UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

S. M. A. C.
UM SISTEMA PARA MANIPULAÇÃO E ARMAZENAGEM DE CONHECIMENTO

por: *Eng. Alberto Signoretti*
orientador: *Prof. Dr. Fernando A. C. Gomide*

/90

Este exemplar corresponde à redação final da tese defendida por Alberto Signoretti e aprovada pela Comissão Julgadora em 08/06/90.

Prof. Dr. FERNANDO GOMIDE
DCA/FEE/UNICAMP

Tese apresentada à Faculdade de Engenharia Elétrica FEE-UNICAMP como parte dos requisitos exigidos para a obtenção do título de MESTRE EM ENGENHARIA.

8 de Junho de 1990

Dedico este trabalho aos meus pais,
ROMOLO E GABRIELLA,
por todo amor e trabalho.

A. Signoretti

Este trabalho contou com o apoio financeiro do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq

AGRADECIMENTOS

Ao meu irmão Marco e sua esposa Adriana, por todo apoio, amizade e carinho que nunca me faltaram.

À minha noiva Patrícia, pelo amor e compreensão que sempre estiveram presentes me dando forças.

Ao Prof. Dr. Fernando Gomide. Não só pela orientação neste trabalho como também pela amizade e confiança em mim depositados.

Ao amigo Waldomiro, pela amizade e ajuda sem a qual este trabalho não teria sido concluido.

Ao amigo Rubén pela amizade e pelas incontáveis discussões que fizeram o trabalho evoluir

Aos amigos Jayme, Romulo, Victor e Mário, e à amiga Maristela pelo companherismo e bons momentos.

A todos os amigos do LCA (e antigo LAB 25).
À Universidade Estadual de Campinas por ter me dado a possibilidade de fazer este trabalho.

Ao Conselho Nacional de Desenvolvimento Cientifico e Tecnológico (CNPq), pelo apoio financeiro.

E a todos aqueles que colaboraram, direta ou indiretamente, para a realização deste trabalho.

RESUMO

O presente trabalho define, especifica e implementa o protótipo de um Sistema para Manipulação e Armazenagem de Conhecimento genérico, o SMAC. Este sistema adota, como forma de representação do conhecimento, arquiteturas tidas como convencionais no universo da Inteligência Artificial (IA), agrupadas e manipuladas de uma maneira a tornar-se compatível com as tendências atuais da área. São utilizadas estruturas baseadas em frames, regras e procedimentos, que podem ser usadas isolada ou conjuntamente, dependendo do tipo de conhecimento a ser representado, do estilo de programação do usuário do sistema e da aplicação em mente. Além da generalidade quanto ao conhecimento a ser representado, o sistema visa ser genérico também com relação à aplicação para a qual pode ser configurado.

ABSTRACT

This work defines, specifies and implements the prototype of a system for knowledge storage and management, the SMAC. The system uses, as knowledge representation schemes, conventional architectures within the Artificial Intelligence (AI) universe, grouped and organized in a form to be compatible with the current tendencies in the area. The system uses, as knowledge representation structures, frames, rules and procedures. The structures can be used either together or one or two of them alone, depending of the kind of knowledge that it is needed to represent, and user programming style. The SMAC intends to be a modular and generic knowledge storage and manipulation tool which can be configured for several engineering and computer science applications.

UM SISTEMA PARA MANIPULAÇÃO E ARMAZENAGEM DE CONHECIMENTO

ÍNDICE

CAPÍTULO 1 : Sistemas Baseados em Conhecimento

1.1- Introdução

Neste capítulo é apresentado um panorama da Inteligência Artificial e suas aplicações em potencial. São mostrados alguns tipos de representação do conhecimento, seus usos e possibilidades de inter-relacionamento entre elas. São abordadas algumas arquiteturas em uso em alguns sistemas baseados em conhecimento. Uma revisão bibliográfica apresenta trabalhos importantes sob o enfoque que visa mostrar as várias estruturas utilizadas, para que tipo de problema e qual o resultado alcançado. A partir disto, são introduzidas as motivações e os fundamentos que levaram ao desenvolvimento deste trabalho.

1.2- IA : Organização e Arquitetura

1.2.1- Definições e características

A Inteligência Artificial (IA) pode ser conceituada dentro de um amplo espectro mas, sob ponto de vista pragmático ela representa um conjunto de técnicas que tentam solucionar alguns problemas de uma forma diferente e, muitas vezes, mais adequada do que as disponíveis em sistemas convencionais. Nestes sistemas as soluções são obtidas a partir de algorítmos, via um número finito de passos. Este tipo de procedimento é tipicamente determinístico e por demais específico ao problema que deve ser resolvido. A IA propõe uma manipulação simbólica, onde os símbolos podem estar associados a valores numéricos ou não. A aplicação de procedimentos de inferência manipulam as relações existentes entre esses símbolos com o objetivo de obter uma conclusão lógica. Assim, os procedimentos de inferência possibilitam um raciocínio não determinístico: as conclusões são obtidas a partir do conhecimento embutido no sistema e dos dados relevantes ao contexto.

Dentro do universo da IA existem várias tecnologias e, uma delas, diz respeito aos Sistemas Baseados em Conhecimento, em particular Sistemas Especialistas (SE). Um sistema especialista é um programa de computador que usa conhecimentos e procedimentos de inferência na solução de problemas que normalmente só seriam resolvidos com a participação de um especialista humano [16].

Um Sistema Especialista é constituido de três partes principais [16] : i) a base de Dados Global que é uma memória de trabalho e armazena informações sobre o estado do sistema, ii) a Base de Conhecimento que contém as verdades sobre o domínio da aplicação, e heurísticas (regras de bom senso e suposições razoáveis que caracterizam o nível da tomada de decisão) relativas à área e, iii) a Máquina de Inferência que controla o fluxo de informação no sistema e aplica a Base de Conhecimento à Base de Dados, para obter novos dados e informações.

A figura 1.1 mostra a interrelação entre os vários componentes de um Sistema Especialista.

Dentro de um Sistema Especialista, são pontos críticos a quantidade de conhecimento disponível na base de conhecimento e a forma pela qual este conhecimento será manipulado com intuito de obter a conclusões corretas. Como consequência, a eficiência do SE está diretamente ligada à estrutura utilizada na representação do conhecimento e à forma de manifestação deste. Diz-se que a base de conhecimento deve ser transparente e flexível.

A flexibilidade traduz a capacidade de adquirir novos conhecimentos e modificar o conhecimento já existente.

A transparência é a capacidade que o SE deve possuir para explicar o raciocínio desenvolvido para atingir uma conclusão [58]. A confiabilidade do sistema se torna maior à medida em que este consegue explicar precisamente ao usuário o caminho seguido na obtenção do resultado.

Qualquer SE deve ter conhecimentos sobre os objetos relevantes de seu domínio, as suas propriedades e as suas relações com os outros objetos. Também deve ter noções sobre os acontecimentos que possam acontecer, os já acontecidos e sobre quando ocorrem acontecimentos novos [16]. Além disso, o sistema deve ter conhecimento sobre os próprios conhecimentos, ou seja, ele deve ter noção sobre a extensão desse conhecimento, sua origem, o grau de confiabilidade de suas informações e a importância relativa de certos fatos [16]. O conhecimento que um sistema especialista tem sobre sua forma de raciocinar é denominado metaconhecimento ou conhecimento de segunda ordem.

A habilidade de rastrear seu próprio raciocínio confere aos sistemas especialistas algumas vantagens em relação aos sistemas convencionais : i) o usuário confia nos resultados pois pode acompanhar passo a passo o processo de raciocínio; ii) o desenvolvimento do sistema é mais rápido pela facilidade de deteção e correção de erros; iii) a operação do sistema é explícita e não implícita como acontece nos sistemas convencionais; iv) os efeitos de mudanças no sistema são facilmente testados.

Idealmente, existem três modos de uso de um sistema especialista [16]:

a) obter soluções de problemas;
b) melhorar ou aumentar os conhecimentos do sistema;
c) colher conhecimentos da base para serem usados por humanos.

Para efetuar tarefas relacionadas com os modos de uso mencionadas acima, é essencial que o SE seja provido com uma interface homem-máquima eficiente para permitir uma interação fácil e adequada com o usuário. Desta forma, os programas que compõem esta interface incluem ferramentas de edição, para facilitar a aquisição e modificação do conhecimento especializado, ou dos dados contidos na base de dados. Além disso, a interface contém programas de entrada e saída, que ajudam o usuário a fornecer ou receber informações do sistema especialista.

A pricipal diferença entre os sistemas especialistas e os programas convencionais é que nos SE existe uma clara separação entre os conhecimentos gerais sobre o problema (base de conhecimento), a informação sobre o problema atual (dados de entrada), e os métodos de manipulação de conhecimentos (máquina de inferência). Nos programas convencionais, os conhecimentos e métodos estão misturados, tornando difíceis as modificações do sistema [16]. O aspecto modular da estrutura dos SE, permite uma manuteção e/ou alteração bastante simplificada.

Dentro do escopo dos sistemas especialistas estão contidos os sistemas que não possuem um conhecimento profundo, mas que atuam com algum conhecimento em seu domínio de aplicação. Existem também os sistemas que são dotados de conhecimento profundo sobre seu domínio de aplicação, sendo chamados sistemas peritos [16,7].

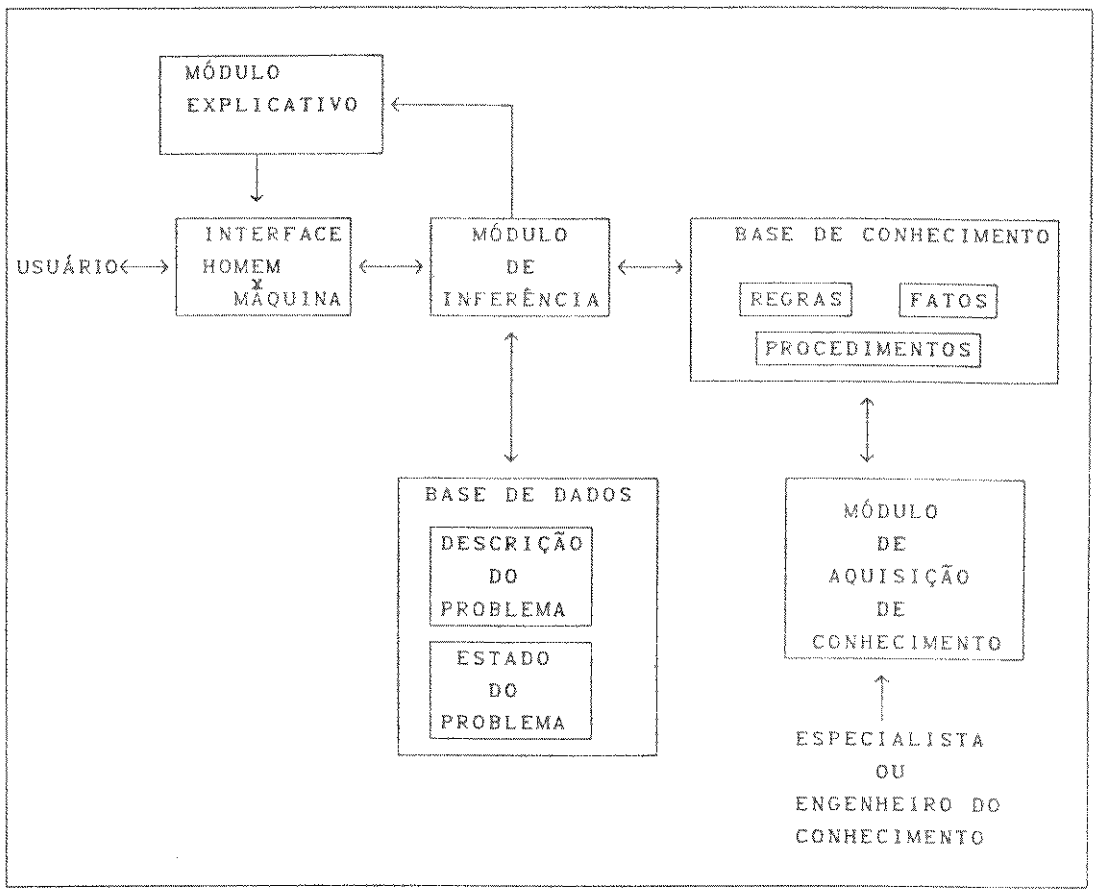


Figura 1.1 : Estrutura Básica de um Sistema Especialista

1.2.2- Sistemas Baseados em Conhecimento: Núcleo Básico

1.2.2.1- Representação do Conhecimento

Uma das áreas cujas pequisas mais avançam é a representação do conhecimento. Entende-se como representação do conhecimento uma combinação de estruturas de dados e procedimentos de interpretação que são usados dentro de um sistema para melhorar o seu comportamento inteligente [16].

Os SE devem manipular uma grande quantidade de conhecimentos especializados e distintos: o conhecimento sobre o problema específico, o conhecimento geral sobre como proceder para solucionar um problema, o conhecimento de como interagir com o usuário para fornecer ou obter

informações, etc [58]. Portanto a escolha da forma de representação do conhecimento apropriada é fundamental para a criação de um SE eficiente.

Existem, dentro da IA, várias técnicas para representação do conhecimento, que podem ser usadas sozinhas ou em conjunto, visando um melhor desempenho. A escolha de uma técnica, ou de um conjunto delas, deve levar em consideração que o conhecimento é utilizado em três estágios: i) aquisição de novos conhecimentos; ii) recuperação de fatos; iii) raciocínio (inferência) com esses fatos.

A lógica, que é o estudo matemático e filosófico mais antigo sobre a natureza do raciocínio e do conhecimento, foi um dos primeiros esquemas de representação usados em IA [16]. Usa-se a **lógica de predicados**, ou lógica de primeira ordem, pois esta permite a representação de conceitos que não podem ser representados na lógica proposicional. Este tipo de representação esbarra na dificuldade de escolher as melhores declarações para um determinado problema. Além disso, a utilização de cláusulas muitas vezes não considera importantes informações heurísticas que estão contidas na apresentação original dos fatos.

O tipo de raciocínio utilizado pela lógica de predicados é dito **monotônico**. Ou seja, o número de fatos conhecidos como verdadeiros só tendem a crescer ao longo do tempo. Se novos fatos são agregados à base de conhecimento, então novos teoremas podem ser provados, mas nenhum conhecimento ou fato anterior se torna inválido [46].

A lógica de primeira ordem não se presta bem a situações em que as informações são incompletas, quando as situações mudam ao longo do tempo ou quando é necessário gerar suposições ao longo da tentativa de solucionar um problema complexo [46].

Para solucionar os problemas descritos anteriormente usa-se o **raciocínio não-monotônico**. Neste tipo de raciocínio, a adição de um novo fato, considerado verdadeiro, à base de dados pode levar à deleção de outro que passe a não ser mais verdade. Portanto, quando um novo fato é agregado toda a base de conhecimento deve ser analisada para verificar a consistência do que nela está armazenado. Este tipo de representação, no entanto, requer um tempo computacional e um espaço de armazenagem bem maior do que nos sistemas monotônicos. Nestes sistemas, a cada teorema provado deve-se providenciar a armazenagem da base de conhecimento que foi utilizada . A tentativa de provar outro teorema modificará a base corrente, destruindo, talvez, fatos necessários à prova do primeiro teorema.

Ainda pensando em lógica, existem situações em que o que se conhece é algo que provavelmente é importante [16,46]. A representação do conhecimento em tais casos faz uso do **raciocínio estatístico e probabilístico**. Desta forma, são utilizadas heurísticas, que representam a informação probabilística necessária para ajudar a direcionar o raciocínio com maior probabilidade de alcançar a solução desejada.

No sentido de manipular o conhecimento incerto existe a **lógica nebulosa** (*fuzzy logic*) que permite tratar fatos não somente dentro do aspecto verdadeiro ou falso. Neste tipo de lógica os fatos recebem um grau de certeza dentro de uma região limitada, por exemplo de -1 a 1, onde -1 representa o que é totalmente falso e o 1 representa o que é totalmente verdade. Os valores

intermediários indicam o grau de certeza que se tem sobre o fato. Assim, deduções em lógica nebulosa são obtidos com um determinado grau de certeza [31,42].

As estruturas de representação de conhecimento discutidas a seguir são as mais difundidas para a elaboração de sistemas especialistas. São estruturas que não possuem um embasamento teórico formal. Porém, mostram-se bastante eficientes no tratamento de alguns dos problemas do mundo da IA.

Além de conhecimentos estáticos, ou seja, fatos sobre objetos, eventos e suas relações, os sistemas de IA também devem saber como usar esses conhecimentos, como encontrar novos fatos, como fazer inferências e chegar a novas conclusões, etc. A melhor maneira de representar esse conhecimento relacionado ao que fazer com a base que se dispõe é através de **procedimentos** [7]. Uma idéia interessante pesquisada atualmente é a combinação de representações declarativas e representações "por procedimentos". Pretende-se representar conhecimentos declarativos acoplados com as instruções para seu uso [30,37,48,50,56].

As **redes semânticas** são formalismos de representação de conhecimento declarativo, onde as informações são representadas através de um conjunto de arcos rotulados e nós. Em geral, os nós representam objetos, conceitos ou situações de aplicação, e os arcos representam as relações entre eles [16].

Dentro das redes semânticas pode-se utilizar a busca de intersecção. Esta visa encontrar relações entre objetos ao espalhar a ativação de conjuntos de nós, verificando, posteriormente, onde a intersecção ocorre. Outro tipo de raciocínio baseia-se na equiparação de estruturas: constrói-se um fragmento de rede representando o objeto que está sendo procurado e depois o fragmento é comparado com a rede armazenada num banco de dados para verificar se o objeto existe.

A teoria de **dependências conceituais** está baseada na noção de primitivas semânticas. A idéia fundamental desta teoria é a de poder representar todos os tipos de ações através de um pequeno número de primitivas. Nestes sistemas, a entrada é dada através de textos em alguma linguagem natural e a "conceituação" do texto é construida. As conceituações não são palavras, mas conceitos. O conceito se refere a um conhecimento contido na linguagem, sendo assim, a representação por dependência conceitual se dá independentemente da liguagem utilizada [16,46].

A representação de conhecimento através de **regras** é um modo natural de descrever processos onde ocorram mudanças rápidas e complexas do meio [58]. Um conjunto de regras especifica como o programa deve reagir a uma alteração dos dados, sem haver necessidade de um fluxo de controle detalhado [58].

No esquema de regras , os conhecimentos são representados através de pares condição-ação chamadas de regras de produção ou simplesmente produções [16]. São usadas sentenças do tipo SE condição ENTÃO ação. Durante a execução, se a parte do antecedente (condição), for satisfeita, ela pode disparar a ação, isto é, a ação contida na parte do consequente (ação) é executada.

A utilização de regras facilita a explicação do raciocínio utilizado pelo sistema na obtenção de uma condição, pois os consequentes de uma regra podem ser os antecedentes de outra regra, até se alcançar uma conclusão. Esse

encadeamento sucessivo é chamado de cadeia de inferência. O encadeamento pode ser do tipo para a frente ou direto (*forward chaining*) onde o sentido de busca é do antecedente para o consequente, ou seja, parte-se do estado inicial para se atingir o estado objetivo. Também existe o encadeamento para trás ou reverso (*backward chaining*) que parte do consequente em direção ao antecedente, ou seja, do estado objetivo para um estado inicial. Em alguns casos, utiliza-se a combinação de ambos os encadeamentos visando obter uma condição de casamento entre os dois tipos.

Apesar das regras serem uma representação de conhecimento basicamente de procedimentos, o formalismo de regras de produção têm várias vantagens dos esquemas declarativos, principalmente a modularidade [16]. Além disso, esta estrutura se assemelha muito à maneira convencional das pessoas expressarem pensamentos. Por isso, o esquema de representação por regras de produção tem sido muito usado como base para desenvolvimento de sistemas especialistas.

Segundo experimentos realizados, parece evidente que os seres humanos utilizam conhecimentos advindos de experiências anteriores para interpretar situações novas. É possível caracterizar situações ou partes de uma, a partir de determinados objetos e/ou sequência de fatos. Esse tipo de conhecimento pode ser representado através de *frames* e roteiros [7].

A estruturas de *frames* é composta por *slots*, que são unidades elementares onde são armazenadas as informações. Dentro destes *slots* podem ser colocados valores, outros *slots*, regras de produção, procedimentos ou mesmo outro *frame* que será usado como descritor detalhado da informação contida naquele *slot*. Em geral a estrutura baseada em *frames* é uma rede onde os nós representam conceitos (descritos pelos seus *slots*) e os arcos indicam o relacionamento entre estes conceitos, estando, aqui, embutido o sentido da hierarquia e herança. Ou seja, os nós inferiores herdam as propriedades dos nós hierarquicamente superiores.

Nesta estrutura se consegue obter uma representação declarativa, que é a coleção de fatos estáticos associados a alguns *slots*, acoplada com uma representação por procedimentos, que engloba os procedimentos e regras associados a outros *slots*. Esta representação facilita a adição de novos fatos ao sistema, sem alterações nem dos fatos antigos, nem dos procedimentos utilizados na manipulação destes fatos.

Os roteiros, ou *scripts* [16,46] são estruturas parecidas com os *frames*, projetadas especificamente para representar sequências de acontecimentos [16]. As sequências de acontecimentos são ligadas a situações esteriotipadas onde a análise para planejamento de "o que fazer" raramente aparece. Levando em consideração que a situação é esteriotipada, as ações são razoalvemente previstas.

Scripts podem ser pensados como estruturas de representação de conhecimento que informam a sequência na qual eventos ocorrem em uma determinada situação, os objetos que podem participar dela, e papéis que estes objetos podem desempenhar. Assim, em qualquer ocasião que a situação é tratada, o computador já está "sabendo" mais a respeito dela do que eventualmente possa ter sido informado até o momento. O principal uso dos roteiros até o momento tem sido no "entendimento" de estórias [46].

1.2.2.2- O Raciocínio e a Solução de Problemas

Um problema central na pesquisa em IA é o de como fazer para que os computadores tirem conclusões automáticas a partir de fatos conhecidos e disponíveis em sua base de conhecimento. A solução de problemas é o processo de desenvolvimento de uma sequência de ações para chegar a uma meta [14].

Para que um computador realize a tarefa de encontrar uma solução é necessário considerar a existência de três problemas básicos [46] :

i)**Explosão Combinatorial**: Ao tentar encontrar a solução, o computador segue por vários caminhos em busca daquele que forneça uma solução factível. A quantidade de caminhos pode tornar-se exageradamente grande, representando um tempo computacional não realizável praticamente.

ii)**Conhecimento Incompleto**: Um SE conta com o conhecimento contido em sua base de conhecimento e com aquele que o usuário pode lhe fornecer (quando há possibilidade de interação usuário-sistema). Sendo assim o conhecimento do sistema é limitado, não sendo possível resolver todos os problemas contidos no seu universo de trabalho.

iii)**Inconsistência**: A base de conhecimento do SE deve sempre ser testada para evitar a possibilidade de que fatos contraditórios possam vir a ser armazenados, e resultados completamente distorcidos venham a ser obtidos.

Dentro da área do raciocínio e solução, os primeiros trabalhos foram orientados para a prova de teoremas matemáticos e de lógica proposional. Posteriormente, foi desenvolvido o método da resolução que aparentava ser suficientemente poderoso para possibilitar a construção de um solucionador de problemas completamente geral, onde os problemas seriam descritos através da lógica de primeira ordem e as soluções seriam deduzidas através de um procedimento geral de provas. Os resultados, porém, foram um pouco desapontadores, já que o espaço de estados gerado pelo método da resolução cresce exponencialmente com o número de fórmulas usadas para descrever o problema. Mesmo a utilização de algumas heurísticas independentes da aplicação não conseguiram resultados satisfatórios.

Apesar dos resultados, a pesquisa sobre estes métodos continua viva pois existe uma classe de problemas para os quais não se tem encontrado outro tipo de método de solução. Nesta classe, incluem-se os problemas para os quais não existem descrições completas em termos dos objetos, propriedades e relações relevantes a ele e portanto um método avaliativo simples não funcionaria.

Um problema pode ser definido como a necessidade de transformar uma situação dada (estado inicial) numa situação desejada (estado final ou

solução), usando operadores de acordo com uma estratégia. Na aplicação de um operador a um estado, um novo estado será obtido e este poderá ser final ou não. Esse tipo de especificação de problema é denominada de "espaço de estados". Uma representação gráfica é possível utilizando grafos nos quais os nós correspondem aos estados e os arcos aos operadores ou regras [16,46].

Partindo dessa definição, pode-se entender uma solução de um problema como sendo um conjunto de estratégias para tentar encontrar uma sequência de operadores que transformem o estado atual no estado final (estado desejado).

Uma das técnicas que explora o espaço de estados é a chamada técnica da busca [46].

Um ponto crítico da busca é o tempo computacional envolvido e a quantidade de memória necessária para encontrar a solução. A procura exaustiva em problemas não triviais raramente é viável, pois o número de estados passíveis de criação,com a aplicação dos operadores, cresce exponencialmente em função do número de operadores disponíveis. Este fenômeno,já mencionado anteriormente, chama-se explosão combinatorial.

O objetivo da busca é encontrar um caminho dentro do espaço de estados que leve do estado inicial ao final. A eficiência com que este processo é feito é uma área de interesse e pesquisa. Vários métodos têm sido usados, entre eles os heurísticos, que utilizam algum conhecimento disponível sobre o problema e sobre os operadores para determinar a sequência mais apropriada de operadores.

A busca pode seguir em dois sentidos: i)para frente, onde parte-se dos estados iniciais visando atingir os estados finais ou, ii) para trás, a partir dos estados finais.

A decisão de qual estratégia utilizar, deve levar em consideração os seguintes aspectos:

a) é sempre interessante partir do conjunto menor para o maior que é mais facilmente atingível;

b) é desejável seguir o sentido do menor fator de ramificação (número médio de nós que pode ser alcançado diretamente a partir de um nó);

c) é importante seguir o sentido em que a forma de raciocínio seja mais parecida com a do usuário.

As principais técnicas de busca existentes hoje são [16,46] :

Geração e Teste (*Generate and Test*): Esta busca parte do estado inicial e, aplicando uma sequência de operadores, tenta chegar a um estado final, testando, posteriormente, se este estado é solução do problema. Se isto ocorrer, a busca é encerrada, caso contrário, o processo é reiniciado com uma nova sequência de operadores.

Métodos de Subida (*Hill Climbing*): Esta técnica assemelha-se à anterior. Porém o teste fornece uma realimentação para orientar o gerador de estados. São métodos do tipo gradiente.

Busca em Amplitude (*Breadth-first Search*): Neste caso a busca segue por níveis. Ao contrário dos métodos anteriores que geram um caminho até o estado final, todos os nós possíveis de serem gerados em um nível são gerados e analisados.

Best-first Search : É uma técnica que combina o que há de melhor entre as técnicas de busca em profundidade e busca em amplitude. A cada nível são gerados todos os nós, e dentre eles é escolhido aquele que aparenta ser o mais promissor para levar a uma solução. A escolha é feita com base em alguma heurística conhecida.

Redução de Problemas (*Problem Reduction*): Um problema complexo é decomposto em subproblemas mais simples (quando o problema original é decomponível). O conjunto de soluções destes subproblemas geram a solução do problema original.

Em alguns casos os subproblemas advindos de uma redução de um problema mais complexo não são independentes. Neste caso, para obter a solução do problema original não basta apenas resolver cada um dos subproblemas originados individualmente, mas devem ser levadas em consideração as interrelações existentes entre eles para garantir que a solução encontrada satisfaça ao problema original. Nestas situações o problema original é considerado não decomponível e outras técnicas devem ser usadas, tal como planejamento por exemplo.

Satisfação de Restrições (*Constraint Satisfaction*): O objetivo é encontrar uma solução, sendo, porém, o espaço de solução do problema limitado pelas restrições impostas.

Análise de Meios e Fins (*Means-Ends Analysis*): A cada passo dado dentro do espaço de estados é calculada a distância entre o estado atual e o estado desejado ou final. Procura-se encontrar operadores que minimizem esta distância.

Outra técnica, para raciocínio e solução de problemas, que pode ser usada na robótica e em jogos é o **planejamento**. Esta técnica tenta decidir um curso de ação antes de atuar; um plano é portanto, uma representação de um curso de ação [16].

A procura por uma solução ótima pode ser obtida com um bom plano, em especial nos casos em que os objetivos (estados finais) não sejam independentes. Nestas situações a ausência de um planejamento adequado pode, facilmente, impedir a obtenção de uma solução.

Além disso, planos podem ser usados para monitorar o andamento do processo para atingir a solução e para detectar erros antes que estes possam trazer prejuízos. Isto é interessante especialmente quando o solucionador não é a única parte do ambiente computacional.

1.2.3 - Arquiteturas para Representação e Raciocínio

1.2.3.1 - *Blackboard*

A técnica do *blackboard* tem sido uma área recente da pesquisa em IA. Esta estrutura tenta reunir o que há de melhor em algumas estruturas já conhecidas, somando uma maior versatilidade e modularidade.

Nesta estrutura usa-se um modelo de raciocínio oportunista, onde partes do conhecimento são utilizados para obter a solução somente no momento mais oportuno. A utilização deste conhecimento pode ocorrer tanto em uma inferência para frente (*forward chaining*) como para trás (*backward chaining*).

O modelo consiste em dois componentes básicos [18,19,24] :

1) **Fontes de Conhecimento** : O conhecimento disponível é particionado em partes independentes entre si. Esta independência se estende à representação deste conhecimento e ao mecanismo de inferência utilizado em cada fonte. Extrapolando, pode-se pensar que cada fonte possa ser escrita na linguagem que melhor se adapte ao conhecimento tratado.

2) *Blackboard* : O estado do problema é mantido numa base de dados geral denominada *blackboard*. As fontes de conhecimento alteram o estado da solução parcial à medida em que vão sendo ativadas. A comunicação e interação entre as fontes existentes no sistema acontece somente via *blackboard*.

À medida que se torne necessário, as fontes de conhecimento vão sendo ativadas e manipulam a solução parcial contida no *blackboard* na medida das suas possibilidades. Esta manipulação acontece de forma ininterrupta, ou seja, cada fonte decide por si só quando finalizar a sua atividade sobre a solução parcial. Desta maneira a solução final é construida passo a passo.

A figura 1.2 apresenta uma visualização da idéia básica do *blackboard*.

O sistema que controla o funcionamento geral do modelo pode ser alocado nas fontes de conhecimento, em um módulo separado, no *blackboard* ou numa combinação destes três lugares.

Vale enfatizar que o modelo do *blackboard* é conceitual e não um modelo com indicações para uma implementação computacional.

A sua versatilidade se dá na medida em que é possível utilizar a forma de representação ou linguagem que é mais adequada a cada fonte de conhecimento. A única restrição é que a forma da informação a ser manipulada por cada fonte deve ser padronizada.

A modularidade está presente no fato de que cada fonte de conhecimento é completamente independente das outras. Sendo assim, qualquer alteração se retringe a um módulo fechado.

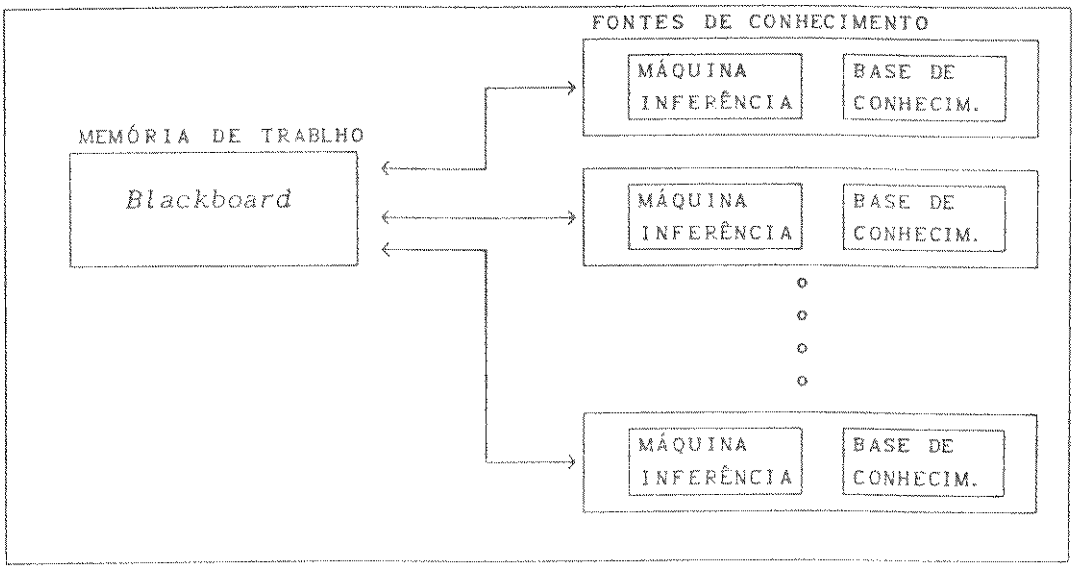


Figura 1.2 : Idéia geral da estrutura *Blackboard*

1.2.3.2 - Estrutura Multi-Agente

Esta é uma arquitetura distribuida, objeto de pesquisa nos últimos anos. É baseada na coordenação inteligente de uma coleção (possivelmente já definida) de agentes independentes e providos de inteligência própria. Essa coordenação consiste em organizar, entre todos os agentes disponíveis no sistema (o seu número pode variar de problema para problema), o conhecimento, os objetivos, as habilidades próprias e os planos, no sentido de , juntos, alcançarem a solução do problema. Dentro desta arquitetura, os agentes podem ou trabalhar todos juntos na solução de um problema global, ou cada um independentemente, em subproblemas que interajam entre si, cuja solução em grupo conduz à solução do problema global.

Nos sistemas multi-agente, os agentes comunicam-se entre si, intercambiando conhecimento sobre os objetivos a serem alcançados, bem como sobre as soluções conseguidas. Isto deve ser feito, porém, sob o processo de coordenação entre agentes.

Neste tipo de arquitetura, a tarefa de coordenação dos agentes é, em determinadas situações, bastante complexa. Existem problemas em que não é possível um controle global, um conhecimento global que seja consistente, um compartilhamento de conhecimento e objetivos, um critério global para definir

a solução e até mesmo não é possível uma representação global para o próprio sistema diante do problema [27,28].

Uma noção simplificada e intuitiva de agente como um módulo computacional leva a um simples processo com seu próprio controle e suas "intenções". Esta visão, porém, é verdadeiramente problemática: os agentes podem ser implementados usando subprocessos concorrentes [1,21], eles podem ter múltiplos e conflitantes objetivos, e a natureza dos conceitos de objetivo e "intenção" não são ainda claros [15,52]. O processo para definir as fronteiras do que esteja compreendido na relação entre um agente e seu mundo exterior é realmente complexo.

1.2.3.3 - Redes Neurais

As redes neurais têm sido estudadas há muito tempo, visando obter um desempenho semelhante ao do ser humano, como por exemplo no campo do reconhecimento de padões e imagens [38]. Os modelos tendem a imitar,aproximadamente, a estrutura do cérebro humano. A rede é constituida por inúmeros nós interligados entre si através de arcos, caracterizados por pesos específicos que, em geral, são alterados visando um melhor desempenho. Ultimamente este campo de pesquisa vem ressurgindo com bastante força devido às novas topologias, novos algorítmos, à técnica de implementação de circuitos VLSI, e também devido à idéia de que o paralelismo é essencial na área do reconhecimento de padrões e imagens.

Estes modelos tentam atingir um bom desempenho através de uma densa interconexão de elementos computacionais basicamente simples. Nas áreas citadas anteriormente, a aplicação desta técnica é interessante, pois muitas hipóteses devem ser analisadas em paralelo, sendo necessário um alto desempenho computacional.

Os elementos computacionais que constituem os nós são não lineares e, em geral, analógicos. As entradas de cada nó são ponderadas, somadas, e passadas pelo elemento não linear, ficando a saída disponível aos outros nós com quem este primeiro tenha ligação. Existem vários tipos de não-linearidades utilizadas. Nós mais complexos podem usar integração temporal, outras funções dependentes do tempo, ou ainda operações matemáticas mais sofisticadas.

O modelo da rede neural é definido pela topologia da rede e pelas regras de aprendizagem [33]. Estas regras especificam os pesos iniciais e como estes pesos devem ser adaptados para melhorar o desempenho da rede. Tanto as topologias quanto as regras de aprendizagem são objetos de muitas pesquisas atualmente.

1.3 - Revisão Bibliográfica

Badiru, em [6], traça um panorama da Inteligência Artificial, em especial dos Sistemas Especialistas (SE), dentro da indústria. São citados algumas áreas de aplicação em potencial como gerenciamento e planejamento da produção, robótica, simulação, alocação de recursos, etc. São discutidas algumas características particulares aos SE's em especial as que complementam e auxiliam as atividades humanas dentro da indústria. Também é apresentada uma sequência para desenvolvimento de um SE, associada a uma lista de problemas que frequentemente aparecem no decorrer de um projeto deste tipo, com algumas soluções possíveis.

Laird et alii, em [36], apresentam um sistema chamado SOAR descrevendo a sua estrutura de representação de problemas dentro dos seus espaços de estado. Neste sistema a solução é encontrada através de uma busca heurística que leva do estado inicial do problema para o estado desejado. Dentro da base de conhecimento são armazenados os operadores com os quais é possível alterar o estado do problema, bem como informações de como podem ser usados. De posse destas informações o sistema pode realizar qualquer dos tipos de busca detalhados anteriormente na seção 1.2.2.2. No SOAR, quando um estado se apresenta de uma forma em que uma decisão sobre qual operador usar não possa ser tomada, um sub-problema é gerado e o processo de solução é reiniciado tendo como estado inicial aquele em que a busca inicial parou. O sistema utiliza dados chamados de "preferências" que representam o conhecimento de como o SOAR vai proceder diante da situação corrente. As "preferências" admitem como dado: rejeição (retorna ao estado anterior), aceitação, o melhor ou o pior movimento ou indiferente.

Gallant, em [20], propõe uma estrutura que permite construir e testar uma base de conhecimento para um Sistema Especialista a partir de exemplos do comportamento desejado para o sistema. O conceito chave está na representação do controle da informação contida na base de conhecimento feita através de uma matriz de inteiros. Esta matriz é denominada matriz de aprendizagem. A estrutura do SE produzido caracteriza-se por uma rede de nós interligados entre si. Para manipular a matriz de aprendizagem o sistema utiliza uma máquina de inferência específica para este tipo de base de conhecimento; MACIE. Esta máquina de inferência é capaz de realizar o encadeamento para frente e também o encadeamento para traz sobre o conhecimento contido na matriz de aprendizagem.

Uma estrutura hierarquizada e modularizada é proposta por Takenouchi e Iwashita em [54], onde diferentes representações de conhecimento podem ser usadas e também diferentes mecanismos de inferência. A linguagem MRL (*Modular Representation Language*) permite criar um sistema modular onde o controle se dá de cima para baixo. Cada módulo é independente dos outros e, dependendo do conhecimento que tenha que manipular, pode conter diversos mecanismos de representação tais como: predicados lógicos, procedimentos, funções lógicas ou ainda regras de produção. Os módulos estão organizados em árvore e ligados uns aos outros através de arcos de relação que estabelecem a hierarquia. Quando o problema é fornecido ao sistema, é o módulo do topo (módulo principal) que o recebe. Este tenta resolve-lo com os próprios conhecimentos. Se neste processo surgem subproblemas que fogem ao escopo do módulo principal, este aciona algum dos seus módulos filhos e este módulo filho repete o processo executado pelo

módulo principal. A escolha de qual módulo filho acionar é feita com base no problema a resolver e no conhecimento disponível em cada módulo filho. Desta forma o problema é desmembrado e solucionado partindo do mais genérico para o mais particular, ou seja,de cima para baixo. Esta estrutura concilia a tendência dos sistemas distribuídos, onde cada módulo é fechado sobre si mesmo, a menos das interfaces de comunicação, com as técnicas mais tradicionais de representação e inferência.

Al-Zabaide e Grimson, em [2], apresentam um sistema chamado DIFEAD que propõe uma coalisão entre sistemas de Base de Dados e Sistemas Especialistas. Neste tipo de estrutura, tanto as fontes de conhecimento como os dados propriamente ditos são armazenados em bases de dados que serão manipuladas por um gerenciador de banco de dados. Esta é uma solução interessante pela modularidade apresentada, já que os Sistemas Especialistas estão desconectados das suas fontes de informação, e também pela possibilidade de manipulação de uma grande quantidade de informação sem sobrecarga por parte dos SE's, pois o gerenciador de banco de dados encarrega-se de manipular os dados eficientemente. A atualização de informações, sejam advindas de deduções feitas pelos SE's ou fornecidas pelos usuários, é feita por um módulo independente que decide onde e como a atualização deve ser processada, deixando os SE's livres desta tarefa também.

Khan et alii, em [32], ilustra uma arquitetura distribuida para um sistema para reconhecimento de objetos. A proposta de Sistemas Especialistas distribuídos permite uma estrutura de organização para bases de conhecimento muito grandes e também o processo de raciocínio é facilitado. O sistema apresenta uma distribuição vertical onde o conhecimento dominado é estruturado na forma de classes hierárquicas. Cada nível é um estado de abstração diferente. Também existe uma distribuição horizontal onde o conhecimento referente ao nível é separado em fontes de conhecimento. Cada nível tem um certo número de fontes e cada uma delas detém um tipo especial de conhecimento para resolver certos aspectos do problema tratado dentro do nível. Estas fontes comunicam-se entre si através da troca de mensagens. O sistema também trabalha com manipulação de incerteza, usando para isso fatores de confiança.

Talukdar et alii, em [55], descreve uma arquitetura distribuida para solucionar problemas. O trabalho dá uma idéia do sistema COPS que é uma versão aprimorada da linguagem OPS5, largamente utilizada para desenvolvimento de programas baseados em regras. A linguagem COPS agrega a possibilidade de que programas independentes, sendo executados em paralelo, possam acessar e alterar o conteúdo de uma memória de trabalho (*blackboard*) que estaria vinculado a um programa mestre (*blackboard program*). Estes programas independentes podem ser escritos em várias linguagens tais como Lisp, C, Pascal. No caso, cada programa tem sua própria função e cada um conta com sua própria máquina de inferência. À medida que cada programa se comunica com o programa mestre, a solução do problema inicial vai sendo construida.

Hayes-Roth, em [24], expõe o problema do controle sobre os sistemas em geral. Como decidir eficiente e rapidamente, dentro do sistema em questão, o que e como fazer, que conhecimento utilizar, que algoritmo se presta melhor ao problema presente, etc. São apresentados vários tipos de controles inteligentes e, por fim, uma estrutura de controle inteligente baseada na arquitetura *blackboard*. O trabalho mostra como o sistema OPM, um sistema de controle usando *blackboard* para planejamento multi-tarefa, explora as capacidades e potencialidades desta estrutura. Também são discutidas algumas

alternativas em contraste com a arquitetura do OPM, e são levantadas as possibilidades de evolução destes tipos de controladores de sistemas.

Erman et alii, em [19], expõe um exemplo prático de uma arquitetura distribuida baseada em *blackboard* para atender ao problema do entendimento da fala. O sistema é o *HEARSAY-II*. A arquitetura proposta foi utilizada também para solucionar outros problemas como interpretação de imagens e compreensão de diálogos. No caso do entendimento da fala, as fontes de conhecimento podem ser encaradas como pares condição-ação. O componente condição será o responsável pela decisão de qual, ou quais, fontes serão utilizadas na solução do problema. A comunicação entre fontes se dá através do *blackboard* que é subdividido nos níveis que representam os níveis intermediários do processo de decodificação da fala. As fontes podem criar novas hipóteses no *blackboard* como também alterar as que já existem. São considerados dois tipos de comportamentos na busca pela solução: análise de meios e fins e redução de problemas. A estrutura permite acomodar ambas as estratégias simultaneamente.

Hayes-Roth, em [26], faz uma elaborada descrição dos sistemas baseados em regras. São apresentados várias qualidades dentre as quais a modularidade, a possibilidade de incrementar e refinar o conhecimento ao longo do tempo, facilidade de explicar o raciocínio utilizado para a obtenção de uma conclusão, etc. São apresentados vários problemas dentro da industria, em que um sistema baseado em regras poderia ser útil na tentativa de solucioná-los ou, pelo menos, minimizá-los. São também ilustrados diagramas de blocos para sistemas baseados em regras em diversos níveis de complexidade com a particularização de cada ítem participante do ambiente.

Bobrow et alii, em [10], trata alguns problemas referentes à construção de Sistemas Especialistas. Por exemplo, a necessidade de levar em consideração que neste tipo de sistema o conhecimento deve ser atualizado e aumentado ao longo do tempo. No caso do sistema R1, estruturado para descrever completamente a estrutura de computadores VAX, o conhecimento foi representado através de regras de produção. A máquina de inferência é do tipo para frente. Para que o sistema tivesse um desempenho aceitável o número de regras foi crescendo ao longo do tempo, passando das 750 usadas no protótipo inicial, para mais de 3500. A representação por regras apresenta a facilidade de permitir uma extensão, atualização e refinamento do conhecimento de uma maneira razoavelmente simples, embora não muito barata, especialmente quando o número de regras da base de conhecimento é muito elevado. A tendência aponta para utilização de sistemas de nível mais elevado que manipulem a base de conhecimento permitindo a aquisição e teste do conhecimento existente e do adquirido.

Oxman e Gero, em [45], propõe um Sistema Especialista para diagnóstico e síntese de projetos. O sistema denominado *PREDIKT* utiliza uma representação de conhecimento baseada em regras de produção e uma máquina de inferência que trabalha para frente e para trás. Um ponto interessante é a possibilidade de usar a mesma base de conhecimento estruturada em regras, tanto para a análise do projeto feito, como para a geração de um projeto a partir de dados iniciais. O sistema utiliza o Prolog como linguagem, pois a linguagem de programação em lógica é um meio bastante poderoso para representar objetos, atributos e, através deles, escrever regras que incorporam estes atributos dentro de um processo de inferência. Neste caso, a estrutura do sistema contém uma interface homem x máquina, a máquina de inferência e a base de conhecimento.

James et alii, em [30], e Taylor et alii, em [56], descrevem o uso de técnicas para programação de Sistemas Especialistas para projetos de compensadores. O resultado foi um sistema chamado *CACE-III* (terceira geração dos sistemas de engenharia de controle auxiliados por computador) que utiliza uma estrutura baseada em regras de produção para representar o conhecimento. A facilidade de entendimento do conhecimento presente e também a sua extensibilidade e aprimoramento são qualidades presentes no sistema. No *CACE-III* estão disponíveis opções que permitem, através de perguntas do tipo "por que?" e "como?", tornar conhecido o processo de raciocínio do sistema. O sistema conta com aproximadamente 300 regras.

Huang e Fan, em [29], descreve o uso de um modelo diferente de base de dados para utilização na engenharia de processos químicos. O modelo relacional não é capaz de preservar a semântica dos dados armazenados. Especificamente, ele não serve para armazenar uma grande variedade de tipos de dados, relacionamentos intrincados e também adicionar a representação do conhecimento. O novo modelo proposto baseia-se nas noções de relação e objeto. É utilizada uma estrutura baseada em *frames* para armazenar dados, atributos e conhecimento. São estabelecidos vários mecanismos de herança para estes *frames*, bem como a utilização de procedimentos associados à manipulação de dados. Também são discutidos aspectos de uma álgebra relacional orientada a objetos que possibilita trabalhar com este novo conceito de base de dados.

Basu et alii, em [9], descreve a arquitetura do sistema CONEX utilizado para projeto e análise de sistemas de controle. A estrutura utilizada para representar o conhecimento lança mão das regras de produção e *frames*. Estes últimos descrevem objetos relevantes ao problema dos sistemas de controle. A base de regras é particionada criando grupos de regras que tenham em comum as mesmas características funcionais. Cada grupo de regras é considerado um nó e existem arcos que ligam estes nós, integrando a base de regras. Diferentes estratégias de busca podem ser usadas em cada grupo. O tipo de estratégia utilizada deve adequar-se ao tipo de regra que cada grupo contém. O sistema também apresenta a possibilidade de explicação, ao longo da execução, de toda a sequência de ações tomadas até o momento.

Mittal et alii, em [41], apresenta um sistema para diagnóstico médico denominado *PATREC*. Este sistema utiliza uma estrutura baseada em *frames*, mostrando que este tipo de arranjo não é somente um depósito passivo de informações, mas uma coleção de agentes ativos que manipulam informações e fazem inferências. Os *frames* são utilizados para representar objetos de interesse. Dentro deles são armazenados valores, atributos e até procedimentos os quais podem ser disparados automaticamente de acordo com as especificações. Os *frames* relacionam-se através de uma herança conceitual, ou seja, possibilitam a implementação do conceito de classe de objetos. Utilizando esta estrutura, o sistema também é capaz de um raciocínio temporal, muito importante do ponto de vista médico e da Inteligência Artificial em geral.

Taylor e Frederick, em [56], propõe uma arquitetura baseada em *frames* e regras de produção, para desenvolver um sistema para projeto auxiliado por computador em engenharia de controle, com as facilidades presentes nos Sistemas Especialistas. O ambiente é um nível superior que envolve os sistemas convencionais do projeto auxiliado por computador. Este nível mais alto fornece uma interação inteligente com o usuário, de forma a facilitar o uso do sistema mesmo para as pessoas menos experientes na área. O sistema também é responsável pela monitoração e validação do projeto que está

sendo desenvolvido. A estrutura tipo *frame* é utilizada para armazenar a descrição do problema e para a construção de sua solução. Os conjuntos de regras interligam os *frames* entre si, os *frames* e os procedimentos, os *frames* e o usuário, e o usuário com os procedimentos. A validação do projeto final e sugestões de correções no projeto são baseadas em um conjunto de regras específicas para o caso. A máquina de inferência que manipula as regras pode efetivar o encadeamento para frente e para traz. A capacidade de mostrar o raciocínio utilizado através de respostas a perguntas do tipo "por que?" também está presente.

1.4 - Motivação e Fundamentos

A motivação deste trabalho parte do propósito de construir um ambiente de programação inteligente que reuna requisitos tais como modularidade, versatilidade quanto à aplicação e facilidade de manutenção e aprimoramento do conhecimento existente. A modularidade visa a facilidade de se criar módulos independentes de conhecimento que possam ser agregados ao ambiente, quando estas informações se tornarem necessárias à solução do problema que se apresenta naquele momento. A tendência aponta para a criação de bibliotecas de conhecimento específicos a vários assuntos. A versatilidade propõe a possibilidade de configurar e utilizar o ambiente para vários fins e não dedicado exclusivamente a uma tarefa apenas. A facilidade de manutenção e aprimoramento, permite alterar o conhecimento contido na base de conhecimento com rapidez, eficiência e simplicidade. Também é permitida a alteração rápida e simples do procedimento principal e caracterizador do ambiente (ver capítulo 2) como dos procedimentos atrelados à base de conhecimento (*frames* , regras e procedimentos).

Pelo que foi descrito nas seções anteriores e, em especial, pelo que foi detalhado na revisão bibliográfica (seção 1.3), uma estrutura baseada em *frames* associada a regras de produção e a procedimentos, tem os elementos necessários para atender aos requisitos descritos acima. A capacidade descritiva e hierárquica dos *frames* associada com a manipulação da herança permite criar bibliotecas independentes de conhecimento, que o sistema pode manipular facilmente. As regras de produção, cujas qualidades foram discutidas na seção 1.3, permitem uma representação clara, eficiente e facilmente alterável do conhecimento. Os procedimentos, por sua vez, podem ser acionados a partir das regras ou dos próprios *frames* para manipular todo e qualquer conhecimento estático que esteja disponível no ambiente. Dessa maneira o ambiente pode ser configurado para trabalhar somente com *frames*, somente com regras, ou somente com procedimentos. As três estruturas podem ser combinadas duas a duas ou utilizadas, as três, simultâneamente.

Um ponto importante é que o ambiente pode ser completamente reconfigurado e continuar usando o conhecimento criado por outra aplicação, pois sua estrutura é padronizada e independente do uso que foi dado ao ambiente. Sendo assim, tem-se um ambiente que pode servir às mais diversas aplicações com a propriedade de ter seu conhecimento acumulado, se assim achar necessário. A possibilidade de configuração para gerar aplicações a problemas específicos (planejamento, projeto auxiliado por computador, simulação, supervisão e monitoração, etc.) constitui-se em outra característica importante do ambiente proposto.

A arquitetura do SMAC (Sistema de Manipulação e Armazenagem de Conhecimento), combina estruturas ditas convencionais para representação do conhecimento de uma forma que se alia às tecnologias mais recentes. Taylor e Fredereick [56], propõem uma arquitetura baseada em *frames* e regras para um caso específico que é o projeto de sistemas de controle. No caso, em primeiro lugar é uma estrutura criada para um problema em que as funções das regras são definidas a *priori* e os *frames* são simples armazenadores de informação. A proposta do SMAC utiliza os *frames* como uma estrutura que, além de simplesmente armazenar informação, pode ser manipulada usando a relação de herança, para efetuar uma inferência ativa como a proposta por Mittal et alii [41]. As regras, no SMAC, são livres para representar o conhecimento que se

achar necessário, pois sua ativação é definida durante a configuração do ambiente para atender uma dada aplicação. A possibilidade de ativar procedimentos a partir de regras, e fazer com que estes tenham participação ativa na inferência que está sendo executada é uma característica singular ao SMAC. No sistema discutido por Basu et alii [9], a estrutura das regras assemelha-se a utilizada no SMAC, no que se refere ao agrupamento de regras específicas divididas em sub-módulos. Porém o sistema apresenta, também, um caráter específico a uma aplicação. O SMAC tem a importante característica de poder ser configurado para atender praticamente todos os sistemas discutidos na seção 1.3. Isto porque o SMAC consegue atender satisfatoriamente a sistemas que utilizem regras , como é o caso de Bobrow et alii [10] ou Oxman e Gero [45], por exemplo, e também consegue atender àqueles que utilizem *frames* apenas, ou associados a regras como os que foram citados anteriormente. Além disso, oferece a capacidade de manipulação de procedimentos associados aos dois tipos de estruturas anteriores.

A idéia de gerar bibliotecas de conhecimento específicas, coloca o SMAC na linha da filosofia do *blackboard* e as fontes de conhecimento, definidas na arquitetura discutida na seção 1.2.3.1 e também em Erman et alii [19], quando o sistema *Hearsay-II* é abordado, e em Hayes-Roth [24] quando se discute o problema do controle de sistemas. Uma extensão prevista para o SMAC discutida no capítulo 4, propõe a possibilidade de manter o conhecimento adquirido em uma inferência, armazenado na memória de trabalho e passível de modificação por outras inferências feitas com outras bibliotecas.

A versatilidade quanto à configuração é a característica principal do sistema SMAC.

1.5 - Resumo

Neste capítulo foi apresentado um panorama da IA e de alguns temas desta área, tendo como enfoque principal as arquiteturas, as formas de inferência, e os esquemas de representação de conhecimento.

A proposta deste trabalho é comparada frente a outras existentes na literatura., Observa-se que a característica básica do sistema SMAC é a versatilidade no tocante à sua reconfiguração como sistema. A possibilidade de manipulação com várias estruturas de representação em diferentes níveis, permite um atendimento a uma larga escala de aplicações. Essa reconfiguração pode seguir várias filosofias de trabalho dentro do campo da IA, inclusive das mais recentes como os sistemas distribuidos. O ambiente proposto combina estruturas já conhecidas, e até consideradas convencionais, de forma a obter uma versatilidade inovadora cujo escopo de atuação é bastante amplo.

No próximo capítulo, a arquitetura do sistema SMAC (Sistema de Manipulação e Armazenamento de Conhecimento) é discutida em detalhes sendo evidenciadas as suas propriedades e características.

CAPÍTULO 2 : Sistema de Manipulação e Armazenagem de Conhecimento

2.1 - Introdução

No presente capítulo a estrutura lógica do Sistema de Manipulação e Armazenagem de Conhecimento (SMAC) é discutida. São apresentadas tanto a arquitetura em si como a funcionalidade de cada parte integrante do sistema, evidenciando como o conhecimento é representado e armazenado. A interface entre o usuário e o ambiente, e a forma pela qual se procede a manipulação do conhecimento embutido no sistema, visando a solução de um determinado problema, também são discutidas. Todo o fundamento usado na formalização da estrutura lógica é discutido tendo por base o que foi apresentado no capítulo 1.

2.2 - Fundamentos, Organização e Arquitetura

O objetivo do sistema SMAC é ser um ambiente onde determinadas qualidades específicas sejam predominantes. Dentre algumas, visa-se privilegiar a versatilidade quanto à aplicação, a modularidade, o baixo custo e a facilidade de manutenção e aprimoramento do sistema. Para tornar o sistema viável na prática, a forma de representação do conhecimento a ser utilizada é uma escolha de fundamental importância. Como já foi visto em seções anteriores, é da representação do conhecimento que, basicamente, depende a eficiência, em vários aspectos, do sistema implementado.

Como uma das características pretendidas para o SMAC é a possibilidade de atender a várias aplicações distintas, um só tipo de representação de conhecimento seria bastante restritivo, pois cada aplicação tem a representação que mais se adapta ao conhecimento envolvido. O problema consiste em encontrar uma arquitetura que, sendo combinação de algumas outras, possa abranger uma quantidade razoável de aplicações.

A representação utilizando *frames* privilegia a representação de conhecimento declarativo e a modularidade. Com a utilização desta estrutura torna-se possível criar módulos de conhecimento independentes. Cada *frame* é fechado sobre si mesmo, havendo entre eles ligações hierárquicas que especificam mecanismos de herança. Para descrever algum objeto utiliza-se o método de generalização-especialização, criando-se uma árvore de *frames* ligados hierarquicamente onde os pais, ou os *frames* dos níveis mais altos, detêm informações sempre mais genéricas que as informações contidas nos *frames* filhos. A herança permite que a informação de níveis mais altos esteja disponível nos níveis mais baixos de uma maneira quase direta. Os mecanismos que manipulam com herança o fazem através de buscas, seja em amplitude, seja em profundidade, dentro de uma árvore de *frames* hierarquicamente organizados. A estrutura da árvore é livre, ou seja, o número de pais e filhos ligados a cada nó não é limitado. Desta forma é possível, na geração dos módulos de conhecimento, adotar a organização que for mais conveniente ao tipo de conhecimento armazenado. Cada árvore é tratada independentemente uma da outra. A quantidade de árvores que podem ser utilizadas ao mesmo tempo é limitado apenas pelo tamanho da memória de trabalho disponível para uso. O armazenamento destes módulos pode ser feito em arquivos individuais ou em arquivos que contenham vários módulos.

A figura 2.1 mostra, utilizando um exemplo simples, o método da generalização-especialização [53] usada na estruturação da hierarquia dos *frames*.

Com a estrutura de *frames* sendo usada desta maneira, torna-se possível obter a modularidade e versatilidade necessária ao sistema. É possível criar bibliotecas de conhecimento a partir dos módulos constituídos por *frames*. Sendo a forma física da estrutura padronizada, nada impede que aplicações diversas utilizem o conhecimento contido em cada biblioteca. Mais adiante será detalhado que cada módulo de conhecimento pode associar os três tipos de representação de conhecimento disponíveis no SMAC, ou seja, *frames*, regras e procedimentos. Cada módulo pode ser considerado uma "ilha" de conhecimento específico em alguma função que podem ser combinados com outras informações via sistema SMAC.

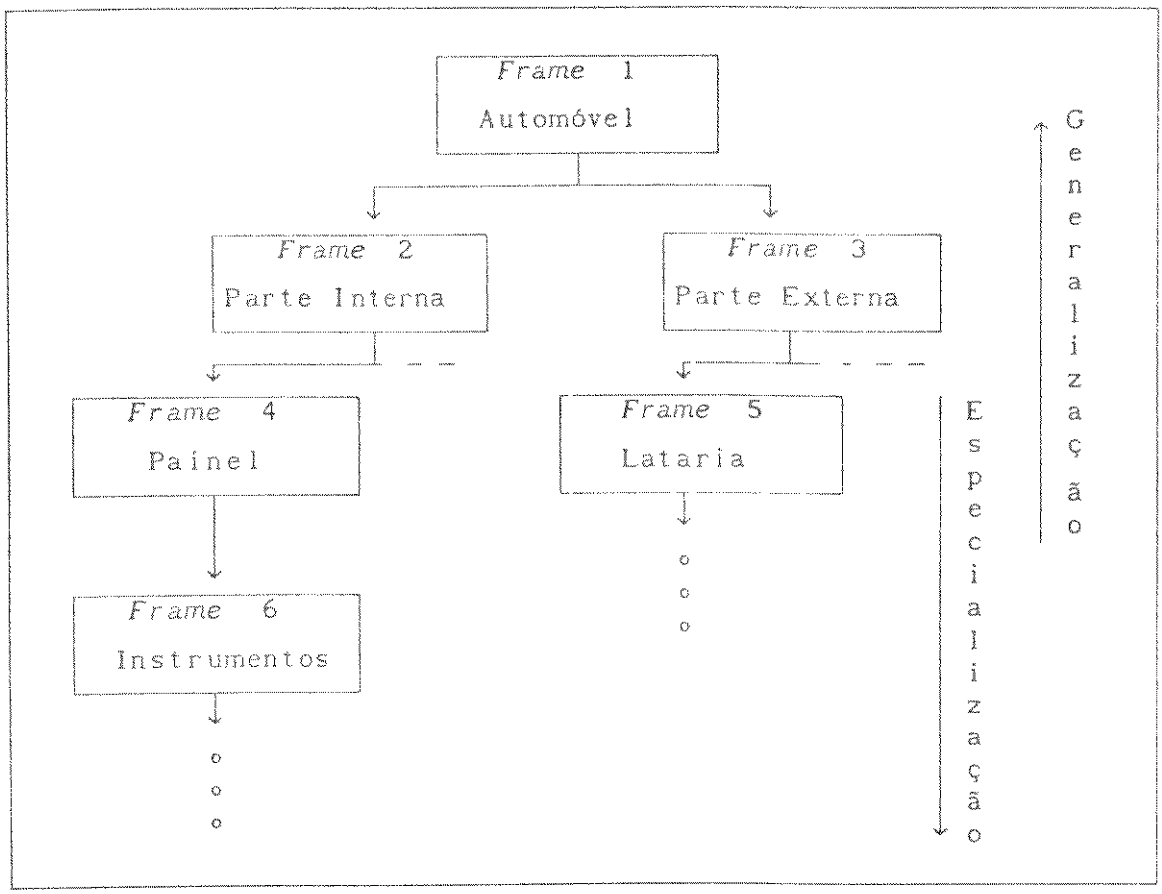


Figura 2.1 : Estrutura Generalização-Especialização

A representação via regras visa obter as qualidades de uma representação por procedimentos e as vantagens da modularidade das representações declarativas. As regras, no SMAC, são descritas através de sentenças do tipo **se** <condição> **então** <ação>. Visando obter a maior modularidade possível, as regras são agrupadas em conjuntos onde cada um pode conter várias bases de regras distintas e independentes. A filosofia continua sendo a de criar módulos independentes e com finalidades sempre mais específicas, para que o sistema possa atender aplicações diversas. Os conjuntos subdivididos em bases de regras independentes, propiciam a divisão de um problema geral em subproblemas menores, de forma que cada base pode tornar-se especialista em uma pequena parte do problema a ser resolvido. O gerenciador do sistema determina qual conjunto deva ser usado e, dentro deste, especifica qual base que deva ser ativada. Os conjuntos de regras ficam sob a supervisão do Sistema Especialista que é comandado pelo Gerenciador (ver figura 2.2).

A representação via procedimentos é utilizada para manipular todo o conhecimento estático contido nos *frames* e nas regras de produção. Os procedimentos estão armazenados em bancos de procedimentos que são supervisionados pelo gerenciador. O sistema permite armazenar procedimentos

dentro dos *frames* e/ou nas regras. Para isso são criados bancos específicos que contém os procedimentos utilizados pela estrutura de *frames* e um outro que contém os procedimentos utilizados pelas regras. Estes procedimentos podem ser programados em linguagens diferentes da que foi utilizada para programar o sistema. O mecanismo de inferência aciona estes procedimentos para que estes, assumindo o controle do sistema e alternando-se um após outro, alterem o estado do problema e conduzam o sistema a uma solução apropriada.

O sistema SMAC permite que cada uma destas representações seja usada independentemente ou em conjunto com as outras. É possível trabalhar somente com *frames*, somente com regras ou somente com procedimentos. A combinação das estruturas duas a duas também é possível. Neste ponto poderíamos ter sistemas trabalhando com *frames* e regras, com regras e procedimentos, ou ainda com *frames* e procedimentos. A maneira, porém, para se obter o máximo de representatividade e eficiência é combinando as três estruturas.

A figura 2.2 mostra um diagrama de blocos que esquematiza o sistema SMAC. A seguir são fornecidas informações sobre a função própria de cada bloco pertencente ao sistema.

Base de *Frames* : consiste de um conjunto de estruturas de linguagem que definem os *frames*, suas ligações, sua hierarquia e indicadores que especificam dados , fatos, conjuntos de regras ou procedimentos armazenados.

Base de Dados : consiste de uma tabela que armazena os dados e atributos utilizados pelos *frames*. Para evitar a sobrecarga da Base de *Frames*, dados, indicadores de conjuntos de regras e indicadores de procedimentos são armazenados numa estrutura separada. Isto permite uma maior modularidade pois uma mesma estrutura de *frames* pode ser usada com vários conjuntos de dados diferentes. A manutenção dos dados também é facilitada por estarem fora da árvore de *frames*.

Manipulador da Base de *Frames* e Dados : consiste de um conjunto de comandos específicos para manipular os *frames* e seus dados. Seus comandos permitem uma manipulação isolada dos *frames* ou dos dados, o que possibilita efetuar alterações e/ou correções mais facilmente. A utilização da herança é definida neste bloco através de comandos especiais. Todo o conjunto de comandos só é disponível diretamente via gerenciador. Isto determina uma centralização de atividades dentro do sistema, que permite a manutenção da integridade das informações, tanto da base de dados como da base de *frames* e das interligações entre essas duas bases.

Interface Homem-Máquina : é o módulo responsável pela interação entre o usuário e o sistema SMAC. É um conjunto de comandos que permite criar um ambiente de trabalho com características amigáveis, o que facilita o uso do sistema por parte daqueles com menos experiência. Esta interface caracteriza a aplicação em que o sistema esteja sendo usado.

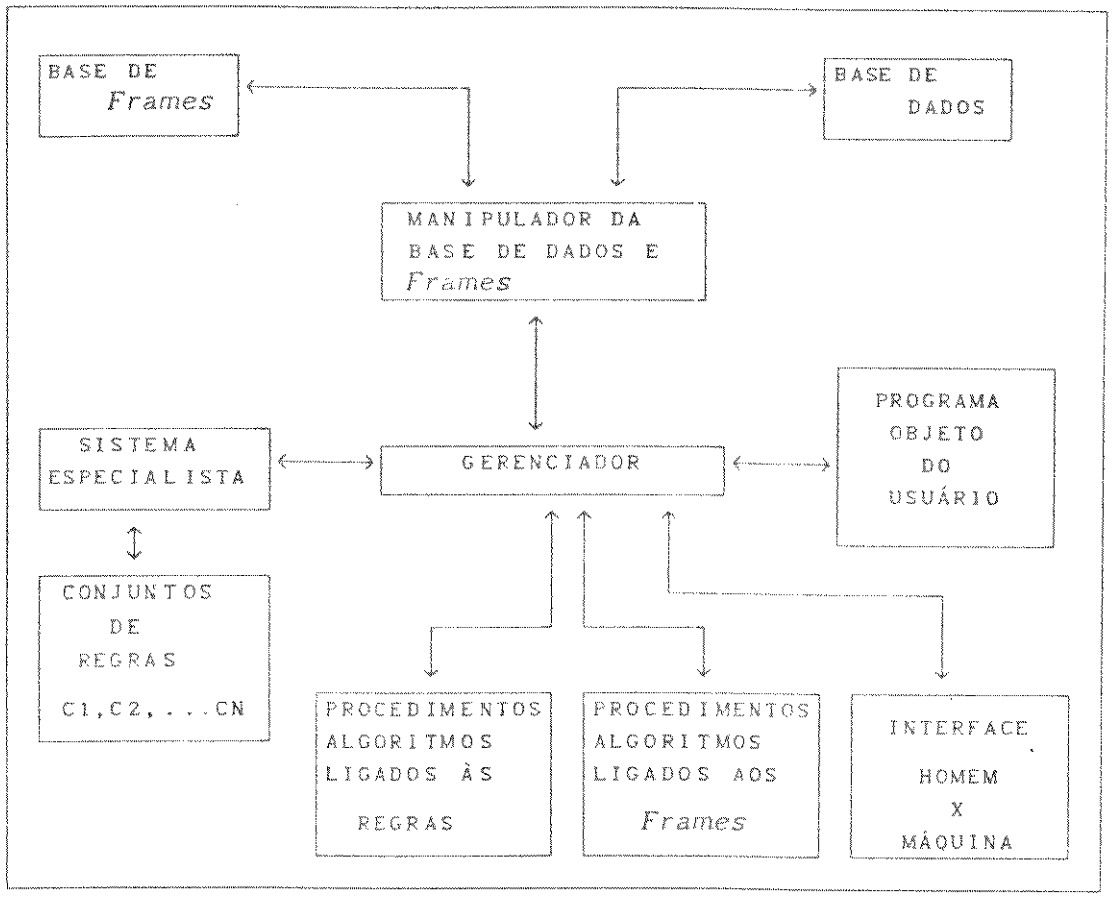


Figura 2.2 : Diagrama de Blocos do Sistema SMAC

Sistema Especialista : consiste de uma máquina de inferência sobre regras de produção, capaz de realizar o mecanismo de encadeamento reverso sobre diversos conjuntos de regras distintos. São disponíveis vários tipos de operações que ativam este sistema a partir do gerenciador. A possibilidade de explicação do raciocínio está presente através das perguntas "porque?" e "como?" durante o tempo de execução da inferência. Fatos novos podem ser adicionados à base de regras em tempo de execução através do usuário, de conclusões próprias da inferência ou como resultado de algum procedimento ativado. Estes fatos são válidos apenas durante a inferência em que foram concluidos. A edição do conjunto ou da base de regras para alteração está presente como um dos tipos de operação do Sistema Especialista.

Base de Regras : consiste de um conjunto de arquivos que define os conjuntos de regras existentes. Cada conjunto de regras pode ter um número qualquer de bases de regras. Os conjuntos são diferenciados a partir dos nomes. As bases, dentro dos conjuntos, são diferenciadas através de indicadores. O Sistema Especialista trabalha com um conjunto de regras de cada vez.

Procedimentos/Algoritmos ligados às Regras : consiste de um bloco computacional onde são agrupados todos os procedimentos que são ativados pelas regras durante uma inferência. Este bloco conterá todos os procedimentos que podem ser acionados por qualquer dos conjuntos de regras presentes no sistema e com alguma probabilidade de ser utilizado para obter alguma informação.

Procedimentos/Algoritmos ligados aos *Frames* : consiste de um bloco computacional semelhante ao anterior, só que os procedimentos se relacionam aos *frames*. Também aqui devem estar contidos todos os procedimentos que estejam relacionados nas árvores de *frames* que, porventura, possam ser usados na solução de algum problema.

Gerenciador : é um conjunto de comandos que centraliza a atividade de todo o sistema. Qualquer acionamento só pode ser efetuado a partir de um ou mais comandos do gerenciador. Ele fornece uma via de acesso ao Sistema Especialista, ao manipulador da base de *frames* e dados e aos procedimentos, sejam eles ligados aos *frames* ou às regras.

Programa Objeto do Usuário : é um programa desenvolvido pelo usuário do sistema SMAC. Não obrigatoriamente deve ser programado na mesma linguagem em que o sistema foi feito. O acionamento e o funcionamento do SMAC é ditado por este programa , que faz uso dos comandos de gerência para acionar os mecanismos disponíveis para armazenar e manipular conhecimento com o propósito de resolver os problemas que se apresentam. A característica que o SMAC assume depende apenas deste programa. A substituição deste fará com que todo o sistema funcione de maneira diferente e possa servir a outra aplicação, tendo a possibilidade, porém, de ter acesso a qualquer conhecimento armazenado anteriormente, se assim se fizer necessário.

2.3 - Representação do Conhecimento

2.3.1 - Representação por *Frames*

A utilização de *frames* para representação de algum objeto leva à estruturação de uma árvore onde a informação vai se especializando à medida que se caminha na direção dos nós folhas. Em cada nó da estrutura em árvore existe um *frame*.

Cada *frame* contém em seu interior um conjunto de *slots* que podem ser preenchidos com símbolos cujo significado pode ser qualquer, seja numérico ou não. Além de símbolos, estes *slots* podem ser preenchidos com outros *slots*, criando uma ramificação, também em forma de árvore, que apresenta, basicamente, as mesmas características que a árvore dos *frames*. Esta semelhança permite uma forma equalitária do tratamento das estruturas, apenas em níveis diferentes. Dentro dos *slots*, a caracterização dos símbolos armazenados é feita através do uso de atributos. Basicamente, existem três tipos de atributos: i) o que especifica o tipo de símbolo em relação à sua origem, ou seja, se ele foi fornecido ou concluído na presente operação do sistema ou se ele é usado por *default*, ii) o que especifica que o símbolo armazenado é o nome de um procedimento. Neste caso, o procedimento é caracterizado quanto ao fato de servir para proteção e policiamento do uso dos dados disponíveis, ou procedimento para inferência ou manipulação de conhecimento, e iii) o que especifica que o símbolo armazenado é o nome de uma base de regras de produção. Neste caso, o nome contém em si próprio a informação sobre o conjunto de regras e, dentro deste, qual a base a ser utilizada.

Dentro da especificação dos procedimentos de proteção são definidos três níveis de atuação : i) a nível de *slot*, ii) a nível de atributo e iii) a nível de dado armazenado. Estes níveis são associados hierarquicamente, ou seja, a proteção ao nível de dado já induz uma proteção parcial aos outros níveis, atributo e *slot*. É dada liberdade para que a hieraquia seja desfeita, dependendo apenas do programador dos procedimentos de proteção. Esta hierarquia não está presente para todos os atributos de proteção, estando disponível apenas naqueles que se relacionem com a retirada e inserção de dados. Onde não existe, a hierarquia pode ser estabelecida com facilidade através da programação dos procedimentos associados a estes atributos.

Cada *frame* possui um cabeçalho inicial formado por um conjunto fixo e definido de *slots* que o caracterizam individualmente, perante todos os demais existentes na base de *frames*. O cabeçalho contém seis *slots* cujos nomes não podem ser usados com outra função a não ser a de especificar o corpo do cabeçalho. Nestes *slots* são especificadas informações do tipo nome, data e tempo (para permitir utlização de processamento temporal), versão (identificar as bibliotecas de conhecimento pela indicação da versão), nome dos pais (identifica a relação de hierarquia partindo do nível mais baixo para o mais alto - *bottom-up*), nome dos filhos (estabelece a relação hierárquica partindo do nível mais alto para o mais baixo - *top-down*) e função (permite identificar a finalidade do *frame* dentro do contexto da aplicação. Por exemplo, utilizando um conjunto de *frames* para descrever um sistema de

controle, o *slot* função especificaria os *frames* cuja descrição representaria blocos como o controlador ou atuador, etc).

Os *slots* podem conter procedimentos e estes têm a liberdade de operarem da maneira que o programador achar mais conveniente. Desta forma, o cabeçalho permite uma localização precisa e inequivoca dentro de todo sistema. Dependendo da situação em que o sistema for utilizado, a necessidade de traçar o caminho que tenha sido seguido por um processo de inferência é imprescindível. Para que isso possa ser realizado eficientemente, o procedimento, quando acionado, deve prover a armazenagem segura das seguintes informações: i) cabeçalho (completo) da instância de partida, ii) *slot* de partida (localização do procedimento), iii) identificação do procedimento chamador, iv) cabeçalho da instância chamada e, v) o(s) *slot(s)* acionado(s) (se existir(em)). Com este tipo de cuidado é possível, com segurança, permitir que os procedimentos atuem sobre qualquer instância dentro de sua classe e até em instâncias de outras classes. A armazenagem de todo o caminho seguido permite avaliar o processo de raciocinio utilizado pelo sistema.

A figura 2.3 mostra esquematicamente o modelo de um *frame* genérico utilizado pelo sistema SMAC. Na figura, o *slot a_kind_of* define a lista dos *frames* pais, e o *slot members* define a lista dos *frames* filhos.

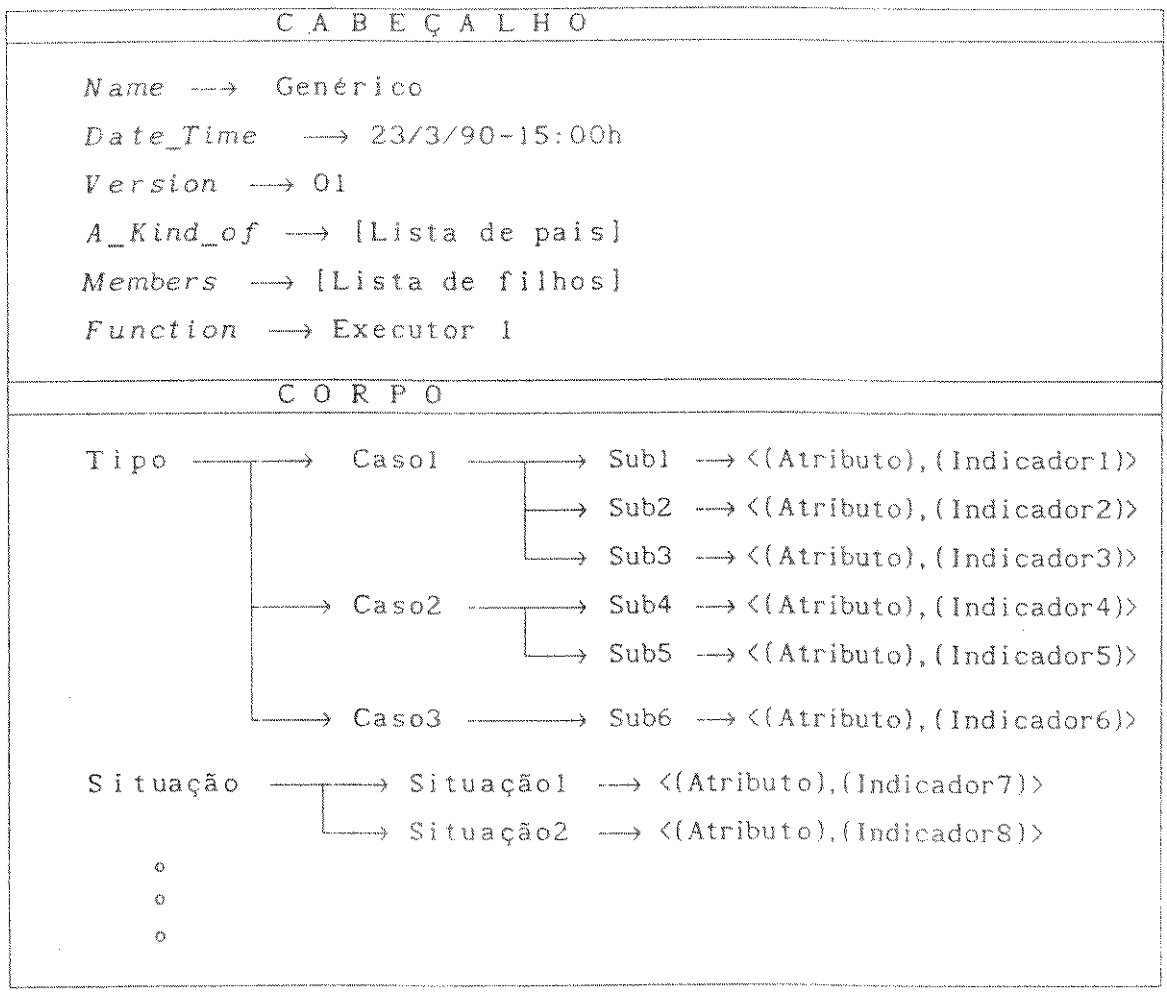


Figura 2.3 : Modelo Lógico de um *Frame* no SMAC

Por definição, todos os dados contidos no cabeçalho de cada *frame* são armazenados utilizando o atributo que especifica valor explícito, ou seja, o atributo *value*.

Na utilização da estrutura para descrever objetos pertencentes ao escopo de utilização do sistema, cria-se uma classe (uma árvore de *frames*) a qual pode descrever um ou mais destes objetos. No caso, o termo objeto relaciona-se a qualquer parte, abstrata ou não, do universo da aplicação que seja passível de descrição. A classe é especificada pelo *frame* de mais alto nível que também é chamado de *frame* raiz. Na raiz são, em geral, armazenadas informações gerais que caracterizam a classe por inteiro, ou seja, a árvore completa. Um exemplo deste tipo de informação está contido no cabeçalho deste *frame*: no *slot a_kind_of* a lista de pais contém o valor *raiz* para identificar a raiz da árvore e, no *slot members*, está armazenada uma lista que identifica todos os *frames* que integram aquela árvore.

Dentro de uma classe (árvore) criada para descrever um objeto, cada *frame* é uma instância da classe que contém informações mais especializadas sobre um determinado aspecto do objeto descrito. Quanto mais próximo das extremidades da ávore (nós folhas), mais especializada será a informação contida na instância.

A figura 2.4 mostra graficamente o exemplo de uma classe. Um *frame* que especifique qualquer dos níveis ilustrados, poderia ser exemplificado pela figura 2.3 anterior.

A quantidade de classes criadas depende do número de objetos do universo da aplicação que necessitam ser descritos. Cada classe é um conjunto fechado que pode ser manipulado através do Gerenciador do sistema. Dependendo da situação, classes podem ser substituidas por outras, de versões diferentes, sendo necessário para isto apenas apagar uma classe e montar outra em seu lugar. O sistema SMAC manipula as informações contidas nas classes, mas a manipulação da classe é sempre efetuada de uma maneira global, ou seja, a classe por inteiro.

Muitas vezes a utilização das classes dá-se com o objetivo de agrupar informações e não de descrição pura e simples de objetos. O exemplo a seguir, ilustrado pelas figuras 2.5 e 2.6, dá uma idéia da utilização das classes como modularização de informações. Neste exemplo, descrito com maiores detalhes no capítulo 4, cada classe é usada para conter informações específicas para projeto e análise de sistemas de controle. A classe *sistema* descreve as características do sistema (tipo de planta, controlador utilizado, especificação do atuador, ...), a classe *requisitos* descreve as necessidades do sistema (características de comportamento desejadas: tipo de transiente, limitações temporais, ...), a classe *solução* armazena a solução obtida e assim por diante.

A estruturação dos *frames* é feita apanas com indicadores no lugar dos símbolos armazenados nos *slots*. Os dados e estruturas estão armazenados em tabelas que possuem relação direta com cada classe. Ou seja, existe uma tabela de dados para cada classe. Dentro destas tabelas, os dados são separados por atributo e cada dado é armazenado com o indicador que especifica a qual instância daquela classe o dado pertence e também a que *slot* ele está ligado. A figura 2.7 ilustra a estrutura genérica de uma tabela de dados associada a uma classe arbitrária.

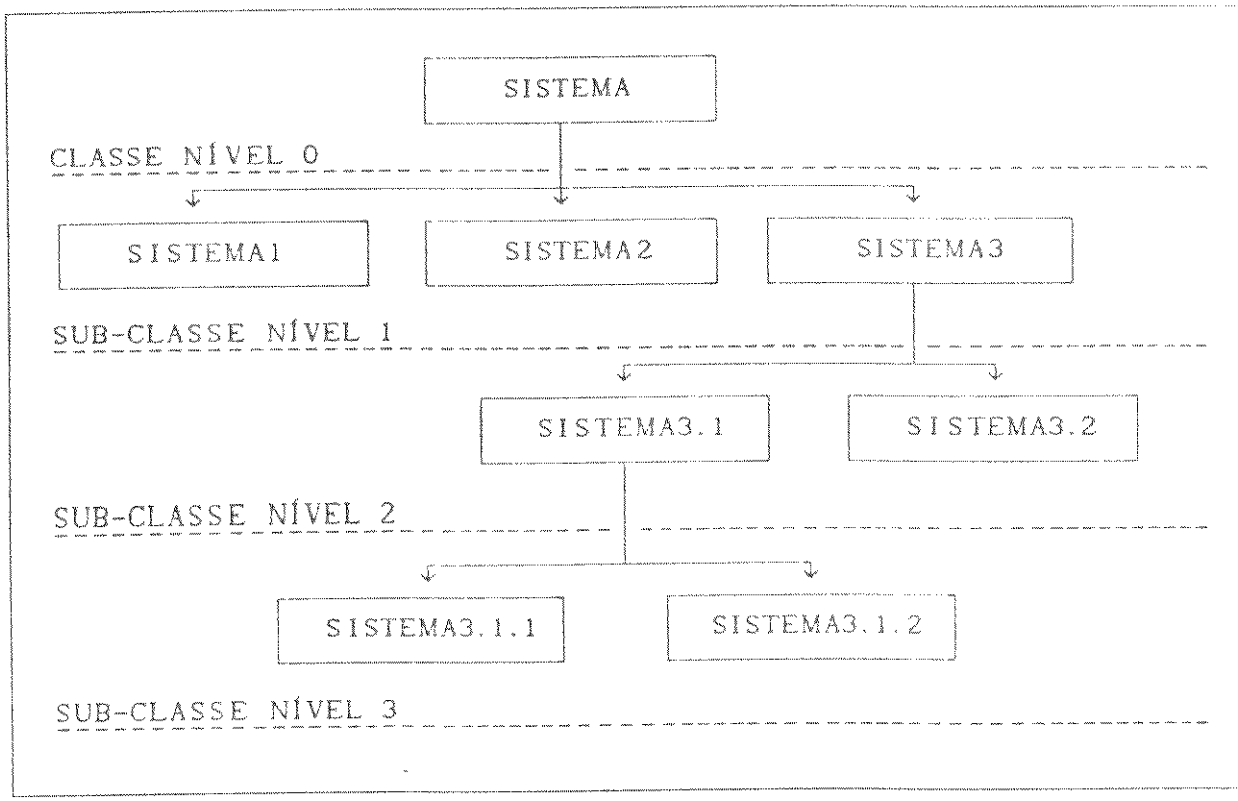


Figura 2.4 : Estrutura de uma Classe no SMAC

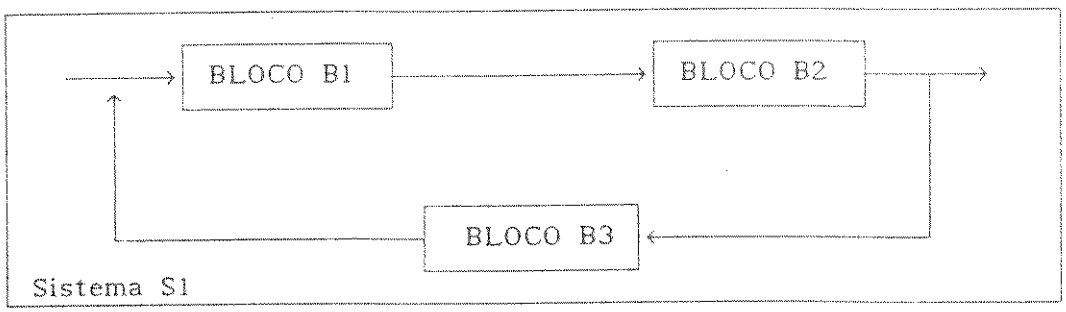


Figura 2.5 : Descrição de um Sistema em Diagrama de Blocos

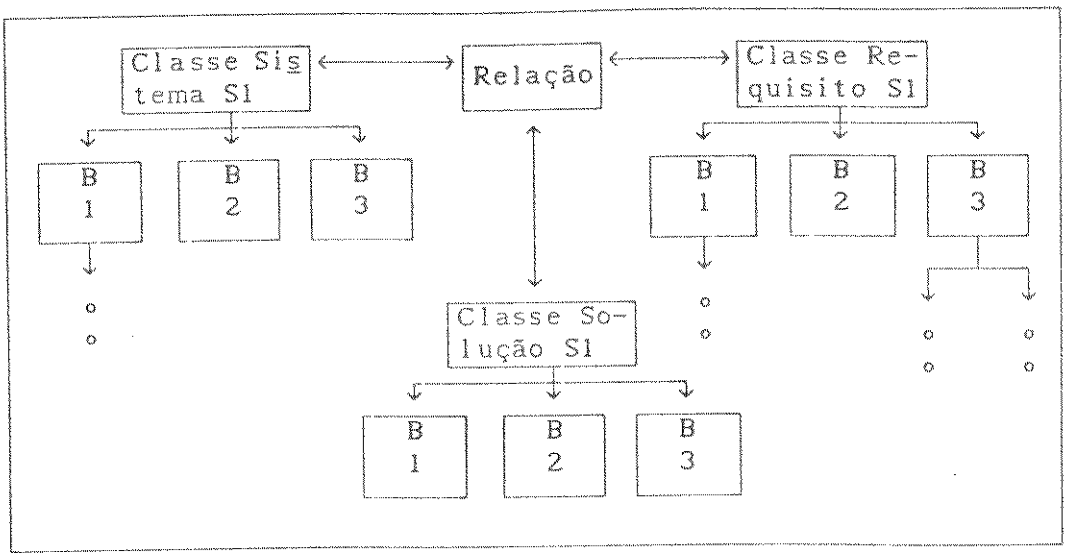


Figura2.6:EstrutuadeClassedoSMACparaoSistemaS1

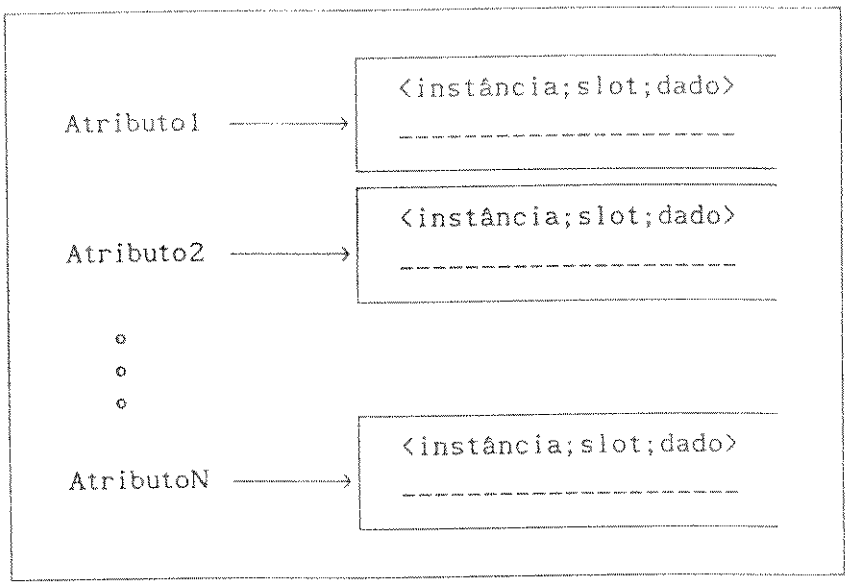


Figura 2.7 : Estrutura de uma Tabela de Dados de Classe

2.3.2 - Representação por Regras

A representação do conhecimento utilizando regras de produção fornece um instrumento poderoso e claro para embutir conhecimento em uma máquina. As regras dentro do SMAC possuem um papel importante, pois são agentes dinamizadores das informações disponíveis. O SMAC fornece facilidades

que permitem escrever regras quase em lingagem coloquial. Com isto, a manutenção, o entendimento e a versatilidade são características reais do sistema.

As regras são contruidas através de segmentos como o ilustrado a seguir:

regra_ind : **se** <condição> **então** <ação>.

A parte <condição> pode conter um ou mais objetivos para serem provados. No caso de haver mais do que um objetivo, o conectivo existente é o da conjunção representado pelo operador **e** (*and*). O conector que representa a disjunção não é disponível diretamente, sendo sua implementação feita através de duas regras cujas partes <ação> sejam iguais. Isto facilita o entendimento e a correção de um conjunto ou base de regras. A precedência da disjunção é imposta pela ordem em que aparecem as regras. O sistema procura regras e fatos sempre no sentido que vai do primeiro ao último. Sendo assim, uma atenção especial deve ser dada para a ordenação do conjunto de regras, pois o raciocínio é afetado de forma importante por este aspecto.

A parte <ação> é sempre formada por apenas uma meta. Esta é a meta de entrada ou, a meta a ser provada pelo sistema responsável pela inferência sobre as regras. O mecanismo realizado por este sistema é o de encadeamento reverso ou *backward chaining*.

A escolha do encadeamento para trás deu-se basicamente pelo fato que é mais simples estruturar conhecimento neste tipo de regras. Uma extensão que inclua o encadeamento direto é razoavelmente simples do ponto de vista do Sistema Especialista e, em relação às regras, basta escrever os conjuntos com a estrutura apropriada para permitir o encadeamento direto.

A parte <condição> pode conter vários objetivos e cada um destes é descrito através de operadores específicos. Estes operadores (descritos com detalhes no capítulo 3) permitem estabelecer estados válidos para os objetivos. Com isso o mesmo objetivo ou meta é usada para diversas condições. Seja, como exemplo, a <condição> dada a seguir:

<condição> : o_tronco_da_árvore eh marrom
e
a_folha eh verde.

O operador é um separador de campos, ou seja, o sistema reconhece o operador **eh** (com significado coloquial: é) e, pela definição dada a este operador, os termos "o_tronco_da_arvore" e "marrom" são isolados. O sistema procura a definição dada ao primeiro termo e verifica se é a mesma do valor do segundo termo isolado. A utilização destes operadores facilita a escrita e a clareza da regra e permite a reutilização dos termos, por exemplo:

<condição> : o_tronco_da_arvore esta seco
e
a_folha esta viva.

Fica fácil verificar que a <condição> acima, apesar de usar os mesmos termos básicos, o_tronco_da_árvore e a_folha, tem um significado completamente diferente da primeira <condição> apresentada.

No tocante aos operadores, a parte <ação> apresenta as mesmas características que a parte <condição>.

Todos os operadores são definidos para a forma afirmativa e para a negativa. O sistema procura sempre provar a afirmação em primeiro lugar, caso

não seja possível, o operador é então invertido para a negação e o fato é armazenado na memória de trabalho como fato temporário concluido da inferência.

O sistema possui um conjunto de operadores pré-definidos que permite uma razoável versatilidade na escrita das regras. É possível aumentar este conjunto de maneira dinâmica, ou seja, dependendo do conjunto de regras a ser utilizado. Cada conjunto pode definir seus próprios operadores para que haja a maior clareza possível na escrita das regras. Quando o conjunto for substituido por outro, os operadores definidos pelo primeiro conjunto são apagados e uma nova definição, se existir no novo conjunto, passa a ser utilizada.

Cada regra é identificada por uma estrutura do tipo: nome-indicador. O nome é utilizado para permitir ao sistema reconstruir a sequência de raciocínio, quando uma demonstração deste caminho é solicitada, pelo usuário, ao sistema, durante uma inferência. O indicador serve para identificar a qual base de regras, dentro do conjunto, pertence a regra que está sendo usada e também define se a regra deve ser utilizada no encadeamento direto ou reverso.

As regras manipulam metas que devem ser provadas, pois não são verdades absolutas. Para concluir a veracidade ou a falsidade destas metas o sistema lança mão de fatos, outras regras, procedimentos, e, em último caso, do próprio usuário. Este último só é questionado se o sistema tiver permissão para fazê-lo. Caso o sistema não tenha acesso ao usuário e tenha esgotado suas outras fontes de conhecimento, a meta é considerada falsa, dentro do escopo do conhecimento embutido no sistema.

Os fatos são metas sobre as quais não se tem nenhuma dúvida sobre a sua veracidade. Levando em consideração a limitação do conhecimento disponível no sistema, podem ser chamados de "verdades absolutas" do domínio da aplicação. A estrutura dos fatos é exemplificada a seguir:

<fato> : fato : <corpo do fato>.

No <corpo do fato> descreve-se a "verdade", utilizando ou não os operadores descritos anteriormente.

Dentro do processo de inferência o sistema utiliza-se um outro tipo de fato cuja existência é temporária. Esta estrutura armazena o fato concluido e considerado verdadeiro, a maneira pela qual ele foi concluido e o indicador que define a qual base de regras este novo fato pertence. Existem três maneiras de concluir um fato: i) através de uma ou mais regras, ii) através da resposta do usuário, e iii) através de um procedimento. Estes fatos são armazenados somente na memória de trabalho, sendo posteriormente eliminados em sua totalidade ao fim de uma seção de trabalho com um conjunto de regras.

As regras podem acionar procedimentos durante a inferência. Estes procedimentos são reconhecidos nas metas a serem provadas através da ausência de operadores como separadores. Junto a cada base de regras existe uma tabela que transforma a meta que se refere a um procedimento, numa chamada a este procedimento, com os devidos argumentos estabelecidos. O mesmo procedimento pode ser usado várias vezes com argumentos diferentes. Para isso basta ter várias metas diferentes e, a cada uma delas, dentro da tabela, definir chamadas com um conjunto específico de argumentos. Isso permite que a legibilidade das regras não seja prejudicada pela presença de chamadas a procedimentos com seus respectivos argumentos no corpo da regra.

O procedimento é livre para executar qualquer função que o programador deseje, pois, ao ser acionado, assume o controle do sistema até o momento do encerramento de suas atividades. O procedimento pode comunicar-se com a memória de trabalho na qual está ocorrendo a inferência, através da armazenagem de respostas fornecidas por este procedimento, como fatos de duração temporária.

O SMAC executa a inferência seguindo uma ordem de prioridades para provar uma determinada meta. A ordem é: i) fatos, ii) fatos temporários, iii) regras, iv) procedimentos e, dependendo da autorização, v) o usuário. Esta ordem é inalterável.

A figura 2.8 ilustra esquematicamente a estrutura utilizada pelo sistema SMAC para trabalhar com regras.

Como se pode observar da figura 2.8, toda e qualquer manipulação sobre a memória de trabalho é centralizada no Sistema Especialista. Isto garante a integridade das informações contidas na memória de trabalho durante a inferência. Esta memória é compartilhada, durante o tempo de execução, por vários conjuntos de regras que são carregados, dependendo da situação a ser resolvida. Não há possibilidade que informações de uma inferência anterior possam ser usadas pelo conjunto que ocupa a memória de trabalho. Isso faz parte de algumas extensões a serem levadas a cabo (ver capítulo 5). Esta seria particularmente interessante pois traz em si algo da filosofia do *blackboard* (ver capítulo 1).

2.3.3 - Representação por Procedimentos

A representação através de procedimentos permite armazenar algoritmos e funções que manipulem o conhecimento disponível no sistema, fazendo com que, ao longo do tempo, o estado do problema seja alterado e direcionado para uma possível solução.

No sistema SMAC, os procedimentos podem ser acionados via *frame*, via regras ou diretamente pelo Programa Objeto. Em todos os casos, qualquer chamada a procedimento é efetuada através de um comando de gerência, centralizando as chamadas em um único módulo. A separação ocorre somente a nível de armazenagem dos procedimentos, pois aqueles ligados aos *frames* estão em um módulo e os ligados a regras estão em outro.

Quando um procedimento é acionado, ele assume o controle de todas as operações tendo total liberdade de ação. A manipulação dos dados é permitida. Porém, esta só pode ser efetuada através do gerenciador que se encarrega de verificar as proteções e de realizar a alteração pretendida nos padrões especificados pelo sistema SMAC. Todo procedimento comunica-se com o seu ativador através de uma resposta que é transmitida, de um para o outro, através do gerenciador. A resposta pode ser especificada da forma que melhor convier ao programador do sistema. No caso dos procedimentos acionados por

regras, a resposta que é enviada pelo gerenciador para o Sistema Especialista, pode ser armazenada ou não, na memória de trabalho, onde se encontram as regras e os fatos usados na inferência. Isto depende da configuração dada à resposta pelo procedimento.

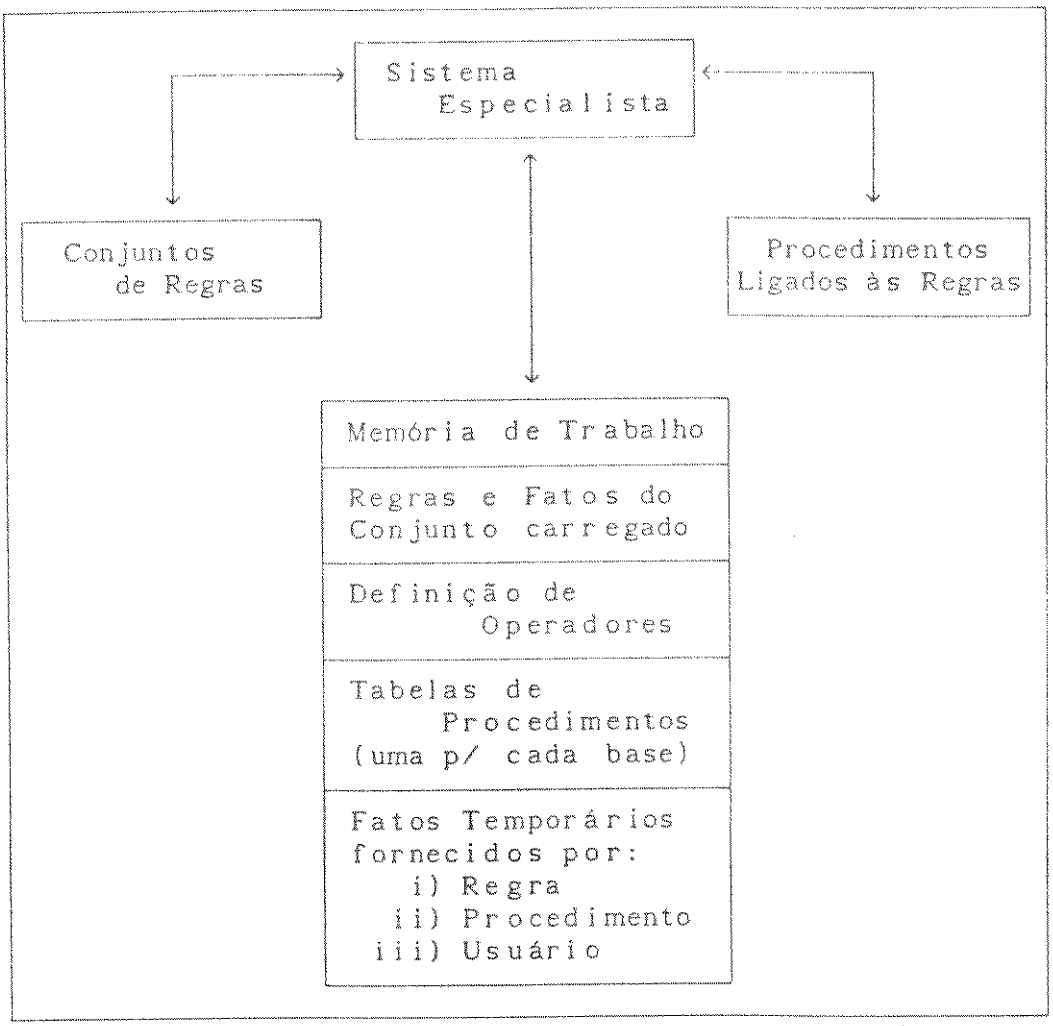


Figura 2.8 : Estrutura do SMAC para utilização de regras

A proteção dos dados contidos nos *frames* é feita via procedimentos. Quando o gerenciador é acionado para manipular algum dado, ele primeiro verifica a existência de algum procedimento de proteção sob o devido atributo. Caso exista, este procedimento é acionado. Isto significa que um procedimento tem liberdade de chamar outro e assim sucessivamente. O procedimento de proteção faz as averiguações necessárias, inclusive comunicação com o usuário, para permitir ou não a manipulação com o dado. Ao encerrar seu funcionamento, o procedimento de proteção envia uma resposta ao procedimento acionador informando a situação. O procedimento acionador decide o que fazer com base nesta resposta.

Os procedimentos, além de poderem ser encadeados, podem ser escritos em outras linguagens, facilitando, assim, sua adequação às tarefas que devam executar.

As figuras 2.9, 2.10 e 2.11 dão uma visão esquemática dos ciclos de ativação dos procedimentos realizados no sistema SMAC. São ilustrados os procedimentos ligados a regras, ligados aos *frames* e, por fim, os procedimentos de proteção de dados.

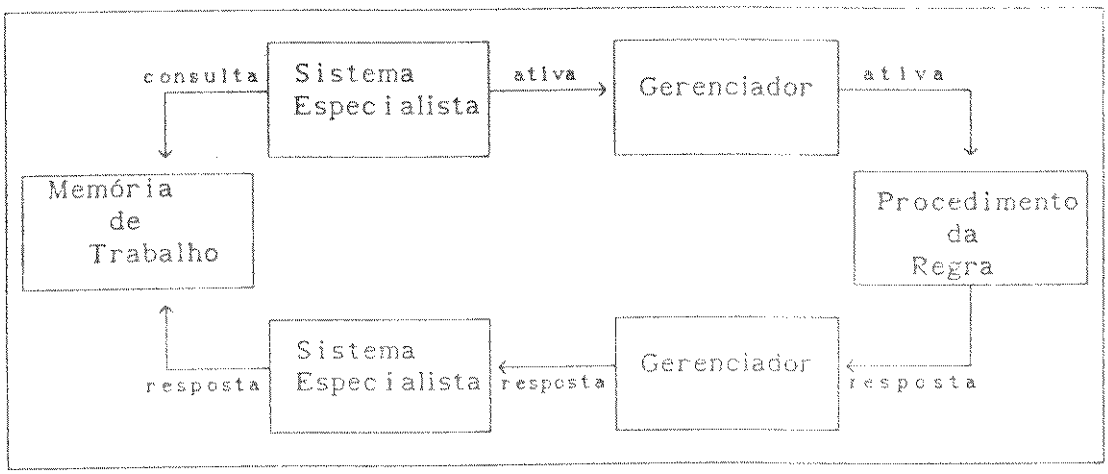


Figura2.9 : Ciclo de Ativação Procedimento de Regra de Produção

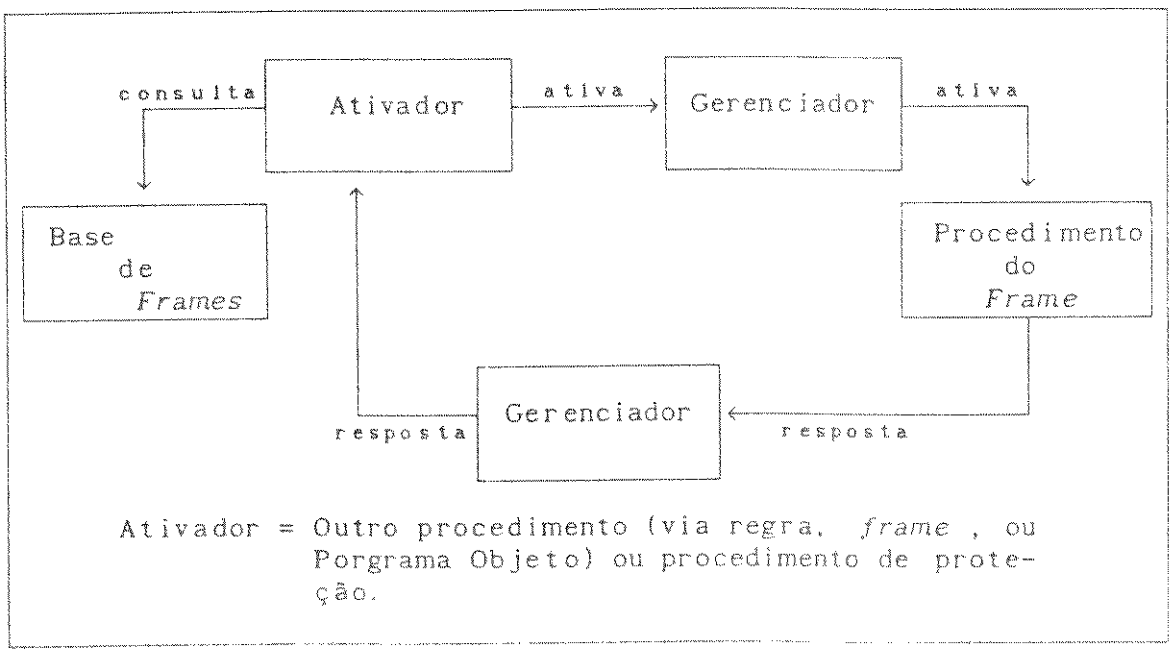


Figura 2.10 : Ciclo de Ativação de Procedimento via *Frame*

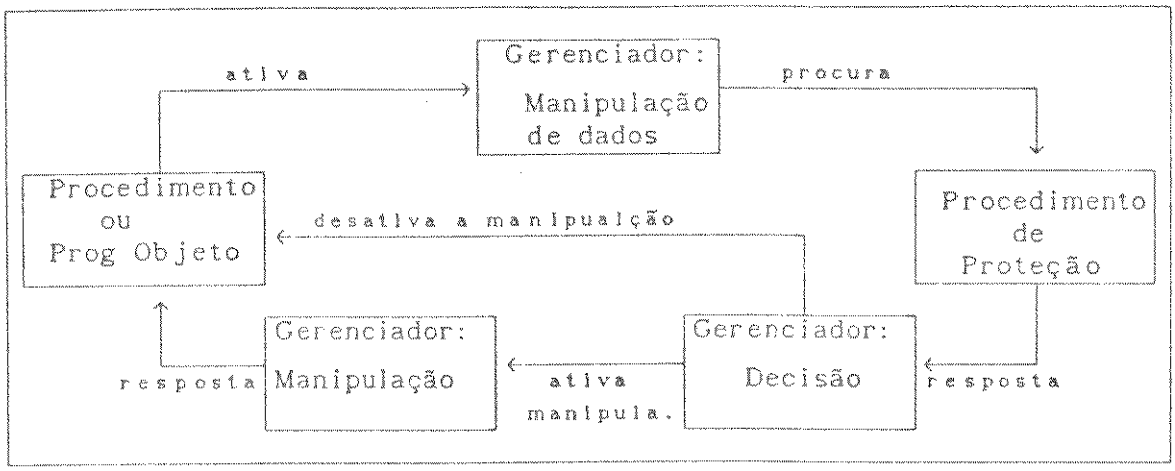


Figura 2.11 : Ciclo de Ativação Procedimento de Proteção

2.4 - Manipulação do Conhecimento

2.4.1 - Manipulador da Base de *Frames* e Dados

Este manipulador consiste em um conjunto de comandos que permitem construir e manipular a base de *frames* e os símbolos nela contidos. Existem três níveis de abstração na implementação dos comandos. O nível mais baixo efetua o trabalho de montagem física da estrutura e suas ligações, permitindo criar, inserir e apagar tanto o *frame* inteiro como parte dele. Estes comandos são responsáveis pela montagem da estrutura em árvore que define o *frame* em si, e pela montagem da tabela de dados associada à classe. A tabela serve para a classe inteira e não a um só *frame* ou instância. Portanto, ela vai sendo criada à medida em que as instâncias da classe vão sendo definidas. O segundo nível é utilizado pelo gerenciador para criação, inserção e retirada de dados e *frames* num nível mais elevado. Nesse nível a coordenação da alocação básica da memória para a estrutura que se deseja manipular é automática. Nesse nível já existe uma comunicação, via resposta, que permite analisar qual o efeito da aplicação do comando. O terceiro nível se utiliza dos outros dois para fornecer algumas facilidades como por exemplo, efetuar a busca de algum dado sob vários atributos ou procurar por um procedimento de definição quando o dado procurado não está disponível. São comandos que permitem uma manipulação com uma maior sofisticação.

Dentre os comandos situados no terceiro nível, os mais importantes são os que manipulam a herança existente dentro da estrutura da classe. Estes comandos, uma vez especificada a instância de partida e o que se deseja, efetuam uma busca dentro de toda a classe. Se a informação é encontrada, essa é fornecida em forma de uma resposta deste comando. Além do dado, são informados a localização (instância) onde o dado foi encontrado, sob que atributo, e o caminho seguido para chegar ao dado. A busca, dentro da árvore das instâncias que definem a classe, pode ser feita de duas maneiras: i) em profundidade, onde são percorridos vários caminhos que vão do ponto de partida até a raiz da árvore e, ii) em amplitude, onde são pesquisados todas as instâncias em cada nível até chegar-se à raiz ou topo da árvore.

Todos estes comandos efetuam a manipulação diretamente. A preocupação com a proteção e integridade dos dados é responsabilidade do nível de gerência do sistema.

2.4.2 - Sistema Especialista Baseado em Regras

Consiste de uma máquina de inferência sobre regras de encadeamento reverso cujo módulo é independente de qualquer outra parte do sistema SMAC. O acesso a este módulo é feito via gerenciador. A ativação da máquina de

inferência pode ser feita em três modos de operação: i) inclusão, que permite inserir novas regras e fatos ao conjunto escolhido, sendo a alteração guardada em arquivo no encerramento do trabalho do Sistema Especialista (SE), ii) ajuda, onde são apresentadas as regras de manuseio do SE , iii) consulta, que permite efetuar uma inferência sobre um conjunto definido, na tentativa de provar uma meta fornecida e, iv) terminar, que permite limpar toda a memória de trabalho do SE sem que nenhum conjunto seja carregado.

Durante a execução da inferência o SE pode fornecer o caminho executado para indicar o raciocínio efetuado até o momento da interrupção. Isto é feito, quando o usuário tem permissão para ser abordado, através da pergunta porque. Como resposta o SE mostra, passo a passo, todas as regras, fatos, fatos temporários e procedimentos que foram utilizados para chegar àquele ponto. O SE pode ser questionado sobre o que sabe a respeito de qualquer meta existente dentro da base de regras corrente, através da pergunta como. Em resposta, o SE mostra as regras e fatos disponíveis sobre a meta em questão.

O SE permite que a meta a ser provada seja fornecida diretamente via comando ou fornecida pelo usuário via interface de comunicação.

O SE é dotado de uma interface homem-máquina que pode ou não ser utilizada, dependendo da necessidade. A interface é seletiva, podendo ser usada por partes. Tal interface é discutida com detalhes no capítulo 3.

Sempre que o SE é ativado, a memória de trabalho é preparada para uma nova inferência. Se o conjunto de regras presente é o que deve ser usado, então somente o conhecimento temporário é apagado e uma nova seção de trabalho é iniciada. Caso haja um conjunto presente e este não seja o escolhido, toda a memória de trabalho é apagada e o conjunto especificado na chamada do SE é carregado.

A interação com o usuário é determinada quando o gerenciador ativa o SE. Dependendo do caso, o conhecimento do operador não é necessário para a solução da meta proposta. Como a interface é seletiva, pode ocorrer que o usuário seja questionado quanto a meta a ser provada e não seja questionado durante a inferência, ou pode acontecer o contrário, ou seja, a meta é fornecida diretamente pelo gerenciador, mas o usuário é questionado durante a inferência.

A comunicação entre o SE e o gerenciador dá-se através de uma resposta fornecida ao término da inferência. A resposta indicará se a meta fornecida no início da seção é falsa ou verdadeira, dentro do escopo do conhecimento disponível para prová-la.

2.4.3 - Gerenciador

É o módulo que efetua o controle geral de todo o sistema SMAC. O controle é feito através de um conjunto de comandos que permitem aos outros módulos executarem funções que envolvam interrelacionamento entre módulos. A

manipulação de dados, por exemplo, só é permitida a um procedimento através de comandos do gerenciador para este fim. A chamada do Sistema Especialista deve ser feita através deste módulo, como também a ativação dos procedimentos.

A função de gerência está distribuida por todo o sistema, pois está presente onde os comandos gerenciadores estiverem. O módulo gerente define tudo o que pode, ou não, ser feito pelo sistema SMAC. Todos os módulos, quando se comunicam ou se relacionam, o fazem através destes comandos que constituem parte de cada um dos módulos.

Os comandos que compõem o módulo de gerência possuem o maior nível de abstração dentro do sistema SMAC. A partir deles, de qualquer lugar, seja de um procedimento, seja do Programa Objeto, é possível atingir qualquer outra parte do sistema sem que haja possibilidade de comprometer a integridade dos dados e do conhecimento disponível.

2.5 - Interfaces e Procedimentos

2.5.1 - Programa Objeto do Usuário

O sistema SMAC visa atender uma vasta gama de aplicações. Desta forma, o comportamento apresentado pelo sistema deve adaptar-se à aplicação para a qual ele visa ser usado. Esta forma específica de comportamento é ditada pelo Programa Objeto do Usuário.

Ao ser acionado, o SMAC procede a uma inicialização que estabelece as condições para que ele possa funcionar. Feito isso, o controle de todas as funções é passado para o Program Objeto que, a partir daí, caracteriza o modo de funcionamento do sistema frente à aplicação.

Este módulo é responsável pela especificação de quais classes serão utilizadas, quais conjuntos de regras deverão estar disponíveis e quais os procedimentos passíveis de utilização. Toda a manipulação é iniciada e finalizada por este módulo. O que ocorre ao longo do processo vai depender de como os procedimentos, que são ativados durante a evolução do estado do problema, vão reagir. Toda vez que um procedimento é ativado, o controle do sistema é a ele repassado e, dependendo do estado em que se encontra o problema, o procedimento pode realizar alguma ação específica. Desta forma, o caminho seguido pelo sistema em busca da solução, depende do direcionamento dado por cada procedimento, quando de posse do controle global do sistema.

O Programa Objeto inicia e finaliza o processo mas, dependendo da sua estruturação, pode reassumir o controle quando for necessário e reconduzir o andamento dos trabalhos. A forma como este módulo é estruturado é de responsabilidade do programador do sistema, para que se obtenha a versatilidade suficiente, de forma que este seja configurado para as mais diversas aplicações.

O Programa Objeto é um programa que usa os comandos do gerenciador acoplados a uma linguagem de programação. Esta linguagem pode ser a mesma em que o sistema SMAC foi desenvovido ou outras como C, Pascal ou Fortran. O problema de utilizar outra linguagem neste módulo é que a utilização de um interpretador, quando necessário, fica impossibilitada, pois estas linguagens são compiladas.

2.5.2 - Interface Homem-Máquina

A interface consiste de um conjunto de comandos que permitem criar facilidades de interação com o usuário. Como o SMAC é um sistema de propósito geral, esta interface não pode ser definida completamente, ou seja, ela também é estruturada para ser de proposito geral.

Cada parte do SMAC, quando necessário, utiliza uma interface própria que é manipulada de acordo com a função que deva ser desempenhada. O Sistema Especialista, por exemplo, apresenta uma interface baseada em janelas e conjuntos de opções que podem ser ativados, ou não, dependendo da situação. O gerenciador também fornece este tipo de facilidade para os procedimentos acionados, podendo ser usada, ou não, dependendo do tipo de procedimento.

Sendo a interface homem-máquina um conjunto de comandos, ela pode ser usada por todo o sistema, independentemente do módulo considerado, pois o acesso também se dá via gerenciador, que está disponível a todos os módulos do sistema.

Cada Programa Objeto manipula estes comandos da maneira mais conveniente, de acordo com a finalidade à qual se destina a utilização do SMAC. Como o Programa Objeto é um programa, no sentido amplo da palavra, nada impede que particularidades da interface sejam programadas diretamente neste módulo.

2.6 - Resumo

Neste capítulo, especial ênfase foi dada à estrutura lógica do Sistema de Manipulação e Armazenagem de Conhecimento (SMAC). Os módulos integrantes do sistema foram discutidos com detalhes. Foram também especificadas as formas de comunicação e de relacionamento entre os diversos módulos. A estrutura foi apresentada e discutida sob o prisma da possibilidade da utilização do SMAC em várias aplicações.

O próximo capítulo trata da implementação do sistema SMAC como um sistema computacional. A ênfase está na implementação e na programação, bem como na linguagem utilizada no seu desenvolvimento.

Capítulo 3 : Implementação e Configuração

3.1 - Introdução

No presente capítulo é discutida a implementação computacional do SMAC. São abordados aspectos como linguagem utilizada, técnicas de programação e estruturação do sistema. A discução se estende a nível de cada bloco lógico discutido no capítulo 2. A estrutura física da base de dados e conhecimento utilizada é descrita a nível tanto da linguagem como da estrutura em si. Isto permite um visão geral do ambiente utilizado para a implementação do sistema. Analisa-se também o SMAC como ferramenta a ser desenvolvida em outras linguagens.

3.2 - Abordagem Geral

3.2.1 - A Linguagem de Programação

Um programa em lógica é um modelo de um determinado problema ou situação, espresso através de um conjunto finito de sentenças lógicas. Ao contrário de programas em FORTRAN ou Pascal, por exemplo, um programa em lógica não é, portanto, a descrição de um procedimento para obter soluções de um problema. De fato, o interpretador ou compilador utilizado para processar os programas em lógica fica inteiramente responsável pelo procedimento adotado para a obtenção de soluções. A programação em Lógica exemplifica assim um estilo mais fundamental, que pode ser chamado de Programação Declarativa (Asserciona1 ou Não-Procedimental), em contraste com a Programação Procedimental (ou Imperativa), típica das linguagens tradicionais. A Programação Declarativa inclui também Programação Funcional, que tem em LISP o seu exemplo mais conhecido, e quase todas as linguagens mais recentes para consulta a bancos de dados, como SQL e QUEL. Lembrando que LISP data de 1960, Programação Funcional é então um estilo conhecido a bastante tempo ao contrário de Programação em Lógica, que só ganhou impeto depois de 1972 com o advento da linguagem Prolog (*PROgramming in LOGic*).

Os conceitos de chamada (ou consulta) e de resposta naturalmente também diferem das noções tradicionais. De fato, uma consulta a um programa em lógica é uma afirmação exprimindo as condições a serem satisfeitas por uma resposta correta em presença da informação descrita pelo programa.

Porém, o ponto fundamental de Programação em Lógica consiste em identificar a noção de computação com a noção de dedução. Mais precisamente, a maioria dos sistemas para programação em Lógica reduzem a busca de respostas corretas à pesquisa de refutações a partir das sentenças do programa e da negação da consulta (uma refutação é uma dedução de uma contradição). Tais sistemas baseiam-se diretamente em procedimentos para pesquisa de refutações, estudos em Prova Automática de Teoremas, e em resultados de Programação em Lógica mostrando como extrair respostas corretas de refutações.

Assim, a resposta de uma consulta a um programa em lógica não se limita apenas a indicar que uma suposição acerca da informação contida no programa é falsa ou verdadeira. A resposta efetivamente exibe informação extraida do programa e pode vir acompanhada de uma explicação sobre como foi obtida, expressa em termos da refutação que a gerou.

A idéia de usar Lógica como um formalismo executável em computador explorada principalmente por Kowalski [34,35] e Hayes [23], recebeu um grande impeto com o advento da linguagem PROLOG, que deve ser entendida como uma implementação da idéias de Programação em Lógica para o subconjunto das cláusulas definidas.

O Prolog é uma linguagem interativa que permite resolver problemas que envolvem representação simbólica de objetos e seus relacionamentos. A linguagem Prolog reforçou a tese de que a Lógica é um formalismo conveniente

para representar e processar conhecimento. Prolog evita que o programador descreva procedimentos para obter a solução de um problema, permitindo que ele expresse declarativamente apenas a estrutura lógica do problema através de termos, fórmulas atômicas e cláusulas.

Um programa Prolog possui três interpretações semânticas básicas. Na interpretação declarativa entende-se que as cláusulas que definem o programa descrevem um teoria de primeira ordem. Na interpretação procedimental as cláusulas são vistas como entrada para um método de refutação. Finalmente, na interpretação operacional as cláusulas são vistas como comandos para um particular procedimento de refutação.

Estas alternativas para a semântica são valiosas em termos de entendimento e codificação de um programa Prolog. A interpretação declarativa permite que o programador modele um dado problema através dos objetos do domínio de discurso, simplificando a tarefa de programar em Prolog, em comparação com linguagens tipicamente procedimentais, como a linguagem Pascal ou C. A interpretação procedimental permite que o programador identifique e descreva o problema pela redução do mesmo a subproblemas, através da definição de um série de chamadas de procedimentos. Por fim, a semântica operacional reintroduz a idéia de controle de execução, que é irrelevante do ponto de vista da semântica declarativa, através da ordem das cláusulas em um programa Prolog e da fórmulas atômicas em uma cláusula Prolog. Ou seja, esta interpretação é semelhante à semântica operacional de muitas linguagens de programação tradicionais e deve ser considerada, principalmente, em grandes programas, por questões de eficiência de execução. É interessante notar que o programador pode comutar de uma interpretação para outra, em busca de um efeito sinérgico, para facilitar a codificação na linguagem Prolog.

Em resumo, as principais características da linguagem Prolog são:

- Orientada a processamento simbólico;
- representa uma implementação da lógica como linguagem de programação;
- apresenta uma semântica declarativa inerente à lógica;
- permite a definição de programas invertíveis, ou seja, programas que não distinguem entre argumentos de entrada e de saída. Como consequência, permite a definição de programas com mais de uma finalidade, que podem ser chamados com formas diferentes de entrada/saída;
- permite a obtenção de respostas alternativas;
- incorpora um mecanismo uniforme para passagem, análise, seleção e criação de estruturas de dados;
- suporta estrutura de dados que permite simular registros ou listas;
- permite recuperação dedutiva de informação;
- suporta codificação recursiva para a descrição de processos e problemas, dispensando os mecanismos tradicionais de controle, como o comando *goto* e laços *do*, *for* e *while*;
- permite aproximar o processo de especificação do processo de codificação de programas;
- representa programas e dados através do mesmo formalismo (cláusulas);
- incorpora facilidade extra e meta-lógicas.

3.2.2 - Estrutura de Armazenagem de Dados e de Conhecimento

3.2.2.1 - Estrutura dos *Frames*

Como foi mencionado no capítulo anterior, os *frames* são organizados em árvores para estruturar uma classe que serve para descrever um objeto pertencente ao domínio da aplicação, ou para agrupar informações sobre determinado ponto de interesse. Dentro de cada *frame*, as informações são armazenadas em *slots* e classificadas quanto ao tipo através de um atributo. Os *slots* podem ser organizados para formar uma árvore onde a informação vai se especializando até chegar a um dado que é armazenado em um nó folha. A figura 2.3, no capítulo 2, mostra claramente esta estruturação. Dentro da classe, em cada nó da árvore existe um *frame*. Dentro do *frame*, existem *slots* que formam ramificações em árvore onde, em cada nó, existe um *slot*. A ramificação dos *slots* é semelhante à dos *frames*, porém possui um nível inferior em versatilidade. As ramificações dos *slots* podem crescer somente no sentido pai para filho. Um nó pode ter quantos filhos forem necessários para explicitar a informação, mas só pode ter um pai. Isto é feito porque, dentro do *frame*, a informação já é especificada, o que implica numa necessidade menor quanto à generalidade da árvore construida internamente. Desta forma, não se tem problema para gerenciar a árvore de *slots* que possuam vários caminhos que conduzam para o mesmo dado.

A estrutura interna do *frame* é uma árvore, e a classe é construida como uma árvore de *frames*. Uma forma interessante de estruturar a sua forma física dentro da memória da máquina é uma árvore balanceada.

Conceitualmente, uma árvore balanceada é um conjunto de nós e ramos (figura 3.1 e 3.2). A busca por algum dado sempre se inicia no nó de mais alto nível. No caso do ambiente de programação utilizado, cada nó pode ter até 127 ramos saindo dele. A especificação normal é de 17 ramos. Cada nó contém informações que determinam qual o ramo deve ser seguido para a efetivação da busca. Cada ramo leva de um nó a outro onde novas informações estarão disponíveis até que seja alcançado um nó folha. Nos nós folhas se encontram os termos armazenados. As árvores balanceadas são eficientes quando se trata de lidar com um grande número de termos semelhantes. Por isso o SMAC utiliza este tipo de armazenagem apenas para os indicadores de dados e não para os dados propriamente ditos.

Ao se utilizar uma base convencional para armazenar dados, a busca é sempre sequencial, o que é bastante ineficiente para uma grande quantidade de dados. A utilização da árvore balanceada reduz consideravelmente o número de comparações a serem feitas até encontrar o dado procurado.

Em geral, o número de ramos da árvore pode ser alterado para aumentar a eficiência da busca. Quando se tem uma grande quantidade de dados, um número maior de ramos resulta em uma busca mais eficiente. No entanto, quando se utiliza um grande número de chaves complexas para particionar a

árvore, um número menor de ramos favorece a rapidez da busca. Como a eficiência se encontra em pontos opostos no tocante a relação quantidade de dados-quantidade de ramos, cada aplicação deve, através da experiência, encontrar a melhor relação para cada caso em particular. No caso do sistema SMAC, as chaves não são complexas, mas a quantidade de dados pode variar muito. Desta forma, optou-se por uma ramificação que seja mais genérica, ou seja, foi utilizada a condição normal do ambiente, até 17 ramos por nó.

Cada classe tem uma árvore balanceada associada. Dentro desta estrutura são armazenados tanto os *frames* que formam a classe como os *slots* pertencentes a cada instância.

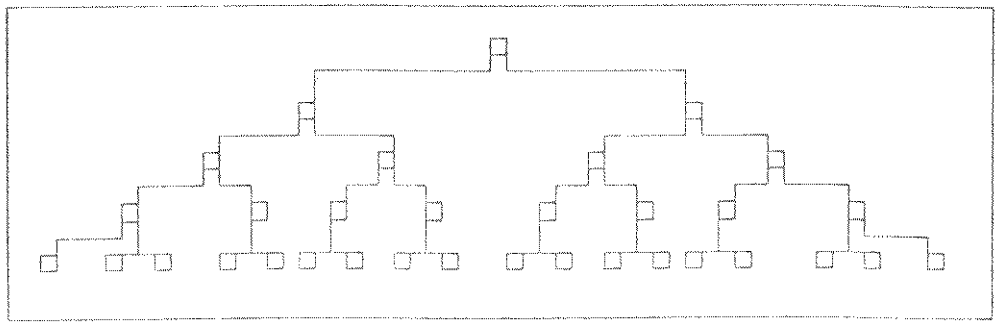


Figura 3.1 : Modelo Conceitual de uma Árvore Balanceada

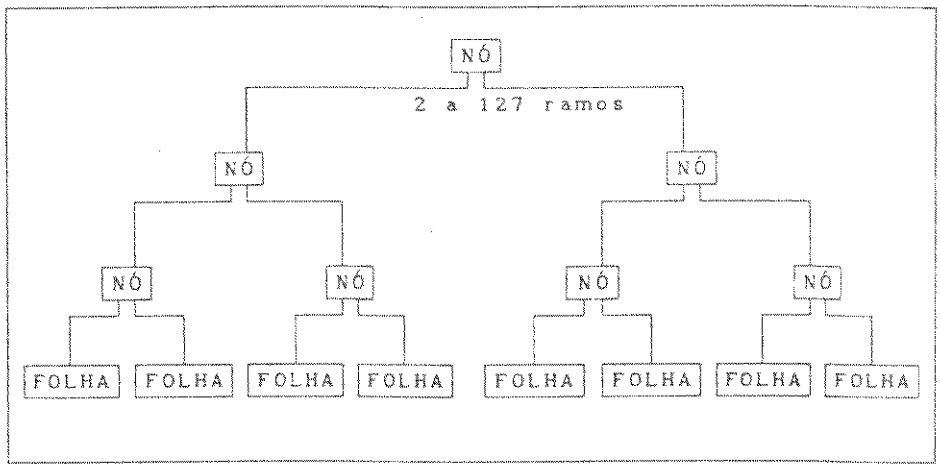


Figura 3.2 : Árvore Balanceada Montada em Prolog

A figura 3.3 ilustra a forma em que os dados são estruturados para a formação das classes.

No SMAC o nome da classe é o nome da árvore, e o nome da instância é o nome da chave sob o qual o dado vai ser armazenado. O dado é uma estrutura formada por um nome e dois argumentos. O nome é comum a todos os dados e serve apenas como aglutinador dos argumentos. O primeiro argumento da estrutura contém o nome do *slot* a quem o segundo argumento está ligado. O segundo argumento pode representar um *slot* filho que, no caso, é especificado por um nome, ou uma lista de atributos. No caso da lista de atributos, sabe-se que o *slot* é o ponto final de uma ramificação, ou seja, é um *slot* folha. A figura 3.4 ilustra várias ramificações possíveis de *slots* e suas respectivas estruturas armazenadas para representá-las.

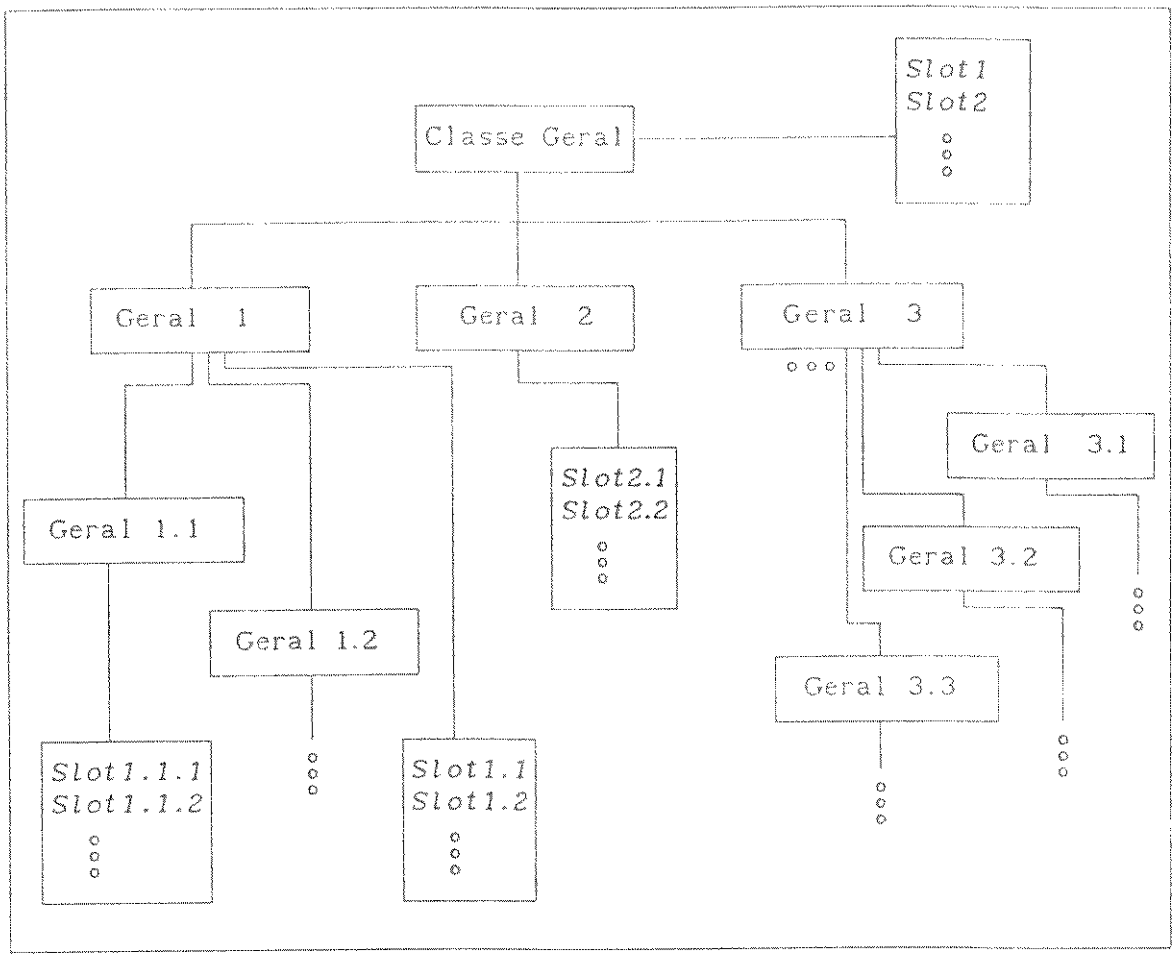


Figura 3.3 : Estrutura das Classes no SMAC

Cada campo sob uma determinada chave dentro da árvore balanceada vai conter uma estrutura do tipo e(arg1,arg2). Isto permite que a manipulação da árvore seja facilitada em virtude do tamanho da estrutura armazenada ser menor. Tanto a busca pela informação se torna mais rápida como também o rebalanceamento, que deve ser feito a cada vez que uma estrutura é inserida ou retirada da árvore.

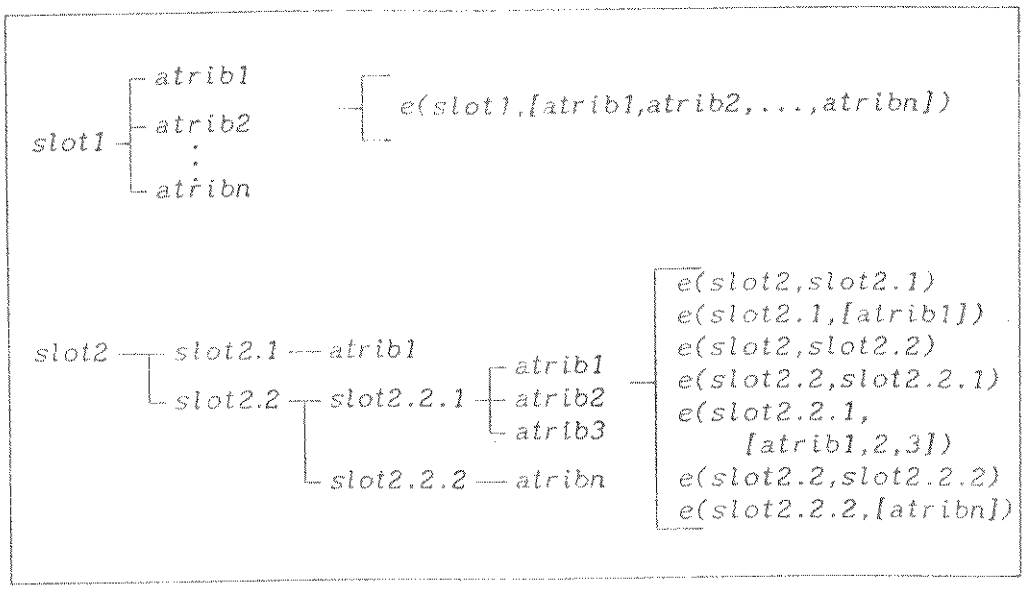


Figura 3.4 : Armazenagem das Árvores de *Slots*

Observando a figura 3.4 verifica-se que a ligação existente entre os nós de uma ramificação é somente no sentido que vai da raiz para as folhas. Além disso, a distinção entre os diferentes *slots* é feita pela nomeação dada a cada um. Dentro de uma mesma instância, os nomes devem ser diferentes. Entre instâncias diferentes isto não é necessário, pois, apesar da árvore balancada ser a mesma, as chaves para a armazenagem são distintas o que transforma cada instância num bloco independente.

Dentro dos *frames* não são armazenados os dados propriamente ditos, mas apenas os atributos que os especificam. Os dados reais são colocados em uma tabela específica para este fim, que será discutida na próxima seção.

Dentro dos *frames* a caracterização dos dados é feita através de um conjunto de atributos. O sistema SMAC utiliza um conjunto definido de atributos que é ilustrado a seguir.

value : define dados explícitos que podem ser fornecidos pelo usuário ou inferidos durante o funcionamento do sistema. Estes dados são reinicializados todas as vezes que se der uma seção de trabalho. A validade destes dados depende do problema que está sendo resolvido.

default : define dados que são utilizados por *default* quando não há disponibilidade de meios para se obter o valor necessário explicitamente. Estes dados, dependendo da configuração dada ao sistema podem ser feitos permanentes ou não.

if_needed : define a existência de um procedimento alocado no *slot*. Este procedimento pode servir para cálculos gerais, obtenção do dado a ser utilizado como valor explícito no *slot*, para definir algum conhecimento não estático embutido no sistema. Este

argumento indica o conhecimento representado via procedimentos dentro dos *frames*.

rule : define a existência de um conjunto de regras alocado no *slot*. Indica o conhecimento representado por regras de produção dentro dos *frames* e que é passível de análise via Sistema Especialista.

Todos os demais atributos ilustrados a seguir são relacionados à proteção dos dados contidos na base de *frames*. Estes procedimentos podem agir da maneira que o programador do sistema achar mais conveniente.

if_read : especifica a existência de um procedimento que efetua alguma proteção quanto a leitura do *slot* ao qual está ligado. Por exemplo, o procedimento pode verificar a procedência da tentativa de leitura e, dependendo do caso, permitir ou não que o dado fique disponível para leitura.

if_write : especifica a existência de um procedimento que efetua alguma proteção quanto a escrita no *slot* em que está alocado. O funcionamento é exatamente igual ao anterior.

if_added : especifica a existência de um procedimento que efetua alguma proteção quanto a inserção de um novo *slot* como filho de algum já existente, criando uma nova ramificação na árvore de *slots*. A adição de novos dados e atributos não é policiada através de atributos dedicados a esta função, pois novos dados e/ou atributos não se constituem em perigo ao bom andamento do sistema. Se for necessário efetivar este tipo de proteção, ela pode ser feita como variação do procedimento contido sob atributo *if_added*.

if_remove : define que o *slot* contém um procedimento de proteção contra remoção. Existem três tipos de remoções possíveis: i) dado, ii) atributo e iii) *slot*. Para cada um dos tipos definidos, existe um atributo específico, a saber:

if_remove_data: para remoção de dados;
if_remove_atrib: para remoção de atributos, e
if_remove_slot: para remoção de *slots*.

Como foi mencionado no capítulo 2, seção 2.3.1, a proteção contra remoção é hierarquizada, ou seja, a proteção a nível de dado se estende também sobre o atributo e o *slot*. Via programação dos procedimentos associados aos atributos é possível desfazer a hierarquia natural do sistema. Esta proteção evita tanto a perda de informação com relação ao que estava armazenado e de que forma, como também a perda de informação contida nas ramificações da árvore de *slots*, se esta existir.

O processo de manipulação de algum dado na árvore de *frames* que define alguma classe, consiste de um procedimento com três argumentos: i) o nome da classe, que indica qual árvore balanceada deve ser manipulada, ii) o nome da instância, que especifica sob que chave a manipulação deve ser feita,

e iii) a estrutura que contém o nome do *slot* folha e os atributos a ele associados. O usuário utiliza os comandos de gerência para efetivar qualquer atividade referenta à base de *frames*. Estes comandos, descritos mais adiante, permitem a facilidade de utilizar argumentos em um nível mais alto, e efetuam, automaticamente e de forma transparente ao usuário, a divisão do que deve ser manipulado nos *frames* e nas tabelas de dados.

A estrutura em árvore não precisa ser criada para depois receber os dados. À medida em que os dados vão sendo inseridos a estrutura vai se formando.

3.2.2.2 - Estrutura dos Dados

Com já foi mencionado anteriormente, os dados não são armazenados diretamente dentro dos *frames*, mas sim em tabelas associadas às classes. Dentre outras vantagens, esse modelo permite que a busca pelos dados seja mais eficiente, pois a manutenção de uma tabela é mais simples e mais rápida do que a de uma árvore balanceada, quando os dados envolvidos são dos mais variados tipos e tamanhos. A separação permite um acesso direto aos dados quando se torna necessária uma manutenção, além de permitir que uma mesma árvore representando uma classe possa ser ligada a varias tabelas de dados diferentes. Dessa forma a mesma estrutura pode servir a vários problemas diferentes.

A tabela implementada no SMAC é do tipo tabela *hash*. Este tipo de tabela organiza os elementos de acordo com a categoria a que pertencem. Ou seja, equivale a construir um arquivo de várias gavetas onde cada uma delas representa uma categoria. A busca se inicia pela procura da categoria e em seguida, dentro da categoria, procura-se o dado desejado. Com isso o escopo da busca fica restrito ao universo da categoria.

Genericamente pode-se dizer que os dados são armazenados de forma direta em um tabela *hash*. Ou seja, cada dado possui uma chave e a sua procura se restringe a achar a chave, o que resulta num acesso direto ao dado com tempo mínimo. Na verdade isto ocorre para uma certa quantidade de dados. A partir de um determinado número limite, a armazenagem passa a ser sequencial. O processo consiste em manipular o dado a ser armazenado através de uma função que vai gerar o endereço de classificação do dado. Esta função é chamada de função *hash*. Esta função é tanto melhor quanto maior for o número de endereços distintos que conseguir fornecer para dados distintos. Haverá uma situação, a partir de uma determinada quantidade de dados armazenados, em que os endereços começam a se duplicar para novos dados. Neste caso, os novos dados são armazenados sequencialmente a partir dos já existentes. Nesta situação, a busca é direta até o endereço e sequencial até o dado. Na situação em que a busca é direta com poucos dados em sequência, a busca numa tabela *hash* é mais eficiente do que a que ocorre numa árvore balanceada. A figura 3.5 mostra graficamente o que foi comentado.

Dentro do sistema SMAC, a tabela de dados associada a cada classe utiliza dois argumentos para localizar dados. O primeiro é o nome da tabela que é o mesmo da classe a qual está ligada, e o segundo é o nome do atributo que caracteriza o dado dentro da classe. Este segundo argumento é a chave que define a categoria do dado dentro da tabela.

Existe uma tabela para cada classe definida e dentro dela são armazenados os dados de todas as instâncias e de todos os *slots*. É necessário então adicionar mais informações ao dado de modo que seja possível identificar, sem ambiguidade, a que *slot* de que instância ele pertence. Para isto, usa-se uma estrutura de dois argumentos que será o "dado" armazenado na tabela. O nome desta estrutura é o nome do *slot* folha a que o dado pertence. O primeiro argumento consiste no nome da instância a que o *slot* pertence e é único. O segundo argumento é o dado propriamente dito. Este dado pode ser de qualquer tipo, numérico ou não.

A figura 3.6 mostra uma estrutura genérica de uma tabela de dados criada no sistema SMAC. Esta figura está associada à figura 3.4 da seção anterior, onde são mostradas algumas ramificações de *slots* e suas estruturas de armazenagem na árvore balanceada.

A figura 3.7 mostra a integração existente entre a árvore de *frames*, que constituem as classes, e as tabelas de dados associados as cada classe.

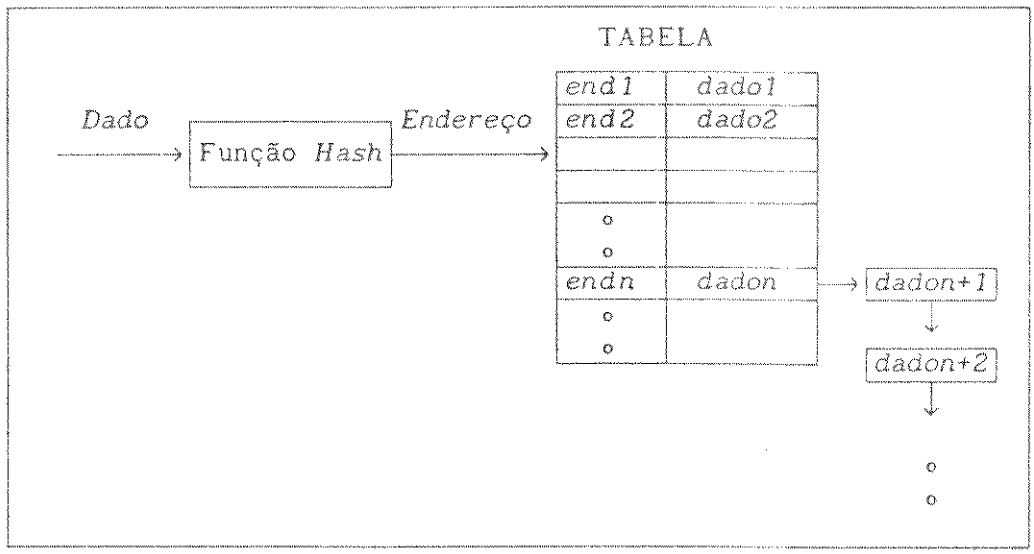


Figura 3.5 : Armazenagem Genérica em uma Tabela *Hash*

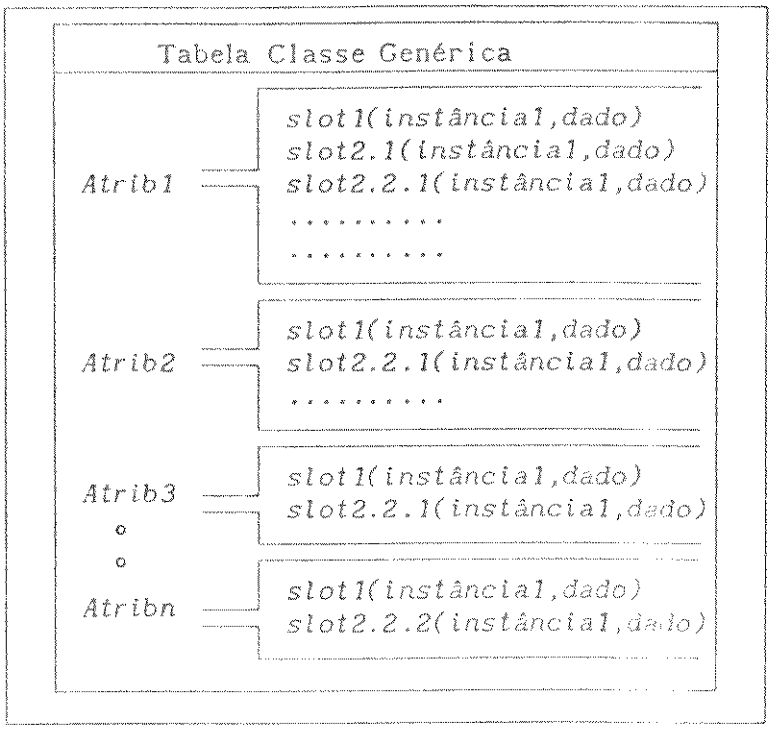


Figura 3.6 : Estrutura de uma Tabela de DadosGenérica

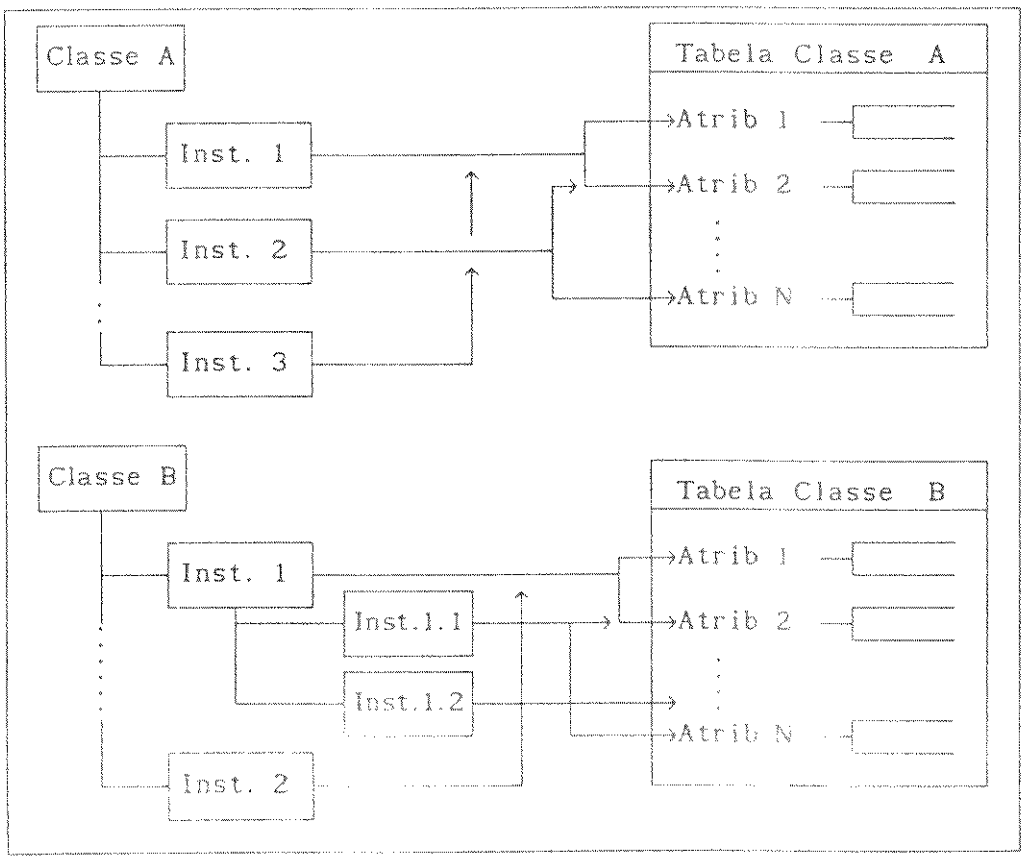


Figura 3.7 : Integração entre Classes e Tabelas de Dados

3.2.2.3 - Estrutura dos Procedimentos

Os procedimentos podem estar ligados aos *frames*, às regras, ou serem disparados diretamente pelo Programa Objeto. O mais comum é alocar os procedimentos, mesmo os que sejam usados diretamente pelo Programa Objeto, em *frames* ou regras. Como será discutido adiante, existem dois modos de trabalho para o SMAC : i) interpretado e ii) compilado. Nos dois casos todos os procedimentos ligados aos *frames* são agrupados no bloco ProcedFR, e os ligados às regras no bloco ProcedBD. Dependendo do modo de trabalho do sistema, estes blocos podem então ser interpretados ou compilados.

Os procedimentos em outras linguagens são excluídos destes blocos quando se trabalha no modo interpretado. Esses procedimentos devem ser compilados e agrupados ao programa principal. Somente as interfaces (explicadas adiante) permanecem interpretadas.

Para uma padronização dos comandos de ativação, todos os procedimentos feitos em outras linguagens que não o Prolog, são interfaceados com o sistema através de um pequeno módulo feito em Prolog. Este módulo tem a função de preparar e ativar o procedimento externo. Com isso todas as chamadas a procedimentos são efetuadas como se todos fossem procedimentos escritos na mesma linguagem do sistema principal. Ou seja, a ativação se torna independente da linguagem em que o procedimento esteja escrito. Esses módulos são denominados de Interfaces.

A figura 3.8 mostra a relação entre sistema, interfaces e procedimentos.

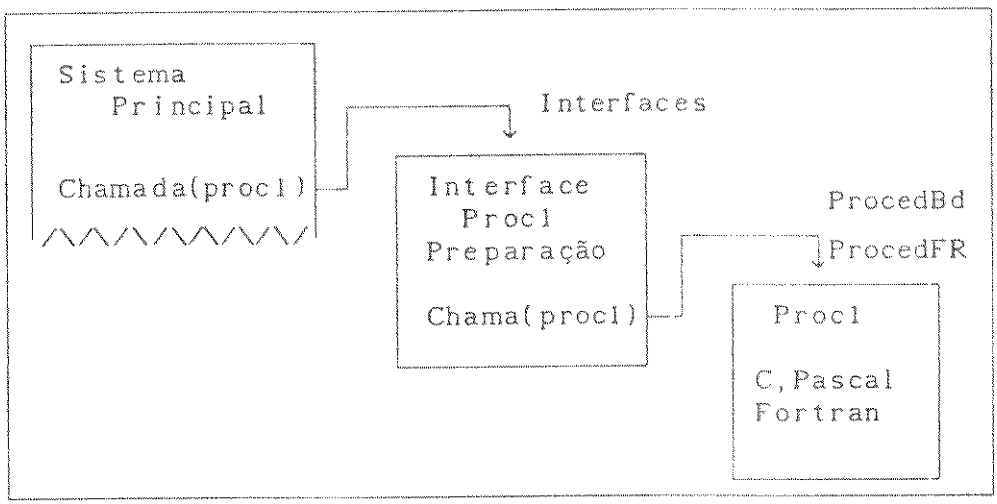


Figura 3.8 : Chamada de Procedimentos em outras Linguagens

A armazenagem de um indicador de procedimento é feita utilizando uma estrutura de lista, seja para os ligados aos *frames* como para os ligados às regras. No caso das regras, esta estrutura se encontra no cabeçalho da base de

regras (ver próxima seção). A lista que representa um procedimento genérico é detalhada a seguir.

Procedimento = [*nome_proced,janela,Arg1,....,Argn*] .

A definição de cada elemento da lista é:

nome_proced : define o nome do procedimento a ser chamado.
janela : estabelece se a janela padrão que o gerenciador utiliza para cada procedimento deve ser ativada ou não. Não é passado como argumento quando da ativação do procedimento. Assume o valor *habilitado* ou *desabilitado*.
Arg1,..,Argn : define a lista de argumentos que deve ser passada ao procedimento quando de sua ativação.

O gerenciador é o responsável pela ativação dos procedimentos, sejam eles alocados nos *frames* ou nas regras. O gerenciador dispõe de comandos específicos para cada caso, porém a estrutura de armazenagem é a mesma para ambos os casos.

Os procedimentos, quando acionados, assumem o controle temporário do sistema e podem definir interfaces de comunicação próprias, não sendo obrigatório o uso da interface existente no sistema.

3.2.2.4 - Estrutura dos Conjuntos de Regras

No capítulo 2 foi apresentada a estrutura lógica do sistema, onde o fato de que as regras de produção estão agrupadas em conjuntos segundo critérios definidos pelo programador do sistema foi abordado. Cada um destes conjuntos é subdividido em várias bases de regras que agrupam o conhecimento dominado pelo conjunto em partes menores e mais específicas.

Cada conjunto tem um pequeno programa associado a ele que é denominado **Programa Chamador do Conjunto**. Este programa é executado antes que o conjunto seja carregado na memória de trabalho. Nele está contido o comando, padronizado para todos os chamadores, que define os operadores diferentes dos pré-definidos pelo Sistema Especialista do SMAC.

O chamador deve ser executado antes para que os operadores, caso existam, possam ser definidos para o sistema e, quando o conjunto de regras for carregado para a memória de trabalho, o SE seja capaz de compreender corretamente a sintaxe das regras que serão utilizadas nas futuras inferências.

O comando que especifica os operadores é estruturado na forma ilustrada a seguir.

operadores([tipo,operador,tipo,operador,....,..]).

O argumento do comando é uma lista que define todos os operadores novos juntamente com o tipo de associatividade e posição estabelecido para eles. O tipo dos operadores pode ser (f=operador; x,y=argumento):

fx : operador do tipo prefixo e não associativo.
fy : operador do tipo prefixo e associativo da esquerda para direita.
xf : operador do tipo pós-fixo e não associativo.
yf : operador do tipo pós-fixo e associativo da direita para a esquerda.
xfx : operador do tipo infixo e não associativo.
xfy : operador do tipo infixo e associativo da direita para a esquerda.

yfx : operador do tipo infixo e associativo da esquerda para a direita.

O operador é dito associativo da direita para a esquerda quando os argumentos mais a direita da expressão são agrupados primeiro. É da esquerda para a direita quando os argumentos mais a esquerda são agrupados primeiro. Quando é do tipo não associativo, significa que um outro operador de mesma precedência não pode ocorrer dentro da mesma expressão. Os operadores de maior precedência são os que têm seus termos agrupados por último. Não importando a notação usada, internamente, o ambiente Prolog transforma todas as expressões para a notação pré-fixa antes de interpretar a expressão.

A figura 3.9 mostra um operador genérico op definido de várias formas e sendo transformado para a notação pré-fixa onde fica fácil identificar o efeito da associatividade. Vale observar que, na notação pré-fixa, os parêntesis mais internos são resolvidos antes que os mais externos.

A figura 3.10 mostra, a partir de um grupo de operadores genéricos, como as regras, e consequentemente os fatos, são colocados na estrutura pré-fixa para serem utilizados pelo SE. Esta alteração faz uso da precedência e do tipo de associatividade do operador.

Os operadores são manipulados de acordo com o grau de precedência que possuem. Os de maior precedência são agrupados primeiro. Dessa forma, os operadores que definem a estrutura das regras, como se e então, são os de mais alta precedência. Aqueles que estruturam as partes de <condição> e <ação> são os de mais baixa precedência. Os operadores que se permite alterar são justamente os de mais baixa precedência para evitar problemas quanto a interpretação da sintaxe das regras por parte do SE.

A precedência dos novos operadores é fixada internamente pelo próprio sistema, deixando a possibilidade do programador definir apenas o nome do operador e seu tipo. Com isso, evita-se que o programador defina, desavisadamente, operadores cuja precedência seja maior da estipulada para os que estruturam as regras, por exemplo, e comprometer o funcionamento do SE.

O comando que define os operadores é único para todas as bases contidas num conjunto de regras, pois os operadores são definidos sobre o global do conhecimento representado.

```
op ⟶ yfx

    a op b op c ⟶ op(op(a,b),c)
op ⟶ xfy

    a op b op c ⟶ op(a,op(b,c))
op ⟶ fx

    op(a....b) ⟶ op(a....b)
op ⟶ xf

    (a....b)op ⟶ op(a....b)
op ⟶ fy

    a op b op c ⟶ op(a(op(b(op c))))
op ⟶ yf
    a op b op c ⟶ op(c(op(b(op a))))
```

Figura 3.9 : Associatividade dos Operadores

Para definir os procedimentos que estão contidos no conjunto, utiliza-se um comando que estabelece uma relação entre as metas das regras e os procedimentos a elas associados. Isso permite escrever as regras sem perder o poder de expressão e entendimento de uma linguagem coloquial para descrever as metas. Cada base de regras, dentro de um conjunto, tem o prórpio comando que define os seus procedimentos. A estrutura do comando é dada a seguir.

```
executar(ind,Meta,Proc) ⟶
            ( meta1 ⟶ [proc,hab,arg1,...,argn] ,
              meta2 ⟶ [proc,des,arg1,...,argm] ,
                .
                .
                .
              metai ⟶ [proc,hab,arg1,...,argy] ;
              nao_atrib).
```

Os argumentos do comando especificam o seguinte:

ind : é o indicador que especifica a base de regras dentro do conjunto.

Meta : especifica a meta para a qual o procedimento deve ser encontrado.

Proc : é um argumento de saída que fornece a estrutura em forma de lista que define o procedimento associado à meta fornecida no argumento anterior.

operador	tipo	precedência
:	xfx	900
entao	xfx	880
se	fx	870
e	xfy	540
eh	xfx	100
ind	xfx	50

```
regra1 ind basea : se folha eh verde
                     e caule eh marrom
                     e raiz eh profunda
                   então planta eh árvore.
                          ↓
:(regra1 ind basea,se folha eh verde e caule eh marrom e raiz
        eh profunda então planta eh árvore).
                          ↓
:(regra1 ind basea,então(folha eh verde e caule eh marrom e
        raiz eh profunda,planta eh árvore)).
                          ↓
:(regra1 ind basea,então(se(e(folha eh verde,e(caule eh marrom,
        raiz eh profunda))),planta eh árvore)).
                          ↓
:(ind(regra1,basea),então(se(e(eh(folha,verde),e(eh(caule,marrom),
        eh(raiz,profunda)))),eh(planta,árvore))).
```

Figura 3.10 : Notação Pré-fixa para Regras no SMAC

Na execução do comando *executar*, caso o procedimento não seja encontrado, o argumento *Proc* tem como valor *nao_atrib* que é passado para o SE como resposta. Este, por sua vez, ativa uma janela de erro que previne o usuário sobre a existência de uma meta que não consegue chamar o procedimento associado, e indica em qual arquivo provavelmente está o erro. Em seguida os trabalhos são paralisados para que os reparos sejam feitos.

Cada base de regras dentro do conjunto contém todos os integrantes discutidos na seção 2.3.2 do capítulo 2, onde a representação do conhecimento usando regras é discutida. Ou seja, cada base contém regras, fatos, fatos

temporários e a tabela para definição dos procedimentos. Durante a inferência, cada base de regra é utilizada independentemente uma da outra.

Cada base é identificada através de um indicador que especifica as regras, os fatos (temporários ou não) e a tabela de procedimentos que constituem esta base. As estruturas citadas têm o formato ilustrado a seguir.

<regra> : regra1 ind basea : se <condição> então <ação>.

<fato> : fato1 ind basea : <corpo do fato>.

<fato temporário> : foi_dito(basea,Meta,Modo).

<tabela de Procedimentos> : executar(basea,Meta,Proc) → ..

Na definição das regras e dos fatos existe o operador *ind* que especifica que a regra ou o fato possuem como indicador o termo *basea*.

Na estrutura do fato temporário, o argumento *Meta* especifica a conclusão obtida e que passa a ser verdade durante o tempo em que a inferência estiver ativa ou até ser modificada. O argumento *Modo* define qual a maneira pela qual a conclusão foi obtida, ou seja, através do usuário (*user*), através de uma regra (nome da regra), ou através de um procedimento (nome do procedimento).

A figura 3.11 apresenta uma visão esquemática de como se apresenta a estrutura de um conjunto de regras genérico dentro do sistema SMAC.

É importante observar que, na figura 3.11, o conjunto é representado dentro da memória de trabalho do sistema. Por isso aparecem os fatos temporários. Na armazenagem em arquivo estes fatos não existem, pois o tempo de vida deles é o tempo de duração da inferência na qual foram concluídos.

3.2.3 - Configuração Física do Sistema SMAC

O sistema SMAC se apresenta em dois modos: i) compilado, e ii) interpretado. O primeiro serve para as aplicações já desenvolvidas e testadas, das quais não há dúvidas quanto ao funcionamento. É a forma mais eficiente e rápida de utilizar o sistema SMAC. O segundo modo é bem menos eficiente em termos de velocidade, porém tem uma flexibilidade muito maior em relação a alteração do código dos procedimentos, do Programa Objeto e dos conjuntos de regras. Esta flexibilidade é essencial quando do desenvolvimento de novos sistemas.

A parte do sistema SMAC que não pode sofrer alteração por parte do usuário, é fornecida como um bloco compilado que deve ser agrupado aos demais blocos montados pelo usuário para formar um programa executável. Os conjuntos de regras permanecem interpretados, para permitir que possam ser alterados e

que novos conjuntos possam ser agregados ao sistema sem que nenhuma alteração estrutural seja necessária. No capitulo 4, são discutidas algumas idéias para aumentar a eficiência do sistema sem, no entanto, perder a versatilidade. Os procedimentos podem ser interpretados ou compilados junto ao programa principal, exceto aqueles feitos em linguagens diferentes do Prolog, que deverão ser obrigatoriamente compilados. O Programa Objeto também pode ser interpretado ou compilado, dependendo da necessidade do programador. O ideal é compilar as partes fixas e trabalhar com as partes sujeitas a alterações frequentes na forma interpretada.

```
    operadores([tipo,op1,....,tipo,opn])
-------------------------------------------------------------
base a
     regra1 ind basea : se <cond> então <ação>.
                :                  :
     regran ind basea : se <cond> então <ação>.

     fato1 ind basea  : <corpo>.
                :
     faton ind basea  : <corpo>.

     foi_dito(basa,< >,< >).
                :
     foi_dito(basa,< >,< >).

     executar(basea,Meta,Proc) ⟶
                    ( meta1  ⟶ [proc,<>,arg0,..,argn],
                        :
                      metan  ⟶ .....).
-------------------------------------------------------------
base b
      regra1 ind baseb : .......
         :
-------------------------------------------------------------
base n
        :
```

Figura 3.11 : Conjunto de Regra na Mem. de Trabalho do SMAC

Dependendo da situação, pode-se particionar o Programa Objeto em uma parte compilada e outra interpretada. Isto pode ser usado também para os procedimentos, porém, o mais comum, é trabalhar com procedimentos totalmente compilados ou totalmente interpretados.

As figuras 3.12 e 3.13 mostram uma visão, em diagramas de blocos, do sistema SMAC compilado e interpretado.

A nomenclatura *ProcedFRI* indica que os procedimentos agrupados neste módulo são interpretados, o mesmo acontecendo com o *ProcedBDI*. O Programa Objeto não apresenta uma nomenclatura específica, pois ele é considerado sempre interpretado na fase inicial da configuração do SMAC para atender alguma aplicação. Mesmo quando for estruturado o sistema compilado, estes arquivos interpretados devem estar presentes no ambiente mesmo que vazios. Isso permite atualização imediata do sistema compilado através da inserção de novos módulos, para posterior validação e incorporação definitiva ao sistema.

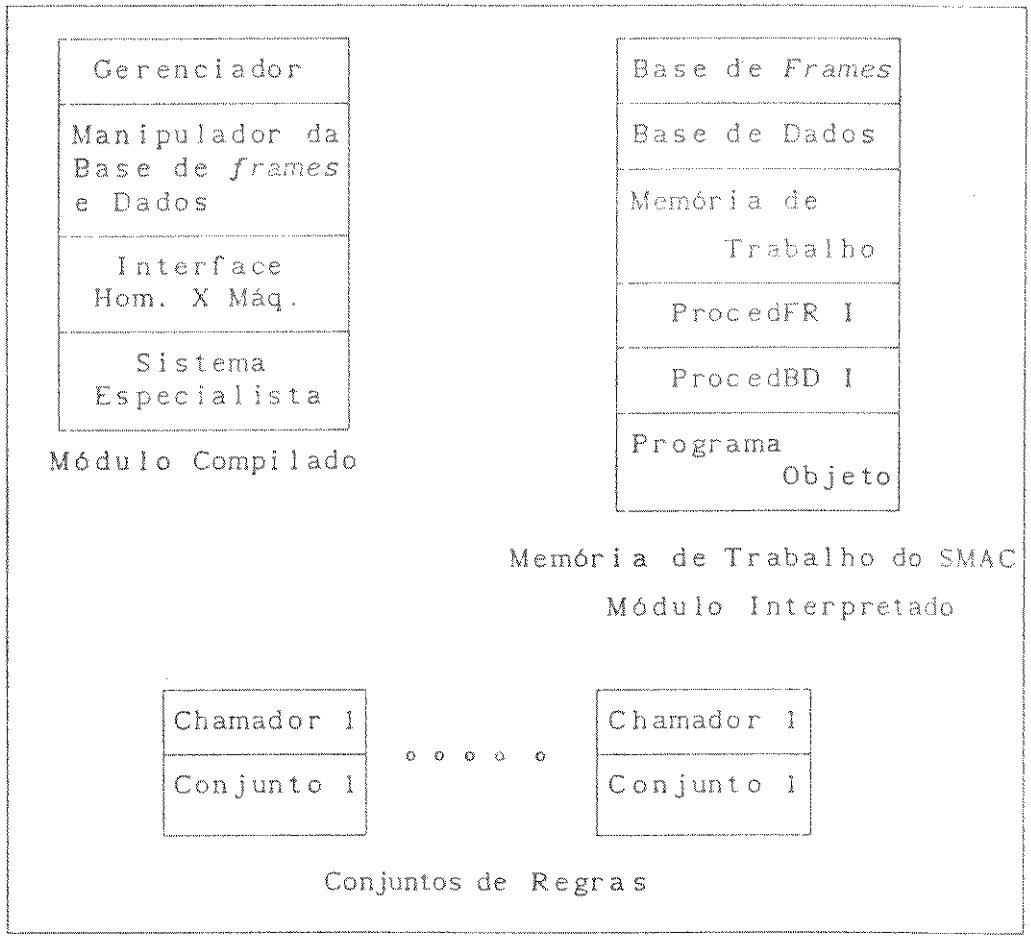


Figura 3.12 : O Sistema Smac Interpretado

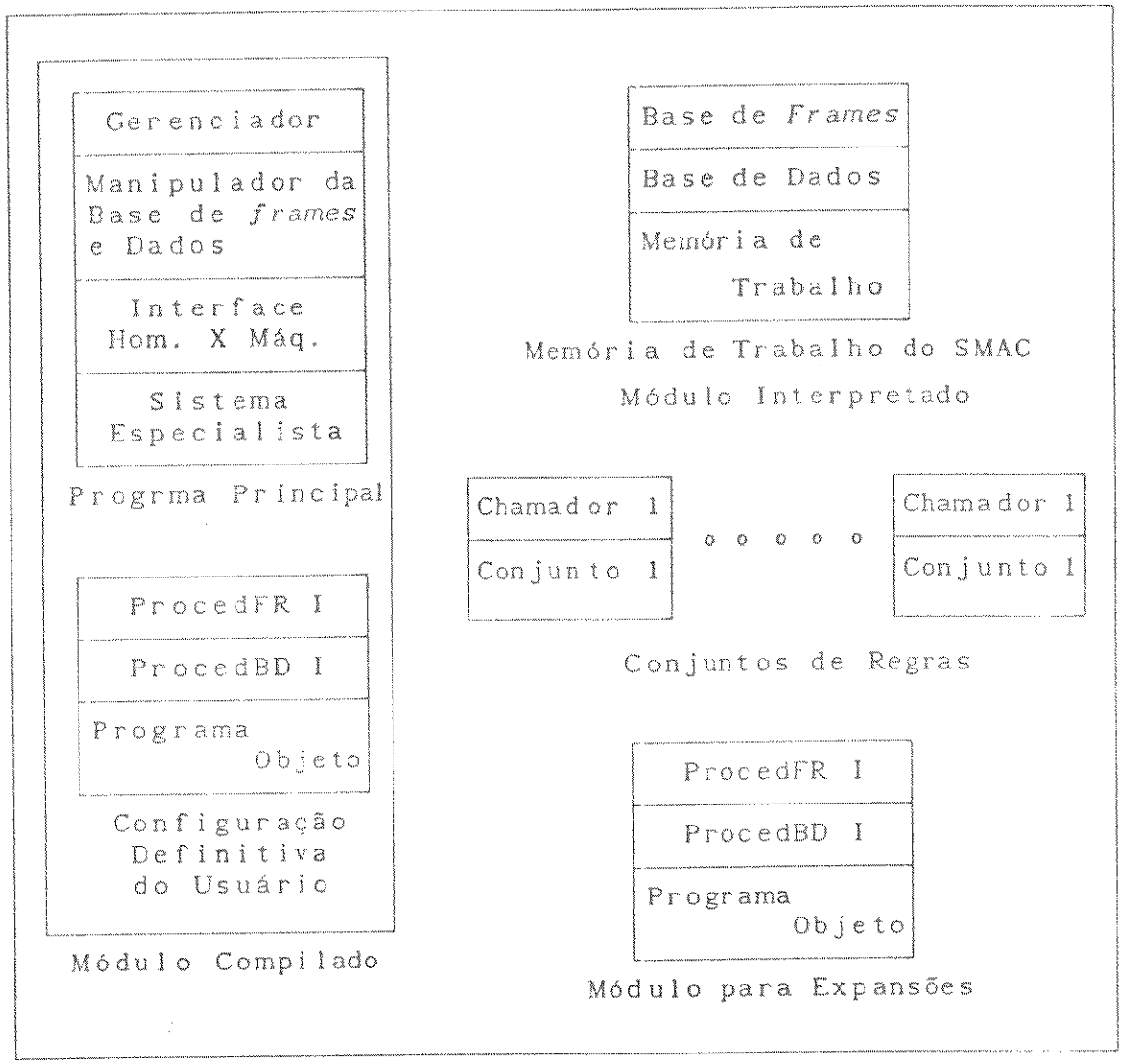


Figura 3.13 : O sistema SMAC Compilado

3.3 - Métodos e Procedimentos

Como foi mencionado anteriormente, o sistema SMAC é constituido por blocos independentes que se comunicam entre si somente através do gerenciador. Nesta seção são apresentados os comandos específicos de cada bloco acompanhados de uma descrição de seu funcionamento e detalhes de implementação. Os comandos serão discutidos integralmente independentemente do fato de serem acessíveis ou não, de modo direto, ao usuário programador do sistema.

3.3.1 - Manipulação de *Frames* e Dados

Todos os procedimentos deste bloco manipulam com os *frames* e com os dados sem nenhuma preocupação com a integridade ou fatores de proteção. Estes comandos são acessados somente pelo gerenciador, não sendo possível ao usuário fazer uso direto dos mesmos.

3.3.1.1 - Comandos do Nível Físico

Estes comandos são responsáveis pela manipulação das estruturas representativas dos *frames*, *slots*, classes, instâncias e dados nas estruturas de árvores balanceadas e tabelas associadas de dados.

A figura 3.14 descreve os argumentos utilizados nos comando de nível físico.

1) **put_in**

Armazena as estruturas que definem os *slots* na árvore balanceada, estruturando as classes e as instâncias desejadas. É responsável pela armazenagem das estruturas que definem as ramificações, dos ramos folhas e dos dados na tabela associada à classe.

put_in(*Frame,Instância,Slotpai,Slot*)

Frame	⟶	Define a classe
Instância	⟶	Define a sub-classe ou instância
Slotpai	⟶	Pai do *slot* definido no argumento seguinte se uma cadeia de *slots* for definida.
Slot	⟶	Define o *slot* sobre o qual a manipulação deve ser feita.
Atrib	⟶	Define o(s) atributo(s) para manipulação
Struct	⟶	Define a(s) estrutura(s) armazenadas na tabela associada de dados

Figura 3.14 : Argumentos dos Comandos de Nivel Fisico

Armazena a estrutura *e(Slotpai,Slot)* que estabelece as ramificações intermediárias dos *slots* na classe *Frame*, na instância *Instância*.

Só atua sobre a estrutura da árvore balanceada.

put_in(*Frame,Instância,Slot,Atrib,Struct*)

Armazena os ramos folha nas árvores balanceadas e os dados nas tabelas associadas. *Atrib* é alista de atributos e *Struct* é a lista de dados ordenada segundo a lista de atributos como ilustra a figura 3.15.

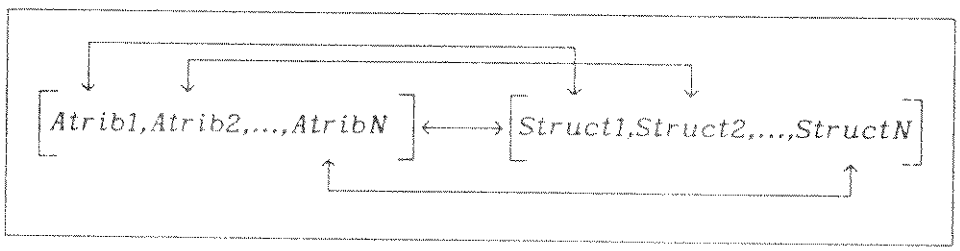


Figura 3.15 : Argumentos *Atrib* e *Struct* do Comando put_in

2)put_out

Realiza a leitura de um *slot* determinado e seus dados.

put_out(*Frame,Instância,Slotpai,Slot*)

Responsável pela leitura dos ramos intermediários das ramificações de *slots*. Sua atuação se restringe a árvore balanceada que estrutura a classe. Serve para determinar as ligações existentes entre os vários *slots*.

put_out(*Frame,Instância,Slot,[Atrib],Struct*)

Responsável pela leitura dos dados na tabela de dados associada à classe *Frame*. A leitura se dá sob o atributo *Atrib* do *slot* definido pelo argumento *Slot*.

A utilização da estrutura *[Atrib]* como argumento, acontece por necessidade de padronização com o comando a nível de classe que faz uso deste comando put_out.

3)erase

Apaga *slots*, suas ramificações e seus dados. Atua tanto na árvore balanceada como na tabela de dados associada à classe cujo o comando se refere.

erase(*Frame,Instância,Slotpai,Slot*)

Responsável pela retirada dos ramos intermediários de uma ramificação de *slots* contidos na classe *Frame*. Só atua sobre a árvore balanceada.

Este comando não realiza uma das caracterísiticas básicas do Prolog que é o *backtrack*. Isto evita que um comando inconsequente possa apagar dados cuja perda seja irreparável.

erase(*Frame,Instância,Slot,Atrib,Struct*)

Apaga os dados aramazenados no *slot* indicado pelo argumento *Slot*. O argumento *Atrib* pode ser dado sob forma de uma lista de atributos. O argumento

Struct fornece uma lista, ordenada segundo a lista de atributos, dos dados apagados. Só atua sobre a tabela associada de dados.

4)put_in_h

Atua somente sobre a tabela associada de dados, permitindo alterar os dados contidos nesta tabela, mesmo quando não houve nenhuma mudança na árvore balanceada que especifica a classe.

Para evitar duplicação de informação dentro da tabela, este comando deve ser usado conjuntamente com o comando **erase**.

A alteração só é efetuada se o *slot* desejado realmente existir dentro da classe indicada.

put_in_h(*Frame,Instância,Slot,Atrib,Struct*)

Os argumentos *Atrib* e *Struct* são iguais aos definidos no comando erase.

3.3.1.2 - Comandos do Nível de Classe e Sub-Classe

Estes comandos formam um nível intermediário entre os comandos de gerência que efetuam a manipulação com *frames* e dados, e os comandos de nível físico discutidos na seção anterior.

Este nível facilita o uso dos comandos do nível físico pois permite uma manipulação completa da árvore balanceada e da tabela associada de dados em comandos únicos, que fornecem respostas de acordo com o resultado da manipulação efetuada.

3.3.1.2.1 - Comandos Básicos

1)**fput** : Comando para criação de classe e sub-classe.

Este comando tem o objetivo de criar classes e sub-classes que constituirão a base de conhecimento declarativo e procedural onde o sistema vai atuar. Ele atua no sentido de criar todas as ramificações intermediárias e as folhas das árvores balanceadas, como também criar o espaço para armazenagem dos dados nas tabelas associadas a cada classe criada.

A formação das árvores de *slots* que definem os *frames* pode se dar de duas formas: i) *slot* convencional que é preenchido diretamente por um dado , ou ii) *slot* encadeado que estabelece uma ramificação que pode levar a várias folhas com vários dados.

A figura 3.16 mostra um exemplo genérico de uma ramificação de *frames* na sequência em que esta é criada.

Cada linha (1,2,3,4 e 5) na figura 3.16 indica um comando **fput**. Cada comando é responsável por uma ramificação do seu ponto inicial até o ramo folha.

fput(*Frame,Instância,Slotpai,Slot,Atrib,Struct*)

A figura 3.17 mostra a descrição dos argumentos utilizados pelo comando.

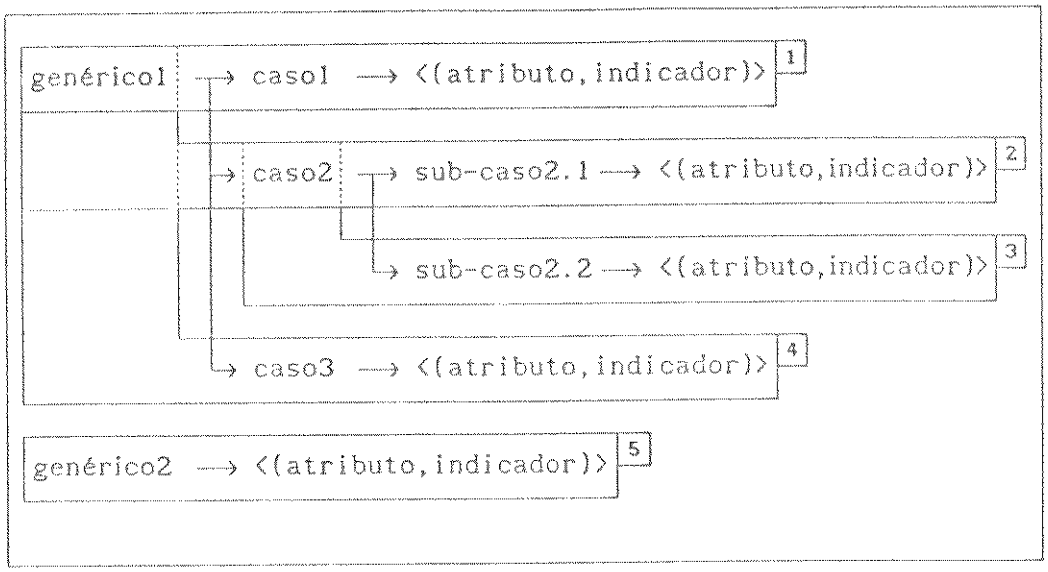


Figura 3.16 : Sequência de Criação de Ramificção em *Slots*

Frame	⟶	Indicador de Classe
Instância	⟶	Indicador de Sub-classe
Slotpai	⟶	*Slot* raiz da ramificação
Slot	⟶	Lista de *slots* da raiz até a folha
Atrib	⟶	Lista de atributos do *slot* folha
Struct	⟶	Lista de dados contidos no *slot* folha

Figura 3.17 : Argumentos do Comando **fput**

A relação entre os argumentos *Atrib* e *Struct* é a mesma ilustrada na figura 3.15.

Sempre que o comando é ativado, se o *slot* não existir ele é criado. No caso de atributo ou dado, a inexistência causa uma inserção no *slot* indicado. A inserção ou criação só são realmente efetuadas se o que o comando tenta efetuar é novidade na base de *frames* e dados.

2)**fget** : Comando para leitura de classe e sub-classe.

Este comando permite fazer leituras nas classes e sub-classes criadas com o comando anterior. Permite a utilização do recurso de *backtrack* para o fornecimento de todo o conjunto de informações que cujo fator comum

esteja especificado na lista de argumentos do comando.

A figura 3.18 mostra a descrição dos argumentos utilizados no comando.

fget(Frame,Instância,Slotpai,Slot,Atrib,Struct)

Frame	⟶	Indicador de Classe
Instância	⟶	Indicador de Sub-classe
Slotpai	⟶	Pai do *slot* desejado
Slot	⟶	*Slot* desejado para leitura
Atrib	⟶	Lista de atributos para pesquisa
Struct	⟶	Lista de dados/respostas obtidos

Figura 3.18 : Argumentos do Comando fget

A pesquisa para a leitura sempre é iniciada na árvore balanceada, e se estende até a tabela associada de dados somente se algum *slot* folha pertencer ao escopo da busca.

As respostas relativas ao funcionamento do comando são dadas através do argumento *struct*. As possibilidades são:

Struct = [*não_atrib*] → Caso não haja informação suficiente para encontrar o ramo desejado. Ou o *slot* não existe ou os argumentos *Frame* e *Instância* não foram fornecidos.

Struct = [*não_especif*] → O *slot* solicitado para leitura não está especificado na classe e instânciafornecidas no comando.

Caso a leitura seja pedida em mais de um dos atributos existentes ou em todos eles, o argumento *Struct* retorna uma lista que informa as condições existentes sob cada atributo pedido. Seja o exemplo:

Strct = [dado1,[],dado2,não_atrib]

Neste caso, o dado que existe sob o primeiro atributo e o terceiro é fornecido normalmente. No segundo atributo é informado que não existe dado armazenado. O quarto atributo especificado no comando não está definido para o *slot* requisitado.

3)**fremove** : Comando para remoção de dados, atributos e *slots*.

Este comando permite retirar dados, atributos e *slots*. A sua ação se inicia na árvore balanceada e se estende à tabela de dados associada à classe. O comando pode ser dividido em duas maneiras específicas de funcionamento: i) trabalhando com *slots* e ii) trabalhando com dados.

i) Trabalhando com *slots* : As remoções são efetuadas somente nos ramos terminais. Isto significa a eliminação do espaço contido na tabela de dados e na eliminação da estrutura representativa do *slot* da árvore balanceada, onde estão descritos os atributos. Quando da retirada de uma terminação na árvore balanceada, toda ramificação referente ao escopo dos parentes da terminação é analisada e re-arranjada de forma que sempre haja uma ramificação terminal em cada ramo da árvore de *slots* atingida pelo comando de remoção. O comando só é liberado para agir se não existirem nem dados e nem atributos especificados no *slot*.

ii) Trabalhando com dados : Os dados são retirados a partir da especificação do *slot* e do atributo onde ele se encontra. Além da retirada do dado permite-se retirar o atributo. Porém, isso só é possível se o dado que estiver armazenado tendo este atributo como indicador, já tiver sido retirado.

fremove(*Frame,Instância,Slot,Atrib,Struct,Proced*)

A figura 3.19 mostra a descrição dos argumentos utilizados no comando.

Frame	⟶	Indicador de Classe
Instância	⟶	Indicador de Sub-classe
Slot	⟶	*Slot* desjado para operação
Atrib	⟶	Lista de atributos sob os quais os dados devem ser eliminados ou lista de atributos a eliminar
Struct	⟶	Lista de dados/respostas obtidos
Proced	⟶	Define se o comando age sobre dados ou sobre atributos

Figura3.19:ArgumentosdoComando**fremove**

Para retirar dados :

i) *Proced = data* e *Atrib = [*]* : Retira dados e todos os atributos alocados no *slot*.

ii) *Proced = data* e *Atrib = Lista* : Retira dados sob os atributos descritos em *Lista*. No argumento *Struct* é fornecida uma lista ordenada segundo o argumento *Lista*. Para os atributos que não existirem é retornado o valor *não_atrib*, e quando não há dado a ser retirado é retornado o valor *[]*.

Para retirar atributos :

i) *Proced = atrib* e *Atrib = [*]* : Retira todos os atributos existentes no *slot*. No argumento *Struct* é fornecida uma lista ordenada pela sequência dos atributos presentes, contendo *ok* para os atributos retirados e *nok* para aqueles que não foram retirados por ainda conterem dados.

ii) *Proced = atrib* e *Atrib = Lista* : Retira os atributos especificados em *Lista*. No argumento *Struct* é fornecida uma lista ordenada segundo o argumento *Lista* contendo *ok* para os atributos retirados, *nok* para aqueles que continham dados e não foram retirados, e *não_atrib* para os atributos não definidos no *slot*.

Quando são retirados todos os atributos, automaticamente, o *slot* é retirado da árvore balanceada.

Desde de que tenham sido retirados todos os dados, todos os atributos e todos os *slots*, a instância também pode ser retirada da base de *frames*.

3.3.1.2.2 - Comandos Auxiliares

Estes comandos são especificados unicamente para operações de leitura. Sendo os atributos *value*, *default* e *if_needed* os mais frequentemente procurados, esses comandos fornecem algumas facilidades com referência a estes atributos.

1)**fget_v_d** : Comando de leitura automática para atributo *value* e *if_needed*.

Este comando executa a leitura de um *slot* procurando por dados sob os atributos *value* ou *default*, sendo a prioridade maior dada ao primeiro.

A figura 3.20 mostra os argumentos específicos para o comando.

fget_v_d(*Frame,Instância,Slotpai,Slot,Atrib,Struct*)

Frame	⟶	Indicador de Classe
Instância	⟶	Indicador de Sub-classe
Slotpai	⟶	Pai do *slot* desejado
Slot	⟶	*Slot* desejado para leitura
Atrib	⟶	Lista de atributos para pesquisa
Struct	⟶	Lista de dados/respostas obtidos

Figura3.20:ArgumentosdoComandofget_v_d

2)fget_v_d_p : Comando de leitura automática para atributo *vaule*, *default* e *if_needed*.

Este comando executa a leitura de um *slot* procurando por dados sob os atributos *value*, *default* e *if_needed*. A prioridade segue a ordem na qual os atributos forma descritos.

No caso do argumento *if_needed*, que é o último em prioridade, o procedimento, quando encontrado, não é diretamente executado. A sua referência é retornada e poderá ser executado por um comando de gerência específico para este fim a ser acionado por quem iniciou a procura.

fget_v_d_p(*Frame,Instância,Slotpai,Slot,Atrib,Struct*)

Os argumentos do comando são os mesmos ilustrados pela figura 3.20.

3.3.1.2.3 - Comandos para Manipulação de Herança

A definição de uma classe permite a criação de uma quantidade ilimitada (a não ser por restrições técnicas da máquina) de sub-classes interligadas entre si através de laços tipo pai-filho. Esta ligação é especificada através dos *slots* que fazem parte do cabeçalho de cada sub-classe e são: *a_kind_of* que define os pais e *members* que define os filhos.

Como a estrutura que define uma classe e suas sub-classes é uma árvore, os ramos folhas herdam todas as informações contidas nos níveis superiores a eles ligados. No caso de uma determinada informação que está contida na raiz ser redefinida em algum nó ou nível seguinte, toda a estrutura dependente deste nó ou nível herdará a informação re-definida, não tendo acesso às definições feitas em níveis superiores.

A herança visa a possibilidade de obter-se dados que não estejam explicitamente definidos dentro da própia instância. Desta forma, a busca sempre se inicia na própria instância e depois se propaga pela árvore dependendo da profundidade especificada e do modo de funcionamento escolhido.

A busca em herança pode ser especificada de duas maneiras: i) em amplitude e ii) em profundidade.

A busca em amplitude efetua uma busca nível a nível até a profundidade especificada. Esta profundidade é indicada em número de níveis a visitar.

A busca em profundidade efetua uma busca por cada ramo possível, do ponto de partida ao ponto que corresponda à profundidade especificada. Esta profundidade é indicada em número de instâncias a visitar.

Em ambos os tipos de busca existem mecanismos para especificar a profundidade máxima (até a raiz) diretamente. Também são definidos mecanismos para evitar busca repetidas em nós já visitados anteriormente. Esse fato ocorre quando as instâncias são definidas tendo vários pais.

1)**fget_deep** : Comando para busca em profundidade.

fget_deep(*Frame,Instância,Slotpai,Slot,Atrib,Struct,Deep,*
Atrib_or,Inst_or,Path).

A figura 3.21 ilustra os argumentos utilizados por este argumento.

Frame	⟶	Indicador da Classe que será escopo da busca
Instância	⟶	Indicador de Sub-classe de parte a busca
Slotpai	⟶	Pai do *slot* desejado
Slot	⟶	*Slot* desjado para leitura
Atrib	⟶	Lista que define sob qual ou quais atributos o dado deve ser procurado
Struct	⟶	Retorna o dado encontrado
Atrib_or	⟶	Retorna o atributo sob o qual o dado foi encontrado
Instan_or	⟶	Retorna a instância onde o dado foi encontrado
Deep	⟶	Define a profundidade da busca
Path	⟶	Retorna o caminho percorrido pela busca

Figura 3.21 : Argumentos do Comando **fget_deep**

O argumento *Deep* define o número de instâncias pesquisadas em cada caminho. Se for *zero* a busca é executada apenas na instância de partida. Se for *raiz* a busca é executada até a raiz da árvore, ou seja, a busca assume a maior profundidade possível.

O argumento *Path* armazena o caminho executado pela busca ao longo do tempo. É utilizado pelo próprio comando para evitar que haja repetição de visitas em um mesmo nó.

O caminho a ser seguido pela busca é estabelecido pela formação de uma lista de pais (LP) a cada ponto. Esta lista é obtida no *slot a_kind_of* presente no cabeçalho de cada instância. A ordem em que esta lista é formada se traduz na ordem em que as visitas serão feitas. Neste tipo de busca, as últimas LPs criadas são as que primeiro serão esgotadas, ou seja, terão suas instâncias visitadas.

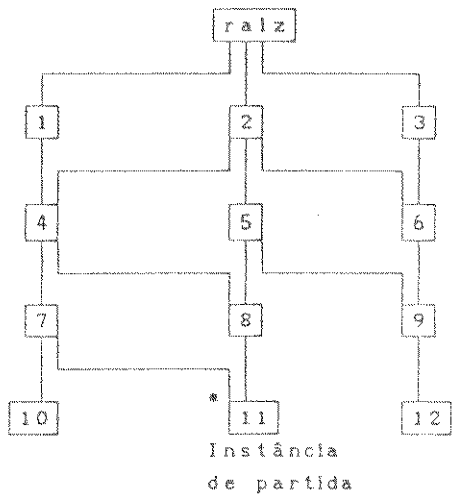
No caso de busca sob múltiplos atributos, o processo se repete em sua totalidade para cada atributo até que algum dado seja encontrado ou a resposta *[não_atrib]* é retornada.

A figura 3.22 ilustra o caminho executado por uma busca genérica que parte de uma instância folha e chega até a raiz.

Na figura 3.22, as LPs só contêm elementos não visitados, ou seja, elementos que não pertençam ao *Path*. Neste caso, este argumento seria dado por:

Path = [11,7,4,1,raiz,2,8,5]

Ainda relativo à fig. 3.22, observa-se que a LP4 foi gerada depois da LP11, porém foi exaurida antes que a segunda instância de LP11 fosse visitada. As últimas LPs são as primeiras a serem exauridas.



Caminho = 11, LP11=[7,8], 7, LP7=[4], 4, LP4=[1,2], 1, LP1=[raiz], raiz, LP4=[2], 2, LP2=[raiz], LP7=[], LP11=[8], 8, LP8=[5], 5, LP5=[], falha.

Figura 3.22 : Busca em Profundidade

Neste tipo de busca, a profundidade é medida pelo número de LPs formadas, ou seja, para o exemplo da fig. 3.22 seria:

LP11 ⟶ *deep 0*(partida)
LP7 ⟶ *deep 1* LP8 ⟶ *deep 1*
LP4 ⟶ *deep 2* LP5 ⟶ *deep 2*
. .
. .
. .

2)fget_level : Comando para busca em amplitude

fget_level(*Frame,Instância,Slotpai,Slot,Atrib,Struct,Deep,*
Atrib_or,Instan_or,Path)

Os argumentos do comando são os mesmos contidos na figura 3.21. Todas as observações do comando anterior são válidas para este comando, exceto as que se referem à profundidade e à maneira pela qual as Listas de Pais são utilizadas.

Neste tipo de busca, as LPs são esgotadas à medida em que vão sendo criadas.

A figura 3.23 mostra o mesmo exemplo da figura 3.22 quando uma busca em amplitude é utilizada.

Neste exemplo o argumento *Path* seria:

Path = [11,7,8,4,5,1,2,raiz].

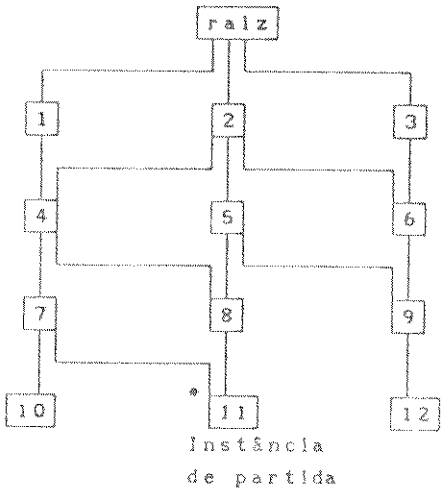
A especificação da profundidade em níveis se dá a cada formação de LP referente a um nível completo. Neste exemplo seria:

LP11 ⟶ *deep 0* (partida)
LP78 ⟶ *deep 1*
LP45 ⟶ *deep 2*
LP12 ⟶ *deep 3*
.
.
.

3.3.2 - Manipulação do Conhecimento Não-Estático

3.3.2.1 - Manipulação de Regras

A manipulação direta com fatos, regras e procedimentos associados aos conjuntos é feita pelo Sistema Especialista. A ativação deste se dá através de um outro comando de gerência específico para este fim, que pode ser utilizado tanto no Programa Objeto como nos procedimentos.



Caminho = 11, LP11=[7,8], 7, LP11=[8], 8, LP11=[], LP7=[4]+LP8=[4,5]
→ LP78=[4,5], 4, LP78=[5], 5, LP78=[], LP4=[1,2]+LP5=[2]
→ LP45=[1,2], 1, LP45=[2], 2, LP45=[],
LP1=[raiz]+LP2[raiz] → LP12=[raiz], raiz, falha.

Figura 3.23 : Busca em Amplitude

O comando de ativação do SE é:

g_chama_SE(*Nome_conjunto,Opção,Goal,Usuário,Interface, Resposta*)

A figura 3.24 mostra os argumentos deste comando.

Quando este comando é executado, o SE define suas janelas de funcionamento, que podem ser usadas ou não, prepara a memória de trabalho para uma nova inferência (ver seção 2.4.2) e começa o processo para tentar provar a meta que lhe foi fornecida.

A meta a ser provada pode ser fornecida ao SE pela parte do sistema que o está acionando ou pelo usuário. Neste caso, o SE estabelece uma interface através de uma janela de comunicação e permite ao usuário fornecer a meta a ser provada.

Quando o usuário não está habilitado a ser questionado e a meta é fornecida pelo módulo que ativa o SE, não existe nenhuma interação visual presente entre o usuário e o sistema. Caso a interface esteja desabilitada, e o usuário possa ser questionado, então a janela de inferência permanece

ativada. A interface fornecida pelo SE é seletiva, podendo ser usada parte a parte segundo as necessidades do módulo usuário.

Nome_conjunto	⟶	Nome do conjunto de regras a ser usado. Este indicador já traz embutida a informação da base a ativar dentro do conjunto
Opção	⟶	Define a maneira de trabalho do SE: incorporação, ajuda, consulta ou terminar (ver seção 2.4.2)
Goal	⟶	Meta a ser provada pelo SE
Usuário	⟶	Especifica se o usuário está habilitado ou não para ser questionado pelo SE
Interface	⟶	Especifica se a interface homemXmáquina do SE está habilitada ou não
Resposta	⟶	Informa se a meta fornecida é verdadeira ou falsa dentro do escopo do conhecimento disponível pelo SE

Figura 3.24 : Argumentos do Comando g_chama_SE

Sempre que o SE é ativado sob uma determinada opção de trabalho, a memória de trabalho é mantida pronta para uma nova inferência caso o conjunto a ser pesquisado seja o mesmo que já esteja presente. Caso não haja um conjunto na memória de trabalho ou o que está presente não é o que deve ser usado, o SE, automaticamente, prepara a memória de trabalho para receber um novo conjunto. Caso seja necessário limpar a memória de trabalho sem que nunhum outro conjunto seja carregado, utiliza-se o comando **g_chama_SE** com o argumento *Opção* tendo o valor *terminar* ou *t*. Os demais argumentos podem assumir qualquer valor.

A resposta fornecida pelo comando de ativação do SE é dada pelo argumento *Resposta*, que assume o valor *true* para a meta provada e *false* para a meta sobre a qual nada foi concluido com o conhecimento disponível pelo sistema.

3.3.2.2 - Manipulação de Procedimentos

Como já foi comentado anteriormente, só o gerenciador pode ativar procedimentos, sejam eles ligados às regras ou aos *frames*. Cada um dos tipos de procedimentos tem um comando de gerência específico para efetuar a chamada e ativação. Os comandos são os seguintes :

1)g_executar_PROC : Comando para ativação de procedimentos ligados aos *frames*.

Este comando serve para qualquer procedimento ligados aos *frames* sob qualquer atributo de especificação : *if_needed, if_write, if_read, if_read, if_added, if_remove.*

g_executar_PROC(*Lista,Process,Resp*)

A figura 3.25 mostra os argumentos utilizados neste comando.

Lista	⟶	Lista de valores para os argumentos estabelecidos para o procedimento
Process	⟶	Procedimento a ser chamado (ver seção 3.2.2.3)
Resp	⟶	Resposta enviada pelo procedimento

Figura 3.25 : Argumentos do Comando g_executar_PROC

O argumento *Lista* deve estar na mesma ordem que a lista de argumentos na estrutura de armazenagem do procedimento no *frame*. Caso haja discrepância entre as listas uma mensagem de erro é enviada ao chamador e a ativação do procedimento é suspensa. Se o procedimento não for achado, uma janela de erro é ativada para avisar que o procedimento pedido não foi encontrado. Quando o procedimento, por alguma razão, tem sua execução bloqueada, o gerenciador suspende a atividade do procedimento e passa ao chamador a informação de falha através do argumento *Resp* para que esse tome as devidas providências para o caso.

O argumento *Process* traz a lista que define o procedimento no *frame*. Este argumento juntamente com o argumento *Lista* permitem ao gerenciador montar a chamada do procedimento. Esta montagem é ilustrada na figura 3.26.

Caso os argumentos já estejam com seus valores definidos na lista de armazenagem ou não existam, o argumento *Lista* é preenchido com o valor *[]* que simboliza uma lista vazia na linguagem Prolog.

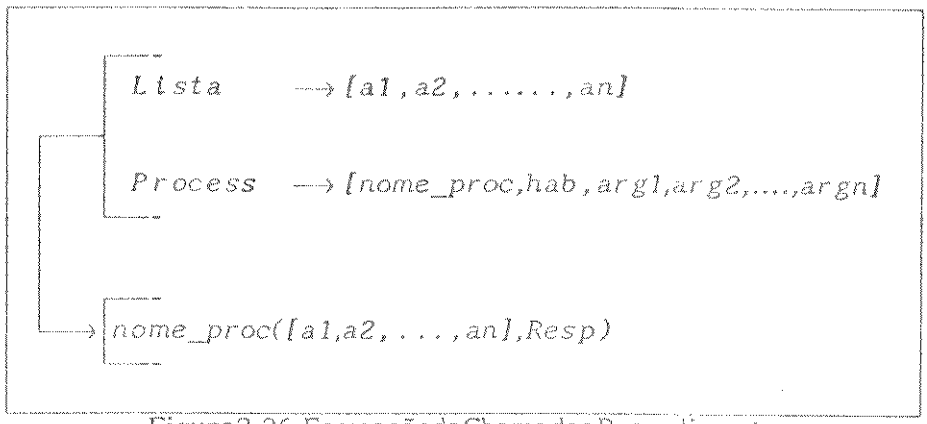


Figura 3.26: Formação da Chamada a Procedimento

Este comando pode definir uma janela para execução do procedimento. Isso é estabelecido pelo segundo elemento da lista contida no argumento *Process*. Caso seu valor seja *habilitado* a janela é definida e aberta, caso contrário fica a cargo do procedimento realizar este tipo de tarefa quando for conveniente.

O argumento *Resp* retorna uma resposta ao procedimento que é repassada ao procedimento chamador para que se estabeleça a comunicação entre o chamador e o procedimento chamado.

2) **g_chama_PROC** : Comando para ativação de procedimentos ligados às regras

Este comando é utilizado pelo SE para ativar os procedimentos que sejam necessários para a conclusão de alguma inferência.

g_chama_PROC(*Process,Meta,Resp*)

A figura 3.27 mostra os argumentos utilizados no comando.

Process	——→	Procedimento a ser chamado (ver seção 3.2.2.4)
Meta	——→	Meta relacionada ao procedimento a ser chamado
Resp	——→	Resposta enviada pelo procedimento

Figura 3.27 : Argumentos do Comando g_chama_PROC

Em caso de falha, o funcionamento deste comando é igual ao anterior. Somente que, neste caso, o chamador é sempre o SE e é ele quem se encarrega de gerenciar estes acontecimentos.

Como no comando anterior, o gerenciador também fornece uma janela padrão para executar o procedimento, a qual pode ser usada ou não. A forma de habilitar ou não esta janela é idêntica ao caso anterior. Também é igual ao comando anterior a formação do nome do procedimento a ser chamado (figura 3.26).

Como o procedimento faz parte de uma inferência, ele pode alterar as conclusões presentes na memória de trabalho. Isto é feito através da resposta enviada ao SE pelo procedimento. Esta resposta pode ser armazenada como um fato temporário (ver seção 3.2.2.4).

Além do que deve ser armazenado na memória de trabalho, a resposta do procedimento pode se utilizar do gerenciador para permitir a visualização de algum dado. Nesta situação o SE ativa o gerenciador através de um comando de visualização para mostrar o dado ao usuário.

A figura 3.28 mostra as estruturas de respostas possíveis e suas consequências.

Ao receber uma resposta que indique que algo deve ser armazenado na memória de trabalho, o SE procura na memória se existe algum outro fato temporário indicado pela mesma meta que chamou o procedimento. Se existir, este fato é apagado e o novo fato é anexado à memória de trabalho. Caso nada seja encontrado, a resposta é anexada como fato temporário normalmente.

A visualização dos dados através do SE é feita utilizando vários comandos do gerenciador que manipulam uma janela específica para esta atividade. Ao fim de uma inferência, a janela é apagada podendo ser usada novamente quando for necessária. Esses comandos são descritos na seção seguinte. Durante a inferência os procedimentos podem fazer uso desta janela de visualização. Porém, o gerenciamento dos dados nela contidos fica a cargo dos procedimentos.

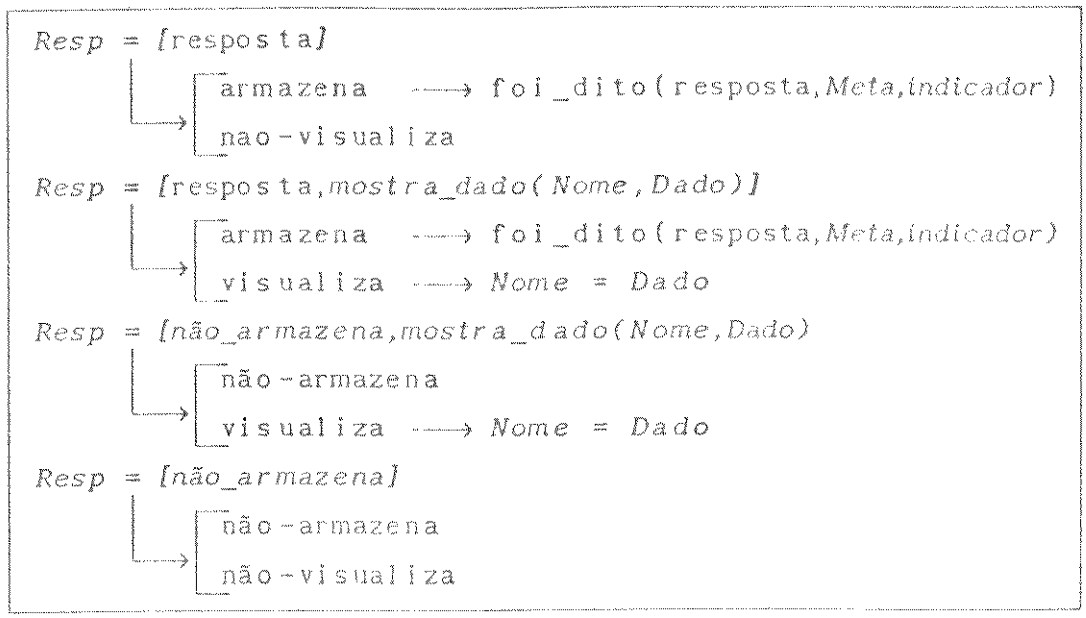


Figura 3.28 : Tipos de Resposta dos Procedimentos Ativados pelo Comando g_chama_PROC

3.3.3 - Comandos de Gerência

As duas seções anteriores também trataram de comandos de gerência, mas se referiam a dois casos particulares cujas características diferem dos comandos descritos a seguir.

Os comandos de gerência fazem uso de todas as qualidades dos comandos do manipulador de *frames* e dados, adicionando a estas a preocupação com a proteção das informações. Além disso, servem de ligação entre os elementos que querem efetuar as manipulações e os comandos realmente as realizam. É através destes comandos que toda troca de mensagens entre o elemento chamador e chamado ocorre.

1)**g_write** : Comando para escrita ou alteração na base de *frames*.

Antes de efetuar qualquer alteração no estado do *slot*, procura pelo procedimento sob o atributo *if_write*. Se este procedimento existe é executado

podendo impedir ou não a operação.

g_write(Frame,Instância,Slotpai,Slot,Atrib,Struct,Resp)

A figura 3.29 mostra os argumentos utilizados pelo comando.

Frame	⟶	Indicador de Classe
Instância	⟶	Indicador de Sub-classe
Slotpai	⟶	Pai do *slot* desejado
Slot	⟶	*Slot* desejado para a operação
Atrib	⟶	Atributo sob o qual deve ocorrer a alteração
Struct	⟶	Dado para operação
Resp	⟶	Resposta do comando enviada ao elemento chamador

Figura 3.29 : Argumentos do Comando **g_write**

As observações seguintes ilustram a comunicação existente entre o elemento chamador, gerenciador e o procedimento chamado.

- Se o procedimento de proteção impede a alteração, a resposta *FAILURE* é enviada ao gerenciador. Este aborta a operação e envia a resposta *Resp = [halt_process]* para o elemento chamador.

- Quando o *slot* procurado não existe, o gerenciador envia a resposta *Resp = [não_especif]*.

- A operação só é permitida sob os atributos *value* e *default*. Caso seja tentado algo diferente, o gerenciador responde com *Resp = [atributo_n_especif]*.

- Quando o atributo pedido não existe o gerenciador responde com *Resp = [não_atrib]*.

- Caso não haja impedimentos e a operação seja realizada com sucesso, o gerenciador responde com *Resp = [no_process]*.

2)**g_read** : Comando para leitura na base de *frames*.

Antes de qualquer leitura verifica a existência de algum procedimento sob o atributo *if_read*. Se este procedimento existe é executado podendo impedir ou não o acesso ao *slot* desejado.

A comunicação estabelecida através deste comando é exatamente a mesma estabelecida pelo comando **g_write**. Exceto pelo fato de que a operação é permitida sob qualquer atributo.

g_read(Frame,Instância,Slotpai,Slot,Atrib,Struct,Resp)

Os argumentos do comando são ilustrados pela figura 3.29, sendo portanto idênticos aos do comando **g_write**.

3)**g_delete_data** : Comando para eliminação de dados.

Verifica inicialmente se o dado existe. Se existe, procura por um procedimento sob o atributo *if_remove_data*. Se este procedimento existe, é executado podendo impedir ou não a eliminação do dado. Se o dado não existe o argumento *Struct* recebe o valor *[[]]*.

g_delete_data(*Frame,Instância,Slotpai,Slot,Atrib,Struct, Resp*)

A comunicação estabelecida através deste comando é exatamente a mesma estabelecida pelo comando **g_write**. Exceto pelo fato de que a operação é permitida sob qualquer atributo.
Os argumentos do comando são idênticos aos mostrados na figura 3.29.

4)**g_delete_atrib** : Comando para eliminação de atributo.

Verifica inicialmente se o atributo existe. Se não existe retorna como resposta *Resp = [não_atrib]*. Se existe, procura por um procedimento sob o atributo *if_remove_atrib*. Se este procedimento existe, é executado e pode impedir ou não a eliminação do atributo. Se não existe procedimento ou este

não bloqueou a retirada do atributo, verifica a existência de dado sob o atributo pedido. Se o dado não existe, o atributo é retirado e a resposta indicando o sucesso da operação é enviada. Se o dado existe, o comando tenta eliminá-lo. Se conseguir, o atributo é eliminado e o dado retirado é enviado no argumento *Struct*. Se o dado for protegido, a resposta de falha é enviada e o dado é fornecido no argumento *Struct*.

O gerenciador se utiliza do comando **fremove** do manipulador para efetivar a eliminação do atributo. Sendo assim, se todos os atributos de um *slot* forem retirados, o *slot* é automaticamente retirado da árvore balanceada.

g_delete_atrib(*Frame,Instância,Slotpai,Slot,Atrib,Struct,Resp*)

A comunicação estabelecida através deste comando é exatamente a mesma daquela estabelecida pelo comando **g_write**. Exceto pelo fato de que a operação é permitida para qualquer atributo.

Os argumentos do comando são os mesmos apresentados pela figura 3.29.

5)g_delete_slot : Comando para eliminação de *slot*.

A eliminação do *slot* é feita usando o fato que o comando **fremove** retira, automaticamente, o *slot* e toda a ramificação dependente dele da árvore balanceada.

Verifica inicialmente se o *slot* existe. Se existe procura um procedimento sob o atributo *if_remove_slot*. Caso este procedimento exista, é executado podendo impedir ou não a eliminação do *slot*. Se o procedimento não existe ou não impede a operação, o comando verifica a existência de dados sob os atributos *value* e *default*. Se existem, tenta elimina-los. Se os dados são protegidos a operação é abortada e a resposta de falha enviada. Se isso não ocorrer, os dados dos atributos de procedimentos, se existirem, são eliminados como todos os demais atributos e por fim o própio *slot*. Em qualquer um dos casos em que dados existam, estes são fornecidos sob forma de lista no argumento *Struct*.

Vale observar que os procedimentos não são na verdade eliminados do sistema, mas apenas os seus indicadores são retirados da base de *frames*.

g_delete_slot(*Frame,Instância,Slotpai,Slot,Struct,Resp*)

A comunicação estabelecida através deste comando é exatamente a mesma estabelecida pelo comando **g_write**. Exceto pelo fato de não haver

comunicação envolvendo atributo.

Os argumentos do comando são idênticos aqueles da figura 3.29. Exceto no que se refere à definição do argumento *Atrib*, que neste comando não é definido.

6)**g_input** : Comando para inserção de novos atributos em *slots* já existentes, criação novos *slots* e criação de novas ramificações.

i) *Slots* convencionais : Se o *slot* não existe é diretamente inserido. Se já existe, só é permitida a inserção de algum atributo se este também não existir no *slot* em questão. No caso de atributo, um procedimento sob o atributo *if_added* é procurado. Se existir, é executado podendo impedir ou não a operação.

ii) *Slots* encadeados : Verifica, no *slot* indicado pelo argumento *Slotpai* do comando, a existência de um procedimento sob o atributo *id_added*. Se existir, é executado podendo impedir a operação ou não. Se for possível, a inserção é feita via comando fput. Se o *slot* indicado pelo argumento *Slotpai* não existir uma nova ramificação é criada.

g_input(*Frame,Instância,Slotpai,Slot,Atrib,Struct,Resp*)

Os argumentos do comando são os mesmos da figura 3.29.

A comunicação utilizando o argumento *Resp* se resume apenas às resposta relativa a falha ou sucesso da operação que são descritas na comunicação estabelecida pelo comando g_write.

7)**g_show_data_set** : Comando para definição de janela para edição de dados genéricos.

8)**g_show_data_reset** : Comando para destruição da janela de edição de dados genéricos.

9)g_show_data : Comando para visualização de dados.

Serve para escrever dados na janela de edição juntamente com a sua especificação.

g_show_data(*Nome,Data*)

Os argumentos do comando são mostrados na figura 3.30.

Nome	⟶	Especificação do dado a ser escrito na janela. É uma cadeia de caracteres
Data	⟶	Dado a ser escrito na tela. O dado deve ser convertido em cadeia de caracteres

Figura 3.30 : Argumentos do comando g_show_data

10)g_show_data_clear : Comando para limpar a janela de edição de dados genéricos.

11)g_read_deep : Comando para leitura em herança usando busca em profundidade.

Este comando tem as mesmas caracterísitcas do comando g_read, só que utiliza o comando de leitura em herança do manipulador que efetua busca em profundidade.

Este comando usa o comando do manipulador para efetivar a busca e, logo em seguida executa, um comando g_read utilizando as informações obtidas pelo comando do manipulador (*Atrib_or* e *Instan_or*) para testar a proteção e efetuar a comunicação.

g_read_deep(*Frame,Instância,Slot_pai,Slot,Atrib,Struct,Deep,Atrib_or, Inst_or ,Path,Resp*).

Os argumentos são mesmos do comando **fget_deep** ilustrados na figura 3.21. O argumento *Resp* é utilizado para comunicação ativador-ativado da mesma forma que no comando **g_read.**

12)**g_read_level** : Comando para leitura em herança usando busca em amplitude.

Este comando funciona de forma idêntica ao comando anterior, exceto pelo fato de utilizar o comando de leitura em herança do manipulador que efetua a busca em amplitude.

3.4 - Resumo

Neste capítulo, o enfoque dado ao sistema SMAC foi o da implementação computacional. Foram discutidos os comandos dos vários níveis de abstração existentes, a troca de mensagens entre os procedimentos e os elementos do sistema resposáveis pela ativação, a configuração compilada e interpretada do sistema, e as características de cada modo de funcionamento.

No próximo capítulo serão discutidos exemplos em que o sistema SMAC foi aplicado com sucesso. São discutidas também, algumas novas características a serem implantadas no sistema e quais as vantagens destas extensões.

Capítulo 4 : Aplicações e Extensões

4.1 - Introdução

Este capítulo mostra a aplicabilidade prática do sistema SMAC. São discutidos dois exemplos de razoável complexidade, onde a utilização da versatilidade e modularidade da representação e manipulação do conhecimento adotadas no SMAC, permitem a implementação de sistemas poderosos aplicados a áreas completamente diversas. Além das aplicações, são discutidas algumas extensões a serem desenvolvidas com o objetivo de eliminar algumas limitações e de acrescentar novas características ao sistema SMAC.

4.2 - Simulação em Sistemas Flexíveis de Manufatura

4.2.1 - Introdução aos SFM e à Simulação

Embora não exista ainda uma definição universal para um Sistema Flexível de Manufatura (SFM), é coerente a definição dada por Groover [22] : " um sistema de manufatura composto por um certo número de máquinas controladas numéricamente, conectadas por um sistema automático de transporte de materiais, tudo sob controle computadorizado, estabelecido para processar uma grande variedade de diferentes partes, variando seu volume de produção de baixo para médio".

Na instalação de uma fábrica pode-se ter ilhas de produção, todas com as características acima descritas, ilhas estas chamadas de Células Flexíveis de Manufatura, que, quando interligadas por computador, formam o Sistema Flexível de Manufatura.

A indicada flexibilidade corresponde à capacidade do sistema de, rápida e eficientemente, alterar seu processo de produção, incluindo uma reprogramação de alguns ou todos os elementos da célula.

Flexibilidade na manufatura é um conceito que tem sido definido de diferentes formas, dependendo do sistema ou problema em análise. No entanto, embora cada instalação ou processo corresponda a uma entidade distinta, as metas para alcançar uma produção ótima, com alta qualidade e mínimo custo são similares.

Alptekin [3] cita flexibilidade como a habilidade de um sistema ou processo decisório adaptar-se a quaisquer circunstâncias de mudanças.

O surgimento dos SFM representou um considerável passo na resposta aos requisitos atuais de competitividade e qualidade de produção de manufaturados.

A implementação de SFMs tem apresentado problemas razoavelmente complexos no que diz respeito à especificação técnica, projeto e operação de tais sistemas. É preciso conciliar requisitos técnico-operacionais, financeiros e institucionais através de uma técnica de escalonamento de tarefas que possa fornecer o melhor rendimento possível [39].

Devido ao alto grau de complexidade e à presença de muitos fatores heurísticos, a modelagem tradicional destes sistemas através de técnicas numéricas é uma tarefa muito difícil. Diante desse quadro, a principal ferramenta utilizada na análise destes sistemas tem sido a simulação. Principalmente no estudo do desempenho global de sistemas com implantação proposta, como auxiliar no processo decisório em sistemas já existentes, na análise do desempenho do sistema frente a diversos planejamentos de produção, na análise do comportamento do sistema em relação às alterações possíveis de serem implementadas, etc.

Simulação em eventos discretos é vista como uma contínua geração de mudanças nos estados do sistema, de uma maneira previamente especificada. As entidades (objetos), descritos por seus atributos, são alteradas de acordo com a lógica estabelecida no modelo. Tal lógica determina a mudanças nos estados do sistema e nos estados das entidades. A lógica se constitui nas regras que governam o comportamento do modelo e descrevem a dinâmica do sistema a ser modelado.

Generalizando, um modelo para simulação é um modelo descritivo de um sistema real ou previsto, usualmente utilizado para a previsão do desempenho do sistema e avaliação das principais estratégias, algoritmos e regras impostas ao sistema.

Através do uso da simulação pode-se captar os detalhes necessários de um sistema dinâmico e complexo. Esta característica é particularmente importante quando se trata de um SFM, devido à grande dificuldade de obter uma descrição satisfatória através de um modelo analítico, que expresse toda a complexidade e detalhamento da sua estrutura.

Conceitualmente um modelo analítico apresenta significativas diferenças em relação à simulação. Os modelos analíticos são apropriados para decisões preliminares, enquanto modelos de simulação são usados para uma decisão operacional detalhada.

Conforme descrito nos ítens acima, muitas das fases de projeto, implementação e operação de SFMs podem ser avaliados previamente através da simulação, sendo que, em termos genéricos, qualquer atividade relacionada com a manufatura poderia ser simulada antes de qualquer posicionamento sobre qualquer tema, tanto da área técnica como da administrativa.

Tendo que servir a uma vasta gama de aplicações, tal sistema simulador deve ser dotado de grande flaxibilidade, caracterizada na possibilidade de modelagem de várias configurações de sistemas, de entidades de trabalho que façam parte do sistema e da modelagem, também, das funções de controle, regras de decisão, sequenciamento de partes, etc.

A evolução e a afirmação da simulação como ferramenta eficiente tem moldado alguns requisitos importantes para ambientes desta natureza, tais como:

- Interface amigável que permita o uso do sistema também por usuários com pouca experiência,

- Modularidade: permitir manutenção e aperfeiçoamento sem uma necessidade de uma re-estruturação completa do sistema,

- Separação entre a modelagem e a execução da simulação e a apresentação dos resultados,

- Permita ser integrado como ferramenta e propicie uma maior comunicação entre os membros do projeto (analista, especialista, engenharia, etc),

- Flexibilidade no sentido de atender a uma quantidade variável e grande de aplicações da simulação em manufatura.

Tendo em vista as características da manufatura totalmente integrada por computador, é interessante que o sistema simulador possa atender níveis hierárquicos mais elevados da organização fabril. Dessa forma, a simulação atenderia desde o controle de máquinas até a análise tecno-econômica de projetos.

Evitando a multiplicidade de recursos, é desejável que o mesmo sistema de simulação possa atender a todas as necessidades sendo apenas necessária uma reconfiguração na base de informações.

O sistema SMAC fornece todas as características necessárias para a implementação de um sistema de simulação como o descrito nessa seção.

4.2.2 - Descrição da Célula Exemplo

A célula a ser modelada corresponde a uma unidade de montagem eletrônica onde componentes são colocados em uma placa de circuito impresso, inicialmente vazia [37].

O conjunto é composto por dois robôs que apresentam uma concorrência de operações, já que seus espaços de trabalho, em alguns momentos, são coincidentes. Um dos robôs é usado para colocar os *chips* na placa virgem e o outro usado para a manipulação da placa e sua soldagem, sendo que a alimentação com *chips* é relizada através de um alimentador gravitacional contendo grande quantidade dos mesmos.

Existem, também, dois *buffers*, sendo que um contém placas virgens e o outro contém as placas já montadas.

Durante a montagem, as placas são mantidas em posição através de um fixador específico, sendo que, após a montagem dos componentes na placa, esta passa por um tanque de soldagem por onda.

A disposição do conjunto de equipamentos que constituem a célula pode ser vista na figura 4.1.

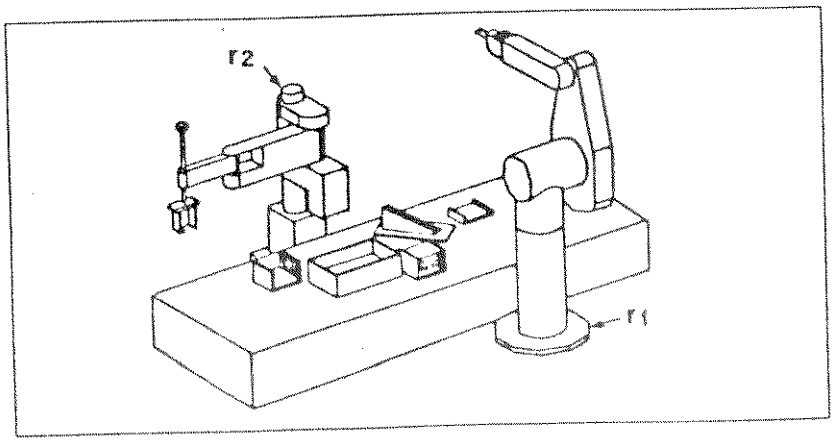


Figura 4.1 : Célula Exemplo

Como elementos a serem modelados foram escolhidos os robôs e os *buffers*, e os demais elementos foram considerados passivos para este exemplo.

A sequência de operações das entidades constituintes da célula podem ser descritas pelos passos abaixo :

a) ROBÔ MONTADOR (ROBÔ 1)

1- espera sinal do robô 2,
2- uma vez recebido o sinal inicia a colocação dos *chips* na placa,
3- terminada a colocação dos componentes move-se para lugar seguro,
4- sinaliza para o robô 2,
5- vai para passo 1.

b) ROBÔ MANIPULADOR (ROBÔ 2)

1- robô apanha a placa vazia no *buffer* de placas vazias (1),
2- coloca a placa no fixador,
3- move-se para lugar seguro,
4- sinaliza para robô montador (ROBÔ 1) iniciar seus movimentos,
5- espera sinal de área livre, emitido pelo robô montador,
6- uma vez recebido o sinal, leva, temporariamente, a placa montada, do fixador para o *buffer* 2,
7- pega placa virgem no *buffer* 1,
8- coloca a placa no fixador,
9- move-se para lugar seguro,
10- sinaliza para robô montador iniciar seus movimentos,
11- pega placa montada no *buffer* 2,
12- executa soldagem da placa,
13- coloca placa soldada no *buffer* 2,
14- vai para passo 5.

c) BUFFER DE PLACAS VIRGENS (BUFFER 1)

Sem atividades detalhadas, apresentando-se apenas como uma forma de manter um controle do número de placas virgens usadas.

d) BUFFER DE PLACAS MONTADAS (BUFFER 2)

Sem atividades detalhadas, apresentando-se apenas como uma forma de manter-se um controle do número de placas montadas.

4.2.3 - Modelagem da Célula Exemplo

A modelagem das entidades segue a estrutura estabelecida no Sistema Servidor de Simulação [39,40]. Neste, as entidades são descritas em níveis

hierárquicos, sendo que cada entidade, em cada nível, é caracterizada por um conjunto de entradas e saidas como ilustrado na figura 4.2.

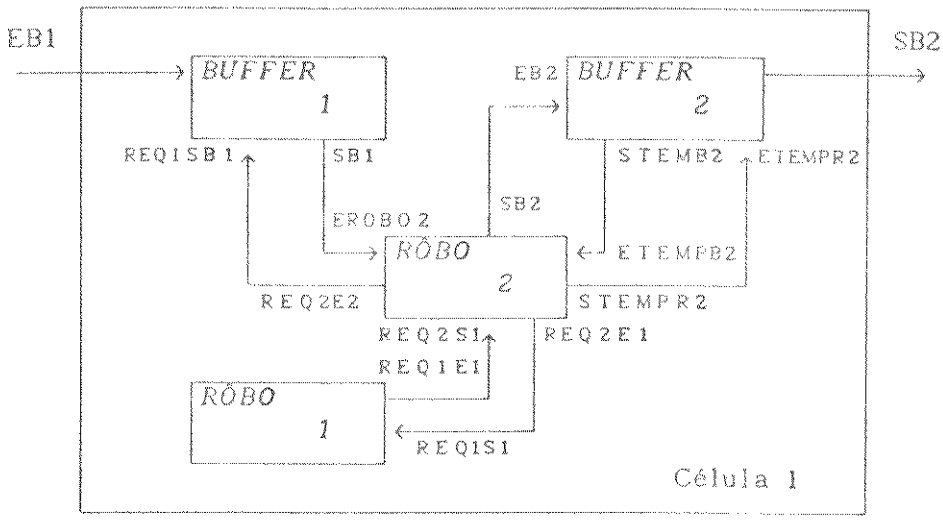


Figura 4.2 : Modelo em Níveis Hierárquicos

As informações referentes à modelagem de cada entidade componente da célula, são apresentadas em *frames*, como os mostrados nas figuras 4.3 e 4.4, onde são reproduzidos, como exemplo, o *frame* de modelagem do *buffer* 2 e do robô 1. Os *frames* representativos dos outros elementos que compõem a célula exemplo seguem exatamente a mesma estruturação.

Cada entidade é representada através de uma classe do sistema SMAC. Esta entidade é fechada e independente podendo ser alterada individualmente ou facilmente substituida por outra. Utilizando esta característica pode-se criar uma biblioteca de entidades com capacidade de compor as mais diversas células flexíveis de manufatura a serem simuladas.

Todos os outros *frames* pertencentes ao sistema simulador seguem o mesmo critério usado para as entidades. Somente os *frames status* e *trace*, discutidos adiante, são montados a cada ciclo de simulação, pois são particulares a cada simulação que vai ser feita.

Os *slots name*, *version* e *date_time* determinam o cabeçalho de especificação de uma determinada entidade, sendo que a especificação <*value*> indica que o *slot* está preenchido com um valor final que univocamente o caracteriza.

Os *slots members* e *a_kind_of* determinam a posição em que cada entidade se encontra dentro da árvore hierárquica que representa o sistema em estudo, sendo que o *slot members* indica qual(is) a(s) entidade(s) filha(s) e o *slot a_kind_of* indica qual a entidade pai da mesma.

Dada a estrutura do Sistema de Manipulação de Conhecimento adotada, dentro de cada entidade o vinculo relacional estabelecido entre os *slots* é de

pai → *filho*, não ocorrendo a definição estrutural em sentido oposto, o que caracterizaria, em muitos casos, uma redundância de informação.

```
        Name : <value> → buffer2
        version    : <value> → 1
        date_time  : <value> → 01-12-89
        members    : <value> → [ ]
        a_kind_of  : <value> → celula1
    Entradas  :ebuffer2 <value> → [1,1,0,_],
               etempb2 <value> → [1,1,0,_].
    Saidas    :sbuffer1 <value> → [1,10000,0,_],
               stempb2 <value> → [1,1,0,_].
    Requisições_de_entrada :
    Requisições_de_saida   :reqsb2 → <if_needed> → reqsb2
    Informações_Gerais :
        - Tempo_médio_entre_quebras : <value> → [ ]
        - Tempo_médio_de_manutenção : <value> → [ ]
    Eventos_Possíveis : <value> → coloca.
    Tabela    :Rot1: <value> → [coloca > [1]].
    Rotina : Rot1: <value> → [1>/.].
    Decisão: Rot1: <rule> → Base_Dec_Rot1
                   <value> → Próximo_evento.
    Eventos: coloca → <if_needed> → [coloca,_,_,0,_,0,T].
    Execução_evento : <value> → (Próximo_evento)
    Regras : <rule> → Base_Buffer2.
```

Figura 4.3 : *Frame* Representando a Modelagem do *Buffer 2*

Devido a isto, há necessidade que, em um mesmo *frame*, os nomes dos possíveis *slots* "filhos" sejam todos distintos entre si, evitando-se ambiguidade de *slots* filhos em relação ao pai ou *slot* raiz.

Todas as entradas e saídas do modelo da entidade, conforme mostrado na figura 4.2, são relacionadas nos *slots* entradas, saidas, requisições_entrada e requisições_saida. Note-se que, como as requisições representam canais de comunicação, estas são representadas pela ocorrência de um evento.

```
      Name : <value> --> Robo1
      version    : <value> --> 1
      date_time : <value> --> 01-12-89
      members    : <value> --> []
      a_kind_of : <value> --> celula1
Entradas :
Saidas   :
Requisições_de_entrada : Reqle1 <if_needed> --> reqle1
Requisições_de_saida   : Reqls1 <if_needed> --> reqls1
Informações_Gerais :
      - Tempo_médio_entre_quebras : <value> --> []
      - Tempo_médio_de_manutenção : <value> --> []
Eventos_Possíveis : <value> --> [inic_parada, fim_parada,
                                 inic_mov1, fim_mov1,
                                 inic_processa1, fim_processa1,
                                 inic_mov2, fim_processa2, reqle
Tabela :Rot1: <value> --> [inic_parada > [1], fim_parada > [2]
                          inic_mov1 > [3], fim_mov1 > [4],
                          inic_processa1 > [5],
                          fim_processa1 > [6], inic_mov2 > [7]
                          fim_processa2 > [8], reqle1 > [9]].
Rotina : Rot1: <value> --> [1>2>3>4>5>6>7>8>9>/.]
Decisão: Rot1: <rule> --> Base_Dec_Rot1
                <value> --> (Próximo_evento).
Eventos: inic_parada --> <if_needed> --> [inic_parada,..._,0,T]
         fim_parada --> <if_needed> --> [fim_parada,..._,0,T],
         inic_mov1 --> <if_needed> -->
                                [inic_mov1,des,descanso,feeder,..
         ...
         reqle1 --> <if_needed> --> [reqle1,des,...,T].
Execução_evento : <value> --> (Próximo_evento)
Regras          : <rule> --> Base_Robo1
```

Figura 4.4 : *Frame* Representando a Modelagem do Róbo 1

As estruturas de entradas e saídas são constituídas por listas cujos elementos são fixos. A estrutura de uma entrada ou saída é da forma que segue:

[Entrada/Saída] ⟶ [número de lotes, tamanho do lote, Quantidade atual, Quant. para requisição]

As requisições não têm argumentos pois são vias específicas de comunicação que, quando ativadas, têm um significado próprio definido na arquitetura da simulação em andamento, a partir dos procedimentos estabelecidos para cada uma. As requisições de saída são recebidas pela entidade como pedido para fornecimento de algum serviço ou produto. As requisições de entrada são enviadas pela entidade como pedido para fornecimento de produto e/ou serviço por parte da outra entidade.

Essas requisições servem para emular o sistema do tipo Kambam na filosofia *just_in_time*. Tais requisições simulam a emissão das fichas utilizadas no Kambam.

Na especificação das requisições de entrada e saída figura o argumento <*if_needed*>, o qual especifica um procedimento associado que realizará alguma ação pertinente à simulação do modelo em estudo.

No *slot* informações_gerais devem estar contidos os dados a respeito de tempos de quebra e de manutenção, bem como outros que se façam necessários, como por exemplo dados financeiros a respeito da entidade.

O *slot* eventos_possiveis recebe o nome de todos os eventos passíveis de ocorrer na entidade, notando-se o fato de que cada atividade é decomposta nos eventos "inicio" e "fim" de atividade.

As distintas ocorrências de cada evento são descritas no *slot* tabela, onde, a cada evento, corresponde um ou mais números que indicam as posições de suas ocorrências em uma dada rotina.

No *slot* rotina são sequenciados os eventos a partir dos números que os identificam em cada rotina. A ocorrência da "/" indica a situação em que uma decisão deve ser tomada para reordenar o fluxo de eventos. Por exemplo, quando múltiplas opções de caminhos de fluxo estão disponíveis.

Quando da ocorrência da "/" o sistema consulta a base de conhecimento de decisão, que se encontra sediada no *slot* decisão, para efetuar o redirecionamento do fluxo para um novo trecho de sequenciamento de eventos. No caso da "/" ser o primeiro elemento especificado em uma rotina, a base de dados deverá conter uma regra chamada decisão_inicial, que será a responsável pelo estabelecimento de qual evento deve iniciar a rotina. Esta decisão será tomada em função das atuais condições do sistema, uma vez que ainda nenhum evento foi realizado nesta entidade.

Cada entidade pode ter quantas rotinas forem necessárias para expressar cada tipo de sequenciamento que a entidade é capaz de realizar. Cada uma destas rotinas é monitorada por uma base de regras própria e independente. A indicação de qual rotina será usada na simulação é dada junto com a indicação de quais entidades estarão presentes no *slot* entidades_sistema no

frame trace, explicado mais adiante. Uma mudança de comportamento pode ser conseguida com uma simples alteração da rotina a ser usada em uma das entidades.

Para descrever as rotinas de uma forma suficientemente genérica, foi criado um pequeno alfabeto:

- cada número é relacionado a um evento e pode ser repetido dentro da rotina.

- cada sequência linear de eventos (sequência sem ramificação) pode ser terminada por:

  . - final de rotina.
  , - indica o final de um bloco linear de eventos. O sistema procura, em outro bloco, a repetição do último evento realizado no bloco que foi finalizado, e passa a executar esse novo bloco linear.
  /, - indicador de bloco finalizado por decisão. A decisão. A decisão de qual bloco seguir é dada pela base de regras da rotina.
  /. - indicador de final de bloco com decisão. A rotina pode ser continuada ou não dependendo da base de regras.

- (>) representa o indicador de sequência.
- (/) representa o indicador de processo decisório

Os fluxogramas que representam as rotinas podem ser escritos utilizando os seguintes símbolos:

- [N] representa evento incondicional.

- [/N] representa um evento condicionado. Quando da eleição do evento a ser executado pela entidade, a base de regras da entidade decide se este evento é passível de execução ou não.

- [/] representa um processo decisório dentro da rotina para encontrar qual será o próximo evento.

- [FIM] representa o final da rotina.

As figuras 4.5 e 4.6 ilustram dois sequenciamentos de eventos genéricos e as respectivas representações no *slot* rotina.

Além da base de regras especificada em cada rotina do *slot* decisão pelo atributo <*rule*>, este *slot* contém um atributo <*value*> onde é armazenada a indicação do próximo evento a ser executado pela entidade, escolhido pelo processo de tomada de decisão.

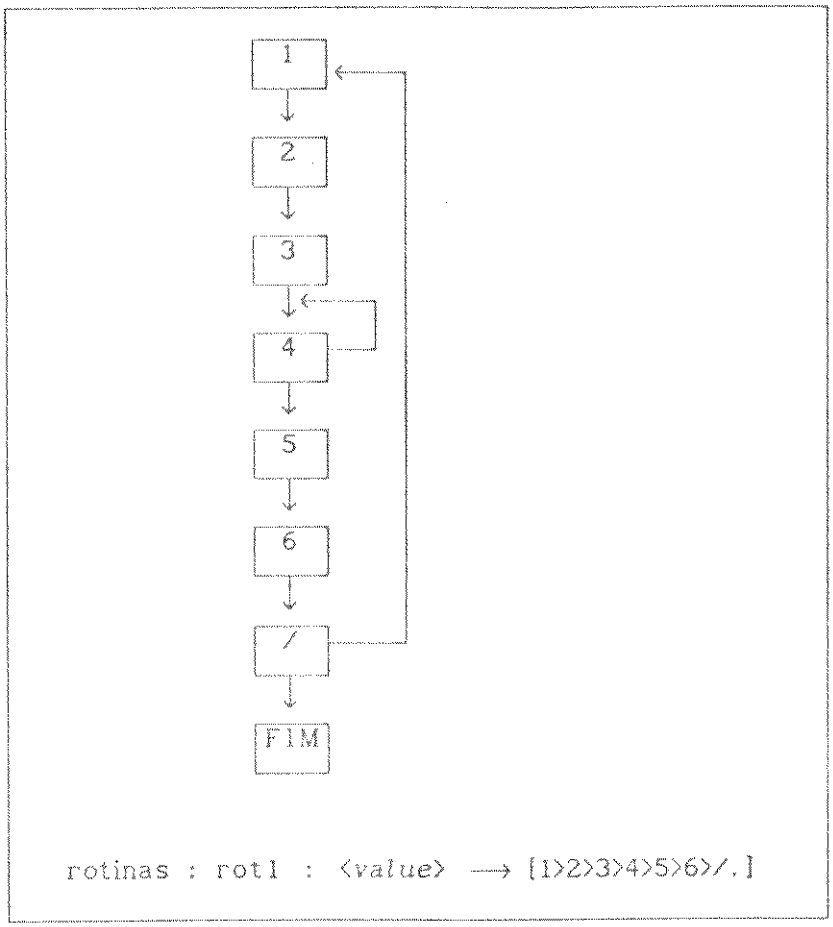


Figura 4.5 : Exemplo de Sequenciamento de Eventos

No *slot* eventos são descritos, com detalhes, todos os eventos que podem ocorrer na entidade modelada. Cada evento é descrito por uma lista que contém 9 argumentos, quais sejam, o nome do evento, dois argumentos genéricos que em caso de movimentos representam, respectivamente, o início e fim do movimento,um argumento indicando tempo médio para ocorrência do próximo evento, o tipo de variação estatística desse tempo, seu desvio médio e por fim o instante de ocorrência deste evento.

Cada evento representa um procedimento a ser executado quando da escolha do evento. A sua representação dentro das entidades segue o padrão dos procedimentos do SMAC. O evento tem a estrutura que segue:

evento : <*value*> ⟶ [nome,hab_janela,listargs]

listargs ⟶ [hab_interferência,de_onde,para_onde,
tem_med_prox_evento,desvio_tipo,
desvio_médio,instante_execução]

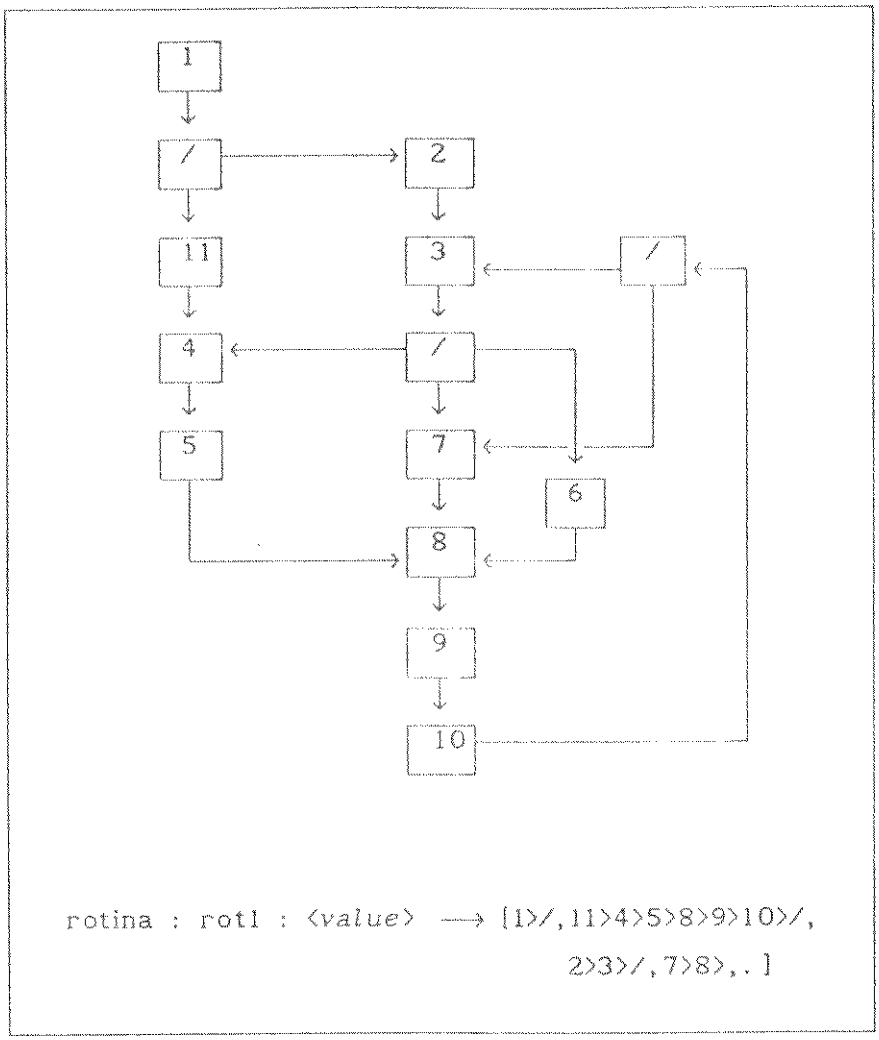


Figura 4.6 : Exemplo de Sequenciamento de Eventos

O argumento hab_inferência quando assume o valor *habilitado* indica que, antes de colocar o evento na fila para execução, a base de regras da entidade deve ser consultada. Quando assume o valor *desabilitado* classifica o evento como incondicional.

No exemplo particular desta célula de montagem, os eventos que, pelas característica de movimentação que possuem, devem ativar as bases de regras das respectivas entidades são:

- fim_parada no rôbo 1
- fim_parada no rôbo 2
- retira no *buffer* 1
- coloca no *buffer* 2

O procedimento de enfileiramento, discutido adiante, ao determinar qual o evento que a entidade deve executar, acrescenta três argumentos à descrição do mesmo. Estes argumentos vão servir como identificadores para o

evento entre os eventos de outras entidades, e para manipulações futuras por outras partes do simulador. Após a manipulação do enfileiramento o evento se apresenta como a seguir:

evento : <*value*> ⟶ [nome,hab_janela,listargs, (posição_na_rotina), (entidade),(rotina)]

Eventos que por ventura não tenham necessidade de especificações de direcionamento, como é o caso dos eventos inic_parada, fim_parada e reqlel do robo 1, o segundo e terceiro argumentos da lista de argumentos permanecem não indexados, podendo ser usados a qualquer momento que seja necessário.

O caso de utilizar o valor zero para o tempo de ocorrência do próximo evento, faz parte da lógica de ressincronização de eventos que por algum motivo tenham ficado esperando o cumprimento de alguma restrição para então serem escalonados. Ou, simplesmente indica a possibilidade de ocorrência imediata do próximo evento escalonado.

O processo de atualização do instante de execução de cada evento ocorre da seguinte forma :

- O instante de execução do evento atual é calculado como sendo o valor do instante de execução do evento anterior acrescido do valor de tempo médio para o próximo evento, também apresentado pelo evento anterior,

- Após este cálculo o sistema verifica se este valor é menor que o valor do *clock* atual. Se isto ocorrer, devido a algum atraso da liberação do evento por falta de condições de execução, seu instante de execução é corrigido para o valor do *clock* atual e então apresentado para o monitor. Caso o valor calculado seja maior que o instante atual de *clock*, o evento é apresentado para o monitor como se encontra.

O processo de enfileiramento decide, dentro de cada entidade, qual o próximo evento a ser executado. Este evento é armazenado no *slot* execução_evento. As regras que serão usadas por este procedimento para a escolha do evento estão armazenadas no *slot* regras. Cada entidade tem um conjunto de regras cujo escopo se limita a ela mesma, não levando em consideração o inter-relacionamento entre entidades.

Ainda referente ao modelagem das entidades a serem simuladas, descreve-se, em um *frame* especifico chamado de Matriz de Ligação, o inter-relacionamento entre as diversas portas de entrada e saida de cada entidade com seus pares, bem como a identificação das requisições de entrada e saida de cada entidade com seus pares.

Através deste artifício tem-se a possibilidade de descrever a modelagem de cada entidade fechada em si mesma, ou seja, modularizada, de forma que eventuais trocas de relacionamentos de entrada-saida entre entidades não requeira um nova modelagem mas apenas uma reorganização da relação entre portas de entrada-saida descrita na matriz de ligação.

Um exemplo do *frame* matriz de ligação, para o exemplo citado é mostrado na figura 4.7.

```
Name : <value> --> Matriz de Ligação
version    : <value> --> 1
date-time  : <value> --> 01-12-89
members    : <value> --> [ ]
a_kind_of  : <value> --> celula1
matriz_de_ligação : <value> --> [req1e1  =  req2s1,
                                 req2e1  =  req1s1,
                                 req2e2  =  req1sb1,
                                 req2e3  =  req1sb2,
                                 SB1     =  EROBO2,
                                 SROBO2  =  EB2,
                                 STEMPB2 =  ETEMPR2,
                                 STEMPR2 =  ETEMPB2.]
flags      : req2s1 <value> --> reset,
             req1sb2 <value> --> reset,
             req1sb1 <value> --> reset,
             req1s1 <value> --> reset.
```

Figura 4.7 : *Frame* Matriz de Ligação

Neste *frame*, o cabeçalho se apresenta como os cabeçalhos de entidades, uma vez que seu formato é padrão, não importando a função do *frame*.

O *slot* matriz_de_ligação apresenta o relacionamento entre cada porta de entrada em uma entidade com a porta de saída de outra, apresentando, também, as relações entre as requisições de entrada em uma entidade com as de saída em outra.

O estado atual de cada requisição é apresentado no *slot flag*, sendo que, a cada modificação deste estado, este valor é atualizado, bem como o valor do estado da *flag* interligada. Ou seja, se ocorre alteração no estado de uma *flag*, o estado da *flag* com a qual a primeira está ligada via *slot* matriz_de_ligação, também é alterado.

Para o desenvolvimento e apresentação da simulação, ainda há a necessidade de outros *frames* que, junto com os já descritos, formam o conjunto de dados do Sistema de Simulação. Tais *frames* são: *status*, condições de terminação, *trace* e monitor de filas.

O *frame* de *status* descreve o estado em que se encontra o sistema, indicando o último evento executado de cada entidade, o valor do *clock* atual

de simulação e uma indicação de todas as entidades e respectivas rotinas, presentes na atual ciclo de simulação, conforme exemplificado na figura 4.8.

As condições em que se deve terminar o ciclo de simulação são estabelecidas através do *frame* de Condições de Terminação. Este, além do cabeçalho padrão já descrito, indica apenas o nome da base de regras onde se encontram as condições para término da simulação.

Este *frame* se apresenta conforme exemplificado na figura 4.9.

O *frame* de *trace* vai sendo preenchido a cada execução de eventos, sendo o responsável pela coleta de dados da história da simulação.

```
Name : <value> --> status
version    : <value> --> 1
date-time  : <value> --> 01-12-89
members    : <value> --> [ ]
a_kind_of  : <value> --> celula1
entidades_sistema : <value> --> [robo1, rot1,
                                 robo2, rot1,
                                 buffer1, rot1,
                                 buffer2, rot1]
entidades : robo1 --> <value> --> [fim_parada]
            robo2 --> <value> --> [fim_mov3]
            buffer1 --> <value> --> [retira]
            buffer2 --> <value> --> [coloca].
clock     : --> <value> --> tempo.
```

Figura 4.8 : *Frame Status*

```
Name : <value> --> Condições de Terminação
version    : <value> --> 1
date-time  : <value> --> 01-12-89
members    : <value> --> [ ]
a_kind_of  : <value> --> celula1
condições_de_terminação : <rule> --> base_para_terminação.
```

Figura 4.9 : *Frame* Condições de Terminação

É com os dados contidos no *frame trace* que os módulos pós-processadores obterão informações para todas as análises requeridas da simulação, inclusive uma eventual animação.

Este *frame* assume uma forma como a exemplificada na figura 4.10.

```
Name : <value> --> trace
version    : <value> --> 1
date-time  : <value> --> 01-12-89
members    : <value> --> [ ]
a_kind_of  : <value> --> celula1
entidades  : entidade1 --> <value> --> [evento_1, evento_2,..,
                                        evento_n].
             entidade2 --> <value> --> [evento_1, evento_2,..,
                                        evento_n].
             ....
             entidaden --> <value> --> [evento_1, evento_2,..,
                                        evento_n].
```

Figura 4.10 : *Frame Trace*

Dentro do *frame trace* os eventos são armazenados numa estrutura mais simplificada, pois muitos dos argumentos que definem e caracterizam o evento passam a ser desnecessários. A forma de armazenagem é descrita a seguir:

entidaden : <value> --> [[nome,instante_execução,posição_rotina],
[nome,...],[nome,...],...].

Os *frames trace* e *status* são montados pelo módulo *start*, ao se iniciar cada ciclo de simulação, já que as entidades disponíveis a cada simulação são variáveis.

O *frame* monitor é fixo dentro do sistema e contém os procedimentos para realizar o monitor e o enfileiramento lógico em cada entidade.

A figura 4.11 ilustra o *frame* monitor montado no sistema

4.2.4 - SMAC - Simulador

4.2.4.1 - Lógica de Simulação

A simulação desenvolve-se segundo um ciclo lógico decomposto nos módulos lógicos representados na figura 4.12.

```
Name : <value> ⟶ monitor
version     : <value> ⟶ 1
date-time   : <value> ⟶ 01-12-89
members     : <value> ⟶ [ ]
a_kind_of   : <value> ⟶ celula1
monitor     : <if_needed> ⟶ [monitor]
enfileiramento : <if_needed> ⟶ [enfileiramento]
condições_de_simulação : <rule> ⟶ base_cond._simulação
evento_a_executar : <value> ⟶ (evento p/ execução)
```

Figura 4.11 : *Frame* Monitor

O início da execução da simulação desenvolve-se no bloco de *start*. Posteriormente o processo é dirigido pelo controlador da simulação, que, dependendo da fase de execução, passa o controle para o monitor de filas, e este para o módulo de enfileiramento de cada entidade, ou de volta para o bloco de *start*.

A seguir são descritas as ações de cada um dos módulos formadores do servidor de simulação.

Módulo *START*

1- inicializa o sistema perante o usuário,
2- solicita, do usuário, a indicação das entidades e bases de dados que compõe o sistema a ser simulado.
3- solicita, do usuário, as condições de término da execução da simulação,
4- carrega a base de dados desejada e monta as árvores balanceadas e as tabelas de dados.
5- dispara o procedimento do controlador de simulação,
6- apresenta o *trace* da simulação,
7- encerra a simulação.

No carregamento da base de dados do sistema são carregados os *frames* de etidades, *satus*, monitor, matriz de ligação, condições de terminação e *trace*. Destes, apenas o *frame* Monitor é fixo e ligado ao sistema, sendo que os demais formam parte integrante da simulação e podem ser alterados quando alguma modificação se faça necessária nas entidades em estudo.

Outro ponto a ser observado é que ao ser feita a carga do *frame* de *status*, este seja lido com condições iniciais, isto é, sem nenhum evento já executado previamente e *clock* zero. Eventualmente pode-se querer prosseguir uma simulação já executada, o que tornaria necessário uma declaração específica para a carga do *frame status* a partir de condições que não as iniciais.

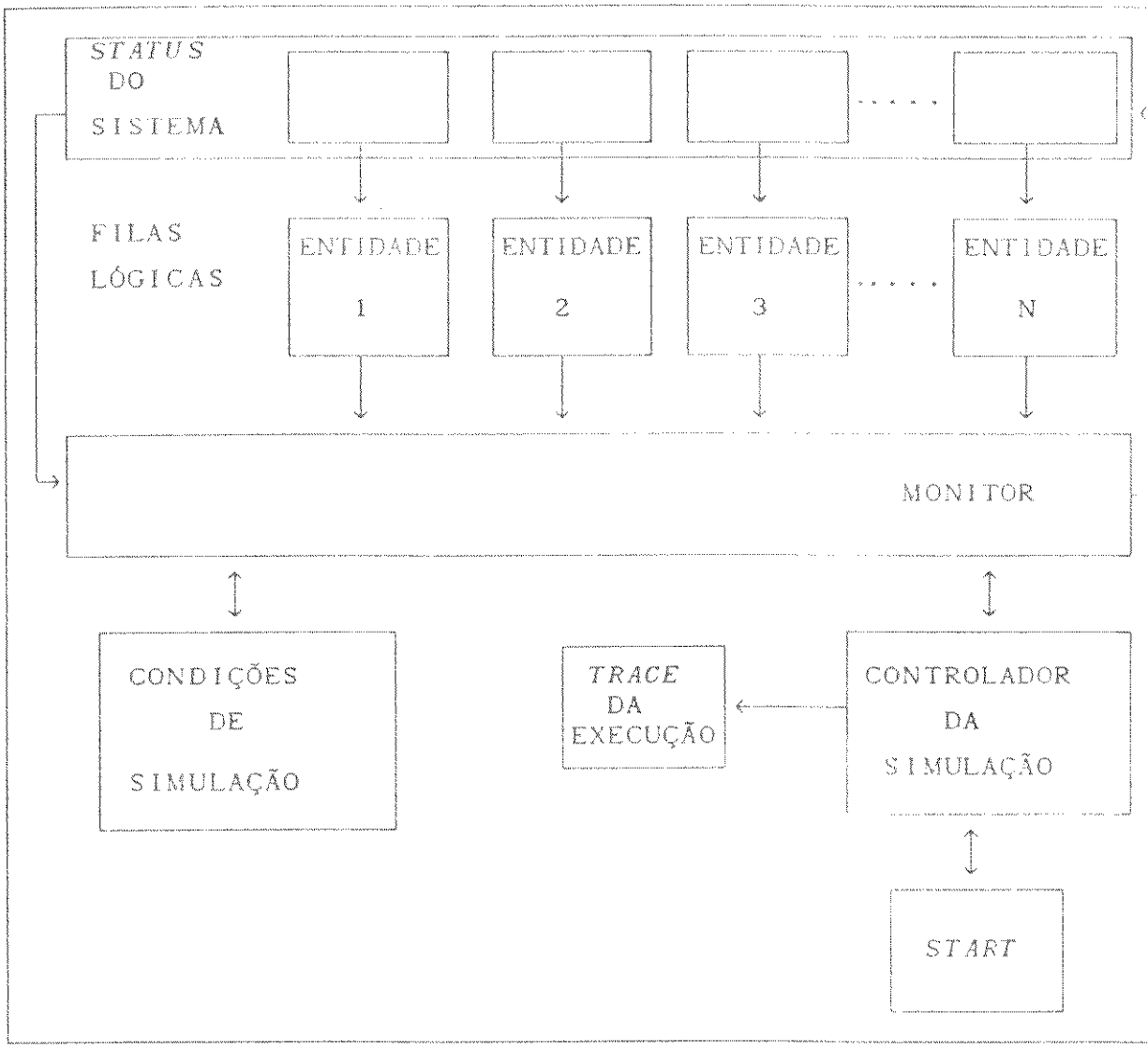


Figura 4.12 : Lógica de Simulação

Módulo CONTROLADOR DA SIMULAÇÃO

O módulo Controlador corresponde ao gerenciador do sistema de simulação, executando o evento a ele apresentado pelo monitor, controlando o *clock* e o endereçamento do evento executado para o *trace* da simulação e para o *frame* de *status*.

As ações por ele executadas são descritas a seguir.

1- Consulta no *frame* monitor o *slot* monitor, sob atributo <*if_needed*> e dispara o procedimento procedimento_de_monitoração.
2- Consulta *slot* <evento_a_executar>, sob atributo <*value*> e obtém o evento a executar e seus argumentos.
3- Lê o argumento instante_execução deste evento e o coloca no *slot* <*clock*>, sob atributo <*value*>, do *frame status*.
4- Consulta *frame trace*, *slot* entidades, *sub-slot* entidade?, agregando aí o novo evento e seu *clock* às listas respectivas.
5- Consulta *frame status*, *slot* entidade?, sob atributo <*value*> e escreve o evento executado e seus argumentos.
6- Consulta o *frame* condições de terminação, *slot* condições_terminação sob atributo <*rule*> e dispara a base de regras para análise das condições de terminação.
7- Caso não haja condição terminal o processo é reiniciado, senão é encerrada a simulação.

Módulo MONITOR DE FILAS

Este módulo é o responsável pela leitura do evento apresentado por cada entidade para ser executado, procedendo a uma arbitragem entre eles, de forma a apresentar um evento por vez para ser realmente "executado" pelo controlador da simulação.

Suas ações ocorrem conforme descrito a seguir.

1- Consulta o *frame status*, no *slot* entidades_sistema, sob atributo <*value*> para saber quais as entidades que estão ativas na presente ciclo de simulação.
2- Dispara o procedimento de enfileiramento para que este escolha em cada entidade qual o evento a executar. Este evento é escrito no *slot* execução_evento de cada entidade.
3- Consulta, em cada entidade, o *slot* execução_evento, sob atributo <*value*> para obter o evento passível de execução, obtendo-o juntamente com seus argumentos.
4- Busca, entre todos os eventos obtidos, o de menor instante_execução.
5- Caso haja mais de um evento com mesmo valor de instante_execução, consulta, em seu *frame*, o *slot* condições_de_simulação, sob atributo <*rule*>, para poder reconhecer qual base de regras deva consultar e proceder ao arbitramento de qual evento deva ser executado primeiro. Este processo continua até que reste um único evento passível de execução.
6- Coloca o evento escolhido para execução, junto com seus argumentos, em seu *frame*, no *slot* evento_a_executar, sob atributo <*value*>. 7- Encerra sua execução, retornando o controle de execução da simulação ao Controlador de Simulação

. Bloco de ENFILEIRAMENTO LÓGICO

Este bloco trabalha exclusivamente interno a cada entidade, procedendo ao enfileiramento e escolha do evento passível de ocorrer, a cada instante, nesta entidade.

As ações por ele executadas são descritas a seguir.

1- Consulta o *frame status*, *slot* entidade?, sob atributo <*value*>, para obter o último evento executado, com seus argumentos.
2- Consulta o *frame status*, *slot clock* para obter o tempo atual de simulação.
3- Lê, do último evento executado, o argumento que define seu posicionamento dentro da rotina.
4- Consulta o *slot* rotina, *sub-slot* rotina? para obter o próximo evento passível de execução.
5- Em caso de processo decisório (/), consulta o *slot* decisão, *sub-slot* rot?, sob stributo <*rule*> e dispara as regras para efetuar tal decisão.
6- Consulta o *slot* tabela, *sub-slot* rot?, sob atributo <*value*>, para obter o nome do evento.
7- Consulta o *slot* eventos, *sub-slot* evento?, para obter o evento e seus argumentos.
8- Consulta o *slot* regras e dispara a base de regras para avaliar o evento quanto à sua executabilidade, se a avaliação estiver habilitada pelo evento.
9- Caso o evento não seja executável, nada é posto sob atributo <*value*> no *slot* execução_evento da entidade, sendo encerrado o procedimento de enfileiramento e o controle de execução devolvido ao controle do monitor. 10- Executa a atualização/ressincronização do argumento instante_execução do evento em questão.
11- Coloca o evento e seus argumentos sob atributo <*value*> do *slot* execução_evento da entidade anexando os argumentos que indicam a posição na rotina, a entidade e a roitna pesquisada.
12- Encerra a execução do enfileiramento, passando o controle ao monitor.

4.2.4.2 - Estrutura do SMAC na Simulação

A implementação do servidor de simulação fez uso de todas as formas de representação de conhecimento oferecidas pelo sistema SMAC. Foram usados *frames*, regras e procedimentos para efetivar praticamente a lógica de simulação apresentada na seção anterior.

A descrição das entidades é feita através de *frames* independentes, de forma que é possível descrever quantas entidades se desejar para depois agrupa-las formando as células que melhor convier ao usuário.

Uma vez escolhidas as entidades, são definidos os *frames* variáveis do sistema. São os *frames* que mudam a cada simulação . São eles: *status*, matriz de ligação, condições de terminação e *trace*.

A parte fixa e invariante do sistema é constituida pelo *frame* monitor e pelo Programa Objeto onde estão codificados os módulos de *start* e controlador da simulação.

O bloco dos procedimentos ligados às regras, ProcedBD, é variável e depende do conjunto de entidades escolhido. No caso , contém apenas procedimentos genéricos para verificação de estado das *flags*, escrita em *slots* e leitura das portas de entrada e saída das entidades.

O bloco dos procedimentos ligados aos *frames*, *ProcedFR*, contém uma parte fixa e outra variável. A parte fixa, que passa a ser compilada quando considerada definitiva e sem erro, é constituida pelos procedimentos monitor e enfileiramento, ambos ligados ao *frame* monitor. A parte variável, que continua sempre interpretada, contém os procedimentos dos eventos de cada entidade utilizada na simulação.

As regras foram reunidas em um único conjunto com bases independentes para as entidades (regras para enfileiramento de eventos e decisão de rotina), condições de terminação e condições de simulação. É interessante formar conjuntos relativos a várias configurações de entidades. O que varia em cada célula, além das características dos elementos, são as condições de terminação que, por ser um *frame* independente, pode ser definido individualmente para cada célula criada, e as condições de simulação, que é uma base de regras ligada ao *frame* monitor e pode ser facilmente alterada via módulo *start* a cada simulação que se deseja efetuar.

A figura 4.13 mostra esquematicamente cada parte do simulador distribuida pelos elementos do sistema SMAC.

O protótipo SMAC-Simulador deve ter agregado como novos procedimentos, algorítmos de análise estatística para averiguação de vários fatores dentro das células simuladas, com previsão para saída gráfica e numérica. Também algorítmos para efetivação de animação gráfica, a partir do uso de ícones especificados pelo sistema e/ou pelo usuário, para os quais será utilizado o esquema representativo do SMAC para definição e utilização.

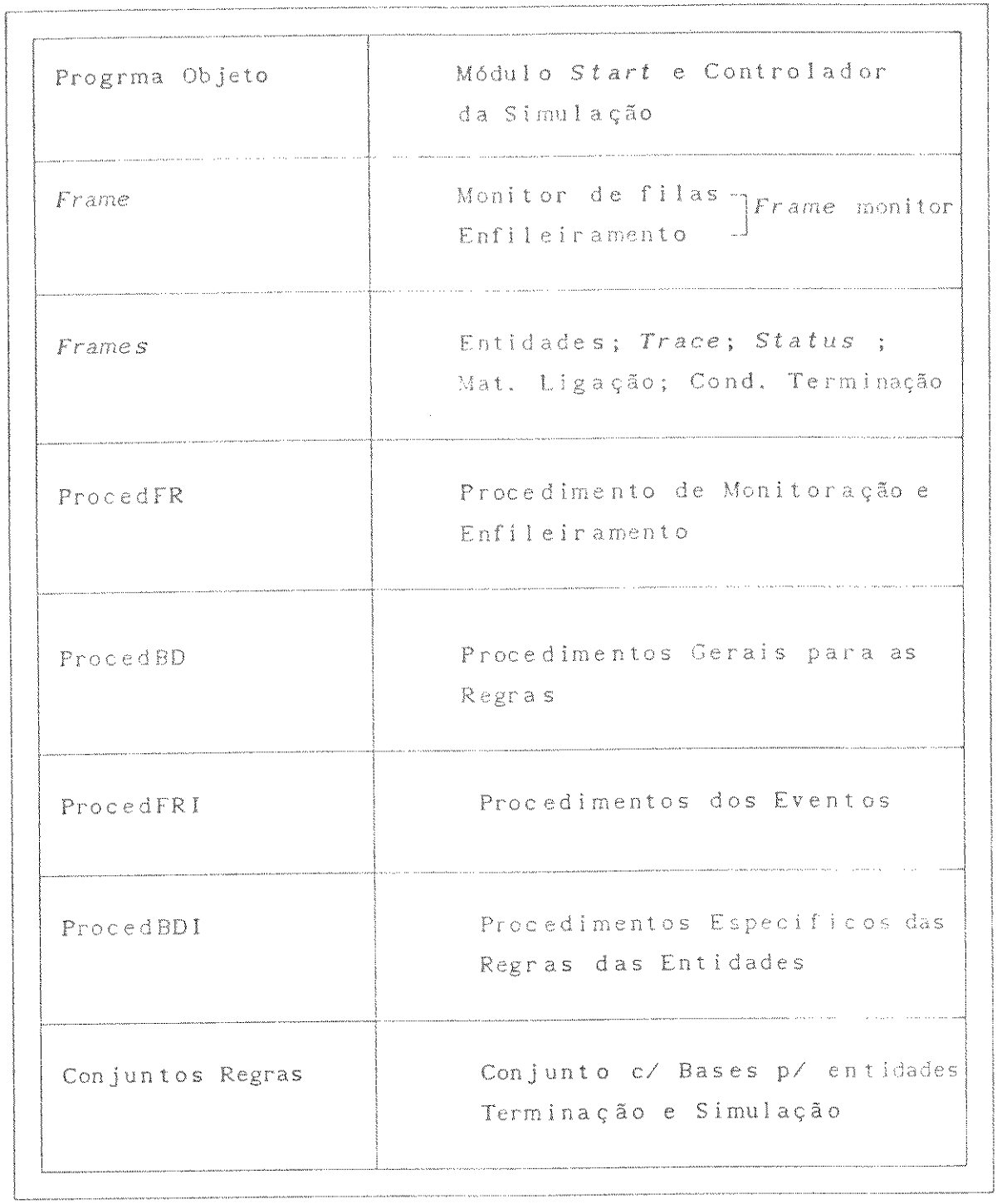


Figura 4.13 : Estrutura do SMAC-Simulador

4.3 - Projeto de Sistemas de Controle

4.3.1 - Introdução a um Ambiente para Projeto em Sistemas de Controle

Um ambiente para projeto auxiliado por computador em engenharia de controle foi definido [17,56] como a composição de um conjunto de ferramentas para projeto, análise, especificação, validação e implementação. Todas estas ferramentas, associadas a uma linguagem orientada para o problema como também a uma base de conhecimento, a qual deve conter conhecimento sobre as ferramentas que estão disponíveis e conhecimento suficiente para auxiliar o usuário na formulação e na solução parcial ou total do problema.

O sistema deve ser visto como um ambiente que mantém em harmonia o conhecimento, as ferramentas e as informações gerais sobre o próprio sistema e seu potencial.

Dentro de um sistema baseado em conhecimento identifica-se ao menos dois tipos de modelos independentes: o formal e o conceitual. O modelo formal é usualmente obtido da representação matemática e a partir do uso de ferramentas e algorítmos fornecidos pela teoria de controle. Já o modelo conceitual contém informações estritamente relacionadas ao modelo formal mas que não podem ser declaradas através de uma linguagem formal. São informações do tipo: limites de aplicabilidade do modelo, quão preciso é o modelo em relação ao sistema real, e as possíveis variações e extensões que podem ser aplicadas. Além disto, existem informações do modelo conceitual que não estão tão intimamente ligadas ao modelo formal, as quais se manifestam sob a forma de conhecimento intuitivo sobre relações, modos de operação, etc e, aqui, a força da representação formal não consegue ser abrangente o suficiente [17]. Um problema interessante e fundamental para o sistema considerado é como conciliar a modelagem formal com as ferramentas para a modelagem, análise e projeto juntamente com o modelo conceitual, cujas informações são geradas da experiência e da heurística.

A tentativa de solucionar o problema da falta de um vínculo entre o modelo formal e o conceitual vem motivando várias abordagens e implementações computacionais que levam este problema em consideração, utilizando uma combinação de regras de produção e *frames* [17,56].

Não se pode esperar para um futuro muito próximo a implementação de funções de alto nível de inteligência apesar do esforço concentrado da pesquisa nessa área. Uma possibilidade a ser aplicada ao problema de engenharia de controle é a *inferência dedutiva*, processo que extrai fatos individuais a partir de um ou vários conjuntos de regras gerais.

Também deve-se levar em consideração que existem dois tipos diferentes de problemas, os quais requerem abordagens de processamento completamente distintas. O primeiro é a *análise*, onde o problema é apresentado a partir de um sistema específico do qual são obtidos algumas propriedades e comportamentos. O segundo é a *síntese*, onde são apresentados um conjunto de propriedades e comportamentos, desejando-se obter a estrutura capaz de oferecer tais resultados. As funções que vão trabalhar sobre estes dois tipos

de modelos para transformá-los em um nível mais baixo na estrutura computacional, devem ser constituidas de uma boa dose de "inteligência" a fim de obter eficiência e generalidade em relação especialmente ao modelo apresentado.

No escopo em que o problema foi apresentado até aqui, a única possibilidade de se usar a *inferência dedutiva* como função inteligente para resolver o problema, é um sistema onde o usuário trata o problema interativamente com a máquina, manipulando o modelo construido, enquanto esta, através de uma base de conhecimento, procedimentos e dados disponíveis, serve de guia. Logicamente, uma linguagem que permita definir, contruir e modificar o modelo é necessária. Neste aspecto, uma linguagem orientada para o problema é um começo razoável.

4.3.2 - Estruturação do Ambiente

Uma das maiores dificuldades dentro da engenharia de controle é a formulação do problema, mesmo que este seja simples. Dentro da máquina é necessario fazer com que o problema se torne uma instância particular de uma representação mais geral. Uma maneira de abordar este caso é decompor a formulação do problema em módulos. Dentro da formulação o conhecimento descritivo é predominante; está contido nas definições, fatos e características particulares de cada caso. A forma básica utilizada para representar o conhecimento é relacionada ao conceito de *frames* [56]. A decomposição em *frames* lógicos propostos é a seguinte: Sistema, Modelagem, Requisitos, Solução, Validação e Implementação.

A idéia principal da estrutura é poder utilizar várias formas do mesmo *frame*, ou seja , instâncias diferentes de um mesmo *frame* que representam problemas diferentes ou, em outra análise, descendentes de algum *frame* mais genérico constituindo uma árvore. Nesta estrutura utiliza-se a idéia da generalização-especialização onde o problema tem sua formulação cada vez mais detalhada à medida em que se desce de nível dentro da árvore.

O conjunto de *frames* para gerar um sistema para projeto em sistemas de controle é discutido a seguir.

Frame de Sistema : É um modelo lógico do mundo real da engenharia de controle. Representa os principais componentes como a planta, sensores, atuadores, controladores e ambiente geral onde o sistema se encontra. A figura 4.14 mostra uma estrutura posível para este *frame*.

Frame de Modelagem : Fornece as principais características do modelo em termos de parâmetros, ordem ou seus valores estimados quando eles são desconhecidos. Neste caso novas informações serão adicionadas para que os procedimentos de estimação sejam ativados e executados. A figura 4.15 mostra uma construção possível.

```
        Name : <value> --> sistema
        version     : <value> --> 1
        date_time   : <value> --> 04-11-89
        members     : <value> --> [ ]
        a_kind_of   : <value> --> raiz
descrição : simbolica : <if_needed> --> [simb_proc,_,..._]
            formal : eq_dif : <if_needed> --> [eq_dif_proc]
                     espa_estado : <if_needed> --> [..]

            func_tranf : <if_needed> --> [f_trans_proc]
            diagr_bloco : <if_needed> -->[...]

comportamento : dom_tempo : over_shoot : <value> --> [..]
                            temp_estab : <value> --> [..]
                              ...

                dom_freq : banda_pas : <value>  --> [..]
                           freq_corte : <value> --> [..]
                             ...

                root_locus : ...

tipo_sistema : linear : <value> ..
               discreto : <value> ..
               continuo : <value> ..

perturbação : ...

    ....
```

Figura 4.14 : *Frame* de Sistema

Frame de Requisitos : estabelece um conjunto de especificações de projeto onde se reconhece o comportamento desejado para o sistema em questão. A figura 4.16 ilustra a idéia.

Frame de Solução : descreve um conjunto de ferramentas de projeto fornecendo uma conexão entre o projetista e os vários procedimentos e algorítmos disponíveis. É responsável pela informação de *status* do sistema ao longo do tempo em uma seção de trabalho. A figura 4.17 mostra uma construção possível.

```
        Name : <value> --> modelagem
        version    : <value> --> 1
        date_time  : <value> --> 04-11-89
        members    : <value> --> [ ]
        a_kind_of  : <value> --> raiz
parametros : <    >   ...
ordem : < > ....
bias : valor_DC : <value> ...
          ...
estimação : deterministica : <   > ...
            estocástica : < > ...
intervalo_validade : <   > ...
          ....
```

Figura 4.15 : *Frame* de Modelagem

Frame de Validação : estabelece critérios e procedimentos que permitem verificar se o *status* corrente do sistema está de acordo com as metas que deveriam ser atingidas. Trata-se apenas de um *frame* avaliador de resultados. A figura 4.18 ilustra a idéia deste *frame*.

Frame de Implementação : é um guia que permite gerar um código executável, fornecendo informações sobre limites, no tocante a características computacionais, como por exemplo, comprimento da palavra, velocidade de memória, tempo de processamento, etc.

A descrição dada limita-se ao modelo conceitual, porém outros dados podem ser incluídos, os quais, apesar de não serem explicitamente utilizados, podem ser importantes em alguns casos. Por exemplo, um problema não linear pode ser considerado linear dentro de um determinado limite imposto pelo usuário. Assim, ferramentas para uma análise linear podem ser usadas e no final, os resultados obtidos podem ser verificados com os esperados de um sistema não linear, chegando a algumas conclusões (auxiliado pelo *frame* de validação).

Dentro dos *slots* podem estar alocados valores, procedimentos ou regras. Ao ativar um procedimento ou uma base de regras, outros *slots* em outros *frames* podem ser ativados e seus procedimentos ou bases de regras disparados e assim sucessivamente. Desta forma o problema particular analisado vai apresentar um conjunto de ligações entre os *frames*, *slots*, procedimentos e regras inerentes a ele. Esta forma com que as ligações acontecem, estabelecem o problema como uma instância particular da estrutura mais geral abrangida pela base de *frames*.

```
        Name : <value> --> requisitos
        version    : <value> --> 1
        date_time  : <value> --> 04-11-89
        members    : <value> --> [ ]
        a_kind_of  : <value> --> raiz
  modelagem : bias : <...>  ..
              variações : <..>
              convergência : <..>
  controle  : banda_pass : <..>
              over_shoot : <..>
  restrições :  modelo : <..>
                controle : <..>
                  ...
    ....
```

Figura 4.16 : *Frame* de Requisitos

```
        Name : <value> --> solução
        version    : <value> --> 1
        date_time  : <value> --> 04-11-89
        members    : <value> --> [ ]
        a_kind_of  : <value> --> raiz
  model0 : estático : minim_quadra : <..>
                      mat_extendida : <..>
           não_estático : fator_esquec : <..>
                          traço_adaptativo : <..>
  controle : clássico : PID : <..>
                        feedforward : <..>
                        compensador : <..>
             moderno : LQ : <..>
                       LQG : <..>
                         ...
```

Figura 4.17 : *Frame* de Solução

```
        Name : <value> --> validação
        version     : <value> --> 1
        date_time   : <value> --> 04-11-89
        members     : <value> --> [ ]
        a_kind_of   : <value> --> raiz

modelo : compara : <..>

controle : compara : <..>

entrada : compara : <..>

saída : compara : <..>

Intervalo_de_validade : compara : <..>
```

Figura 4.18 : *Frame* de Validação

4.3.3 - SMAC - CAD

Na implementação deste sistema, as classes são fixas em suas estruturas básicas. Alguns problemas mais complexos podem requerer outras informações que podem ser agregadas na classe em que forem necessárias durante o tempo de execução.

Os conjuntos de regras devem tratar as mais diversas características dentro do contexto do projeto de sistemas de controle, tanto na fase de definição de parâmetros que estabelecem o sistema a ser projetado ou analisado, como na fase de ajuste destes parâmetros visando um ajuste fino para um bom funcionamento. Além disto, regras para auxílio ao projetista diante das várias situações que se apresentem durante o projeto devem ser fornecidas.

Os procedimentos devem cobrir todas as ferramentas usualmente utilizadas, tanto na síntese como na análise de projetos de sistemas.

A união dos procedimentos com os conjuntos de regras define um ambiente onde se pode alterar o estado do problema e daí realizar uma averiguação da situação para, dependendo da situação, decidir que nova alteração deva ser feita, que procedimento acionar, ou ainda pesquisar algum conjunto de regras para uma análise mais criteriosa de situação.

Nos *frames* serão armazenados os resultados e informações até que a solução seja montada de uma maneira que, do ponto de vista do sistema e do

usuário, seja considerada satisfatória. Neles estarão estruturas fixas que definem o ambiente de projeto e informações pertencentes ao problema em particular que está sendo tratado no momento. Pela modularidade oferecida pelo SMAC, esse conjunto pode ser armazenado para posterior uso ou para completar, em um outro momento, um processamento que tenha sido interrompido.

Cada problema pode ser então armazenado criando-se uma biblioteca de sistemas sintetisados ou analisados. Desta forma, problemas novos podem ser facilmente resolvidos pela composição das informações já conhecidas e armazenadas nas respectivas classes.

A estrutura de *frames* que define o ambiente, sem as informações pertinentes a nenhum problema particular, devem estar sempre presentes, para permitir que novos problemas possam ser estruturados sem que nenhuma informação já conhecida seja usada.

O Programa Objeto será o *gerenciador* de todos os passos efetuados pelo sistema. Promoverá o preenchimento do *frame* de sistema e, à medida em que os dados estiverem presentes, dispara os procedimentos e regras relativas ao conjunto de dados conhecido e ao conjunto desejado. Para efetivar todo o gerenciamento, o Programa Objeto deverá contar com procedimentos e conjunto de regras para uso direto, ou seja, sem conexão com nenhum *frame*.

Pode-se optar por criar um *frame* de gerenciamento para o sistema, para conter as regras e os procedimentos relativos a esta função. Este *frame* seria fixo, inalterável e ligado ao sistema principal, não pertencendo à biblioteca de nenhum problema em particular.

4.4 - Extensões ao SMAC

O sistema SMAC ainda necessita de uma série de melhoramentos para tornar-se um produto no sentido exato da palavra. O objetivo deste trabalho é o de mostrar a viabilidade de uma idéia nova e basicamente diferente da existente na literatura até então. O objetivo foi alcançado e demonstrado na prática com as implementações realizadas.

A seguir serão apresentadas algumas extensões a serem inseridas no sistema com a intensão de torná-lo mais versátil e abrangente, bem como facilitar seu uso, mesmo para os menos experientes. Estas extensões, em alguns casos, já estão em andamento para implementar um servidor de simulação completo [39,40].

Editor de *Frames* : Programa auxiliar que pode ser acionado antes e durante a execução. A segunda opção é decidida durante a programação do sistema. Permite a edição completa de qualquer classe, visualizando a estrutura a nível de classe, e sub-classe, a nível de *frame* mostrando as ramificações dos *slots* até os ramos folhas, e a nível de tabela de dados. Permite efetivar alterações em todos os níveis visualizados, sendo possível uma armazenagem definitiva da alteração. A liberdade para alterar as classes durante a execução é responsabilidade do programador do sistema, tendo em vista a aplicação para o qual o SMAC esteja sendo configurado.

O objetivo principal é fornecer ao programador uma ferramenta de alto nível para montar a base de *frames* com todas as facilidades de um bom editor que permita : criar, alterar e visualizar o que já existe pronto.

Editor de Regras : Semelhante ao anterior só que relacionado aos conjuntos de regras. Todas as características fornecidas à base de *frames* devem ser fornecidas para os conjunto de regras.

Verificador da Base de Conhecimento : Programa auxiliar que verifica a consistência da base de conhecimento geral (*frames*, dados, regras e procedimentos). A verificação se dá a nível sintático, ou seja, se os procedimentos indicados realmente existem, tanto para os *frames* como para as regras e Programa Objeto (usam-se comandos de teste em cada procedimento), se os dados estão configurados corretamente, se os conjuntos e bases de regras que podem ser requisitadas realmente existem, etc. Informa ao usuário/programador se o conheciemtno disponível está interligado de forma que o sistema possa ser acionado. A verificação lógica ou semântica do conhecimento deve ser feita pelo próprio programador à medida em que o sistema vai sendo usado.

Essas ferramentas são mais necessárias durante o processo de configuração do SMAC para atender uma detereminada aplicação. Ou seja, quando os procedimentos estão ainda interpretados (exceto aqueles feitos em outras linguagens, para os quais somente as interfaces estarão interpretados) e quando as regras também forem interpretadas. A partir do momento em que se encerra a configuração, o uso desses utilitários se reduzirá ao caso em que alguma alteração ou visualização da base de *frames* ou regras se faça necessária.

Manipulação de Incerteza : Associar aos fatos e regras coeficientes de certeza, os quais indicarão até que ponto a informação é confiável. Desta maneira, todas as conclusões seriam manipuladas dentro de um intervalo de certeza. Esta manipulação pode fazer uso de mecanismos consolidados na lógica *fuzzy* [31,42].

Filosofia *Blackboard* : Permitir, como já comentado anteriormente, que resultados e conclusões de outras inferências sejam usados e alterados por novos conjuntos de regras e novas inferências. Isto permitiria que algumas conclusões seguissem o curso de solução do problema e, talvez, se tornassem parte desta solução.

Processamento Temporal : Associar aos *frames* e aos conjuntos de regras a dimensão tempo, para que este fator seja considerado como ocorre no mundo real. Isso permitirá utilizar e tratar informações alteráveis com o tempo, tanto nos *frames* como nas regras. Com este tipo de tratamento, a base de conhecimento passa a ser não estática em relação ao tempo, mas sim função dele. O fator temporal seria importante, por exemplo, na implementação da filosofia *blackboard*.

Regras Compiladas : Inserir a possibilidade que, ao longo da configuração e uso do SMAC em uma aplicação, os conjuntos e bases de regras que se tornarem decididamente fixos possam ser compilados junto com o programa principal. Isso resultaria num programa mais rápido e mais fácil de manipular, pois a fração alterável do sistema se tornaria menor.

Procedimentos Executáveis : Utilizar a possibilidade da memória expandida presente em sistemas operacionais mais modernos, para permitir que procedimentos executáveis possam ser acionados. Estes procedimentos seriam ativados pelo sistema principal e usariam uma área reservada de memória. A comunicação entre o sistema e os procedimentos seria via arquivos de dados estruturados de forma padronizada.

Multiplicidade de Conj. de Regras na Mem. de Trabalho : Permitir que mais de um conjunto de regras esteja presente dentro da memória de trabalho. Cada conjunto teria sua área independente de trabalho. Seria implementada uma área de acesso comum onde vários conjuntos poderiam comunicar-se entre si e efetivar uma arquitetura tipo *balckboard*. Além da possibilidade de comunicação, uma maior rapidez e eficiência seria atingida quando da ativação dos conjuntos de regras.

Encadeamento de Bases ou Conjuntos de Regras : Estruturar os procedimentos ligados às bases de regras de forma que estes possam acionar outros conjuntos ou bases de regras, para solucionar partes de um problema maior. Quando se tratar de uma outra base contida no mesmo conjunto, deve-se abrir um novo segmento de memória de trabalho para que a nova inferência se processe e assim sucessivamente, permitindo que várias inferências sejam encadeadas até que alguma conclusão seja encontrada. No caso de um novo conjunto ser ativado, este é sobreposto ao anterior sem que o primeiro seja apagado, criando-se um novo segmento completo de memória de trabalho. Assim, vários conjuntos podem ser encadeados sucessivamente. Cada um desses novos conjuntos ou bases ativados, podem estabelecer comunicação com as bases ou conjuntos anteriores através da memória de acesso comum. Torna-se possível ativar conjuntos e, dentro destes, ativar várias bases e assim sucessivamente.

4.5 - Resumo

Neste capítulo mostrou-se que o sistema SMAC é uma realidade prática, que pode ser muito útil em várias aplicações onde as características inerentes a este sistema sejam necessárias. O tempo para configurar o SMAC é também um fator importante, já que, para gerar e testar uma configuração do SMAC, gasta-se um tempo muito menor do que o gasto no desenvolvimento de um aplicativo específico.

As extensões citadas pretendem aprimorar o sistema SMAC no sentido de torná-lo um produto de fácil utilização e raio de ação cada vez maior e mais eficiente.

O próximo capítulo conclui o presente trabalho mostrando uma síntese do que foi atingido e detalhando perspectivas futuras dentro da área.

Capítulo 5 : Conclusões

5.1 - Introdução

Neste capítulo apresenta-se uma finalização do trabalho descrito ao longo dos demais capítulos. É aberta uma discussão sobre a validação do ambiente proposto, sobre a situação deste no panorama atual das pesquisas e desenvolvimentos em IA, sobre as limitações que atualmente fazem parte do sistema, sobre extensões em desenvolvimento que futuramente devam ser atingidas. São discutidas, também, áreas de interesse para projeções futuras, tanto do sistema aqui discutido, como de trabalhos correlatos.

5.2 - Conclusões e Trabalhos Futuros

A seção 1.4 descrita no capítulo 1, mostra uma perspectiva do sistema SMAC, no que diz respeito à arquitetura e forma de manipulação de conhecimento, inserida num panorama atual de alguns outros trabalhos presentes na literatura.

A arquitetura do sistema utiliza estruturas basicamente convencionais para a representação e manipulação de conhecimento. A proposta inova na forma em que as estruturas de *frames*, regras e procedimentos são agregadas e utilizadas pelo sistema. Essencialmente, as estruturas não são usadas ou modeladas para atender a um tipo específico e definido de conhecimento, que seja oriundo de alguma aplicação em particular. O interesse é criar subsídios para que uma forma abstrata e genérica de conhecimento possa ser expressa fácil e eficientemente dentro das possibilidades oferecidas pelo sistema. Uma vez criada a base de conhecimentos, a manipulação desta deve permitir a obtenção de conclusões e resultados advindos de inferências, de forma que uma aplicação também genérica possa ser atendida.

O sistema SMAC apresenta-se como um ambiente em que uma vasta gama de conhecimentos podem ser trabalhados e representados. Essa possiblidade permite configurar o sistema para atender às mais variadas aplicações.

A implementação do protótipo SMAC-Simulador descrito no capítulo 4, permitiu verificar, na prática, os atributos discutidos na teoria em relação ao sistema. A possibilidade da estruturação de classes utilizando *frames* abre o escopo para uma descrição detalhada do ambiente de manufatura a nível de células flexíveis. O recurso da herança e a liberdade na construção das ramificações da árvore de cada classe, permite uma descrição especializada ao nível que mais se achar necessário e, além disso, acrescenta à base de *frames* um caráter dinâmico, deixando de ser um depósito estático de informações e passando a ser uma estrutura ativa dentro da inferência em busca de conclusões. A utilização de regras escritas na linguagem coloquial e ainda com a possibilidade de definição de operadores próprios e específicos, permite, tanto ao programador descrever, como ao usuário entender, todo o contexto do conhecimento presente dentro dos conjuntos e bases de regras. A facilidade de associar procedimentos às instâncias e às regras permite a criação de procedimentos associados a cada evento descrito dentro de cada classe. Assim, um evento executado é representado por uma ação efetivamente realizada pelo sistema e não somente um fato representativo sem valor funcional.

Tanto a implementação do SMAC-Simulador, como a estruturação do SMAC-CAD, vislumbra um campo promissor para o tipo de arquitetura proposta. Melhoramentos são necessários. Porém a estrutura básica para manipulação e representação do conhecimento mostrou ser eficiente, fácil de ser manuseada e, sobretudo, suficientemente genérica para ser considerada aberta de forma a servir às mais diversas necessidades.

Da forma como se apresenta atualmente, o sistema SMAC contém algumas limitações. Por se tratar de um protótipo, a interação com o usuário a nível de interface homem-máquina apresenta-se constituída pelo mínimo necessário ao funcionamento. A construção dos programas auxiliares como o editor de *frames*, o editor de regras e verificador da consistência da base de conhecimento e

dados, discutidos no capítulo 4, seção 4.4, são passos importantes na direção de um ambiente amigável em que seja facilitado o trabalho do programador como também do usuário.

Um fator limitante em termos da utilização de conhecimento é o fato do Sistema Especialista trabalhar somente com encadeamento reverso. A implementação do encadeamento direto permitiria estruturar um esquema de trabalho onde inferências diretas poderiam ser conjugadas com inferências reversas e vice-versa. A partir da existência dos dois tipos de encadeamento com regras, seria possível a implementação do encadeamento misto. Isso daria, à estrutura de regras, mais versatilidade, tanto no que diz respeito à quantidade de conhecimento possível de embutir no sistema, como também na manipulação do mesmo.

Outra limitação é a impossibilidade do múltiplo encadeamento de inferências. No capítulo 4 seção 4.4, uma solução é discutida e também sugerida a utilização de múltiplos conjuntos de regras simultâneas na memória de trabalho do SE.

A discussão das limitações e as possibilidades para que estas sejam, se não eliminadas, pelo menos reduzidas, traz à tona uma série de temas que podem ser direcionados ao sistema tornando-o mais versátil e abrangente.

A idéia discutida anteriormente sobre as inferências encadeadas e os múltiplos conjuntos de regras simultaneamente presentes na memória de trabalho do SE, aponta para a filosofia *blackboard*. Essa possibilidade, já previamente discutida na seção 4.4, é atraente por ser uma filosofia relativamente moderna cujas tendências apontam para um futuro promissor. Dentro do sistema SMAC, a utilização da memória comum pode ser feita pelas bases de regras, durante um inferência normal, ou seja, procurar conhecimento que esteja disponível nesta memória, que permita efetivar uma conclusão dentro do escopo da base de regra consultada; pode ser manipulada via procedimentos das regras, pela base de *frames* ou mesmo diretamente pelo Programa Objeto. A estrutura do conhecimento presente nesta memória deve conter informações relativas à sua origem, idade e tempo de vida, para que possa ser utilizada por outras frações independentes do sistema. Além disto, a memória comum pode ser particionada em níveis específicos a serem programados e definidos pelo usuário, de forma que haveria possibilidade de fazer acesso à informação de forma seletiva, levando em consideração o tipo de informação solicitada e quem é o solicitante desta informação.

A filosofia *blackboard* é essencialmente distribuida. A arquitetura do sistema SMAC apresenta-se distribuida em vários módulos, os quais são discutidos no capítulo 2. Cada parte do sistema é fechada sobre si mesma. A base de *frames* e regras pode ser intercambiada sem que o funcionamento seja prejudicado, desde que o padrão da comunicação entre os módulos seja mantido dentro do especificado. A maneira em que o SMAC foi estruturado, apresenta a característica que proporciona ao sistema a possibilidade de atender a vários propósitos diferentes, em locais diferentes, dentro de um mesmo sistema distribuido em rede. Cada ponto da rede pode solicitar os serviços do SMAC, fornecendo-lhe os módulos que são característicos daquele ponto, ou seja, o Programa Objeto, os módulos de procedimentos ligados às regras e nos *frames*, os conjuntos de regras e a base de *frames*. Cada ponto tem a sua base de conhecimento construida da forma que melhor se adapte ao tipo de conhecimento que deva ser representado. Desta forma vê-se o SMAC atuando em um sistema

distribuido onde , cada ponto, pode tratar assuntos completamente distintos. Vale salientar que a abordagem levada em consideração é em termos de estrutura do sistema SMAC e não da sua implementação técnica. Em termos de tecnologia, algumas alterações são necessárias para que o sistema possa trabalhar corretamente em uma rede.

Em termos de sistemas distribuídos seria necessário implementar uma interface de comunicação bem estruturada, tanto para servir à ligação usuário-sistema, como à ligação entre o sistema e as bases de conhecimento e dados que, neste tipo de sistema, pode ser, e quase sempre serão, remotas. O problema de garantir a integridade das informações manuseadas torna-se crítico. Essa interface deve ser suficientemente segura para permitir não só o acesso do SMAC às bases de conhecimento remotas, como também permitir que outros sistemas possam comunicar-se com o SMAC e solicitar-lhe serviços. Essa solicitação por parte de algum outro sistema pode ser consolidada fazendo com que o Programa Objeto seja executado pelo próprio sistema solicitante dos serviços, utilizando a interface para ativar os comandos do SMAC.

A idéia de um sistema distribuído é apresentada na descrição de um servidor de simulação para manufatura integrada por computador [39,40]. O SMAC-Simulador, discutido no capítulo 4, é um protótipo para construção deste servidor para redes industriais.

Tendo em mente que o objetivo é construir um ambiente que possa representar, o mais precisamente possível, a realidade para a qual foi projetado, a inserção do tratamento temporal no sistema SMAC faz parte da evolução lógica do sistema. A escala em que este fator será utilizadlo não terá muita influência, mas a utilização desta dimensão física dentro das inferências com o conhecimento traz à tona a idéia de vida útil das afirmações disponíveis. Isto é um fator fundamental na aproximação da manipulação do conhecimento executado no computador com o que aconteceria na realidade do ambiente de trabalho (ver capítulo 4, seção 4.4).

Ainda levando em consideração a aproximação com a realidade, é necessário pensar que todo o conhecimento disponível tem embutido em si uma certa quantidade de incerteza. Esta incerteza deve ser de alguma forma manipulada para que, desse modo, as conclusões obtidas possam ser consideradas dentro de um intervalo de precisão. Isto faria com que o funcionamento do sistema se tornasse parecido com a atitude de um especialista envolvido em seu ambiente de trabalho.

No contexto de colocar o sistema computacional em contato com o ambiente real para o qual ele foi projetado, uma forma de realizar esta junção seria através de um meio físico, ou seja, através de sensores. Estes sensores permitiriam, no caso do simulador de células flexíveis de manufatura, por exemplo, efetuar uma simulação como se máquinas reais estivessem operando. Numa situação como esta, seria possível visualizar o funcionamento da célula de uma maneira muito próxima ao que seria obtido programando a célula para funcionar do modo em que foi simulada. Isto corresponderia a trazer o "chão de fábrica" para dentro do computador sem que houvesse uma descaracterização da realidade.

Diante do ambiente real, um sistema baseado em conhecimento deve conter a maior quantidade possível de conhecimento armazenado para que seu funcionamento possa ser considerado, pelo menos, satisfatório. Dentro de

qualquer ambiente o conhecimento disponível não é totalmente explícito e nem tampouco, completamente definido. O especialista envolve-se com o ambiente, interage com o meio e absorve conhecimento sobre o qual não existe formalismo algum. É necessário fazer com que o ambiente computacional tenha ferramentas para efetuar um comportamento semelhante. Em extensões futuras é necessário dotar o SMAC de ferramentas que sejam capazes de analisar as conclusões obtidas das inferências, os fatos temporários (capítulo 2, seção 2.3.2), e deles extrair conhecimento em formas de regras, fatos e *frames*, que possam ser armazenados em conjuntos de regras e bases de *frames* especiais que são estruturados pelo próprio sistema. Estas regras e *frames* seriam pesquisados sempre que o conhecimento para realizar uma determinada inferência não seja suficiente chegar a uma conclusão ou para simples validação de uma conclusão obtida como forma de conseguir mais conhecimento.

A título de trabalhos futuros pretende-se aplicar o sistema nas mesmas áreas discutidas no capítulo 4, mas a níveis diferentes. O SMAC-Simulador mostrou-se um sistema de auxílio à decisão dentro do universo de uma célula flexível de manufatura. Diante da versatilidade da lógica de simulação e do próprio SMAC, pode-se pensar em utilizá-lo como sistema de auxílio à decisão a nível gerencial. Dentro deste novo universo, o conhecimento também pode ser representado eficientemente usando *frames*, regras e procedimentos. Os acontecimentos próprios do nível de gerência podem ser descritos como eventos discretos e manipulados pela lógica de simulação anteriormente descrita. Isso seria um avanço na configuração do servidor de simulação [39,40] onde a simulação pretende ser útil aos mais diversos níveis de uma indústria.

No tocante ao SMAC-CAD, o interfaceamento com uma linguagem de manipualção algébrica permitiria criar uma ferramenta poderosa, no sentido de estruturar um sistema para projeto auxiliado por computador para sistemas de controle. Este sistema, além das ferramentas próprias para sistemas de controle, seria útil em qualquer problema onde descrição e manipulação algébrica fosse necessária.

5.3 - Resumo

Neste capítulo foram discutidos aspectos relevantes dentro do panorama da IA, no sentido de tornar os sistemas baseados em conhecimento mais próximos da realidade dos ambientes para os quais foram projetados para servir. Foram discutidas algumas extensões para o sistema SMAC, tendo como base a experiência obtida na implementação do SMAC-Simulador. A estrutura do sistema é comentada no contexto do que existe na atualidade em questão de sistemas baseados em conhecimento. Também são apresentadas algumas sugestões de desenvolvimentos futuros tomando como ponto de partida o protótipo do SMAC-Simulador e a estrutura proposta para o SMAC-CAD.

Referências Bibliográficas

[1] - Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA.

[2] - Al-Zabaide, A.; Grimson, J. B. (1987). *Expert Systems and Database Systems: how can they serve each other?*. Expert Systems, vol 4, nº 1, fevereiro, pp 30-37.

[3] - Alptekin, S.; Weber, D. L. (1988). *A Systematic Transition to Flexible Manufacturing*. Computers and Industrial Engineering, vol. 15, nº 1, pp. 8-13.

[4] - Amaral, W.; Bingualc, S.; Comide, F.; et alii (1988). *A Knowledge Based Environment for Computer Aided Control Engineering*. Proc 4th IFAC CADCS'88, Beijing, China.

[5] - Arruda, L. V. (1988). *Um Supervisor Baseado em Conhecimento para Modelagem de Processos*. Tese de Mestrado, UNICAMP.

[6] - Badiru, B. A. (1988). *Expert Systems and Industrial Engineers: A Pratical Guide to a Successful Partnership*. Computers Ind. Engng., vol 4, nº 1, pp 1-13.

[7] - Bar & Feigenbaum, E. A. (1981). *The Hand-book of Artificial Intelligence*. Los Altos, California, William Kaufmann Inc., vol 1 e 2.

[8] - Barbuceanu, M. (1985). *An Object-Centred Framework for Expert Systems in Computer-Aided Design*. Knowledge Engeneering in Computer-Aided Design, J. S. Gero (Editor), Elsevier Science Publishers B. V. (North-Holland).

[9] - Basu, A.; Majumdar, A. K.; Sinha, S. (1988). *An Expert System Approach to Control System Design and Analysis*. IEEE Transactions on Systems, Man, and Cybernetics, vol 18, nº 5, stembro/outubro.

[10] - Bobrow, D. G.; Mittal, S.; Stefik, M. J. (1986). *Expert Systems: Perils and Promise*. Communications of the ACM, vol 25, nº 9, stembro.

[11] - Bratko, I. (1986). *Prolog Programming for Artificial Intelligence*. Addison Wesley Publishing Company, Menlo Park, California.

[12] - Casanova, M. A.; Giorno, F. A. C.; Fortado, A. L. (1987). *Programação em Lógica e a Linguagem Prolog.* Editora Edgar Blücher Ltda, São Paulo, SP.

[13] - Clocksim, W. F.; Mellish, C. S. (1984). *Programming in Prolog,* Springer-Verlang, Berlim.

[14] - Coehn, P.; Feigenbaum, E. A. (1981). *The hand-book of Artificial Intelligence,* Los Altos, California, William Kaufmann, vol 3.

[15] - Cohen, P. R.; Levesque, H. J. (1987). *Intention = Choice+Commitment.* Proceeding of 1987 Conference Of American Association for Artificial Intelligence, pp 410-415.

[16] - Cohn, P. G.; Gomide, F. A. C.; Marques, M. L.(1987). *Automação, Controle e Inteligência Artificial. Parte I: Uma visão Integrada.* SBA: Controle & Automação, vol 1, nº 4, pp 277-290.

[17] - Denham, M. J. (1984). *Design Issues for CASCD Systems.* Proceedings of IEEE, vol 72, nº 12, dezembro.

[18] - Engelmor, R.; Morgan, T. (1988). *Blackboard Systems,* Addison Wesley Publishing Co., New York.

[19] - Erman, L. D.; Hayes-Roth, F.; Lesser, V. R.; Reddy, D. R. (1980). *The Hersay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty.* Computing Surveys, vol 12, nº 2, junho.

[20] - Gallant, S. I. (1985). *Automative Generation of Expert Systems from Examples.* IEEE Proceedings.

[21] - Gasser, L.; Braganza, C.; Herman, N. (1987). *MACE: A Flexible Testbed for Distributed AI Research.* Distributed Artificial Intelligence, Pitman Publishing/Morgan Kaufmann Publishers, San Mateo, CA, pp 119-152.

[22] - Groover, M. P. (1987). *Automation Production Systems and Computer Integrated Manufacturing.* Prentice-Hall Inc.

[23] - Hayes, P. J. (1973). *Computation and Deduction.* Anais do 2nd Mathematical Foundations of Computer Science Symp., Czechoslovak Academy of Sciences, pp. 105-118.

[24] - Hayes-Roth, B. (1985). *A Blackboard Architecture for Control.* Artificial Intelligence, nº 26, pp 251-321.

[25] - Hayes-Roth,F.; Waterman, D. A. ; Lenat, D. B. (1983). *Building Expert Systems.* Addison-Wesley Publishing Co, Inc, Massachusetts, USA.

[26] - Hayes-Roth, F. (1985). *Rule-Based Systems.* Communications of the ACM, vol 28, nº 9, setembro, pp 921-932.

[27] - Hewitt, C. E. (1985). *The Challenge of Open Systems.* Byte, 10(4), abril, pp 223-242.

[28] - Hewitt, C. E. (1986). *Offices are Open Systems.* ACM Transactions on Office Information Systems, 4(3), pp 271-287.

[29] - Huang, Y. W.; Fan, L. T. (1988). *Designing an Object-Relation Hybrid Database for Chemical Process Engineering.* Computer Chem. Engng., vol 12, nº 9/10, pp 973-983.

[30] - James, J. R.; Frederick, D. K.; Taylor, J. H. (1987). *Use of Expert-Systems Programming Techniques for the Design of Lead-Lag Compensators.* IEEE Proceedings, vol 134, nº 3, maio.

[31] - Kaufmann, A.; Zadeh, L. A.; Swanson, D. L. (1975). *Theory of Subsets, volume 1 - Fundamental Theorical Elements.* Academic Press, Inc, New York.

[32] - Khan, N. A.; Jain, R. (1986). *Explaning Uncertainty in a Distributed Expert System.* Coupling Symbolic and Numerical Computing in Expert System. Elservier Science Publishers B. V., North-Holland.

[33] - Kohonen, T. (1984). *Self Organization and Associative Memory.* Springer-Verlag, New York.

[34] - Kowalski, R. A. (1972). *The Predicate Calculus as a Programming Language.* Anais do International Symposium and Summer School on Mathematical Foundations of Computer Science, Jablona near Warsow, Poland.

[35] - Kowalski, R. A. (1974). *Predicate Logic as a Programming Language.* Anais do IFIP'74 World Congress, North-Holland Publishing Company, pp. 574-689.

[36] - Laird, J. E.; Newell, A.; Rosenbloom, P. S. (1987). *SOAR: An Architecture for General Intelligence.* Artificial Intelligence, nº 33, pp 1-64.

[37] - Levas, A.; Jayaraman, R.(1989). *WADE: An Object-Oriented Environment for Modeling and Simulation of Workcell Applications.* IEEE Transactions on Robotics and Automation, vol. 5, nº 3, Junho.

[38] - Lippmann, R. P. (1987). *An Introduction to Computing with Neural Nets.* IEEE ASSP Magazine, volume 4, nº 2, abril, pp 4-22.

[39] - Loyolla, W. P. D. C.; Mendes, M. J. (1989). *Arquitetura de um Servidor de Simulação em Redes Industriais da Manufatura.* Submetido ao 1º Simpósio de Automação Integrada, Curitiba, Paraná, Julho de 1990.

[40] - Loyolla, W. P. D. C.; Mendes, M. J. (1989). *Lógica Temporal na Especificação de Simuladores para Sistemas Discretos na Manufatura.* Submetido ao 8º CBA, Belém, Pará, Setembro de 1990.

[41] - Mittal, S.; Chandrasekaran, B.; Sticklen, J. (1984). *Patrec: A Knowledge-Directed Database for a Diagnostic Expert System.* Computer IEEE, setembro, pp 51-58.

[42] - Negoita, C. V. (1985). *Experts Systems and Fuzzy Systems.* The Benjamin/Cummings Publishing Company, Inc, Menlo Park, California.

[43] - Nilsson, N. L. (1982). *Principles of Artificial Intelligence.* Tioga P. Company, Palo Alto, California.

[44] - Oshuga, S. (1985). *Conceptual Design of CAD Systems Involving Knowledge Bases.* J. S. Gero (Editor), Knowledge Engineering in Computer Aided Design, North-Holland, pp 29-56.

[45] - Oxman,R. ; Gero, J. S. (1987). *Using Expert Systems for Design Diagnosis and Design Synthesis.* Expert Systems, vol 4, nº 1, fevereiro, pp 4 - 15.

[46] - Rich,Elaine (1983). *Artificial Intelligence.* McGraw Hill, Japan.

[47] - Rimvall, M. (1989). *On the Use of ADA in Computer Aided Engineering.* IEEE 1989 National Aerospace and Electronics Conference, NAECON, vol 4, maio.

[48] - Signoretti, A.; Rezende, M.; Gomide, F.; Bingulac, S. (1989). *Implementation of Knowledge Based Environment for Computer Aided Control Engineering.* IEEE 1989 National Aerospace and Electronics Conference, NAECON, voll1, maio.

[49] - Signoretti, A. (1990). *SMAC - Sistema para Manipulação e Armazenagem de Conhecimento : Programas.* Relatório técnico RT-DCA-2/90, DCA-FEE-UNICAMP.

[50] - Signoretti, A.; Gomide, F.; Bingulac, S. (1990). *Um Ambiente Baseado em Conhecimento para Projeto de Sistemas de Controle.* Submetido ao 8º CBA, Belém, Pará, setembro de 1990.

[51] - Silva, M. A.; Gomide, F.; Amaral, W. (1988) *An Expert System for Tuning of Industrial PID Controllers.* 2º Congresso Brasileiro Automático, S. José dos Campos.

[52] - Suchman, L. (1987). *Plans and Situated Actions.* The Problem Of Human-Machine Communication, Cambridge University Press, New York.

[53] - Takahashi, T. (1989). *O pradigma de Objetos - Introdução e Tendências.* IX Congresso da SBC, VIII Jornada de Atualização em Informática, Uberlândia, Minas Gerais.

[54] - Takenouchi, H. ; Iwashita, Y. (1987). *An Integrated Knowledge Representation Scheme for Expert Systems.* Expert Systems, vol 4, nº 1, fevereiro, pp 38-43.

[55] - Talukdar, S. N.; Cardozo, E.; et alii (1986). *A System for Distributed Problem Solving.* Coupling Symbolic and Numerical Computing in Expert Systems. Elservier Science Publishers B. V., North-Holland.

[56] - Taylor, J. H.; Frederick, D. K. (1984). *An Expert System Architecture for Computer-Aided Control Engineering.* Proceedings of IEEE, vol 72, nº 12, dezembro.

[57] - Taylor, J. H.; Mckeehen, P. D. (1989). *A Computer-Aided Control Engineering Environment for Multi-Disciplinary Experimental Design (MEAD).* IEEE 1989 National Aerospace and Electronics Conf. - NAECON, vol 4, maio.

[58] - Waterman, D. A. (1986) - *A Guide to Expert Systems.* Addison-Wesley

[59] - Wiston, P. H. (1984). *Artificial Intelligence.* 2nd Edition, Addison-Wesley Publishing Co., USA.