UNIVERSIDADE SALVADOR – UNIFACS
NUPERC – NÚCLEO INTERDEPARTAMENTAL DE PESQUISAS EM REDES DE COMPUTADORES
MESTRADO EM REDES DE COMPUTADORES

RODRIGO RACHE DE ALMEIDA SOUZA

EXPERT-RETE:
UM MOTOR DE INFERÊNCIA PARA AGENTES COGNITIVOS

Salvador
2005

RODRIGO RACHE DE ALMEIDA SOUZA

EXPERT-RETE:
UM MOTOR DE INFERÊNCIA PARA AGENTES COGNITIVOS

Dissertação apresentada ao Núcleo de Pesquisa em Redes de Computadores – NUPERC, Universidade Salvador - UNIFACS, como parte dos requisitos para obtenção do título de Mestre em Redes de Computadores.

Orientador: Prof. Dr. Augusto Loureiro da Costa

Salvador

2005

Agradecimentos

Aos meus pais Raimundo e Pilar e ao meu avô Léo que sempre me incentivam e me apoiam em tudo.

A minha querida Mariana que também está presente ao meu lado.

Ao professor Augusto pela paciência e boa vontade que teve ao longo da orientação desta dissertação.

A Desenbahia pelo incentivo e apoio durante o curso inteiro de Mestrado.

A todos os meus amigos, colegas de trabalho e demais pessoas que sempre me ajudaram.

RESUMO

Esta dissertação possui como objetivo principal o estudo de técnicas de inferência utilizadas na Inteligência Artificial, demonstrando a importância do motor de inferência em Sistemas Multiagentes. Foi implementado um motor de inferência de uso geral baseado no algoritmo Rete, o Expert-Rete, incluindo como contribuição a otimização dos nós de uma entrada; a otimização dos nós de junção; utilização de 6 estratégias de resolução de conflitos; adoção de 3 formalismos para representação de conhecimentos; outros parâmetros para configuração, como número máximo de interações entre os fatos e as regras, número máximo de ciclos de inferência, dentre outros. O motor de inferência Expert-Rete foi incorporado à biblioteca para construção de sistemas Multiagentes Expert-Coop++. Após isto, o mesmo foi avaliado sob diversos aspectos, confrontando com as características do motor de inferência baseado no encadeamento progressivo, já implementado na biblioteca, comprovando que o Expert-Rete é mais eficiente que o método de inferência anterior para bases de regras mais complexas.

Palavras-chave: Inteligência Artificial; Sistemas Multiagentes, Motor de inferência, Expert-Coop++, Algoritmo Rete, Agentes Cognitivos.

ABSTRACT

This dissertation has focus on inference techniques used in Artificial Intelligence, demonstrating the relevance of the inference engine in Multiagent Systems. A general-purpose inference engine is implemented based on the Rete Algorithm, the Expert-Rete, includig as contributions the optimization of the one-input nodes; the optimization of the two-input nodes; 6 strategies for conflict resolution; adoption of 3 formalisms to knowledge representation; other configuration parameters, like maximum number of interaction between rules and facts, maximum number of inference cycles. The Expert-Rete inference engine was incorporated in the library Expert-Coop++, used to build Multiagent Systems. After this implementation, the Expert-Rete was analyzed among some aspects, comparing with the inference engine based on the forward chaining, already developed in the library, proving that Expert-Rete is more efficient than the former method for complex base rules.

Keywords: Artificial Intelligence; Multiagent Systems; Inference Engine; Expert-Coop++; Rete Algorithm, Cognitive Agents.

LISTA DE ABREVIATURAS E SIGLAS

IA	Inteligência Artifical
IAD	Inteligência Artifical Distribuída
SBC	Sistemas Baseados em Conhecimento
SDP	Solução Distribuída de Problemas
SMA	Sistemas Multiagentes
FIPA	*Foundation for Intelligent Physical Agents*
OMG	*Object Management Group*
SE	Sistemas Especialistas
NSBC	Núcleo do Sistema Baseado em Conhecimento
MCD	Módulo Coletor de Dados
MI	Motor de Inferência
BC	Base de Conhecimento
ME	Módulo de Explicações
MT	Memória de trabalho
LHS	*Left Hand Side*
RHS	*Right Hand Side*
CLIPS	*C Language Integrated Production System*
CSP	*Constraint Saticfaction Problems*
HTN	*Hierarchical Task Network Planning*
GPS	*General Problem Solver*
STL	*Standard Template Library*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

1.1 Contexto

A Inteligência Artificial (IA) como ciência tem mais 40 anos de existência, pois oficialmente a IA nasceu em 1956 tendo por base idéias filosóficas, científicas e tecnológicas herdadas de outras ciências bem antigas (BITTENCOURT, 2001a). Muitos autores consideram o ano de 1956 como um marco importante para a IA pois foi neste período com uma conferência de verão em Dartmouth College, NH, USA que houve a primeira menção oficial à expressão “Inteligência Artificial”.

Desde o princípio a IA gera polêmica pelo próprio nome, definição de seus objetivos e metodologias (BITTENCOURT, 2001a), entretanto o objetivo central da IA tem foco na criação de teorias e modelos cognitivos, e sua conseqüente implementação de sistemas computacionais baseados nestes modelos.

Existem duas linhas principais de pesquisa para a construção de sistemas inteligentes: a linha conexionista e a linha simbólica (BITTENCOURT, 2001a). A linha conexionista visa a modelagem da inteligência humana através da simulação dos componentes do cérebro, ou seja, dos neurônios e suas conexões. A partir daí originou-se a área de redes neuronais. Já a linha simbólica segue a tradição da lógica (BITTENCOURT, 2001a). Nesta linha o comportamento inteligente é adquirido através da manipulação de símbolos.

Outros autores como Bigus, Joseph e Bigus, Jeniffer (2001) destacam três fases distintas na história da IA. No princípio a IA se baseou em problemas formais que eram

estruturados e tinham escopo bem delimitados. Neste início incluiu-se muito da matemática, prova de teoremas, geometria, cálculo e jogos como xadrez. Na primeira fase, houve uma ênfase na criação de “máquinas que pensam” genéricas que seriam capazes de solucionar uma grande classe de problemas. Estes sistemas incluíam técnicas sofisticadas de busca e raciocínio.

Uma segunda fase se iniciou com o reconhecimento de que os projetos de maior sucesso na IA se deram em problemas bem específicos, incluindo o conhecimento específico sobre o problema a ser resolvido. Este processo de incluir domínios de conhecimento específico a sistemas de raciocínio genéricos trouxe o primeiro sucesso comercial da IA: os Sistemas Especialistas. Sistemas Especialistas baseados em regras de produção foram desenvolvidos para desempenhar várias tarefas, incluindo aí análises químicas, como o DENDRAL (FEIGENBAUM, BUCHANAN e LEDERBERG, 1971), conFiguração de sistemas computacionais, como o sistema XCON/R1 (McDERMOTT, 1982), e diagnóstico de condições médicas em pacientes, como o sistema MYCIN (SHOTLIFFE, 1976). Nesta fase pode-se dizer que a IA ganhou visibilidade em aplicações comerciais. Ao mesmo tempo, nesta fase, máquinas foram desenvolvidas para executar especificamente aplicações em Lisp, Prolog, e Smaltalk. Estas máquinas possuíam ambientes integrados de desenvolvimento poderosos que estavam anos a frente de outros ambientes comerciais da época (BIGUS, Joseph e BIGUS, Jeniffer, 2001).

Já a terceira fase é a que atualmente estamos passando, desde o final dos anos 80. Grande parte da comunidade da IA tem trabalhado em solucionar problemas complexos como visão de máquina, comunicação, compreensão de linguagem e tradução, raciocínio comum e controle de robôs. Técnicas de aprendizado ganharam popularidade e expandiram sua utilização em aplicações comerciais através da utilização de redes neurais para tecnologias como mineração de dados, modelagem e controle adaptativo. Métodos biológicos, tais como

algoritmos genéticos e sistemas de lógica alternativa, como lógica nebulosa (lógica *Fuzzy*), descritos em (DOBOIS e PRADE, 1980) e (DOBOIS e PRADE, 1988), têm se combinado com outros métodos da IA Simbólica. Mais recentemente, com o crescimento da Internet e da computação distribuída, há o emprego de agentes que se interagem para a realização de tarefas com objetivos comuns.

Atualmente as áreas de reconhecimento de padrões e principalmente os avanços na IA distribuída (IAD), resolução de problemas, raciocínio, aprendizado e planejamento, ao longo desses anos, juntamente com as aplicações web, demonstram que a IA está se tornando uma área bastante promissora na Ciência da Computação.

E afinal como é possível proporcionar inteligência ou comportamento inteligente a aplicações ? Existem diversos comportamentos que podem ser classificados como inteligente. O comportamento inteligente pode ser produzido através da manipulação de símbolos. Na verdade os símbolos são padrões que representam objetos do mundo real ou idéias que podem ser representadas por um computador. Pesquisadores então construíram sistemas inteligentes com o objetivo de realizar reconhecimento de padrões, raciocínio, aprendizado e planejamento através da manipulação de símbolos. Para técnicas de raciocínio e planejamento a utilização de símbolos se mostrou mais apropriada, em relação ao reconhecimento de padrões e aprendizado propriamente ditos (RUSSEL E NORVIG, 1995).

Existem diversas maneiras típicas para manipular símbolos com relativa eficiência. Uma das abordagens mais comuns está na utilização de símbolos nas formulações de regras do tipo “se-então” que são processadas com a utilização de técnicas de raciocínio. Estas técnicas permitem aos sistemas “deduzir” ou inferir novas informações de um dado conjunto de dados de entrada (BIGUS, Joseph e BIGUS, Jeniffer, 2001).

As técnicas de processamento de símbolos representam um nível alto no processo cognitivo (BIGUS, Joseph e BIGUS, Jeniffer, 2001). Por uma perspectiva da cognição, o

processamento de símbolos corresponde ao pensamento consciente, onde o conhecimento é expressamente representado e o mesmo pode ser examinado e manipulado.

1.2 Objetivos

Este trabalho de dissertação tem como objetivo o estudo na área de raciocínio na IA Simbólica e dar suporte a tecnologia de agentes autônomos. Dentro da área de raciocínio estaremos tratando do processo de inferência em motores específicos para realizar este processo.

É apresentado ao longo deste estudo a conceituação de Sistemas Multiagentes (SMA) e aspectos relacionados, pois a biblioteca Expert-Coop++ (COSTA e outros, 2003), um framework para a construção de SMA, é o ponto de partida para este trabalho.

Após isso, são descritos conceitos com relação a Sistemas Baseados em Conhecimento(SBC) e, mais tarde técnicas de inferência. O agente do Expert-Coop++ implementa um SBC tanto para o nível cognitivo como para o nível instintivo, como veremos nos capítulos que se seguem.

Este trabalho traz a implementação de um motor de inferência de uso geral, o Expert-Rete, que é mais tarde incorporado à biblioteca para construção de Sistemas Multiagentes Expert-Coop++. O Expert-Rete apresenta contribuições não existentes no algoritmo Rete originalmente descrito (FORGY, 1982), como o compartilhamento de nós, além de atributos e características que tornam o Expert-Rete mais adaptável a depender da situação em que se pretenda utilizar o mesmo.

O Expert-Rete é descrito em detalhes, servindo de guia para quem desejar utilizar o motor de inferência, ou ainda para quem desejar utilizar o mesmo com outra finalidade.

Os resultados obtidos com este trabalho são comparados ao motor de inferência já implementado no Expert-Coop++ que é baseado no encadeamento progressivo simples. Tendo como base diversos índices utilizados para a comparação, é comprovado que o Expert-Rete é mais eficiente para base de regras complexas.

Este estudo pode ainda servir de base para outros trabalhos com outras técnicas de inferência além do algoritmo Rete que também são descritas ao longo desta dissertação. Outra possibilidade é estender a utilização do Expert-Rete para outras aplicacões, como é descrito no Capítulo 7.

1.3 Organização da Dissertação

Após a contextualização do problema central desta dissertação, apresentação dos objetivos e do trabalho desenvolvido, o Capítulo 3 trata de Sistemas Baseados em Conhecimento, onde são descritos conceitos gerais sobre os mesmos e arquitetura genérica.

Já o Capítulo 2 tem como assunto principal os Sistemas Multiagentes, incluindo a conceituação de agentes, além dos conceitos envolvidos na biblioteca Expert-Coop++.

No Capítulo 4 há uma descrição detalhada sobre os principais métodos de inferência encontrados na literatura, mostrando como os mesmos funcionam internamente.

O Capítulo 5 apresenta a implementação do Expert-Rete, que é uma outra opção de motor de inferência para a biblioteca Expert-Coop++, utilizando-se de umas das técnicas descritas no capítulo 4, o Algoritmo Rete.

O Capítulo 6 apresenta os resultados obtidos com o Expert-Rete, incluindo avaliações de performance comparadas ao motor de inferência já implementado no Expert-Coop++ (encadeamento progressivo).

E, por fim, o Capítulo 7 mostra a conclusão do trabalho como um todo, resumindo tudo o que foi realizado, as contribuições dadas ao final do trabalho e os possíveis trabalhos futuros que podem ser realizados com o resultado obtido.

2. SISTEMAS BASEADOS EM CONHECIMENTO

Sistemas Baseados em Conhecimento (SBC) estão sendo cada vez mais utilizados tanto no meio comercial como no meio acadêmico. Um dos pontos chaves da construção de SBC está na capacidade destes sistemas de preservar, aproveitar e fazer uso do conhecimento humano aliado à experiência dos mesmos no processo de tomada de decisões.

Atualmente existem diversas aplicações para Sistemas Baseados em Conhecimento, dentre elas as áreas de engenharia, ciências, medicina e demais negócios (REZENDE, 2003). Neste capítulo serão mostrados os conceitos principais, arquitetura e tipos de aplicações de SBC.

2.1 Conceitos de Sistemas Baseados em Conhecimento

SBC utilizam o conhecimento representado explicitamente para resolver problemas onde é necessária uma quantidade considerável de conhecimento humano e de especialização (REZENDE, 2003).

Em SBC, o conhecimento e o processo de resolução de problemas são os aspectos chaves no desenvolvimento dos mesmos.

Baseado em Rezende (2003), o processo de resolução de problemas possui a distinção de dois tipos de operações usadas nesse processo. A primeira é a capacidade de raciocínio, isto é, como se chega a certas conclusões (ou se gera um novo fato) interpretando o conhecimento adquirido até o momento. É cada vez mais reconhecido que o uso restrito dessa

capacidade não é suficiente para a resolução adequada de problemas. Isto é realizado fazendo uso da segunda operação, que consiste em guiar o processo de raciocínio.

É interessante ressaltar que usando somente o raciocínio dedutivo, pode-se eventualmente chegar ao resultado esperado, porém, algumas vezes, de forma ineficiente. De outro lado, para resolver problemas de uma forma rápida e clara é necessário guiar o processo de raciocínio de maneira que apenas conclusões relevantes ao problema em questão sejam consideradas. Essa capacidade é denominada método para resolução de problemas, e o raciocínio é identificado como estratégia de raciocínio ou estratégia de inferência (REZENDE, 2003).

De maneira geral, os SBC devem possuir a funcionalidade de questionar o usuário para reunir as informações de que necessita; desenvolver uma linha de raciocínio a partir de informações coletadas ou do conhecimento nele embutido; explicar o modo de raciocínio e possuir um desempenho satisfatório que compense seus possíveis erros, caso aconteçam (REZENDE, 2003).

Convém ainda diferenciar Sistemas Baseados em Conhecimento dos Sistemas Especialistas. Pode-se dizer que os SBC são capazes de resolver problemas usando conhecimento específico sobre o domínio da aplicação enquanto que os SE são SBC que resolvem problemas ordinariamente resolvidos por um especialista humano. Os SE requerem, portanto conhecimento sobre a habilidade, a experiência e as heurísticas usadas pelos especialistas, sendo a interação com os especialistas no processo de desenvolvimento, de fundamental importância (REZENDE, 2003).

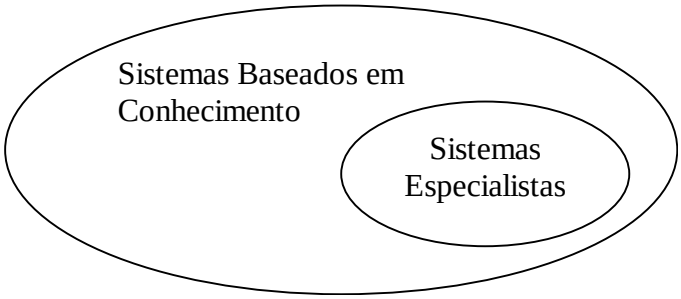

Figura 2.1: Diferença entre SBCs e SEs (REZENDE, 2003)

2.2 Sistemas de Produção

Conforme descrito em (BITTENCOURT, 2001a) *Sistemas de Produção* é um nome genérico para todos os sistemas baseados em *regras de produção*, isto é, pares de expressões consistindo em uma condição e uma ação. A idéia inicial dos sistemas de produção foi introduzida por Post, em 1936, quando ele propôs os hoje chamados sistemas de Post (POST, 1943). Um sistema de Post consiste em um conjunto de regras para a especificação sintática de transformações sobre cadeias de caracteres, e representa, como demonstrou Post, um método geral para o processamento de dados.

Um modelo de sistema de produções, na sua forma mais simples, apresenta dois componentes passivos: o conjunto de regras e a memória de trabalho, e um componente ativo: o interpretador.

Regras de produção são utilizadas predominantemente para se representar um conhecimento heurístico sobre o mundo (FIGUEIRA FILHO 2000), especificando um conjunto de ações que devem ser realizadas para uma dada situação. Uma regra é composta por uma parte de condições (também conhecida por parte “se” da regra, ou lado esquerdo da regra, ou LHS – left-hand side) e uma parte de ações ou conclusões (a parte “então” da regra, ou lado direito da regra, ou RHS – de right-hand side)

Os componentes dos sistemas de produção são então definidos, de acordo com (BITTENCOURT, 2001a):

- Regras: conjunto ordenado de pares (LHS- Left Hand Side, RHS – Right Hand Side) , onde LHS e RHS são sequência de caracteres. LHS é definido portanto como o lado esquerdo (condições) de uma regra e RHS o respectivo lado direito da mesma (conclusões/ações);
- Memória de trabalho: uma sequência de caracteres;
- Interpretador: para cada regra (LHS, RHS), se a sequência de caracteres LHS está contida na memória de trabalho, então o interpretador substitui os caracteres LHS na memória de trabalho pelos caracteres de RHS; se não, passa para a próxima regra.

Os sistemas de produção foram redescobertos durante os anos setenta como uma ferramenta para modelagem da psicologia humana. O formato condição-ação se adapta à modelagem de todos os comportamentos baseados em pares estímulo-resposta.

De fato, os sistemas de produção influenciaram diversos SBCs que se inspiraram na idéia de que o processo de tomada de decisão humano poderia ser modelado por meio de regras do tipo se *condições* então *conclusões/ações* (REZENDE, 2003).

2.3 Arquitetura Geral de um Sistema Baseado em Conhecimento

A arquitetura geral de um sistema baseado em conhecimento é formada pelos módulos de Núcleo do Sistema Baseado em Conhecimento ou Shell, Base de Conhecimento, Memória de Trabalho, Base de Dados e interface com o usuário. Dentre estes módulos se destacam (REZENDE, 2003) os responsáveis pelo armazenamento da Base de Conhecimento (BC) e pelo mecanismo de inferência como veremos a seguir.

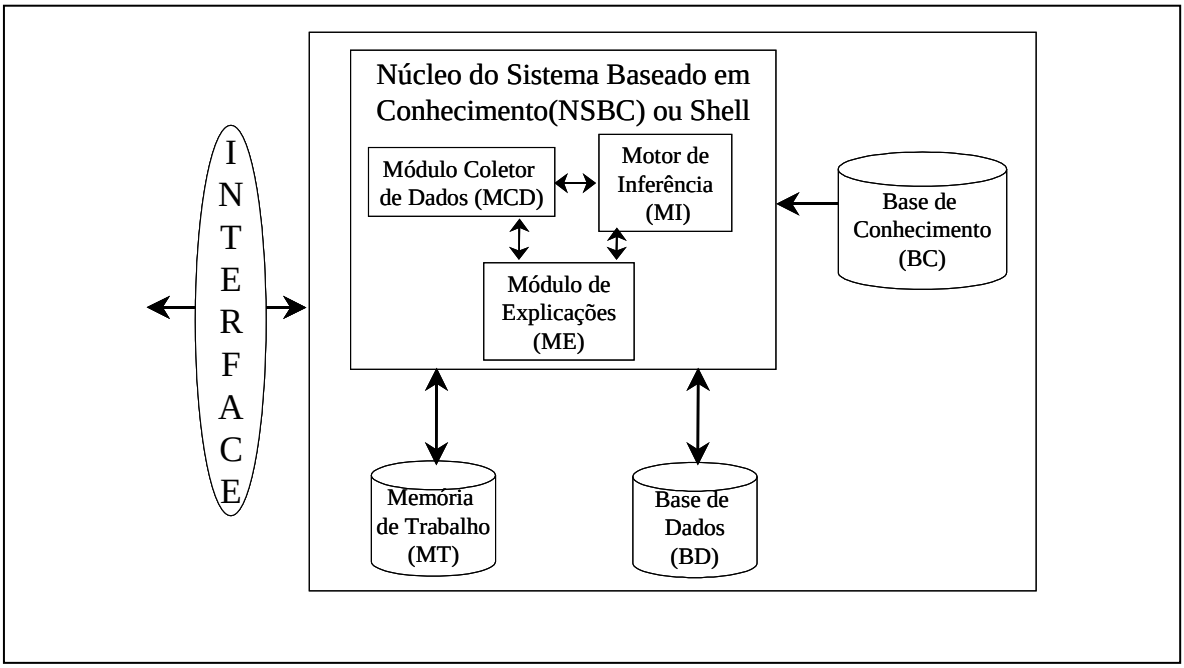

Figura 2.2: Arquitetura Geral de um Sistema Baseado em Conhecimento(REZENDE, 2003).

2.3.1 Núcleo do Sistema Baseado em Conhecimento

O Núcleo do Sistema Baseado em Conhecimento (NSBC) ou Shell, desempenha as principais funções do SBC, sendo responsável (REZENDE, 2003) pelo controle da interação com o usuário ou com equipamentos externos, processamento do conhecimento utilizando alguma linha de raciocínio, justificativa ou explicação das conclusões obtidas a partir do raciocínio.

O NSBC é composto de três submódulos interdependentes, cada qual com funções específicas. O primeiro módulo é o Coletor de Dados (MCD), responsável pela interação com o usuário, obtendo informações do problema em questão, através da formulação de sucessivas perguntas. O MCD pode ainda ser ativado pelo Motor de Inferência para fazer perguntas necessárias, validando as respostas do usuário baseando-se em funções de crítica preestabelecidas (REZENDE, 2003).

O submódulo de Motor de Inferência (MI) é responsável pelo desenvolvimento do raciocínio baseado nas informações obtidas pelo MCD e no conhecimento representado na

BC. O MI precisa conhecer a linguagem de representação utilizada na Base de Conhecimento para daí gerar e percorrer o espaço de busca sempre que necessário. Existem diversas linhas de raciocínio ou mecanismos de inferência que podem ser seguidas pelos SBC que vão determinar o modo como o raciocínio se desenvolve no sistema.

Já o submódulo de Explicações (ME) é responsável pela explicação, ou justificativas, das conclusões obtidas e dos motivos pelos quais o SBC fez determinadas perguntas. Os principais tipos de explicações são:

- *por que*, que neste caso é ativado se o usuário questionar o porquê da pergunta, onde o MCD ativa o ME justificando a sua necessidade;
- *como* que neste caso o ME fornece uma seqüência de raciocínio que levou o SBC a chegar àquela conclusão, se o usuário deseja saber como as conclusões foram obtidas;
- *o que acontece se* ativado se o usuário desejar saber qual seria o resultado obtido se uma ou mais informações fossem modificadas. O ME, neste caso pergunta quais as condições a serem modificadas, apaga da MT antigas informações referentes a tais condições e ativa o MI, que reinicia o processo de inferência;
- *por que não* onde o ME analisa o processo de inferência e descobre qual a razão daquela conclusão não ter sido alcançada.

O NSBC deve interagir também com a memória de trabalho (MT), onde deve gravar e acessar os dados temporários obtidos no processo. Além disso, deve manipular a Base de Conhecimento, a qual contém representado todo o conhecimento.

3.3.2 Base de Conhecimento

A Base de Conhecimento (BC) contém a descrição do conhecimento necessário para a resolução do problema abordado na aplicação (REZENDE, 2003). Isto inclui informações sobre o domínio de conhecimento, regras que descrevem relações nesse domínio e, em alguns casos heurísticas e métodos de resolução de problemas.

Uma BC possui uma série de estruturas que representam ações e acontecimentos no ambiente ao qual o sistema está inserido. Neste ponto as linguagens de representação de conhecimento são utilizadas, baseando-se em diferentes técnicas de representação como: regras de produção, redes semânticas, *frames* e lógica.

As BCs são compostas por diversas sentenças (em uma BC, uma sentença é uma representação individual de conhecimento) e estas sentenças podem apresentar diferentes graus de generalidade e complexidade. A maioria das sentenças descreve relações de causa e efeito no domínio e outras descrevem o conhecimento sobre como guiar a busca por uma solução (metaconhecimento) (REZENDE, 2003).

BCs consistentes, pequenas e precisas são ideais e desejáveis para que Sistemas Baseados em Conhecimento cheguem a soluções de maneira mais eficiente e eficaz, entretanto, nem sempre isso acontece (REZENDE, 2003). Por este motivo, é interessante que existam meios de analisar evidências de cada conclusão para se escolher qual deve ser a resposta do SBC. Estas evidências devem ser analisadas por procedimentos de resolução de conflitos e filtros capazes de sensibilizar conclusões que realmente interessam ao SBC.

O conhecimento em SBC pode ainda não ser completo, ou seja, nem todo conhecimento requerido para gerar uma resposta se encontra na base (REZENDE, 2003). O SBC deve neste caso encontrar a melhor resposta diante desta falta de conhecimento.

2.3.3 Memória de Trabalho (MT)

A MT representa a área de trabalho de um SBC, na qual são registradas todas as respostas fornecidas pelo usuário durantes as interações realizadas com o sistema. A MT traz a vantagem de fornecer ao usuário toda a linha de raciocínio correspondente às conclusões obtidas, evitando a repetição de uma mesma pergunta ao usuário, e ainda evita a realização repetida de seqüências de raciocínio correspondente para obtenção de conclusões intermediárias.

A Memória de trabalho armazena condições iniciais, conclusões e decisões intermediárias, além de soluções finais (REZENDE, 2003).

2.3.4 Interface

A interface é responsável pela interação entre o SBC e o usuário, proporcionando a comunicação em ambas direções e realizando a intermediação entre a representação interna do sistema e a representação mental do sistema (REZENDE, 2003). Geralmente a linguagem utilizada na interface de um SBC é mais próxima da linguagem do usuário, sendo entretanto, mais restrita que a linguagem do dia-a-dia e mais abstrata que a linguagem utilizada na nas bases de conhecimento.

As interfaces de SBC precisam ser altamente dinâmicas (REZENDE, 2003), já que as informações a serem requisitadas e as conclusões a serem alcançadas só são realmente definidas em tempo de resolução de um problema em particular.

2.4 Representação de Conhecimento

Parte muito importante para Sistemas Baseados em Conhecimento é a forma ou escolha das representações de conhecimento a serem utilizadas. A linguagem associada ao método escolhido deve ser suficientemente expressiva (mas não mais do que suficiente) para permitir a representação do conhecimento a respeito do domínio escolhido de maneira completa e eficiente (BITTENCOURT, 2001a). Cada formalismo apresenta pontos positivos ou negativos em termos de eficiência, facilidade de uso e a necessidade de expressar conhecimento incerto e incompleto.

Existem muitas formas de representação do conhecimento através de formalismos computacionais como lógica, frames, redes semânticas, regras de produção, dentre outros. Nas próximas seções estes formalismos são descritos brevemente.

2.4.1 Lógica

Segundo Bittencourt (2001) a lógica é a base para a maioria dos formalismos de representação de conhecimento, seja de forma explícita ou na forma de representações específicas que podem facilmente ser interpretadas como proposições ou predicados lógicos como por exemplo na forma:

(<atributo>, <objeto>, <valor>, <coeficiente de certeza>) ou ainda com os três primeiros argumentos.

A lógica matemática é uma linguagem formal. Diferentemente de linguagens naturais, nas quais as regras gramaticais são imprecisas, nas linguagens formais sempre se pode dizer se uma seqüência de símbolos está de acordo com as regras para a construção de expressões (fórmulas) da linguagem (REZENDE, 2003).

Exemplo de representação de conhecimento em Lógica:

(logic (progenitor Rita Alan))

No exemplo anterior fica descrito que o progenitor(atributo) de Rita(objeto) é Alan(valor). A palavra "*logic*" encontrada no exemplo significa que o formalismo está representado utilizando a lógica.

Já a lógica de descrições, como descrito em (FREITAS, 2003) e (FREITAS, 2002), fornece nativamente atributos expressivos, com semântica e inferência associadas, porém cuidadosamente escolhidos de forma a garantir um algoritmo de inferência decidível e eficiente. Ainda segundo (FREITAS, 2003) este formalismo fornece atributos como:

- Herança Múltipla: exemplo, *(PRIMITIVE (AND CAR EXPENSIVE-THING) sports-car).* Neste caso a instância do conceito carro-esporte herda as descrições de *CAR* e *EXPENSIVE-THING*.
- Papéis: exemplo, *(FILLS thing-readable PCMagazine),* significando que PCMagazine tem o papel de algo que serve para ser lido.
- Restrição de Valores: exemplo *(ALL thing-readable VISIBLE),* demonstrando que tudo que serve para ser lido está no domínio visível.
- Restrição de Limites: exemplo, *(AT-LEAST 3 wheel),* significando que qualquer objeto relacionado a 3 outros objetos distintos que possuem o papel de roda (*wheel*).
- Co-referência: exemplo, *(SAME-AS (driver) (insurance-payer)),* descrevendo que todos os indivíduos que atuam como motoristas têm o papel de segurandos.

2.4.2 Redes Semânticas

Uma rede semântica é um grafo rotulado e direcionado formado por um conjunto de nós representando os objetos (indivíduos, coisas, conceitos, situações em um domínio) e por um conjunto de arcos representando as relações entre os objetos (REZENDE, 2003).

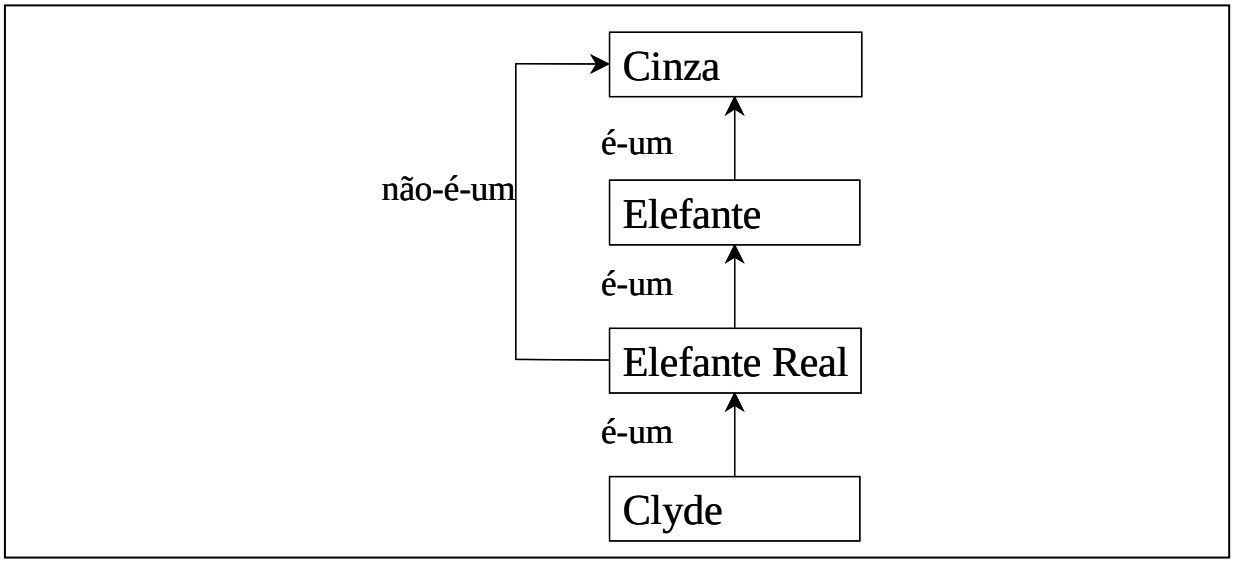

Figura 2.3: Exemplo de Rede Semântica

Objetos complexos muitas vezes podem ser decompostos em objetos mais simples. Essas decomposições produzem relações diversas que podem ser, por exemplo, “é-um”, “é-parte” (primeiras árvores semânticas) representando a herança de propriedades entre os nós e demais arcos representando relações específicas do domínio ou propriedades de conceitos.

A herança de propriedades através de caminhos formados por arcos é uma das características mais importantes do método de representação por redes semânticas (BITTENCOURT, 2001a). Esta característica faz com que se economize memória no processamento e são bastante eficientes quando da forma de árvores, entretanto caso sejam permitidas heranças múltiplas entre os nós a rede semântica pode se tornar bastante complexa. Esta complexidade ocorre por conta da dependência da escolha dos significados dos arcos da rede.

Outro mecanismo de inferência utilizado em redes semânticas é a correspondência de um fragmento de rede em relação a uma rede dada (BITTENCOURT, 2001a). A

especificação da semântica deste mecanismo é ainda mais complexa que a do mecanismo de herança por depender da escolha do significado dos arcos da rede. Ver exemplo a seguir:

Neste exemplo Clyde deve ser considerado Cinza por ser um elefante e não Cinza por ser um Elefante Real. A estratégia de herança, geralmente utilizada, determina que propriedades ligadas a conceitos mais específicos devem ter prioridade, ou seja, no caso Clyde deve ser considerado não Cinza, pois Elefante Real é mais específico que Elefante, só que exemplos mais complexos podem não ser tão intuitivos.

2.4.3 Quadros (*Frames*)

Frames são agrupamentos de conhecimentos relevantes a uma coisa, um indivíduo, uma situação ou um conceito. Os frames são estruturas de dados que possuem um nome que identifica o conceito que os mesmos definem e possuem ainda um conjunto de atributos (denominados *slots*) (MINSKY, 1974). De maneira semelhante ao conceito de objetos os frames se constituem de uma maneira útil para modelar objetos do mundo real.

Frame: Cômodo	Super-Frame: Lugar-coberto		
Atributos	Default	Tipo	Se-necessário
Número de paredes	4	número	
Formato	Retangular	símbolo	
Altura	3	número(m)	
Área		número(m^2)	
Volume		número(m^3)	(Área * Altura)

é-um

Frame:Sala	Super-Frame:Cômodo	
Atributos	Default	Tipo
Mobiliário	(sofá, mesa, cadeiras)	lista de símbolos
Finalidade	convivência	símbolo

Figura 3.4: Exemplos de Quadros

Através dos valores dos atributos, os quadros são descritos através das características do objeto representado pelo quadro. Os valores atribuídos a estes atributos podem ser outros

quadros, criando uma rede de dependências entre os mesmos. Os quadros são também organizados em uma hierarquia de especialização, criando uma outra dimensão de dependência entre eles (BITTENCOURT, 2001a). Os atributos também apresentam propriedades, que dizem respeito ao tipo de valores e às restrições de número que podem ser associados a cada atributo.

Os atributos dos quadros também apresentam propriedades, que dizem respeito ao tipo de valores e às restrições de número que podem ser associados a cada atributo. Estas propriedades são denominadas de facetas.

Pela descrição mostrada na Figura 3.4, conclui-se que uma sala é um tipo de cômodo, normalmente com quatro paredes e de formato retangular, com um mobiliário específico. As facetas como demonstradas na Figura, especificam os tipos de valores esperados e, se for o caso, procedimentos adequados para calcular o valor do atributo.

2.5 Motor de Inferência

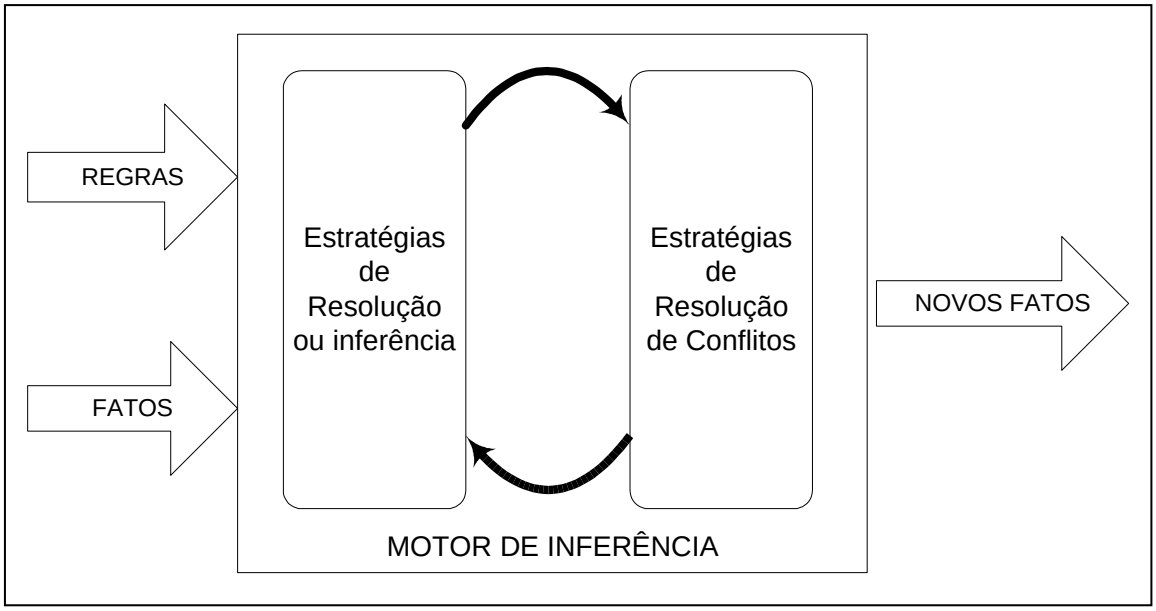

Figura 2.5: Arquitetura de motor de inferência

O módulo motor de inferência é responsável pelo desenvolvimento do raciocínio baseado nas informações obtidas pelo MCD e no conhecimento representado na BC

(REZENDE, 2003). O motor de inferência representa a forma de manipular o conhecimento já representado na BC, a fim de resolver o problema. Ele determina a ordem com que serão processadas as informações, manipulando os dados a fim de inferir novos fatos, chegar a conclusões ou recomendar ações (FIGUEIRA FILHO 2000).

Uma possível regra em um sistema que utiliza regras para representar conhecimento , no caso do futebol de robôs, por exemplo, poderia ser “Se está livre de marcação, com a bola no pé e próximo ao gol, então chutar a gol”. Deste modo, com o fato “está livre de marcação” atrelado também ao fato de que “está com a bola no pé” e está “próximo ao gol”, pode-se inferir que a ação “chutar a gol” deve ser disparada. Estes fatos gerados também podem ser armazenados em uma base de fatos local ou mesmo um banco de dados e pode dar origem a novos fatos até se chegar a um resultado final.

O motor de inferência é aí utilizado para manipular os fatos existentes com as regras internas a fim de produzir novos fatos. Há ainda a possibilidade de se denominar a primeira parte da regra (até “ENTÃO”) como LHS (*Left Hand Side*) ou lado esquerdo da regra, e a segunda parte (após o “ENTÃO”) como RHS (*Right Hand Side*) ou lado direito da regra.

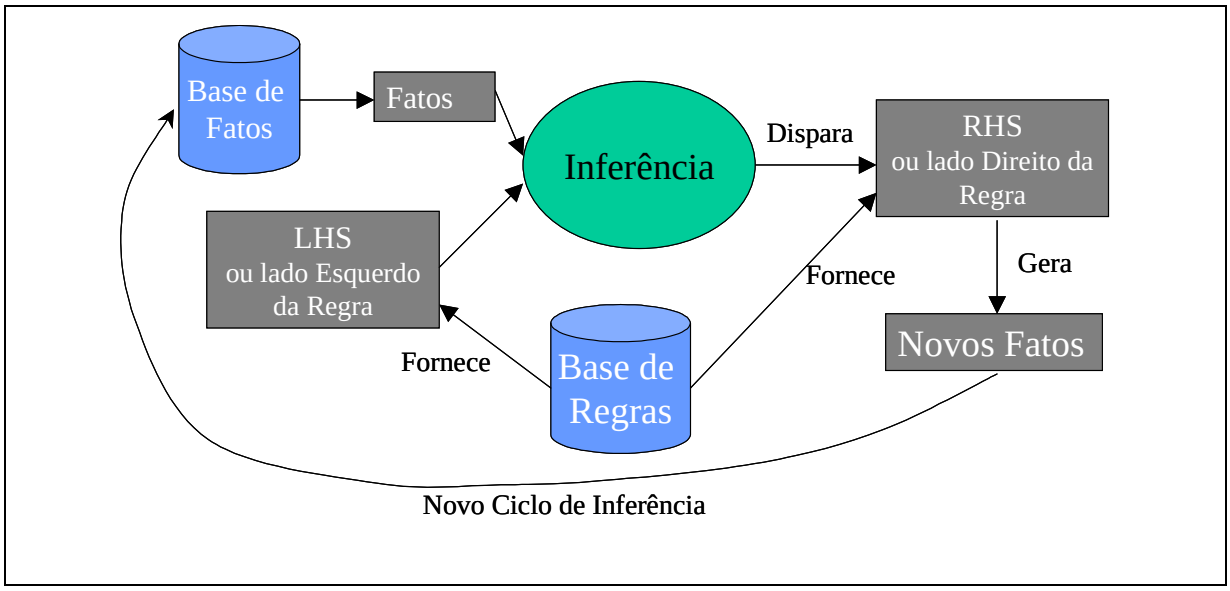

Figura 2.6: Processo de Inferência utilizado nos motores de Inferência convencionais

Como descrito em (FIGUEIRA FILHO 2000), o ciclo de execução de um sistema de produção, com encadeamento progressivo (técnica de inferência que veremos no capítulo a seguir) têm três passos principais:

- Unificação, onde o sistema procura unificar os fatos da memória de trabalho com as declarações das regras de modo a tornar verdade os predicados que compõem as condições das mesmas. A cada unificação bem sucedida, o par <regra, fatos> é inserido no conjunto de conflitos;
- Resolução de Conflitos, onde é escolhido um par <regra,fatos> do conjunto de conflitos, de acordo com a política de resolução de conflitos utilizada;
- Disparo da Regra, no qual as ações da regra escolhida na etapa anterior são executadas com suas variáveis substituídas pelos fatos que tornaram a regra disparável.

Este ciclo se reprete até que não haja mais nenhuma regra disparável. A ação da regra pode alterar a lista de regras disparáveis, através do acréscimo ou da remoção de fatos da memória de trabalho.

Um ciclo de inferência pode ser definido como a passagem de todos os fatos inicialmente carregados no motor de inferência que podem ou não sensibilizar o lado esquerdo das regras (esta descrito um ciclo de inferência). Combinando o lado esquerdo com todos os fatos, o lado direito das regras será disparado dando origem a novos fatos que serão adicionados na base de fatos dando origem a outros ciclos de inferência. Dado o exemplo:

Base de Regras:

- *(Regra 1)Dor-de-Cabeça(x) ^ Dor-no-pescoço(x) ⇒ Cefaléia(x)*
- *(Regra 2)Cefaléia(x) ^ Febre(x) ⇒ Possivel-dengue(x)*

E Base de Fatos contendo inicialmente:

- *Dor-de-Cabeça(João)*
- *Dor-no-pescoço(João)*
- *Febre(João)*

Em um primeiro ciclo de inferência, o lado esquerdo da Regra 1 será sensibilizado, gerando o fato *Cefaléia(João)* e, num segundo ciclo de inferência a Regra 2 será disparada. Com o disparo da Regra 2 o novo fato gerado (*Possível-dengue(João)*) vai dar origem a um novo ciclo de inferência em que nenhuma regra será disparada, finalizando a inferência.

2.6 CLIPS

O CLIPS (*C Language Integrated Production System*) é uma ferramenta utilizada para o desenvolvimento de Sistemas Especialistas que proporcionam um ambiente completo para construção de SE baseados em regras e/ou objetos (RILEY, 2005). O CLIPS foi criado pela divisão de IA da NASA em meados de 1985 e hoje é mantido de maneira independente.

Atualmente o CLIPS possui uma variedade de representação de conhecimento com suporte para três paradigmas de programação diferentes: baseado em regras (de produção), usadas principalmente para a representação de conhecimento heurístico baseado em experiência; funções específicas e funções genéricas, para representação de conhecimento procedimental; ou programação orientada a objetos, também utilizada na representação de conhecimento procedimental (com uma certa carga declarativa, dada a possibilidade de definição de relações entre classes) (FIGUEIRA FILHO 2000).

CLIPS pode ser usada em conjunto com liguagens de programação convencionais como C, Ada ou C++, entretanto o mesmo define uma linguagem própria (COOL, ou CLIPS Object-Oriented Language). Com a COOL uma aplicação pode ser inteiramente definida no ambiente CLIPS, dissociando do mundo de objetos da linguagem hospedeira.

Outra característica interessante do CLIPS é a integração e extensibilidade já que o mesmo pode ser incorporado em código, chamado como uma sub-rotina, e integrado com

linguagens de programação. Desta maneira o CLIPS pode ser utilizado por um usuário para utilização em diversas aplicações.

O motor de inferência do CLIPS funciona segundo o Algoritmo Rete (FORGY, 1982), método de inferência este que será descrito em maior detalhe no capítulo 4.

O controle de disparo das regras do CLIPS é baseado em uma das 7 estratégias de resolução de conflitos suportadas pelo sistema (FIGUEIRA FILHO 2000). Neste contexto, para cada regra, pode-se especificar prioridade, e dentre as regras de mesma prioridade o sistema utiliza a estratégia escolhida para selecionar a próxima regra a ser disparada. As estratégias existentes incluem:

- Profundidade, na qual as regras recém ativadas tem maior prioridade que as mais antigas;
- Largura em que as regras recém ativadas tem menor prioridade;
- Simplicidade, em que as regras que precisam de menos comparações adquirem maior prioridade;
- Complexidade, que é o contrário da estratégia anterior;
- Aleatória, como o próprio nome já descreve;
- MEA, onde as regras com objetos mais recentes adquirem maior prioridade;
- LEX, que é o contrário da anterior;

A integração de CLIPS com as linguagens hospedeiras, apesar de existir, não é completa (FIGUEIRA FILHO 2000). A sintaxe das regras no CLIPS está mais próxima da linguagem LISP que a de C.

Outro problema do CLIPS está na filosofia de que tudo pode ser feito em COOL, só que todos os serviços tem que ser reconstruídos nesta linguagem, dificultando a reutilização dos objetos da linguagem hospedeira.

2.7 JESS

O JESS também pode ser classificado como uma ferramenta para construir SE (FRIEDMAN-HILL, 2001). Esta ferramenta é inteiramente escrita em Java e disponibiliza um conjunto de classes que podem ser incorporadas em diversas aplicações.

O JESS é baseado no CLIPS e originalmente era apenas um clone em Java do CLIPS (FRIEDMAN-HILL, 2001), entretanto atualmente já possui diversas características que diferenciam o mesmo do CLIPS. O JESS possui otimizações sobre o algoritmo Rete original como o compartilhamento de nós no grafo como forma de economizar memória e diminuir o espaço de busca para as regras. O JESS pode ainda operar com motor de inferência baseado no encadeamento regressivo.

Assim como o CLIPS, JESS define toda uma linguagem para ser usada na criação de aplicações, e não apenas de regras (FIGUEIRA FILHO 2000). Toda uma aplicação pode ser desenvolvida em JESS, já que comandos de criação de interfaces gráficas, comunicação ou entrada/saída são disponibilizados no ambiente do JESS.

O JESS utiliza a estratégia de interpretação de regras, fazendo com que ele seja um ambiente de *scripting* para Java. Através deste ambiente, é possível criar objetos e invocar métodos da linguagem Java sem a necessidade de compilação.

Assim como o CLIPS o JESS possui baixa integração do sistema com a linguagem Java, existindo uma separação entre o mundo dos objetos Java dos objetos ou fatos JESS. A sintaxe para definição das regras no JESS é totalmente diferente da linguagem Java, o que dificulta o entendimento do sistema.

2.8 Resumo do Capítulo

Neste capítulo foram apresentados os principais conceitos a respeito de Sistemas Baseados em Conhecimento. Foi descrita ainda a arquitetura geral de um SBC, descrevendo o núcleo do SBC, a Base de Conhecimento e formalismos utilizados para representação de conhecimento mais encontrados, a memória de trabalho, a interface e o motor de inferência.

Ao final são descritas duas ferramentas de apoio a construção de SBC bastante conhecidas o CLIPS e o JESS. O CLIPS é classificado como uma *Shell* e o JESS como uma ferramenta de apoio que oferece diversas características para a criação de SBC.

Este capítulo é de extrema relevância a este estudo já que o nível cognitivo de um agente do Expert-Coop++ implementa um SBC, que por conseqüência possui um motor de inferência baseado no encadeamento progressivo, e ao final deste trabalho, o Expert-Rete como opção de motor de inferência.

3. IA E SISTEMAS MULTIAGENTES

Neste capítulo será apresentada a tecnologia de agentes, sistemas multiagentes com o foco na biblioteca para construção de Sistemas Multiagentes Expert-Coop++ que mais tarde será modificada ao longo deste estudo.

3.1 Inteligência Artificial Distribuída

Desde o final da década de 70 com o nascimento do domínio da pesquisa da Inteligência Artificial Distribuída (IAD) (WEISS, 1999) a utilização de Sistemas Multiagentes se evoluiu e se diversificou de maneira muito intensa. Atualmente este é um ramo da pesquisa estabelecido em IA e que promete inovações em economia, sociologia, administração e filosofia.

Existem diversos motivos para distribuir Sistemas Inteligentes sendo que o principal deles é que em alguns domínios de aplicação, como controle de tráfego aéreo, distribuição de energia elétrica, por exemplo, são problemas inerentemente distribuídos no espaço (BITTENCOURT, 2001a). E ainda em alguns casos a solução por meio de sistemas centralizados é impossível de ser realizada (WEISS, 1999), pois estes sistemas e seus dados pertencem a organizações independentes que desejam manter suas informações de maneira privada e segura para fins de competição.

Demais razões para se distribuir sistemas inteligentes recaem na melhoria da adaptabilidade, confiabilidade e autonomia dos sistemas; redução do custo de

desenvolvimento e manutenção; aumento da eficiência e velocidade de processamento; integração de sistemas inteligentes para aumentar capacidade de processamento e a possibilidade de integração de computadores nas redes de atividades humanas (BITTENCOURT, 2001a).

3.1.1 Agentes

Apesar de não haver uma definição universalmente aceita do conceito de agente, é possível definir o mesmo como uma entidade computacional que é situado em algum ambiente, e que é capaz de uma ação autônoma sobre este ambiente para alcançar os objetivos para os quais ele foi construído (WEISS, 1999).

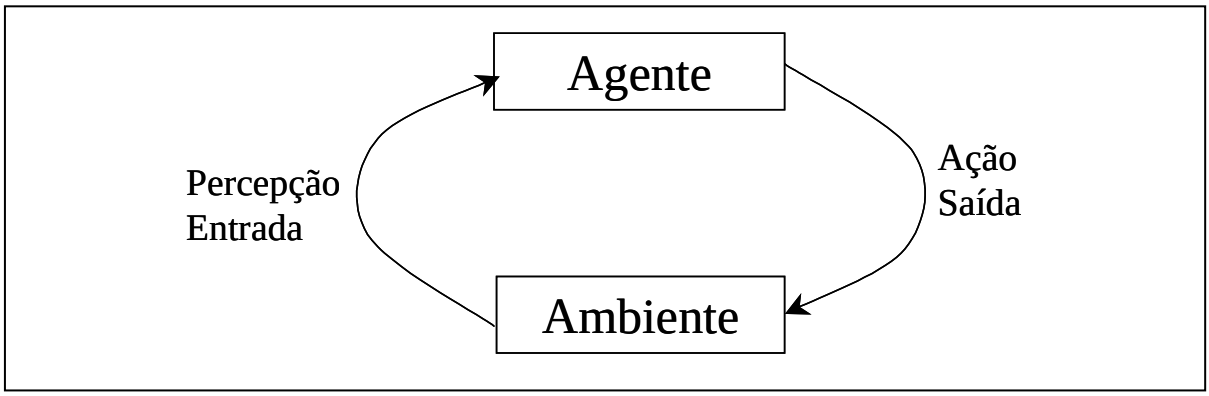

Figura 3.1: um agente em seu ambiente (WEISS, 1999)

Já os agentes inteligentes também podem ser definidos como agentes que são capazes de ação flexível e autônoma para alcançar os objetivos para os quais os mesmos foram contruídos. Neste caso a flexibilidade significa (WEISS, 1999):

- Reatividade: agentes inteligentes são capazes de perceber seu ambiente, e reponder, em um tempo apropriado, a mudanças que ocorrem para satirfazer seus objetivos;
- Pró-atividade: agentes inteligentes são capazes de exibir comportamento direcionado a objetivos de maneira a tomar a iniciativa para satisfazer seus objetivos;

- Habilidade Social: agentes inteligentes são capazes de interagir com outros agentes para satisfazer seus objetivos.

Um agente, portanto, é uma entidade real ou virtual, capaz de agir num ambiente, de se comunicar com outros agentes, que é movida por um conjunto de inclinações (sejam objetivos individuais a atingir ou uma função de satisfação a otimizar); que possui recursos próprios; que é capaz de perceber seu ambiente (de modo limitado); que dispõe (eventualmente) de uma representação parcial deste ambiente; que possui competência e oferece serviços; que pode eventualmente se reproduzir e cujo comportamento tende a atingir seus objetivos utilizando as competências e os recursos que dispõe e levando em conta os resultados de suas funções de percepção e comunicação, bem como suas representações internas (REZENDE, 2003).

Com a definição acima, percebe-se que os agentes compreendem a utilização de diversas técnicas da IA, entretanto algumas características ajudam ainda mais a caracterizar o agente em questão, como as que se destacam abaixo (REZENDE, 2003):

- Autonomia de decisão - Capacidade de analisar uma situação, gerar alternativas de atuação e escolher a melhor alternativa para alcançar seu objetivo;
- Autonomia de Execução – Capacidade de operar no ambiente sem intervenção de outro agente;
- Competência para Decidir – Capacidade de conFigurar sua atuação sem intervenção externa;
- Existência de uma Agenda Própria – Capacidade de criar uma agenda (lista) de objetivos que concretizem suas metas;
- Reatividade – Capacidade de reagir a mudanças do ambiente a partir do reconhecimento de um contexto conhecido;

- Adaptabilidade – Capacidade de adaptar seu processo de decisão frente a situações desconhecidas;
- Mobilidade – Capacidade de mover-se e ser executado em outras plataformas;
- Personalidade – Capacidade do agente de personificar-se, utilizando recursos que lembrem características humanas;
- Interatividade com o usuário – Capacidade de interagir com usuários e, considerando os possíveis mal-entendidos, reagir às falhas de comunicação de maneira aceitável;
- Ambiente de Atuação – Caracteriza o local onde o agente vai atuar, isto é, em ambientes fechados (desktop) ou abertos (internet);
- Comunicabilidade – Capacidade de interagir com outros agentes computacionais para a obtenção de suas metas.

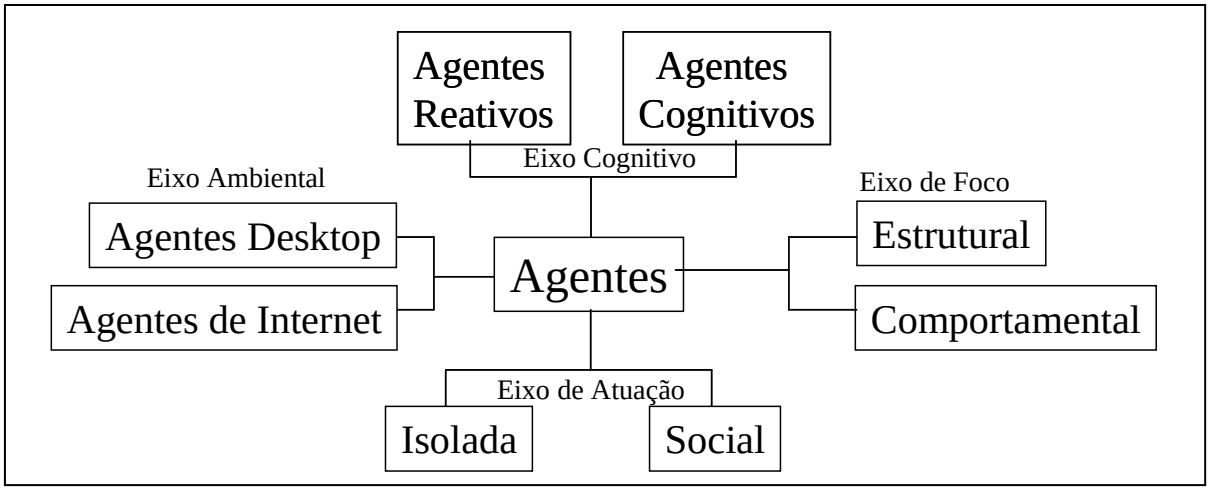

Figura 3.2: Taxonomia de Agentes (REZENDE, 2003)

De acordo com (REZENDE, 2003) os agentes podem ser classificados quanto:

- Cognição – Um agente pode conter um modelo de representação interna do ambiente e dos outros agentes baseado em estados mentais com um modelo racional de decisão (Agente Cognitivo) ou agir baseado em reações aos estímulos provocados pelo ambiente (Agente Reativo);

- Foco – Um agente pode ter similaridades físicas com humanos (Estrutural) ou similaridades comportamentais(Comportamental);
- Atuação – Um agente pode atuar de forma isolada (Isolada) ou interagindo com outros agentes (Social);
- Ambiental – Um agente individual pode atuar localmente no microcomputador (Agente de Desktop) ou em uma rede Internet ou Intranet (Agente de Internet).

Dentro dessa classificação se destacam no eixo cognitivo os agentes reativos e os agentes cognitivos.

Os agentes reativos são aqueles que não possuem representação de seu ambiente, pois as informações relativas ao seu comportamento se encontram no próprio ambiente e suas reações dependem unicamente da percepção do mesmo. Estes agentes não possuem memória de suas ações, sua organização é etológica (similar a dos animais) em oposição à organização social dos agentes cognitivos e há um grande número de membros (da ordem de milhares) (ROCHA, ALVES e JÚNIOR, 2003) como descrito na arquitetura de subsunção (BROOKS, 1986).

Já os agentes cognitivos são baseados em organizações sociais humanas como grupos, hierarquias e mercados. Os agentes aqui possuem uma representação explícita do ambiente e dos outros agentes, dispõem de memória e são capazes de planejar suas ações futuras. Estes agentes podem ainda comunicar-se entre si diretamente e o número de membros em uma sociedade de agentes cognitivos é, no máximo, da ordem de algumas dezenas (BITTENCOURT, 2001).

A IAD estuda o conhecimento e as técnicas de raciocínio que podem ser necessárias ou úteis para que agentes computacionais participem de sociedade de agentes, dentre outras atividades. Um agente pode então novamente ser definido como uma entidade computacional tal como um software ou um robô que pode perceber e atuar sobre seu ambiente e sua

autonomia sobre seu comportamento depende, ao menos parcialmente, da sua própria experiência.

Por motivos históricos, a IAD se dividiu em dois enfoques: a Solução Distribuída de Problemas (SDP) e Sistemas Multiagentes (SMA) (BITTENCOURT, 2001a). Esta subdivisão, entretanto, não é clara, ocorrendo 3 visões alternativas na tentativa de determinar uma relação entre SDP e SMA (BITTENCOURT, 2001a). A primeira é a de que SDP é um subconjunto de SMA, a segunda visão é a de que SMA fornecem a base para SDP e a terceira e última visão é a de que SMA e SDP são linhas de pesquisa complementares.

3.2 Solução Distribuída de Problemas

A SDP tem como foco o problema, conforme a tradição da IA Simbólica, da qual este enfoque é diretamente derivado (BITTENCOURT, 2001a). Neste ramo da IAD a capacidade de processamento e robustez das redes de computadores são utilizadas para atacar os problemas distribuídos. Dentre os objetivos mais genéricos da SDP estão o aumento da eficiência através do paralelismo, aumento do número de tarefas através do compartilhamento de recursos, aumento da confiabilidade de diminuição da interferência entre tarefas.

A SDP é aplicável quando se enxerga claramente uma divisão de tarefas entre os agentes, que interagem entre si (FREITAS, 2002). os agentes são designados para resolverem um problema em particular, dentro de uma concepção fechada de mundo. Isto significa que os agentes são projetados para resolver um tipo específico de problema apenas e não podem ser utilizados em qualquer outro problema, mesmo que seja similar. Desta maneira, o número de agentes será fixo, sendo que cada agente possui uma visão específica e incompleta do problema. Então, para a resolução de um problema, os agentes devem cooperar entre si,

compartilhando conhecimento sobre o problema e sobre o processo para obter uma solução (JAQUES, 1999).

De acordo Sichman (1995) a estratégia de resolução segundo a SDP apresenta as seguintes características:

- O problema é resolvido por um conjunto de agentes, fisicamente distribuídos em diversas máquinas conectadas via rede. Tais agentes são concebidos para solucionar um determinado problema particular;
- Um organização é concebida para restringir o comportamento destes agentes. Tal organização, geralmente, é definida durante a fase de concepção do sistema;
- A interação entre os agentes é realizada por troca de mensagens, ou por meio do compartilhamento de dados comuns. A estrutura destas trocas é geralmente definida durante a fase de concepção do sistema, sendo intimamente relacionada ao modelo algorítmico subjacente (como por exemplo, o quadro-negro, que veremos a seguir) e ao problema que o sistema deve resolver;
- Os agentes são executados de modo concorrente, para aumentar a velocidade de resolução;
- Os agentes cooperam, dividindo entre si as diversas partes do problema original (sub-problemas / tarefas), ou podem até mesmo aplicar diferentes estratégias de resolução para uma mesma tarefa;
- Existe a noção de um controle global, na maior parte dos casos, implícito nos agentes, que garante um comportamento global coerente do sistema, conforme a organização inicialmente prevista. Tal controle pode ser implementado quer de modo centralizado (pela criação de um agente responsável pela gerência do sistema), quer de modo distribuído (pela criação de agentes auto-controlados).

O projeto de um sistema SDP é realizado por um projetista que, primeiramente, realizará uma análise do problema a ser resolvido para, então, identificar os agentes necessários para a solução desse problema. Desta maneira, a tarefa de resolução será decomposta entre os vários agentes, buscando melhorar o processamento do sistema através da execução paralela.

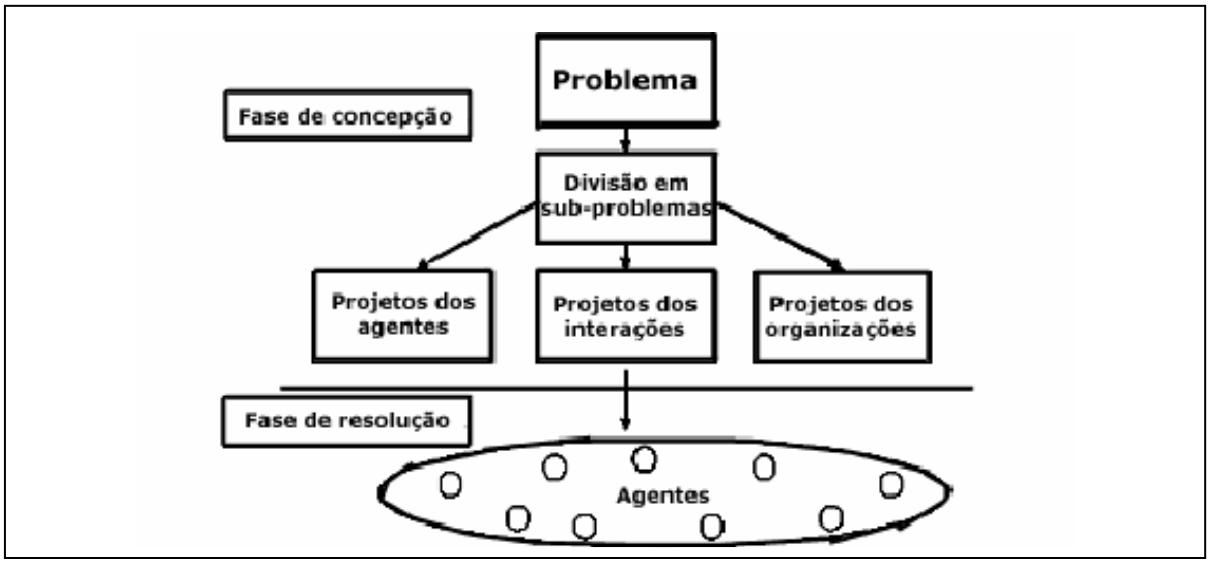

Figura 3.3: Resolução Distribuída de Problemas (SICHMAN e ALVARES, 1997)

3.3 Sistemas Multiagentes

Já os agentes em Sistemas Multiagentes (SMA) são projetados para resolverem uma classe de problemas, e não um problema específico, como acontece na SDP (ROCHA, JÚNIOR e ALVES, 2003). Isto ocorre porque nos SMA os agentes são entidades autônomas que têm conhecimento da sua própria existência e da existência de outros agentes e, portanto, colaboram uns com os outros para atingirem um objetivo comum dentro de um ambiente.

Os agentes, em SMA, devem ainda possuir conhecimentos e habilidades para executar uma determinada tarefa e devem, a princípio, cooperar para atingir um objetivo global.

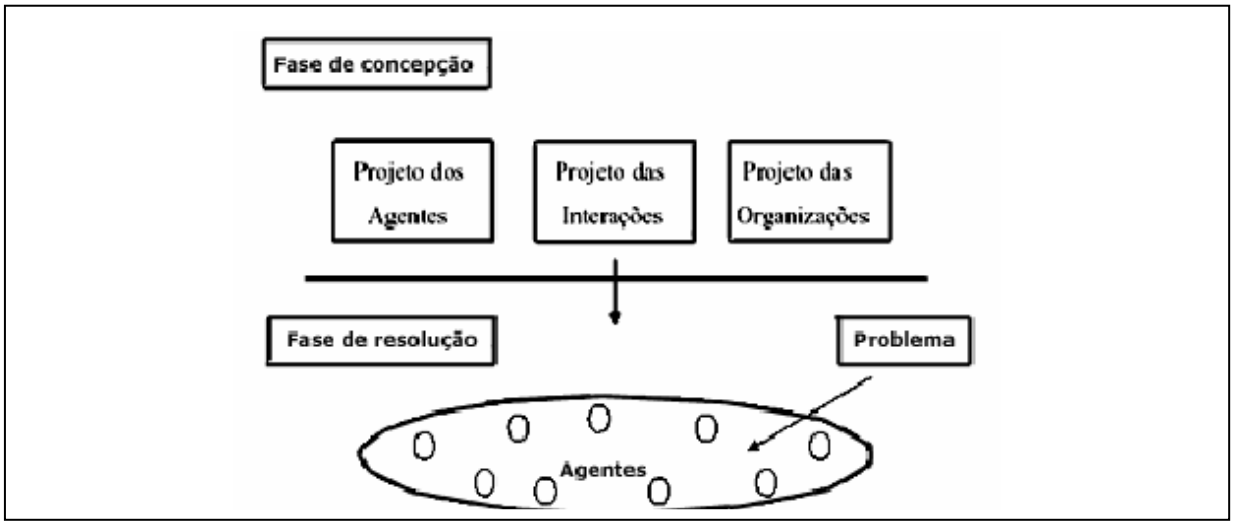

Figura 3.4: Abordagem SMA (SICHMAN e ALVARES, 1997)

(Demazeau, 1995) e (SICHMAN e ALVARES, 1997) descrevem algumas considerações sobre a abordagem SMA, nas quais os agentes:

- Devem ser capazes de decompor as tarefas baseando-se no conhecimento que eles possuem de si próprios e dos outros agentes;
- Por serem autônomos, podem possuir metas próprias e decidir o que fazer, a qualquer momento;
- Possuem capacidade para resolver seus problemas e os problemas que surgirem no ambiente;
- Podem entrar e sair do ambiente a qualquer momento. Portanto, em SMA os agentes devem ser capazes de modificar o conhecimento que possuem dos outros agentes do ambiente;
- Devem ser capazes de reconhecer modificações no ambiente quando estas ocorrerem, alterando sua representação interna do ambiente.

3.3.1 Percepção

Para que um agente execute uma ação inteligente, o mesmo deve estar capacitado a perceber as mudanças que ocorrem no ambiente em que o agente está inserido. Para animais

este problema é resolvido através de sensores para o tato, cheiro, gosto, som e visão. Isto pode ainda trazer um problema para o agente, pois muita informação pode ser gerada por este fluxo constante de informação gerado pelos sensores. Para minimizar este efeito deve-se filtrar as informações recebidas pelos sensores e ignorar as ocorrências que já são esperadas ou comuns. O agente deve, desta maneira focar sua atenção em mudanças não esperadas em seu ambiente.

Com isso os sensores do agente devem estar habilitados a distinguir eventos normais de eventos significativos. Estes sensores irão captar informações do ambiente de maneira ativa através de pesquisas com outros agentes ou passivamente através do recebimento de mensagens de evento do sistema, usuário ou outros agentes.

3.3.2 Comunicação

Após a percepção do agente, o mesmo pode, antes de tomar alguma ação ou estar convicto de que não há ação a tomar, mandar uma mensagem a outro agente. Neste sentido a comunicação tem papel fundamental na integração entre os agentes.

Para que uma sociedade de agentes coopere a fim de atingir um determinado objetivo, é necessário que seja definida uma arquitetura que possibilite a interação entre esses agentes. Nestas interações irá ocorrer troca de conhecimento, objetivos, planos ou escolhas através da comunicação, que pode ser direta ou indireta (JAQUES, 1999).

Na comunicação direta os agentes se conhecem e por isso trocam informações diretamente entre si. Já na comunicação indireta os agentes não se conhecem e, desta maneira a comunicação irá ocorrer através de uma estrutura de dados compartilhada, como na abordagem da arquitetura de quadro-negro que veremos mais tarde.

Na comunicação direta os agentes trocam mensagens entre si diretamente, para atingirem um objetivo global. Desta maneira, cada agente possuirá uma representação dos outros agentes, tais como capacidade, objetivo, conhecimento e crenças (JAQUES, 1999).

Segundo (DEMAZEAU e MÜLLER, 1990), existem basicamente três tipos de interações entre os agentes. Neste caso, o tipo de informação que será compartilhada, conhecimento, possibilidades ou escolha, irá determinar o tipo de interação.

Quando há uma interação forte entre os agentes, eles compartilham conhecimentos, possibilidades e escolhas. Neste tipo de comunicação ocorrem protocolos de comunicação sofisticados para os agentes, tais como informar, requisitar ou convencer.

Quando os agentes compartilham apenas conhecimento e possibilidades, a interação entre os agentes é média. Este tipo de interação ocorre quando os agentes desejam apenas saber o que os outros agentes pretendem fazer. Desta maneira, os agentes irão conhecer os planos para execução de tarefas dos outros agentes para que um plano comum seja encontrado.

A interação pode ainda ser fraca, quando os agentes trocam conhecimento apenas. Esta interação ocorre através de trocas de mensagens que serão ocasionadas pela percepção de alterações no ambiente.

A comunicação direta entre agentes abrange dois modelos: O modelo Cliente-Servidor e o modelo *peer-to-peer* (FREITAS, 2002). O modelo Cliente-Servidor baseia-se em chamadas a procedimentos remotos, enquanto internamente efetua uma comunicação do tipo pedido-resposta com os parâmetros do procedimento solicitado.

Já o modelo *peer-to-peer* é baseado na Teoria dos Atos de Fala (FREITAS, 2002). A teoria de atos de fala propõe uma categorização de primitivas de comunicação como, por exemplo, *inform*, *ask-to-do*, *answer*, *propose*, etc. Essas primitivas são associadas às suas conseqüências (ALVARES e SICHMAN, 1997).

Os atos de fala podem ainda ser classificados como assertivos (informar), diretivos (pedir ou consultar), comissivos (prometer ou comprometer-se), proibitivos, declarativos (causar eventos para o próprio comunicador) e expressivos (emoções) (FREITAS, 2002).

Quando duas entidades autônomas, inteligentes tentam se comunicar, as mesmas devem falar a mesma linguagem que utiliza o mesmo conjunto de átomos ou símbolos. As mensagens devem referir-se a um contexto e vocabulário comuns, sobre o qual a troca de mensagens seja efetuada dentre de uma semântica bem definida e sem ambigüidades. Os agentes precisam ter um vocabulário compartilhado denominado de ontologia (BIGUS, Joseph e BIGUS, Jeniffer, 2001) que é descrito em maior detalhe no capítulo a seguir.

3.3.3 Linguagens de Interação entre Agentes

Dentre as diversas linguagens para interação entre agentes se destaca a KQML (Knowledge Query and Manipulation Language) que fornece um framework para um conjunto de agentes independentes para comunicar e cooperar utilizando mensagens específicas.

KQML é uma linguagem de alto nível usada por sistemas baseados em conhecimento para compartilhar conhecimento e informação em tempo de execução. KQML é uma linguagem externa que foca principalmente o formato das mensagens. Em KQML, apenas protocolos de mensagens são definidos, sendo que pode ser usada uma outra linguagem para expressar o conteúdo da mensagem. As mensagens KQML são chamadas performativas e estão baseadas na Teoria de Atos de Fala. Cada mensagem é enviada com o objetivo de gerar uma determinada ação (JAQUES, 1999).

KQML é uma linguagem dividida em três camadas: a camada do conteúdo, a camada da mensagem e a camada da comunicação. O conteúdo refere-se ao atual conteúdo da

mensagem, a qual toda implementação KQML ignora, exceto para determinar as delimitações deste conteúdo (JAQUES, 1999).

O nível de comunicação descreve parâmetros de nível mais baixo, tais como identidade do emissor e receptor da mensagem e um único identificador associado à comunicação. A camada de mensagem, por sua vez, identifica o tipo de mensagem que um agente quer passar para outro, o que vai determinar o tipo de interação que vai ocorrer entre eles. A função primária desta camada é identificar o protocolo a ser usado e fornecer uma performativa a qual o emissor irá unir ao conteúdo. Então, como o conteúdo não é conhecido por KQML, serão também adicionadas outras descrições que irão identificar a linguagem e a ontologia que descreve o conteúdo.

```
(ask-one                          (tell
:sender agente1                   :sender agente2
 :content (PRECO MOUSE ?preco)     :content (PRECO MOUSE 14)
 :receiver agente2                 :receiver agente1
 :reply-with estoque-mouse         :in-reply-to estoque-mouse
 :language LPROLOG                 :language LPROLOG
 :ontology ONT1)                   :ontology ONT1)
```

Figura 3.5: Exemplo de troca de mensagens em KQML

Nesta mensagem da Figura acima, a performativa KQML é *ask-one*, o conteúdo é *(preco mouse ?preco)*, a ontologia é *ont1*, o receptor da mensagem é um agente denominado de agente2 e a consulta é escrita em uma linguagem chamada LPROLOG. O valor do conteúdo é a camada de conteúdo, os valores das palavras *:reply-with*, *:sender*, *:receiver* formam a camada de comunicação e o nome da performativa juntamente com *:language* e *:ontology* formam a camada da mensagem. O agente *agente2*, fornece a *agente1* a mensagem que se encontra a direita.

3.3.4 Ação

Uma vez que um agente tenha percebido e reconhecido que um evento significativo tenha ocorrido, o próximo passo é tomar alguma ação. Como diversos animais, os agentes atuam sobre seu ambiente através de atuadores (como os músculos, por exemplo).

A ação é visualizada como uma forma de intervenção do agente no ambiente, sendo capaz de modificá-lo. Quando existirem no ambiente intermediários, outros agentes ou sistemas desconhecidos, então algumas precauções e checagens devem ser realizadas.

3.3.5 Arquitetura de Quadro-Negro

Como já foi dito anteriormente quando os agentes não se conhecem é necessário que haja uma estrutura de dados compartilhada para que os agentes interajam entre si. Neste tipo de sociedade, uma boa abordagem a ser utilizada é a arquitetura de quadro-negro (*blackboard*).

O quadro-negro é uma estrutura única e compartilhada entre vários agentes, onde as informações serão escritas e lidas durante o desenvolvimento das tarefas. Como não há comunicação direta entre os agentes, eles devem consultar a estrutura de tempos em tempos para verificar se existe alguma informação destinada a eles. A estrutura de quadro-negro pode armazenar primitivas simbólicas (hipóteses), fatos e regras. O esquema da estrutura de quadro-negro pode ser visualizado na Figura a seguir.

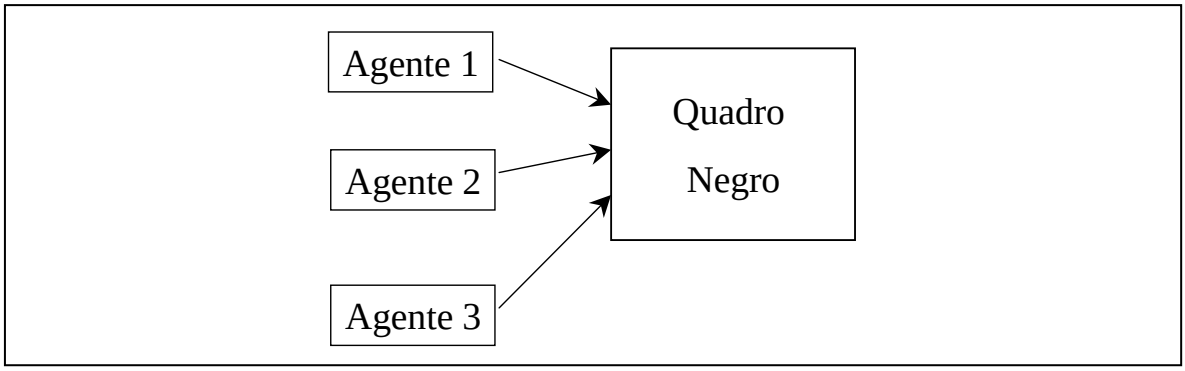

Figura 3.6: Estrutura de Quadro-Negro (OLIVEIRA, 1996)

As estruturas de quadro-negro são anteriores aos sistemas multiagentes (JAQUES, 1999). Um dos primeiros sistemas a utilizar essa abordagem foi o HEARSAY II (OLIVEIRA, 1996), um sistema para interpretação da fala. No HEARSAY II existe um sistema de controle que supervisiona a escrita e leitura no quadro-negro. Estas informações serão colocadas na estrutura por um conjunto de processos, que no sistema HEARSAY II eram conhecidos como *knowledge sources.*

Embora essa abordagem seja simples de implementar e eficiente, um dos problemas que ocorrem na estrutura de quadro-negro é a sincronização e resolução de conflitos entre os agentes, devido à centralização da informação e do controle (JAQUES, 1999).

3.4 Expert-Coop++

O ambiente Expert-Coop++ é uma biblioteca para construção de sistemas multiagentes orientada a objetos sob restrições de tempo real do tipo melhor esforço (COSTA e outros, 2003). O ambiente é na verdade uma biblioteca de classes escrita em C++ do qual o desenvolvedor pode reutilizar o código para desenvolver aplicações específicas.

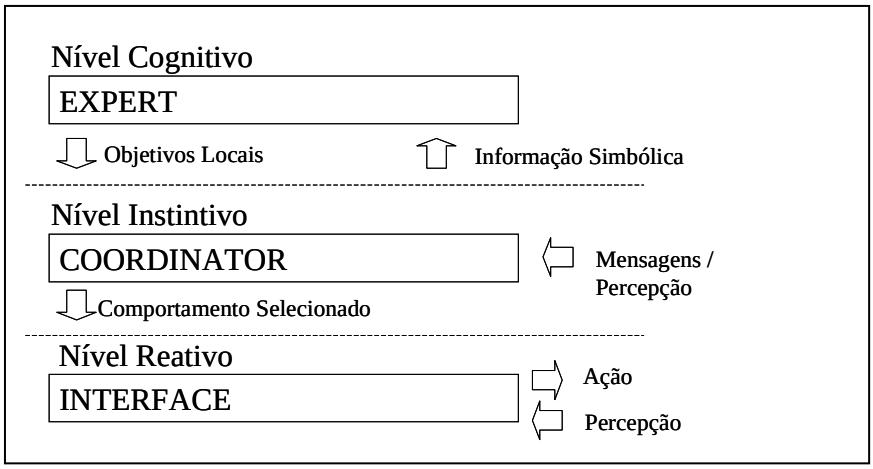

Figura 3.7: Arquitetura do Agente Autônomo Concorrente (COSTA e outros, 2003)

Dentre os componentes que estão inseridos na Expert-Coop++ se destaca o modelo de arquitetura pré-definido para o agente cognitivo denominada de Agente Autônomo

Concorrente (COSTA e Bittencourt, 1999) ; uma linguagem de comunicação de agentes, Parla (COSTA e Bittencourt, 1997); e três estratégias de cooperação para agentes cognitivos, *Contract Net Protocol* (CNP); uma variação da Coalisão Baseada em Dependência (CBD*), a estratégia de cooperação Conhecimento Social Dinâmico (CSD) e um conjunto de comportamentos reativos implementados através de controladores difusos, para o problema do futebol de robôs (COSTA e outros, 2003), segundo a categoria de robôs simulados da RoboCup Federation (KITANO e outros, 1997).

3.4.1 Arquitetura do Agente Autônomo Concorrente

A arquitetura do Agente Autônomo Concorrente possui 3 níveis decisórios que são executados de maneira concorrente. Estes níveis são utilizados através de 3 processos: *Interface*, *Coordinator* e *Expert*, sendo que cada um é responsável pelos níveis decisórios: reativo, instintivo e cognitivo.

Conforme mostrado na Figura 2.7, o nível reativo é implementado no processo Interface. Através de controladores difusos este processo percebe as mudanças no ambiente e ao mesmo tempo atua sobre o mesmo através de comportamentos reativos. Apenas um controlador é selecionado para estar ativo a cada instante de acordo com as variáveis de estado conFiguradas de acordo com o ambiente.

Segundo Costa (2003) o nível reativo realiza as seguintes atividades:

1. Lê a mensagem armazenada no mailbox;
2. Se a mensagem foi enviada pelo nível instintivo, então um dos seguintes métodos poder ser acionados: trocar o controlador ativo, ativar ou desativar a saída do processo.
3. Caso a mensagem seja procedente do ambiente, as variáveis linguísticas são atualizadas, o controlador difuso ativo é acionado, as variáveis linguísticas de

saída são “defuzificadas” e convertidas em comandos a serem enviados novamente ao ambiente.

Já o nível instintivo é implementado através do processo *Coordinator* e tem como função reconhecer as mudanças de estado do ambiente; atualizar a informação simbólica utilizada no nível cognitivo; escolher o comportamento reativo mais adequado (COSTA e outros, 2003). Este comportamento vai selecionar qual controlador difuso vai estar ativado em um dado momento para que uma meta local ou um estado seja alcançado.

De acordo com Costa e outros (2003) o funcionamento do nível instintivo implementado no processo Coordinator pode ser brevemente descrito por:

1. Lê a mensagem armazenada no mailbox;
2. Armazena na base de fatos;
3. Carrega fatos e regras no motor de inferência[1];
4. Envia mensagens para os níveis reativo e/ou cognitivo.

O processo *Coordinator* possui em sua estrutura objetos que executam um motor de inferência de apenas um ciclo de inferência utilizado para selecionar a sequência de comportamentos a serem utilizados no agente. Neste caso o sistema baseado em conhecimento é formado por uma base de regras fixa, uma base de fatos que armazena a percepção enviada pelo nível instintivo e as mensagens enviadas pelo nível cognitivo contendo a meta local (COSTA e outros, 2003).

Na verdade as regras no nível instintivo é que vão classificar os estados do ambiente e escolher o comportamento mais adequado para daí gerar as informações para o nível cognitivo. São admitidas as formas de representação de conhecimento de lógica e quadros no lado esquerdo e direito das regras, além de mensagens no formato da linguagem Parla.

[1] Estaremos tratando em maiores detalhes sobre o motor de inferência no próximo capítulo.

Já o nível cognitivo tem a responsabilidade de manter um modelo lógico do ambiente, estabelecer as metas locais do agente e interagir com outros agentes da comunidade, com o objetivo de estabelecer metas globais para a realização de uma ação cooperativa.

3.4.2 O Nível Cognitivo no Expert-Coop++

O nível cognitivo no Expert-Coop++ possui um sistema baseado em conhecimento encapsulado, que manipula tanto as informações simbólicas recebidas do nível instintivo, quanto as mensagens assíncronas recebidas dos demais agentes da comunidade (COSTA e outros, 2003). Da mesma maneira que o nível instintivo, o nível cognitivo precisa de um arquivo de regras para sua base local e social. Aqui se destaca o motor de inferência atualmente implementado com base no encadeamento progressivo clássico.

Como mostrado na Figura 2.8, a base de fatos armazena informações simbólicas enviadas pelo nível instintivo além de mensagens recebidas dos demais agentes. Além destas informações o motor de inferência com o disparo das regras também gera informações para a base de fatos. As mensagens de saída podem ser mensagens no formato Parla enviadas a outros agentes ou mensagens enviadas ao nível instintivo, sendo que neste último caso as mensagens irão se tornar metas locais. Para mensagens enviadas a outros agentes, as mesmas se constituem em informações a serem utilizadas em ação cooperativa dos agentes. Como já pode-se supor a base de regras local armazena regras que serão utilizadas para as metas locais e as base de regras social é utilizada para as metas que visam ação cooperativa.

O nível cognitivo determina a meta local e a envia para o nível instintivo. Esta meta vai disparar uma regra no motor de inferência do nível instintivo, selecionando o comportamento reativo mais adequado de acordo com o estado do ambiente e a meta desejada. Enquanto a meta não é atingida ou ocorre alguma falha, o nível cognitivo utiliza este tempo livre para realização de planejamento. Este planejamento vai determinar as

possíveis metas futuras e as especificações das requisições de cooperação para alcançar as metas globais. Além das metas locais, o nível cognitivo determina as interações de um agente com os demais da comunidade, buscando atingir as metas globais.

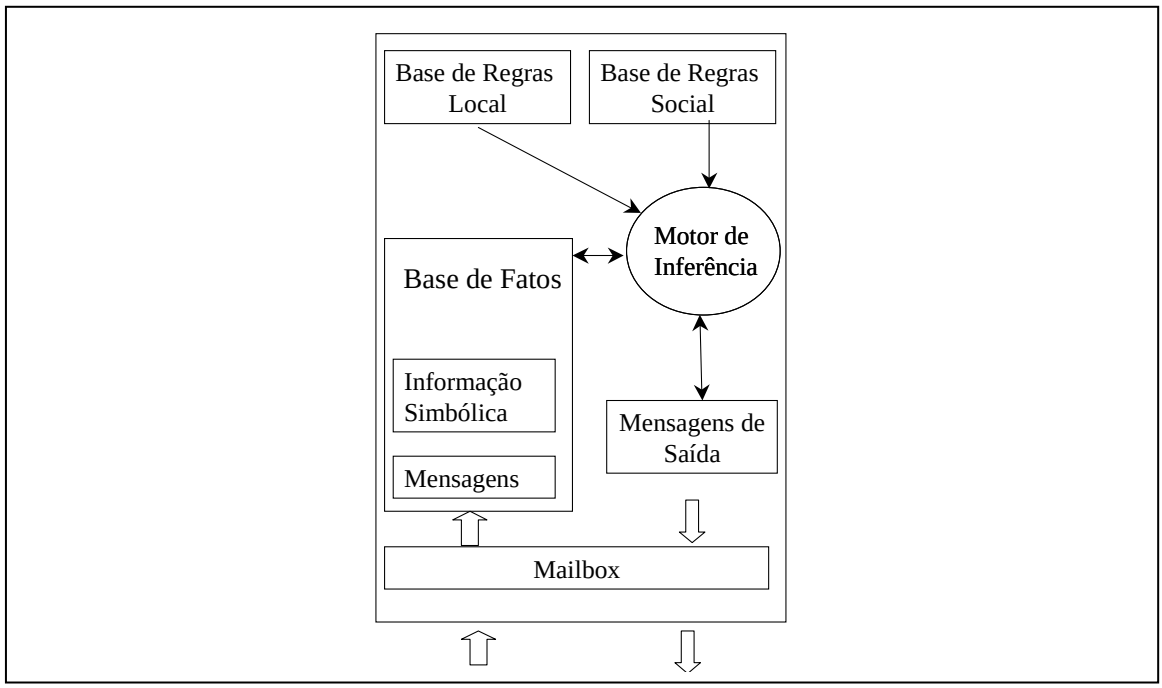

Figura 3.8: Nível Cognitivo (COSTA e outros, 2003)

Tanto o nível cognitivo como o instintivo são implementados segundo um sistema baseado em conhecimento, entretanto com objetivos e restrições de tempo diferentes. O nível cognitivo pode gastar mais tempo com planejamento por esta atividade ser mais complexa que as demais. O nível instintivo e reativo, por sua vez são responsáveis pela interação do agente com o ambiente com restrições de tempo real.

3.4.3 Base de Conhecimento Social

A base de conhecimento social localizada no nível cognitivo que é utilizada para realizar a cooperação entre os agentes consiste em (COSTA e outros, 2003):

- Lista de Conjunto de Planos, que define quais as ações cooperativas que o agente está apto a integrar;
- Uma meta global;

- Plano ativo, que é o plano que está sendo apreciado pelos agentes que integram o processo de cooperação;
- Uma lista de agentes, que podem integrar o processo de cooperação;
- Uma base de regras, que define como o agente vai responder às inciativas dos demais em uma ação cooperativa.

3.5 Representação de Conhecimento no Expert-Coop++

A Base de regras no Expert-Coop++ pode conter tanto elementos do tipo lógica, como quadros ou mensagens seguindo o formato da linguagem Parla. Vejamos o exemplo de uma regra do Expert-Coop++ abaixo:

```
(rule_000
     (if   (logic ( local_goal new ?x1 ))
           (frame ( ?xst1 ((goal r) (distance ?x3)))))
  (filter ( !=  ?x1 none ))
     (then (message ((to RL) (from Coord) (deadline 0)
           (grade 0.0) (alpha 0.0) (round 0.0)
           (body (INFORM ((logic (rule_000 ))))))))
        (logic ( local_goal current ?x1 ))
        (logic ( local_goal status active ))
        (logic ( local_goal defense none ))))
```

Com esta regra podemos perceber que variáveis podem ser colocadas tanto no lado esquerdo como no lado direito das regras (*?x1* é uma variável no exemplo). É possível ainda a utilização de filtros para estas variáveis (no exemplo anterior `(filter ( != ?x1 none ))` é uma instrução de filtro). Quando existirem variáveis no lado direito, a substituição é realizada para que na base de fatos. Se no exemplo anterior confrontarmos o fato `(logic ( local_goal new move_to_ball ))`, a variável ?x1 será substituída por *move_to_ball* quando a mesma for disparada.

Para o caso dos quadros, no exemplo `(frame ( ?xst1 ((goal r) (distance ?x3)))),` o Expert-Coop++ possui alguns elementos fixos, como os *slots goal* e *distance* utilizados para representar elementos visuais para o agente, como nome do

objeto, distância, direção, dentre outros. Para realizar a inferência, neste caso, não será necessária a checagem de cada um dos elementos do fato, sendo necessário somente a checagem dos atributos especificados na regra. Se o fato possui mais atributos que a regra, estes demais não serão checados na inferência. Com o quadro `(frame ( bola ((goal r) (distance 5.0) (direction 3))))` o mesmo seria sensibilizado na inferência, apesar do atributo *direction* não se encontrar na regra. O mesmo pode ocorrer com as mensagens no formato Parla que são trocadas entre os agentes (COSTA e Bittencourt, 1997).

3.6 Resumo do Capítulo

Neste capítulo foram apresentados conceitos sobre Sistemas Multiagentes, incluindo a contextualização em que está inserido dentro da IAD. Foram descritos ainda conceitos relacionados a tecnologia de agentes, bem como suas atividades principais, o enfoque SDP e SMA, e a biblioteca Expert-Coop++.

O motor de inferência descrito no capítulo 5 é incorporado à biblioteca Expert-Coop++, uma biblioteca em C++ para a construção de Sistemas Multiagentes.

O Expert-Rete, ao final deste trabalho é integrado ao Expert-Coop++ e, portanto, interage com diversas classes do mesmo.

4. MÉTODOS DE INFERÊNCIA

Em um Sistema Baseado em Conhecimento, o motor de inferência determina quais as regras (ou, no caso de regras de produção, mais especificamente o lado esquerdo das mesmas), se existirem, que são satisfeitas pelos fatos. Dois métodos são comumente utilizados para realizar a inferência: o *forward chaining* ou encadeamento progressivo e o *backward chaining* ou encadeamento regressivo. Estes, entretanto, não são os dois únicos métodos de inferência existentes, pois existem ainda outros métodos como *means-ends-analysis* ou análise de significado final, e o algoritmo Rete.

As técnicas de inferência acima citadas são apresentadas neste capítulo, sendo que o motor de inferência Expert-Rete se baseia em um desses métodos, o algoritmo Rete.

4.1 Lógica Matemática

Segundo (BITTENCOURT, 2001a), os formalismos lógicos e suas extensões possuem grande influência nas ciências cognitivas e hoje grande parte da pesquisa em IA está ligada direta ou indiretamente na lógica. No enfoque simbólico da IA a lógica tem papel fundamental.

De maneira geral, um sistema lógico consiste em um conjunto de fórmulas e um conjunto de regras de inferência. As fórmulas são sentenças pertencentes a uma linguagem formal cuja sintaxe é dada. Cada fórmula pode ser associada a um valor verdade(V) ou um valor falso(F).

Uma regra de inferência é uma regra sintática que, quando aplicada repetidamente a uma ou mais fórmulas verdadeiras, gera apenas fórmulas verdadeiras (BITTENCOURT, 2001a). As regras de inferência fornecem uma estrutura dedutiva à linguagem lógica. A sequência de fórmulas geradas através da aplicação de regras de inferência sobre um conjunto inicial de fórmulas é denominada de prova.

A partir da introdução por Robinson (1965) de procedimentos eficientes para demonstração automática de teoremas por computador, a lógica passou a ser estudada também como método computacional para a solução de problemas. Este procedimentos exploram o fato de expressões lógicas poderem ser colocadas em formas *canônicas*, ou seja, expressões com um número de operadores restringido (em geral “e”, “ou” e “não”).

A teoria de resolução, proposta por Robinson (1965) parte da transformação a ser provada para a forma canônica conhecida como forma normal conjuntiva ou forma clausal. O método é baseado em uma inferência única, chamada regra de resolução, e utiliza intensivamente um algoritmo de casamento de padrões denominado de algoritmo de unificação. Já o método *tableaux* (SMULLYAN, 1968) consiste em construir uma forma canônica conhecida como forma normal disjuntiva ou forma clausal dual. Estes são métodos de refutação, ou seja, para provar um teorema a partir de um conjunto de hipóteses, nega-se o teorema e prova-se que o conjunto de expressões formado pelas hipóteses e pelo teorema negado é contraditório.

Formalmente, uma linguagem lógica de primeira ordem possui a notação L(P, F, C, V), onde estes símbolos representam(BITTENCOURT, 2001a):

- Conjunto P de símbolos de predicado
- Conjunto F de símbolos de função
- Conjunto C de símbolos de constante
- Conjunto V de símbolos de variável

Segue abaixo um exemplo de uma linguagem lógica de primeira ordem escrita na notação BNF (Backus-Naur Formalism):

<termo>::=<variável> | <constante> | <função> (<termo>$_1$, ..., <termo>$_n$)

<fórmula-atômica>::= V | F | <predicado> (<termo>$_1$, ..., <termo>$_n$)

<fórmula>::=<fórmula-atômica> | ¬ (<fórmula>) | (<fórmula>$_1$ ∧ <fórmula>$_2$) | (<fórmula>$_1$ ∨ <fórmula>$_2$) | (<fórmula>$_1$ → <fórmula>$_2$) | (∀<variável> . (<fórmula>)) | (∃<variável> .(<fórmula>))

<constante>::= a,b,c,d e outras palavras iniciadas por minúsculas

<variável>::= x, y, z, w com ou sem índices

<função>::= f, g, h e outras palavras iniciadas por minúsculas

<predicado>::= P, Q, R e outras palavras iniciadas por maiúsculas

Neste caso os símbolos V e F são denominados de verdadeiro e falso. Os símbolos de operadores lógicos têm os seguintes nomes: ¬ “não”, ∧ “e”, ∨ “ou”, → “implica”, ∀ “qualquer ” e ∃ “existe”. Estes dois últimos operadores são chamados de quantificadores.

Com esta linguagem é possível a geração de fórmulas como estas abaixo:

((Pai(a,b) ∧ Pai(b,c) → Avo(a,c)), ¬(Ama(brutus,cesar)), ∀x.((P(x) ∧¬(P(a))) → Q(b)), Ama(amelia,z), ∀x. ∃y.(Gosta(x, y)).

Vale ressaltar que as variáveis podem ocorrer de duas maneiras: livres ou ligadas. As variáveis ligadas devem estar sob o escopo de algum quantificador, como as variáveis x e y em ∀x. ∃y.(Gosta(x, y)). As variáveis livres não sofrem esta restrição, como a variável z em Ama(amelia,z).

4.1.1 Métodos de Prova

Métodos de prova são métodos operacionais capazes de determinar se uma fórmula é ou não consequência lógica de um conjunto de fórmulas (BITTENCOURT, 2001a). Os métodos de prova se utilizam de regras de inferência a partir de um dado conjunto de

fórmulas para que seja possível a geração de novas fórmulas, provando que uma dada fórmula é consequência lógica do conjunto inicial.

Desta maneira, se a partir de fórmulas verdadeiras, uma regra de inferência produzir apenas fórmulas verdadeiras, a regra é dita correta. Por serem sempre verdadeiras, tautologias, quando utilizadas como base para regras de inferência, levam a regras corretas. O que permite que tautologias sejam utilizadas como regras de inferência é uma outra regra de inferência, mais geral, conhecida como *regra da substituição*, que afirma que uma tautologia formada por símbolos proposicionais permanece uma tautologia quando estes símbolos são substituídos por fórmulas lógicas arbitrárias (BITTENCOURT, 2001a). As seguintes tautologias são frequentemente utilizadas como regras de inferência, para A e B fórmulas genéricas :

- $(A \wedge (A \rightarrow B)) \rightarrow B$, denominada de *Modus Ponens*
- $(\neg B \wedge (A \rightarrow B)) \rightarrow \neg A$, denominada de *Modus Tollens*
- $((A \rightarrow B) \wedge (B \rightarrow C)) \rightarrow (A \rightarrow C)$, conhecida como Silogismo Hipotético
- $\forall x.A \rightarrow A\{x/a\}$, Especialização
- $A\{x/a\} \rightarrow \exists x.A$, Generalização

4.1.2 Unificação

Duas fórmulas são unificáveis se existir uma substituição que, ao ser aplicada a ambas as fórmulas, torna-as indênticas (BITTENCOURT, 2001a). As fórmulas atômicas Ama(x, carlos, z) e Ama(amelia, y, z) podem ser unificadas pela aplicação da substituição {x/amelia, y/carlos, z/muito}, resultando na fórmula Ama(amelia, carlos, muito), por exemplo. As substituições , por outro lado, que unificam duas fórmulas atômicas serão, forçosamente, composições construídas a partir de uma única substituição, chamada de unificador geral.

O algoritmo descrito formalmente por Robinson (1965) é capaz de encontrar o unificador mais geral entre duas fórmulas atômicas, ou retornar falha caso o unificador mais geral não exista. O algoritmo é definido recursivamente e pode ser aplicado tanto em fórmulas atômicas quanto a termos. A recursão termina quando é encontrado um símbolo atômico, isto é, um símbolo proposicional, um símbolo de constante ou um símbolo de variável.

Este algoritmo consiste em dois casos principais: o primeiro quando uma das fórmulas é um símbolo atômico, ou seja, uma variável ou um símbolo de constante, e o segundo quando ambas as fórmulas são fórmulas atômicas ou termos compostos por mais de um símbolo. No primeiro caso existem três possibilidades:

1. As duas fórmulas são símbolos e estes são iguais, resultando na substituição vazia;
2. Uma das fórmulas é uma variável que não ocorre na outra, resultando em uma substituição com um único par;
3. Outros casos resultando em falha.

Já no segundo caso, se ambas as fórmulas forem termos de mesma aridade e mesmo símbolo de predicado ou função, então o resultado é a composição das diversas substituições necessárias para unificar os subtermos correspondentes em ambas as fórmulas. Caso algum par de subtermos não seja unficável, ou caso não seja possível combinar as diversas substituições, o algoritmo retorna a falha.

Desde a proposta de Robinson (1965) foram propostos diversos algoritmos de unificação mais eficientes (BITTENCOURT, 2001a), já que o interesse nestes algoritmos é grande devido à posição estratégica ocupada pela unificação na maioria das ferramentas de IA.

4.2 “Forward Chaining” ou Encadeamento Progressivo

Ao conjunto de várias inferências que ligam o problema com a sua solução dá-se o nome de *chaining* ou encadeamento (OLIVEIRA, 2001). O encadeamento que é percorrido tendo como ponto de partida o problema e como alvo a solução é denominado de *forward chaining* ou encadeamento progressivo.

Conforme descrito em Russel e Norvig (1995), a idéia é a de iniciar com sentenças atômicas na base de conhecimento e aplicar *Modus Ponens* em uma direção progressiva, adicionando novas sentenças atômicas (lógica de primeira ordem), até que nenhuma inferência adicional tenha a possibilidade de ser aplicada. Cláusulas reduzidas tais como Situação $\rightarrow$ Resposta são especialmente úteis para sistemas que realizam inferências em resposta a novas informações. Muitos sistemas podem ser definidos desta maneira, e o raciocínio com o encadeamento progressivo pode ser mais eficiente do que somente por resolução de teoremas. De qualquer forma é sempre interessante construir uma base de conhecimento utilizando somente cláusulas reduzidas de maneira que o custo na resolução pode ser evitado.

No encadeamento progressivo a parte esquerda da regra é comparada é com a descrição da situação atual (fatos), contida na memória de trabalho. O raciocínio aí encontrado é no sentido dos fatos para conclusões resultantes, daí a denominação de encadeamento dirigido por dados. Desta maneira, as regras que satisfazem a esta descrição têm sua parte direita executada, o que, em geral, significa a introdução de novos fatos (BITTENCOURT, 2001a) à memória de trabalho.

Demonstrando o processo envolvido no forward chaining temos as seguintes regras:

- nome(x) $\rightarrow$ homem(x)
- homem(x) $\rightarrow$ mortal(x)

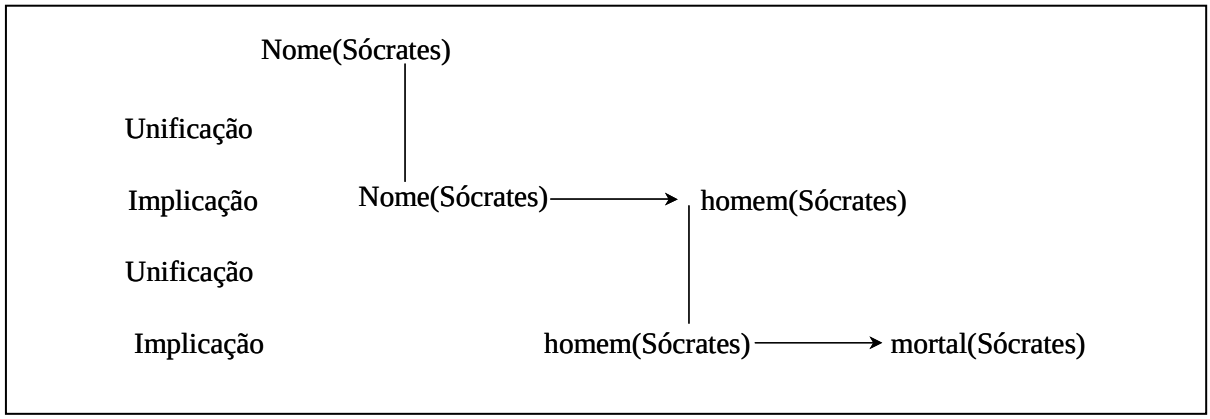

Figura 4.1: Encadeamento Progressivo

Estas regras são passíveis de serem utilizadas em um encadeamento progressivo de inferências que deduz que Sócrates é mortal, dado que Sócrates é homem, como mostra a Figura a seguir

Na Figura 4.1 o encadeamento é representado pela seqüência de barras verticais, ligando a conclusão de uma regra ao antecedente de outra . Estas barras também indicam a unificação das variáveis aos fatos. Por exemplo, a variável x no predicado homem(x) tem que ser unificada com o fato homem (Sócrates) antes da regra homem poder ser aplicada, ou seja, o encadeamento causal é, na verdade, uma seqüência de implicações e unificações, como mostra a Figura a seguir.

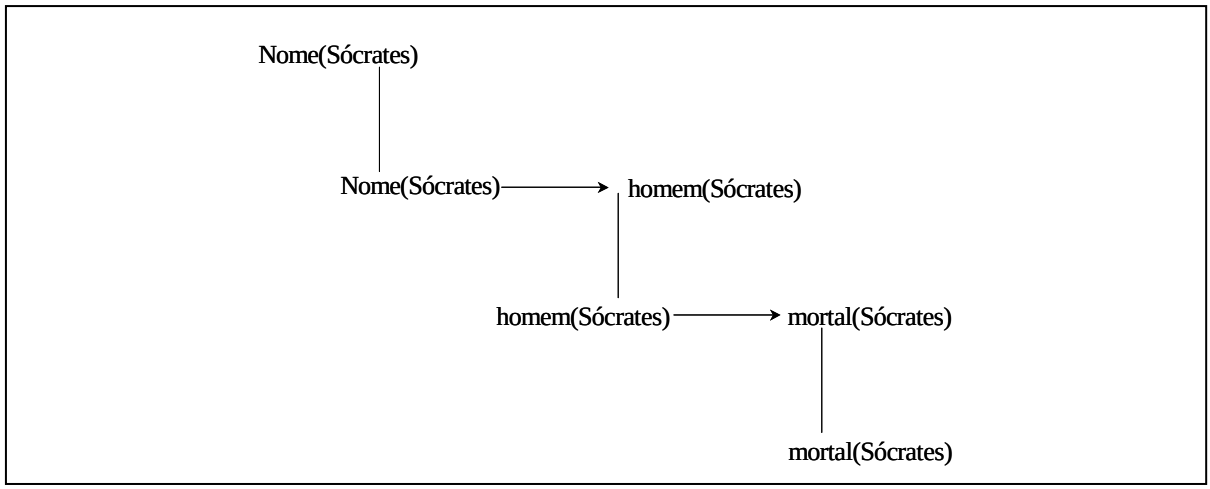

Figura 4.2: Encadeamento Progressivo mostrando seqüência de implicações e unificações

O encadeamento progressivo é também denominado raciocínio *bottom-up* (OLIVEIRA, 2001) porque parte de evidências de baixo nível, ou fatos, para as conclusões de alto nível que são baseadas nos fatos. Este tipo de raciocínio é análogo à programação convencional *bottom-up*. Os fatos são as unidades elementares do paradigma baseado em conhecimento, uma vez que não podem ser decompostos (elementos atômicos).

Uma cláusula reduzida de primeira ordem é atômica, ou é uma implicação do qual o antecedente é uma conjunção de literais positivos no qual conseqüente é um único positivo (RUSSEL E NORVIG, 1995). O exemplo a seguir ilustra melhor o que são clausulas reduzidas de primeira ordem:

$\text{Reino}(x) \wedge \text{Ambicioso}(x) \rightarrow \text{Malvado}(x)$.

Reino(João)

Ambicioso(y)

Ao contrário de literais proposicionais, literais de primeira ordem podem incluir variáveis, nas quais estas variáveis podem ser assumidas como universalmente quantificadas.

Nem toda base de conhecimento pode ser convertida em um conjunto de cláusulas reduzidas, mas muitas tem essa capacidade (RUSSEL E NORVIG, 1995). Considere o seguinte problema a ser analisado:

As leis proíbem que habitantes de um país, denominado País_Origem, vendam armas a nações hostis. O país País_Destino, considerado hostil, possui alguns mísseis, dos quais foram vendidos por Indivíduo_Origem, habitante de País_Origem.

Primeiramente serão representados os fatos na forma de cláusulas reduzidas de primeira ordem. A partir daí será mostrado como um algoritmo de encadeamento progressivo soluciona o problema.

Para a sentença “As leis proíbem ...vendam armas a nações hostis...”:

$\textit{País\_Origem}(x) \wedge \textit{Arma}(y) \wedge \textit{Vende}(x, y, z) \wedge \textit{Hostil}(z) \rightarrow \textit{Criminoso}(x)$ ***(R1)***

Já na trecho “País_Destino ...possui alguns mísseis...” a sentença $\exists x$ *Possui(Pais_Destino,x)* $\wedge$ *Misseis(x)* é transformado em duas cláusulas reduzidas pela regra de eliminação existencial, introduzindo uma nova constante C:

*Possui(Pais_Destino, C)****(R2)***

Mísseis(C) ***(R3)***

E “...dos quais foram vendidos por Indivíduo_Origem” tem-se:

Mísseis(x) $\wedge$ *Possui(País_Destino, x)* $\rightarrow$ *Vende(Indivíduo_Origem, x,País_Destino) **(R4)***

Temos que saber que mísseis são armas:

Mísseis(x) $\rightarrow$ *Arma(x)* ***(R5)***

*Hostil(x)=Hostil(Pais_Destino)****(R6)***

De “...por Indivíduo_Origem, habitante de País_Origem.” tem-se:

*Habitante_Pais_Origem(Indivíduo_Origem)***(R7)**

E, finalmente, de “*País_Destino, considerado hostil,...*”:

*Hostil(Pais_Destino,Pais_Origem)***(R8)**

Esta base de conhecimento não contém nenhum símbolo de função, sendo composto por um conjunto de cláusulas de primeira ordem. Este fato facilita muito o processo de inferência. O problema é que no mundo real as cláusulas nem sempre se encontram reduzidas e desenvolver um algoritmo para reduzir sempre em cláusulas reduzidas é algo complexo e custoso.

Um algoritmo simples de encadeamento progressivo se inicia de fatos conhecidos, dispara todas as regras cujas premissas são satisfeitas e adiciona suas conclusões a fatos conhecidos. O processo se repete até que o problema encontra solução ou nenhum fato é adicionado. Para um conjunto genérico de cláusulas reduzidas, um simples algoritmo de encadeamento progressivo pode gerar muitos novos fatos de maneira infinita. Para

determinados casos deve-se usar o teorema de Herbrand (RUSSEL E NORVIG, 1995) para estabelecer que o algoritmo vai encontrar uma prova.

Desta forma, para o conjunto de cláusulas reduzidas, as sentenças de implicação seriam as **R1**, **R4** e **R5,** sendo então necessárias duas iterações:

- Na primeira iteração, a regra R1 possui premissas não satisfeitas

  Regra R4 é satisfeita com {x/M1}, e *Vende(Indivíduo_Origem, M1,País_Destino)* é adicionado à base de conhecimento.

  Regra R5 é satisfeita com {x/M1}, e *Arma(M1)* é adicionada à base de conhecimento.

- Na segunda iteração, a regra R1 é satisfeita com *{x/Indivíduo_Origem, y/M1, z/País_Destino}*, e Criminoso(Indivíduo_Origem) é adicionado à base de conhecimento.

Neste exemplo pode ser gerada uma árvore de prova, pois cada sentença que pode ser concluída pelo encadeamento progressivo já deve estar contida explicitamente na base de conhecimento. Nesta árvore os fatos iniciais aparecem no nível mais baixo, fatos inferidos na primeira iteração em um nível intermediário, e fatos inferidos na segunda iteração no nível mais alto.

Para um algoritmo eficiente de encadeamento progressivo existem 3 fatores de complexidade que devem merecer atenção. O primeiro passo do algoritmo envolve encontrar todos os possíveis unificadores, com a premissa de que uma regra é unificada com um determinado conjunto de fatos na base de conhecimento. Este processo é denominado de casamento de padrão e pode ser muito custoso ao algoritmo (RUSSEL E NORVIG, 1995). No segundo ponto, o algoritmo verifica cada regra em cada iteração para ver se suas premissas são satisfeitas, mesmo se forem adicionados poucos elementos à base de conhecimento a cada

iteração. E por último, o algoritmo pode gerar muitos fatos que são irrelevantes ao objetivo global.

Para aplicar uma simples regra é necessário encontrar todos os fatos que se unificam ao lado direito da regra e isto pode ser realizado em períodos constantes de tempo. Já uma regra mais complexa pode ter o problema de ordenação de conjunções que representa encontrar uma ordenação para as conjunções da premissa da regra de tal maneira que custo total seja minimizado.

Segundo (RUSSEL E NORVIG, 1995) ainda 3 aspectos que devem ser levados em consideração quanto a base de regras:

- A maioria das regras em bases de conhecimento do mundo real são pequenas e simples ao invés de grandes e complexas. É comum no ambiente de banco de dados assumir que tanto o tamanho das regras e as aridades (número de argumentos) dos predicados estão atreladas a uma constante e a preocupação maior é quanto à complexidade dos dados, ou seja, complexidade da inferência como uma função do número de fatos no banco de dados.
- Podem ser consideradas subclasses de regras para que o reconhecimento seja eficiente. Os trechos mais relevantes das regras são utilizados e com isso o algoritmo pode chegar ao resultado final mais rapidamente.
- Pode-se direcionar um esforço para eliminar o trabalho de reconhecimento de regras redundantes no algoritmo de encadeamento progressivo.

4.3 “Backward Chaining” ou Encadeamento Regressivo

Ao contrário do encadeamento progressivo, quando um conjunto de inferências que ligam o problema com sua solução é percorrido a partir da hipótese até os fatos que a

suportam, este processo é denominado de *backward chaining* ou encadeamento regressivo (o processo é o inverso do encadeamento progressivo). Como o raciocínio parte dos níveis mais altos para os fatos de níveis mais baixos que suportam as hipóteses o raciocínio no encadeamento regressivo é também conhecido como *top-down* (OLIVEIRA, 2001).

No encadeamento regressivo um objetivo pode ser satisfeito diretamente por um elemento da memória de trabalho, ou podem existir regras que permitam inferir alguns dos objetivos correntes, isto é, contenham uma descrição deste objetivo em suas partes direitas (BITTENCOURT, 2001a). As regras que satisfazem esta condição têm as instâncias correspondentes às suas partes esquerdas adicionadas à lista de objetivos correntes. Caso uma dessas regras tenha todas as suas condições satisfeitas diretamente pela memória de trabalho, o objetivo em sua parte direita é também adicionado à memória de trabalho. Um objetivo que não possa ser satisfeito diretamente pela memória de trabalho, nem inferido através de uma regra, é abandonado. Quando o objetivo inicial é satisfeito, ou não há mais objetivos, o processo termina.

Um simples algoritmo de encadeamento regressivo é chamado com uma lista de objetivos contendo um único elemento, a pesquisa original, e retorna um conjunto de todas as substituições que satisfaçam a pesquisa. A lista de objetivos podem ser idealizados como uma estrutura de dados de pilha e se todos os objetivos podem ser satisfeitos, então o subsistema de provas é satisfeito. O algoritmo obtém o primeiro objetivo na lista e encontra cada cláusula na base de conhecimento, que seja positivo literal, ou chave, e realiza o processo de unificação com o objetivo. Cada cláusula como essa cria uma nova chamada recursiva na qual a premissa da cláusula é adicionada à pilha de objetivos.

O encadeamento regressivo é claramente um algoritmo do tipo busca em profundidade (RUSSEL E NORVIG, 1995). Isto significa que seu espaço de requisitos são lineares no

tamanho da prova. Isto também quer dizer que este tipo de mecanismo sofre de problemas com estados repetidos e incompletude.

É interessante observar que linguagens como o Prolog são executadas através do encadeamento regressivo (RUSSEL E NORVIG, 1995), onde cláusulas são dispostas na ordem na qual elas são escritas na base de conhecimento. Dentre outras características, a decisão do projeto do Prolog representou um compromisso entre a declaratividade e eficiência de execução. Existem ainda dois fatores encontrados no Prolog para maximizar a velocidade de processamento que podem ser considerados em algoritmos de encadeamento regressivo:

- O primeiro aspecto é o de que ao invés de construir uma lista de todas as possíveis respostas para cada sub objetivo antes de continuar para o próximo passo, o interpretador Prolog gera uma resposta e uma "promessa" de gerar o restante quando a resposta corrente tiver sido completamente explorada. Esta promessa é denominada de ponto de decisão (*choice point*). Quando a busca em profundidade completa a exploração das possíveis soluções que chegam da resposta corrente e retorna para o ponto de decisão, o ponto de decisão é expandido para receber uma nova resposta ao sub objetivo e a um novo ponto de decisão. Esta técnica economiza tanto em tempo como em espaço. Isto também proporciona uma interface simples para depuração por que em todo o tempo há somente um único caminho de solução sob consideração.
- Por fim, um simples algoritmo que implemente o encadeamento regressivo gasta muito tempo gerando e compondo substituições. O Prolog implementa substituições utilizando variáveis lógicas que podem ser lembradas de suas ligações. A qualquer instante no tempo, cada variável no programa é atrelada ou não a algum valor. Juntas, estas variáveis e valores definem implicitamente a substituição para o pedaço corrente de prova. Indo além de que o caminho só

> pode adicionar novas ligações de variáveis, por que uma tentativa de adicionar diferentes ligações para uma variável já ligada resulta numa falha de unificação. Quando um caminho na busca falha, o Prolog vai voltar para o ponto de decisão e então pode ter que desatrelar algumas variáveis. Quando um objetivo falha e é hora de voltar ao ponto de decisão prévio, cada uma das variáveis é desatrelada.

4.4 Comparação entre o Encadeamento Progressivo e Regressivo

O encadeamento progressivo é indicado claramente para solução de problemas que começam coletando informações, as quais levam a deduzir conclusões lógicas. Já no encadeamento regressivo se inicia com algumas hipóteses e então tenta-se provar estas hipóteses através do acúmulo de informação. Daí a característica encontrada no encadeamento progressivo de ser dirigido a dados e o regressivo ser dirigido por objetivos.

Dentre as vantagens do encadeamento progressivo está a de que ele é eficiente quando o problema geralmente começa pelo acúmulo de informações e o próprio problema pode estar inserido no contexto. Existe o problema deste tipo de mecanismo de inferência não reconhecer que algumas evidências são mais importantes que as outras por si só.

Já o encadeamento regressivo tem a vantagem de trabalhar bem quando o problema geralmente começa formando uma hipótese, e então é concluído que este pode ser provado. A desvantagem aí está em que este tipo de mecanismo sempre continuará, neste caso, seguindo uma determinada linha de raciocínio. O encadeamento regressivo também possui problemas com computações redundantes (RUSSEL E NORVIG, 1995). Muitas vezes o algoritmo envolve encontrar caminhos para sua solução na qual os objetivos são inalcançáveis

O tipo de encadeamento é normalmente definido de acordo com o tipo de problema a ser resolvido (BITTENCOURT, 2001a). Problemas de planejamento, projeto e classificação tipicamente utilizam encadeamento progressivo, enquanto que problemas de diagnóstico, onde existam apenas algumas saídas possíveis mas um grande número de estados iniciais, utilizam encadeamento regressivo.

Encadeamento Progressivo	**Encadeamento Regressivo**
Planejamento, monitoração e controle	Diagnósticos
Presente para o futuro	Presente para o passado
Antecedente para conseqüente	Conseqüente para antecedente
Raciocínio bottom-up baseado nos dados	Raciocínio top-down baseado no objetivo
Trabalhar para frente para encontrar soluções que derivam dos fatos	Trabalhar para trás para encontrar os fatos que suportam a hipótese
Busca em largura facilitada	Busca em profundidade facilitada
Antecedentes determinam a procura	Conseqüências determinam a procura
Explicação poderá não estar facilitada	Explicação está facilitada

Tabela 4.1: Comparação Encadeamentos Progressivo e Regressivo (OLIVEIRA, 2001).

Alguns sistemas especialistas utilizam as duas técnicas juntas de modo que os dois tipos de encadeamento se alternem de acordo com o desenvolvimento da solução do problema e com a disponibilidade dos dados.

4.5. Algoritmo Rete

O algoritmo Rete é um algoritmo conhecido, eficiente para casamento de padrões (baseado no encadeamento progressivo) para implementação de sistemas especialistas baseados em regras (UGAI, 2003). Este algoritmo foi desenvolvido inicialmente pelo Dr. Charles L. Forgy da Universidade de Carnegie Mellon em 1979 (FORGY, 1982). Rete se tornou a base para muitos sistemas especialistas populares, incluindo OPS5, CLIPS, JESS, e LISA. O algoritmo Rete foi desenvolvido com a intenção de aumentar a velocidade de sistemas de regras baseado no encadeamento progressivo por limitar o esforço de re-computação os conflitos após uma regra ser disparada. Ele obtém vantagem de duas observações empíricas (FORGY, 1982):

- Redundância Temporal: o disparo de uma regra freqüentemente muda somente alguns fatos, e somente algumas regras são afetadas por cada uma dessas mudanças.
- Similaridade Estrutural: o mesmo padrão constantemente aparece no lado esquerdo de mais de uma regra.

Os padrões encontrados em uma regra podem ser definidos como sendo expressões atômicas que são escritas em algum formalismo, com o objetivo de representar o conhecimento, podendo ou não conter variáveis. Os padrões aqui referenciados sempre se encontram no lado esquerdo das regras.

Uma implementação simples de um sistema especialista deve checar cada regra em relação a fatos conhecidos em sua base de conhecimento, disparando aquela regra, se necessário, movendo para a próxima regra (e voltando para a primeira regra quando o processo tiver terminado, num loop). Mesmo que para regras e fatos moderados em tamanho, este modelo de implementação pode ter problemas de performance.

O algoritmo Rete (“rete” em Latin significa rede) proporciona a base para uma implementação mais eficiente em um sistema baseado em conhecimento (UGAI, 2003). Um sistema baseado no Rete constrói uma rede de nós, onde cada nó (exceto a raiz) corresponde a um padrão de ocorrência no lado esquerdo de uma regra. A rede de nós deve ser, portanto, construída antes da realização da inferência propriamente dita.

O caminho do nó raiz até um nó folha no algoritmo define o lado esquerdo de uma regra. Cada nó possui uma memória de fatos que satisfaz o padrão. Assim que novos fatos são inseridos ou modificados, eles caminham pela rede, gerando o alerta dos nós quando os mesmos encontram o padrão definido nos fatos. Quando um fato ou uma combinação de fatos encontram todos os padrões de uma dada regra a ser satisfeita, o nó folha é alcançado e a regra correspondente é disparada.

O algoritmo Rete foi desenvolvido com o propósito de sacrificar a memória para aumentar a velocidade de processamento (UGAI, 2003). Em muitos casos o aumento em velocidade sobre simples implementações é muitas vezes maior (o algoritmo é em teoria independente do número de regras do sistema). Em sistemas robustos, com muitos padrões, entretanto, o algoritmo Rete tende a ter problemas de consumo de memória. A seguir serão mostrados detalhes do algoritmo.

4.5.1. Detalhes do Algoritmo Rete

O algoritmo Rete utiliza na verdade um grafo acíclico direcionado de raiz, a rede, onde os nós, com exceção da raiz, representam padrões, e caminhos da raiz para as folhas representam o lado esquerdo das regras. Em cada nó é armazenada informação sobre os fatos satisfeitos pelos padrões dos nós nos caminhos da raiz em diante e incluindo o próprio nó. Esta informação é uma relação representando os possíveis valores da ocorrência das variáveis

nos padrões do caminho. O Rete consiste de um nó raiz, de nós de padrão de uma entrada, e nós de junção de duas entradas.

O algoritmo Rete mantém a informação associada com os nós no grafo. Quando um fato é adicionado ou removido da memória de trabalho, um *token* representando o fato e a operação entra na raiz do grafo e é propagado para suas folhas, modificando a informação apropriada associada com os nós. Quando um fato é modificado, normalmente essa operação é expressa como uma retirada de um fato antigo e a adição de um novo fato.

A seguir será mostrado um exemplo simples do Rete, supondo as regras:

(R1 (possui-objetivo ?x simplificar)
(expressão ?x 0 + ?y)
⇒ ....)
(R2 (possui-objetivo ?x simplificar)
(expressão ?x 0 * ?y)
⇒ ....)

E os seguintes fatos:

(possui-objetivo e1 simplificado)
(expressão e1 0 + 3)
(possui-objetivo e2 simplificado)
(expressão e2 simplificado)
(expressão e2 0 + 5)
(possui-objetivo e3 simplificado)
(expressão e3 0 * 2)

Então o Rete resultante seria:

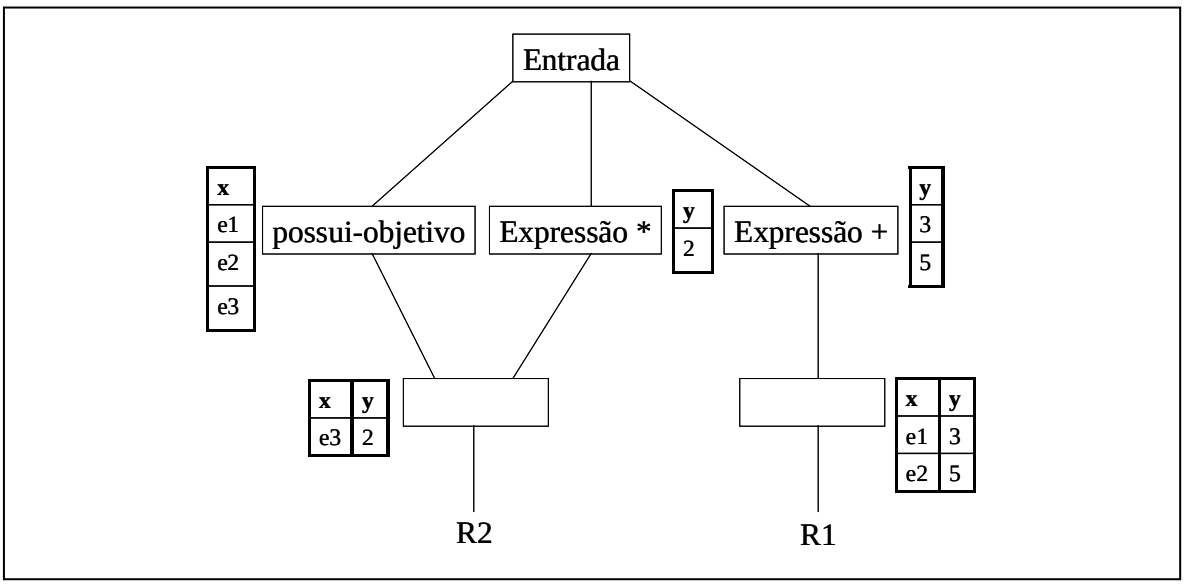

Figura 4.3: Exemplo de Rete

A ordem dos padrões, entretanto, no lado esquerdo das regras podem afetar os requisitos de armazenamento para o algoritmo Rete. Isso não é afetado somente no Rete, pois outras implementações de sistemas baseados em regras são afetados de maneira similar por conta da ordem das variáveis.

4.6 Demais Mecanismos de Inferência

Os demais mecanismos que podem ser utilizados como mecanismos de inferência são algoritmos que surgiram como métodos de busca para resolução de problemas e que podem ser utilizados para inferir determinadas ações. Em geral, estas técnicas são utilizadas para encontrar uma seqüência de ações, devendo levar de um estado inicial para um ou diversos estados objetivo. A seguir será mostrado um resumo destes métodos.

4.6.1 Análise de Significado Final (*Means-ends-Analysis)*

A técnica de inferência denominada de *means-ends-analysis* permite tanto a busca regressiva como progressiva. O processo se dá através da solução da partes principais do problema inicialmente e então retorna para os problemas menores, quando monta a solução final.

O GPS (General Problem Solver), ou solucionador geral de problemas foi o primeiro programa da IA a explorar a técnica de *meas-ends-analysis* (MARSHALL, 1997).

Um algoritmo que utilize *means-ends-analysis*, de maneira resumida realiza os seguintes passos (MARSHALL, 1997):

1. Até que o objetivo seja alcançado ou nenhum procedimento esteja disponível:

- Descrever o estado corrente, o estado objetivo e as diferenças entre os dois.
- Utilizar essa diferença para descrever o procedimento que irá, caso seja realizado com sucesso, chegar mais próximo do objetivo.
- Utilizar a procedimento e atualizar o estado corrente.

2. Se o objetivo é alcançado então retornar sucesso, caso contrário retornar falha.

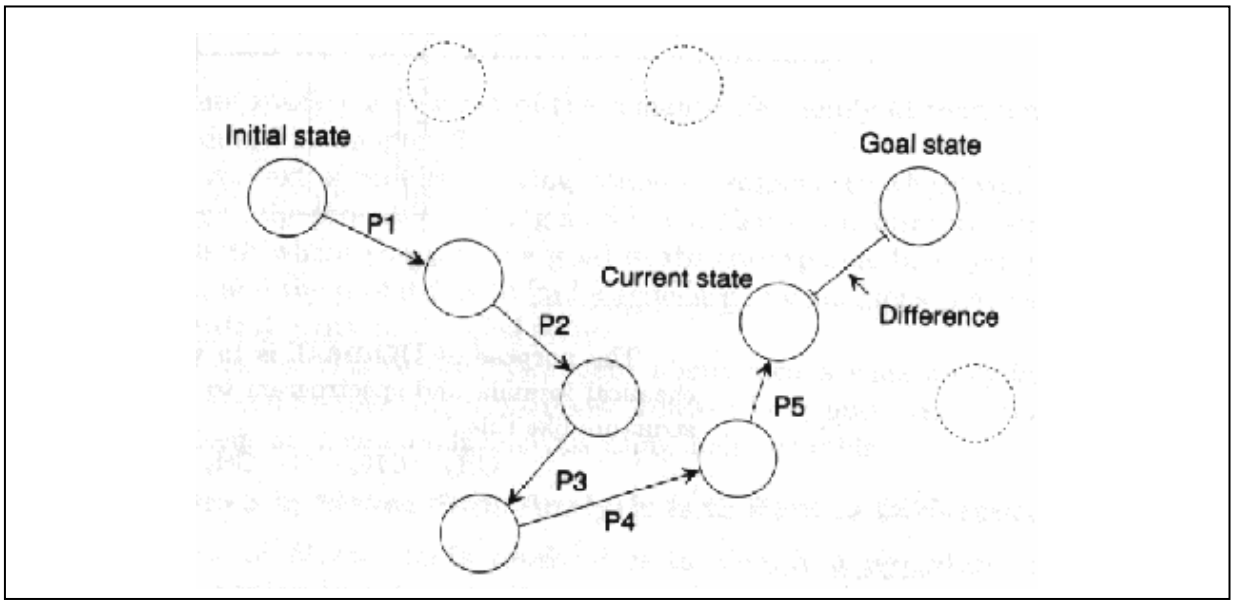

Figura 4.4: Análise de significado final e caminho através do espaço de estados (MARSHALL, 1997)

Como mostrado na Figura 4.5, o estado corrente, o estado objetivo, e uma descrição da sua diferença determina qual procedimento deve ser adotado em seguida.

4.6.2 Satisfação de Restrições(*Constraint Satisfaction CSP*[2])

O problema geral está em encontrar uma solução que satisfaça um conjunto de restrições ou requisitos. Heurísticas então não são utilizadas para estimar a distância até o objetivo, mas para decidir que passo expandir em seguida (MARSHALL, 1997).

[2] A sigla CSP é usada para descrever tanto ”Constraint Satisfaction Programming”, descrita nesta seção, como também para descrever “Constraint Satisfaction Problems”, uma classe de problemas.

O algoritmo se resume aos seguintes passos (MARSHALL, 1997):

1. Propagar todos os requisitos disponíveis:
    - Abrir todos os objetos a que devem ser atribuídos valores em uma solução completa.
    - Repetir até a inconsistência ou todos os objetos serem atribuídos valores válidos:
        i. Selecionar um objeto e avaliar tanto quanto possível o conjunto de requisitos que se aplicam ao objeto.
        ii. Se o conjunto de requisitos diferem de um conjunto prévio então abra todos objetos que compartilham algum desses requisitos.
        iii. Remover o objeto selecionado
2. Se a união dos requisitos descobertos anteriormente define uma solução retornar a mesma.
3. Se a união dos requisitos descobertos anteriormente define uma contradição retornar falha.

4.7 Resumo do Capítulo

Neste capítulo são descritos os principais métodos de inferência encontrados. Em destaque estão o encadeamento progressivo e regressivo com suas diferenças. Muitos dos demais métodos possuem processamento semelhante a estes dois métodos de inferência. Estas descrições aqui apresentadas podem servir de ponto de partida também para trabalhos futuros com este métodos

O algoritmo Rete descrito é hoje um dos mais utilizados para casamento de padrões em motores de inferência. Sua eficiência está na performance de processamento independente do tamanho da base de regras. A seguir veremos detalhes da implementação do Expert-Rete.

5. IMPLEMENTAÇÃO DO EXPERT-RETE

A seguir são vistos aspectos utilizados para a construção do Expert-Rete, motor de inferência adaptado ao Expert-Coop++ utilizando como base o algoritmo Rete para realizar inferência em tempo real.

O ciclo de vida escolhido para o desenvolvimento do Expert-Rete foi o de prototipação evolutiva (PRESSMAN, 1998). A prototipação evolutiva se utiliza de protótipos que permitem a validação dos requisitos iniciais e acrescenta novas características ao longo do desenvolvimento. A grande vantagem da utilização deste modelo de prototipagem é a garantia de um produto estável e de qualidade controlada, uma vez que os requisitos são avaliados ao final de cada etapa.

O motor de inferência Expert-Rete se utilizou do compilador g++ para a linguagem C++ na plataforma Linux.

5.1. Biblioteca STL

O “coração” da biblioteca padrão C++, a parte que influenciou toda sua arquitetura é a STL (*Standard Template Library*) (JOSUTTIS, 1999). A STL é uma biblioteca genérica que fornece soluções para gerenciar coleções de dados com algoritmos eficientes.

A STL pode ser classificada de acordo com suas estruturas:

- *Containeres*: utilizados para gerenciar coleções de objetos de um certo tipo. Cada tipo de container possui suas próprias vantagens e desvantagens. Os

containeres podem ser implementados como vetores ou listas encadeadas, ou podem ter uma chave especial para cada elemento.

- *Iterators*: utilizados para acessar elementos de coleções de objetos. Estas coleções podem ser containeres ou subconjuntos de containeres. A maior vantagem dos *iterators* é a de que os mesmos oferecem uma interface comum para qualquer tipo de container.
- Algoritmos: utilizados para processar os elementos de coleções. Os algoritmos podem realizar buscas, ordenação, modificação, ou simplesmente utilizar elementos para diferentes propósitos. Algoritmos utilizam *iterators*, entretanto os mesmos devem ser escritos somente uma vez para operar com containeres arbitrários porque a interface *iterator* para *iterators* é comum a todos os tipos de containeres.

O passo inicial e muito importante para um projeto que vai utilizar a biblioteca STL é a escolha dos containeres adequados, já que cada container possui características muito peculiares e que vão determinar a eficiência da solução.

O conceito da STL é baseado (JOSUTTIS, 1999) na separação de dados e operações. Os dados são gerenciados por classes do tipo containeres, e as operações são definidas por algoritmos configuráveis. Os *Iterators* são utilizados para unir esses dois últimos componentes. De certo modo, o conceito da STL se contrapõe à idéia original de programação orientada a objeto: a STL separa dados de algoritmos ao invés de combiná-los. Entretanto, a razão para isso é muito importante. Em princípio, pode-se combinar qualquer tipo de container com qualquer tipo de algoritmo, então o resultado é que temos um *framework* extremamente flexível e simples.

São exemplos da utilização da STL:

- *vector<Pessoa> vec1* – um vetor de objetos da classe “Pessoa”;

- *vector<Pessoa>::iterator iter* – um iterator para vetores de objetos da classe “Pessoa”;
- *reverse(vec1.begin(), vec1.end())* – utilização dos algoritmos na STL.

Outro aspecto fundamental da STL é que todos os componentes trabalham com tipos primitivos arbitrários. Como o próprio nome já indica ("standard template library" ou biblioteca de *templates* padrão), todos os componentes são modelos para qualquer tipo.

Quanto aos *containeres* existem dois tipos principais, a saber:

- *Containeres* seqüenciais que são coleções ordenadas nas quais cada elemento possui uma certa posição. Esta posição depende do tempo e lugar inserido, mas é independente do valor do elemento. Por exemplo, se vamos colocar seis elementos em uma coleção, inserindo cada elemento no final da coleção atual, estes elementos estarão exatamente na mesma ordem em que foram colocados. A STL contém três classes containeres predefinidos: *vector*, *deque* e *list*.
- *Containeres* associativos são coleções ordenadas nas quais a posição atual de um elemento depende do seu valor em relação a algum critério de ordenação. Se vamos colocar seis elementos em uma coleção, sua ordenação vai depender do seu valor. A ordem de inserção neste caso não importa. A STL contém quatro classes de containeres associativos: *set*, *multiset*, *map* e *multimap*.

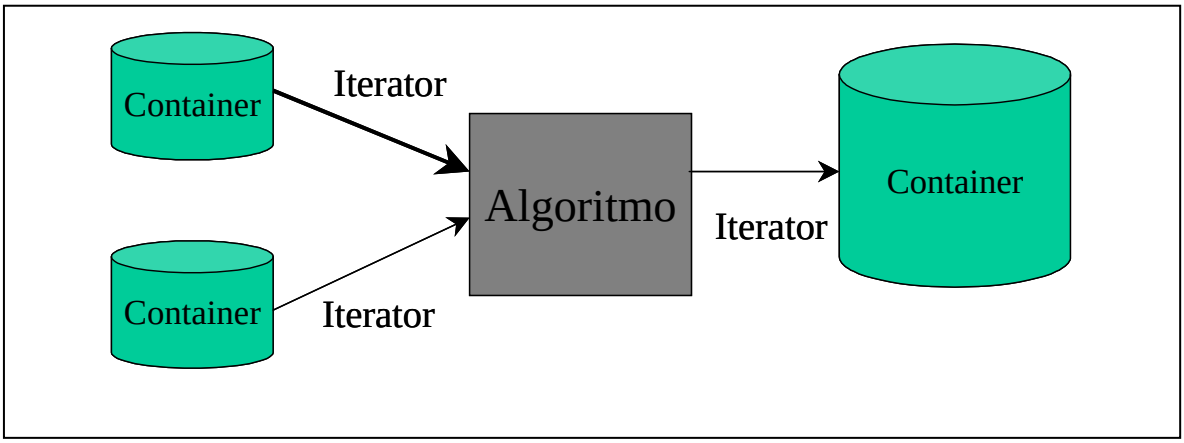

Figura 5.1: Interações entre os elementos da STL (JOSUTTIS, 1999)

5.2. STL e Expert-Coop++

A biblioteca Expert-Coop++ utiliza em muitas classes a biblioteca STL. No caso do motor de inferência já desenvolvido baseado no encadeamento progressivo, a utilização se dá através principalmente através dos containeres seqüenciais.

Estes *containeres* seqüenciais são utilizados no motor de inferência para guardar os elementos da base de regras e fatos. Estes containeres são utilizados para realizar as interações necessárias no encadeamento progressivo a fim de manipular as representações de conhecimento guardadas e assim disparar regras e inferir novos fatos.

O Expert-Coop++ se destaca pela utilização do *container* seqüencial *list*. Um *container list* é implementado como uma lista duplamente encadeada de elementos. Isto significa que cada elemento em uma lista possui seu próprio segmento de memória e se refere a seu predecessor e seu sucessor. As *lists* não permitem acesso aleatório. Por exemplo, para acessar o décimo elemento, você deve navegar pelos nove primeiros elementos seguindo os encadeamentos. Entretanto, seguir um passo para o elemento seguinte ou posterior é sempre possível em tempo constante. Além disso, acessos gerais a elementos arbitrários podem tomar

tempo (a distância média é proporcional ao número de elementos). Isto é bem pior do que tempo constante de acesso utilizado pelos *vectors* e *deques*.

A vantagem principal do container *list* é que a inserção ou remoção de um elemento é rápida em qualquer posição. Somente os encadeamentos devem ser alterados. Isto implica que mover um elemento no meio de um *list* é muito rápido se comparado com a movimentação de um elemento em um *vector* ou um *deque*.

A seguir foi extraído um trecho do código do Expert-Coop++, classe *Inference_Engine* para análise:

```
        list<Substituition> Inference_Engine::Forward_Chanig(Rule _rule){
              list<Logic_Pattern> x_logic_lhs;
              list<Frame> x_frame_lhs;
              list<Message> x_msg_lhs;
              Rule x_rule;
              Logic_Pattern x_pattern;
              Frame x_frame;
              Message x_msg;
              Facts x_facts_base;
              list<Substituition>       subst_list,       final_subst_list,
x_subst_list;
              Substituition x_subst;
              x_rule = _rule;
              x_facts_base = facts_base;
              x_logic_lhs = x_rule.Logic_lhs();
              x_frame_lhs = x_rule.Frame_lhs();
              x_msg_lhs = x_rule.Message_lhs();
              final_subst_list.clear();
              while (x_logic_lhs.size() > 0) {
                 x_pattern = x_logic_lhs.front();
                 subst_list = x_facts_base.Logic_Query(x_pattern);
                 if (subst_list.size() == 0 ){
                    if(verbose) {
                       cout<<x_rule.Id()<<" Fail at Logic Query :\n";
                       x_pattern.Print();
                    }
                    return subst_list;

                 }
                 final_subst_list.insert(final_subst_list.end(),
              subst_list.begin(), subst_list.end());
                    x_logic_lhs.pop_front();
              }
        (… Continuação do método)
```

Este trecho mostra o método *Forward_Chanig* utilizado para realizar a inferência em uma regra passada como parâmetro. As operações sobre as *lists* utilizadas ocorrem da seguinte maneira:

1. Atribui-se normalmente a um *list* local o list passado como parâmetro (`x_logic_lhs = x_rule.Logic_lhs()`, no exemplo anterior);
2. Constrói-se um *loop* normalmente com *while* para enquanto o tamanho do *list* for maior que zero (`while (x_logic_lhs.size() > 0)`, no exemplo);
3. Dentro do *while* a primeira operação é pegar o primeiro elemento do *list* com o método da STL *front()* (`x_pattern = x_logic_lhs.front()`);
4. Realizam-se operações definidas no método
5. Ao final do *loop* se exclui o primeiro elemento do *list* (`x_logic_lhs.pop_front()`).

Estas operações descritas anteriormente podem levar a um acréscimo excessivo e algumas vezes desnecessário de utilização de memória, pois as estruturas internas são sempre copiadas internamente nos métodos. Este acréscimo de memória poderá desencadear problemas quando utilizado com grandes base de regras ou em quando grandes regras (regras com muitos padrões a checar) ou ainda grandes base de fatos. Por este mesmo motivo para implementações em *list*, segundo (MEYERS, 2001), o método *size()* pode consumir tempos lineares, sendo recomendado a utilização do método *empty()* em substituição ao mesmo. O método size() sempre que chamado vai percorrer toda a lista para então retornar o valor do número de elementos.

Além disso, as operações de exclusão do primeiro elemento do loop consomem implicitamente um processamento na atribuição de ponteiros ao cabeçalho do *list*. Estes cabeçalhos são utilizados em listas para indicar o início da lista encadeada. Neste tipo de operação só é preciso ler seqüencialmente o *list* e não realizar tais operações.

5.3. STL e o Expert-Rete

Assim como o motor de inferência do Expert-Coop++ utiliza a biblioteca através principalmente do container *list*, o Expert-Rete também utiliza a STL só que se procurou utilizar para as novas estruturas internas não existentes anteriormente o *container vector*.

Os *vectors* também são *containeres* seqüenciais, com a diferença de que implementam vetores dinâmicos. Os *vectors* copiam seus elementos dentro de seu vetor dinâmico interno. Os elementos sempre possuem uma ordem e os *vectors* são um tipo de coleção ordenada. Os *vectors* possuem a capacidade de fornecer acesso aleatório, então é possível acessar cada elemento diretamente em tempo determinado e constante se soubermos sua posição.

Vectors proporcionam boa performance (MEYERS, 2001) se inserirmos novos elementos no final do mesmo ou apagarmos elementos. Se inserir ou apagar no meio ou no início, a performance é pior. Isto porque cada elemento próximo deve ser movido para outra posição. De fato, a operador de atribuição será chamado para cada elemento seguinte.

A performance dos *vectors* se torna ainda melhor se reservarmos a memória antes de inserir os elementos, entretanto, a sua utilização no Expert-Rete se deu de maneira dinâmica. A característica dos *vectors* que foi fundamental foi a de poder acessar aleatoriamente os elementos. Desta maneira os *tokens* percorrem o grafo através desse acesso aleatório utilizado.

5.4 Algoritmo de inferência Expert-Rete

No Expert-Rete o algoritmo em pseudo código é o que se segue, considerando que a base de fatos e base de regras já foram carregadas:

- Para a montagem do grafo Rete:

Enquanto houverem regras faça

Obter regra seguinte

Obter lado esquerdo da regra

Para cada padrão escrito em um formalismo faça

Alocar padrão em nó de uma entrada auxiliar

Se existir um nó já inserido no grafo com o mesmo padrão do nó auxiliar

Nó de uma entrada auxiliar é descartado

Nó de uma entrada encontrado no grafo é ligado a nó de multipla entrada corrente

Senão

Ligar nó de uma entrada a nó de múltipla entrada

Se existir nó de múltipla entrada com os mesmos pais do nó de múltipla entrada corrente

Descartar nó de múltipla entrada corrente

Ligar nó de múltipla entrada encontrado a novo nó de múltipla entrada

Se houver novo nó de uma entrada criado

Inserir nó de uma entrada no grafo

Se houver novo nó de múltipla entrada criado

Inserir nó de múltipla entrada no grafo

Fim - Para cada padrão escrito em um formalismo faça

Obter nó múltipla entrada corrente e ligar a Novo nó terminal

Inserir nó terminal no grafo

Fim - Enquanto houverem regras faça

- Para a realização da inferência:

Enquanto (houverem regras a serem disparadas)

ou (limitação no número de ciclos de inferência tenha sido alcançada)

ou (se nenhuma regra tenha sido aneriormente disparada)

Fatos ainda não utilizados são colocados em Base de fatos auxiliar para ciclo

Para cada padrão escrito em um formalismo na base de fatos auxiliar para ciclo

(ciclo de inferência)

Criar token com formalismo corrente
Para cada Nó no grafo
Se Nó não está ativado
Se padrão do token é equivalente ao padrão do
nó de uma entrada
Ativar Nó de uma entrada
Passar para os Nós subordinados
Enquanto houverem nós subordinados a checar
Se for Nó Multipla entrada
Se nó subordinado já
possui o outro nó pai ativado
Ativar Nó Subordinado
Senão (Nó terminal)
Ativar nó terminal
Inclui regra no conjunto
de conflitos
Fim - enquanto
Fim -Para cada Nó no grafo
Fim- Para cada padrão escrito em um formalismo na base de fatos
Módulo de resolução de conflitos é chamado contendo conjunto atual de regras a serem disparadas
Regras são disparadas após estratégia de resolução de conflitos ter sido executada
Fatos novos são adicionada a base de fatos auxiliar para o próximo ciclo de inferência
Fim - Enquanto não houverem regras a serem disparadas ou (limitação no número de ciclos de inferência tenha sido alcançada) ou (se nenhuma regra tenha sido aneriormente disparada)

5.4. Classes Expert-Rete

O algoritmo Rete, como já vimos anteriormente, utiliza um grafo acíclico orientado, com raiz, onde os nós representam padrões e os caminhos que iniciam na raiz até as folhas

representam as pré-condições ou inferências intermediárias. Em cada nó são armazenadas informações sobre os fatos satisfeitos pelos padrões dos nós, nos caminhos que iniciam na raiz e acabem naquele nó.

Identificamos as estruturas básicas como segue na Figura logo abaixo:

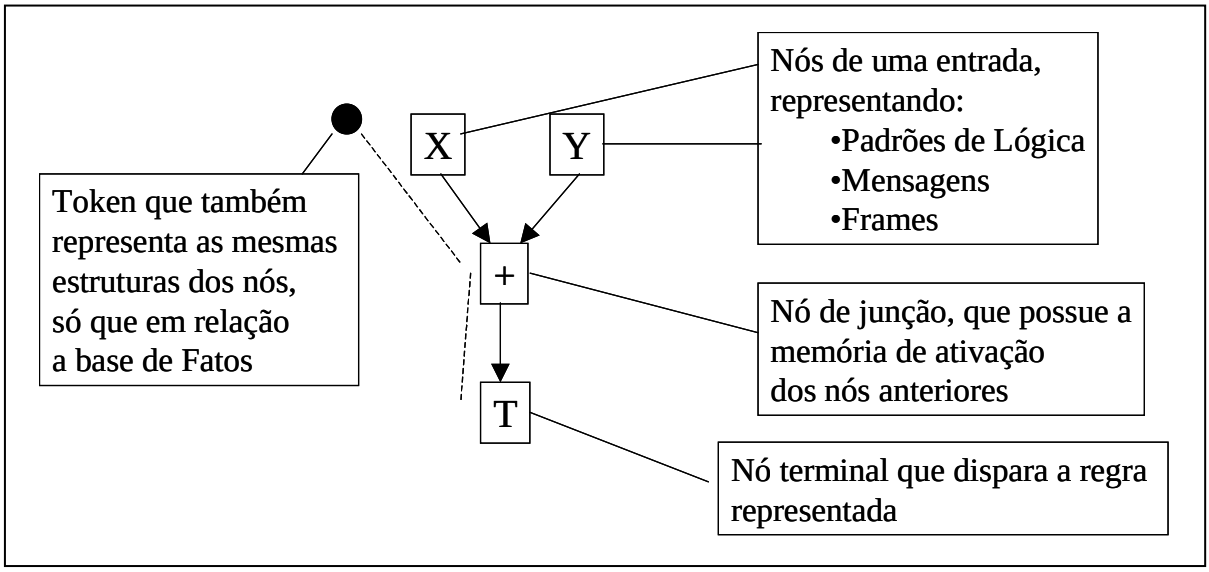

Figura 5.2: Estruturas identificadas na Rede do Algoritmo Rete.

Para implementar um motor de inferência baseado no algoritmo Rete e ainda representar todos os elementos que o Rete possui no Expert-Rete temos o modelo de classes (ver também Anexo I) com o seguinte resumo de funcionalidade a saber (todas classes compostas de um arquivo *header* .h e fonte .cc):

- **RepresentaçãoConhecimento** – classe que define a representação de conhecimento no motor de inferência tanto para os nós raiz que representam os padrões e são ligados à base de regras como para os tokens que representam a base de fatos.

  A implementação da representação do conhecimento conta ainda com a implementação de um método redefinido (operador sobrecarregado) o "**==**" para ser utilizado quando do casamento dos padrões no processo inferência e montagem do grafo.

  Esta classe contém sempre um objeto que representa um padrão conforme os formalismos *Frame*, *Message* ou ainda *Logic* de maneira exclusiva. Estes padrões já se

encontravam implementados no Expert-Coop++ e só passou por pequenas alterações para a incorporação ao Expert-Rete. Caso um outro formalismo seja habilitado para compor tanto regras como fatos, o mesmo deverá ser incorporado a esta classe.

- **No** – esta classe define de maneira abstrata os nós do Expert-Rete. Esta classe é definida como superclasse das classes: *NoUmaEntrada*, *NoMultiplaEntrada* e *NoTerminal*.

  Dentre os atributos mais importantes para esta classe se destacam o *id* (devem ser únicos para o Rete), o *idtipo* (deve ser único para cada tipo: *NoUmaEntrada*, *NoMultiplaEntrada* e *NoTerminal*) e ainda m_*sucessores* que é um vetor de *ids* de sucessores no Rete. Estes atributos são importantes na localização dos Nós interligados.

  Os nós nesta implementação representam os lado esquerdo das regras(LHS) a serem disparados, sendo que cada tipo de nó vai representar funcionalidades específicas.

- **NoUmaEntrada** – esta classe que é subclasse da classe *No* define os nós que realizam testes em fatos individuais. Estes nós são os nós raiz no grafo e portanto um objeto da classe *RepresentacaoConhecimento*.

  Temos ainda uma sobrecarga de operador nesta classe (operador "**==**") utilizada para comparar nós equivalentes. Por definição dois nós do tipo *NoUmaEntrada* são equivalentes se possuem objeto *RepresentacaoConhecimento* equivalente. Esta busca por nós equivalentes é utilizada para fins de otimização do grafo.

- **NoMultiplaEntrada** – também subclasse da classe *No* esta classe define os nós de junção ou nós que representam inferências intermediárias. No grafo, um objeto desta classe só poderá ter dois nós pais que podem ser do tipo *NoUmaEntrada* ou um outro *NoMultiplaEntrada*.

Uma sobrecarga no operador "**==**" (Redefinição do operador) também é implementada pelo mesmo motivo na classe demonstrada anteriormente com a diferença de que dois nós do tipo NoMultiplaEntrada são equivalentes se possuem pais com o mesmo *id*.

- **NoTerminal** – como já dito, também é subclasse da classe *No* e na prática define que regra será disparada com o caminho percorrido no grafo através da inferência. Por este motivo o mesmo contém objeto do tipo *Rule*.
- **Token** – esta classe define o *token* no processo de inferência segundo o algoritmo Rete. Como a classe está ligada a padrões da base de fatos, a mesma contém objeto do tipo *RepresentacaoConhecimento* utilizado para realizar o percurso no grafo até disparar uma regra no *NoTerminal*.
- **Inference_Engine** – esta classe na verdade já existe no Expert-Coop++ só que a mesma já implementava o encadeamento progressivo clássico para realizar o processo de inferência. No Expert-Rete esta classe foi adaptada e definida como superclasse de duas classes: *ExpertRete*, que implementa inferência baseando-se no algoritmo Rete e *Inference_Classic* (já implementado no Expert-Coop++), que implementa de fato o encadeamento progressivo clássico.
- **AtivacaoRegra –** esta classe define quais regras serão ativadas no processo de inferência, controlando a inserção de novos fatos com o disparo de regras. A cada ciclo de inferência a classe *ExpertRete* insere as regras a serem disparadas nesta classe para que conjuntamente com a classe *ResolucaoConflito* as trate e determine a ordem de disparo das mesmas.
- **ResolucaoConflito –** esta classe define quais regrais serão executadas em que ordem. Podem ser utilizados diversos critérios para determinar a prioridade em um ciclo de inferência. Esta classe implementou métodos de resolução de conflitos de maneira semelhante ao Expert-Coop++.

- **ExpertRete** – classe principal do motor de inferência implementado baseado no algoritmo Rete. Foram implementados métodos para montagem do grafo bem como a inferência propriamente dita.

  Por ser esta a principal classe, ela interage com quase todas classes relacionadas anteriormente direta ou indiretamente. Como já foi dito anteriormente, o *ExpertRete* é subclasse da classe *Inference_Engine*.

  Destacam-se como atributos da ExpertRete os *vector* contendo elementos da classe *No* com o nome de *Nos* que possuem todos os nós do grafo, *vector* de *NoUmaEntrada* com o nome de *m_NosRaiz* que contém os nós raiz do grafo, vector de *NoMultiplaEntrada* com o nome *m_NosME* contendo todos os nós de junção e *vector* de *NoTerminal* contendo todos os nós terminais do grafo.

  Esta classe é que faz o controle dos *Ids* dos nós através dos atributos do tipo *int m_seqNoUE* e *m_seqNoUE*. Estes *Ids* serão importantes no momento da inferência para localização dos nós e o relacionamento entre os mesmos.

  Quanto aos métodos se destacam o *MontaRete* e o *AdicionaRegra* que são utilizados para montar todo o grafo com o acesso a base de regras e cada padrão dentro de cada regra encontrado. Há também o método *inference* responsável pela inferência propriamente dita, pois é através dele que são disparados os ciclos de inferência para cada padrão encontrado na base fatos e fazer com que os tokens percorram o grafo em busca de regras a serem disparadas.

O diagrama de classes em UML simplificado com as principais classes do expert-Coop++ utilizadas se encontra na Figura 5.3.

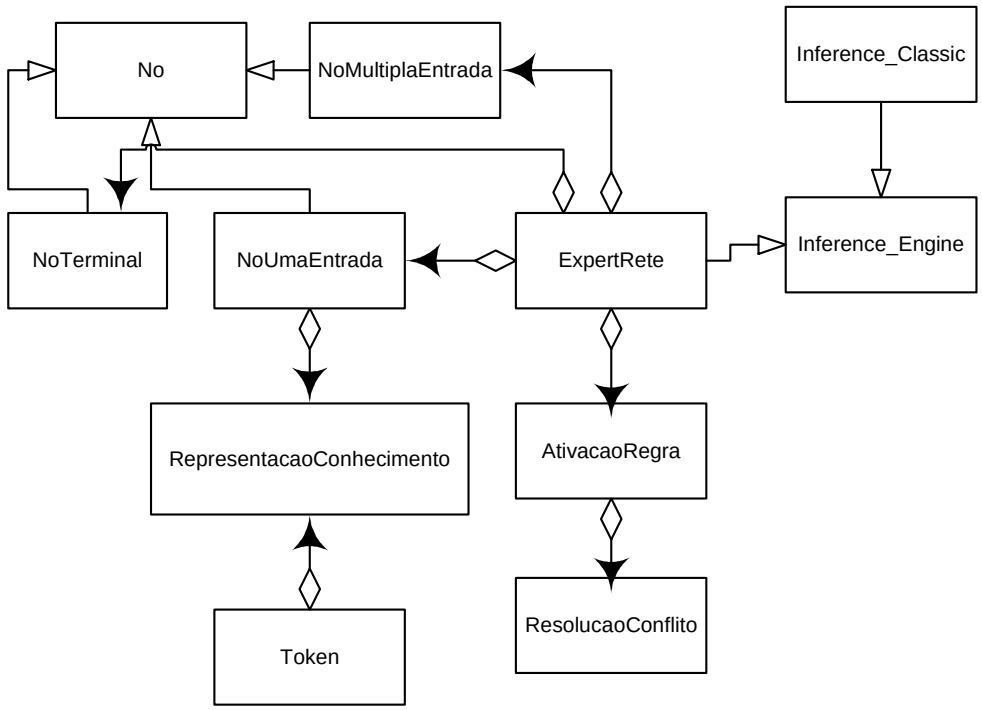

Figura 5.3: Diagrama de Classes Simplificado do Expert-Rete

5.5. Montagem da Rede no Expert-Rete

O Expert-Rete se baseia em grande parte no algoritmo Rete original (UGAI, 2003). Desta maneira, a construção do grafo se utiliza da base de regras e seus respectivos padrões.

Com a base de regras e fatos já carregada em um objeto da classe *ExpertRete* o método *MontaRete()* varre a base de Regras em busca das regras a serem utilizadas no grafo, como segue descrito no trecho abaixo deste método:

```
for(unsigned int i = 0 ; i < tamRegras ; ++i){
        //Obtem regra e chama método para adicioná-la
        AdicionaRegra(l_Regras.front());
        l_Regras.pop_front();
   }
```

No trecho anterior a variável *tamRegras* é do tipo inteiro e é atribuída ao tamanho da base de regras, ou seja, a quantidade de elementos. Observemos que para cada regra é chamado o método *AdicionaRegra* passando como parâmetro a regra corrente do *loop*.

Dentro do método *AdicionaRegra* temos o seguinte trecho de código:

```
//Obtem Lista de Padroes Logic
l_padroesLogic = regra.Logic_lhs();
...
//Instancia Novo Nó MultiplaEntrada
NoMultiplaEntrada NovoNoME(m_proxIDNoME, ProxSeqME(), m_seqRamo);
```

Neste trecho são obtidos os padrões *Logic* (*list*) da regra para uma variável local de trabalho a *l_padroesLogic* e é criado um nó do tipo *NoMultipaEntrada* que servirá para conectar os nós raiz.

Este mesmo processo de obtenção dos padrões *Logic* ocorre para os padrões em *Message* e *Frame*, sendo que para cada tipo de padrão, é chamado o método *MontaPadroesLogicRegra*, *MontaPadroesFrameRegra*, *MontaPadroesMessageRegra* para montar o grafo de acordo com os padrões encontrados na regra. Para cada tipo de padrão nesses métodos existem *Loops* que vão verificar se o Nó com o padrão recém criado já existe no grafo dependendo de como a conFiguração do grafo esteja.

É criado então o nó do tipo *NoUmaEntrada* com o identificador determinado pelo atributo *m_proxIDNoUE* , identificador para nós de uma entrada determinado pelo método da classe *ProxSeqUE()* e ramo no grafo determinado pelo atributo *m_seqRamo*. Observemos que ao se determinar o padrão como *Logic* estamos internamente conFigurando a representação de conhecimento do nó.

São realizadas algumas operações para descobrir antes de tudo se o padrão selecionado possui variáveis, se o mesmo está ligado a uma regra que possui filtros e se o grafo está conFigurado para realizar otimizações. Após isso é utilizado o método *find* da própria biblioteca STL para procurar no vetor de nós raiz por algum nó de uma entrada equivalente ao nó recentemente criado. Isto é possível através da sobrecarga do operador == realizada sobre a classe *NoUmaEntrada*. Se o objeto apontado *l_InoUE* for diferente de final de vetor então significa que outro nó equivalente ao recentemente criado foi encontrado. Desta forma, várias

operações são realizadas para fazer com que o nó encontrado na busca seja utilizado para o ramo em construção. É chamado o método *TratamentoPadraoNaoEncontrouUE* para qualquer tipo de padrão caso o nó recém criado não possua nó equivalente no grafo e *TratamentoPadraoEncontrouUE* caso o nó tenha equivalente.

Vale ressaltar que para cada nó do tipo *NoMultiplaEntrada* o mesmo deve possuir no máximo dois nós como pai e assim utilizamos o método *AdicionaNoMultiplaEntradaRete* para interligar o último nó *NoMultiplaEntrada* a um novo do mesmo tipo:

```
NovoNoME = AdicionaNoMultiplaEntradaRete(NovoNoME,true);
```

Com este trecho obtemos um novo nó ligado ao último *NoMultiplaEntrada* criado. O último parâmetro do método indica um *flag* para determinar se o nó já foi incluído ou não no grafo para que o mesmo seja atualizado.

Ao final deste loop temos o seguinte trecho:

```
//Adiciona Sucessor
NovoNoUE.AdicionaSucessor(NovoNoME);
//Adiciona no vetor de Nó Raiz
AdicionaNo(NovoNoUE);
```

Com este trecho o último nó do tipo *NoMultiplaEntrada* é colocado como sucessor do nó do tipo *NoUmaEntrada* corrente e como todas as interligações entre os nós estão atualizadas, o nó de uma entrada é finalmente adicionado no grafo. As operações entre os nós são realizadas por inteiro inicialmente para mais tarde serem adicionadas aos vetores.

5.6. Representação de Conhecimento no Expert-Rete

A representação de conhecimento no Expert-Rete é utilizada através da classe *RepresentacaoConhecimento* como já foi dito anteriormente e aplicado como objeto, atributo das classes *NoUmaEntrada* e *Token*.

Na classe *NoUmaEntrada* temos o objeto *m_RepreConhecimento* que será responsável por armazenar o padrão do nó. De forma semelhante temos na classe *Token* um objeto com o mesmo nome do nó uma entrada.

A representação de conhecimento será fundamental no nó de uma entrada para a sobrecarga do operador "**==**" no momento da construção do grafo Rete como mostrado a seguir:

```
//Operador de igualdade para comparaçao de nós
bool NoUmaEntrada::operator == (const NoUmaEntrada &n){
    RepresentacaoConhecimento Lrc = n.m_RepreConhecimento;
    if (m_RepreConhecimento == Lrc ){
          return true;
    }else
          return false;
}
```

Nesta sobrecarga de operador nota-se que dois nós são considerados equivalentes se possuem representação de conhecimento também equivalentes.

Os formalismos atualmente suportados pelo Expert-Rete são:

- Lógica (*logic*);
- Quadros (*frame*);
- Mensagem (*message*);

Estes formalismos são utilizados da mesma maneira como descrito anteriormente na seções 3.5.1, 3.5.3 e 3.7 do capítulo 3.

5.7. Processo de inferência no ExpertRete

O processo de inferência se dá com o método *Inference* na classe *ExpertRete*. Primeiramente neste método temos um *loop* para enquanto o *flag* que determina se há regras sendo disparadas for verdadeiro, como mostrado no trecho a seguir:

```
while (m_stRegraDisparada){
          m_stRegraDisparada= false;
          //Obtem itens da Base de Fatos
          l_tamFatosLogic = m_FatosLogic.size();
          l_tamFatosFrame = m_FatosFrame.size();
          l_tamFatosMessage = m_FatosMessage.size();
          //Loop para a lista de Fatos Logic
          for (int i = 0  ; i < l_tamFatosLogic; i++ ){
          //Realiza Ciclo inferência Logic dos Fatos
                //originalmente carregados
                CicloInferenciaLogic(m_FatosLogic.front());
                //Disparando Regras após Ciclo de Inferencia
                if (AtivRegra.PossuiRegras() ){
                      AtivRegra.DisparaRegra( //Parâmetros
                            m_FatosLogic,
                            m_FatosFrame,
                            m_FatosMessage,
                            m_Fatos,
                            m_NovosFatos);
                      //Atualiza Base de Fatos
                      AtualizaBaseFatos();
                }
                m_FatosLogic.pop_front();
          }
...
```

A primeira instrução dentro do loop é a atribuição da variável que controla o loop *m_stRegraDisparada* para o valor falso. Em seguida obtém-se o tamanho de elementos da lista de padrões em *logic* e demais padrões da base de fatos e é realizado um *loop* para os mesmos.

Com o método *CicloInferenciaLogic* obtém-se o primeiro elemento da lista de padrões *logic* para criar o *token* e varrer o grafo em busca de ativações de regras. Segue código deste método a seguir:

```
Token tk;
tk = Token( FatoPadraoLogic);
CicloToken(tk);
```

E conseqüente no método *CicloToken*:

```
for (int j = 0 ; j < m_tamNosRaiz ; j++){
      //Testa se Nó já foi ativado
```

```
        if ( ! m_NosRaiz(j).isAtivado()){
              //Testa se Padrão é Igual
              if (m_NosRaiz(j).processaNo(PToken)){
                    m_NosRaiz(j).setRepreFatoAtivado(
                          PToken.getUnidadeRC());
                    m_NosRaiz(j).ativaNo();
                    AtivaSucessores(m_NosRaiz(j));
              }
        }
  }
```

Neste código o *token* é criado com o padrão *logic* passado como parâmetro do método e então para cada nó raiz, ou seja, nós que representam padrões, é chamado o método do nó *processaNo* para checar se a representação de conhecimento entre o nó e o *token* são equivalentes. Se o nó e o *token* possuem esta representação de conhecimento equivalente então o nó ativado colocando o *flag* do nó (*m_ativado*) para o valor verdadeiro e é chamado o método *AtivaSucessores* passando como parâmetro o próprio nó a ser ativado para que os sucessores do nó sejam ativados na medida do possível.

O método *AtivaSucessores* é então chamado para ativar os sucessores de um nó raiz com o seguinte código:

```
    //Obtem ID dos sucessores do Nó
    vector<int> sucessores = PNo.getSucessores();
    int l_tamSucessores = sucessores.size();
        for( int i=0; i < l_tamSucessores; i++){
              int idTipoSuc = Nos(sucessores(i)).idTipo;
              NoMultiplaEntrada NoME = m_NosME(idTipoSuc);
              if (NoME.m_idPaiEsquerda == PNo.id){
                    NoME.setPaiEsquerdaAtivado();
              }
              if (NoME.m_idPaiDireita == PNo.id){
                    NoME.setPaiDireitaAtivado();
              }
              //Testa se as duas entradas estão ativas para ativar os
  sucessores
              if (
                    (NoME.getQtdNosPai() > 1 &&
                          NoME.isPaiEsquerdaAtivado() &&
                                NoME.isPaiDireitaAtivado()) ||
                     (NoME.getQtdNosPai() == 1 &&
                          NoME.isPaiEsquerdaAtivado())
              ){
                    //Ativa o Nó por inteiro
                    NoME.ativaNo();
                    AtivaSucessores(NoME);
              }
              //Atualizando no vetor
              m_NosME(idTipoSuc) =NoME;
```

```
            Nos(sucessores(i)) = NoME;
        }
```

Deste modo o método primeiro obtém o vetor de *Ids* dos nós sucessores do nó passado como parâmetro e é realizado um *loop* para interagir com estes sucessores a fim de ativar o nó do tipo *NoMultiplaEntrada* para o pai na esquerda ou direita (nós de múltipla entrada só podem possuir dois nós pai). Caso os dois nós pai estejam ativados, o nó é completamente ativado (flag *m_ativado* da classe *No*) e o método *AtivaSucessores* é chamado novamente passando como parâmetro o nó de múltipla entrada.

Neste método com o mesmo nome do anterior, mas com a assinatura diferente (é passado como parâmetro um nó do tipo *NoMultiplaEntrada*). Se o nó seguinte for de multipla entrada o procedimento é igual ao do método anterior, entretanto se tivermos um nó terminal, temos o seguinte trecho de código, internamente no *loop* que varre os nós sucessores:

```
if (Nos(sucessores(i)).getTipoNo() == 3){
    int idTipoSuc = Nos(sucessores(i)).idTipo;
    Nos(sucessores(i)).ativaNo();
    m_NosTE(idTipoSuc).ativaNo();
    //Adiciona Regra para ser disparada posteriormente
    AdicionaRegra(m_NosTE(idTipoSuc));
    m_stRegraDisparada = true;
}else{
    //É um NoMultiplaEntrada
    ...Continuação do método
```

Desta maneira o nó terminal é ativado da mesma forma que os nós anteriores e a regra é adicionada ao objeto *AtivRegra* da classe *AtivacaoRegra*. A variável de controle *m_stRegraDisparada* é ainda colocado para verdadeiro para que o *loop* no método *Inference* continue a operar.

Voltando ao método *Inference*, após um ciclo de inferência para um determinado padrão as regras sensibilizadas são disparadas através do método *DisparaRegra* é chamado passando alguns parâmetros para que o objeto da classe *AtivacaoRegra* operando conjuntamente com a classe *ResolucaoConflito* ordene as regras a serem disparadas.

Definimos como sendo um ciclo de inferência para um padrão (*logic*, *frame* ou *message*) como sendo a busca de um *token* para este padrão sobre todo o grafo em busca de regras que vejam ser satisfeitas.

Com o disparo das regras novos fatos são gerados com o lado direito das regras sendo disparados gerando novos *tokens* para efetuar o mesmo processo de inferência.

5.8. Resolução de Conflitos

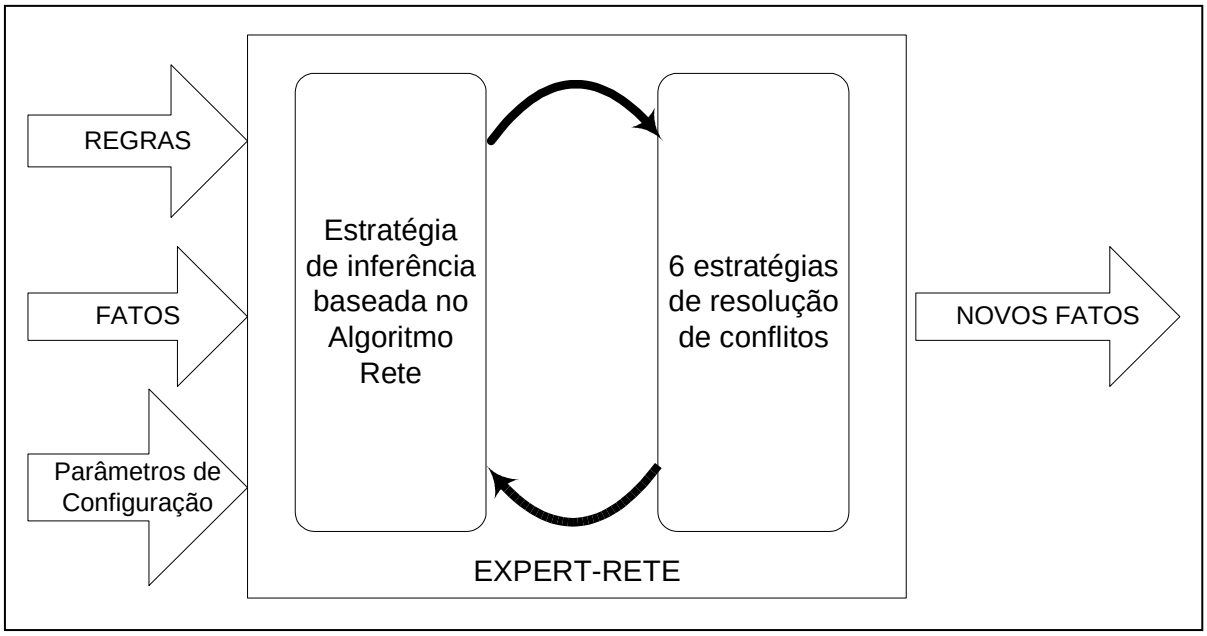

Figura 5.4: Arquitetura Básica do Expert-Rete

O Expert-Rete trabalha com uma classe para resolução de conflitos, sendo que são disponibilizadas 6 estratégias para resolução de conflitos, conforme descrito a seguir:

- A primeira estratégia consistem de realizar uma média aritmética dos tempos lógicos dos fatos que ativaram o lado esquerdo das regras, em cada regra do conjunto de conflitos. Com isso as regras que possuem as menores médias aritméticas são disparadas antes das demais. Esta estratégia depende, portanto, dos tempos lógicos dos fatos;

- A segunda estratégia é utilizada através da quantidade de padrões na existentes no lado esquerdo das regras. Quanto maior o número de padrões, maior a prioridade de disparo da regra no conjunto de conflitos.
- A terceira estratégia é o inverso da anterior, com a prioridade sendo determinada também pela quantidade de padrões no lado esquerdo, sendo que quanto menor for a quantidade maior a prioridade de execução.
- A quarta estratégia é utilizada através de identificadores de prioridade que devem ser explicitamente conFigurados nas regras antes da montagem do grafo. Quanto maior for a prioridade conFigurada, maior a sua prioridade de execução.
- A estratégia seguinte contabiliza a quantidade de padrões do lado esquerdo, lado direito, somado à quantidade de itens de filtro existentes na regra para efetuar a comparação. Quanto menor esta soma der, maior a prioridade de execução frente às outras regras, ou seja, maior a simplicidade, maior a prioridade de execução.
- A última estratégia que é o inverso da anterior, ou seja somadas as quantidades de padrões do lado esquerdo, lado direito e itens de filtro, quanto maior o valor, maior a prioridade de execução. Desta forma, as regras mais complexas são disparadas inicialmente frente às demais.

5.9. Otimizações do Algoritmo Rete

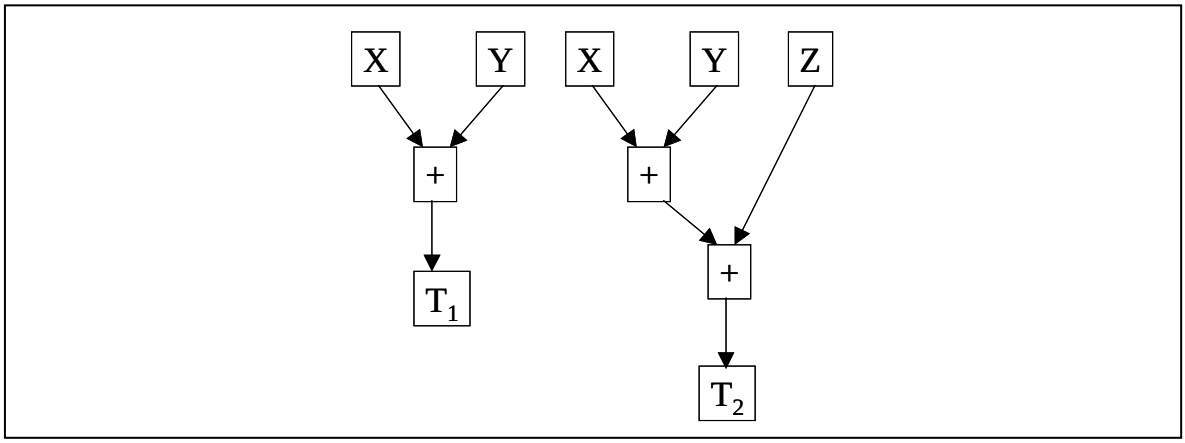

Figura 5.5: Uma rede não otimizada

O Expert-Rete promove contribuições ao algoritmo Rete conforme diversas ferramentas já utilizam. O objetivo é otimizar tanto a utilização de memória que o algoritmo utiliza como diminuir o espaço de busca para realizar o processo de inferência.

A primeira otimização diz respeito aos nós de uma entrada, na montagem do grafo, fazendo com que se duas regras por exemplo possuam algum padrão em comum, as mesmas vão compartilhar nós de uma entrada, como mostrado na Figura comparativa abaixo para duas regras:

Regra 1 – X ^ Y => RHS1

Regra 1 – X ^ Y ^ Z => RHS2

Com a otimização obtém-se o grafo mostrado na Figura 5.6.

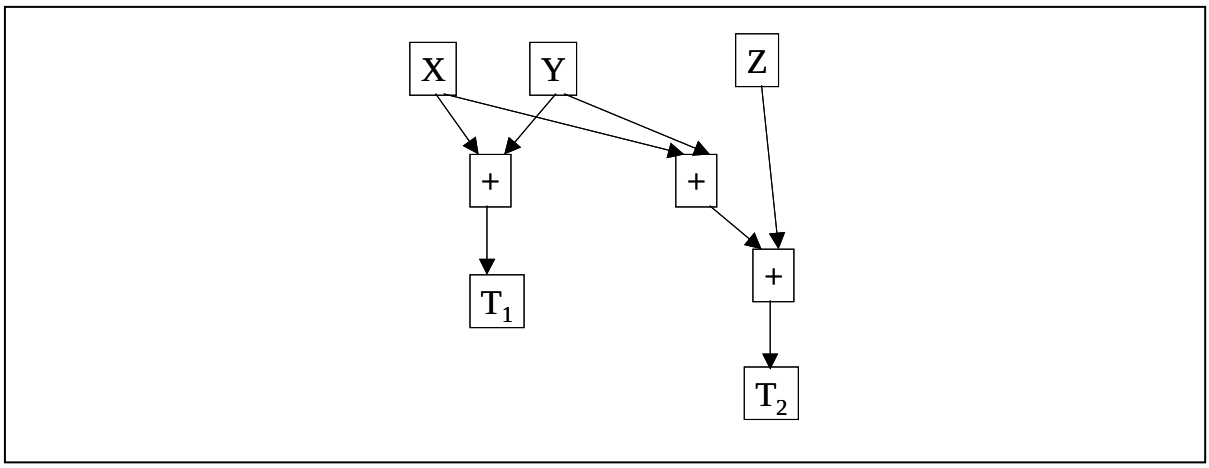

Figura 5.6: Rete otimizado quanto aos nós de uma entrada

Desta forma, os nós raiz do grafo, ou nós que representam padrões de regra, são compartilhados reduzindo na quantidade dos mesmos. Quando da realização de inferência em grandes base de regras, tanto o espaço de busca reduzido quanto a utilização de memória ser otimizada fará diferença em relação ao modelo anterior.

A segunda otimização proposta tem objetivo semelhante à proposta anterior só que em relação aos nós de junção (*NoMultiplaEntrada*). Com o compartilhamento teríamos o grafo mostrado na Figura 5.7.

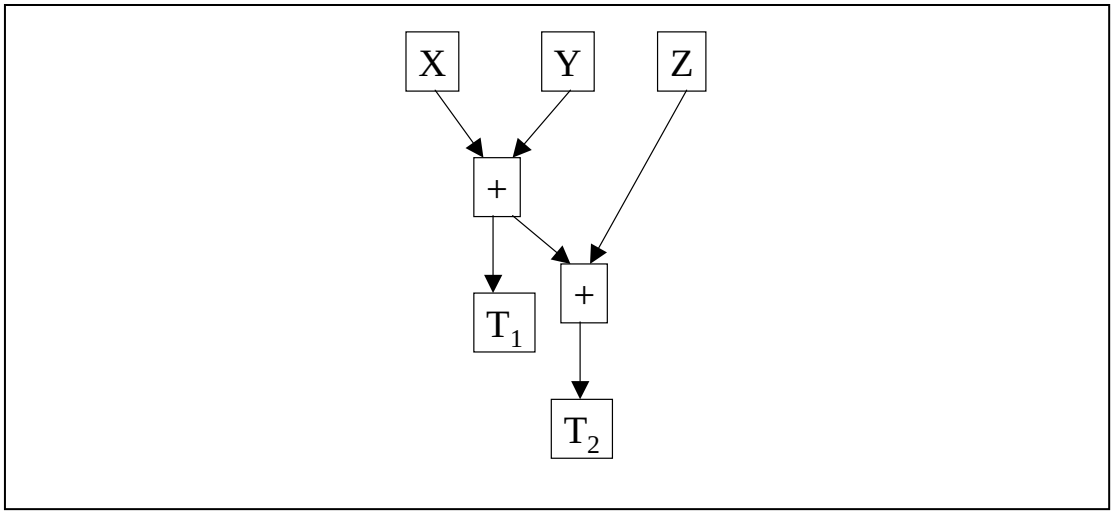

Figura 5.7: Rete otimizado em relação aos nós raiz e nós de junção

Para realizar estas duas otimizações foram necessárias as sobrecargas do operador == tanto para os nós do tipo *NoUmaEntrada* como para os nós do tipo *NoMultiplaEntrada* como já foi dito na seção de explicação de montagem do grafo.

Saímos no exemplo citado com um grafo contendo 10 nós para um outro otimizado contendo 7 nós. Vale ressaltar que com essa otimização a performance para a montagem do Rete ficará prejudicada, pois são necessárias pesquisas a cada inserção de um novo nó para verificar a existência prévia dos mesmos. Por outro lado, a inferência propriamente dita ganha em eficiência, com um espaço de busca mais reduzido e otimizado.

A ferramenta JESS já descrita anteriormente também utiliza o algoritmo Rete como motor de inferência e também implementa a otimização dos nós no grafo de maneira semelhante ao Expert-Rete. O JESS, entretanto, possui mais de 10 classes para representação dos nós no grafo com diversos tipos próprios, fornecendo inclusive operações sobre os nós para o usuário (inserção e retirada de nós). As representações de conhecimento utilizadas no JESS também são bem diferentes das encontradas no Expert-Rete.

O Expert-Rete possui, entretanto, uma implementação menos complexa e totalmente diferente da implementação do JESS, consumindo menos memória, pois no Expert-Rete as interligações entre os nós são representadas através de referências aos identificadores e no JESS são utilizados os objetos inteiros.

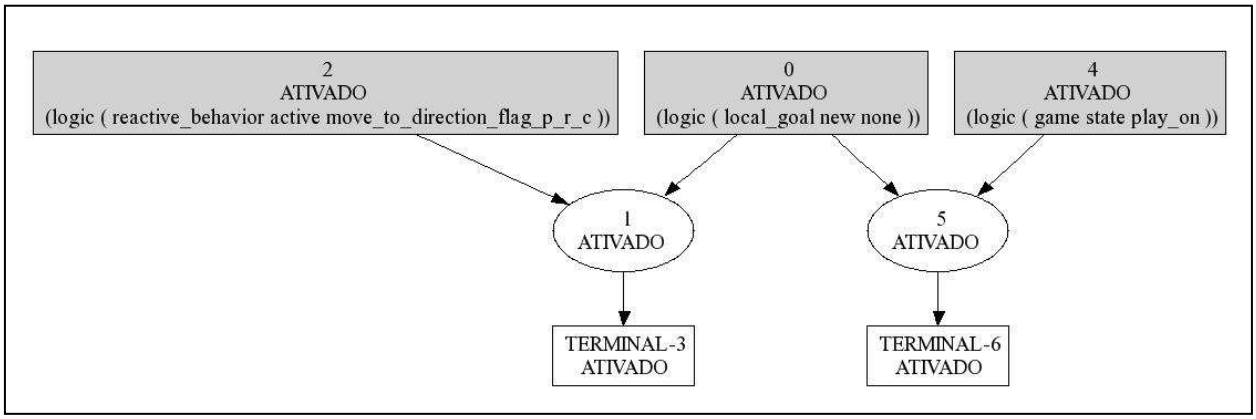

Figura 5.8: Grafo gerado através da ferramenta GraphViz em base de regras do Expert-Rete

5.10 Resumo do Capítulo

Neste capítulo foram descritos detalhes da implementação do Expert-Rete, incluindo detalhes importantes das estruturas de dados utilizados através da biblioteca STL da

linguagem C++. Além disso, foi descrito o modelo de classes do motor de inferência proposto, métodos importantes da classe principal *ExpertRete*, a forma como as representações de conhecimento foram tratadas na implementação.

O objetivo da implementação foi o de simplificar a solução através dos conceitos do algoritmo Rete original, procurando ao mesmo tempo otimizar a utilização de memória e o espaço de busca, de modo a tornar o processo de inferência em si, mais eficiente que o encadeamento progressivo puro.

6. RESULTADOS

A seguir serão mostrados os resultados obtidos através da execução do processo de inferência isoladamente e o processo de execução de um agente no Expert-Coop++. Estas duas formas foram executadas com o motor de inferência já implementado, denominado de método clássico ou encadeamento progressivo, comparando-se com o Expert-Rete, motor de inferência implementado baseado no algoritmo *Rete*.

Para analisar o comportamento na máquina foi utilizado o *KDE System Guard* e a ferramenta *top*, aplicativo instalado juntamente com o ambiente gráfico KDE. Foi utilizado um Pentium IV 2.2 Ghz 256 MB RAM sobre o sistema operacional *SuSE* na versão 9.0.
Como já foi dito anteriormente para verificar as formações dos grafos foi utilizado o pacote *GraphViz* para *Linux*. Para a construção do grafo foi utilizada a ferramenta *dot* (grafos direcionados) (Gansner, Koutsofios and North 2002) deste pacote. Deste modo, ao final da inferência a classe *ExpertRete* exporta informações necessárias para visualização do grafo (pode ser em *jpeg*, *png*, *ps* ou *pdf*), mostrando inclusive os nós ativados.

Os parâmetros da análise foram o de quantidade de memória alocada para o processo, e tempo de execução. Este tempo de execução foi obtido através dos objetos do tipo *clock_t* que são comumente utilizados para se obter tempo de uso do processador para o processo. Desta forma o procedimento adotado para se encontrar o tempo mais preciso alocado a cada procedimento ocorreu segundo o código:

```
clock_t tempo_inicial, tempo_final;
double duracao;
tempo_inicial = clock();
```

```
(procedimentos...)
tempo_final = clock();
duracao = (double) (tempo_inicial – tempo_final)/ CLOCKS_PER_SEC;
```

Desta forma a variável *duracao* guardará o tempo de execução de qualquer procedimento com precisão de centésimos de segundo.

Para a realização dos testes em termos de conFiguração de base de regras e fatos, primeiramente foi executado em ambiente contendo 14 padrões nos fatos e apenas uma regra, em seguida com uma base de regras de 50 regras, 100 regras, 150 regras, 341 regras para um único ciclo de inferência. Mais tarde foram utilizadas bases de regras contendo 347, 350 e 354 regras em 10,13 e 17 ciclos de inferência respectivamente, para os testes em múltiplos ciclos. Todos os testes realizados foram observados nas mesmas condições tanto para o motor de inferência clássico como para o Expert-Rete.

6.1 Motor de Inferência Método Clássico (Encadeamento Progressivo)

Para obter estes resultados foram utilizados somente as classes do Expert-Coop++ que são necessárias para o motor de inferência a ser executado, dentre elas estão: *Rule*, *Rules_Base*, *Frame*, *Logic_Pattern*, *Message*, *Filter_Operation*, *Substituition*, *Visual_Obj*, *Subs_Pair*, *Plan*, *Plan_Set*, *Task*, *Logic_Clock*, *Facts*, *Filter* e *Inference_Engine*.

Esta forma de execução se tornou interessante, pois o método de inferência foi isolado para somente se comparar a performance no mesmo sem se preocupar com demais aspectos tratados nas demais classes da solução.

Na primeira fase, foi realizado um teste somente com a conFiguração de um ciclo. O teste verificou a utilização de memória, processamento e o tempo de duração para determinadas base de regras. Em todas as bases de dados o teste demonstrou baixo consumo de memória (no máximo 0,9% do sistema). O consumo de processamento só alcançou picos de 87% de utilização sobre o total.Vale lembrar que o código original sofreu uma pequena

alteração para efetuar a inferência enquanto regras forem disparadas. Analisando-se o tempo de duração para a realização da inferência de acordo com o número de regras, o comportamento é o que é mostrado no gráfico abaixo.

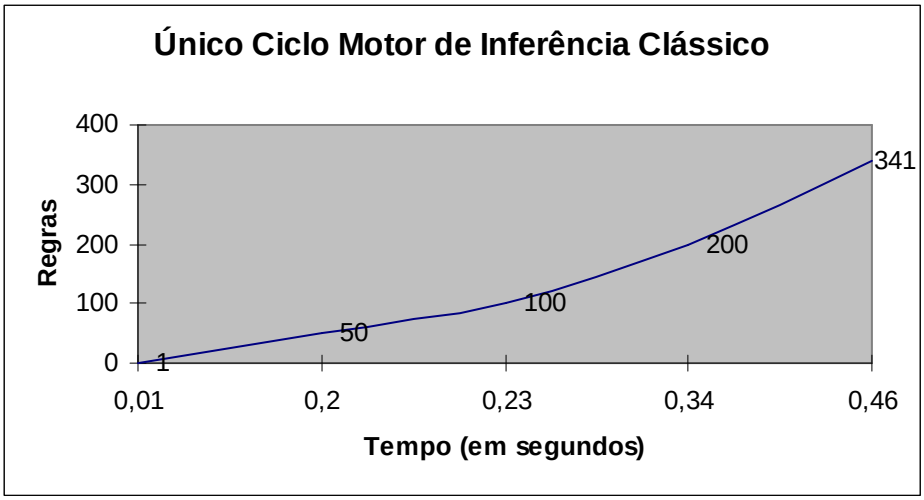

Figura 6.1: Gráfico de execução para um Ciclo no motor de inferência Clássico

Pode-se perceber que a tendência é que o motor de inferência clássico execute a inferência de maneira que quanto maior for o número de regras, maior tempo gasto para executar a inferência.

Já com múltiplos ciclos as escalas de tempo são maiores, comportamento este que já era esperado. A porcentagem de memória utilizada para o processo permaneceu inalterada e o uso de processamento teve picos de 91%, o que também já era esperado. Ver gráfico da Figura 6.2.

Interessante notar que no pior caso, com 354 regras, em 17 ciclos de inferência, o motor de inferência clássico obteve um tempo de processamento para a realização de todos os ciclos de 12,52 segundos. Significa dizer que o motor de inferência saiu de bases de regras equivalentes onde uma possuía 3 ciclos de inferência para outra de 10 ciclos, passando de 2,5 segundos para 8,8 segundos.

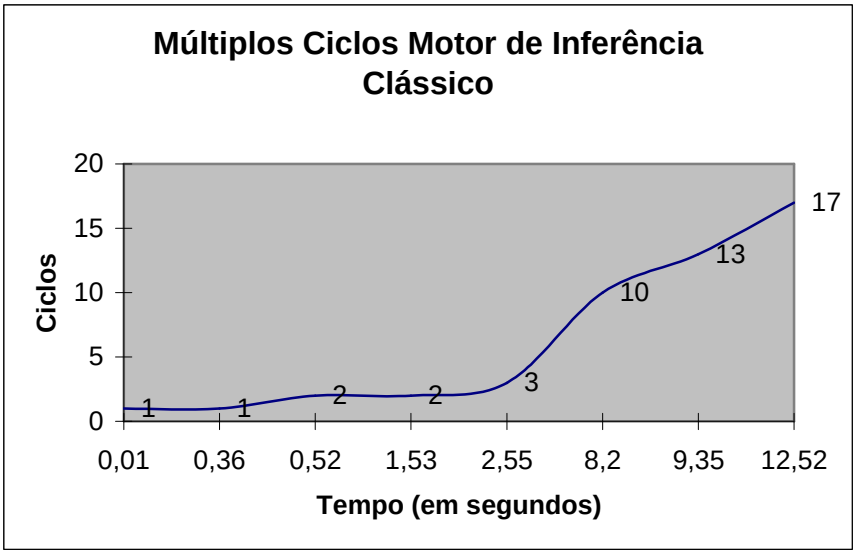

Figura 6.2: Comportamento do motor de inferência clássico para múltiplos ciclos [3]

6.2 Motor de Inferência Método Rete

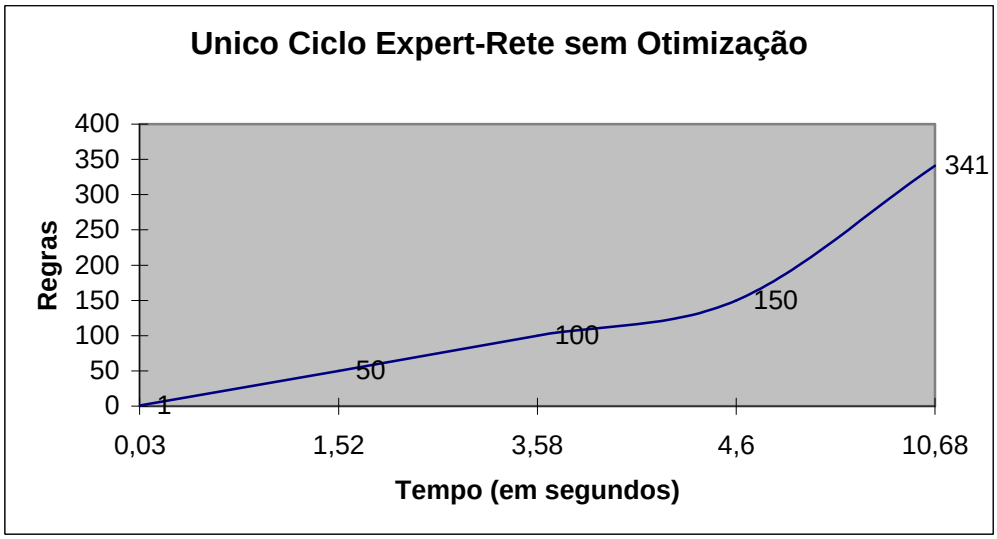

Figura 6.3: Comportamento do *Expert-Rete* em um Ciclo sem Otimização do Grafo

Para efetuar a inferência utilizando a classe *ExpertRete* foi utilizada a mesma conFiguração usada para o motor de inferência clássico. Sendo assim na primeira fase o teste foi realizado executando em apenas um ciclo de inferência sem a otimização do grafo. Desta forma foram detectados picos de processamento de até 94% com utilização de memória de até 4,5%. No Expert-Rete outra variável deve ser considerada, o tempo de duração para

[3] A repetição dos ciclos no gráfico ocorreu com o aumento no número de regras

construção do grafo. Nesta primeira fase dos testes o tempo de duração para construção do grafo foi de até 12,4 segundos (341 regras). O tempo de duração para a realização das inferências se comportou conforme o gráfico da Figura 6.3. Fica evidenciado pelo gráfico que o tempo de resposta obtido foi alto para o sistema.

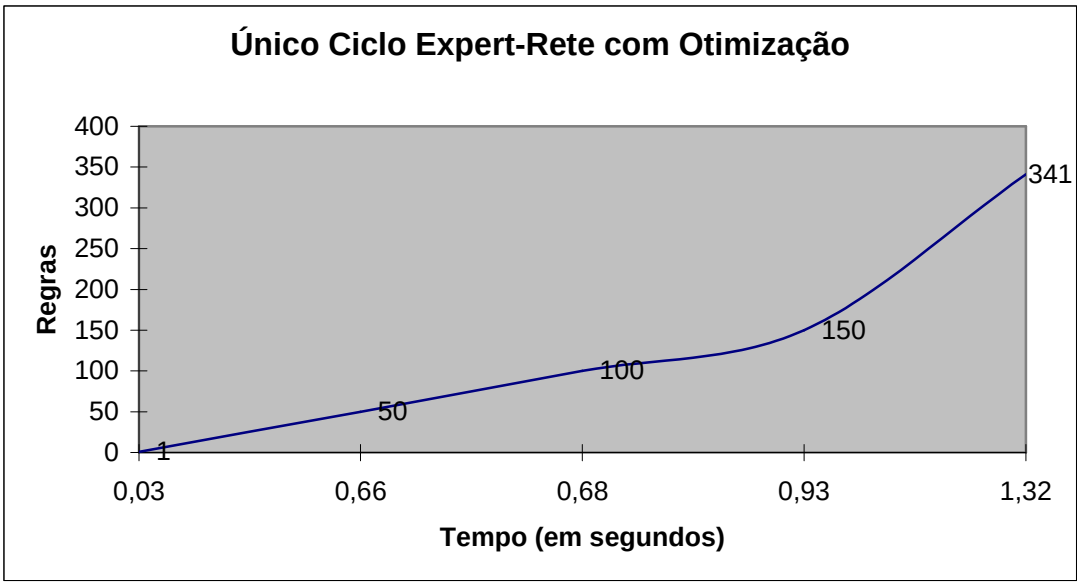

Figura 6.4: Comportamento do Expert-Rete em um Ciclo com Otimização do Grafo

Já com a conFiguração do motor de inferência para único ciclo com a otimização do grafo o percentual de uso do processador se manteve o mesmo, com um aumento no tempo de construção do grafo e redução do percentual de memória (2,1% no pior caso). Este comportamento era esperado já que com a otimização, o objetivo é reduzir o número de nós no grafo e conseqüentemente a quantidade de memória alocada. Há um aumento no tempo de duração para construção do grafo devido às diversas operações que são utilizadas para a montagem do grafo otimizado. O comportamento é o que se segue na Figura 6.4.

O gráfico da Figura 6.5 demonstra uma sensível melhora no tempo de duração para realização da inferência, pois neste caso se reduziu o espaço de busca dos nós. Nesta última fase de teste o motor de inferência foi executado utilizando-se a conFiguração de múltiplos ciclos de inferência com a otimização do grafo. Neste caso não houve variação de utilização de processamento e tempo de duração para montagem do grafo. Ficou registrado um pequeno

aumento de memória (picos de até 2,3% do sistema) e tempos de duração da inferência conforme o gráfico a seguir.

Com a análise dos 10 ciclos de inferência a Expert-Rete terminou os ciclos em 2,52 segundos. Ou seja, aumentando-se de 3 para 10 ciclos, o tempo de processamento passou de 2,1 segundos para 2,52 segundos.

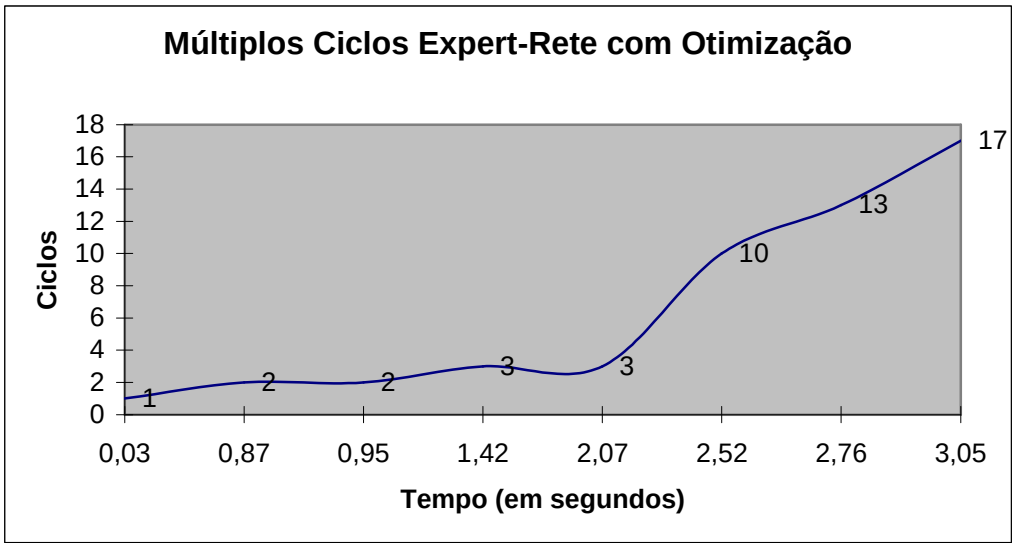

Figura 6.5: Expert-Rete em Múltiplos Ciclos com Otimização do Grafo[4]

6.3 Agente Expert-Coop++ utilizando ExpertRete.

Foi realizado um teste do Expert-Rete, utilizando a biblioteca Expert-Coop++ com o motor de inferência proposto. O Expert-Rete foi, portanto, incorporado à biblioteca Expert-Coop++ se tornando mais uma opção de motor de inferência na implementação de agentes.

O Expert-Rete apresenta a implementação do método *isReteMontado* na classe *ExpertRete* para retornar verdadeiro se o grafo já tiver sido montado.

Na biblioteca *Expert-Coop++* foi incluído um diretório nos fontes denominado *Expert-Rete* contendo todas as classes próprias deste motor de inferência.

[4] A repetição dos ciclos no gráfico ocorreu com o aumento no número de regras

Com isso só foi alterado o código do processo *Expert* (nível cognitivo) conforme o seguinte trecho de código:

```
ExpertRete inference_engine;
(continuação do código...)
if( msg.to == "Expert" ) {
                local_base.Update_Logic_Pattern( msg );
                inference_engine.Load_Facts(local_base);
      if (inference_engine.isReteMontado()){
            inference_engine.Load_Rules(
                  cognitive_rules_base.Rules_List());
      }else{
            inference_engine.ResetRete();
      }
      local_base = inference_engine.Inference(
                        logic_clock.get_timestamp());
      export_msg = local_base.Output_Messages();
 }
```

Como a base de regras do *Expert-Coop++* pode não sofrer alteração da execução do agente, não há, muitas vezes, a necessidade do processo *Expert* carregar várias vezes a base de regras selecionada, economizando tempo de montagem do grafo. A idéia foi a de que o agente ganhe o tempo do carregamento da base de regras e “chame” somente o método *ResetRete* para retirar as referências da base de fatos anterior, retirar as referências de nós no grafo que já foram sensibilizados, para daí então carregar nova base de fatos. Com o grafo todo reiniciado e com uma nova base de fatos, o processo de inferência já pode ser novamente iniciado, obtendo assim maior eficiência no motor de inferência.

6.4 Resumo do Capítulo

Este capítulo apresentou os resultados obtidos do motor de inferência proposto Expert-Rete implementado. Foi descrito o ambiente onde os testes foram realizados e os critérios estabelecidos para realizar a comparação de eficiência e performance com o motor de inferência já implementado baseado no encadeamento progressivo.

O fator analisado foi o tempo para a realização da inferência em conjunto com o número de regras e número de ciclos, para comparação da inferência em múltiplos ciclos. No Expert-Rete foi analisado ainda a otimização ou não do grafo sob esses mesmos aspectos. Vale salientar que para todas as situações foram utilizadas as mesmas bases de regras tanto para o motor de inferência clássico como para o Expert-Rete.

Os resultados obtidos neste capítulo vão servir de base para a realização da conclusão do trabalho, que será tratada no próximo capítulo.

7. CONCLUSÃO

Este trabalho é um estudo sobre as técnicas de inferências. Ele buscou dar uma visão geral a respeito das principais tecnologias encontradas, localizando a importância dos motores de inferência em Sistemas Baseados em Conhecimento e especificamente para o Expert-Coop++.

A seguir são descritas as considerações finais do trabalho, as principais contribuições obtidas e aplicações futuras que podem ser utilizadas como forma de conclusão para esta dissertação.

7.1 Considerações Finais

Sistemas que utilizam o motor de inferência baseado no encadeamento progressivo, podem ter mecanismos eficientes, se bem desenvolvidos. A medida em que a base de regras aumenta, o método de checar cada condição de regra com cada fato na memória de trabalho pode se tornar um trabalho computacionalmente custoso.

Como descrito em Freeman-Hargis (2005), no encadeamento progressivo tradicional a complexidade computacional é da ordem de $R*A^C$, onde R é o número de regras, C é o número aproximado de condições por regra, e A é o número de acessos na memória de trabalho. Esta complexidade é, portanto, exponencial com o número de condições.

Existem maneiras de minimizar este problema e então tornar o motor de inferência mais próximo da realidade de problemas do mundo real. Uma das soluções mais efetivas para

realizar este processo é através do algoritmo Rete. O algoritmo Rete, como já comprovado ao longo deste trabalho, reduz o número de comparações entre condições de regra e fatos na memória de trabalho. Com isso o tempo de resposta do sistema se reduz muito.

A complexidade computacional, segundo Freeman-Hargis (2005), se reduz com Rete para (RAC), portanto, uma complexidade linear no lugar de exponencial que se encontra no encadeamento progressivo tradicional.

O algoritmo Rete, entretanto, requer processamento e memória adicionais para guardar o estado do sistema de ciclo para ciclo, embora nos testes realizados a memória não tenha aumentando demasiadamente. O custo de memória e processamento pode ser considerável, mas justificável para o aumento de eficiência. Para grandes bases de dados em muitos ciclos de inferência, o Rete é justificável. Já para pequenos problemas, ou em aqueles em que a velocidade de processamento não é necessária, e a memória sim é escassa, o método Rete pode não ser a melhor opção.

Foi implementado, então, um motor de inferência para a biblioteca Expert-Coop++ denominado de Expert-Rete, com processo de inferência baseado no algoritmo Rete. Com a apresentação dos resultados fica evidenciado que o Expert-Rete da forma como foi desenvolvido é mais eficiente e eficaz sob determinadas condições que serão descritas a seguir.

Para situações sob restrições de tempo em que possivelmente o primeiro ciclo de inferência é suficiente para o sistema, o motor de inferência clássico possui menor tempo de resposta. No caso do Expert-Coop++ é recomendado o motor de inferência baseado no encadeamento progressivo para o nível Instintivo. É importante salientar que o Expert-Rete possui uma propriedade em que se pode determinar também se a inferência será realizada em um ciclo ou em vários, e uma outra propriedade que pode ser adaptada para ser utilizada no nível instintivo que determina o número máximo de interações. Após este número máximo ser

atingido, o motor de inferência retorna a base de fatos modificada, mesmo se após a inferência esta base de fatos não representa o resultado ótimo do sistema.

Com esta possibilidade de se determinar o número de interações no ambiente (número máximo de checagens de padrões) para a realização da inferência, Expert-Rete pode ser tornar também um motor de inferência preparado para operar sob restrições de tempo, embora este não tenha sido o objetivo principal de sua implementação.

Para outras situações em que podem ser utilizadas bases de regras mais ricas que realizem múltiplos ciclos de inferência, o Expert-Rete apresenta menor tempo de resposta que o motor de inferência clássico. Está diferença se torna ainda mais discrepante quando se aumenta o número de ciclos.

Apesar de já se saber que o algoritmo Rete tem como objetivo a utilização máxima da memória para a construção do grafo e realização da inferência, com as otimizações realizadas este não se tornou fator de grande relevância sobre o total utilizado pelo sistema (diferença de até 4% sobre o motor clássico sob as condições de simulação realizadas).

Originalmente o algoritmo Rete não possui a característica de se economizar os nós de uma entrada e nós de junção, característica esta implementada no Expert-Rete que tem como objetivo realizar, em contraponto, uma certa economia de memória e diminuir o espaço de busca pelos nós aumentando a performance da inferência propriamente dita. É importante ressaltar que neste caso percebeu-se nos testes um tempo de resposta maior para a realização da montagem do grafo, característica esta que pode perfeitamente ser utilizada no nível cognitivo do Expert-Coop++, pois a base de regras nem sempre é alterada. Caso o usuário do Expert-Rete deseje optar por não otimizar o grafo ou diminuir o tempo de resposta para montagem do grafo, são disponibilizados métodos no motor de inferência para tanto. Segue tabela comparativa a seguir contendo as situações analisadas em seus piores casos.

A Tabela 7.1 descreve comparativamente o método clássico (encadeamento progressivo tradicional) e o Expert-Rete nas piores situações, em um ciclo de inferência e em múltiplos ciclos. Desta maneira o método clássico precisou de 0,46 segundos para realizar a inferência em um ciclo, contendo 341 regras. Já o Expert-Rete realizou a inferência em 1,32 segundos com a mesma base de regras.

Por outro lado, em múltiplos ciclos a situação se reverte com o método clássico realizando a inferência em 12,52 segundos, contendo 17 ciclos de inferência, e o Expert-Rete realizou a inferência em 3,05 segundos com a mesma base de regras.

Situação	Método Clássico (encadeamento progressivo tradicional)	Expert-Rete Otimizado
Único Ciclo	0,46	1,32
Múltiplos Ciclos	12,52	3,05

Tabela 7.1: Comparação entre os métodos de inferência (em segundos)

7.2 Contribuições

Ao final deste trabalho o Expert-Rete se tornou uma sub-biblioteca incorporada ao Expert-Coop++ se tornando mais uma opção a ser adotada por usuários do Expert-Coop++. A partir daí o usuário desenvolvedor poderá optar em utilizar a biblioteca com o motor de inferência clássico ou o Expert-Rete.

Se destaca no Expert-Rete a contribuição de otimização do grafo como forma de aumentar a performance no momento da inferência. Demais propriedades se referem à capacidade do Expert-Rete de se adaptar a diversas situações, dando flexibilidade ao mesmo. No Expert-Rete os seus métodos e propriedades podem ser utilizados sob estratégias de tempo de resposta, além de estratégias de resolução de conflitos na execução do motor de inferência.

Como o Expert-Rete é totalmente desenvolvido por meio de classes, o código do mesmo pode ser facilmente reutilizado para diversas outras aplicações. A fundamentação teórica deste trabalho também pode ser ponto de partida para entendimento das principais técnicas de inferência existentes.

7.3 Aplicações Futuras do Expert-Rete

Como continuação deste estudo há a possibilidade de implementar agentes autônomos em SMA utilizando a arquitetura do Agente Autônomo Concorrente, onde o Expert-Rete será utilizado no nível cognitivo. Pode ser utilizado, por exemplo, um agente autônomo para navegação de robôs móveis, dentre outras aplicações.

Em problemas específicos em se precise de um motor de inferência com as características do Expert-Rete é possível ainda que um usuário qualquer da biblioteca Expert-Coop++ instancie somente as classes necessárias ao motor de inferência para a utilização em qualquer aplicação.

REFERÊNCIAS

AMIR, Eyal. "**Expert Systems".** Notes by Eyal Amir. 4 de Abril de 1997. Obtido em http://www.cs.uiuc.edu/~eyal/qual/, 2004.

CAWSEY, Alison. "**Essence of Artificial Intelligence (Essence of Computing) - Problem Reduction"**. Obtido em http://www.cee.hw.ac.uk/~alison/ai3notes/ subsection2_6_3_3.html, 1994

BAXTER, Jeremy e HEPPLEWHITE, Richard. "**A Hierarchical Distributed Planning Framework for Simulated Battlefield Entities**". DERA 2000. Obtido em http://citeseer.nj.nec.com/baxter00hierarchical.html

BIGUS, Joseph P. ; BIGUS, Jeniffer. "**Constructing Intelligent Agents using Java**". John Wiley & Sons 2001.

BITTENCOURT, Guilherme a. **Inteligência Artificial Ferramentas e Teorias**. 2ª Edição, Editora da UFSC, Brasil, 2001.

BITTENCOURT, Guilherme b. **Sistemas Especialistas**. Departamento de Automação e Sistemas, UFSC 2001.

BROOKS, R. "**A robust layered control system for a mobile robot**". In: IEEE Journal of Robotics and Automation, v. 2, n 1, p.14-23, 1986.

COSTA Augusto L. d.; Bittencourt, Guilherme. "**Parla: A cooperation language for cognitive multi-agent systems**, UFSC, 1997

COSTA Augusto L. d. ; BITTENCOURT, Guilherme. "**From a concurrent architecture to a concurrent autonomous agents architecture**". IJCAI'99, Third International Workshop in RoboCup, pages 85-90. Springer, Lecture Notes in Artificial Inteligence.

COSTA, Augusto L. da; BITTENCOURT, Guilherme, SILVA Luciano; GONÇALVES, Eder. **Expert-Coop++: Ambiente para Desenvolvimento de Sistemas Multiagente**. Núcleo de Pesquisa em Redes de Computadores e Departamento de Automação e Sistemas, 2003.

COSTA, Augusto L. da; LOPES, Grinaldo; FERREIRA Joacir; SOARES, Ivan; RACHE, Rodrigo; ROCHA, Lícia, **Arquitetura de Agentes**. Mestrado em Redes de Computadores – NUPERC, relatório interno, UNIFACS 2003.

DERMAZEAU, Yves. "**From interactions to collective behaviour agent-based systems**". In: Proc. of the 1st European Conference on Cognitive Science, France 1995.

DEMAZEU, Yves; MÜLLER, Jean-Pierre. "**Descentralized Artificial Intelligence**". North-Holland: Elselvier Science Publishers, 1990. p.3-13.

DOBOIS, D. e PRADE, H. "**Fuzzy Sets and Systems: Theory and Applications**". Academic Press, 1980.

DUBOIS, D. e PRADE, H. "**Possibility Theory – An Approach to the Computerized Processing of Uncertainty**". Academic Press, 1988.

EROL, Kutluhan; HENDLER, James; NAU Dana S.. "**Semantics for Hierarchical Task-Network Planning**". Computer Science Department, Institute for Systems Research. University of Maryland, College Park 1994

FEIGENBAUM, et al. "**A case study using the dendral program**". B. Meltzer e D. Michie editors, Mchine Intelligence, volume 6 páginas 165-190. Edinburgh University Press, Edinburgh, GB, 1971.

FIGUEIRA FILHO, Carlos Santos da. **JEOPS – Integração entre Objetos e Regras de Produção em Java**. Dissertação de Mestrado, Universidade Federal de Pernambuco – Centro de Informática. Recife 2000.

FIPA – Foundation for Intelligent Physical Agents, "**FIPA Agent Management Specification**". Obtido em http://www.fipa.org/specs/fipa00023/

FORGY, Charles L.. **"Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem"**, Artificial Intelligence 19 ,páginas 17-37, 1982.

FREITAS, Frederico Luís Goncalves de. **Sistemas Multiagentes Cognitivos para Recuperação, Classificação e Extração Integradas de Informação da Web**. Tese submetida à Universidade Federal de Santa Catarina com parte dos requisitos para a obtenção do grau de Doutor em Engenharia Elétrica. Florianópolis 2002.

FREITAS, Frederico Luís Goncalves de. **Ontologias e a Web Semântica**. Artigo para o Programa de Pós-Graduação em Informática, Universidade Católica de Santos - UniSantos. Santos 2003.

FREEMAN-HARGIS, James. "**Methods of Rule-Based Systems**" . AI Depot. Obtido em http://ai-depot.com/Tutorial/RuleBased-Methods.html , 2005.

FRIEDMAN-HILL, Ernest J. . Jess, "**The Java Expert System Shell. Distributed Computing Systems**", Sandia National Laboratories. Obtido em http://herzberg.ca.sandia.gov/jess/docs/52/ Maio de 2001.

GANSNER, Emden; KOUTSOFIOS, Eleftherios; NORTH, Stephen. "**Drawing graphs with *dot***" dot User's Manual. February 2002.

HOWE, Denis. "**The Free On-line Dictionary of Computing**". Obtido em http://dictionary.reference.com/search?q=backtracking. 1993

JAQUES, Patrícia Augustin. **Agentes de Software na Monitoração da Colaboração em Ambientes Telemáticos de Ensino**. Dissertação de Mestrado. Curso de Mestrado em Informática – PUCRS, 1999.

JOSUTTIS, M. Nicolai. "**The C++ Standard Library – A Tutorial and Reference**". First Edition, Addison Wesley Longman, Inc. 1999.

KITANO, et al. "**The robocup synthetic agent challenge, 97**". International Joint Conference on Artificial Intelligence (IJCAI97). Nagoya, Japan, 1997.

KNOBLOCK, Craig A.. "**Search Reduction in Hierarchical Problem Solving**". School of Computer Science, University of Carnegie Mellon, 1991.

KNOBLOCK, Craig A.. "**Automatically Generating Abstractions for Planning**". Information Sciences Institute & Computer Science Department. University of Southern California, 1994. Obtido em http://citeseer.nj.nec.com/ knoblock94automatically.html

McDERMOTT, J.. "R1: A rule-based configurer of computer systems". Artificial Intelligence, 19(1):39-88, Setembro 1982.

MARSHALL, A D. "**AI2 COURSEWARE - Lecture notes + integrated exercises, solutions and marking**". Obtido em http://www.cs.cf.ac.uk/Dave/. Cardiff University, 1997.

MEYERS, Scott. "**Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library**". Addison Wesley, Inc. 2001.

MINSKY, Marvin. "**A Framework for Representing Knowledge**" . MIT-AI Laboratory Memo 306. 1974. obtido em http://web.media.mit.edu/~minsky/papers/Frames/frames.html em 2005.

NAREYEK, Alexander. "**EXCALIBUR - Adaptive Constraint-Based Agents in Artificial Environments**", Hierarchical Planning. Obtido em http://www.ai-center.com/projects/excalibur/documentation/intro/search/refinement/hierarchical.html, 2001.

OLIVEIRA, Ricardo Ferraz de. **Conceitos que Compõem um Expert System,** Universidade do Algarve, Portugal, 28/06/2001. Obtido em http://w3.ualg.pt/~hshah/expert-system/expert-system.html

OLIVEIRA, Flávio Moreira de. **Inteligência Artificial Distribuída**. In: ESCOLA REGIONAL DE INFORMÁTICA, 4, 1996, Canoas. Sociedade Brasileira de Computação, 1996. p. 54-73.

POST, E. L. . "**Formal Reductions of the General Combinatorial Decision Problem**". American Journal of Mathematics, 65:197-268, 1943.

PRESSMAN, Roger S. **Engenharia de Software**. Mc Graw Hill 4ª edição, 1998.

REZENDE , Solange Oliveira. **Sistemas Inteligentes Fundamentos e Aplicações**. 1ª Edição, Editora Manole, Brasil, 2003.

RILEY Gary. "**CLIPS *A Tool for Building Expert Systems***". Última atualização em 3 de Fevereiro de 2005. Obtido em http://www.ghg.net/clips/CLIPS.html

ROBINSON, J. A. "**A machine-oriented logic based on the resolution principle**". Journal of the ACM, 12(1):23-41, Janeiro, 1965.

ROCHA, Anderson de Rezende; JÚNIOR, Elmo Melquíades de Souza e ALVES, Júlio César. **Introdução aos Agentes Inteligentes e aos Sistemas Multiagentes**. UFLA – Universidade Federal de Lavras. DCC – Departamento de Ciência da Computação. 2003

RUSSEL, S. J. e NORVIG, P.. "**Artificial Inteligence: A Modern Approach**". New Jersey, USA: Prentice Hall, 1995.

SHOTLIFFE, E. H.. "Computer-Based Medical Consultations: MYCIN". American Elsevier, New York, 1976.

SICHMAN, Jaime Simão. "**Du raisonnement social chez les agents: une approche fondée sur la theorie de la dépendance**". Thése de Doctorat de l'INPG. France 1995.

SICHMAN, Jaime Simão e ALVARES, Luis Otávio. **Introdução aos Sistemas Multiagentes**. Anais da Jornada de Atualização em Informática – JAI 97. São Paulo, 1997, editora da Sociedade Brasileira de Computação, SBC.

SMITH, E. David; JEREMY, Frank e JÓNSSON, Ari K.. "**Bridging the Gap Between Planning and Scheduling**". NASA Ames Research Center, 2000. Obtido em http://ic.arc.nasa.gov/people/de2smith/publications/KER00.pdf

SMULLYAN, Raymond M. "**First Order Logic**". Springer-Verlag, 1968.

UGAI Lectures "**The Rete Algorithm**". Workshop's Lectures available online. Temple University, Department of Computer & Information Sciences. Obtido em http://yoda.cis.temple.edu:8080/UGAIWWW/lectures/rete.html em Julho de 2003.

WATTERMAN, D.; HAYES-ROTH, F. "**Pattern-Directed Inference Systems**". Orlando: Academic Press, 1978.

WEISS Gerhard. “**Multiagent Systems A Modern Approach to Distributed Modern Approach to Artificial Intelligence**”, Massachusetts Institute of Technology 1999.

WIKIPEDIA **The Free Encyclopedia**. Obtido em http://www.wikipedia.org/wiki/Rete_algorithm, Julho de 2000.