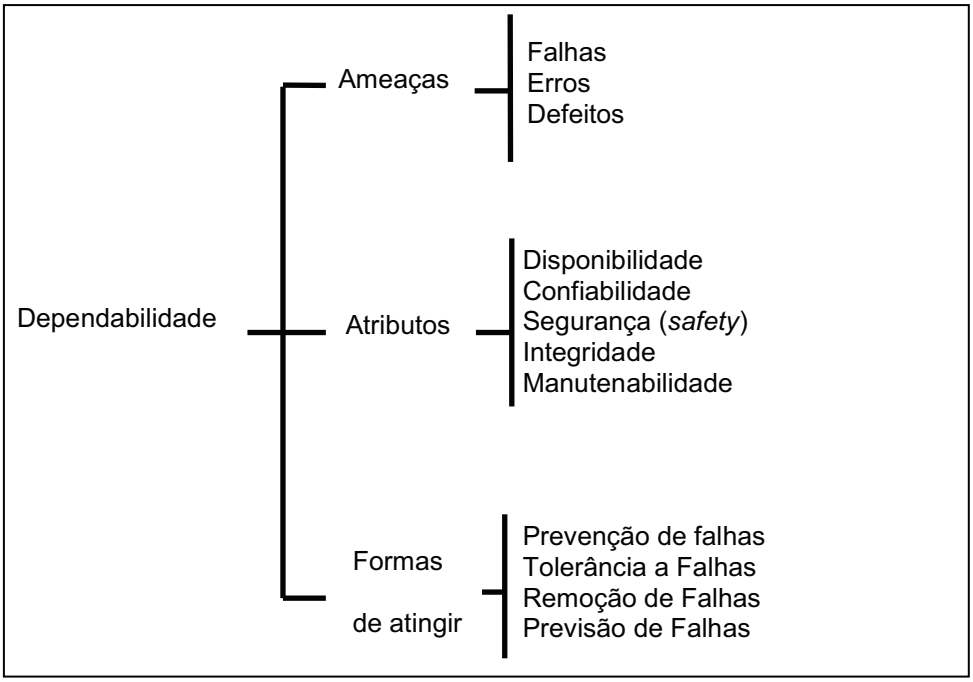

Figura 1.1: Características da dependabilidade

Conforme apresentado na Figura 1.1, as quatro formas de se atingir confiabilidade de um sistema são:

- ✓ **Prevenção de falhas:** prevenir a ocorrência ou introdução de falhas.
- ✓ **Tolerância a falhas:** evitar o defeito do serviço na presença de falhas
- ✓ **Remoção de falhas:** eliminar o número e gravidade dos efeitos das falhas.
- ✓ **Previsão de falhas:** estimar o número atual e futuro de falhas, além de estimar as prováveis conseqüências das falhas.

Apesar da prevenção, remoção e previsão de falhas auxiliarem, e muito, no aumento da confiança em um sistema funcionar conforme especificado, elas não garantem que um sistema esteja livre de falhas residuais. A Tolerância a Falhas é a forma de se atingir dependabilidade mesmo na presença de falhas, permitindo que todos os serviços continuem a operar, com o desempenho correto, de acordo com as suas especificações funcionais mesmo na presença de falhas.

Um aspecto fundamental para obter tolerância a falhas é a detecção eficiente de erros. Para se conseguir detectar erros em um sistema, é necessário conhecer o estado atual e os estados desejados do sistema. O conceito de Tolerância a Falhas foi apresentado originalmente por Avizienis em 1967, entretanto estratégias para se construir sistemas mais confiáveis eram usadas desde os primeiros computadores. Apesar de envolver técnicas e estratégias tão antigas, a tolerância a falhas ainda não é uma preocupação rotineira de projetistas e usuários, ficando sua aplicação quase sempre restrita a sistemas críticos [WEB03]. Falhas são inevitáveis, mas as conseqüências das falhas (interrupção no fornecimento de um serviço ou até perda de dados) podem ser evitadas através do uso de técnicas adequadas e viáveis de tolerância a falhas. Entretanto, as técnicas que toleram falhas tem custo associado, avaliar relação do custo e benefício para cada projeto específico, determinando a melhor técnica que se encaixa no orçamento da empresa.

Para se atingir tolerância a falhas, é necessário o uso de redundância. Todas as técnicas de tolerância envolvem alguma forma de redundância, seja ela de componentes de software, hardware, informação ou tempo. Todas as formas de redundância têm algum impacto no sistema, seja de custo, desempenho ou consumo. Desta forma, o uso da redundância em qualquer projeto deve ser bem ponderado.

Outro aspecto que deve ser levado em consideração, quando da construção de um sistema confiável, é o tipo de ambiente sobre o qual o mesmo será construído. Sistemas distribuídos ou paralelos possuem diversos aspectos que devem ser levados em consideração, dado que mais de uma linha de execução (processo, *threads*, ...), acontece ao mesmo tempo. Neste sentido a complexidade do tratamento de falhas existentes em linhas de execução diferentes aumenta.

Um mecanismo proposto para coordenar interações simultâneas são as Ações Atômicas Coordenadas (*Coordinated Atomic actions - CA actions*) [ROM97]. As *CA actions* asseguram o acesso consistente a objetos na presença de falhas simultâneas, permitindo a recuperação de erro entre os diversos participantes da ação. *CA actions* fornecem um modelo conceitual para diferentes tipos de concorrência, ou seja, concorrência competitiva e concorrência cooperativa, estendendo dois conceitos complementares: transações [KOR98] e conversações [RAN75]. As conversações são usadas para controlar a concorrência cooperativa e implementar a recuperação de erro coordenado. As transações, por outro lado, são usadas para manter a consistência de recursos compartilhados na presença das falhas e da concorrência competitiva [ROM97].

CA actions têm sido utilizadas para desenvolver diversos estudos de caso [ROM97] [ROM96] [ZOR99b]. A grande maioria destes estudos de casos foram desenvolvidos com a utilização de um *framework* projetado para trabalhar com Interações Multiparticipantes Confiáveis (DMIs – *Dependable Multiparty Interactions*). Uma DMI é uma abstração de controle geral que é usada para uma interação entre diversos participantes e fornece recursos para o tratamento de exceções concorrentes. Tem como função auxiliar na construção de atividades concorrentes complexas e auxiliar na recuperação de erros em sistemas onde existe cooperação entre as atividades concorrentes [ZOR99a].

DMI é um tipo de interação multiparticipante que fornece instrumentos para tratamento de exceções concorrentes: quando uma exceção ocorre em um dos participantes da interação, mas não é tratada por ele, a exceção deve ser propagada para todos os participantes da interação. Uma DMI também assegura a consistência na saída: um participante somente deixa sua interação quando todos tiverem terminado suas funções e os dados que são acessados competitivamente por diversas interações estiverem em um estado consistente.

Além dos dois mecanismos para tolerância a falhas citados anteriormente, diversos outros foram propostos na literatura, por exemplo, programação com N-versões [AVI85] e blocos de recuperação [RAN75].

Programação com N-versões [AVI85] (NVP – *n-version programming*) ou programação multiversão é um método ou processo de engenharia de software onde múltiplos programas, equivalentes funcionalmente, são independentemente gerados pela mesma especificação inicial. O conceito foi introduzido por Avizienis em 1985 com a conjectura que "a independência dos esforços de programação reduz significativamente a probabilidade de existirem falhas idênticas em software em duas ou mais versões do programa" [CHE95].

Blocos de recuperação (*recovery blocks*) [RAN75] é um método de estruturação de programas que tem objetivo de auxiliar na detecção de erro e facilitar a recuperação do mesmo. A idéia central dos blocos de recuperação é estabelecer um ponto de recuperação e então executar uma versão do programa. Caso o resultado seja aceito por um teste de aceitação o programa continua normalmente. Caso contrário uma nova versão do programa é executada. Se o resultado de nenhuma versão passar pelo teste de aceitação, então o estado salvo é recuperado e um sinal é enviado para o usuário do programa.

Várias técnicas têm sido propostas para prover confiabilidade para sistemas distribuídos, que são desenvolvidos utilizando técnicas de programação orientada a objetos. Entretanto, a programação orientada a agentes está sendo proposta como uma nova abordagem para o desenvolvimento de sistemas distribuídos. Seu principal recurso é ser uma poderosa ferramenta que modela comportamento do sistema em termos de interação entre entidades autônomas [ZOR05].

Com o progresso da programação orientada a agentes (*Agent Oriented Programming*) e novas metodologias de programas baseados em agentes criadas, vários autores já manifestaram preocupação sobre utilização de tolerância a falhas em Sistemas Multiagentes (*Multi-Agent System*) [WOO02][WEI99]. Sistemas Multiagentes (SMA) são mais flexíveis que sistemas distribuídos tradicionais devido à sua capacidade de raciocinar sobre situações imprevisíveis e mais confiáveis devido ao fato que os agentes são facilmente replicados [WEI99]. Além disso, é importante para o SMA ter alguns tipos de mecanismos que permitem a um grupo de agentes executar algumas tarefas para tentar deixar o sistema estável no caso de falha, ou caso contrário, notificar o sistema sobre falhas catastróficas.

Uma vantagem do SMA é a robustez e a confiabilidade pois a falha de um ou mais agentes não torna o sistema inteiro inutilizável, porque outros agentes que estejam disponíveis podem assumir as tarefas.

Um dos modelos mais utilizados pelo SMA é o modelo de Crenças, Desejos e Intenções (*BDI - Belief, Desire, Intention*) [BRA87] que estuda comportamento de agentes, transformando o raciocínio humano compreensível para o computador através dos estados mentais, mostrando crenças, desejos e intenções. Para implementar os estados de agentes BDI utilizamos o interpretador Jason que é uma extensão da linguagem AgentSpeak, permitindo descrever as crenças, desejos e intenções de um agente BDI.

O interpretador Jason permite modelar agentes BDI, mas precisa de um mecanismo para manter o sistema estável mesmo na presença de falhas, principalmente por estar trabalhando distribuído em vários ambientes. Para isso integramos os conceitos de Interação Multiparticipante Confiável (DMI – Dependable Multiparty Interaction) e agentes BDI, desenvolvendo um modelo de agente BDI tolerante a falhas.

A dissertação está organizada da seguinte forma:

O Capítulo 2 apresenta conceitos, características e principais arquiteturas de agentes. Apresentamos também neste capítulo conceitos dos estados mentais, utilizando o modelo BDI.

O Capítulo 3 aborda a Linguagem *AgentSpeak,* utilizando o interpretador *Jason* para criar um agente BDI, explicando o funcionamento do interpretador através da criação de crenças, objetivos, planos e ambiente.

O Capítulo 4 descreve o mecanismo de Tolerância a Falhas DMI, mostrando os principais componentes e o funcionamento do mecanismo DMI.

O Capítulo 5 apresenta o modelo de um agente BDI Tolerante a Falhas proposto, onde descrevemos o agente, bem como a estrutura dos estados mentais. Aplicamos esse modelo proposto no estudo de caso da Célula de Produção, desenvolvendo agentes com as características de um agente tolerante a falhas.

Capítulo 2

Modelo BDI

Este capítulo apresenta os principais aspectos relacionados com o modelo BDI, ou seja, definição de agentes inteligentes (Seção 2.1), estados mentais (Seção 2.2), arquitetura genérica para o modelo BDI (Seção 2.3), algumas arquiteturas BDI são descritas nas Seções 2.3 e 2.4 e as ferramentas utilizadas para desenvolvimento de agentes que utilizam o modelo BDI (Seção 2.5).

Entre as diversas áreas estudadas pela Inteligência Artificial (IA), algumas são dedicadas ao estudo de estados mentais (crenças, desejos e intenções), (BDI - *Belief, Desire and Intention*)[BRA87]. Um modelo baseado nesses estados mentais representa os processos internos através de um mecanismo de controle que seleciona de forma racional o curso das ações a serem executadas.

O modelo BDI sobre ação racional humana foi originalmente desenvolvida pelo filósofo Michael Bratman [BRA87]. Ele é um modelo sobre raciocínio prático, o qual consiste em ponderar considerações conflitantes a favor e contra alternativas competitivas, onde as considerações relevantes são determinadas pelos desejos e crenças do agente. Por exemplo, processo de decisão de uma pessoa viajar ou estudar, são ações que podem ser tomadas, escolhe-se uma alternativa e traça-se planos para conseguir realizar essa alternativa.

2.1 – Agentes inteligentes

Apesar de ser um termo atualmente muito utilizado, ainda não existe uma definição universal para o termo agente, cada grupo de pesquisa segue uma determinada linha, conforme seus próprios objetivos, apresentando sua definição personalizada do termo agente. Por exemplo, um agente pode ser definido como um sistema de computador que está situado em algum ambiente, capaz de ações autônomas nesse ambiente a fim de satisfazer seus objetivos [WEI99]; ou como um sistema capaz de perceber através de sensores e agir em um dado ambiente através de atuadores [RUS96].

As definições acima referem ao termo “agente” e não a agentes inteligentes, a diferença entre eles é que os agentes inteligentes podem deliberar, ou seja, escolher o curso de suas ações. Agentes inteligentes são capazes de agir sem a intervenção de pessoas ou outros sistemas, tendo o controle de seus próprios estado interno, e também do seu comportamento.

Na Figura 2.1 apresentamos uma visão geral de um agente. Neste esquema podemos ver que o agente tem entradas sensoriais provenientes do ambiente, produzindo saídas geradas pela ação do agente, que afetam o seu ambiente. Normalmente um agente terá diversas ações disponíveis, este conjunto de ações possíveis representa a capacidade de modificar seus ambientes. Algumas ações podem não atender todas as situações, podendo existir ações que possam falhar por não ter uma ação aplicável aquela situação. Ações possuem pré-condições associadas a elas, que define as possíveis situações as quais elas podem ser aplicadas.

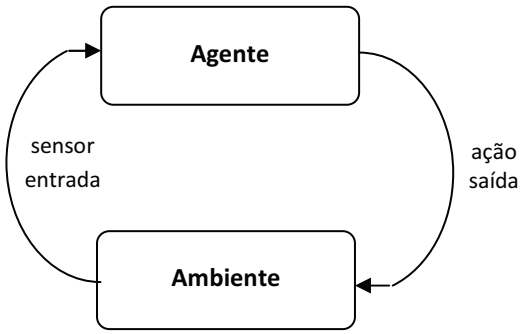

Figura 2.1: Agente em seu ambiente [WEI99]

Russel e Norvig [RUS96] propuseram uma definição de um agente inteligente ideal: “para cada possível seqüência de percepções, um agente inteligente ideal deve agir de modo a obter maior sucesso, com base na evidência fornecida pela seqüência de percepções e qualquer conhecimento embutido que o mesmo possua.”

Segundo Bordini [BOR07], agentes são sistemas que estão situados em um ambiente, sendo capazes de detectar seu ambiente (através de sensores) e utilizando um conjunto de possíveis ações que eles podem executar (através de atuadores) para modificar seu ambiente como podemos visualizar na Figura 2.2.

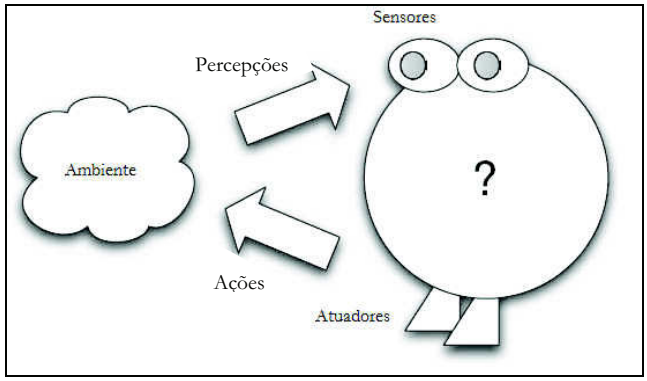

Figura 2.2: Agente e ambiente [BOR07]

Em Wooldridge e Jennings [WOO95] estão descritos um conjunto de propriedades usualmente aceitas e que os agentes devem exibir. Algumas propriedades consideradas para construção de agentes são:

- **Autonomia:** um agente deve ser capaz de operar sem intervenção direta de um usuário, e ter algum tipo de controle sobre seu comportamento e seu estado interno;
- **Habilidade Social:** um agente deve ser capaz de interagir com outros agentes, humanos ou não, em prol de atingir seus objetivos;
- **Pró-atividade:** um agente inteligente não deve simplesmente responder aos estímulos vindos de outros agentes e do seu ambiente, deve também tomar iniciativa para que seus objetivos sejam atingidos;
- **Reatividade:** um agente deve ser capaz de perceber o seu ambiente, e responder de uma forma oportuna às mudanças que ocorrem no ambiente, a fim de atender os seus objetivos;

Além das propriedades citadas acima também existem outras características que são importantes para o conceito de agente na concepção de Sistemas Multiagentes (SMA), apresentadas a seguir [HÜB04]:

- **Percepção:** o agente é capaz de perceber alterações no ambiente;
- **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;

- **Comunicação:** umas das ações possíveis de um agente é comunicar-se com outros agentes da sociedade (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 (por exemplo, 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; 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 de pesquisa que ainda requer bastante investigação.

2.1.1 – Sistemas Multiagentes

Segundo Hübner, Bordini e Vieira [HÜB04], Sistema Multiagentes (SMA) abordam o comportamento de um grupo organizado de agentes autônomos, que cooperam na resolução de problemas, sendo que, se os agentes não estivessem organizados em grupos não poderiam solucionar. Um Sistema Multiagentes possui características e vantagens que determinam sua viabilidade e funcionalidade como ambiente virtual de uma coletividade baseada em recursos computacionais.

Uma dessas características do SMA é possuir capacidade de adaptar-se a novas situações, mudando sua organização e/ou determinando a formação da coletividade de agentes, permitindo um alto nível de abstração por apresentar-se de uma forma natural para a modelagem de sistemas complexos de forma distribuída, onde o conhecimento, o controle e os recursos podem estar distribuídos entre várias e diferentes plataformas, arquiteturas e sistemas.

Um agente autônomo pode ocupar um ambiente isoladamente ou um ambiente que contêm outros agentes independentes, formando nesse último, um sistema multiagente. A Figura 2.3 apresenta uma visão de um sistema multiagente; na parte inferior da figura podemos ver o ambiente compartilhado que o agente ocupa, cada agente tem uma "esfera da influência" (porção do ambiente que eles são capazes de controlar totalmente ou parcialmente) neste ambiente. Mas existe a possibilidade de sobrepor essa "esfera da influência", ou seja, o ambiente pode ser controlado em conjunto com outros agentes, agrupados através dos seus conhecimentos em comum, denominado de relacionamento organizacional, visualizados na parte superior da Figura 2.3 [BOR07].

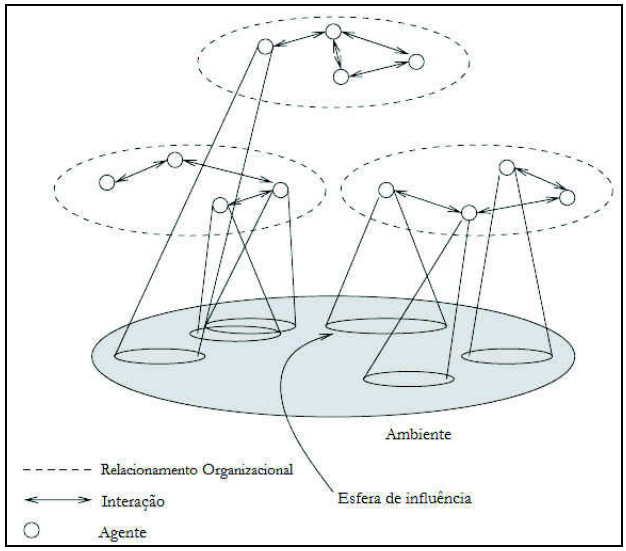

Figura 2.3: Estrutura de um Sistema Multiagentes [BOR07]

2.1.2 – Ambientes

Os ambientes provêm informações que são captadas pelas percepções do agente. Segundo Russel e Norvig [RUS96], os ambientes podem ser classificados quanto a suas propriedades:

• **Acessível ou inacessível:** Um ambiente acessível é aquele no qual o agente pode obter informações completas, precisas e atualizadas sobre o seu estado. Caso contrário, é dito inacessível;

• **Determinístico ou não-determinístico**: Um ambiente determinístico é aquele no qual uma ação tem um único efeito possível. Quando não é possível saber o estado que o ambiente assumirá após a execução de uma ação, dizemos que o ambiente é não-determinístico;

• **Estático ou dinâmico**: Um ambiente é estático para um agente quando permanece inalterado até o momento em que ele executa alguma ação. Em um ambiente dinâmico, além das ações desempenhadas pelos agentes, existem processos que operam sobre o ambiente, alterando o estado do mesmo;

• **Discreto ou contínuo:** Um ambiente discreto é aquele que possui um número finito de estados. Um exemplo de ambiente discreto é o jogo de damas, o qual possui um número limitado de movimentos a cada jogada. Um ambiente contínuo é aquele no qual é possível assumir incontáveis estados. É o caso de um veículo em movimento, o qual possui a sua localização representada dentro de um intervalo de valores contínuos.

2.2 – Estados Mentais

Alguns agentes, no modelo BDI, possuem estados internos que se relacionam com o estado do ambiente com o qual eles interagem. Estes estados correspondem aos estados mentais humanos, apresentam um vínculo com o mundo através do qual estabelecem sua existência e significância. A característica pela qual os estados mentais humanos referem-se a objetos ou situações do mundo é chamada de intencionalidade. Crença, desejo, expectativa, capacidade e intenção são exemplos de estados intencionais [ZAM01].

A intencionalidade, encontrada nos estados mentais, indica certa direcionalidade, ou seja, uma propriedade de direcionamento do mundo para o agente e vice-versa. Como exemplo, a afirmação "a porta está fechada" é uma crença sobre a porta, do mundo para o agente; e a afirmação "entrar na sala" é um desejo, do agente para o mundo. Logo, intencionalidade não está ligada ao estado mental "intenção", mas sim a todos os estados que possuem a propriedade de direcionalidade [ZAM01].

2.2.1 – Crenças

Uma crença é um estado mental intencional fundamental para as interações dos agentes com noção idêntica a de conhecimento [ZAM01]. A crença contém a visão fundamental do agente com relação ao seu ambiente, ele as usa para expressar suas expectativas sobre possíveis estados futuros. As crenças de um agente podem ser vistas como o provável estado do ambiente, isto é, como um componente informativo do estado do sistema.

2.2.2 – Desejos

É um dos estados mentais humanos que não tem uma definição precisa, sendo, portanto, usado e explicado de diversas maneiras [ZAM01]. Diferentes dos estados prováveis representados pelas crenças, os desejos representam estados desejáveis que o sistema poderia apresentar. Para um desejo deixar de ser "desejável" e tornar-se algo mais real, falta o conhecimento inerente a ele e o contexto favorável à sua realização. Os desejos motivam o agente a agir de forma a realizar metas, tais ações são realizadas através das intenções causadas pelos desejos.

Em alguns trabalhos de arquitetura BDI, os desejos são tratados no sentido de objetivos a serem alcançados por um agente.

2.2.3 – Intenções

As intenções dos agentes são um subconjunto dos desejos. Se um agente decide seguir uma meta específica (desejo), então esta meta (desejo) torna-se uma intenção. As intenções determinam o processo de raciocínio prático, pois a propriedade mais óbvia das intenções é que elas determinam as ações a serem realizadas, sendo o resultado da escolha com comprometimento, a qual leva o agente à ação.

Bratman [BRA87] distingue o conceito de fazer algo intencionalmente (ação) e possuir intenção de fazê-la (estado mental). A palavra intenção tem no uso cotidiano maneiras distintas: ações (físicas, verbais etc) e ações mentais (estados da mente).

2.3 – Arquitetura BDI

Entre as arquiteturas de estados mentais está a arquitetura BDI. Segundo Giraffa [ZAM01], as idéias básicas do modelo BDI são: descrever o processo interno de um agente utilizando o conjunto de estados mentais (crença, desejo e intenções); e definir uma

arquitetura de controle através da qual o agente possa selecionar o curso de suas ações. Além destes componentes, algumas arquiteturas BDI usam o conceito de planos. Planos seriam o conjunto de sub-tarefas que deve ser seguido, quando gerada uma intenção, para a realização de uma tarefa sobre o ambiente [WEI99][MEN05].

Na Arquitetura BDI, Figura 2.4, um agente BDI executa as seguintes funções: gerar uma lista de opções (baseadas nos desejos e pelo refinamento do agente), selecionar nesta lista de opções disponíveis (baseado nos desejos, crenças e pelo refinamento do agente), gerando as novas intenções (adicionadas na estrutura de intenção), que são as opções que o agente decidiu adotar, gerando uma ação interna ou externa (enviada ao ambiente).

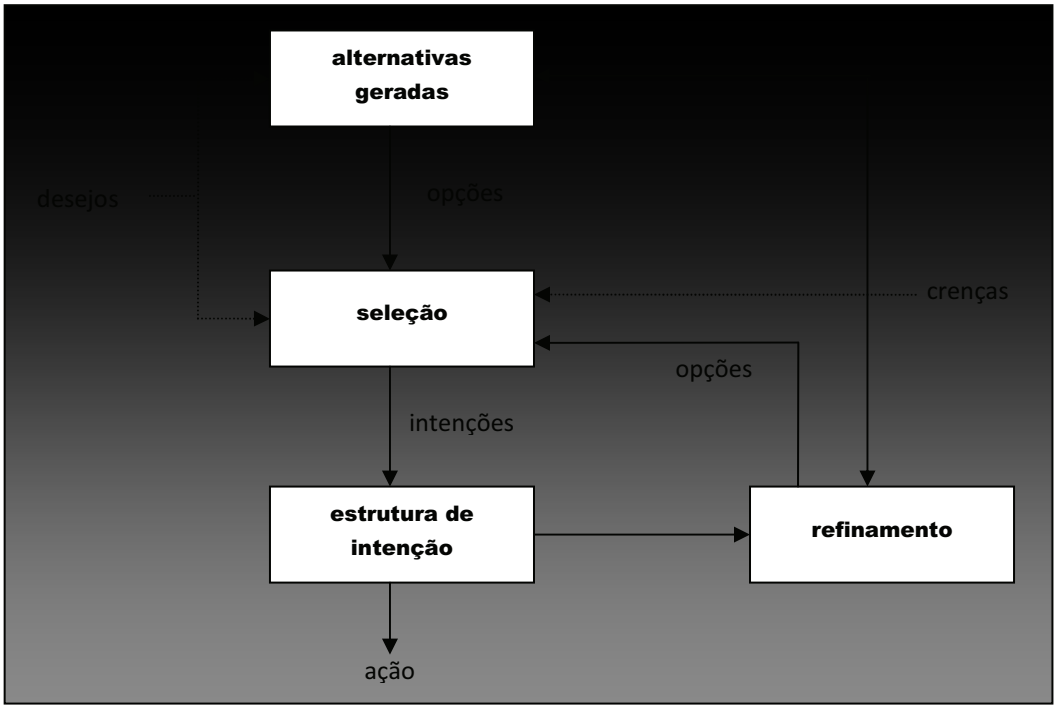

Figura 2.4: Arquitetura BDI [ZAM01]

Ao projetar um agente baseando-se no modelo BDI, são especificados suas crenças e seus desejos, mas a escolha das intenções fica sob a responsabilidade do próprio agente, isto é, de uma auto-análise desses estados inicialmente disponíveis.

O problema no desenvolvimento do raciocínio prático do agente é como alcançar um balanceamento entre estes diferentes conceitos, principalmente no que se refere a abandonar intenções.

Em Weiss [WEI99] são definidos sete componentes importantes de uma arquitetura BDI (Figura 2.5):

- Um conjunto de crenças atuais, que representam as informações que o agente tem sobre seu ambiente atual;
- Uma função de revisão de crenças, que a partir da entrada percebida e com as crenças atuais do agente, determina um novo conjunto de crenças;
- Uma função geradora de opções, que determina as opções disponíveis para o agente, ou seja, seus desejos, tendo como base suas crenças atuais sobre seu ambiente e suas intenções atuais;
- Um conjunto de desejos atuais, representando possíveis cursos de ações disponíveis para o agente;
- Uma função filtro, que representa o processo de deliberação do agente, e que determina as intenções dos agentes, tendo como base suas crenças, desejos e intenções atuais;
- Um conjunto de intenções atuais, representando o foco atual do agente;
- Uma função de seleção de ação, que determina uma ação para executar, tendo como base as intenções atuais.

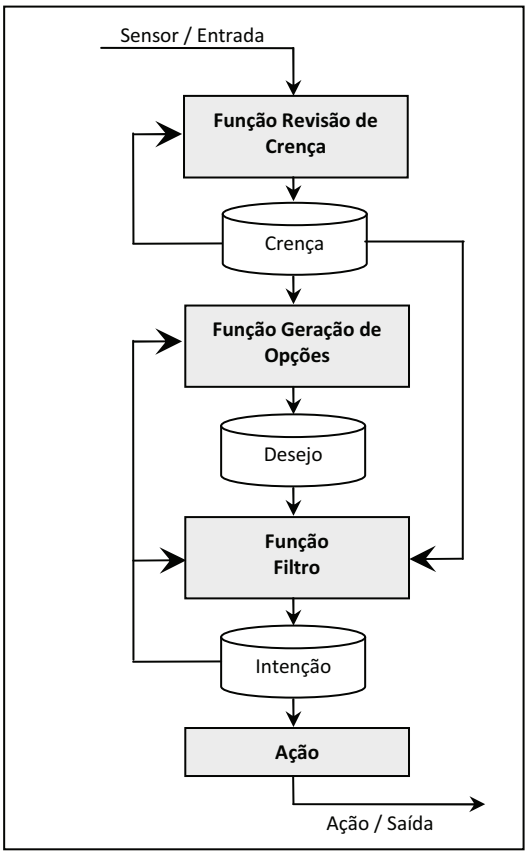

Figura 2.5: Componentes de uma arquitetura BDI [WEI99]

2.4 – Implementações da Arquitetura BDI

A teoria de raciocínio prático de *Bratman* [BRA87] levou ao desenvolvimento do modelo BDI de agentes, permitindo a criação de vários trabalhos visando a sua implementação em sistemas computacionais reais, podemos destacar as seguintes arquiteturas:

- **IRMA** *(Intelligent Resource-bounded Machine Architecture)* foi definida com o objetivo de demonstrar a viabilidade do modelo de raciocínio prático de Bratman. O objetivo da IRMA é prover um mecanismo de raciocínio para um agente que leva em consideração seus recursos limitados. Outro aspecto desta arquitetura é ser uma das primeiras a incorporar intenções como estado mental fundamental no processo de raciocínio prático. A organização interna da IRMA contém dois tipos básicos de entidades: processos e entidades de armazenamento.
- **PRS** (*Procedural Reasoning System*) [GEO87] foi criado visando uma arquitetura BDI que pudesse ser utilizada em aplicações do mundo real. Também tinha como característica suportar raciocínio tanto orientado a objetivos quanto reativo. O sistema foi utilizado inicialmente na implementação de um sistema de controle de tarefas em um simulador de espaçonaves da NASA. O PRS é organizado em uma série de componentes arquiteturais que são controlados pelo interpretador.
- **AgentSpeak(L)** [RAO96] foi criada a fim de diminuir a distância entre a teoria e a prática de agentes BDI. Esta distância é resultante de diversos fatores [RAO96][D'IN98], como por exemplo, a tênue relação entre as lógicas utilizadas nas teorias com problemas reais. É uma linguagem baseada em regras que expressam o conhecimento do agente e estas, são usadas em planos. A leitura informal dessas regras é: "se o agente quer realizar o objetivo "O" e acredita que a situação "S" é o caso, então o plano "P" pode ser a coisa correta a fazer". Um agente modelado na linguagem AgentSpeak(L) possui um conjunto de crenças, desejos, regras de planos, intenções e eventos que constituem seu estado mental.

A existência de vários modelos de arquiteturas formais de agentes BDI permitiu o entendimento conceitual do funcionamento dessa arquitetura, a implementação e construção de sistemas multiagentes. Podemos encontrar algumas ferramentas baseadas em alguns desses modelos citados acima, na próxima seção descreveremos duas dessas ferramentas: X-BDI e Jason.

2.5 – Ferramentas BDI

As ferramentas apresentadas nessa seção foram desenvolvidas com a intenção comum de diminuir a distância existente entre as teorias formais para especificação de agentes e aplicações, tais como arquiteturas abstratas para construção de agentes de software e linguagem orientada a agente.

Nesta seção apresentaremos as seguintes ferramentas: X-BDI [MÓR99] e Jason [BOR03] [HÜB04], dando ênfase conceitual e prática nessa última.

2.5.1 – X-BDI

O modelo X-BDI de agentes foi proposto com o objetivo de ser simultaneamente uma ferramenta de especificação formal e um *framework* executável de agentes [MÓR99].

O X-BDI é um ambiente que permite a descrição formal de agentes cognitivos baseados no modelo BDI, sendo ao mesmo tempo, uma linguagem para implementação desses agentes. Segundo [MÓR99], sua proposta disponibiliza um sistema formal cuja linguagem é adequada para a representação dos estados mentais de agentes. Esse sistema fornece suporte para diversos tipos de raciocínios, tratados de forma computacional, além de fornecer as ferramentas necessárias para se modelar os estados mentais da tríade BDI.

O modelo possui duas características: serve como ambiente de especificação de agentes, em que é possível definir formalmente um agente através da descrição dos seus estados mentais (crenças, desejos e intenções) e de suas respectivas propriedades. Este ambiente pode, também, fazer a verificação destas propriedades; serve como um ambiente de implementação de agentes. O X-BDI é um ambiente onde se formaliza e executa o agente, verificando se o comportamento desejado ocorreu ou não. O X-BDI possui um ambiente de implementação com alto nível de abstração (os estados mentais), o que reduz a complexidade no desenvolvimento de sistemas baseados em agentes. Inicialmente, o agente

possui uma base de crenças, um conjunto de desejos potencialmente factíveis e que podem ser dinamicamente atualizados ao longo das interações.

Apesar de também ser uma ferramenta de desenvolvimento de agentes, o X-BDI é antes de tudo, um modelo formal de agentes. O X-BDI, como modelo formal de agentes, reduz a distância entre especificação e implementação de agentes. Preserva-se tanto as principais características dos modelos formais (habilidade para definir e verificar formalmente agentes), como fornece mecanismo para agentes raciocinarem [MÓR99]. Em particular, o que se deseja é possuir uma ferramenta que permita modelar agentes que possuam comportamento autônomo e flexível. Esse agente, para ser considerado inteligente deve possuir: autonomia (por exemplo, um robô decide por si que ordens priorizar, que ações tomar para satisfazer estas ordens), habilidade social (por exemplo, um robô recebe ordens dos professores e funcionários, comunica-se com os visitantes para conduzi-los ao seu destino), reatividade (por exemplo, um robô locomove-se sem esbarrar em obstáculos, imediatamente recarrega a bateria ao detectá-la com pouca carga), pró-atividade (por exemplo, um robô é capaz de construir planos para chegar nos gabinetes desejados, prioriza o atendimento dos visitantes para que estes não esperem ou caminhem em demasia) e adaptabilidade (por exemplo, um robô re-planeja quando constata que o elevador está com problemas, interrompe a entrega do envelope quando percebe que tem pouca carga na bateria e retoma a entrega depois de fazer a recarga).

A principal contribuição do X-BDI é a especificação de um modelo executável de agente. Entretanto, O X-BDI não foi implementado com preocupações de desempenho e sua utilização em situações com tempo restrito é limitada.

Em [MEN05] foi desenvolvida a versão X2-BDI, que é uma extensão do modelo e interpretador de agentes X-BDI. X2-BDI é composto essencialmente de três partes, o kernel do X-BDI, implementado em Prolog, uma biblioteca de planejamento contendo uma implementação em C++ do Graphplan [BLU97] e uma interface gráfica (AgentViewer) em Java utilizada para facilitar a operação do X-BDI e a visualização da interação do mesmo com o ambiente. A modificação fez uma inclusão dos mecanismos de mapeamento entre os estados mentais do agente para problemas de planejamento proposicional e de planos proposicionais de volta para os estados mentais do agente.

2.5.2 – JASON

Jason (*A Java-based AgentSpeak Interpreter Used with Saci For Multi-Agent Distribution Over the Net*) é um interpretador da linguagem *AgentSpeak* [BOR03] [HÜB04]. Inclui o

tratamento de mecanismos de comunicação, de funções de confiança além da arquitetura baseada no modelo BDI (crenças, desejos e intenções). Uma aplicação sob o Jason pode ser distribuída para rodar em múltiplas máquinas. Como característica fundamental, Jason é portável, ou seja, multi-plataforma, pois roda em Java e pode ser executado em qualquer sistema e máquina que possui a JVM instalada [APP05]; possui biblioteca de ações internas que estão definidas na base BDI e podem ser referidas nos planos dos agentes; além de possibilitar a extensão dessas ações através de programação pelo usuário-programador, originando o que se denomina ações externas, especificadas no ambiente.

Ações internas são ações definidas pelo usuário que executam internamente (no agente). Essas ações são chamadas de 'ação interna' a fim de distingui-las das ações que estão no corpo de um plano e denotam ações que o agente deseja executar para alterar o ambiente (chamadas 'ações básicas' e que representam a atuação do agente) [BOR07].

Além de ser um interpretador da linguagem *AgentSpeak(L),* o Jason apresenta outros recursos interessantes, podendo ser citado [HÜB04]:

- **Negação forte:** determinar o que o agente acredita ser verdadeiro, acredita ser falso, ou desconhece.
- **Tratamento de falhas de planos:** como os sistemas multiagentes atuam em ambientes dinâmicos e imprevisíveis, os planos usados para atingir objetivos do agente freqüentemente falham. Por isto, é fundamental que as linguagens de agentes forneçam mecanismos para programar o comportamento do agente quando planos falham.
- **Comunicação baseada em atos de fala:** todo trabalho na área de comunicação de agentes é baseada na teoria dos atos de fala. Em sistemas multiagentes, interação entre os agentes é fundamental, essa característica auxilia na implementação deste aspecto.
- **Anotações:** tanto crenças quanto planos podem ter anotações que são usadas, por exemplo, para registrar qual a fonte de informação que gerou uma crença, ou informações de meta-nível que podem ser utilizadas na escolha de planos a serem executados.

Algumas das características da plataforma Jason, além do interpretador para a variante de *AgentSpeak* mencionada acima, são [HÜB04]:

Distribuição: a plataforma permite que se rode agentes em máquinas diferentes de forma bem fácil, com o uso da infra-estrutura SACI (http://www.lti.pcs.usp.br/saci).

Além disto, os usuários podem customizar a infra-estrutura para distribuição a ser utilizada pela plataforma.

Ambientes: existe suporte para a criação, em Java, do ambiente a ser compartilhados pelos agentes.

Customização: os usuário podem redefinir aspectos do agente (funções de seleção utilizadas pelo interpretador, revisão de crenças, relações sociais com outros agentes, etc.) e da arquitetura geral (forma de percepção, de ação, de troca de mensagens, etc.).

Extensibilidade: novas ações internas a serem usadas no código *AgentSpeak* podem ser programadas em Java, o que também facilita a integração de código legado a ser utilizado de forma bem elegante no raciocínio prático do agente (programado em *AgentSpeak*).

IDE: a IDE é um plug-in para o jEdit (http://www.jedit.org/). Um dos aspectos interessantes da IDE é o "inspetor de mente" que facilita a depuração de agentes.

2.6 – Considerações Finais

Este Capítulo apresentou os principais aspectos relacionados com o modelo de agentes BDI. Conforme pode ser verificado até o momento, existem poucos trabalhos que consideram técnicas de tolerância a falhas, por exemplo, Blocos de Recuperação [RAN75], Programação com N-versões [AVI85] ou Interações Multiparticipantes Confiáveis [ZOR99].

Um dos objetivos desta dissertação, conforme mencionado na Introdução, é integrar algum mecanismo de Tolerância a Falhas a uma ferramenta para descrição de agentes. Assim, o Capítulo 3 descreverá em detalhes a linguagem AgentSpeak (aplicada no interpretador Jason) e o Capítulo 4 descreverá o mecanismo de Tolerância a Falhas de Interações Multiparticipantes Confiáveis (DMI – Dependable Multiparty Interactions).

Capítulo 3

Linguagem AgentSpeak(L)

A linguagem *AgentSpeak(L),* primeiramente apresentada em [RA096], é uma extensão natural e elegante de programação em lógica para arquitetura de agentes BDI, representando um modelo abstrato para programação de agentes e tem sido a abordagem predominante na implementação de agentes inteligentes ou "racionais" [WOO00].

Foi projetada para programação de agentes BDI na forma de sistemas de planejamento reativos (*reactive planning systems)*, que são sistemas em permanente execução, reagindo a eventos que acontecem no ambiente [BOR03].

Um agente *AgentSpeak* corresponde à especificação de um conjunto de crenças, planos, eventos ativadores e um conjunto de ações básicas que o agente executa no ambiente. As informações sobre os desejos (estados futuros a serem atingidos), além das alternativas disponíveis ao agente para ativar as intenções (atingir seus objetivos), estão implicitamente representados nos planos [HÜB04]. Com isso podemos afirmar que os planos são referências às ações básicas de agentes no ambiente, ou seja, um plano determina uma forma de atingir um determinado objetivo.

A Figura 3.1 apresenta um exemplo clássico, descrito em [BOR03], de planos *AgentSpeak*(L), o objetivo do plano é a reserva de ingressos (assentos) para o concerto 'A' no local 'V' através de contato telefônico, o qual é executado pelo agente assumindo que esta é a ação básica que o agente é capaz de executar no ambiente.

Para esse fim são apresentados dois planos, no primeiro é apresentada a opção do concerto do artista 'A' no local 'V', esse corresponde a adição de uma crença '+concerto(A,V)', obtida pela percepção do ambiente. Caso o agente escolha por essa opção ("gosta(A)"), o objetivo do agente será reservar ingressos para esse concerto ("!reserva_tickets (A,V)").

O segundo plano mostra que ao adotar o objetivo de reservar ingressos (reserva_tickets(A,V)), verifica se a linha telefônica está ocupada (~ocupado(fone)), se não

estiver então o agente pode executar o plano que consiste em executar a ação básica de fazer contato telefônico com o local do concerto 'V' (liga(V)), seguindo de um determinado protocolo de reserva de ingressos (indicado por '...'), e que é finalizado com a execução de um subplano (escolhe_assento(A,V)) para realizar a escolha de assentos em eventos desse tipo naquele local [BOR03].

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

+! reserva_tickets(A,V) : ~ocupado(fone)
     ← liga(V);
        ...;
        !escolhe_assento(A,V).
```

Figura 3.1: Exemplo de plano em AgentSpeak

3.1 – Interpretador Jason

O interpretador Jason permite desenvolver e por em funcionamento agentes cognitivos baseados no modelo BDI programados em *AgentSpeak(L)*, incluindo comunicação entre agentes baseados na teoria de atos de fala [HÜB04]. O interpretador é um ambiente para modelagem e execução de agentes AgentSpeak, sendo possível descrever planos, crenças e o ambiente em que os agentes estão inseridos.

De acordo com [HÜB04], como a teoria BDI é baseada na literatura filosófica do raciocínio prático [BRA87], a comunicação através do ambiente em um sistema multiagentes é tipicamente baseado na teoria dos atos da fala.

Segundo [BOR07], a arquitetura do agente é a estrutura do programa dentro da qual um programa agente é executado, um exemplo de arquitetura do agente é o PRS (ver Seção 2.4), onde os planos são programas que não residem nessa arquitetura. O programa direciona o comportamento do agente, as ações e eventos que o agente tem, é determinado pela sua arquitetura, sem ter que o programador se preocupar, como por exemplo, em perceber o ambiente e atualizar a base de crenças conforme foi percebido; o interpretador já faz isso constantemente (sem estar explicitamente programado).

A interpretação da linguagem no programa agente efetivamente determina o ciclo de raciocínio do agente. Um agente está constantemente percebendo o ambiente, raciocinando sobre como agir para alcançar seus objetivos e agindo para mudar o ambiente. O raciocínio prático é parte do comportamento cíclico do agente, em AgentSpeak, é feito

de acordo com os planos que o agente tem na biblioteca de planos. Inicialmente, esta biblioteca é formada por planos que o programador escreve em um programa AgentSpeak.

Agentes BDI práticos são sistemas de planejamento reativo, que estão em permanente execução, reagindo em forma de eventos. Através da execução de planos que o sistema de planejamento reativo reage a cada evento; os planos são curso de ações que os agentes se comprometem a executar para tratar os eventos. As ações mudam o ambiente do agente, de forma que podemos esperar que os objetivos dos agentes sejam alcançados.

Um SMA desenvolvido na ferramenta Jason possui um ambiente onde os agentes estão situados e um conjunto de instâncias de agentes *AgentSpeak(L)*. O ambiente dos agentes deve ser desenvolvido na linguagem Java.

Descreveremos todos os componentes da linguagem que podem ser utilizados pelo programador, e que podem ser separados em três principais categorias: crenças, objetivos e planos.

3.1.1 – Crenças

Para representar as crenças do agente utilizamos a base de crenças, que é formada por uma coleção de literais. A informação é representada de forma simbólica por predicados tais como: alta(maria). Que expressam uma propriedade particular (alta) de um objeto ou indíviduo (maria). Para representar certo relacionamento entre dois ou mais objetos, podemos usar predicado como: gosta(maria,chocolate), que descreve que Maria gosta de chocolate. Um literal é um predicado ou negação.

Uma crença é um predicado de primeira ordem na notação de lógica usual (ou fatos, no sentido de lógica de programação) e literais de crenças são átomos de crenças ou suas negações que formarão a base de crenças do agente.

A classificação dos vários tipos de termos disponíveis no interpretador Jason pode ser vista na Figura 3.2.

O estado atual do agente é representado pelas crenças de outros agentes, percepções do ambientes e as crenças sobre si mesmo naquele momento. Os estados que o agente pretende alcançar baseados em seus estímulos internos ou externos (ambientes ou outros agentes) podem ser vistos como desejos que ele almeja atingir, para isso utilizam as intenções que são as ações que o agente se compromete a executar para realizar um desejo.

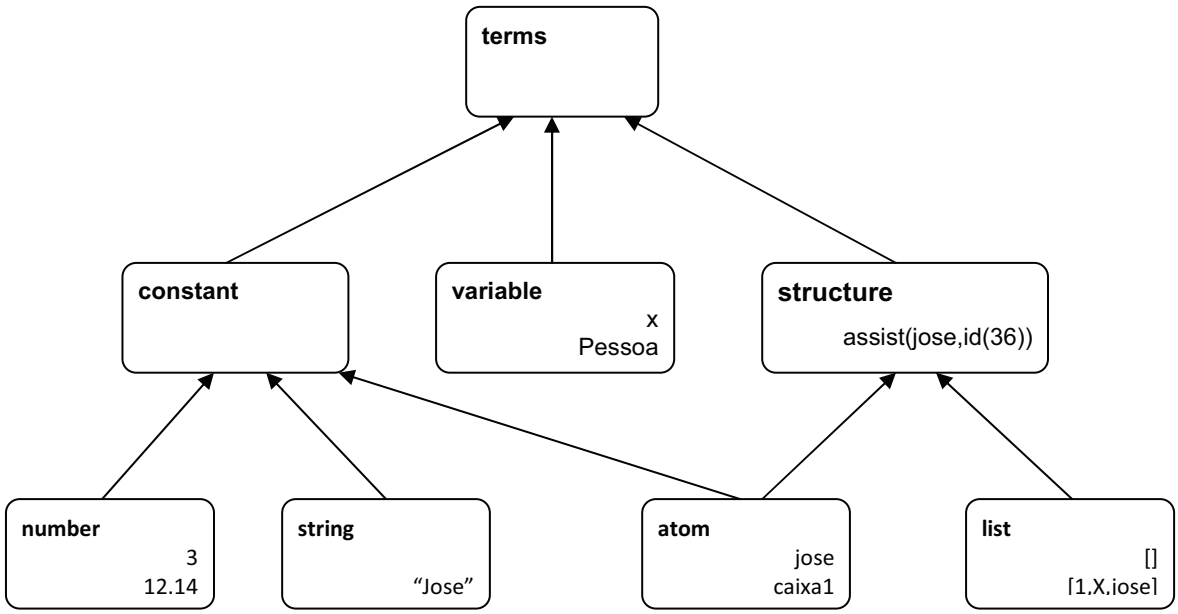

Figura 3.2: Tipos de termos AgentSpeak em Jason [BOR07]

Podemos representar crenças utilizando anotações, que são situações que podemos esperar, tem a vantagem de tornar o código mais elegante, permitindo melhor visualização pelo programador, facilitando o gerenciamento da base de crenças. Podem existir três diferentes tipos de origem de informação, descritas entre colchetes:

- **Informação de percepção**: representação simbólica de uma propriedade do ambiente, adquirindo certas crenças através da percepção do ambiente.
  - *Exemplo*: porta(aberta) [source(percept)]
- **Comunicação**: representar a informação adquirida de outros agentes, conhecendo qual agente passou a informação.
  - *Exemplo*: porta(aberta) [source(maria)]
- **Notas mentais**: informações que o agente precisa lembrar em circunstâncias futuras.
  - *Exemplo*: porta(aberta) [source(self)]

Para representar a negação de uma crença existem dois tipos, o tipo que usa o operador '*not*', a negação da fórmula é verdadeira se o interpretador falhar para derivar a fórmula usando os fatos e regras no programa. O outro tipo de negação, denotada pelo operador '~', chamada de negação forte, é usada para expressar que um agente *acredita explicitamente que algo seja falso.*

Exemplo: ~porta(open)[source(self)]

Uma outra forma de representar crenças é a utilização de regras, que permitem concluir novas informações (crenças) baseadas em coisas que já sabemos. No exemplo abaixo adquirimos a crença de ligar carro (ligar(carro)) baseada nas crenças de que a porta está fechada e o motorista estar pronto.

Exemplo: ligar(carro) :- porta(fechada) & motorista(pronto)

3.1.2 – Objetivos

São declarados pelo agente, expressa as propriedades de estados do mundo que o agente deseja obter. Quando representamos um objetivo (*g*) em um programa significa que o agente está comprometido em agir de modo para fazer mudanças no ambiente para um estado no qual o agente acredita que *g* é de fato verdadeiro,.

AgentSpeak(L) distingue dois tipos de objetivos: *objetivos de realização* e *objetivos de teste*; objetivos são predicados, tais como crenças, porém com operadores prefixados '!' e '?', respectivamente:

Objetivos de realização (*achievemente goals*): expressam os desejos do agente; estados no ambiente que ele quer alcançar onde o predicado associado ao objetivo é verdadeiro. Na prática, esses objetivos iniciam a execução de subplanos.

Objetivos de teste (*test goals*): retornam a unificação do predicado de teste com uma crença do agente ou pode falhar quando não existir nenhuma crença que seja satisfeita.

3.1.3 – Planos

Planos fazem referência a *ações básicas* que um agente é capaz de executar em seu ambiente, sendo composto por um evento ativador, contexto e corpo. O *evento ativador* que denota o propósito do plano, ou seja, em quais eventos específicos o plano é usado, seguido de uma conjunção de literais de crença representando um *contexto*. O *contexto* relata um aspecto importante de um sistema reativo e é usado para checar se o plano escolhido será considerado *aplicável*.

A sintaxe de um plano é representada: evento_ativador : contexto <- corpo. Um exemplo de um plano: +!abre(porta): porta(fechada) <- abrir.

Os planos são sensíveis ao *contexto*, por exemplo, necessitam que certas condições (crenças) sejam satisfeitas para serem executados, sendo que o contexto deve ser uma conseqüência lógica da base de crenças do agente no momento em que o evento é

selecionado pelo agente para o plano ser considerável aplicável. O *corpo 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. Essas *ações* são definidas por predicados com símbolos especiais (chamados símbolos de ação) usados para distinguir ações de outros predicados. As ações básicas do agente podem ser classificadas de duas maneiras, sendo elas:

a) Ações internas são prefixadas por um ponto (“.”) e não alteram o ambiente.

   Exemplo: .print(“Crença:”); esta ação imprime no console do Jason a frase “Crença”.

b) Ações externas são ações que o agente efetua no ambiente.

   Exemplo: mover(3,2) ; esta ação move para uma coordenada (x,y) passada como argumento para o ambiente.

Um *evento ativador* (*triggering event*) define quais eventos podem iniciar a execução de um plano considerado verdadeiro. Um *evento* pode ser interno, quando gerado pela execução de um plano em que um subobjetivo precisa ser alcançado, ou externo, quando originados pela percepção do ambiente ou das ações dos outros agentes. 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 (‘-’).

Quando o agente percebe informações sobre o ambiente, é gerado um evento com esta percepção, sendo adicionado a sua base de crenças. É gerada uma lista com os planos que podem ser executados com essa percepção e testado o contexto do plano para verificar quais planos podem ser aplicados. Por fim é selecionado um plano da lista de planos e o agente executa o plano selecionado.

3.1.4 – Ciclo de Raciocínio

Apresentaremos em detalhes o funcionamento de um interpretador *AgentSpeak*(L). Na Figura 3.3 [MAC01] o conjunto 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)*. Alguns componentes estão rotulados com números para identificação e descrição no texto.

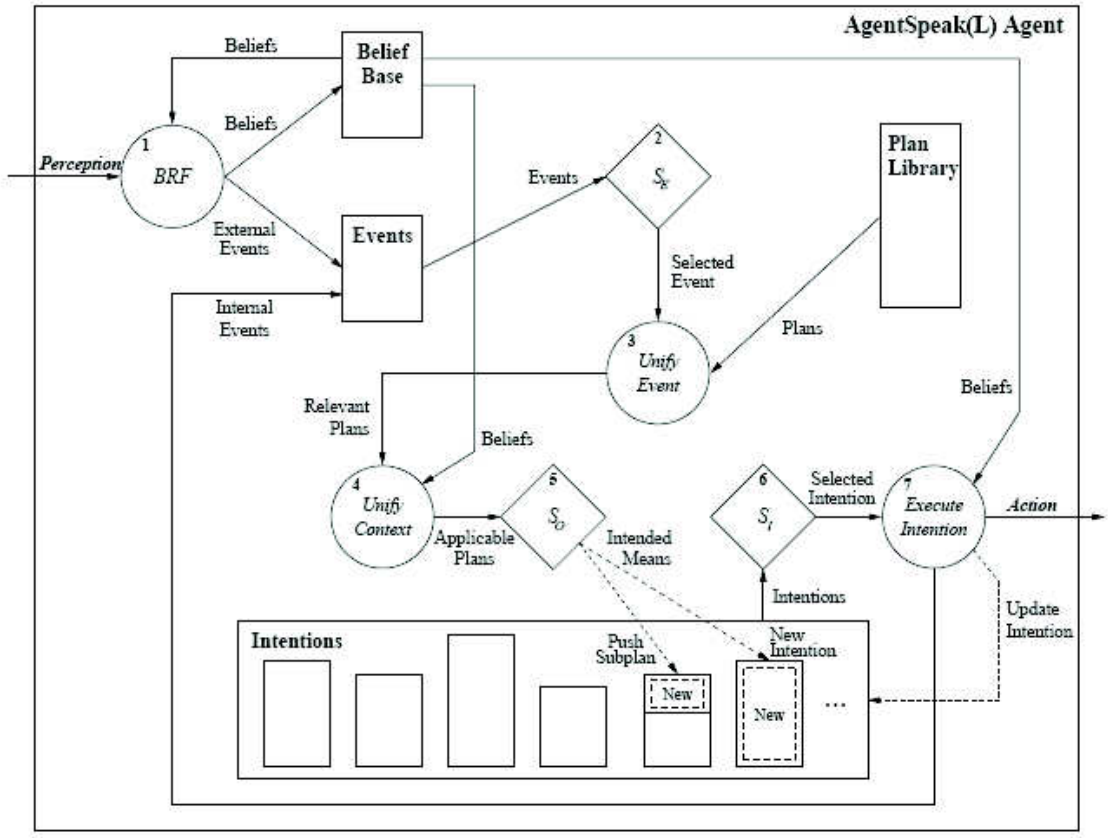

Figura 3.3: Ciclo de interpretação de um programa em *AgentSpeak(L)* [MAC01]

A função de revisão de crenças (BRF, rótulo 1) recebe as percepções do ambiente, adicionando um evento na lista de eventos e uma crença a cada percepção recebida. O conjunto de eventos é atualizada através das percepções do ambiente ou através da execução de novas intenções (por exemplo, quando subobjetivos são especificados no corpo do plano). A cada adição ou remoção de crença é gerado um evento. Eventos também podem ser gerados quando o agente se compromete a realizar um objetivo. A 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).

Após ter selecionado um evento S_E (rótulo 2), o interpretador AgentSpeak(L) unifica o evento selecionado com eventos ativadores existentes, para isso é necessário verificar a biblioteca de planos do agente (rótulo 3). Isso gera um conjunto de todos os planos relevantes para o evento escolhido, através da unificação do contexto dos planos. É verificado o contexto de cada plano (rótulo 4) para gerar um conjunto de planos aplicáveis (planos que podem ser usados, na situação presente, para tratar o evento selecionado

naquele ciclo) e descartar os planos que não podem ser aplicados. Depois, S_A (rótulo 5) escolhe um único plano do conjunto de planos aplicáveis 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, por exemplo gerado pela percepção do ambiente) [BOR04], 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 (rótulo 6) seleciona uma intenção do agente (as intenções do agente são instâncias de planos 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 (rótulo 7), implicando que 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* (verificando a 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 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 do conjunto de intenções. Quando todas as fórmulas no corpo de um plano forem removidas (por exemplo, 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.

Apresentamos informalmente alguns aspectos importantes da semântica e do ciclo de raciocínio do interpretador Jason. A semântica formal da linguagem e o ciclo completo são encontrados no livro [BOR07].

3.1.5 – Sintaxe SMA

A definição do SMA deve incluir um conjunto de agentes *AgentSpeak* e um ambiente no qual todos agentes estão situados. Estas definições devem estar informadas em um arquivo texto com extensão `.mas2j` e em conformidade com a sintaxe definida pela gramática apresentada na Figura 3.3, na qual:

<NUMBER>: é um número inteiro;

<ASID>: são identificadores AgentSpeak iniciados por letra maiúscula;

<ID>: qualquer identificador;

<PATH>: nome do arquivo e caminho.

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

Figura 3.4: Gramática de arquivo *.mas2j* [HÜB04]

Na gramática apresentada na Figura 3.4, o nome do projeto está indicado no <ID> que segue a palavra reservada '`MAS`'. A palavra reservada '`architecture`' é usada para indicar qual a arquitetura utilizada na criação dos agentes, sendo a opção padrão '`Centralised`'.

O valor que segue a palavra '`environment`' é o nome da classe *Java* que implementa o ambiente (opcionalmente é possível informar o nome do computador em que o ambiente irá executar).

A palavra reservada '`agent`' é usada para definir o conjunto de agentes que farão parte do SMA. Inicialmente um agente é definido por seu nome simbólico, podendo ser seguido do nome do arquivo código *AgentSpeak* desse agente. Como padrão, o Jason assume que estes códigos estão em arquivos com o nome '`<name>.asl`', onde '`<name>`' é o nome simbólico do agente.

Para indicar o número de instâncias que deverão ser criadas para o agente, pode ser utilizado um número precedido de '#' (desta forma o nome dos agentes terão como sufixo um número seqüencial iniciando em 1).

3.1.6 – Sintaxe de Agentes

A gramática abaixo especifica a sintaxe AgentSpeak de programação de agentes que é aceita pelo *Jason* (Figura 3.5) [HÜB04], onde:

- <ATOM> : é um identificador que inicia com letra minúscula ou '.';
- <VAR>: (uma variável) identificador que inicia com letra maiúscula;
- <NUMBER>: qualquer tipo inteiro ou real;
- <STRING>: tipo string delimitada por aspas duplas.

Os predicados podem ter notações, sendo que uma notação é composta por uma lista de termos delimitadas por colchetes as quais seguem predicado. As notações na base de crenças são utilizadas para registrar as fontes das informações (do ambiente, interna, de outros agentes, entre outras). Os átomos *'percent'* e *'self'* são considerados especiais e são usados para indicar a origem de uma crença. As crenças iniciais que fazem parte do código fonte de um agente são chamadas de crenças internas (notação '*[self]*'), a menos que o predicado tenha outra notação definida pelo programador.

Os agentes ficam situados em conjunto de instâncias de agentes, possuindo uma biblioteca de 'ações internas' (qualquer símbolo iniciado ou que tenha '.' em algum lugar, denota uma 'ação interna') que estão definidas na base BDI as quais podem ser referenciadas nos planos dos agentes. No Jason, as ações internas são definidas pelo usuário na linguagem Java. Segue abaixo algumas ações internas que fazem parte da biblioteca padrão [BOR07]:

.send: usada para enviar mensagem a um agente; O primeiro argumento (receiver) é simplesmente o nome do receptor da mensagem; os valores disponíveis para o segundo

argumento são *tell, untell, achieve, unachieve, tellHow e untellHow*. O terceiro argumento é o conteúdo da mensagem.

.broadcast: envia mensagens a todos agentes da sociedade;

.print: usado para imprimir mensagem na console;

.desire: ação que recebe um literal como argumento e tem sucesso se o literal for um dos desejos do agente;

.sum: esta ação recebe termos como argumento 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;

.intend: similar ao **'.desire'**, mas refere-se a uma intenção específica.

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

Figura 3.5: Gramática aceita pelo Jason [HÜB04]

3.1.6 – Exemplo de um Projeto utilizando a Ferramenta Jason

A ferramenta Jason utiliza três tipos de arquivos: configuração SMA, agente e ambiente. A configuração do SMA é feita no arquivo de extensão ".mas2j", basicamente é informado a arquitetura do SMA, pode ser "*Centralized*" ou "*Saci*" (centralizado ou distribuída), o ambiente onde os agentes estão situados e os agentes criados. O arquivo de agente, onde é feita a programação dos agentes *AgentSpeak(L),* de extensão ".asl", onde descrevemos os planos, crenças, objetivos e ações realizada por cada agente. E o arquivo de ambiente, de extensão ".java", onde são descritas as ações que serão realizadas no ambiente e adição, atualização e remoção das percepções.

Mostraremos nesta seção um exemplo de projeto utilizando alguns comandos da ferramenta Jason para aplicar e conhecer os conceitos básicos da linguagem *AgentSpeak*(L).

A descrição do nosso exemplo é a seguinte: três agentes participam desse projeto SMA, onde dois deles estão em um quarto, um agente claustrofóbico quer manter a porta sempre destrancada, enquanto o outro agente paranóico, quer mantê-la trancada. O terceiro agente é o porteiro que é capaz de trancar e destrancar a porta, agindo no ambiente, os agentes, claustrofóbico e paranóico, solicitam o seu serviço.

O arquivo de configurações do projeto de SMA utilizado na ferramenta Jason é mostrada na Figura 3.6. Na figura é informado o tipo da arquitetura (no caso, arquitetura centralizada, execução em uma única máquina), a classe *Java* que simula o ambiente (classe *AmbienteQuarto*) dos agentes e os agentes (os agentes *porteiro*, *claustrofóbico* e *paranóico*, tem seu código respectivamente nos arquivos, *porteiro.asl*, *claustrofóbico.asl* e *paranóico.asl*).

```
MAS quarto {

  infrastructure: Centralised
  environment: AmbienteQuarto
  agents: porteiro; claustrofobico; paranoico;

}
```

Figura 3.6: Arquivo de configuração MAS quarto

Na Figura 3.7 são apresentados o código *AgentSpeak* dos agentes, com suas respectivas crenças, planos (são denotados por um rótulo *Pn*, para podermos fazer referência ao planos no texto) e ações para o ambiente quarto:

Agente Porteiro (*porteiro.asl*)**:**

```
+!trancada(porta)[source(paranoico)]                    (P1)
   : ~trancada(porta)
   <- trancar.

+!destrancada(porta)[source(claustrofobico)]            (P2)
   : trancada(porta)
   <- destrancar.
```

Agente Claustrofóbico (*claustrofobico.asl*)**:**

```
+trancada(porta) : true                                 (P3)
   <- .send(porteiro,achieve,destrancada(porta)).

-trancada(porta) : true                                 (P4)
   <- .print("Obrigada por destrancar a porta!").
```

Agente Paranóico (*paranoico.asl*)**:**

```
   +~trancada(porta) : true                             (P5)
     <- .send(porteiro,achieve,trancada(porta)).

   +trancada(porta) : true                              (P6)
     <- .print("Obrigada por fechar a porta!").
```

Figura 3.7: Planos dos agentes Porteiro, Claustrofóbico e Paranóico

Como podemos observar no código acima, o agente *porteiro* tem dois planos de ações, o plano *P1* é executado sempre que o agente *paranóico* enviar o objetivo a ser alcançado "*destrancada(porta)*" e tiver a percepção de que a porta não está trancada (contexto do plano) e o plano *P2,* quando receber a mensagem de objetivo a ser alcançado "*trancada(porta)*" e tiver a percepção que a porta está trancada. No agente *claustrofóbico* temos o plano *P3* que se o agente perceber no ambiente de que a porta está trancada, é ativado o evento "*trancada(porta)"*, enviando mensagem ao agente *porteiro* que destranque a porta, com a crença atualizada (porta destrancada) é impresso mensagem de agradecimento (*P4*). Já no agente *paranóico* existe o plano *P5* para que se o agente receber a crença de que a porta não está trancada, é enviada ao agente *porteiro* para que tranque a porta, com a crença atualizada (porta trancada) envio mensagem de agradecimento (*P6*).

Ao executarmos o projeto temos a seguinte execução do programa (Figura 3.8): o agente *porteiro* executa a ação de destrancar e trancar a porta na medida em que lhe é solicitado o serviço, seja pelo agente *paranóico,* que agradece o trancamento da porta, seja pelo agente *claustrofóbico*, que agradece o destrancamento da porta.

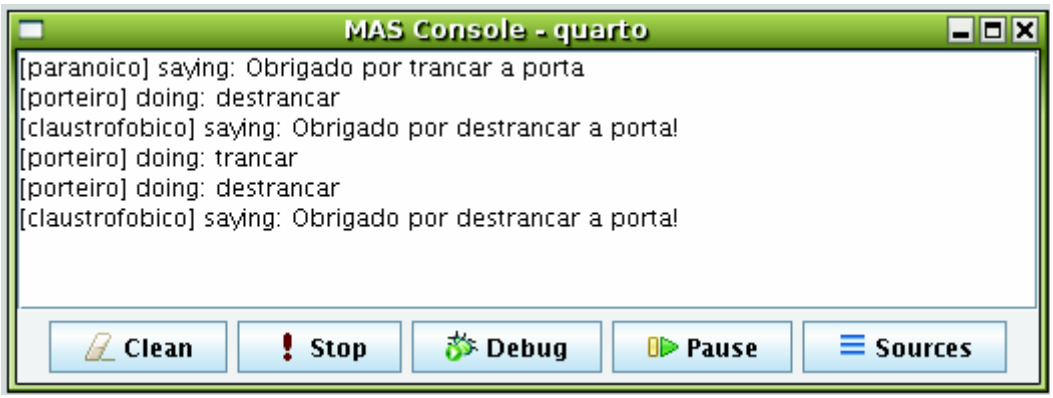

Figura 3.8: Resultado da execução do SMA no Jason

Na Figura 3.7, o agente porteiro executa as ações *trancar* e *destrancar* no ambiente, essas ações são descritas no arquivo do ambiente (*ambienteQuarto.java*), veja Figura 3.9. Em um SMA o ambiente é compartilhado por diversos agentes, com isso uma ação de um agente podem interferir nas ações dos demais agentes que compartilham desse mesmo ambiente, um ambiente é implementado em uma classe Java. Essa classe provê percepções individuais, mantendo uma estrutura de dados para armazenar percepções que os agentes tem acesso de forma individual (específica para cada agente) ou global (todos agentes tem acesso).

Criamos o ambiente no qual os agentes percebem as propriedades do ambiente e atuam através da execução de ações (trancar e destrancar), o nome da classe criada é AmbienteQuarto. Essa classe possui o método '*getPercepts*' que retorna uma lista de informações relevantes (percepções) para o agente em particular que executou o método, o programador pode incluir ou remover percepções que os agentes podem ter. O ambiente onde está sendo simulado nosso exemplo tem como percepção inicial de que porta está fechada (ld), como mostra o comando: `addPercept(ld);`

Na maioria das vezes, a maior parte do código do ambiente é descrita no método '*executeAction*'. Sempre que um agente tenta executar uma ação básica, o nome do agente e um objeto *Term* representado pela ação escolhida pelo agente são passados ao método. Desta forma, o código do método deve verificar se a ação é valida ou não, e após isso realiza 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. Em nosso exemplo temos as ações "trancar" e "destrancar", que alteram o valor da variável "portaTrancada", para true e false, respectivamente, como mostra a Figura 3.9:

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

public class AmbienteQuarto extends Environment {

    Literal ld  = Literal.parseLiteral("trancada(porta)");
    Literal nld = Literal.parseLiteral("~trancada(porta)");
    boolean portaTrancada = true;

       // Inicializa as percepções
    @Override
    public void init(String[] args) {

           addPercept(ld);
    }

    @Override
    public boolean executeAction(String ag, Structure act) {

     clearPercepts();

     if (act.getFunctor().equals("trancar"))
           portaTrancada = true;

     if (act.getFunctor().equals("destrancar"))
           portaTrancada = false;

     if (portaTrancada) {
           addPercept(ld);
     }
     else {
           addPercept(nld);
     }
     return true;
    }
}
```

Figura 3.9: Código do *ambienteQuarto.java*

3.2 - Defeito de planos

A abordagem de sistemas multiagentes para desenvolvimento de sistemas são aplicadas por áreas onde os ambientes são dinâmicos e geralmente imprevisíveis. Um plano pode apresentar um defeito ao tentar alcançar um objetivo num determinado ambiente. Da mesma forma que criamos planos com curso de ações alternativos para que os agentes utilizem para alcançar um objetivo, podemos precisar dar ao agente planos que auxiliem quando um plano selecionado apresentar um defeito. A melhor forma de pensar neste tipo de plano é com planos de contingência. Usamos eventos geradores na forma '+!g' para

planos a alcançar '*g*', e a notação de evento gerador '-!g' para identificar um plano de contingência para quando o plano '+!g' apresentar um defeito.

Antes temos que entender as circunstâncias que um plano pode apresentar um defeito, três principais causas para defeitos nos planos podem ser identificadas:

Falta de um plano relevante ou aplicável para alcançar um objetivo. O agente não sabe como alcançar algum desejo (em uma situação atual). É o caso onde um plano está sendo executado que requer um subobjetivo a ser alcançado, e o agente não consegue alcançar esse subobjetivo. Isto acontece porque o agente simplesmente não tem o "saber como", ou seja, não existem planos relevantes para aquele subjetivo (acontece com uma simples omissão de programação, onde o programador não fornece o plano requerido) ou porque todas as formas conhecidas de alcançar objetivo não são utilizadas (existem planos mas seus contextos não combinam com a base de crenças atuais do agente por isso não são aplicáveis).

Defeito de um objetivo de teste. Isto representa uma situação onde o agente 'espera' acreditar em certa propriedade do mundo, mas a propriedade não acontece como se esperava. Se não conseguirmos recuperar da base de crenças a informação necessária para satisfazer um *objetivo teste*, o interpretador tenta gerar um evento para um plano ser executado, que deve ser capaz de responder ao objetivo de teste. Se este também apresentar defeito, somente o *objetivo teste* é considerado sem sucesso e o plano aparece como defeituoso.

Defeito de ação. Ação são atuadores do ambiente, a arquitetura do agente provê ao interpretador uma resposta, que são o estado da ação, se elas foram executadas ou não. Se uma ação apresentar defeitos, o plano onde elas aparecem também apresentam defeitos.

Se um plano que deseja alcançar um objetivo (+!g) apresenta defeito, o interpretador Jason gera um evento de deleção de objetivo (na forma de '-!g'). O tipo de mecanismo de tratamento de defeitos de planos utiliza a notação de deleção de objetivo (-!g), fazendo com que o agente apague completamente esse objetivo defeituoso, escrevendo planos na forma '-!g'.

A idéia de deleção de planos tem como objetivo de limpar planos defeituosos, executando a prioridade para a possibilidade de retrocesso (*backtracking*), tomando o cuidado de não criar planos de deleção de objetivo defeituosos, gerando outros planos defeituosos.

Os programas *AgentSpeak* geram uma seqüência de ações que o agente executa no ambiente externo (alterando o estado do ambiente), os efeitos disto não podem ser desfeitos por um simples retrocesso (*backtracking*), exigindo ações adicionais no ambiente a fim de que se consiga desfazer a ação.

O programador especifica um plano de deleção de objetivo para todo objetivo adicionado com o contexto vazio e com o mesmo objetivo no corpo (-!g : true <- !g). Com isto sempre teremos uma possibilidade de retrocesso, até determinar que defeitos podem realmente acontecer no sistema.

Na Figura 3.10 [BOR07], supõe a existência da falha 'a', depois dessa ação acontece um evento para *!g2*, que foi criado mas não possui um plano aplicável para tratar esse evento, ou *?g2* não está na base de crenças, não existindo plano aplicável para tratar o evento +*?g2*. Em qualquer um desses casos, a intenção está suspensa e o evento para –!g1 é gerado. Assumindo que o programador incluiu um plano para –*!g1*, e o plano é aplicável no evento selecionado, a intenção pode ser vista na Figura 3.9(b). Se o defeito propagado retorna ao plano de um evento externo, este é recolocado enquanto a intenção é apagada, conforme configuração do interpretador *Jason*.

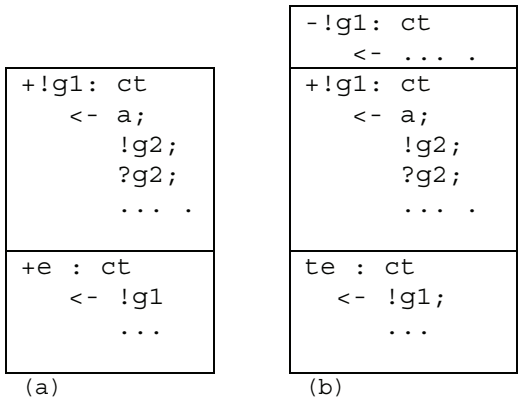

Figura 3.10: Defeito de plano de **a**. (a) uma intenção antes do plano apresentar defeito; (b) a intenção depois do plano apresentar defeito [BOR07]

Apesar do interpretador Jason apresentar uma forma de tratar defeitos em agentes, o mesmo não o faz de uma maneira organizada. Por exemplo, como lidar com defeitos em diversos agentes se os mesmos estiverem cooperando? Assim no Capítulo 4, apresentaremos uma forma organizada de trabalhar com agentes que cooperam para atingir um objetivo comum.

Capítulo 4

Interações Multiparticipantes Confiáveis

Linguagens de programação somente tratam o sincronismo entre processos, deixando para o programador a responsabilidade de garantir que os processos envolvidos em uma atividade cooperativa não interfiram, ou sofram interferência de, outros processos não envolvidos nessa atividade. Esses processos podem interagir com outros em execução, sendo necessário coordenar o sincronismo e comunicação dessas interações para que outros processos não interfiram nesta execução, isto tem que ser feito de maneira flexível e confiável. Um mecanismo que reduz o risco de um sistema apresentar defeitos e tratar possíveis falhas restantes durante o projeto/implementação do sistema, é o de Interações Multiparticipantes Confiáveis.

Interações Multiparticipantes Confiáveis (DMI) [ZOR99a] é uma abstração de controle geral que é usada para uma interação entre diversos participantes e fornece instrumentos para o tratamento de exceções concorrentes. Tem como função auxiliar na construção de atividades concorrentes complexas e auxiliar na recuperação de erros em sistemas onde existe cooperação entre as atividades concorrentes.

Uma DMI é um tipo de interação multiparticipante que fornece instrumentos para tratamento de exceções concorrentes: quando uma exceção ocorre em um dos participantes da interação, mas não é tratada por ele, a exceção deve ser propagada para todos os participantes da interação. Uma DMI também assegura a consistência na saída: um participante somente deixa sua interação quando todos tiverem terminado suas funções e os dados que são acessados competitivamente por diversas interações estiverem em um estado consistente. Algumas propriedades para interações multiparticipantes confiáveis [ZOR99a]:

- sincronização após a entrada dos participantes da interação;
- utilização de um guarda para verificar as condições necessárias para executar a interação, daí a necessidade de sincronização após a entrada;

- confirmação depois do término da interação para garantir que um conjunto de pós-condições foi satisfeita pela execução da interação;
- atomicidade de dados externos para assegurar que resultados intermediários não sejam passados para outros processos antes do término da interação.

DMI é um mecanismo de interação multiparticipante que provê facilidades para a manipulação de exceções, em particular incluindo meios de [ZOR00]:

- **Manipulação de exceções concorrentes**: quando uma exceção ocorre em um dos participantes, e se não for tratada por esse participante, a exceção deve ser propagada a todos os participantes das interações [ROM96]. Uma DMI deve prover uma maneira de tratar as exceções que podem ser geradas por um ou mais participantes. Se diversas exceções diferentes forem tratadas concorrentemente, o mecanismo DMI tem que decidir qual exceção será repassada para todos os participantes [ZOR00].
- **Assegurar consistência na saída:** um participante pode somente deixar uma interação quando todos eles tiverem finalizado as regras e os objetos externos estão em um estado consistente. Esta propriedade garante que se alguma coisa der errado na atividade executada por um dos participantes, então todos os participantes têm uma oportunidade de recuperar de possíveis erros [ZOR00].

A idéia principal para manipulação de exceções é construir uma DMI a partir de uma corrente de interações multiparticipantes, onde cada elo é o manipulador de exceções do elo anterior. A Figura 4.1 mostra como uma interação multiparticipante básica e interações multiparticipantes que tratam exceções (*exception handling*) são encadeadas para tratar possíveis exceções que são geradas durante a execução de uma DMI. Como mostrado na Figura, a interação multiparticipante básica pode finalizar normalmente, gerando exceções que são tratadas por uma exceção de interações multiparticipantes, ou gerando exceções que não são tratadas na DMI. Se a interação multiparticipante terminar normalmente, o fluxo de controle é passado para quem ativou a DMI. Se uma exceção é gerada, então existem duas possíveis execuções:

i) se existir uma interação multiparticipante para tratar aquela exceção, então ela é ativada por todos os participantes da DMI;

ii) se não existir uma interação multiparticipante para tratar a exceção gerada, então esta é passada para quem ativou a DMI.

O conjunto composto pela interação multiparticipante básica mais as interações multiparticipantes que tratam exceções formam uma entidade fechada, isto é, não são visualizadas por quem está fora da interação, de modo a evitar que alguém visualize a exceção sendo tratada internamente, fora do contexto da DMI.

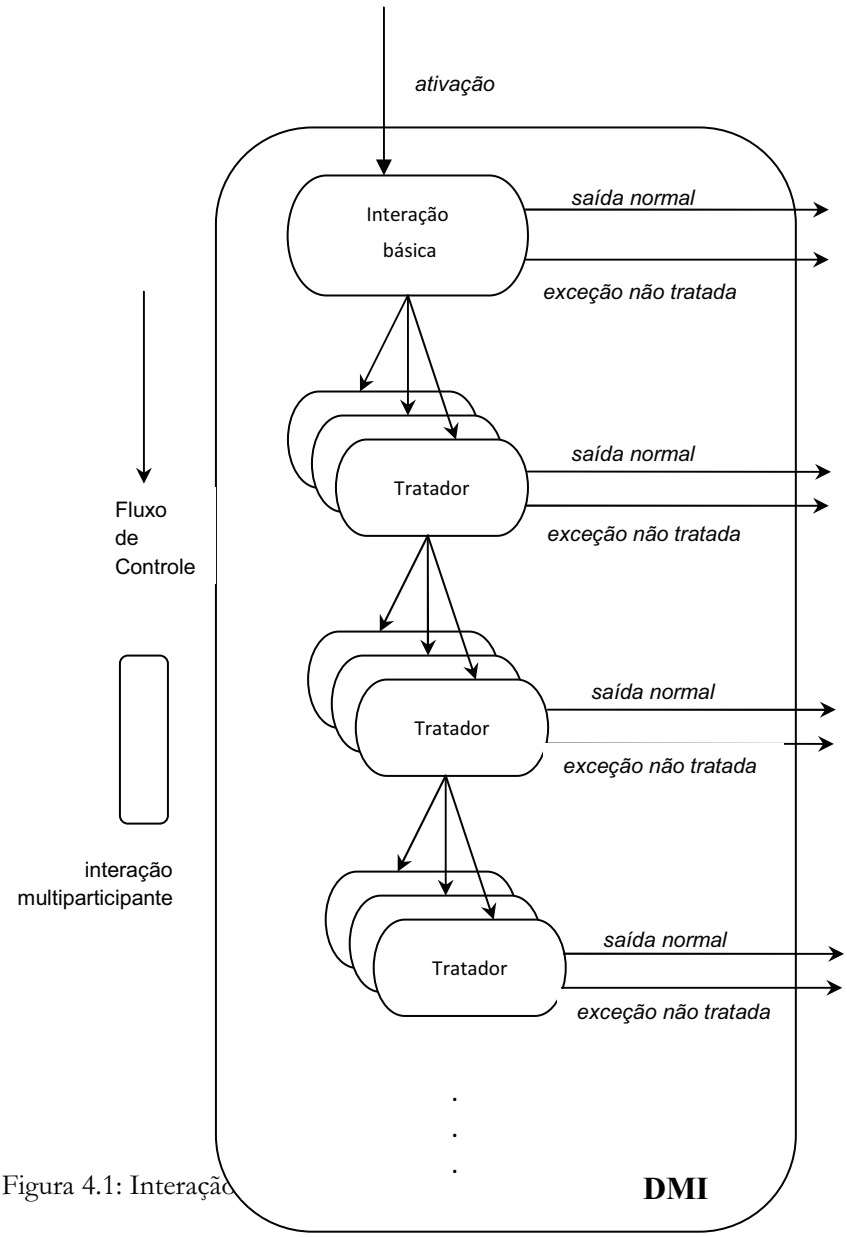

Figura 4.1: Interação

4.1 – Descrição do Framework DMI

Uma maneira de implementar uma DMI é através de um *framework* geral orientado a objetos, que trata falhas de *hardware, sofwtware* e ambiente, proposto em [ZOR99a], composto por quatro tipos de objetos:

- *Papéis (Roles)* que hospedam o conjunto de operações para os participantes das interações.

- *Gerentes (Manager)* que são responsáveis por manter o controle dos componentes da interação, gerenciar o sincronismo dos participantes, testar as pré e pós-condições para interação, e decidir qual exceção irá ser manipulada por todos os participantes da interação.

- *Objetos compartilhados (SharedObject)* que são usados para cooperação entre os participantes.

- *Objetos externos (ExternalObject)* que cuidam do estado do sistema dentro e fora da interação.

Cada DMI é representada por vários conjuntos destes objetos remotos: um conjunto para interações quando não houver nenhuma falha, por exemplo uma *interação básica* (Figura 4.1), e vários conjuntos para tratar as exceções que possam ser geradas durante a execução da interação (durante a interação básica ou durante uma *interação da manipulação de exceção*).

A Figura 4.2 mostra o diagrama de classe UML que representa o *framework*. A figura mostra que cada *role* está associado a somente um gerente, e que cada gerente controla apenas um *role*. Os gerentes são associados a um gerente especial, chamado de gerente líder, e esse é o único que é associado com objetos compartilhados. Objetos compartilhados são criados pelos *roles*, e (eventualmente) exportados para o gerente líder e desta forma tornam-se acessíveis para outros *roles,* usados para comunicação entre as *roles*. Objetos externos são associados com ambos gerentes e *roles*. Os gerentes manterão estes objetos para possível processo de recuperação. O gerente líder tem uma referência para estes objetos em ordem para informar as *roles* sobre a existência deles. Como mostrado na Figura 4.2, não existe uma classe para representar uma interação multiparticipante básica ou uma DMI.

Uma interação multiparticipante consiste do conjunto de gerentes vinculadas através de um gerente líder (representado pela associação ***leads***). Interações multiparticipantes são conectadas para criar uma DMI (representada pela associação ***activates***). Um conjunto de **gerentes,** na interação multiparticipante, ativará o conjunto de *roles* apropriadas de acordo com a exceção levantada. Este novo conjunto de regras será controlado por um diferente conjunto de gerentes.

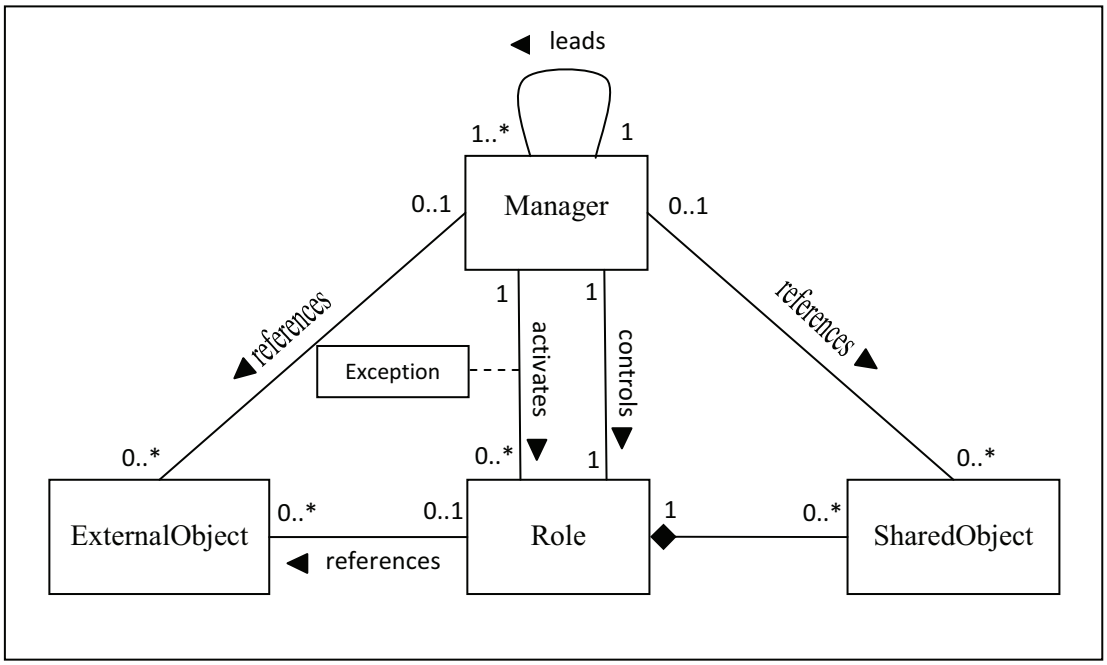

Figura 4.2: Framework DMI [ZOR99a]

Os gerentes de todos os *roles* irão compor o controle da interação. Cada gerente no momento da criação é informado de qual gerente vai atuar como *líder* na interação. O líder é responsável por controlar protocolos de sincronização entre gerentes, para o algoritmo de resolução de exceção, e para manter-se informado sobre os objetos locais compartilhados. Todos os gerentes são um potencial líder no *framework*, evitando a possibilidade de ponto único de falha.

4.1.1 – Gerente

A classe Gerente é a principal classe do *framework*. Cada *role* tem que ser controlado por um gerente diferente. Quando instanciar um objeto Gerente, o gerente tem que ser informado de seu nome, o nome da interação, o líder da interação (um gerente sem um líder é seu próprio líder) e uma lista de exceções que será tratada pelo gerente. Cada exceção na lista é associada com um *role* para manipular exceções de interações. A lista de

exceções anexada para os gerentes é a ligação entre as interações multiparticipantes de uma DMI. Exceções que são geradas em interações multiparticipantes que não são tratadas são propagadas. O seguinte comando *Java* mostra como instanciar um conjunto de gerentes para uma interação:

```
Manager mgr1=new Manager("mgr1", "DMIname", eh1, null);
Manager mgr2=new Manager("mgr2", "DMIname", eh2, null);
```

Figura 4.3: Criando Gerentes [ZOR99a]

A Figura 4.3 mostra dois gerentes que foram criados em uma mesma DMI (com *mgr1* atuando como líder de uma DMI e *mgr2* sendo liderado por *mgr1*). A tabela *eh1* contém a lista de exceções que são tratadas pelo *mgr1* e os roles que são ativados no caso de uma das exceções contidas na lista ser levantada. Todas tabelas com exceções devem conter a mesma lista de exceções a serem tratatadas e uma DMI. Se as tabelas não contiverem a mesma lista de exceções, então uma exceção é levantada em tempo de criação.

4.1.2 – Papéis

Depois que um novo objeto Gerente foi criado, o programador da interação multiparticipante tem que criar um objeto role que será controlado por este gerente. Este objeto Role tem que ser uma instância de uma nova classe role derivada de uma classe Role fornecida pelo *framework*. Cada nova classe derivada do Role contém o código principal para um dos roles que compõem a interação multiparticipante. Somente objetos cujos tipos derivam do objeto Role podem pertencer a uma interação multiparticipante. Quando derivar uma nova classe para a classe Role o programador deve implementar ao menos um método: o método *body*, o qual conterá o código da aplicação principal para este role. Este método não retorna torna nenhum valor, simplesmente recebe uma lista de objetos externos como parâmetro. Se uma exceção é originada dentro deste role, então esta exceção pode ser tratada localmente pelo role, se esta exceção não afetar outros roles. Se a exceção gerada tiver algum efeito nas outras roles, deve ser repassado para o gerente deste role, que notificará o líder e irá interromper todos os roles do DMI. Depois que todos os roles forem interrompidos o líder resolve qual exceção será tratada por todos.

Extensões de classe *role* são também responsáveis por declarar objetos locais compartilhados usados para coordenar os roles dentro de uma interação particular, e para verificar partes das pré e pós condições da interação. Depois que os objetos locais compartilhados foram criados, o role deve informar seu gerente sobre estes objetos usando

o método *sharedObject*. Este método publica os objetos locais compartilhados para que outros roles nesta interação possam usá-los mais tarde.

As pré e pós-condições de uma interação podem ser verificadas de uma maneira distribuída; cada role verifica parte das condições, ou um role pode ser delegado para verificar todas as pré e pós condições da interação. A delegação pode ser conseguida utilizando objetos locais compartilhados entre os roles. Os métodos em que os testes de pré e pós-condições são programados são chamados: *preCondition* e *postCondition*. Estes métodos podem ser refinados na nova classe role.

4.1.3 – Manipulação de exceções

Por padrão, a classe gerente fornece um mecanismo de resolução interno. Este mecanismo trabalha da seguinte forma, quando um *role* gera uma exceção, seu gerente correspondente é notificado desta exceção. O gerente então informa o líder que interrompe todos os roles que não geraram exceções. Depois que todos os roles foram interrompidos ou notificaram o gerente do líder sobre uma exceção (exceções podem ser geradas concorrentemente), um algoritmo de resolução de exceções é executado pelo líder. Este algoritmo tenta encontrar uma exceção comum para todas as exceções levantadas. Quando uma exceção comum é encontrada, o líder informa todos os gerentes sobre a exceção e um manipulador de exceções é ativado. Se não houver nenhum tratador para esta exceção, um tratador para exceções de mais alto nível é ativado, se não houver nenhum tratador para a exceção, então a exceção é sinalizada para quem ativou a DMI.

Analisamos os principais aspectos conceituais relacionados ao funcionamento de uma DMI, no Capítulo 5 aplicaremos esses conceitos para descrever um modelo de agentes BDI utilizando mecanismo de tolerância a falhas DMI, em seguida realizamos um estudo de caso onde aplicamos esse modelo de agentes BDI.

Capítulo 5

Proposta de um Modelo de Agente

Este capítulo tem como objetivo descrever e implementar a arquitetura de um agente modelado com o mecanismo de tolerância a falhas DMI. Os aspectos conceituais de agentes BDI (Capítulo 2) e da DMI (Capítulo 4) foram apresentados anteriormente e são fundamentais para apresentarmos um modelo de agentes tolerante a falhas. Na Figura 5.1 temos um modelo de agente BDI tolerante a falhas, descrevendo uma DMI genérica, onde as crenças obtidas pelo agente ativam *roles* específicas, gerando ações ou levantando exceções.

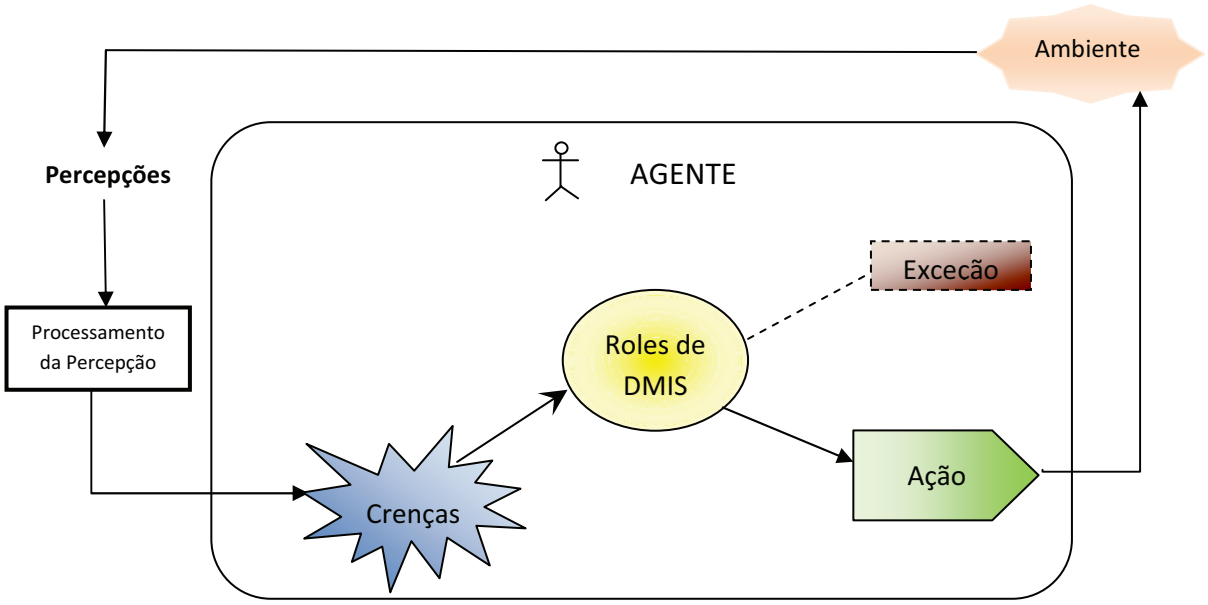

Figura 5.1: Modelo de um agente BDI tolerante a falhas

O mecanismo DMI para tratamento de falhas no modelo de agente BDI proposto permite que os agentes participantes de uma interação manipulem exceções, tratando falhas ocorridas entre os participantes, finalizando a interação apenas quando os participantes estiverem em estado consistente. As percepções do ambiente são processadas e transformadas em crenças, a execução das interações está condicionada a um determinado

estado dos participantes (verificada através das percepções obtidas através do ambiente), atendendo essa condição de guarda (crenças), inicia-se aquela interação, onde cada componente participantes da interação ativa seu plano (role) que executam as ações no ambiente como se observa na Figura 5.1.

O modelo baseia-se na perspectiva de planos a serem executados, onde cada agente tem um objetivo a ser alcançado em cada interação, cooperando com outros agentes para executar ações no ambiente, conforme as crenças adquiridas no ambiente, definindo estratégias e planos para atingir um estado consistente ao término da interação. As crenças constituem a base de conhecimento do agente, baseada nelas são ativadas as interações, optando pelas estratégias de ação mais adequadas em cada situação.

De acordo com o modelo proposto, um agente é caracterizado pelos seguintes elementos:

- Crenças: representa os conhecimentos que o agente possui do ambiente, descrevendo as características e estados do agente. São obtidas e atualizadas pelas percepções do ambiente.
- Role: são os papéis que os agentes têm em comum para realizar as interações, definindo objetivos que expressam estados que o agente deseja obter. Definindo estratégias para atingir e alcançar os objetivos pretendidos.
- Ação: agir para atingir os objetivos, transformando o ambiente de um estado para o estado que o agente deseja.

Estes três elementos, em conjunto, formam o modelo proposto da descrição de um agente, que pode modelar outros agentes. A formalização da caracterização de um agente, bem como a implementação do modelo proposto será aplicado nas próximas seções.

A definição de um agente baseado no modelo de agentes tolerante a falhas, permite criarmos um modelo de agente que utiliza o mecanismo DMI, interagindo com outros agentes que possuem a mesma *role*. Deste modo, o modelo de agente proposto permite modelarmos diferentes formas de especificação. Cada agente executa ações divergentes no ambiente, e cada agente tem seu conjunto de papéis e ações no ambiente, permitindo coordenar a execução das ações no ambiente.

5.1 – Estudo de Caso

O uso de equipamentos em instalações industriais é considerado como sendo um problema complexo, principalmente realizar a produção de vários componentes, onde cada componente utiliza um subconjunto de equipamentos disponíveis. Para auxiliar em uma utilização racional modelamos um estudo de caso da célula de produção, utilizando agentes baseados no modelo BDI.

Em Zorzo [ZOR99a] é descrito o uso de DMIs para projetar um programa de controle para o estudo de caso da célula de produção (utilizando o simulador FZI - Forschungszentrum Informatik), composta por oito dispositivos/componentes (Figura 5.2): duas esteiras de transporte (esteira de alimentação e de depósito), mesa giratória elevada, duas prensas, um robô com dois braços ortogonais, dois semáforos.

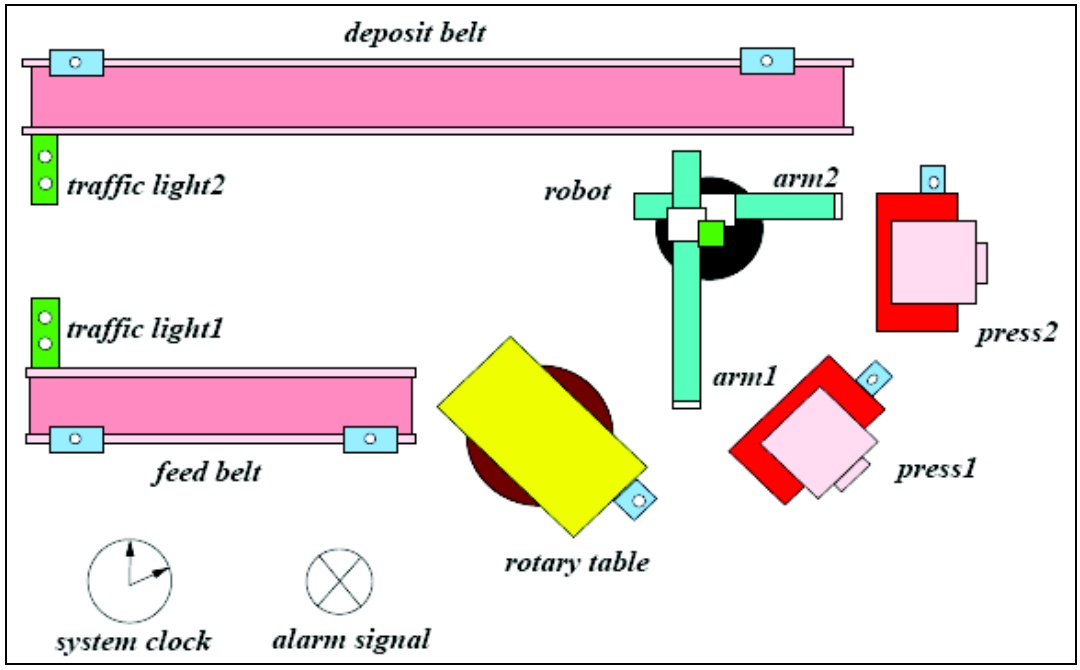

Figura 5.2: Célula de produção tolerante a falhas [ZOR99a]

O ciclo completo de produção de placas de metal pode ser descrito como segue abaixo:

i. Se o semáforo (TrafficLight1) no início da esteira de alimentação (FeedBelt) está verde, então uma placa de metal pode ser adicionada sobre a esteira de alimentação;
ii. A esteira de alimentação repassa a placa de metal para mesa giratória elevada;
iii. A mesa (RotaryTable) gira e eleva para a posição onde o robô possa agarrar a placa;
iv. O primeiro braço do robô (Robot Arm1) agarra a placa e coloca na prensa livre (Press1 ou Press2);

v. A prensa escolhida forja a placa;
vi. O segundo braço do robô (RobotArm2) remove a placa forjada da prensa e coloca sobre a esteira de depósito (DepositBelt);
vii. Se o semáforo (TrafficLight2) do fim da esteira de depósito estiver verde, então a placa é retirada da célula de produção pela esteira de depósito;

As DMIs são agrupadas para indicar quais componentes estão envolvidos naquela interação (pelas caixas pontilhadas), DMIs próximas não podem ser executadas em conjunto pois compartilham alguns componentes da Célula de Produção, como mostrado na Figura 5.3. Na Figura podemos observar que a DMI LoadPlate não pode ser executada concorrentemente com a DMI LoadTable pois compartilham o mesmo componente, os componentes "FeedBelt" e "TrafficLight1" que podem participar de apenas uma das interações por vez devido aos estados e ações serem diferentes para cada interação.

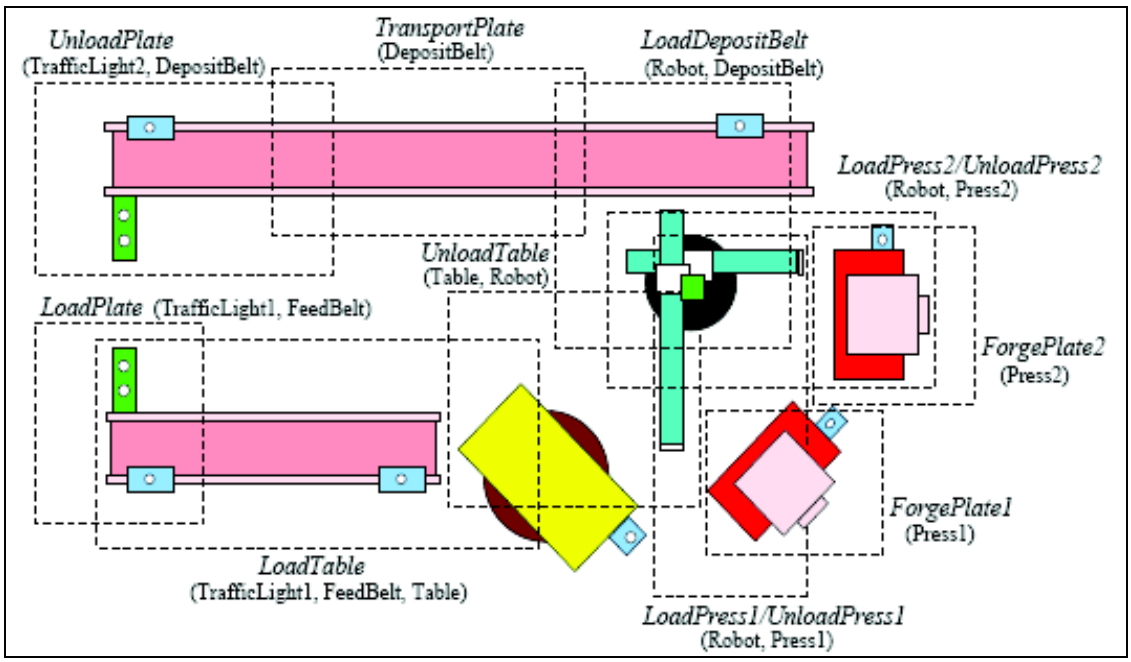

Figura 5.3: Célula de Produção [ZOR99a]

O controle da célula de produção é feito por agentes BDI implementados na Ferramenta Jason, que escalona o trabalho da célula de produção através de suas crenças e desejos, re-escalonando o trabalho da célula quando ocorrer alguma mudança do estado do sistema.

Cada agente BDI possui uma regra (role), definida como objetivo a ser alcançado, que possuem um conjunto de ações para tratar a DMI, cada role (objetivo) é específico para cada agente, e é ativado conforme as crenças recebidas do simulador da Célula de Produção. Na Figura 5.4 podemos observar o modelo de agentes criado para interação com o simulador, onde temos o agente (com sua role), que são ativadas de acordo com as

crenças recebidas pelo ambiente *AmbCP.java* (definido na ferramenta Jason), quando ativadas, geram um conjunto de operações que executam ações através do *AmbCP*. No ambiente AmbCP são feitas as atualizações na base de crenças dos agentes e enviado comandos para o Simulador da célula de produção.

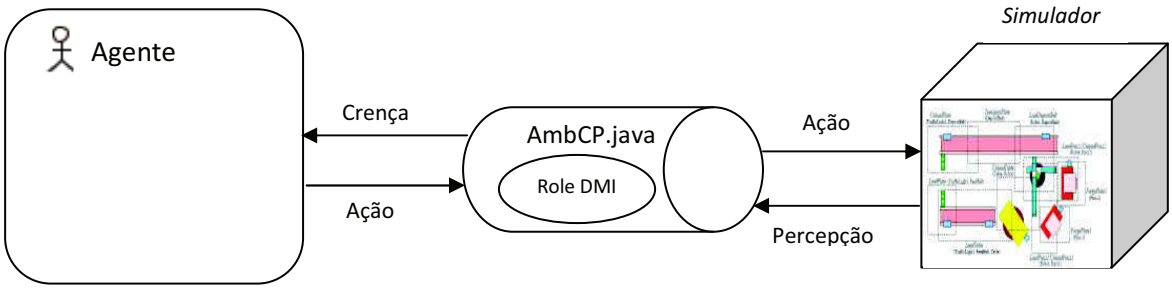

Figura 5.4: Modelo de agente para controle da Célula de Produção

Para desenvolvimento do controlador da Célula o SMA através do interpretador Jason criamos um projeto, representado pela extensão ".mas2j", onde são definidos a infra-estrutura do SMA e os agentes, conforme ilustra a Figura 5.5.

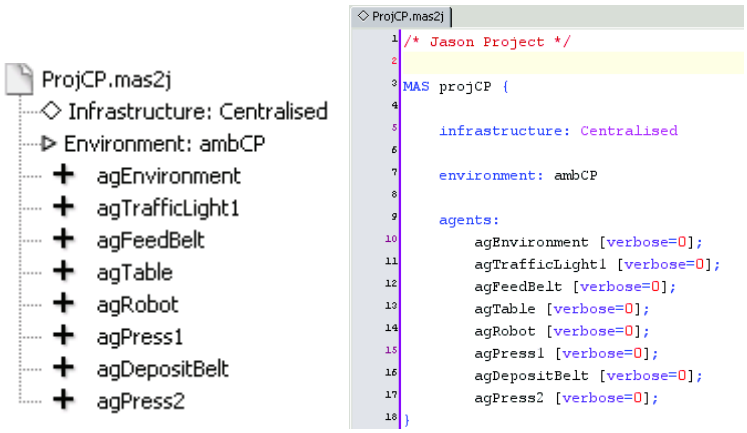

Figura 5.5: Estrutura e descrição do Projeto ProjCP na Ferramenta Jason

A infra-estrutura pode ser definida como "saci" (SACI – Simple Agent Communication Infrastructure), onde os agentes podem ser executados de maneira distribuída, ou "centralized", opção utilizada nesse trabalho pois os agentes são executados localmente no ambiente do simulador.

O identificador MAS representado na Figura 5.5 indica o nome da sociedade dos agentes, e a palavra "agents" é usada para definir um conjunto de agentes que irão fazer parte do SMA. Definido o projeto, inicia-se a programação dos agentes AgentSpeak(L), feita individualmente em arquivos com extensão ".asl".

De uma forma geral, a maior parte das ações executadas pelos agentes na utilização da ferramenta consiste em inserir, atualizar e consultar dados (crenças tratando-se de SMA), isso ocorre através de mensagens que os agentes recebem da interface.

Utilizaremos as classes que definem e acessam os dispositivos, desenvolvido em [ZOR99a], facilitando a captação dos estados dos sensores do simulador da Célula de Produção. As crenças são obtidas através da leitura de todos os sensores dos dispositivos que são lidas e atualizadas das percepções de estado do ambiente. Antes de atualizarmos as percepções utilizamos o comando "clearPercepts()" no ambiente ambCP para limpar todas as percepções do ambiente antes de adicionarmos, com isto teremos as percepções sempre atualizadas.

Para enviarmos ações ao simulador da Célula FZI utilizamos o ambiente *ambCP*, que utiliza o método "*executeAction*" definido em Jason, que envia os comandos para o simulador, agindo e alterando o estado do ambiente. No agente "Traffic Light" executamos a ação "trafficLight1_off" (linha 10 da Figura 5.6a) e no ambCP.java (Figura 5.6b) definimos o que essa ação realiza no ambiente, nesse caso envia o método trafficLight1.light_off(); (linha 322 da Figura), que desliga a luz do componente semáforo (traffic light 1), mudando o estado para vermelho.

```
5         /* Role Load Cell */
6  +feedBeltLOADED: trafficLight1GREEN
7  <- !roleLoadCell.
8
9  +!roleLoadCell: true
10 <- trafficLight1_off.
11
12        /* Role Load Table */
13 +feedBeltFREE: trafficLight1RED
14 <- !roleLoadTable.
15
16 +!roleLoadTable: true
17 <-  trafficLight1_on.
18
```

```
318         // Agente Traffic Light
319         if (action.getFunctor().equals("trafficLight1_on")){
320             trafficLight1.light_on();
321         } else if (action.getFunctor().equals("trafficLight1_off")){
322                 trafficLight1.light_off();
323           }
```

Figura 5.6: (a) chama a ação no agente e (b)descrição e definição da ação no *ambCP*

Para obter o estado da célula de produção, utilizamos os métodos que verificam o estado de cada componente e sensor da célula, utilizando uma *thread*, definida em Zorzo [ZOR99a], que fica em permanente execução lendo os sensores da célula e atualizando as percepções, gerando novas crenças conforme a leitura do estado do ambiente. Na Figura 5.7 temos a adição das crenças pela leitura do sensor do dispositivo Traffic Light, que chama o método isGreen(), se verdadeiro adiciona a percepção "trafficLight1GREEN", conforme mostrado na linha 121 da Figura 5.7. E o método isRed() que se verdadeiro será adicionada a crença trafficLight1RED (linha 123).

```
118
119                                    /* Ag Traffic Light1 */
120                 if (trafficLight1.isGreen()){
121                     addPercept(Literal.parseLiteral("trafficLight1GREEN"));
122                 } else if (trafficLight1.isRed()){
123                         addPercept(Literal.parseLiteral("trafficLight1RED"));
124                 }
```

Figura 5.7: Adição da Crença ao Agente Traffic Light1

5.2 – Agentes descritos na Célula de Produção

Cada componente da célula de produção foi definido como um agente para controle da célula de produção, descrevemos a seguir cada um dos agentes com seu conjunto de percepções, ações e objetivos que compõem o sistema multiagentes desenvolvido.

5.2.1 - AGENTE ENVIRONMENT (agEnvironment)

O **agente Environment** (agEnvironment) representa o componente ambiente que adiciona e remove a placa da Célula de Produção, conforme as crenças que são percebidas do ambiente.

Percepções: placa está no ambiente (plateOnEnv) para ser inserida na célula (Figura 5.8).

Ações: adicionar placa (blank_add) e remover placa (blank_remove) da célula de produção.

Objetivos: Execução de roles das DMIs Load Cell e Unload Cell, respectivamente, carregar e descarregar a célula de produção (Figura 5.8).

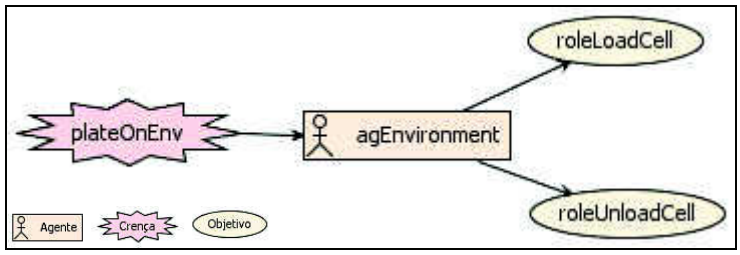

Figura 5.8: AgEnvironment com suas crenças e objetivos

A crença "plateOnEnv" (Figura 5.8) é uma crença controlada pelo programador, utiliza a ação "plateNOnEnv", que remove a crença "plateOnEnv" para que não seja colocada outra placa sem ter terminado a execução da Role DMI Load Cell e LoadTable.

5.2.2 - AGENTE TRAFFIC LIGHT (agTrafficLight1)

O **agente TrafficLight** (agTrafficLight1) representa o componente semáforo que é uma máquina de estado, podendo mudar de verde para vermelho e vice-versa. O estado inicial para o agente é verde, que significa que novas placas podem ser inseridas na célula.

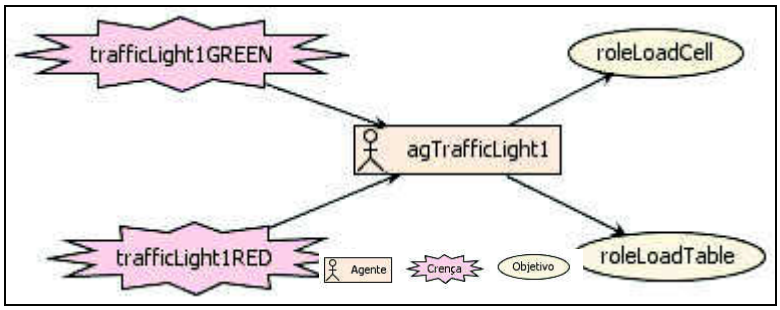

Figura 5.9: AgTrafficLight1 com suas crenças e objetivos

Percepções: semáforo verde (trafficLight1GREEN) e semáforo vermelho (trafficLight1GREEN) (Figura 5.9).

Ações: ligar e desligar o semáforo de tráfego.

Objetivos: Execução de roles das DMIs Load Cell e Load Table, respectivamente, carregar célula de produção (colocar placa na esteira de alimentação) e carregar mesa (Figura 5.9).

5.2.3 - AGENTE FEEDBELT (agFeedBelt)

O **agente FeedBelt** (agFeedBelt) representa o componente esteira de alimentação que transporta a placa de metal até o componente mesa (somente pode transportar uma placa por vez), o dispositivo funciona através de um motor elétrico que pode ser iniciado ou parado pelo agente. Possui dois sensores instalados nas extremidades da esteira que indica se a placa entrou ou saiu da esteira.

Pode ser representada por três máquinas de estado: atuador da esteira, sensor no começo da esteira e sensor no fim da esteira. O estado inicial para esteira de alimentação é sem placas no começo da esteira e sem placas no fim da esteira de alimentação.

Percepções: esteira vazia (feedBeltFREE), esteira carregada com placa (feedBeltLOADED), início da esteira carregado com placa (feedBegload). (Figura 5.10)

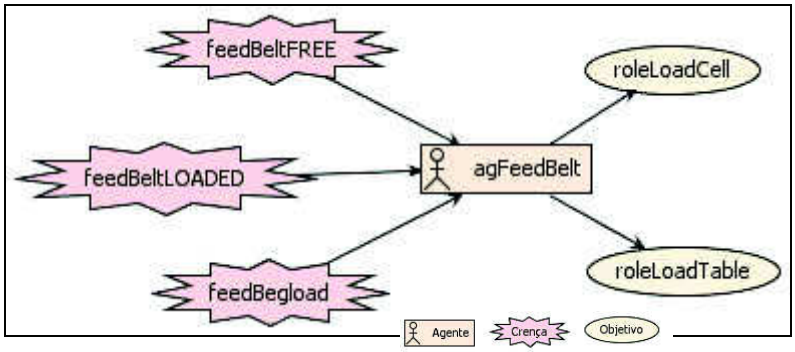

Figura 5.10: AgFeedBelt com suas crenças e objetivos

Ações: mudar o estado da esteira para carregada ou vazia, ligar e desligar esteira.

Objetivos: Execução de role das DMIs Load Cell e Load Table. (Figura 5.10)

5.2.4 - AGENTE TABLE (agTable)

O **agente Table** (agTable) representa o componente mesa que é capaz de girar e elevar, o movimento vertical (elevação) é necessário para que o braço do robô consiga pegar a placa na esteira pois estão localizadas em diferentes níveis e o robô não realiza movimentos verticais. A rotação da mesa é também necessária para que o robô pegue a placa de forma correta e a coloque na prensa na posição certa.

Pode ser representada por três estados: um para representar o estado do ângulo da mesa, uma para indicar se a mesa está em posição abaixada ou levantada e uma para indicar se a placa esta sobre a mesa.

Percepções: mesa vazia (tableFREE) ou carregada (tableLOADED), mesa abaixada (tableLOWER) ou levantada (tableUPPER), mesa posicionada para esteira (table(pos_feedbelt)) ou para o robô (table(pos_robot)). Utilizamos o sensor existente na mesa que indica se ela esta livre ou carregada, se o sensor está vermelho (sensorTRED), ou seja, existe placa na mesa, ou se está verde (sensorTGREEN) que mostra que a mesa não possui placa. (Figura 5.11)

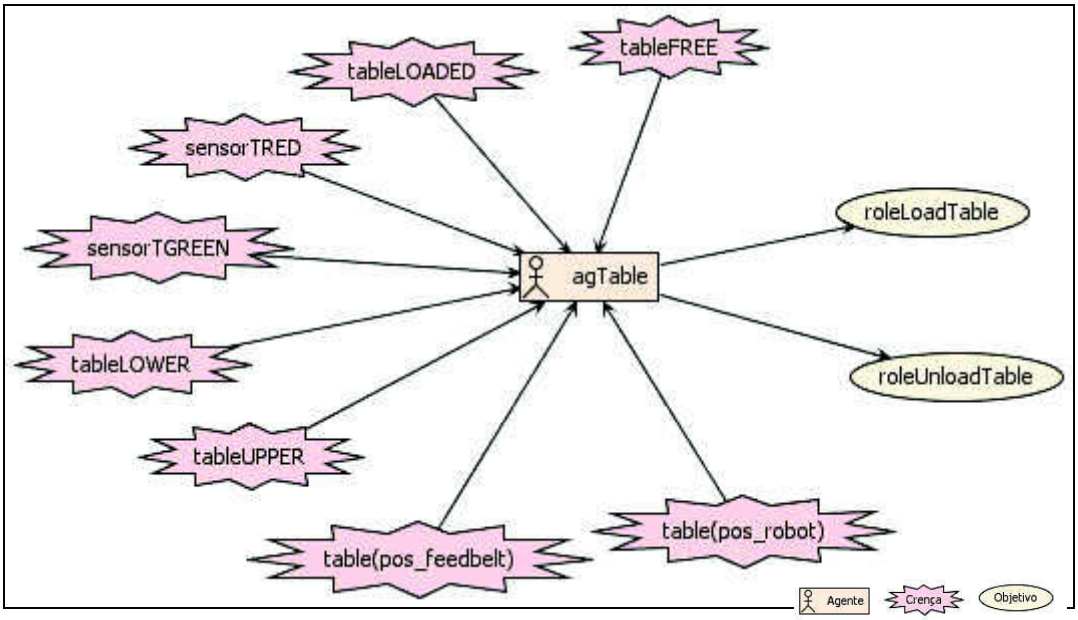

Figura 5.11: AgTable com suas crenças e objetivos

Ações: posicionar a mesa (rotacionando e elevando) para ser carregada ou descarregada, mudar o estado do ambiente da mesa para carregada ou vazia.

Objetivos: execução de role das DMIs Load Table e UnLoadTable

5.2.5 - AGENTE ROBOT (agRobot)

O **agente Robot** (agRobot) representa o componente robô que possui dois braços ortogonais fixados em dois níveis diferentes, cada braço pode retrair e estender horizontalmente. Os braços rotacionam juntos, e possuem um eletroímã em suas extremidades que permitem que o braço pegue as placas metálicas. O agente robô (braço 1) pega a placa não forjada do agente mesa e leva para o agente prensa (que estiver disponível), depois transporta a placa forjada pela prensa para esteira de depósito (utilizando o braço 2); o braço 1 retira a placa da mesa e coloca na prensa que estiver livre, enquanto o braço 2 retira a placa forjada da prensa (da que estiver com a placa já forjada) e a coloca na esteira de depósito.

Percepções: Braço 1 do robô carregado (arm1LOADED) ou livre (arm1FREE), braço 2 do robô carregado (arm2LOADED) ou livre (arm2FREE). Braço 1 do robô posicionado para mesa (robotA1(pos_table)) ou para prensa 1 ((robotA1(pos_press1)) ou para prensa 2 (robotA1(pos_press2)); braço 2 do robô posicionado para prensa 1 (robotA2(pos_press1)) ou para esteira de depósito (robotA2(pos_depositBelt)). (Figura 5.12)

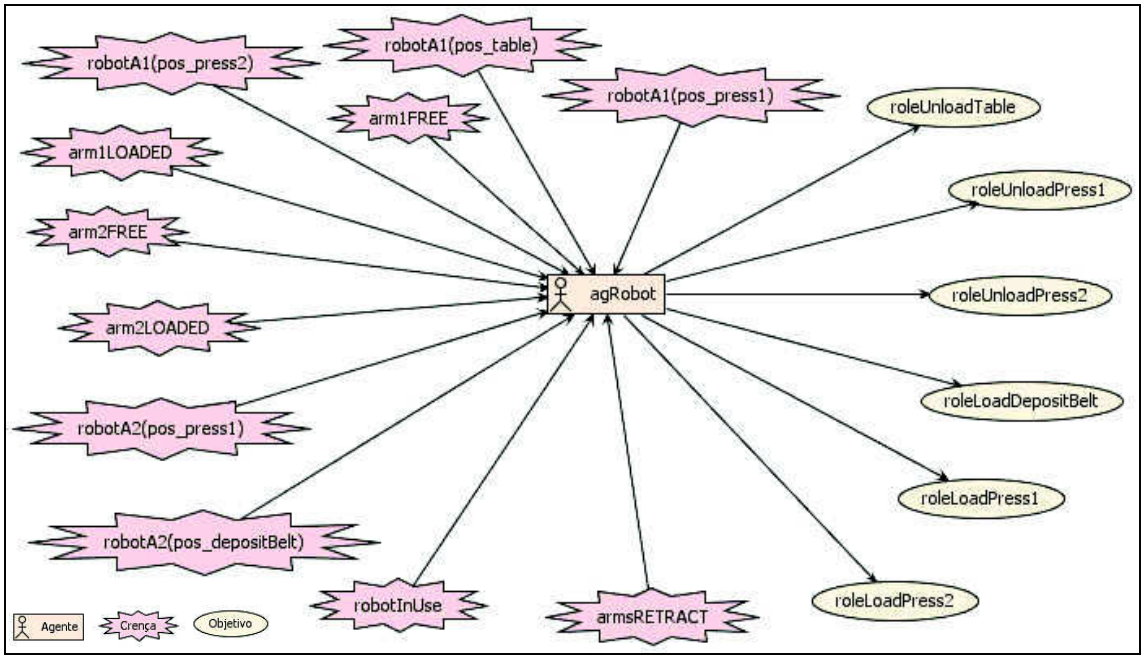

Figura 5.12: AgRobot com suas crenças e objetivos

Ações: posicionar os braços do robô, rotacionando para mesa, prensas ou esteira de depósito, pegar placas ou descarregar placas na mesa, prensas e na esteira de depósito.

Objetivos: execução de role das DMIs UnLoad Table, UnLoadPress1, UnLoadPress2, LoadPress1, LoadPress2.

A crença "robotInUse" foi criada para que o robô finalize a interação da DMI que está utilizando-o, para liberar o robô utilizamos a ação "robotNotInUse" que faz com que seja removida a crença que o robô está em uso, permitindo que outra interação utilize o robô.

A crença "armsRETRACT" é gerada quando se possui as crenças de que os dois braços do robô estão retraídos (arm2(retPos_ret) e arm1(retPos_ret)), definida como uma regra que permite-nos concluir novas crenças baseadas em crenças que possuímos. No agente agRobot a linha "armsRETRACT :- arm2(retPos_ret) & arm1(retPos_ret)." define a regra, ou seja, a nova crença, com base nas crenças obtidas.

5.2.6 - AGENTE PRESS (agPress1 e agPress2)

O **agente Press** (agPress1 e agPress2) representa o componente prensa que forja as placas. A prensa é constituída por duas plataformas horizontais, uma fica na parte inferior e se move verticalmente e a outra na parte superior, funciona pressionando a plataforma inferior contra a plataforma superior. Possui três posições: inferior (onde o

braço 2 do robô pega a placa já forjada), intermediária (onde o braço 1 do robô carrega a placa na prensa para ser forjada) e superior (forja a placa). (Figura 5.13)

Percepções: prensas carregadas (press1LOADED, press2LOADED), forjadas(press1FORGED, press2FORGED) ou livre (press1FREE, press2FREE), sensor de uso da prensas (sensorP1RED, sensorP2RED) ou não uso (sensorP1GREEN,sensorP2GREEN), posicionamento da prensa inferior (press1(lower), press2(lower)), intermediária(press1(middle), press2(middle)) ou superior (press1(upper), press2(upper)).

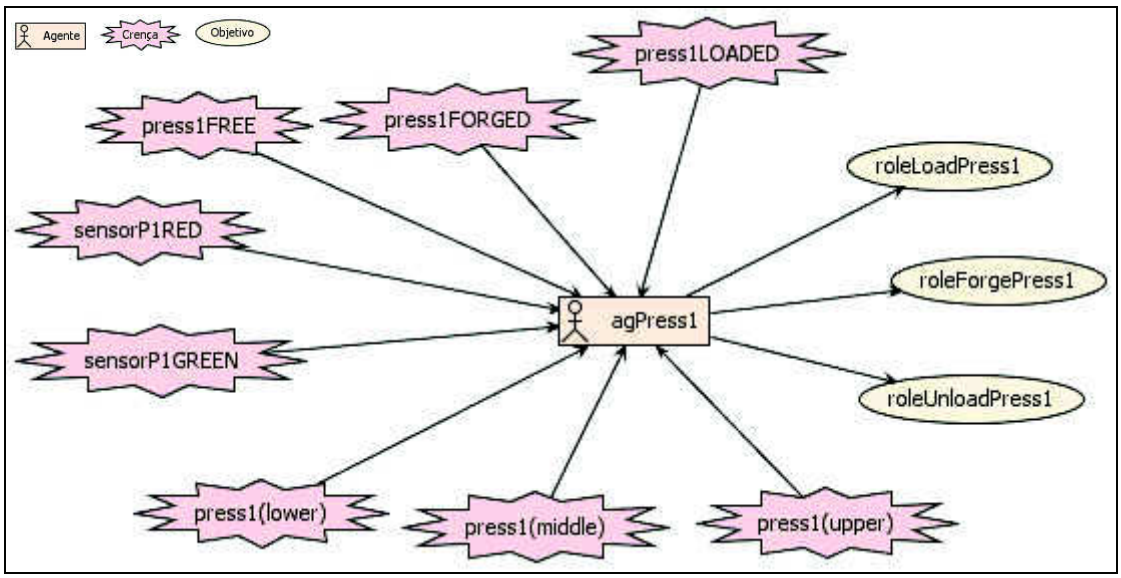

Figura 5.13: AgPress1 com suas crenças e objetivos

Ações: posicionar os braços do robô, rotacionando para mesa, prensas ou esteira de depósito, pegar placas ou descarregar placas na mesa, prensas e na esteira de depósito.

Objetivos: execução de role das DMIs *roleLoadPress1*, *roleForgePress1*, *roleUnloadPress1*.

5.2.7 - AGENTE DEPOSIT BELT (agDepositBelt)

O **agente DepositBelt** (agDepositBelt) representa o componente esteira de depósito que transporta a placa forjada para ser removida da célula de produção. Possui sensor instalado no fim da esteira que informa quando placa de metal está no fim da esteira, parando a esteira, e reinicializada quando existir outra placa a ser transportada.

Percepções: esteira de depósito pode estar livre (depositBeltFREE), carregada (depositBeltLOADED) ou já ter transportado (depositBeltTRANSP), representadas na Figura 5.14.

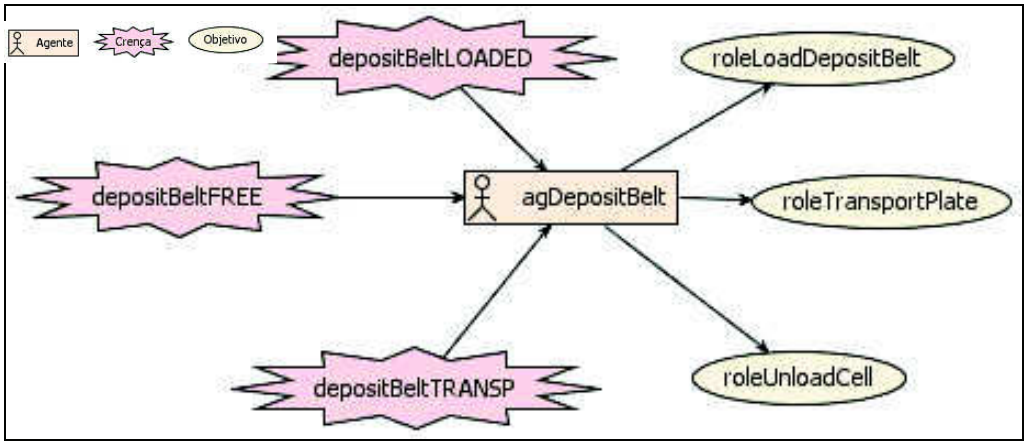

Figura 5.14: AgDepositBelt com suas crenças e objetivos

Ações: ligar ou desligar esteira de depósito, mudar o estado da percepção para esteira de depósito carregada, livre ou transportado.

Objetivos: execução de role das DMIs roleLoadDepositBelt, roleTransportPlate, roleUnloadCell.

5.3 - Implementação das DMIs nos agentes

As DMIS foram descritas como objetivo a ser alcançado para cada agente conforme as crenças percebidas no ambiente, executando a DMI correspondente com sua role que foi definida para cada agente, para melhor compreensão utilizaremos um diagrama para descrever os agentes, crenças, objetivos e ações. Adotaremos um padrão nos diagramas que serão descritos através do uso de formas geométricas utilizadas na modelagem de agentes BDI, descritas conforme ilustrado na Figura 5.15.

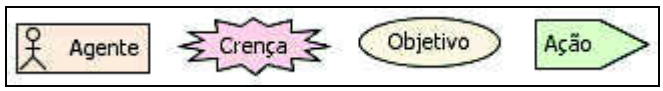

Figura 5.15: Legenda dos diagramas para agente, crença, objetivos e ações

Os diagramas descritos nessa seção mostram os agentes participantes da interação da DMI, que será tomada como um objetivo a ser alcançado pelos agentes que participam daquela interação, mediante as pré-condições na forma das crenças que serão atendidas, executando ações no ambiente. Após execução das ações dos agentes no ambiente teremos alteração do estado do ambiente e com isto atualização das crenças dos agentes.

Na Figura 5.16 temos os agentes que estão envolvidos na execução da DMI Load Cell (que tem a funcionalidade de carregar uma placa na célula de produção), esses agentes trabalham paralelamente, cada um com seu conjunto de crenças e planos, interagindo de forma que cada agente ative, conforme as crenças obtidas através da percepção do ambiente, sua respectiva ação (denominada como *role*) no ambiente do simulador (utilizamos as roles descritas por Zorzo [ZOR99a]); as ações são descritas na Figura como roleEnvironmentLC, roleTrafficLightLC e roleFeedBeltLC, juntas formam a DMI Load Cell.

As crenças e as ações ativadas por cada agente para carregamento da célula de produção estão descritas na Figura 5.16, onde mostramos quais os agentes envolvidos na interação da DMI LoadCell (agentes Environment, FeedBelt e TrafficLight1).

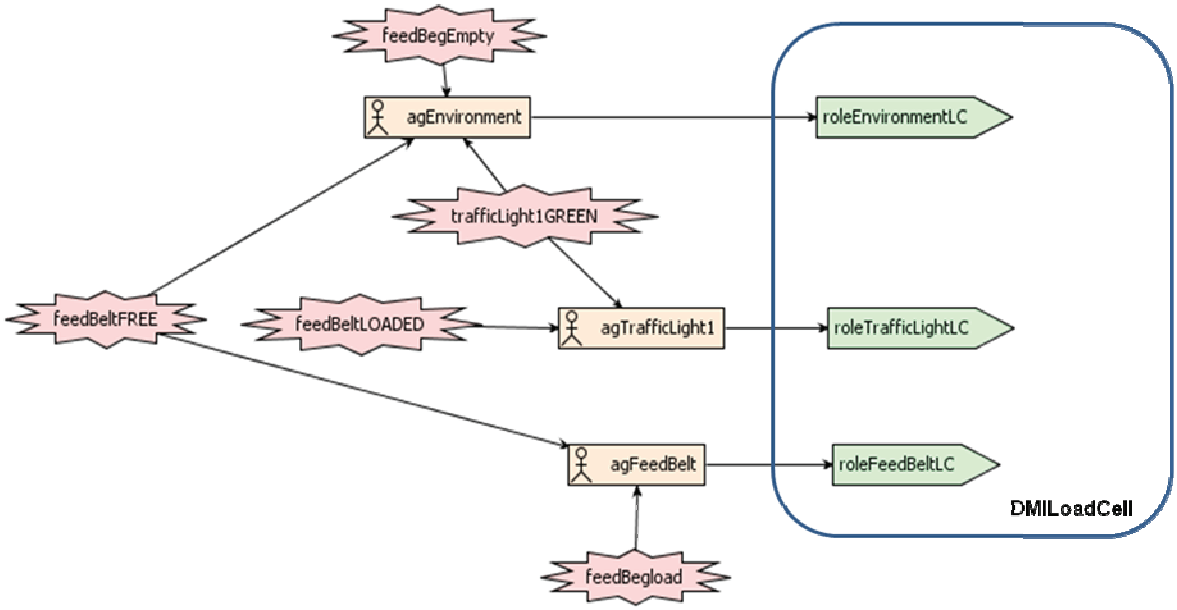

Figura 5.16: Agentes da DMI Load Cell

A DMI Load Table permite a interação entre três agentes da célula (TrafficLight1, Table, FeedBelt), a DMI tem a seguinte funcionalidade: uma placa é enviada para DMI através do componente esteira de alimentação e esta é enviada ao componente mesa, que se for preciso se posiciona para receber a placa, em seguida o componente semáforo abre o tráfego no início da esteira de alimentação.

As ações (roles) referentes a DMI Load Table são iniciadas se o conjunto de estados de crenças do agente forem respeitadas, os agentes utilizam e reagem com as seguinte crenças:

- **agTable**: mesa levantada (tableUPPER), posicionada para o robô (table(pos_robot)) e vazia (tableFREE/sensorTRED);
- **agFeedBelt**: esteira carregada (feedBeltLOADED), mesa livre (tableFREE/sensorTGREEN) e abaixada (tableLOWER);
- **agTrafficLight1**: semáforo desligado (trafficLight1RED) e esteira livre (feedBeltFREE);

Atendendo as pré-condições (crenças), inicia-se a execução nos agentes das suas respectivas roles (denominadas roleTableLT, roleFeedBeltLT e roleTrafficLight1LT), cada agente verifica as suas crenças para dar início as ações contidas nas roles de cada agente.

Na Figura 5.17 mostramos as crenças e ações dos agentes para iniciarem a interação de carregamento da mesa, a DMI LoadTable. As crenças de que a esteira está carregada (feedBeltLOADED), mesa está livre (tableFREE e sensorTGREEN) e posicionada para receber a placa (tableLOWER), fazem com seja executado a role (ação) "roleFeedBeltLT" do agente FeedBelt (que faz com que se ligue a esteira de alimentação e a desligue assim que a mesa estiver carregada, alterando o estado da esteira para livre). No Agente Table a crença do sensor da mesa ligado (sensorTRED) e que a mesa está livre (tableFREE) fazem com seja executada a ação roleTableLT (mudando o estado da mesa para carregada). No agente TrafficLight1, as crenças de que a esteira está livre e o semáforo está vermelho fazem com seja executada a ação "roleTrafficLight1LT" (mudando o estado do semáforo para verde).

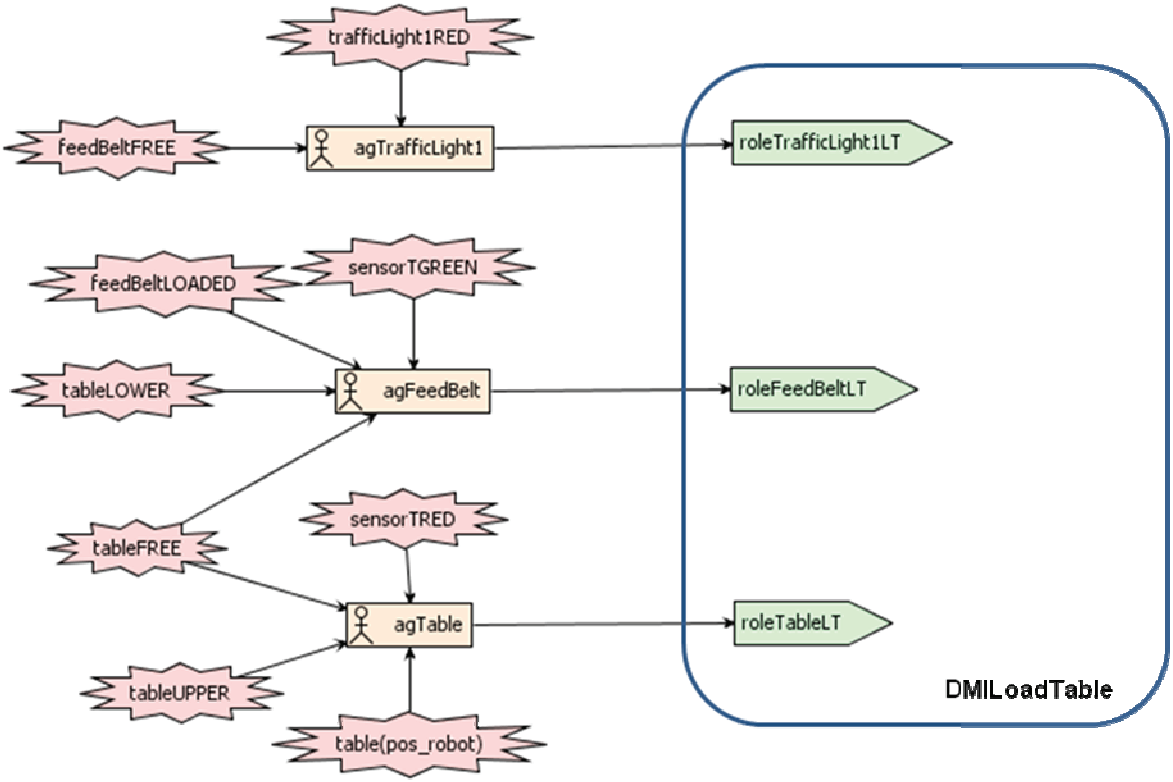

Figura 5.17: DMI Load Table

Descreveremos a seguir o código de programação do interpretador Jason nos agentes para ativar a DMI Load Table em todos os agentes envolvidos nessa interação DMI:

- **agFeedBelt** (Figura 5.18) tendo a crença de que a esteira está carregada (feedBeltLOADED) e o contexto de que a mesa está livre (sensorTGREEN) e a mesa não está levantada (not tableUPPER), executa a ação feedBeltLT, que no ambiente (ambCP.java) do Jason executa a role feedBeltRoleLT descrita em Zorzo [ZOR99a].

```
agFeedBelt.asl | agTable.asl | agTrafficLight1.asl | ambCP.java
10
11                  /* Role Load Table */
12  +feedBeltLOADED: tableFREE & sensorTGREEN & not tableUPPER
13  <- feedBeltLT.
14
agFeedBelt.asl | agTable.asl | agTrafficLight1.asl | ambCP.java
326
327                                     /*  Agente FeedBelt  */
328             if (action.getFunctor().equals("feedBeltLT")) {
329                 feedBeltRoleLT.execute();
330             } else if (action.getFunctor().equals("feedBeltLC")){
331                     feedBeltRoleLC.execute();
332             }
333
```

Figura 5.18: Crenças que ativam a ação feedBeltLT no ambCP do agente FeedBelt

- **agTable** (Figura 5.19), a crença "sensorTRED", que identifica que o sensor da mesa está ligado, ou seja, a mesa está carregada e tendo como verdadeiro o contexto de que

a mesa está livre, executando a ação tableLT, que no ambiente (ambCP.java) do Jason executa a role tableRoleLT descrita em Zorzo [ZOR99a].

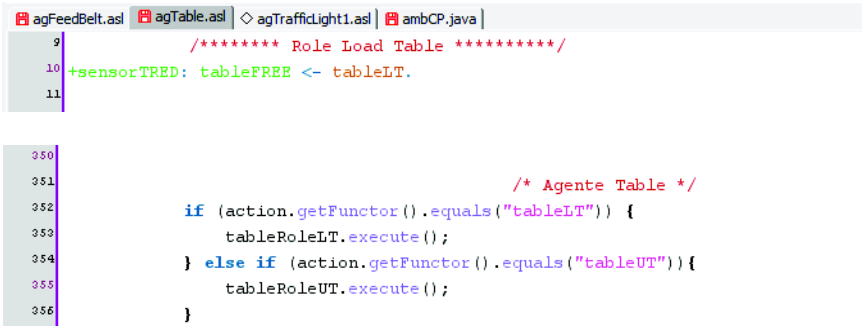

Figura 5.19: Crenças que ativam a ação tableLT no ambCP do agente Table

- **agTrafficLight1** (Figura 5.20), tendo a crença de que a esteira está livre, mas possuindo o contexto/crença de que o semáforo está verde, executando a ação trafficLightLT, que no ambiente (ambCP.java) do Jason executa a role trafficLightRoleLT descrita em Zorzo [ZOR99a].

Figura 5.20: Crenças que ativam a ação trafficLightLT no ambCP do agente TrafficLight1

As roles feedBeltRoleLT, tableRoleLT e trafficLightRoleLT descritas no ambiente da célula de produção (ambCP.java) são executadas utilizando pacote de métodos descritas por Zorzo [ZOR99], as ações que cada role executa são descritas abaixo:

- feedBeltRoleLT: a esteira é ligada, enviando a placa para a mesa, que quando carregada, a esteira é desligada, alterando o estado da esteira para vazia/livre.

- tableRoleLT: verifica se a mesa está abaixada para receber a placa, se não estiver posiciona a mesa, em seguida a mesa recebe a placa e muda seu estado para carregada.

- trafficLightRoleLT: se a esteira terminou de enviar a placa para mesa, o estado do semáforo é alterado para verde.

Alguns sensores e estados dos componentes das células são lidos e geram as crenças (sensores que verificam se a placa está próximo dele ou não) e outros sensores tem que receber a ação que levará a mudança do estado daquele componente, como por exemplo, mesa livre ou carregada, semáforo ligado (verde) ou desligado (vermelho), esteira vazia (feedBeltFREE) ou esteira carregada (feedBeltLOADED), esses estados são modificados através de comandos enviados ao ambiente pelo simulador da Célula de Produção, permitindo alterarmos o estado daquele componente através das ações realizadas no ambiente.

Cada agente descrito na Célula de Produção FZI trata um conjunto de exceções, geralmente referente a problemas que podem ser gerados por aquele componente da célula de produção, o tratamento é feito internamente por cada role executada. Caso seja levantada uma exceção por alguma role participante de uma interação DMI, role suspende a interação e tenta resolver a exceção, se não conseguir repassa as outras roles participantes para que as mesmas tentem tratar a exceção, se nenhuma das roles conseguirem tratar a exceção que foi levantada a DMI finaliza e informa que não conseguiu tratar exceção, deixando o sistema em um estado consistente antes de finalizar.

As exceções são tratadas internamente pelas roles que são executadas no ambiente "ambCP.java", tratando exceções concorrentes caso ocorram. Por exemplo, a DMI Load Table tem um conjunto de exceções para cada agente da interação, ou seja, mesa (table), semáforo (trafficLight1) e esteira de alimentação (feedBelt) tem uma lista de soluções para problemas que podem ocorrer durante uma interação. O agente Table pode ter problemas ao girar, esse problema é tratado dentro da role onde ocorreu a exceção, por exemplo, se ocorrer na role feedBeltRoleLT, esta suspende a interação DMI Load Table, tentando tratar a exceção, se não conseguir repassa para as duas outras roles participantes, trafficLightRoleLT e tableRoleLT, nessa última se consegue tratar a exceção e finalizar a interação.

Capítulo 6

Conclusão

Este trabalho apresentou um estudo de tolerância a falhas em SMA, onde alguns elementos podem apresentar falhas e outros continuam funcionando, dificultando o tratamento de falhas. A tolerância a falhas é uma das formas de se atingir a confiabilidade de um sistema, permitindo que um sistema continue funcionando mesmo na presença de falhas. DMI é um mecanismo de tolerância a falhas, onde diversos participantes interagem para executar alguma atividade em conjunto possibilitando o tratamento de problemas que podem ocorrer durante essa execução.

Os participantes interagem de forma dinâmica, alterando o estado e percepção do ambiente, as características da interação se identificam com a teoria de agentes BDI, que também apresentam comportamento dinâmico e são capazes de avaliar o estado do ambiente e tomar decisões de ações. Os participantes foram descritos utilizando o modelo de agente BDI, interagindo com outros agentes participantes da interação (DMI) e fornecendo instrumentos para tratamento de exceções concorrentes.

O estudo de caso foi desenvolvido em uma ferramenta que permite modelar um SMA, criando agentes que tem a responsabilidade de planejar o conjunto de ações que são executados. Esse conjunto de ações foram executados no simulador da Célula de Produção FZI, permitindo refletir resultados no ambiente que os agentes estavam inseridos.

Foram definidas nove interações (DMI) que são ativadas pelo agente através das crenças obtidas pela leitura dos sensores dos estados da célula de produção, cada dispositivo da célula corresponde a um agente que executa uma ação (role DMI) em conjunto com outros agentes e quando estão interagindo (executando role em comum) formam a DMI. Cada agente controla um passo (role) do processamento da placa e repassa a placa entre dois dispositivos, formando a DMI.

Os agentes foram desenvolvidos permitindo que sejam inseridos placas à medida que as DMIs são finalizadas com sucesso, fazendo que diversas interações sejam

executadas ao mesmo tempo. A execução de cada DMI está condicionada a atender as pré condições descritas em cada objetivo para execução de cada DMI em seu agente correspondente. Não foi possível implementar da injeção de falhas nesse modelo de agentes, ficando dessa forma como trabalho futuro a ser desenvolvido.

A vantagem da abordagem de agentes é o comportamento de cada agente é definido de forma simples, sem executar planos complexos, onde cada plano executa ação que corresponde a uma role de DMI do agente, que em conjunto com outros agentes executam uma DMI. As interações entre os agentes são delimitadas pela interação multiparticipante confiável (DMI), resultando em uma forma alinhada de implementar agentes que representam os dispositivos reais na célula de produção.

O uso da arquitetura BDI utilizando um mecanismo tolerante a falhas mostrou se bastante eficaz, a arquitetura proporciona a criação de agentes com certo nível de inteligência, permitindo com que as atitudes dos agentes fiquem em constante mudança conforme o estado atual da célula de produção.

Apesar do simulador FZI auxiliar na programação dos agentes da célula, este se mostrou muito lento ao se ter mais de três placas sendo processadas por diferentes DMI da célula. Isso pode ter sido gerado pela forma de obtenção das percepções, onde o comando é enviado freqüentemente, atualizando as crenças de uma forma muito dinâmica, aumentando consideravelmente o processamento.

Referências

[APP07] APPIO, Alisson Rafael; HÜBNER, Jomi Fred. Sistema multiagentes utilizando a linguagem AgentSpeak(L) para criar estratégias de armadilha e cooperação em um jogo tipo PacMan. Em Hübner and Grahl, pages 127-138. 2005. Disponível em: <http://www.inf.furb.br/seminco/2005/artigos/105-vf.pdf>. Acesso em: jul. 2007.

[AVI04] AVIZIENIS, Algirdas; LAPRIE, Jean-Claude; RANDELL, Brian; LANDWEHR, Carl. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Transactions on Dependable and Secure Computing,** v. 1, n. 1. Jan.-Mar. 2004.

[AVI85] AVIZIENIS, Algirdas. The n-version approach to fault-tolerant software. **IEEE Transactions on Software Engineering,** Piscataway, v.11, c. 12, p. 1491-1501, 1985.

[BLU97] BLUM, Avrim. L.; FURST, Merrick L.. Fast planning through planning graph analysis. Artificial Intelligence**, The Netherlands: Elsevier Science Publishers**, v. 90, n.1-2, p.281-300, 1997.

[BOR03] BORDINI, Rafael; VIEIRA, Renata. Linguagens de programação Orientada a Agentes: uma introdução baseada em AgentSpeak(L). **RITA**, v. 10, n. 1, 2003.

[BOR07] BORDINI, Rafael H.; HÜBNER, Jomi Fred; WOOLDRIDGE, Michael. Programming multi-agent systems in AgentSpeak using Jason. West Sussex: Wiley-Interscience, 2007. 273 p.

[BRA87] BRATMAN, Michael. "Intention, Plans and Practical Reason". Cambridge: Harvard University Press, 1987.

[CHE95] CHEN, Liming; AVIZIENIS, Algirdas. "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation". **Twenty-Fifth International Symposium on Fault-Tolerant Computing**, 1995, v., c., 27-30, p.113, Jun 1995.

[D'IN98] D'INVERNO, Mark; KINNY, David; LUCK, Michael; WOOLDRIDGE, Michael. "A formal specification of dMARS". In: Singh, M. P.; Rao, A. S.; Wooldridge, M., editors, AGENT THEORIES, ARCHITECTURES, AND LANGUAGES, v.1365 of Lecture Notes in Computer Science, p.155–176. Springer-Verlag, Germany, 1998.

[GEO87] GEORGEFF, Michael; LANSKY, Amy. "Reactive reasoning and planning". **PROCEEDINGS OF THE AMERICAN ASSOCIATION FOR ATIFICIAL INTELLIGENCE (AAAI)**, Seattle, WA: Morgan Kaufmann Publishers, 1987, p.677-682.

[HÜB04] HÜBNER, Jomi; BORDINI, Rafael; VIEIRA, Renata. "Introdução ao Desenvolvimento de Sistemas Multiagentes com Jason". ***XII Escola de***

Informática da SBC**,** v. 2, p. 51-89, Guarapuava: UNICENTRO, 2004. Disponível em: <www.inf.furb.br/~jomi/pubs>. Acesso em: jul. 2007.

[KOR98] KORTH, Henry ; SILBERSCHATZ, Abraham. Sistemas de Bancos de dados. São Paulo: Makron Books, 1998.

[MAC01] MACHADO, Rodrigo; BORDINI, Rafael. Running AgentSpeak(L) agents on SIM_AGENT. In John-Jules Meyer and Milind Tambe, 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, 2002. Springer-Verlag.

[MEN05] MENEGUZZI, Felipe Rech. Planejamento Proposicional em Agentes BDI. 2005. 116 f. Dissertação (Mestrado em Computação – Faculdade de Informática, PUCRS, Porto Alegre, 2005.

[MÓR99] MÓRA, Michael. "Um Modelo Formal e Executável de Agentes BDI". Porto Alegre: CPGCC/UFRGS, 1999. Tese de Doutorado.

[RAN75] RANDELL, Brian. System Structure for Software Fault Tolerance. **IEEE Transactions on software Engineering**, Los Angeles, v. SE-1, n. 2, p. 437-449, June 1975. Disponível em: http://portal.acm.org. Acesso em: 30 jan. 2007.

[RAO96] RAO, Anand. "AgentSpeak (L): BDI agents speak out in a logical computable language". **7TH European Workshop On Modelling Autononomous Agents In A Multi-Agent World**, v. 1038 of Lecture Notes on Computer Science**,** p. 42-55. Springer-Verlag, Eindhoven, Netherland, 1996.

[ROM96] ROMANOVSKY, Alexander; XU, Jie; RANDELL, Brian. "Exception handling and resolution in distributed object-oriented systems". ***16th IEEE International Conference on Distributed Computing Systems,*** p. 545-552. IEEE Computer Society Press, 1996.

[ROM97] ROMANOVSKY, Alexander; ZORZO, Avelino. "On distributed of coordinated atomic actions". **ACM SIGOPS Operating Systems Review**, v. 31, c. 4. ACM Press, Outubro, 1997.

[RUS96] RUSSEL, Stuart; NORVIG, Peter. Artificial intelligence: a modern approach. New Jersey: Prentice Hall, 1996.

[WEB07] WEBER, Taisy. "Tolerância a falhas: conceitos e exemplos". UFRGS, 2003. Disponível em: <www.inf.ufrgs.br/~taisy/disciplinas/textos>. Acesso em: jan. 2007.

[WEI99] WEISS, Gerhard. "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence". The MIT Press: Cambridge, 1999.

[WOO00] WOOLDRIDGE, Michael. Reasoning about Rational Agents. The MIT Press, Cambridge, MA, 2000.

[WOO95] WOOLDRIDGE, Michael; JENNINGS, Nicholas. Intelligent agents: theory an practice.London: QM and Wc, 1995.

[WOO02] WOOLDRIDGE, Michael. "An Introduction to MultiAgent Systems". John Wiley & Sons, 2002.

[ZAM01] ZAMBERLAM, Alexandre; GIRAFFA, Lúcia. "Modelagem de agentes utilizando a arquitetura BDI". Porto Alegre: Faculdade de Informática, PUCRS, 2001. (Relatório Técnico n.008)

[ZOR00] ZORZO, Avelino; STROUD, Robert. "Towards a Formal Specification of Dependable Multiparty Interactions". Porto Alegre: Faculdade de Informática, PUCRS, 2000. (Relatório Técnico n. 001).

[ZOR05] ZORZO, Avelino; MENEGUZZI, Felipe. "An Agent Model for fault-tolerant systems". **Proceedings Of The 2005 Acm Symposium On Applied Computing**, 2005. p. 60-65.

[ZOR99a] ZORZO, Avelino; STROUD, Robert. "A Distributed Object-Oriented Framework for Dependable Multiparty Interactions". **ACM Sigplan Notices**, v. 34, c. 10, p. 435–446, 1999.

[ZOR99b] ZORZO, Avelino; ROMANOVSKY, Alexander; RANDELL, Brian; XU, Jie; STROUD, Robert; WELCH, Ian."Using Coordinated Atomic Actions to Design Safety-Critical Systems: A Production Cell Case Study. **Software: Practice and Experience**, v. 29, c. 8, p.677-697, 1999.