GUSTAVO HENRIQUE MENDES MATIAS

CONSTRUINDO MODELOS INTELIGENTES UTILIZANDO A ARQUITETURA COGNITIVA ACT-R

Assis
2009

CONSTRUINDO MODELOS INTELIGENTES UTILIZANDO A ARQUITETURA COGNITIVA ACT-R

GUSTAVO HENRIQUE MENDES MATIAS

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, analisado pela seguinte comissão examinadora:

Orientador: Luiz Carlos Begosso

Analisador: Rita de Cássia Cassiano Lopes

Assis
2009

GUSTAVO HENRIQUE MENDES MATIAS

CONSTRUINDO MODELOS INTELIGENTES UTILIZANDO A ARQUITETURA COGNITIVA ACT-R

Trabalho de Conclusão de Curso apresentado ao
Instituto Municipal de Ensino Superior de Assis,
como requisito do Curso de Graduação, analisado
pela seguinte comissão examinadora:

Orientador: Luiz Carlos Begosso

Área de Concentração: Inteligência Artificial

Assis
2009

DEDICATÓRIA

Dedico este trabalho aos meus pais, que me deram toda ajuda necessária no decorrer dos estudos, ao meu orientador, Luiz Carlos Begosso, que me deu ideias, conselhos e ânimo para aceitar o desafio que foi a realização deste trabalho, e a todos os meus amigos e amigas, que me deram apoio moral e torceram por mim nesta batalha.

RESUMO

Este trabalho apresenta os conceitos da Inteligência Artificial e aprofunda-se na arquitetura cognitiva ACT-R (*Adaptive Control of Thought – Rational*). O objetivo é aplicar a teoria ACT-R na construção de modelos inteligentes que simulam a cognição humana durante a realização de uma certa tarefa.

Palavras-chave: Inteligência artificial. ACT-R. Arquitetura cognitiva. Modelos inteligentes.

ABSTRACT

This paper presents the concepts of Artificial Intelligence and deepens the cognitive architecture ACT-R (Adaptive Control of Thought - Rational). The goal is to apply the theory ACT-R in the construction of intelligent models that simulate human cognition during the performance of a certain task.

Keywords: Artificial intelligence. ACT-R. Cognitive architecture. Intelligent models.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

Este trabalho trata de uma potencial área da ciência da computação, a IA (Inteligência Artificial), especificando-se na arquitetura cognitiva ACT-R (*Adaptive Control of Thought* – Rational, J. R. Anderson & Lebiere, 1998), e mostra a construção de dois modelos inteligentes do comportamento humano.

O ACT-R busca entender e representar computacionalmente como a mente humana funciona, atingindo interesses nas áreas de pesquisas em IA, interação homem-computador, psicologia cognitiva, educação e neurociência. Para a construção de modelos inteligentes que simulam determinadas tarefas da cognição humana, será utilizada sua ferramenta de implementação, o ACT-R 6.0.

1.1. OBJETIVOS

Com este trabalho pretende-se aplicar a teoria cognitiva ACT-R e demonstrar a criação de modelos inteligentes que simulem a cognição humana na execução de uma tarefa específica, expandindo a pesquisa para além da graduação. Serão implementados conceitos de como a cognição humana funciona e, a partir disto, criar programas capazes de desempenharem tarefas que são de natureza humana, como por exemplo realizar um cálculo e pressionar uma tecla correspondente a letra visualizada na tela.

Os modelos a serem implementados serão detalhados no Capítulo 4 e terão base nos estudos feitos sobre a teoria ACT-R, servindo de introdução e primeira experiência com modelos inteligentes.

1.2. JUSTIFICATIVAS E MOTIVAÇÃO

IA é a área da Ciência da Computação que mais fascina o autor deste trabalho. Em um futuro não muito distante, quando robôs estiverem se comportando como seres

humanos, será uma honra participar do grupo de cientistas que fizeram isso acontecer. Buscar-se-á ajudar os robôs agirem como humanos, aprenderem e até mesmo, se assim possível, criarem, terem criatividade como os humanos, fazendo do mundo um lugar como o cinema de ficção científica sempre sonhou, cheio de robôs convivendo e trabalhando para os humanos, dando-nos conforto e facilidade em tudo que fizermos no dia-a-dia.

Visões de um futuro talvez distante, como esta, são essenciais para a formação de pensamentos que incentivam o trabalho que deve ser iniciado desde o mais baixo nível, ou seja, o que lidamos hoje em dia, criando programas simples simuladores de comportamento humano, mas que um dia poderão ser aperfeiçoados e tornarem-se redes neurais artificiais capazes de desenvolverem uma personalidade própria, podendo surgir assim uma máquina realmente inteligente, com vontade própria.

No Brasil, sendo que as pesquisas na área de IA não sejam destaque devido à falta de investimento, há poucos lugares em que se possa trabalhar com este tipo de desenvolvimento. Entretanto, há poucos profissionais para a área, mas a divulgação desta cultura pode fazer com que a mesma tenha chances de crescer se resultados favoráveis forem alcançados com o trabalho destes profissionais.

Como já foi citado, cada vez mais a IA é aplicada nos mais diversos programas de computadores, fazendo com que estes tenham um melhor desempenho e facilidade de utilização. Isto indica que, no futuro, a IA será cada vez mais importante no desenvolvimento computacional, pois desde sua idealização, em 1956, ela vem crescendo e mostrando que é importante e eficiente. Em um futuro provável, onde as máquinas inteligentes estarão por todos os lares, programas comuns ainda existirão, mas os mais modernos aparelhos eletrônicos terão funções e comportamento inteligente para que o usuário tenha mais conforto, facilidade e prazer em utilizá-lo. Isto faz com que pesquisas nesta área sejam de vital importância para o avanço tecnológico, afinal tudo acaba, e um dia o tempo dos computadores “burros” também acabará.

1.3. ESTRUTURA DO TRABALHO

O capítulo 1, esta introdução, apresenta uma breve visão sobre Inteligência Artificial no mundo, os objetivos deste trabalho, as justificativas e a motivação para a pesquisa.

O capítulo 2 trata sobre o tema do trabalho, a Inteligência Artificial, esclarecendo suas definições, utilizações, abordagens, modelos, diferenças entre inteligência e conhecimento, métodos de busca existentes e um pouco sobre representação de conhecimento.

O capítulo 3 explica com detalhes a arquitetura escolhida para ser explorada, o ACT-R, apresentando seus conceitos e estrutura, necessários para um aprendizado inicial e atingir o objetivo deste trabalho.

O capítulo 4 apresenta a ferramenta utilizada para o desenvolvimento dos modelos, o projeto dos modelos inteligentes criados no trabalho, como eles foram feitos e como funcionam.

No capítulo 5 encontram-se descritas as conclusões e as propostas para futuros projetos com base no resultado deste trabalho.

Finalmente, na última parte deste trabalho, estão disponíveis as referências bibliográficas utilizadas contexto do trabalho desenvolvido.

2. INTELIGÊNCIA ARTIFICIAL

Ao longo da história vários filósofos e cientistas se dedicaram à análise de vários aspectos constitutivos da inteligência humana. E, embora o estudo sobre a inteligência tenha se iniciado dentro do campo de estudo da filosofia, o mesmo extrapolou o âmbito filosófico e a inteligência passou a ser estudada de forma científica por outros campos do saber humano, tais como engenharia, psicologia, computação, entre outros, visando aspectos práticos e comerciais. (GANASCIA, 1993).

Da mesma maneira que ocorreu com outras ciências, que antes pertenciam ao campo de estudo da filosofia e depois se tornaram ciências independentes ou ramo de outras ciências, aconteceu com o estudo da inteligência, que hoje é alvo de estudo da ciência conhecida como Inteligência Artificial. A IA tem se destacado na busca por compreender a inteligência e englobar diversos campos do conhecimento com o objetivo prático de simulá-la. (GANASCIA, 1993).

2.1. DEFINIÇÕES

A palavra “inteligência” se originou do latim *inter* e *legere*, que significam, consecutivamente, “entre” e “escolher”. Inteligência é aquilo que permite ao ser humano escolher entre uma coisa ou outra; é a habilidade de realizar uma tarefa de maneira eficiente. Já a palavra “artificial” vem do latim *artificiale*, que significa algo não natural, ou seja, produzido pelo homem. Logo, pode-se definir claramente “inteligência artificial” como um tipo de inteligência produzida pelo homem para dotar os computadores de habilidades que simulem a inteligência do homem. (FERNANDES, 2005).

Segundo Rich & Knight (1994),

> Inteligência Artificial é a área da ciência da computação orientada ao entendimento, construção e validação de sistemas inteligentes, isto é, que exibem de alguma forma, características associadas ao que chamamos inteligência.

Em outras palavras, Feigenbaum (1992) cita que

> IA é a parte da ciência da computação voltada para o desenvolvimento de sistemas de computadores inteligentes, isto é, sistemas que exibem características, as quais associam-se com a inteligência no comportamento humano – por exemplo, compreensão da linguagem, aprendizado, raciocínio, resolução de problemas, etc.

A IA busca entender a mente humana e imitar seu comportamento, levantando questões tais como: Como ocorre o pensar? Como o homem extrai o conhecimento do mundo? Como a memória, os sentidos e a linguagem ajudam no desenvolvimento da inteligência? Como surgem as ideias? Como a mente processa informações e tira conclusões decidindo por um coisa ao invés de outra? Essas são algumas perguntas que a IA precisa responder para simular o raciocínio humano e implementar aspectos da inteligência. (BOOSE, 1994).

A inteligência humana está aliada à sua capacidade de interagir com o meio através de habilidades cognitivas (sentidos) e conotativas (ação), ou seja, se movimentar, reconhecer sons (fala) e imagens, se expressar, etc. Existe um esforço, principalmente no campo da robótica, no sentido de implementar essas habilidades nas máquinas inteligentes, de modo a propiciar uma maior interação com o meio e desenvolver padrões de inteligência envolvidos na aquisição do conhecimento, reconhecimento, aprendizado, etc. (FERNANDES, 2005).

Cada vez mais as máquinas possuem softwares capazes de simular as ações dos usuários, prevenir erros e automatizar tarefas de maneiras eficientes. Estes e outros avanços relacionados aos programas de computadores existem graças às pesquisas em IA.

2.2. UTILIZAÇÕES

Atualmente, os desenvolvedores de IA desenvolvem técnicas que permitem a implementação de sistemas de comportamento inteligente específico, no qual a inteligência exibida possa ser resultado de um compromisso entre o que é

necessário e o que é possível obter sem que o mesmo deixe de ser eficiente. Estes sistemas recebem o nome “sistemas especialistas”.

Ao contrário dos sistemas tradicionais, os sistemas especialistas têm algumas facilidades que aumentam sua flexibilidade e eficiência, tais como a possibilidade para construção de regras e a tomada lógica de decisões sob imprecisão ou na ausência de informações. Em um programa tradicional, o método de busca é baseado no conhecimento anteriormente codificado no sistema; quando surge novo conhecimento, é necessário reescrever o código. Já os sistemas especialistas podem recuperar novos fatos e regras e utilizá-los sem ter que modificar a estratégia da busca. (SABBATINI, 1993).

A seguir encontram-se alguns exemplos de sistemas especialistas que já são possíveis graças à atual maturidade da Inteligência Artificial:

- Um programa suficientemente inteligente capaz de aprender regras de como as pessoas fazem compras em um supermercado, a partir de informações existentes sobre as diversas compras realizadas anteriormente. Um sistema como este ajudaria o usuário a ir diretamente nos locais que ele precisa, evitando que ele esqueça itens essenciais e não se preocupe com as mesmas coisas todas as vezes que for ao supermercado;
- Um sistema que reconheça e entenda padrões de comportamento do mercado de ações e, com base nesses padrões, aconselhe os corretores a sobre o que fazer em cada momento;
- Um programa capaz de auxiliar no diagnóstico médico;
- Um programa para fazer diagnósticos em máquinas complexas, evitando o trabalhoso e demorado trabalho humano na verificação de cada parte ou peça da máquina em busca do problema;
- Um sistema que joga xadrez tão bem quanto o melhor jogador humano da atualidade, como ocorreu em 1997 na disputa do campeão mundial Garry Kasparov contra o supercomputador Deep Blue, criado pela IBM;

- Um programa capaz de reconhecer e interpretar frases em uma linguagem natural de um campo científico qualquer ou até mesmo de manter um diálogo simples sobre assuntos específicos com o usuário;
- Um sistema que aprenda as preferências pessoais do usuário, capacitado a aconselhá-lo sobre qual roupa vestir em certa ocasião, qual livro ler, qual programa de televisão ou filme assistir, para onde sair devido a variáveis como clima ou hora do dia, etc.

Vale lembrar que programas como estes citados acima são muito bons em fazer aquilo que foram programados. Por esse motivo é justo terem a palavra “especialista” no nome, pois especialista é uma pessoa (ou sistema, programa de computador) que se ocupa exclusivamente de um ramo particular de uma ciência, de uma arte, etc.

Sistemas especialistas são criados para uma única coisa, sem possibilidade disso mudar. Ele poderá aprender sempre, mas somente se tais informações forem relacionadas ao seu campo de atividade. Logo, um programa capaz de fazer um robô se movimentar de maneira tão natural quanto uma modelo humana nas passarelas não pode sequer responder a uma equação simples que seja submetida a ele externamente.

2.3. ABORDAGENS

Com base nos diversos campos de estudo, Simons (1988) apontou duas abordagens para Inteligência Artificial: a abordagem cognitiva e a abordagem conexionista.

2.3.1. Abordagem cognitiva

Também denominada de “descendente” ou “simbolista”, dá ênfase aos processos cognitivos, ou seja, a forma como o ser humano raciocina. Objetiva encontrar uma explicação para comportamentos inteligentes baseados em aspectos psicológicos e processos algorítmicos. Os pioneiros desta corrente foram McCarthy, Marvin Minsky, Newell e Simon (apud FERNANDES, 2005).

2.3.2. Abordagem conexionista (conexões neurais)

Também denominada de “biológica” ou “ascendente”, dá ênfase no modelo de funcionamento do cérebro, dos neurônios e das conexões neurais. Os pioneiros desta corrente foram McCulloch, Pitts, Heldo, Rosenblatt e Widrow. Em 1943 surgiu a representação e formalização matemática dos neurônios artificiais. A corrente conexionista sofreu grande impacto quando os cientistas Marvin Minsky e Seymour Papert publicaram, em 1969, o livro *Perceptrons*, no qual criticavam e sustentavam que os modelos das redes neurais não tinham sustentação matemática suficiente que lhes fosse possível atribuir alguma confiabilidade. Apesar de as pesquisas nesta área não terem parado, foi apenas na década de 1980 que John Hopfield conseguiu recuperar a credibilidade da utilização de redes neurais (apud FERNANDES, 2005).

2.4. MODELOS

Dentre os modelos de IA existentes, os principais são: Algoritmos Genéticos, Programação Evolutiva, Lógica Fuzzy, Sistemas Baseados em Conhecimento, Raciocínio Baseado em Casos e Redes Neurais.

A seguir, encontra-se uma breve descrição de cada um destes modelos citados acima.

2.4.1. Algoritmo Genético

É um modelo para aprendizado de máquina, inspirado no livro *Origem das Espécies*. Através da seleção natural, escrito pelo naturalista inglês Charles Darwin (1809 – 1882), criador da teoria evolucionista, segundo a qual somente os mais aptos sobrevivem. É um método utilizado pelos algoritmos Evolutivos, que inclui o estudo dos algoritmos genéticos, estratégia de evolução, programação evolutiva e sistemas classificatórios. Os algoritmos genéticos foram criados por Holland (1975) e objetivam emular operadores genéticos (específicos, como cruzamento, mutação e reprodução) da mesma forma como são observados na natureza. Isto é feito criando-se dentro da máquina uma população de indivíduos representados por cromossomos. Os indivíduos passam por um processo simulado de evolução, seleção e reprodução, gerando uma nova população (FERNANDES, 2005).

A Figura 1 demonstra, em forma de fluxograma, a estrutura de um algoritmo genético.

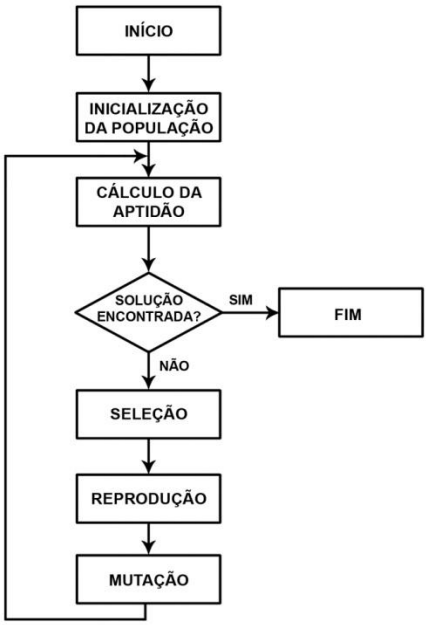

Figura 1. Estrutura básica de um Algoritmo Genético simples

Com referência a Figura 1, observa-se que cada iteração do algoritmo genético corresponde à aplicação de um conjunto de quatro operações básicas: cálculo de aptidão, seleção, cruzamento e mutação. Ao fim destas operações cria-se uma nova população, chamada de geração que, espera-se, representa uma melhor aproximação da solução do problema de otimização que a população anterior. A população inicial é gerada atribuindo-se aleatoriamente valores aos genes de cada cromossomo. A aptidão bruta de um indivíduo da população é medida por uma função de erro, também chamada de função objetivo do problema de otimização. A aptidão bruta é em seguida normalizada (aptidão normalizada), para permitir um melhor controle do processo de seleção. Como critérios de parada do algoritmo em geral são usados a aptidão do melhor indivíduo em conjunto com a limitação do número de gerações. Outros critérios podem envolver, por exemplo, um erro abaixo de um valor especificado pelo projetista para um determinado parâmetro do problema (MITCHELL, 1996).

2.4.2. Programação Evolutiva

Campo da IA concebido por Fogel (1960), assemelha-se aos algoritmos genéticos, sendo que dá maior ênfase na relação comportamental entre os parentes e seus descendentes. As soluções para os problemas são obtidas por meio de tentativas e transmitidas para a nossa população (simulada em programas) (apud FERNANDES, 2005).

2.4.3. Lógica Fuzzy

Também denominada “lógica difusa” ou “lógica nebulosa”. Foi estruturada por Lofti Zadeh, na Universidade da Califórnia, no ano de 1965. É uma metodologia que serve para representar, manipular e modelar informações incertas (FERNANDES, 2005). É uma extensão da Lógica Booleana que admite infinitos valores lógicos intermediários entre o “falso” e o “verdadeiro”, bem como o valor médio “talvez” (0,5).

As implementações da Lógica Fuzzy permitem que estados indeterminados possam ser tratados por dispositivos de controle. Desse modo, é possível avaliar conceitos não quantificáveis. Casos práticos: avaliar a temperatura (*quente*, *morno*, *frio*, etc.), o sentimento de felicidade (*radiante*, *feliz*, *apático*, *triste*, etc.), a veracidade de um argumento (*corretíssimo*, *correto*, *contra-argumentativo*, *incoerente*, *falso*, *totalmente errôneo*, etc.) e assim por diante.

A Tabela 1 exemplifica os valores dos dados em uma resolução com Lógica Fuzzy, levando em conta variações de idades de uma pessoa em relação à fase etária que a pessoa se encontra, considerada socialmente.

Idade da pessoa	Jovem?	Adulto?	Idoso?
10	1	0	0
20	0,8	0,8	0,1
30	0,5	1	0,2
40	0,2	1	0,4
50	0,1	1	0,6
60	0	1	0,8
70	0	1	1

Tabela 1. Faixa etária de uma pessoa em representação fuzzy

De acordo com a tabela acima, pode-se considerar que uma pessoa de 10 anos é jovem, somente; já uma de 20 anos é tão jovem quanto adulto, possuindo um valor "quase verdadeiro" em ambos; finalmente, uma pessoa de 50 anos é um adulto pleno, mas também está se tornando um idoso, pois possui seu valor na coluna "idoso" se aproxima de "verdadeiro" (1).

2.4.4. Sistemas Baseados em Regras

Segundo Fernandes (2005), "são sistemas que implementam comportamentos inteligentes de especialistas humanos". Em outras palavras, são basicamente os sistemas especialistas, apresentados na seção 2.2 deste trabalho.

2.4.5. Raciocínio Baseado em Casos

É o campo de estudo da IA que utiliza uma grande biblioteca de casos para consulta e resolução de problemas. Os problemas atuais são resolvidos através da recuperação e consulta de casos já solucionados e da conseqüente adaptação das soluções encontradas (FERNANDES, 2005). Este modelo possui um certo poder de aprendizado através de análise, pois tem capacidade de prever resultados e tomadas de decisões baseado em resultados anteriores.

Segundo Lee (1998), as aplicações desenvolvidas em sistemas de RBC (Raciocínio Baseado em Casos) são um forte indicativo de onde pode chegar essa técnica. São três os tópicos importantes:

- Quais os tipos de aplicações adequados;
- Quais as ferramentas disponíveis para implementação;
- Quais as aplicações importantes que podem demonstrar o potencial dos sistemas de Raciocínio Baseado em Casos.

Existem algumas aplicações que, apesar de nenhuma delas se aproximarem dos objetivos propostos pelo sistema, exemplificam a utilização do RBC:

- **Abby**, um conselheiro romântico;
- **Archie**, para auxilio em projetos arquitetônicos;
- **Celia**, que produz diagnósticos de automóveis;
- **Chef**, que auxilia no planejamento de receitas, na culinária;
- **Grebe**, capaz de identificar e explicar conseqüências legais de uma situação;
- **Plexus**, para o auxílio no planejamento de tarefas cotidianas;
- **Squad**, que faz o controle de qualidade de softwares.

2.4.6. Redes Neurais

Possui várias denominações, dentre elas: Redes Neuronais, Modelo Conexionista, Neurocomputação, Modelo de Processamento Paralelo Distribuído, Sistemas Neuromórficos e Computadores Biológicos. São consideradas uma classe de modelagem de prognóstico que trabalha por ajuste repetido de parâmetro. Estruturalmente, uma rede neural consiste em um número de elementos interconectados (chamados neurônios) organizados em camadas que aprendem pela modificação da conexão firmemente conectando as camadas (FERNANDES, 2005).

Uma rede neural artificial é composta por várias unidades de processamento, cujo funcionamento é bastante simples. Essas unidades, geralmente são conectadas por canais de comunicação que estão associados a determinado peso. As unidades fazem operações apenas sobre seus dados locais, que são entradas recebidas pelas suas conexões. O comportamento inteligente de uma RNA vem das interações entre sãs unidades de processamento da rede (GURNEY, 1997).

A operação de uma unidade de processamento, proposta por McCulloch e Pitts em 1943, mostrado na Figura 2, pode ser resumida da seguinte maneira:

- Sinais são apresentados à entrada;
- Cada sinal é multiplicado por um número, ou peso, que indica a sua influência na saída da unidade;
- É feita a soma ponderada dos sinais que produz um nível de atividade;
- Se este nível de atividade exceder um certo limite, a unidade produz uma determinada resposta de saída.

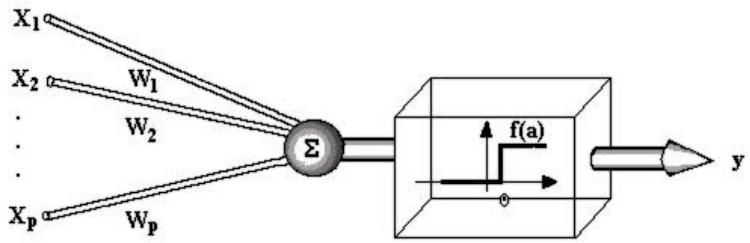

Figura 2. Esquema de unidade de McCulloch e Pitts

De acordo com Lippmann (1987), um neurônio soma todos os pesos das entradas e passa o resultado para uma função de ativação não-linear. Há três tipos de não-lineares: limitadores elevados, elementos limiares lógicos e não-lineares sigmóides.

A maioria dos modelos de redes neurais possui alguma regra de treinamento, onde os pesos de suas conexões são ajustados de acordo com os padrões apresentados. Em outras palavras, elas aprendem através de exemplos (FERNANDES, 2005).

As arquiteturas de RNA são organizadas em várias camadas, possuindo em cada camada unidades que podem estar conectadas às unidades da camada posterior, assim como mostra na Figura 3.

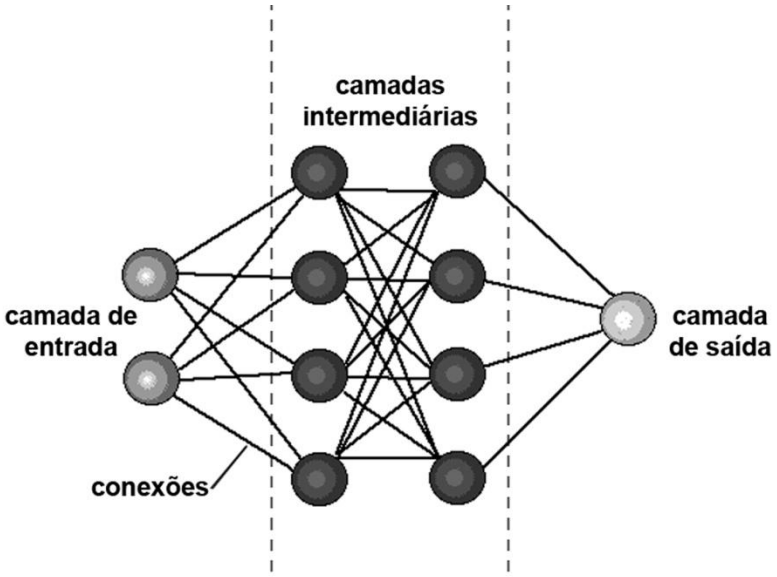

Figura 3. Camadas de uma RNA

Na camada de entrada os padrões são apresentados à rede; nas camadas intermediárias, também consideradas como extratoras de características, é feita a maior parte do processamento, através das conexões ponderadas; por último, na camada de saída o resultado final é concluído e apresentado.

2.5. A INTELIGÊNCIA E O CONHECIMENTO

Para que se possa compreender uma ação inteligente, é necessário que sejam analisados todos os aspectos relativos à aquisição e desenvolvimento da inteligência. Dentro desse contexto, o conhecimento é o que faz com que sejam possíveis o encadeamento e o desenvolvimento da inteligência (RICH, 1993).

Com base nesse raciocínio, encontra-se a necessidade de adquirir conhecimento e incorporá-lo a sistemas computacionais de IA. Porém, há certas características do conhecimento que devem ser levadas em conta:

- Volumoso, ou seja, possui diversos aspectos, características e detalhes. Quando mais se que esmiuçá-lo, mais e mais conhecimento a pessoa terá (FERNANDES, 2005);
- De difícil caracterização, isto é, muitas vezes tem-se apenas o conhecimento, mas não se sabe como foi adquirido, ou então, não se sabe explicá-lo. De fato, muitas vezes não se tem consciência do conhecimento que se possui (FERNANDES, 2005);
- Está em constante mudança, sempre crescendo, modificando e aperfeiçoando-se;
- Difere de simples dados por organizar-se de uma maneira que corresponde ao modo como será usado (RICH, 1993). Dados fazem parte do conhecimento, sendo compostos de forma lógica de modo a permitir sua interpretação (FERNANDES, 2005). Em outras palavras, conhecimento é formado por dados agrupados de maneira eficientemente organizada.

Portanto, onde é que ficamos na nossa tentativa de definir as técnicas de IA? Segundo Rich (1993), uma técnica de IA é um método que explora o conhecimento, que deve ser representado de tal forma que:

- O conhecimento capture generalizações. Em outras palavras, não é que não seja necessário representar separadamente cada situação individual. Em vez disso, as situações que compartilham propriedades importantes são agrupadas. Se o conhecimento não tem essa propriedade, haverá grande necessidade de memória e atualização. É por isso que normalmente chamamos algo sem essa propriedade de dados ao invés de conhecimento;
- Ele precisa ser compreendido pelas pessoas que o fornecem. Embora para muitos programas o grosso dos dados possa ser adquirido automaticamente (por exemplo, através das leituras fornecidas por instrumentos), em muitos domínios da IA, grande parte do conhecimento que um programa possui precisa basicamente ser fornecido pelas pessoas em termos que elas compreendam.

- Ele pode ser facilmente modificado para corrigir erros e refletir mudanças do mundo e da nossa visão do mundo;
- Ele pode ser usado em inúmeras situações, mesmo que não seja totalmente preciso nem esteja completo;
- Ele pode ser usado para ajudar a superar seu próprio volume, auxiliando a limitar as várias possibilidades que em geral tem de ser consideradas.

As técnicas de IA precisam ser projetadas levando-se em conta essas restrições impostas pelos problemas de IA; entretanto, existe um certo grau de independência entre os problemas e as técnicas de solução dos mesmos. É possível solucionar problemas de IA sem que se usem técnicas de IA (embora essas soluções provavelmente não sejam muito boas). E é possível aplicar técnicas de IA à solução de problemas não relacionados à Inteligência Artificial. Este é um bom caminho a seguir no caso de problemas que possuam muitas das mesmas características dos problemas de IA (RICH, 1993).

2.6. MÉTODOS DE BUSCA

Dentre as técnicas de Inteligência Artificial, encontra-se a Busca Heurística, que proporciona um meio de solucionar problemas complexos, para os quais não há disponível uma abordagem mais direta nem uma estrutura na qual qualquer técnica direta disponível possa ser inserida (FERNANDES, 2005).

Ainda de acordo com Fernandes (2005), a palavra “heurística” vem do grego *heuriskein*, que significa descobrir. A heurística é um procedimento para resolver problemas através de um enfoque, em geral racional, no qual a estrutura do problema passa a ser interpretada e explorada inteligentemente para obter uma solução razoável. Para os estudiosos da Inteligência Artificial, as heurísticas são critérios, métodos ou princípios para decidir, entre vários cursos de ação alternativos, aquele que parecer mais efetivo para atingir algum objetivo.

Um bom exemplo problema resolvido com a utilização desta técnica é o de traço de menor rota entre dois locais, exemplificado na Figura 4.

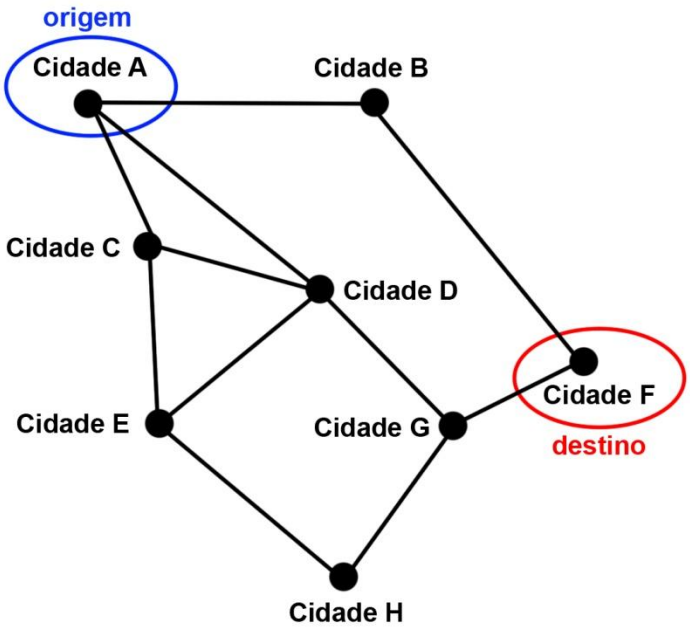

Figura 4. Exemplo de problema solucionado pela busca heurística

No exemplo acima, considere que um viajante deseja sair de sua atual localização (Cidade A) e chegar até seu destino (Cidade F). Um programa feito com técnicas de busca heurística receberia esses dados como entrada e faria uma árvore com todas as cidades existentes no cenário do problema; assim, ele percorreria esta árvore da raiz (Cidade A) até suas folhas, buscando pela específica “Cidade F”, podendo determinar então, considerando cada cidade e com qual cada uma está diretamente conectada e a distância entre as mesmas, qual o caminho mais curto até ela.

Entretanto, há várias técnicas diferentes de busca heurística; as mais básicas são apresentadas a seguir.

2.6.1. Busca em Profundidade

É também denominada de primeiro em profundidade (ou *depth first*) e explora o caminho para o objetivo, dando preferência aos nós que estão mais distantes da raiz da árvore de busca. É aplicável quando as soluções são total e igualmente desejadas ou quando, em uma varredura prévia, direções incorretas são detectadas (FERNANDES, 2005).

A busca por profundidade utiliza a estratégia de expandir o nó mais profundo no caminho atual da árvore, até quando o nó mais profundo não tem mais sucessores (filhos), conforme é apresentada pela Figura 5. À medida que esses nós são expandidos, eles são retirados da fila, retornando a busca ao nó seguinte mais raso que ainda tem filhos que não foram visitados (explorados). Quanto à memória, ela só precisa armazenar um único caminho da raiz até o nó folha, juntamente com os nós irmãos não expandidos de cada nó do caminho (VIEIRA, 2006).

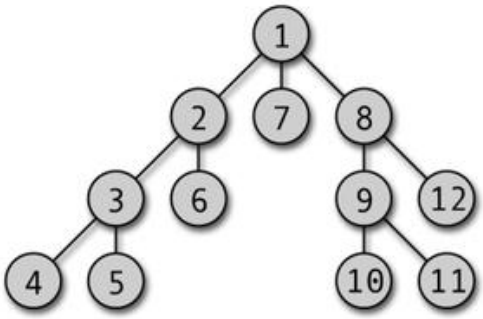

Figura 5. Ordem que os nós são explorados na busca em profundidade (http://pt.wikipedia.org/wiki/Busca_em_profundidade)

Rich (1993) apresentou da seguinte maneira o algoritmo de busca em profundidade:

1. Se o estado inicial é um estado de meta, saia e retorne sucesso;
2. Caso contrário, faça o seguinte até a sinalização de sucesso ou fracasso:

a. Gere um sucessor, *E*, do estado inicial. Se não houver mais sucessores, sinalize fracasso;
b. Chame Busca em Profundidade (este algoritmo) com *E* como estado inicial;
c. Se for retornado sucesso, sinalize sucesso. Caso contrário, continue nesse laço (loop).

A busca em profundidade requer menos memória, já que apenas os nós do caminho corrente são armazenados. Isto contrasta com a busca em amplitude (veja a seguir) em que toda a árvore gerada até o momento precisa ser armazenada. Por acaso, esta técnica pode encontrar a solução sem examinar grande parte do espaço de busca, do contrário que faz a busca por amplitude, na qual todas as partes da árvore precisam se examinadas no nível *n* antes de serem examinados os nós do nível *n* + 1. Isto é particularmente significativo se existirem muitas soluções aceitáveis. A busca em profundidade pode parar quando uma delas for encontrada (RICH, 1993).

2.6.2. Busca em Amplitude

De acordo com Fernandes (2005), esta técnica é o oposto da busca em profundidade e trabalha sobre um critério de FIFO (First In First Out). É também denominada de busca em largura, busca em nível ou *breadth first*. Todos os nós de certo nível da árvore são examinados antes do nível abaixo, caso exista uma solução, e se o grafo é finito, a solução será encontrada. Existem alguns inconvenientes na utilização desta técnica: requer muita memória (pois o número de nós é exponencial em relação à profundidade); exige um esforço computacional relativamente grande se o comprimento dos caminhos não for pequeno; o esforço com os operadores de pouca importância assume as mesmas proporções dos operadores mais importantes. Veja na Figura 6 a ordem que esta técnica percorre a árvore.

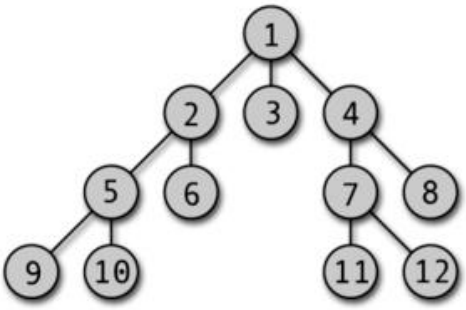

Figura 6. Ordem que os nós são explorados na busca em amplitude (http://pt.wikipedia.org/wiki/Busca_em_largura)

A seguir, o algoritmo da busca em amplitude, segundo Rich (1993):

1. Crie uma variável chamada *Lista-de-Nós* e ajuste-a para o estado inicial;
2. Até ser encontrado um estado meta ou *Lista-de-Nós* ficar vazia, faça o seguinte:
    a. Remova o primeiro elemento de *Lista-de-Nós* e chame-o de *E.* se *Lista-de-Nós* estiver vazia, saia;
    b. Para cada maneira como cada regra pode ser casada com o estado descrito em *E*, faça o seguinte:
        I. Aplique a regra para gerar um novo estado;
        II. Se o novo estado for um estado meta, saia e retorne este estado;
        III. Caso contrário, acrescente o novo estado ao final de *Lista-de-Nós*.

Apesar de exigir mais memória, a busca em amplitude evita o problema de ciclos em árvores, o que não ocorre com a busca em profundidade. A busca em amplitude não cairá na armadilha de explorar um beco sem saída. Se houver uma solução, então a *breadth first* certamente a encontrará; além disso, se houver várias soluções, então uma solução mínima (um que exija o menor número de etapas) será encontrada. Isto é garantido pelo fato de que os caminhos mais longos só são explorados depois que os mais curtos já tenham sido. Este procedimento contrasta com a busca em

profundidade, que pode encontrar um caminho longo para uma solução em uma parte da árvore, quando existe um caminho mais curto, em alguma outra parte inexplorada da mesma (RICH, 1993).

2.6.3. Gera e Testa

É um procedimento de busca em profundidade, pois as soluções devem estar completas para serem testadas. A sua forma aleatória não garante que alguma solução seja encontrada e também que todas elas sejam geradas. A forma maia adequada de implementar o método é usar a busca em profundidade adicionando *backtracking* (ato de retroceder na busca a um nível anterior, em geral com o objetivo de tentar uma outra alternativa de solução). É aceitável quando se trata da solução de problemas simples ou quando for usado como método auxiliar de outra técnica (FERNANDES, 2005).

Existe um algoritmo clássico do tipo “gera e testa” chamado A-Star (A*), descrito por Rich (1993). O A-Star seleciona o próximo nó a ser aberto (no caso, a próxima situação a ser testada), que lhe garante encontrar em primeiro lugar a melhor solução, podendo então interromper a busca. Para conseguir isso, o algoritmo A-Star trabalha com a ideia de “busca melhor primeiro” e, para determinar qual deverá ser o possível próximo melhor passo para efetuar todo o seu percurso, ele baseia-se na distância percorrida até encontrar o tal nó mais a menor distância possível para encontrar o nó-destino, obtida por meio de uma função heurística.

O que garante o sucesso do algoritmo A-Star é justamente essa análise não somente da qualidade do nó para chegar ao destino, obtida por uma função heurística h, mas também da qualidade do caminho percorrido até chegar àquele nó a partir da origem, conhecida aqui como função g. Denominando a soma de ambas de f = g + h, teremos então que o algoritmo de A-Star funciona da seguinte forma:

1. Inicialmente temos somente o nó origem em uma lista de nós abertos. Calculamos valores para g (igual a 0), h (distância heurística até o nó destino) e f (igual a h). Fechados é inicialmente uma lista vazia;

2. Verifique se abertos está vazia, se sim, retorne erro e encerre o algoritmo. Caso contrário, seleciona-se o nó em abertos com o menor valor de f, remove-o e chama-o de melhor_no. Se este for igual ao nó destino, ou seja, se for solução, retorne a solução (que pode ser somente este nó ou todo o percurso, o que nos obriga a ter um ponteiro para cada "nó-pai" a fim de recuperar qual foi o percurso até chegar a ele) com sucesso e encerra o algoritmo. Caso contrário, precisamos verificar cada sucessor de melhor_no a fim de procurar o caminho. Inicialmente, calcule o g(sucessor) como sendo g(melhor_no) + custo para ir de melhor_no a sucessor;

3. Verifique se sucessor já está em abertos, pois se estiver, precisamos verificar qual o caminho com melhor g, se o antigo ou o novo. Se o antigo caminho (aquele que usava o mesmo sucesso anteriormente) for o melhor (com o g menor), simplesmente ignore este sucessor e vá para o próximo. Caso contrário, faça aquele nó que representa o sucessor apontar para o melhor_no e atualize seu g para o valor calculado no item anterior;

4. Se sucessor não estava em abertos, precisamos ainda verificar se o mesmo não já está em fechados. Se ele estiver, significa que não somente já criamos esse nó, como também já criamos os seus vizinhos/sucessores, em outras palavras, precisamos ajustar não somente ele, mas também os que estão após ele em seu percurso. Para tal, verifique se o antigo nó (aquele que você encontrou em fechados) possui menor g que o atual sucessor. Se sim, não é necessário atualizar, pois o anterior já era um melhor caminho. Caso contrário, comece atualizando o g daquele nó e fazendo ele apontar para melhor_no. Depois disso, recursivamente, atualize o valor de g e f em cada um dos nós sucessores dele, dos nós sucessores de seus sucessores e assim por diante, de forma recursiva;

5. Se o nó sucessor analisado até então não está nem em abertos nem em fechados, coloque-o em abertos e acrescente-o à lista de sucessores de melhor_no. Após fazer isso para todos os possíveis nós sucessores de melhor_no, coloque melhor_no na lista de fechados e retorne ao segundo passo/item (RICH, 1993).

2.6.4. Busca em Feixe

É uma alternativa do método de busca em amplitude. Há uma restrição ao número de nós que serão considerados em cada nível, por exemplo, se este número não ultrapassar três, então, no máximo, será explorada três vezes a profundidade de nós. Isto reduz o esforço de busca e proporciona bons resultados quando houver bons critérios para escolha de nós (FERNANDES, 2005).

Uma representação simples do algoritmo para esta técnica de busca é a seguinte, segundo Simões (2006):

1. Ao invés de manter um único estado na memória, mantém o controle de *k* estados;

2. Começa com *k* estados gerados aleatoriamente. Em cada passo são gerados todos os sucessores de todos os *k* estados;

    a. Se qualquer um destes for um objetivo, o algoritmo terminar;

    b. Caso contrário, ele selecionará os *k* melhores sucessores a partir da lista completa e repetirá a ação.

2.6.5. Subindo o Morro

Também chamada de “subida de encosta”, “subida da montanha” ou *Hill Climbing*, pode ser uma variante da técnica “gera e testa”, acrescentando-se as informações que ajudam a decidir em qual direção prosseguir. É um método que usa a ideia de que o objetivo deve ser atingido com menor número de passos. O suporte é dado pela ideia heurística de que o número de passos para atingir um objetivo é inversamente proporcional ao seu tamanho (FERNANDES, 1993).

Castro & Zuben (2002) descreveram o algoritmo padrão *Hill Climbing* da seguinte maneira:

1. Inicialize (aleatoriamente) o ponto x na região factível do problema;
2. A cada iteração, um novo ponto x' é selecionado aplicando-se uma pequena perturbação no ponto atual, ou seja, selecionando-se um ponto x' que esteja na vizinhança de x, $x' \hat{I} N(x)$;
    a. **Se** este novo ponto apresenta um melhor valor para a função de avaliação, **então** o novo ponto torna-se o ponto atual;
    b. O método é terminado quando nenhuma melhora significativa é alcançada, um número fixo de iterações foi efetuado, ou um objetivo foi atingido.

2.7. REPRESENTAÇÃO DO CONHECIMENTO

Para solucionar os problemas complexos encontrados na Inteligência Artificial, é preciso uma grade quantidade de conhecimento armazenado e também alguns mecanismos para a manipulação desse conhecimento. Uma série de maneiras de representar o conhecimento (fatos) tem sido explorada por programas de IA. Para isso, deve-se considerar o seguinte ponto, que tem a ver com todas as discussões sobre representação, especificamente o fato de estarmos lidando com dois tipos diferentes de entidades (RICH, 1993):

- Fatos, verdades em algum mundo relevante. São as coisas que busca-se representar;
- Representação de fatos em algum formalismo escolhido. São as coisas que realmente consegue-se manipular.

Ainda de acordo com Rich (1993), pode-se pensar em estruturar essas entidades em dois níveis:

- **Nível de conhecimento**, no qual os fatos (inclusive o comportamento e os objetivos atuais de cada agente) são descritos;
- **Nível de símbolo**, no qual as representações dos objetivos no nível de conhecimento são definidas em termos de símbolos que podem ser manipulados por programas.

Newell (1982) apresenta uma exposição detalhada sobre este ponto de vista no contexto dos agentes e seus objetivos e comportamentos.

Fernandes (2005, p. 9) citou que

> Representação do conhecimento pode ser definida como um conjunto de convenções sintáticas e semânticas que tornam possível descrever coisas. A representação do conhecimento para sistemas de IA é uma visão conexionista de que várias unidades interconectadas idênticas são coletivamente responsáveis para representar vários conceitos. Um conceito é representado num senso distribuído e é indicado por um envolvimento em atividades sobre uma coleção de unidades.

São características das formas de representação do conhecimento: i) escopo e granularidade: partes do domínio considerado, detalhadamente; ii) indeterminação e definição das noções primitivas de representação: alternativas de modelagem; iii) modularidade, compreensibilidade: “clusterização” do conhecimento, legibilidade; iv) conhecimento explícito e flexibilidade: toda a informação necessária à solução do problema está na base de conhecimento e não embutida em outro componente (FERNANDES, 2005).

Existem diversos paradigmas de representação do conhecimento que tem emergido destas perspectivas:

- **Conhecimento procedural:** o conhecimento é representado em forma de funções ou procedimentos;
- **Redes:** o conhecimento é representado por um rótulo de grafos direcionados, cujos nós representam conceitos e entidades, enquanto os arcos representam a relação entre entidades e conceitos;
- **Frames:** muito parecidos com a rede semântica, exceto que cada nó representa conceitos e/ou situações. Cada nó tem varias propriedades que podem ser especificadas ou herdadas por padrão;
- **Lógica:** um modo de declaração que representa o conhecimento;
- Árvores de decisão: conceitos são organizados em formas de árvores;
- **Conhecimento estatístico:** uso de fatores de certeza, Redes Bayeseanas, Teoria de Dempster-Shaper, Lógica Fuzzy;
- **Regras:** sistemas de produção para codificar regras de condição / ação;
- **Processamento paralelo distribuído:** utiliza-se de modelos conexionistas;
- **Esquemas híbridos:** qualquer representação do formalismo que emprega a combinação de esquemas de representação do conhecimento;
- **Casos:** usa experiência passada, acumulando casos e tentando descobrir, por analogia, soluções para outros problemas.

3. ACT-R

O ACT-R (*Adaptive Control of Thought – Rational*) é uma teoria geral sobre a cognição humana com ênfase na aquisição de conhecimento. Começou a ser desenvolvida em 1976 e uma de suas primeiras versões, a ACT, foi proposta em 1983. A versão atual, que ficou pronta em 1993, é conhecida como ACT-R. As principais características da teoria ACT que não se alteraram são:

- Distinção entre procedural (como usamos o conhecimento adquirido) e declarativo (tudo aquilo que sabemos);
- Construção do conhecimento: a teoria assume que regras de como usar o conhecimento só podem ser aprendidas aplicando-se o conhecimento declarativo ao contexto da resolução de um determinado problema;
- Reforço: só é possível tornar o conhecimento sólido (tanto procedural como declarativo) com a prática.

Exteriormente, ACT-R aparenta uma linguagem de programação, no entanto, suas construções refletem suposições sobre a cognição humana, baseadas em inúmeros fatos derivados de experimentos psicológicos. Na verdade, é como uma linguagem de programação, um *framework*: para cada diferente tarefa, pesquisadores criam modelos (ou programas) escritos em ACT-R e, apesar de incorporar a visão de cognição da própria teoria ACT-R, adicionam suas próprias hipóteses sobre a tarefa em particular. Estas hipóteses podem ser testadas através de comparação de resultados do modelo com resultados de pessoas fazendo a mesma tarefa. Por "resultados" entende-se as tradicionais medidas da psicologia cognitiva, que são tempo de desempenho, precisão na realização da tarefa e dados neurológicos obtidos na RMF (Ressonância Magnética Funcional). Gattass (2000) apresenta detalhes sobre RMF.

A Figura 7 mostra o esquema das hipóteses em ação com a teoria cognitiva e os modelos ACT-R.

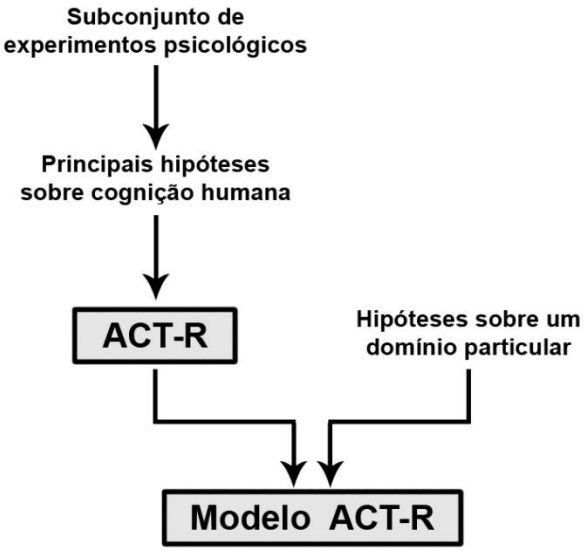

Figura 7. Hipóteses em ACT-R

O ACT-R tem sido usado com sucesso na criação de modelos em domínios como:

- Aprendizado e memória;
- Resolução de problemas e tomada de decisão;
- Linguagem e comunicação;
- Percepção e atenção;
- Desenvolvimento cognitivo.

E além de aplicações na psicologia cognitiva, ACT-R tem sido usado também em outras áreas:

- **Interação humano-computador**, para construir modelos capazes de simular o usuário e fazer a avaliação de diferentes interfaces;
- **Educação**, na construção de sistemas tutores, que encontram as dificuldades que o estudante pode ter no aprendizado e provêm ajuda focalizada;
- **Forças geradas por computador**, para fornecer agentes cognitivos que habitam ambientes de formação;
- **Neurociência e psicologia**, para interpretar dados de RMF.

Neste capítulo serão utilizadas muitas palavras em inglês devido à natureza da arquitetura ACT-R.

3.1. ARQUITETURA

O ACT-R está dividido em três grupos de módulos:

- Os módulos de interface, composto pelos módulos de percepção (responsáveis pela visão, audição e fala) e motor (que controla as atividades motoras do modelo), responsáveis por simularem o sistema perceptivo e motor humano;
- Os módulos de memória e cognição, distinguindo dois tipos de conhecimento de longa duração: o conhecimento declarativo (aquilo que sabemos) e o conhecimento procedural (que diz respeito a como usamos o conhecimento adquirido), simulando o sistema cognitivo humano;
- O módulo de metas, que tem por objetivo sustentar um comportamento cognitivo mantendo o estado interno do que fora planejado a ser executado naquele momento, fazendo o modelo sempre tem algo pelo que buscar até atingir seu objetivo ou resultar em erro.

A coordenação que estrutura estes módulos é realizada por um sistema central de produção. Na Figura 8 é mostrada a organização básica da arquitetura ACT-R, incluindo os módulos de percepção e motor para interface com o ambiente externo, os módulos de memória procedural e declarativa para manipulação do conhecimento, os *buffers* para trabalhar em paralelo com os módulos e o sistema de produção composto pela busca de padrões e execução das produções. Cada componente da mesma será explicado no decorrer deste capítulo.

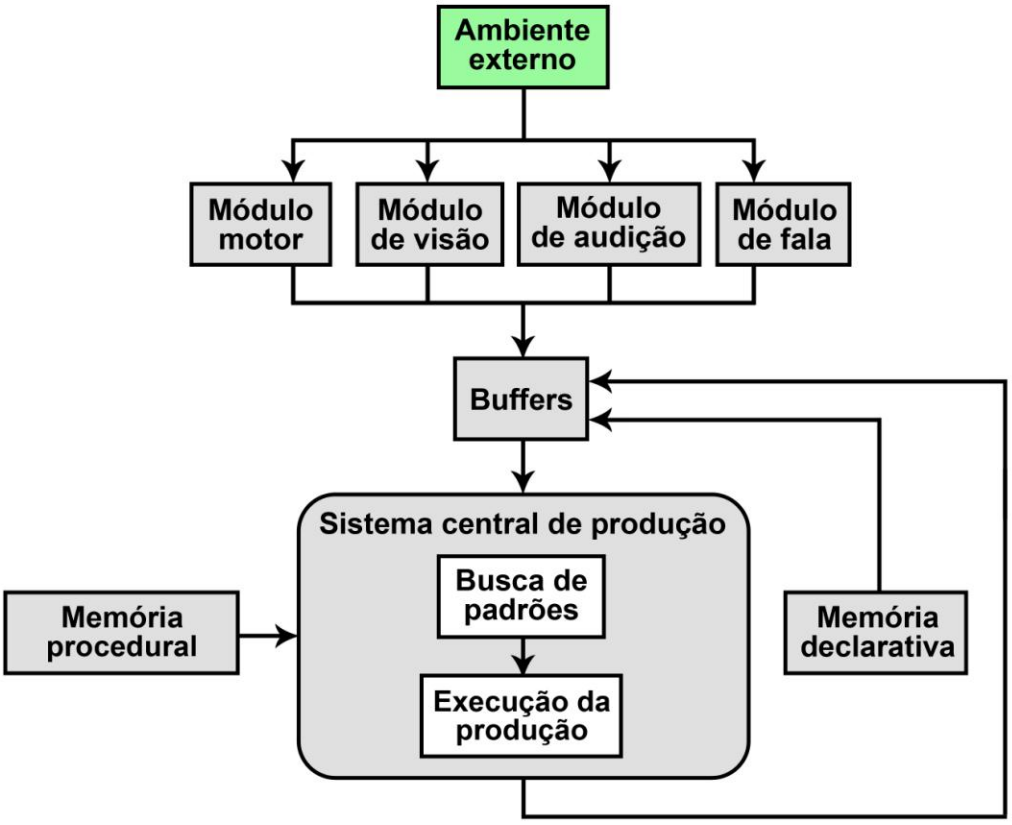

Figura 8. Estrutura básica do ACT-R

3.2. O SISTEMA CENTRAL DE PRODUÇÃO

Como já foi mencionado, há dois tipos de conhecimento na teoria ACT-R: o declarativo e o procedural. Conhecimento declarativo é aquilo que nós sabemos e podemos facilmente descrever aos outros. Conhecimento procedural é aquilo que nós sabemos mas não estamos conscientes disto, como por exemplo descrever as regras da linguagem, nós sabemos falar mas não sabemos como nosso cérebro se comporta para fazer tal coisa.

Na arquitetura ACT-R o conhecimento declarativo é representado em estruturas chamadas *chunks*; o conhecimento procedural é representado por regras

denominadas *produções*. Estes dois elementos são as estruturas básicas de todo modelo criado em ACT-R (BOTHELL, 2007a).

O sistema central de produção é responsável por ficar o tempo todo procurando produções que possam ser executadas e dar continuidade ao andamento do modelo. Caso não encontre nenhuma, o modelo é finalizado imediatamente, atingindo ou não seu objetivo.

3.2.1. *Chunks*

Os *chunks* são elementos da memória declarativa na teoria ACT-R e são utilizados para troca de informação entre os módulos através dos *buffers*. Um *chunk* é definido por *type* e *slots*, que são, respectivamente, seu tipo e seus atributos. Podemos pensar em *chunk-type* (tipo de *chunk*) como sendo uma categoria (por exemplo, pássaro) e em *slots* (atributos de *chunk*) como sendo características (por exemplo, cor e tamanho).

Os tipos de *chunk* de um modelo podem ser organizados em hierarquia. Na criação de um novo *chunk-type*, um tipo já existente pode ser especificado para ser seu tipo "pai" ou supertipo. Assim, o novo tipo irá herdar todos os atributos do "pai" e poderá também ter novos atributos, caso seja desejado. Estes conceitos lembram muito programação orientada a objeto, onde cada objeto (*chunk*) pertence a uma classe (*chunk-type*) e toda classe que herda de outra possui os mesmos atributos de sua superclasse (supertipo).

Um *chunk* possui um nome para ser referenciado, mas este nome não faz parte do *chunk* em si, de seus atributos. A seguir há um exemplo simples de *chunk*.

Tucano

isa	pássaro
tamanho	médio
cor	preto

Um *chunk* é exibido por seu nome, seguido da palavra *isa* que quer dizer “do tipo” e seu valor (tipo), ou seja, no exemplo acima o tipo do *chunk* (ou *chunk-type*) é pássaro. Logo em seguida há os atributos (*slots*) formando pares de palavras, sendo a primeira o nome do atributo e a segunda o valor deste.

3.2.2. Produções

Uma regra de produção é um conjunto de condições que controlam o comportamento, por exemplo:

SE a meta é classificar uma pessoa
E ela é solteira,
ENTÃO classifique-a como senhorita.

SE a meta é imprimir os números d1 e d2 na tela
E d1 + d2 = d3,
ENTÃO considere como meta imprimir d3 na tela.

A condição da regra de produção (a parte SE) consiste de uma especificação dos *chunks* nos vários *buffers*. A ação da produção (a parte ENTÃO) consiste em modificações dos *chunks* nos *buffers*, solicitações de outros *chunks* para serem colocados nos *buffers* ou solicitações de outras ações a serem realizadas (BEGOSSO, 2006).

3.2.3. Gerando Conhecimento

Para criar *chunks*, *chunk-types* e produções, é necessário utilizar comandos ACT-R. Todos estes comandos são funções Lisp e, portanto, são especificados entre parênteses. Nas subseções a seguir, será mostrado como criar cada elemento de conhecimento em ACT-R citado até agora.

3.2.3.1. Criando Tipos de *Chunks* (*Chunk-types*)

Para criar um novo tipo de *chunk* chamado pássaro, por exemplo, é necessário utilizar o comando *chunk-type*, que requer o nome do tipo e os nomes dos atributos. Um *chunk-type* pode ter quantos atributos forem necessário, basta especificá-los no momento de sua criação.

```
(chunk-type nome atributo-1 atributo-2 ... atributo-n)
```

Abaixo, segue um exemplo da criação do *chunk* pássaro, que possui os seguintes atributos: *tamanho* e *cor*.

```
(chunk-type pássaro tamanho cor)
```

3.2.3.2. Criando *Chunks*

O comando para criar *chunks* e automaticamente jogá-los na memória declarativa é o *add-dm*. Vários *chunks* podem ser criados de uma só vez, através de um só comando, como no exemplo abaixo:

```
(add-dm
        (tucano isa pássaro médio preto)
        (canário isa pássaro pequeno amarelo)
        (gavião isa passado grande marrom)
)
```

Cada *chunk* deve ser especificado em uma linha, entre parênteses, dentro do comando *add-dm*. A palavra *isa* indica que o *chunk*, por exemplo, de nome tucano é do tipo pássaro e possui os valores de atributos “médio” e “preto”, sendo respectivamente valores dos atributos tamanho e cor, definidos na criação do tipo do *chunk* anteriormente.

Caso seja necessário criar um *chunk* com algum atributo vazio, deve-se utilizar o valor *nil*, que indica que aquele atributo não possui valor nenhum, está vazio.

3.2.4. *Buffers*

Os *buffers* são a interface entre o sistema de memória procedural e os outros componentes da arquitetura ACT-R (os módulos). Por exemplo, o *buffer goal* (ou meta) é a interface para o módulo *goal*. Cada *buffer* pode armazenar um *chunk* por vez e as ações das produções afetam o conteúdo dos *buffers*. Essencialmente, os *buffers* trabalham como auxiliares para a criação e modificação dos *chunks*. Cada *buffer* será tratado no decorrer deste capítulo, junto de seu módulo correspondente.

3.2.5. Regras de Produção

Uma regra de produção é um par condição-ação. A parte de condição (também chamada de LHS – *left-hand side*, ou lado esquerdo) especifica um padrão de *chunks* que precisam estar presentes nos *buffers* para a regra ser aplicada, para que a produção seja executada. A parte de ação (também chamada de RHS – *right-hand side*, ou lado direito), que é tudo que vem após o símbolo "==>", especifica as ações a serem tomadas quando a produção for disparada, executada.

O comando para criação de uma produção deve iniciar um "p" antes do nome da mesma, tudo entre parênteses, como mostra a estrutura geral de uma regra de produção a seguir:

(p nome "comentário opcional"
    Testes de buffers (condição, LHS)
==>
    Alteração de buffers e solicitações (ação, RHS)
)

Cada produção dever ter um único nome e pode conter um comentário opcional para descrever claramente o que ela faz. Os testes de *buffers* consistem em um conjunto de padrões a serem combinados com o atual conteúdo dos *buffers*; se todos os padrões (ou condições) combinarem, ou seja, forem verdadeiros, então a produção é selecionada e executada.

É possível que mais de uma produção combine ao mesmo tempo, ou seja, esteja disponível para execução ao mesmo tempo; no entanto, somente uma por vez pode

ser disparada. Para isto o ACT-R utiliza uma função de resolução de conflitos. Após uma produção ser executada, a resolução de conflitos será novamente chamada até não possuir mais nenhuma produção que combine e o modelo pare, resultando em sucesso ao atingir seu objetivo ou erro.

A seguir, um exemplo de especificação de uma regra de produção (ACT-R, 1998):

```
(p counting-example
        =goal>
                isa             count
                state           incrementing
                number          =num1
        =retrieval>
                isa             count-order
                first           =num1
                second          =num2
==>
        =goal>
                number          =num2
        +retrieval>
                isa             count-order
                first           =num2
)
```

A produção acima diz que SE o *buffer goal* possuir um *chunk* do tipo *count*, com o atributo *state* de valor igual *increment*, há uma variável chamada =num1 que armazenará o valor do atributo *first* deste *chunk*, E se o *buffer retrieval* possuir um *chunk* do tipo *count-order*, com o atributo *first* igual à variável =num1, há uma variável chamada =num2 que armazenará o valor do atributo *second* deste *chunk*, ENTÃO mude o *buffer goal*, alterando o atributo *number* do *chunk* que ele possui para o valor da variável =num2, E solicite uma recuperação (*buffer retrieval*) de um *chunk* do tipo *count-order*, cujo valor de seu atributo *first* é igual ao valor da variável =num2.

3.2.5.1. Condições da Regra de Produção

Nota-se que na regra de produção os *buffers* são citados com o sinal “=” antes de seu nome e “>” depois. Logo após, o *chunk* no *buffer* deve ser testado pela palavra *isa*, que verifica se seu tipo é o especificado a frente. Pode ser testada também qualquer quantidade de atributos que for necessária.

As variáveis em uma produção tem seu nome iniciado pelo sinal “=”, sendo que podem ser utilizadas para dois propósitos: no LHS, para comparação de atributos entre diferentes *chunks*, sendo que a primeira vez que a variável é usada faz com que ela guarde o valor daquele atributo e as vezes seguidas ela apenas mantenha o mesmo valor para fins de comparação; no RHS, para a alteração de atributos nos *chunks* presentes nos *buffers*. Variáveis não recebem valores de atributos igual a *nil* (vazio).

Em resumo, a produção do exemplo anterior será disparada caso o *chunk* no *buffer goal* for do tipo *count*, o *chunk* no *buffer retrieval* for do tipo *count-order*, o atributo *state* do *chunk* no *buffer goal* ter o explícito valor *incrementing*, o atributo *number* do *chunk* no *buffer goal* for igual ao atributo *first* do *chunk* no *buffer retrieval* e existir algum valor no atributo *second* do *chunk* no *buffer retrieval*.

Algo interessante de se saber é que quando uma produção combina e é selecionada, ela demora 50 milissegundos para ser disparada. Este pode ser considerado uma simulação do tempo de preparação do cérebro humano entre a transição do estado de tomada de decisão para o estado de ação. Sobre o tempo tomado por um modelo em ACT-R será visto mais detalhes nas subseções seguintes.

3.2.5.2. Ações da Regra de Produção

As ações são especificadas de modo semelhante às condições, seguindo o mesmo padrão de escrita dos nomes dos *buffers*, porém com alguns detalhes importantes.

As modificações nos *buffers* são feitas a partir do sinal que precede o nome deste *buffer* no RHS. Por exemplo, esta ação modifica o valor do atributo do *chunk* presente no *buffer goal* para o valor da variável =num2, sem alterar mais nada:

```
=goal>
        number        =num2
```

Há três tipos de modificadores de *buffer* para ser usados no RHS, antes do nome do mesmo:

- **Modificador de *chunk*:** é representado pelo sinal "=" (como exemplo acima), e faz com que somente aquele atributo seja modificado, sem retirar o *chunk* do *buffer* ou fazer qualquer outra alteração não programada;
- **Solicitação de *chunk*:** é representado pelo sinal "+" (por exemplo, *+goal>*) e faz com que o atual *chunk* seja retirado do *buffer* para que um novo venha a ocupar seu lugar;
- **Limpeza de *chunk*:** é representado pelo sinal "-" e faz com que o *buffer* fique fazio, retirando qualquer que seja o *chunk* presente nele naquele momento.

3.2.6. Traço do Modelo

Quando um modelo ACT-R é executado, uma saída é exibida na aplicação. Esta saída é chamada de traço do modelo e é estruturada como mostra a Figura 9.

```
0.000   GOAL                SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
0.000   PROCEDURAL          CONFLICT-RESOLUTION
0.000   PROCEDURAL          PRODUCTION-SELECTED FIND-UNATTENDED-LETTER
0.000   PROCEDURAL          BUFFER-READ-ACTION GOAL
0.050   PROCEDURAL          PRODUCTION-FIRED FIND-UNATTENDED-LETTER
0.050   PROCEDURAL          MOD-BUFFER-CHUNK GOAL
0.050   PROCEDURAL          MODULE-REQUEST VISUAL-LOCATION
0.050   PROCEDURAL          CLEAR-BUFFER VISUAL-LOCATION
0.050   VISION              Find-location
0.050   VISION              FIND-LOC-FAILURE
0.050   PROCEDURAL          CONFLICT-RESOLUTION
0.050   ------              Stopped because no events left to process
```

Figura 9. Exemplo de um traço de modelo

O traço mostra os eventos ocorridos na execução do modelo, desde testes e resoluções de conflito até solicitações e alterações de *chunks*. São exibidas três colunas: o tempo em segundos e milissegundos em que o evento ocorreu a partir do início da execução do modelo, o módulo que foi utilizado, e o que foi feito especificamente naquele momento. Qualquer outra saída do modelo também é mostrada no traço, como por exemplo, um resultado atingido que deve ser mostrado para o usuário em forma de texto predeterminado na programação.

3.3. OS MÓDULOS

Os módulos em ACT-R são mecanismos que fornecem habilidades ao sistema de produção, como se fossem, por exemplo, nossa memória, nossos olhos e mãos. No começo havia somente o módulo declarativo e prodecimental, que armazenava informações na memória e executava produções. Mais tarde, outros módulos foram desenvolvidos por Mike Byrne e chamados de módulos de percepção e motor. Inicialmente foram referenciados como ACT-R/PM por trabalharem separados, mas atualmente são integrados na arquitetura ACT-R. Este conjunto de módulos permite que sejam criados modelos com habilidade motora, de visão, de audição e de fala, todas simuladas e representadas na teoria ACT-R (BOTHELL, 2007a).

Nesta seção serão descritos de maneira essencial os módulos utilizados para criar o modelo neste trabalho. Para maiores detalhes sobre cada módulo ou *buffer*, consulte Bothell (2007a).

3.3.1. Módulo *Declarative* (Memória Declarativa)

O módulo *declarative* fornece ao modelo uma memória declarativa que armazena os *chunks* que são gerados pelas produções e também um mecanismo para recuperá-los. A recuperação de *chunks* da memória declarativa depende de muitos fatores

que afetam a precisão e a velocidade com que uma parte pode ser recuperada, que é baseado em pesquisas de desempenho da memória humana.

A memória declarativa do modelo (DM – *declarative memory*) é constituída por todos os *chunks* que são explicitamente adicionados a ele pelo programador, bem como todos os pedaços que são retirados dos *buffers*. Sempre que um *chunk* é eliminado de um *buffer*, o módulo *declarative* guarda este *chunk* na memória do modelo. O processo de fusão compara o *chunk* que está sendo adicionado aos que já existem na DM.

A recuperação de *chunks* da memória declarativa é feita através de solicitações para o módulo. Uma solicitação especificar a descrição de um *chunk* que é desejado obter. Se há algum *chunk* na memória declarativa que combina com a especificação solicitada, então um destes *chunks* é colocado no *buffer retrieval* como resposta; caso contrário, o modelo informa o erro.

Como há possibilidade de mais de um *chunk* combinar quando uma solicitação é feita, há alguns parâmetros de solicitação que interferem na resolução deste conflito. Primeiramente, a configuração do parâmetro *:esc* determina grosseiramente qual será utilizado; se este parâmetro tiver valor igual a *nil*, então somente a combinação simbólica é considerada, ou seja, não há nada de relevante no *chunk* que o fará ter prioridade; caso *:esc* seja igual a *t*, então a seleção de cada *chunk* é retornada e o tempo que isto levará é controlado por um valor chamado *ativation* (ativação), mas que não será detalhado neste trabalho porque isto não é relevante na construção do modelo aqui realizada. Depois, há o parâmetro *:er*; se este parâmetro for igual a *t* então a escolha é feita randomicamente; se for igual a *:nil*, um processo determinístico é usado para que o mesmo *chunk* seja escolhido pelo modelo toda vez que o mesmo conjunto de *chunks* for retornado.

3.3.2. Módulo *Imaginal*

Este módulo apresenta um *buffer* chamado *imaginal* e tem a finalidade de criar novos *chunks*, que são representações internas de conhecimento do modelo, durante sua execução; é como se esta representação fosse suas “imagens internas”, por isso o nome *imaginal*, referente à imaginação. Como qualquer outro *buffer*, os *chunks* no módulo *imaginal* podem ser modificados por produções para gerar essas representações usando as ações do RHS, como visto anteriormente.

É importante ressaltar a forma como um *chunk* é inserido no *buffer* do módulo *imaginal*. Diferente do *chunk* do *buffer goal*, que é criado manual e previamente e colocado no *buffer* no início da execução do modelo, o módulo *imaginal* irá criar o *chunk* para o seu *buffer* em resposta a uma solicitação de uma produção.

Todas as solicitações para o módulo *imaginal* através do *buffer imaginal* são solicitações para criar um novo *chunk*, logo, no RHS, o nome deste *buffer* sempre será precedido do sinal “+”, que indica a inserção de um novo *chunk*. O módulo *imaginal* irá criar um novo *chunk* usando o tipo e os atributos solicitados na produção e o colocará no *buffer*. Vale destacar que cada *chunk* criado pelo módulo *imaginal* é também armazenado na memória declarativa, passando a ser novo conhecimento para o modelo durante aquela execução.

Um exemplo de criação de *chunk* usando o módulo *imaginal* é mostrado a seguir:

```
+imaginal>
        isa             pássaro
        tamanho         pequeno
```

Acima um *chunk* do tipo pássaro e com o atributo tamanho contendo o valor “pequeno” é criado. Se o *chunk-type* possuir mais atributos, mas estes não forem citados na criação, quer dizer que o valor dos mesmos será *nil*.

Um detalhe importante sobre as solicitações para este módulo é que o *chunk* não é imediatamente inserido no *buffer* como um resultado de uma solicitação. Ele leva 0,2 segundos antes de disponibilizar o *chunk*. Este é um aspecto importante do módulo,

seu tempo para construir a representação, como se fosse para uma pessoa imaginar, pensar em algo. Também só é possível criar um *chunk* por vez, sendo eficaz fazer um teste de disponibilidade do módulo toda vez que for utilizá-lo.

3.3.3. Módulo *Vision* (Visão)

Muitas tarefas envolvem interação com estímulos visuais e o módulo *vision* (ou módulo visual) provê uma maneira de adquirir informações visuais. É designado como um sistema para modelagem da atenção visual. Ele assume que há processos de baixo nível de percepção que geram as representações com o qual ele trabalha, como por exemplo a leitura de caracteres na tela e identificação de cores, mas não é adequado para processos de percepção detalhada, como a varredura de pixels em uma imagem muito grande. Inclui alguns mecanismos padrões para leitura de texto e outros recursos visuais simples, como também uma interface que pode ser usada para extensão quando necessário.

Este módulo possui dois *buffers*: o *buffer visual*, que representa o “o que” no módulo, e o *buffer visual-location*, que representa o “onde”, ambos descritos nas subseções seguintes. Como todos os módulos, ele também responde às buscas dos *buffers* solicitadas nas produções.

3.3.3.1. O *Buffer Visual-location*

Este *buffer* é responsável por armazenar um *chunk* que representa a localização de um objeto no campo visual, encontrado pelo módulo visual. Um *chunk* do tipo *visual-location* é composto por várias constantes (seus atributos), como mostrado no exemplo:

```
visual-location0-0-1
        isa             visual-location
        screen-x        130
        screen-y        160
        distance        15.0
        kind            text
        color           black
        value           text
        height          10
        width           7
        size            0.199
```

O módulo *vision*, além de atender solicitações, estará também colocando *chunks* neste *buffer* automaticamente, conforme for localizando objetos no campo visual. As solicitações feitas através do *buffer visual-loction* são compostas por constantes que fazem a busca basear-se no respectivo atributo. Se os atributos solicitados combinarem com algum objeto no campo visual, significa que o módulo *visual* é capaz de identificá-lo, retornando ao *buffer* a representação da localização deste objeto em forma de *chunk*. Se mais de um objeto combinar, mais *chunks* serão retornados, mas somente um poderá ser selecionado. Logo, aquele que foi criado mais recentemente tem a prioridade; caso eles possuem o mesmo tempo de existência, um será escolhido randomicamente. Enfim, se não houver objetos no campo visual que atenda a solicitação, o *buffer* ficará vazio e um erro será informado.

3.3.3.2. O *Buffer Visual*

Armazena um *chunk* que representa um objeto no campo visual. Seu uso primário é atender à localização adquirida pelo *chunk* do *buffer visual-location*, solicitando ao módulo visual que identifique o objeto naquele local do campo visual, processe e retorne detalhes sobre o que é o objeto. Isto fará com que o módulo mova sua atenção para aquele ponto, evento também chamado de *move-attention*.

A Figura 10 mostra um exemplo do módulo *vision* identificando um objeto.

```
0.100   PROCEDURAL          PRODUCTION-FIRED ATTEND-LETTER
0.100   PROCEDURAL          MOD-BUFFER-CHUNK GOAL
0.100   PROCEDURAL          MODULE-REQUEST VISUAL
0.100   PROCEDURAL          CLEAR-BUFFER VISUAL-LOCATION
0.100   PROCEDURAL          CLEAR-BUFFER VISUAL
0.100   VISION              Move-attention VISUAL-LOCATION0-0-1 NIL
0.100   PROCEDURAL          CONFLICT-RESOLUTION
0.185   VISION              Encoding-complete VISUAL-LOCATION0-0-1 NIL
0.185   VISION              SET-BUFFER-CHUNK VISUAL TEXT1
```

Figura 10. Traço que mostra o módulo *vision* identificando um objeto

Nota-se uma diferença no tempo de execução nas últimas duas linhas do exemplo de traço. Isto ocorre porque o módulo visual leva 85 milissegundos para mover a atenção até um local e criar o objeto visual.

Vale destacar que um *chunk* no *buffer visual* é um objeto visual, ou seja, representações codificadas para o modelo de um objeto no campo visual do mesmo. Logo, se um *chunk* deste tipo possui o valor “3”, significa que o modelo entende este valor como uma representação gráfica, visual, não como um valor aritmético disponível para utilização em operações matemáticas.

3.3.4. Módulo Motor

Este módulo tem por objetivo simular as mãos de um operador manipulando dispositivos computacionais. Seu conceito é baseado no trabalho de Kieras & Meyer (1996), *EPIC’s Manual Motor Processor*. O EPIC (Executive Process / Interative Control) é outra arquitetura cognitiva, e é bastante semelhante em muitos aspectos com o ACT-R. Ele provê ao modelo provê ao modelo a habilidade de operar um teclado de computador e mouse virtuais, como padrão, mas é possível estender as ações e dispositivos a serem utilizados caso seja desejado (algo que não será

tratado neste trabalho). Bothell (2007a) fornece mais detalhes sobre este tipo de utilização.

As mãos do operador simulado no modelo são responsáveis por operar um teclado virtual que é organizado da maneira mostrada na Figura 11.

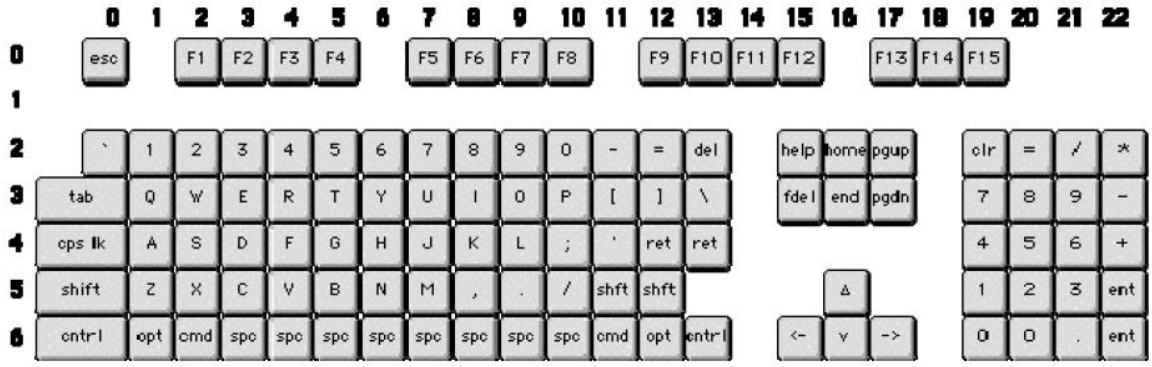

Figura 11. Representação do teclado virtual utilizado pelo módulo motor (BOTHELL, 2007a)

As teclas são encontradas pelo módulo utilizando os índices de coluna e linha, respectivamente. Sendo assim, se o modelo deve pressionar a tecla “G”, a representação interna da posição desta no teclado é (5 4), que significa coluna 5 e linha 4.

O mouse virtual disponível para o modelo possui apenas um botão e é como se fosse operado pela mão direita quando utilizado, considerando que sua posição relativa ao teclado virtual é (28 2), ou seja, coluna 28 e linha 2, como se estivesse ao lado direito do teclado virtual.

Estes dispositivos virtuais recebem ações do módulo motor e passam as informações destas ao sistema central de produção do modelo e este assume como se tivesse vindo de um usuário real do computador, até mesmo movimentando o cursor do mouse na tela caso seja desejado. Por padrão, as mãos do modelo iniciam

sua execução na posição regular de digitação, isto é, com os dedos indicador esquerdo e direito sobre as teclas “F” (4 4) e “J” (7 4), respectivamente.

Geralmente, solicitações de movimento requerem a especificação do tipo do movimento, denominado seu estilo de movimento (o qual é determinado pelo tipo do *chunk* da solicitação) e um ou mais parâmetros, tais como a mão ou dedo que irá fazer o movimento. O módulo motor inclui vários estilos de movimentos baseados no *EPIC's Manual Motor Processor*, sendo que os básicos são os seguintes:

- ***Punch*:** é um simples movimento para baixo seguido de um movimento para cima de um dedo, resultando em um pressionar de uma tecla que já esteja diretamente abaixo deste dedo;

- ***Peck*:** este é um movimento direcionado de um dedo até uma localização específica seguido do pressionamento da tecla que se encontra naquele local, tudo como um movimento contínuo;

- ***Peck-recoil*:** o mesmo que o estilo *peck* acima, porém, após pressionar a tecla, o dedo retorna à posição que estava quando começou este movimento;

- ***Ply*:** move um dispositivo (por exemplo, um mouse) até um dado local;

- ***Point-hand*:** move uma mão para uma nova localização.

Como exemplo de utilização dos estilos de movimento, pode-se criar uma produção com o seguinte teste, que fará com que o dedo médio da mão esquerda do modelo pressione a tecla que estiver exatamente abaixo dele naquele momento:

```
=manual>
        isa     punch
        hand    left
        finger  middle
```

O módulo motor não coloca nenhum *chunk* no seu *buffer*, chamado *buffer manual*, em resposta às solicitações. O *buffer* deve estar sempre vazio, sendo o estado mais importante para o modelo quando este utiliza os recursos motores.

Este módulo possui três estados internos: *preparation*, *processor* e *execution*. Quando uma solicitação de movimento é feita, ele passará por três fases referentes aos três estados, que são preparação, iniciação e execução, respectivamente. Este trabalho não abordará os detalhes de cada fase. Para mais detalhes sobre o assunto, Bothell (2007a) fornece toda informação necessária.

4. A CONSTRUÇÃO DOS MODELOS

Este capítulo apresenta o *framework* utilizado para criação dos modelos e explicará detalhes a construção e constituição de cada um. O conteúdo sobre a arquitetura ACT-R descrito até o presente momento servirá de base para a compreensão dos códigos dos modelos e extensão das informações sobre o mesmo, pois serão abordados detalhes ainda não citados no capítulo anterior, como por exemplo, detalhes sobre um parâmetro específico de um *buffer* que deve ser compreendido para que se entenda a criação e funcionamento dos modelos.

O primeiro modelo terá a função de realizar a contagem de um número até outro, ambos de um único dígito. O segundo modelo irá efetuar a leitura de uma letra no campo visual, identificá-la, processá-la e pressionar a tecla, no teclado virtual, correspondente aquela letra lida. Sendo assim, no primeiro modelo serão utilizados apenas os módulos de memória (pois este trabalhará com o sistema cognitivo); no segundo modelo, além dos módulos de memória, também serão utilizados dois de interface com o ambiente externo: o de visão (responsável por enxergar a letra na janela) e o motor (para pressionar a devida tecla no teclado virtual).

4.1. O SOFTWARE DE DESENVOLVIMENTO (*FRAMEWORK*)

Esta seção descreve as funcionalidades básicas do software (ou *framework*) utilizado para a construção dos modelos a serem simulados – o *ACT-R 6 Standalone Environment*, encontrado na seção de downloads do site da teoria (ACT-R, 1998). Esta versão tem a vantagem de executar sem a necessidade de qualquer instalação no computador, permitindo uma iniciação prática e simples.

O *framework* não tem nada diretamente relacionado com a teoria ACT-R, é apenas uma interface para o programador gerar código Lisp. Porém, por ser designado justamente para o desenvolvimento de modelos ACT-R, ele possui um painel de controle (v. Figura 12) repleto de botões que servem para executar alguns comandos Lisp e exibir informações sobre os modelos, tais como *chunks* presentes em cada *buffer*, valor de *chunks*, produções, etc. Mais detalhes serão abordados no decorrer desta seção.

Contudo, é através do *framework* que o programador executa, testa e analisa o modelo previamente criado em algum editor Lisp ou editor de texto qualquer. O modelo criado neste trabalho foi editado no editor de texto padrão do Windows 7, o *Notepad* (ou Bloco de Notas), mas poderia ter sido feito em qualquer outro aplicativo, desde que o arquivo seja salvo com a extensão ".lisp" (que é a extensão dos arquivos Lisp), pois como já citado neste trabalho, o código dos modelos ACT-R são comandos Lisp adaptados à implementação da teoria ACT-R.

Os comandos e nomes em ACT-R não são *case-sensitive*, ou seja, não há diferença entre letras minúsculas e maiúsculas, pois sempre serão interpretados da mesma forma.

4.1.1. A Janela *Listener*

Além do painel de controle, o *framework* possui uma janela chamada *Listener*, mostrada na Figura 12, que exibe as saídas da execução do modelo (como por exemplo, mensagens da compilação e o traço do modelo) e também uma caixa de entrada, utilizada para o usuário digitar comandos Lisp a fim de interagir com o modelo.

Todos os comandos Lisp que forem digitados em sua entrada farão efeito na execução do modelo, sendo que os principais comandos possuem um botão de ação equivalente no painel de controle.

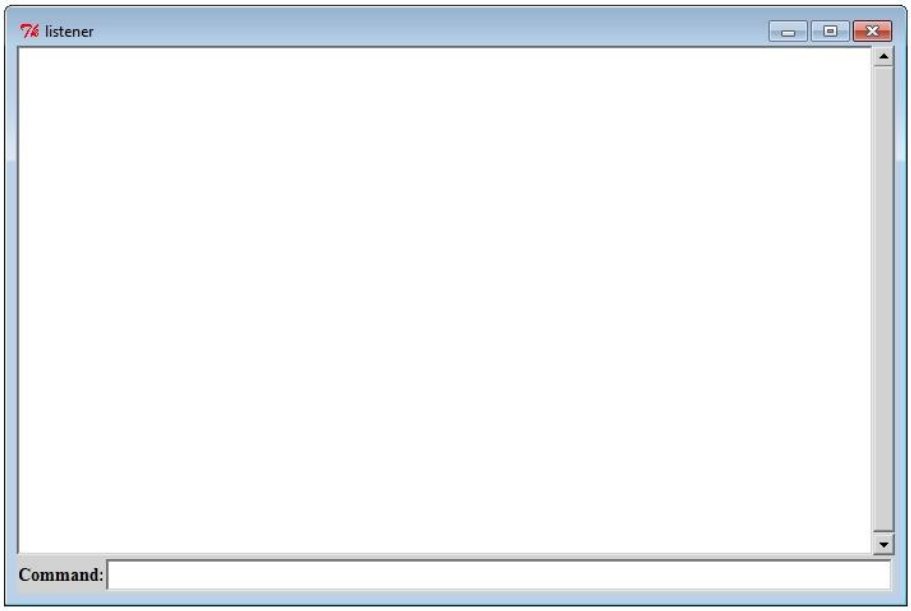

Figura 12. Janela *Listener* do ACT-R 6.0

4.1.2. O Painel de Controle

Os botões do painel de controle são agrupados em seções de acordo com suas funcionalidades (v. Figura 13).

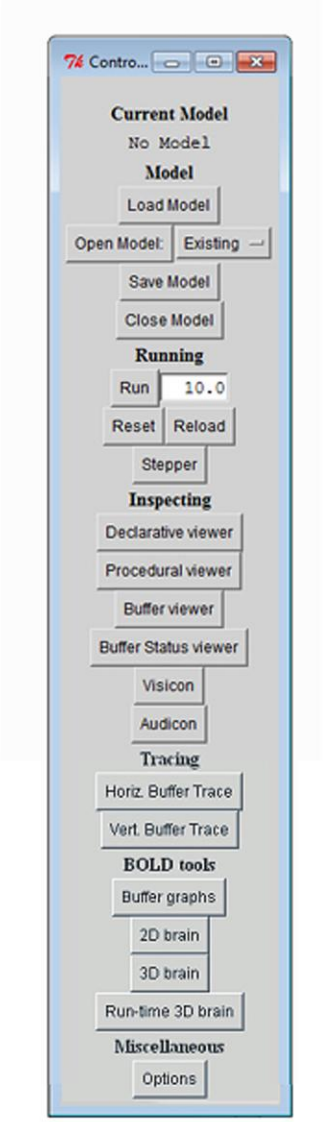

Figura 13. Painel de controle do ACT-R 6.0.

A seção *Current Model* apenas informa o nome do modelo que está atualmente carregado. Quando o *framework* for iniciado, não haverá nenhum modelo carregado na memória, logo *Current Model* será vazio, informando não existir modelo carregado (*No Model*). As demais seções e suas funcionalidades serão brevemente explicadas a seguir.

4.1.2.1. A Seção *Model*

Esta seção possui controles para trabalhar com os arquivos de modelos. O botão *Load Model* é usado pra carregar um modelo no *framework*, escolhido pelo usuário através de uma caixa de diálogo. Quando um modelo é carregado, uma caixa de mensagem será exibida na tela informando erro, alerta ou sucesso no comando. Se a mensagem for semelhante à apresentada na Figura 14, o modelo foi carregado corretamente e está pronto para uso. Quando o código contiver algum erro de compilação, a mensagem irá indicar o erro para que o programador faça a correção necessária.

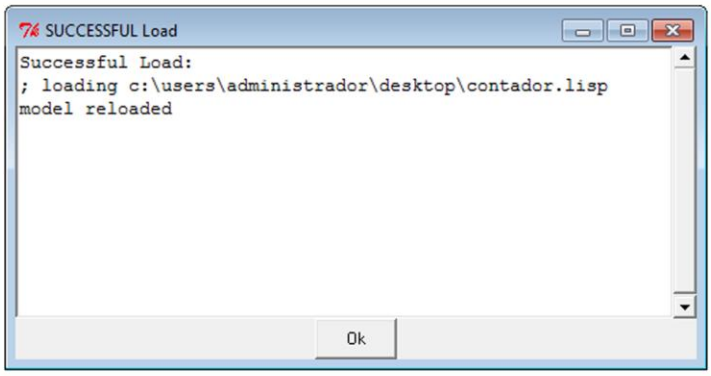

Figura 14. Caixa de mensagem exibida ao carregar corretamente um modelo

O botão *Open Model* abre um arquivo de modelo para a edição do mesmo. Ele utiliza o editor do próprio *framework* que vem junto da versão. Em frente a este botão há uma caixa de escolha que por padrão esta marcado *Existing*. Isto informa se o arquivo do modelo a ser aberto é um já existente (*Existing*) ou um arquivo novo, que será criado naquele momento *(New)*. Antes de clicar em *Open Model*, o programador deve escolher qual destas duas opções atende sua necessidade. Note que para abrir um arquivo, não deve existir nenhum outro aberto; caso exista, uma mensagem de erro será exibida e o comando será cancelado.

O botão *Save Model* salva o arquivo de modelo que foi aberto pelo botão *Open Model*. Já o botão *Close Model* irá fechar este arquivo sem salvar qualquer alteração feita.

4.1.2.2. A Seção *Running*

Contém controles para execução e restauração do modelo. O botão *Run* é equivalente a digitar o comando Lisp *run* na entrada da janela *Listener*. A caixa de texto presente ao lado do botão contém o tempo de execução do modelo (*run-time*), ou seja, o tempo máximo que o modelo irá executar, em segundos. O comando *run* irá executar o código do modelo de forma sequencial, do início ao fim. Fará com que o sistema central de produção ACT-R trabalhe em função da meta do modelo até que não haja mais eventos a serem executados ou o tempo de execução termine.

O botão *Reset* é usado para reiniciar o modelo ao seu estado atual, equivalente a digitar o comando *reset* na entrada da janela *Listener*.

Já o botão *Reload*, equivalente ao comando *reload*, recarrega o atual modelo na memória novamente. Ele não fará efeito caso o modelo esteja em execução ou a ferramenta *Stepper* esteja aberta.

Esta ferramenta denominada *Stepper*, tem a função de executar o modelo passo a passo, pausando a execução a cada evento. É acionada através do botão de mesmo nome e pode ser a mais útil em todo o software. Quando acionada, uma janela é aberta, como mostrada na Figura 15.

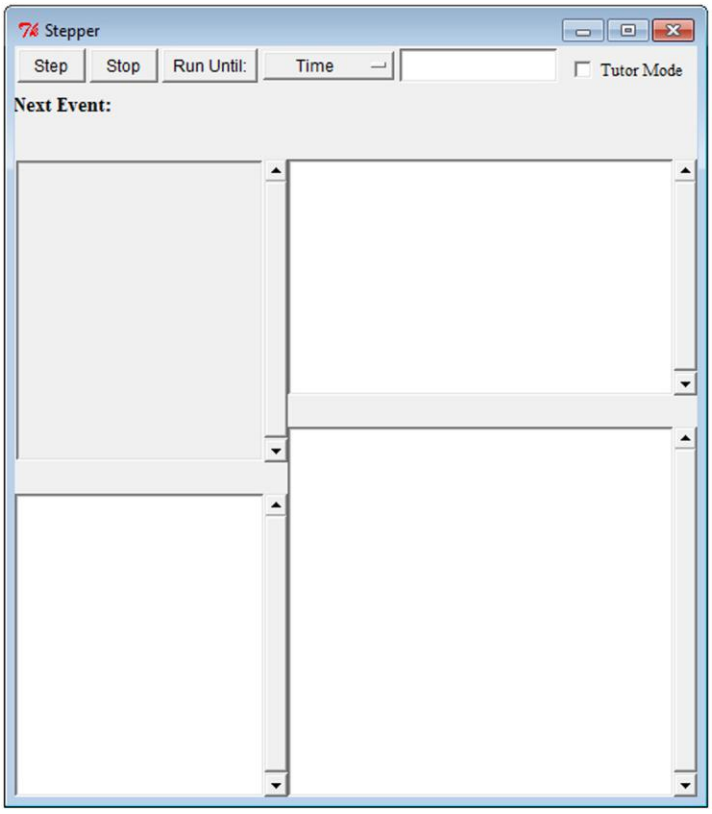

Figura 15. Ferramenta *Stepper* do ACT-R 6.0

Quando um modelo está carregado e esta ferramenta acionada, o usuário poderá pressionar o botão *Run* no painel de controle para iniciar a execução do modelo. Nisto, o primeiro evento será exibido *Next event*, na janela *Stepper*. Assim, se o botão *Step* for pressionado, este evento será gravado no traço, exibido na saída da janela *Listener* e, em consequência, o próximo evento a ser executado aparecerá em *Next event*. Este botão poderá ser pressionado até que o modelo finalize sua execução e a cada pausa antes de um evento, o *buffer*, *chunks* e produção envolvidos são exibidos na janela da ferramenta. Assim e possível analisar o comportamento do modelo a cada instante, conferir valores de parâmetros, quais os possíveis *chunks* ou produções para o próximo evento, qual *chunk* está sendo carregado em qual *buffer*, etc. A Figura 16 mostra a janela da ferramenta carregada.

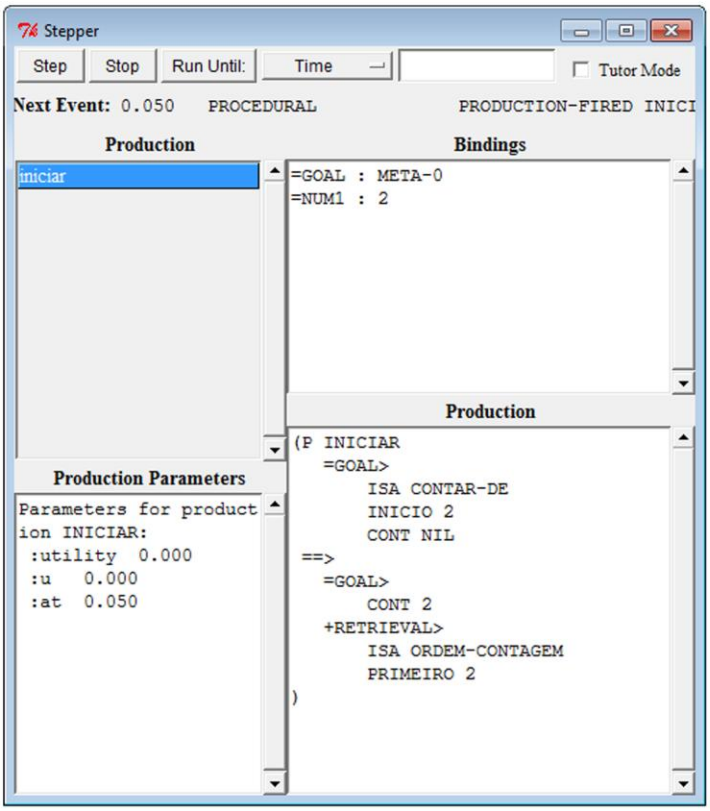

Figura 16. Exemplo da ferramenta *Stepper* com um evento carregado

O botão *Stop* da janela *Stepper* faz com que a ferramenta encerre. Caso um modelo esteja no meio de sua execução, um evento de finalização é realizado e a execução para onde estiver.

Ao lado de *Stop*, há o botão *Run until* que trabalha junto da caixa de seleção a sua direita e a caixa de texto, em seguida. Este tem a função de executar o modelo normalmente, pausando somente quando chegar no tempo desejado, por exemplo. A caixa de seleção informa que tipo de dado será buscado, dentre eles *Time* (tempo), *Production* (produção) e *Module* (módulo). Se a opção *Time* estiver selecionada, a caixa de texto a sua direita deverá conter o tempo, em segundos, que o usuário deseja que a execução do modelo pause; logo, se ele informar “2” e pressionar *Run Until*, quando a execução chegar em 2 segundos, uma pausa acontecerá, como as pausas do botão *Step*. A opção *Prodution* requer o nome de uma produção que o usuário deseja que a execução do modelo pause quando esta

atingir tal produção. Já a opção *Module* faz com que a execução do modelo pause no momento que alguma produção for utilizar o módulo especificado no campo de texto, como por exemplo, o módulo *vision* ou *declarative*.

Finalmente, a caixa de marcação *Tutor Mode* na janela do *Stepper* fará com que a ferramenta pergunte ao usuário os valores envolvidos na produção do atual evento, fazendo-o clicar nos chunks e *buffers* marcados na tela, digitando o valor que ele acredita conter em cada um. Caso esteja correto, o valor será mostrado na janela do *Stepper*; se estiver errado, nada ocorrerá, forçando o usuário a tentar novamente.

4.1.2.3. A Seção *Inspecting*

Esta seção contém botões para a visualização de informação detalhada de componentes em particular do sistema. Podem ser muito úteis em uso conjunto com o *Stepper* para analisar o atual estado das coisas antes e depois que um certo evento ocorrer.

O botão *Declarative Viewer* abre uma janela que exibe todos os *chunks* gravados na memória declarativa do modelo atualmente carregado. Se o usuário clicar sobre o nome de um *chunk*, os detalhes (tais como tipo e atributos) aparecerão na caixa à direita. No topo desta janela há uma caixa de seleção chamada *Filter*; que serve para filtrar os *chunks* na memória e exibir somente aqueles relacionados com o valor do filtro. Na Figura 17 pode ser visualizada a janela da ferramenta com alguns *chunks* na lista a esquerda e um deles selecionado, exibindo seus detalhes a direita.

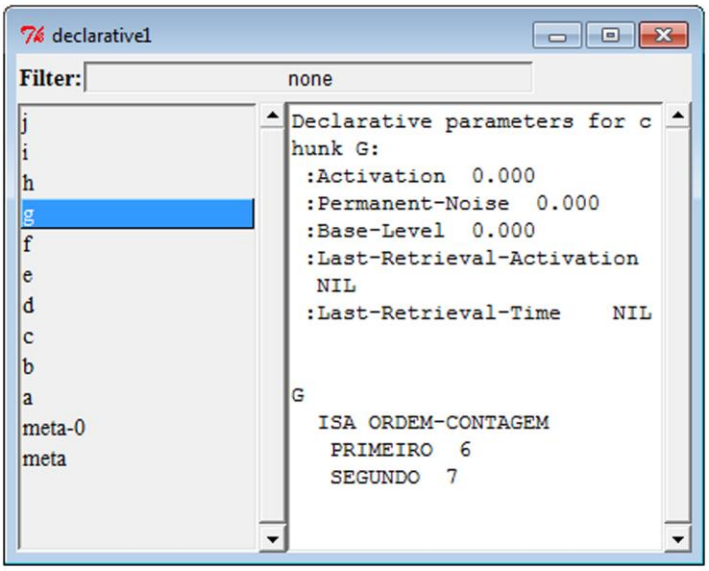

Figura 17. Janela *Declarative Viewer* do ACT-R 6.0

O *Procedural Viewer* funciona de maneira semelhante ao *Declarative Viewer*, mas ao invés de tratar a memória declarativa, trata a memória procedural, exibindo as produções existentes no modelo. Ainda no topo desta janela existe o botão *Why not?*, que abre uma outra janela, funcionando como um *debug* e informando o usuário se há algum erro na produção, onde está o erro e porque do erro. A Figura 18 mostra a janela da ferramenta com um exemplo carregado, contendo três produções na lista a esquerda (iniciar, incrementar e parar), sendo que a produção iniciar está selecionada, tendo seus detalhes exibidos à direita.

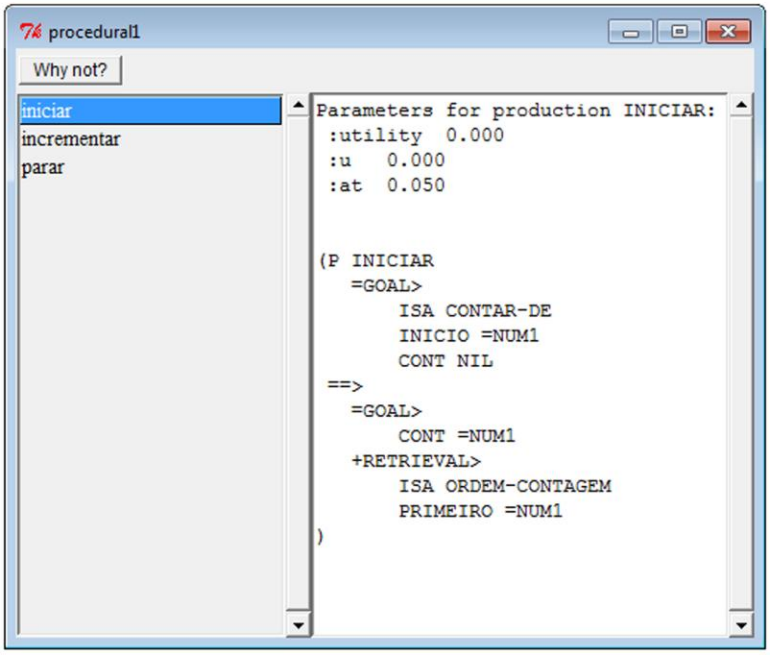

Figura 18. Janela *Procedural Viewer* do ACT-R 6.0

O botão *Buffer Viewer* abre uma janela que exibe uma lista de todos os *buffers* ACT-R. Se o usuário clicar em álbum *buffer*, no campo ao lado direito será exibido seu conteúdo, ou seja, qual *chunk* está atualmente carregado naquele *buffer*. Caso o *buffer* não esteja em uso, apenas informará que o *buffer* está vazio, como mostra a Figura 19, que exemplifica o *buffer manual* sendo selecionado na lista e a ferramenta informando o usuário de que este está vazio.

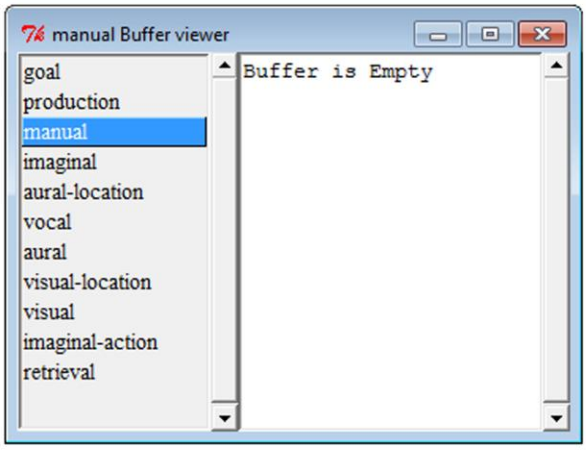

Figura 19. Janela *Buffer Viewer* do ACT-R 6.0

O *Buffer Status Viewer* abre uma janela que mostra ao usuário os *buffers* ATC-R e os detalhes de seu atual estado na execução do modelo, quando selecionado na lista. Este estado envolve informações do tipo falso (*nil*) ou verdadeiro (*t*). Por exemplo, se o *buffer* está vazio, se está cheio, se foi solicitado, se está livre para ser solicitado, se está ocupado, etc. A Figura 20 mostra um exemplo de exibição do status do *buffer retrieval* após a execução de um modelo qualquer, utilizando a ferramenta *Buffer Status Viewer*.

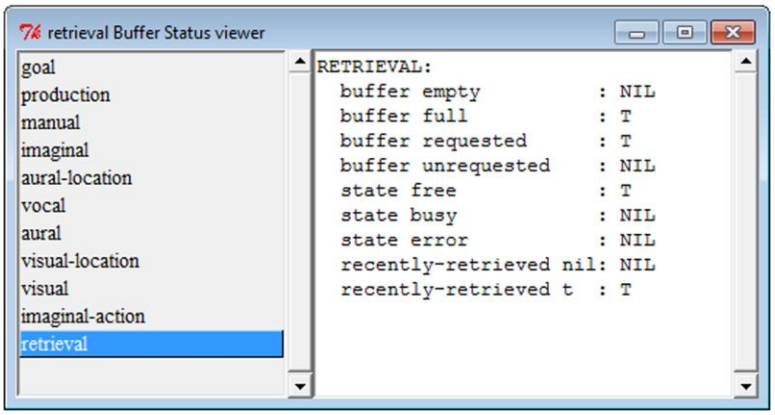

Figura 20. Janela *Buffer Status Viewer* do ACT-R 6.0

O botão *Visicon* abre uma janela exibindo as informações atualmente disponíveis para o módulo *vision* do modelo, ou seja, os objetos no campo de visão identificados pelo módulo. Este botão é equivalente ao comando *print-visicon* que pode ser digitado na entrada da janela *Listener*, fazendo com que estas informações sejam exibidas na saída.

Por último, o botão *Audicon* tem a funcionalidade semelhante ao *Visicon*, porém exibe as informações que estão atualmente no módulo de audição, que não é discutido neste trabalho.

4.1.2.4. A Seção *Tracing*

As ferramentas desta seção provêm uma representação gráfica do traço do modelo, facilitando o entendimento do progresso da execução do modelo e disponibilizando acesso direto às informações das produções e *chunks* envolvidos em cada evento.

Há duas ferramentas com este recurso. A primeira é acionada pelo botão *Horiz. Buffer Trace*, conforme a Figura 21.

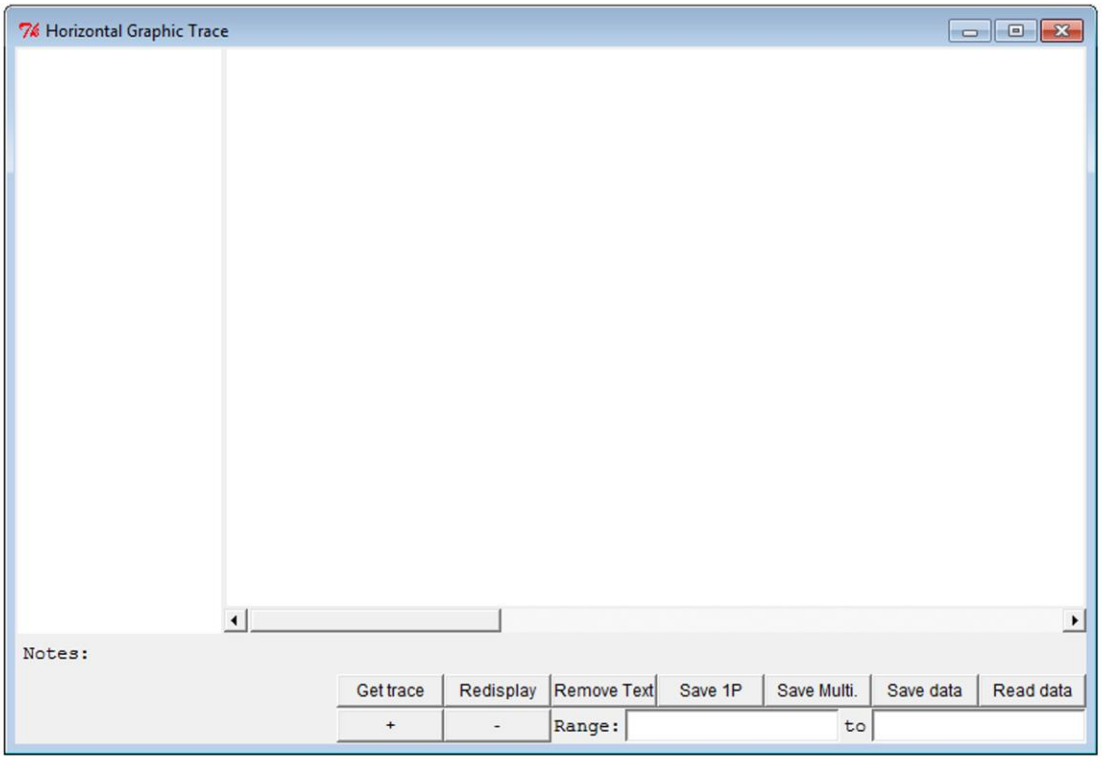

Figura 21. Janela *Horiz. Buffer Trace* do ACT-R 6.0

Quando a ferramenta é acionada, a janela não conterá nenhum gráfico, pois não há informação disponível, como mostra a Figura 21. Esta informação fica disponível se algum modelo for executado. Dessa forma, o usuário poderá clicar no botão *Get trace*, que faz a leitura do traço do modelo recém executado e o exibem em forma de gráfico na janela da ferramenta. O gráfico é apresentado de maneira a organizar as produções e *chunks* em forma de bloco à frente do *buffer* que foi utilizado. Os *buffers* são apresentados em uma lista ao lado esquerdo da janela. Na parte inferior do gráfico há a marcação do tempo de execução em que ocorreu tal evento, em segundos. Um exemplo do gráfico gerado será mostrado e analisado em cada modelo criado nas seções seguintes.

Os demais botões fornecem funcionalidades que não são importantes agora, portanto não serão explicados neste trabalho. Por exemplo, é possível retirar o texto da visualização do gráfico (botão *Remove text*), salvar o gráfico como uma imagem ".eps" (botão *Save 1P*), etc. Para mais informações, ver Bothell (2007a).

O outro botão da seção *Tracing*, chamado *Vert. Buffer Trace*, aciona uma ferramenta que tem a mesma função que a *Horiz. Buffer Trace*, a diferença é que o gráfico é exibido verticalmente e não horizontalmente. Os botões existentes na janela têm as mesmas funcionalidades e a maneira de operação é idêntica, somente a visualização que muda de uma ferramenta para a outra.

4.1.2.5. A Seção *Bold Tools* e *Miscellaneous*

Estas duas últimas seções do painel de controle não serão explicadas neste trabalho. Bothell (2007a) disponibiliza todos os detalhes sobre isto no material referenciado.

4.2. O MODELO CONTADOR.LISP

O primeiro modelo apresentado neste trabalho é um contador, que realiza uma contagem crescente de um número até outro, ambos de um único dígito. Este será construído para contar de 2 até 4, ou seja, o modelo simula um ser humano gerando os valores 2, 3 e 4. O resultado será exibido na tela, junto do traço do modelo, conforme for realizando a contagem de cada número. Ele trabalhará apenas com o sistema cognitivo, utilizando os módulos *declarative*, *procedural* e *goal*, que são os módulos de memória declarativa, memória procedural e meta, respectivamente, já descritos no capítulo anterior.

A seguir apresenta-se o código do modelo, seguido das subseções explicando cada parte do mesmo. As palavras encontradas na frente de um ponto-e-vírgula “;” não influem na execução pois são comentários do código, utilizados para organizar e orientar o programador.

```
(clear-all)

; Início do código do modelo Contador.lisp
(define-model contador

        ; Configuração dos parâmetros do modelo
        (sgp
                :esc t
                :lf .05
                :trace-detail high
        )

        ; Declaração dos tipos de chunk
        (chunk-type ordem-contagem primeiro segundo)
        (chunk-type contar-de inicio fim cont)

        ; Criação dos chunks
        (add-dm
                (a      ISA     ordem-contagem     primeiro 0     segundo 1)
                (b      ISA     ordem-contagem     primeiro 1     segundo 2)
                (c      ISA     ordem-contagem     primeiro 2     segundo 3)
                (d      ISA     ordem-contagem     primeiro 3     segundo 4)
                (e      ISA     ordem-contagem     primeiro 4     segundo 5)
                (f      ISA     ordem-contagem     primeiro 5     segundo 6)
                (g      ISA     ordem-contagem     primeiro 6     segundo 7)
                (h      ISA     ordem-contagem     primeiro 7     segundo 8)
                (i      ISA     ordem-contagem     primeiro 8     segundo 9)
                (j      ISA     ordem-contagem     primeiro 9     segundo 0)
                (meta   ISA     contar-de      inicio 2        fim 4)
        )

        ; Produções
        (p iniciar
                =goal>
                        ISA     contar-de
                        inicio  =num1
                        cont    nil
        ==>
                =goal>
                        cont    =num1
                +retrieval>
                        ISA     ordem-contagem
                        primeiro =num1
        )
        (p incrementar
                =goal>
                        ISA     contar-de
                        cont    =num1
                        - fim   =num1
                =retrieval>
```

```
                ISA     ordem-contagem
                primeiro =num1
                segundo =num2
        ==>
                =goal>
                        cont    =num2
                +retrieval>
                        ISA     ordem-contagem
                        primeiro =num2
                !output!(=num1)
        )
        (p parar
                =goal>
                        ISA     contar-de
                        cont    =num
                        fim     =num
        ==>
                -goal>
                !output!(=num)
        )

        ; Configuração do chunk de meta
        (goal-focus meta)
) ; Fim do código do modelo Contador.lisp
```

4.2.1. O Início do Código

O primeiro comando visto no modelo Contador.lisp é o comando *clear-all*. Ele serve para reiniciar o ACT-R para um estado completamente limpo, deixando todos os *buffers* e módulos livres de quaisquer possíveis restos de informação utilizados anteriormente. Não deve ser o primeiro comando existente no arquivo, mas é necessário que esteja antes da definição de algum modelo, que é realizada pelo comando *define-model*:

```
(clear-all)

; Início do código do modelo Contador.lisp
(define-model contador
        ...
)
```

O comando *define-model* é usado para especificar exatamente quais os componentes do modelo e dar um nome ao mesmo, localizado logo em seguida do comando. Neste caso, o nome do modelo é “contador”. Tudo que existir entre o nome do modelo e o parêntese que fecha o comando *define-model*, encontrado na última linha do código, é considerado a instanciação e configuração inicial do modelo. Os comandos são processados seqüencialmente conforme vão sendo requisitados, ou seja, quando uma produção é executada, suas linhas são executadas uma a uma, da primeira a última, como qualquer outro programa de computador.

Logo iniciando o interior do código do modelo há a configuração de seus parâmetros, dada pelo comando *sgp*:

```
; Configuração dos parâmetros do modelo
(sgp
        :esc t
        :lf .05
        :trace-detail high
)
```

O comando *sgp* é utilizado para configurar ou obter o valor dos módulos do modelo. Basta o programador informar, dentro dos parênteses e após o nome do comando, os parâmetros que deseja configurar, seguido de seus respectivos valores. Note que cada parâmetro tem seu nome precedido de dois pontos “:”. No modelo Contador.lisp, três parâmetros são tratados:

- O parâmetro ***:esc***, sigla de *Enable Subsymbolic Computations* (que significa Habilitar Computações Subsimbólicas), especifica se os módulos deverão trabalhar de forma puramente simbólica ou utilizar seu processamento simbólico completo. O valor padrão é *nil*, que faz com que os módulos trabalhem de forma puramente simbólica; se o valor for alterado para *t*, então os módulos deverão usar quaisquer computações subsimbólicas que fornecerem (por exemplo, de utilidade para seleção de produção no módulo *procedural* e ativação para seleção de *chunks* no módulo *declarative*). No modelo Contador.lisp, o valor de *:esc* será *t*;

- O parâmetro ***:lf*** é o valor do fator latência (F) na equação de tempo de retorno dos módulos. O valor padrão é 1, mas aqui será configurado para 0.05, a fim de retornar resultados mais rapidamente;

- Por último, o ***:trace-detail*** informa o nível de detalhe do traço do modelo, controlando quais eventos poderão ser exibidos ou não. Pode ser configurado com os valores *high*, *medium* ou *low*, que significam alto, médio e baixo, respectivamente. O valor padrão é *medium*, mas neste modelo ele será configurado para *high*. Assim, todos os eventos que resultarem em uma saída diferente de *nil* serão exibidos no traço.

4.2.2. A Declaração de *Chunk-types*

Prosseguindo na leitura do código, encontram-se as linhas:

```
(chunk-type ordem-contagem primeiro segundo)
(chunk-type contar-de inicio fim cont)
```

Estes são os dois tipos de *chunk* que o modelo Contador.lisp trabalha, definidos conforme explicado no Capítulo 3. O tipo ordem-contagem possui dois atributos: *primeiro* e *segundo*; *chunks* deste tipo guardam um número no atributo *primeiro* e o número que subsequente no atributo *segundo*. Por exemplo, o *chunk* “b” possui os atributos *primeiro* igual a 1 e *segundo* igual a 2; logo, o modelo sabe que após o número 1 vem o número 2, podendo realizar uma contagem sabendo que está seguindo corretamente. Esta ideia aplica a teoria de como os números são organizados na memória humana: é sabido que após o número 1 vem o 2, e que após o 2 vem o 3, e assim por diante; então, quando realizamos uma contagem, subconscientemente colocamos um após o outro em uma fila conforme progredimos a contagem. O modelo Contador.lisp trabalha de maneira semelhante, tentando simular exatamente este comportamento.

Já o segundo tipo de *chunk* é o contar-de e possui três atributos: *inicio*, *fim* e *cont*. Um *chunk* deste tipo é utilizado como *chunk* meta para o modelo, sendo que seu

atributo *inicio* guardará o número pelo qual a contagem será iniciada, o atributo *fim* guardará o número em que a contagem irá parar, e o atributo *cont* serve como auxiliar na troca de valores durante a execução do modelo.

4.2.3. A Criação dos *Chunks*

Logo após a declaração dos *chunk-types*, encontra-se o comando *add-dm*, que serve para criar *chunks* na memória declarativa do modelo, gerando seu conhecimento inicial.

```
(add-dm
    (a     ISA   ordem-contagem   primeiro 0   segundo 1)
    (b     ISA   ordem-contagem   primeiro 1   segundo 2)
    (c     ISA   ordem-contagem   primeiro 2   segundo 3)
    (d     ISA   ordem-contagem   primeiro 3   segundo 4)
    (e     ISA   ordem-contagem   primeiro 4   segundo 5)
    (f     ISA   ordem-contagem   primeiro 5   segundo 6)
    (g     ISA   ordem-contagem   primeiro 6   segundo 7)
    (h     ISA   ordem-contagem   primeiro 7   segundo 8)
    (i     ISA   ordem-contagem   primeiro 8   segundo 9)
    (j     ISA   ordem-contagem   primeiro 9   segundo 0)
    (meta  ISA   contar-de   inicio 2   fim 4)
)
```

Aqui são criados dez *chunks* do tipo *ordem-contagem* e um *chunk* do tipo *contar-de*. Os dez do tipo *ordem-contagem* guardam os números de 0 a 9 junto de seus respectivos procedentes, um no atributo *primeiro*, outro no atributo *segundo*. O *chunk* meta é iniciado com o atributo *inicio* valendo 2 e o atributo *fim* valendo 4; o atributo *cont*, que não foi citado na criação, é logo atribuído como *nil*. Este *chunk* do tipo *contar-de* está informando que o modelo irá executar uma contagem de 2 até 4.

4.2.4. As Produções

4.2.4.1. A Produção *Iniciar*

Na parte de testes desta produção, o LHS, há uma busca no *buffer goal* por um *chunk* do tipo contar-de, que contenha algum valor em seu atributo *inicio*, fazendo a variável =num1 guardar este valor, e que também contenha o atributo *cont* vazio, ou seja, valor igual à *nil*.

```
=goal>
        ISA     contar-de
        inicio  =num1
        cont    nil
```

Um *chunk* com estas propriedades só existirá quando o modelo for iniciado, que será o *chunk* meta, cuja sua criação foi citada anteriormente. Sendo assim, se o modelo estiver iniciando, a produção iniciar irá combinar, pois haverá um *chunk* com as propriedades requisitadas pelo LHS.

Quando a produção é executada, ela faz uma alteração no *chunk* que está no *buffer goal*, atribuindo o valor da variável =num1 ao seu atributo *cont*. Após isso é feita uma solicitação através do *buffer retrieval* de um *chunk* do tipo *ordem-contagem* que tenha o atributo *primeiro* com valor igual à variável =num1.

```
=goal>
        cont    =num1
+retrieval>
        ISA     ordem-contagem
        primeiro =num1
```

De maneira mais simples, o RHS da produção faz com atribui o valor do atributo *inicio* do *chunk* no *buffer goal* ao atributo *cont* deste mesmo *buffer* e adiciona no *buffer retrieval* um *chunk* do tipo *ordem-contagem* que possua seu atributo *primeiro* igual a este valor.

4.2.4.2. A Produção *Incrementar*

Esta produção fará o modelo avançar na contagem de um número pra seu sucessor.

```
=goal>
        ISA     contar-de
        cont    =num1
        - fim   =num1
=retrieval>
        ISA     ordem-contagem
        primeiro =num1
        segundo =num2
```

Seu LHS faz dois testes. Primeiro testa se há um *chunk* no *buffer goal* do tipo *contar-de*, atribuindo o valor de seu atributo *cont* à variável =num1 e testando se seu atributo *fim* não é igual à variável =num1. O sinal "-" em um teste de atributo do *chunk* em um *buffer* significa negação; logo, esta linha do código está dizendo que se fim não for igual a variável =num1, o resultado é verdadeiro. Após isto, há um teste que verifica se há no *buffer retrieval* um *chunk* do tipo *ordem-contagem*, com valor de seu atributo *primeiro* igual a variável =num1 e atribui o valor de seu atributo *segundo* à variável =num2.

Se estes testes resultarem todos em verdadeiro, o RHS é processado.

```
=goal>
        cont    =num2
+retrieval>
        ISA     ordem-contagem
        primeiro =num2
!output!(=num1)
```

Também composto de duas ações, esta produção, ao ser executada, altera o valor do atributo cont do *chunk* existente no *buffer goal* para o valor da variável =num2. Então, faz uma solicitação através do *buffer retrieval* de um *chunk* do tipo *ordem-contagem* com o atributo *primeiro* igual ao valor da variável =num2 e finaliza a produção imprimindo no traço do modelo o valor da variável =num1.

Resumindo, a produção pega um *chunk* que tenha o valor do atributo *primeiro* igual ao atual número que está sendo contado (atributo *cont* do *chunk* no *buffer goal*),

copia o valor do atributo *segundo* pra o atributo *cont* do *chunk* no *buffer goal*, retorna um *chunk* no *buffer retrieval* que seja o sucessor do atual número, ou seja, que tenha o atributo *primeiro* igual ao número contado, e imprime no traço do modelo o número que foi contado.

4.2.4.3. A Produção *Parar*

Esta é a produção que identifica quando a contagem chegou ao ponto de parar e finalizar a execução do modelo.

```
=goal>
        ISA     contar-de
        cont    =num
        fim     =num
```

Sua parte de teste verifica se o *buffer goal* possui um *chunk* do tipo *contar-de* e o valor de seus atributos *cont* e *fim* são iguais. Caso isso seja verdadeiro, significa que a contagem (controlada pelo atributo cont) chegou ao seu fim (determinado pelo atributo *fim*). Isto é feito atribuindo o valor do atributo *cont* à variável =num e testando se o atributo *fim* é igual a essa variável. Ocorrendo isto, o RHS da produção fará a limpeza do *buffer goal* (por isso o sinal "-" antes de seu nome), não havendo mais nenhum *chunk* de meta para o modelo, tornando-o finalizado com sucesso em seu objetivo; por fim, fará a impressão do valor da variável =num (que é o valor final da contagem) e o modelo será encerrado automaticamente quando o sistema central de produção não encontrar nenhuma produção que execute.

```
-goal>
!output!(=num)
```

4.2.3. O Fim do Código

Para que o modelo tenha um objetivo, um *chunk* do tipo contar-de é inserido manualmente (pelo programador) no *buffer goal*, através da seguinte linha de comando:

```
(goal-focus meta)
```

Isto faz com que o *chunk* meta esteja carregado no *buffer* já no início da execução do modelo, permitindo que a produção *iniciar* combine e seja disparada, e assim o modelo atinja seu objetivo.

4.2.4. O Traço do Modelo

Quando o modelo Contador.lisp é executado (clicando no botão *Run* do painel de controle ou entrando com o comando *run* da entrada da janela *Listener*) até completar seu objetivo (contar de 2 até 4), o traço de eventos gerado é exatamente como mostra a Figura 22.

```
     0.000   GOAL                SET-BUFFER-CHUNK GOAL META REQUESTED NIL
     0.000   PROCEDURAL          CONFLICT-RESOLUTION
     0.000   PROCEDURAL          PRODUCTION-SELECTED INICIAR
     0.000   PROCEDURAL          BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL          PRODUCTION-FIRED INICIAR
     0.050   PROCEDURAL          MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL          MODULE-REQUEST RETRIEVAL
     0.050   PROCEDURAL          CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE         START-RETRIEVAL
     0.050   PROCEDURAL          CONFLICT-RESOLUTION
     0.100   DECLARATIVE         RETRIEVED-CHUNK C
     0.100   DECLARATIVE         SET-BUFFER-CHUNK RETRIEVAL C
     0.100   PROCEDURAL          CONFLICT-RESOLUTION
     0.100   PROCEDURAL          PRODUCTION-SELECTED INCREMENTAR
     0.100   PROCEDURAL          BUFFER-READ-ACTION GOAL
     0.100   PROCEDURAL          BUFFER-READ-ACTION RETRIEVAL
     0.150   PROCEDURAL          PRODUCTION-FIRED INCREMENTAR
2
     0.150   PROCEDURAL          MOD-BUFFER-CHUNK GOAL
     0.150   PROCEDURAL          MODULE-REQUEST RETRIEVAL
     0.150   PROCEDURAL          CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE         START-RETRIEVAL
     0.150   PROCEDURAL          CONFLICT-RESOLUTION
     0.200   DECLARATIVE         RETRIEVED-CHUNK D
     0.200   DECLARATIVE         SET-BUFFER-CHUNK RETRIEVAL D
     0.200   PROCEDURAL          CONFLICT-RESOLUTION
     0.200   PROCEDURAL          PRODUCTION-SELECTED INCREMENTAR
     0.200   PROCEDURAL          BUFFER-READ-ACTION GOAL
     0.200   PROCEDURAL          BUFFER-READ-ACTION RETRIEVAL
     0.250   PROCEDURAL          PRODUCTION-FIRED INCREMENTAR
3
     0.250   PROCEDURAL          MOD-BUFFER-CHUNK GOAL
     0.250   PROCEDURAL          MODULE-REQUEST RETRIEVAL
     0.250   PROCEDURAL          CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE         START-RETRIEVAL
     0.250   PROCEDURAL          CONFLICT-RESOLUTION
     0.250   PROCEDURAL          PRODUCTION-SELECTED PARAR
     0.250   PROCEDURAL          BUFFER-READ-ACTION GOAL
     0.300   DECLARATIVE         RETRIEVED-CHUNK E
     0.300   DECLARATIVE         SET-BUFFER-CHUNK RETRIEVAL E
     0.300   PROCEDURAL          PRODUCTION-FIRED PARAR
4
     0.300   PROCEDURAL          CLEAR-BUFFER GOAL
     0.300   PROCEDURAL          CONFLICT-RESOLUTION
     0.300   ------              Stopped because no events left to process
```

Figura 22. Traço do modelo Contador.lisp

À esquerda do traço, nos tempos 0.150 e 0.250 após o disparo da produção incrementar (evento *production-fired incrementar*), e 0.300 após o disparo da produção parar (evento *production-fired parar*), podem ser visualizados os números 2, 3 e 4, respectivamente. Estes são os números contados pelo modelo. Note que a produção incrementar foi executada duas vezes e a produção parar uma vez, todas antes do número contado impresso no traço. As linhas que contém o evento *conflict-resolution* (resolução de conflito) são os momentos em que o sistema de produção do modelo procurou por produções disponíveis e *chunks* que combinassem com os

requerimentos naquele momento. Ao final, a resolução de conflito não encontra mais produções para ser disparada e considera o modelo finalizado, parando a execução.

Na Figura 23 é demonstrada uma outra forma de análise do traço do modelo.

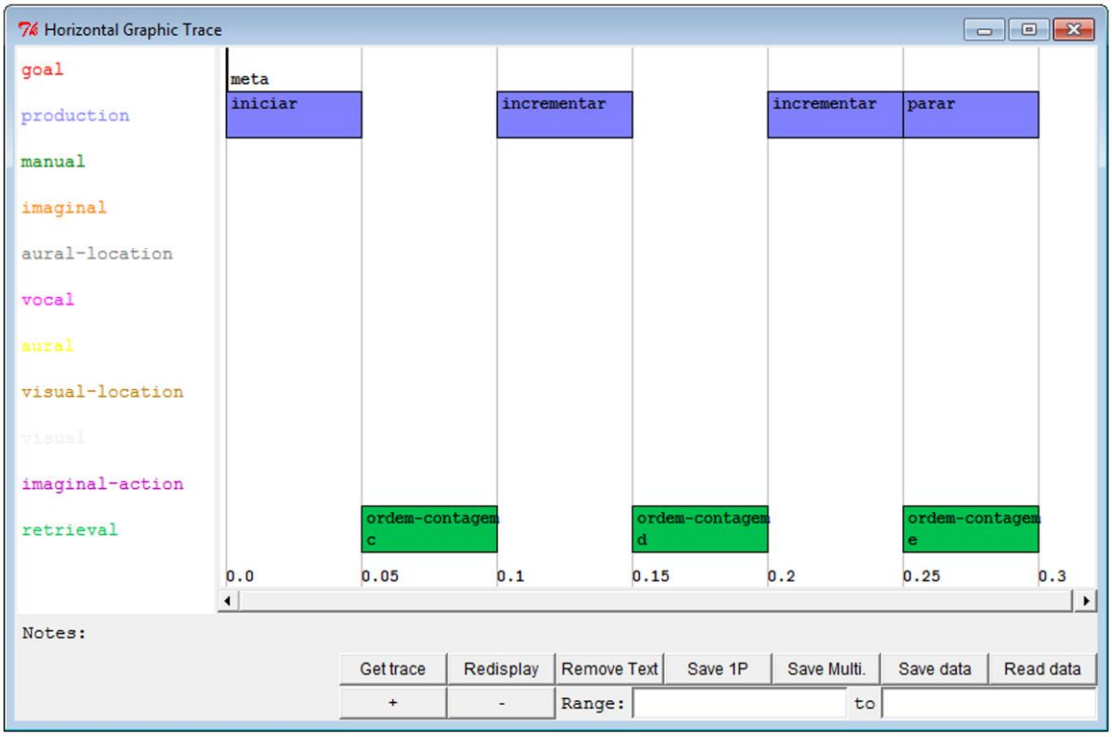

Figura 23. Ferramenta *Horiz. Buffer Trace* com o modelo Contador.lisp

Após o modelo ser executado, a ferramenta *Horiz. Buffer Trace* do ACT-R 6.0 mostra os *buffers* utilizados pelo modelo, o tempo em que isto ocorreu e os *chunks* envolvidos em cada um. Se o usuário clicar obre álbum bloco no gráfico, os detalhes da produção ou *chunk* referente aquele bloco será exibido em uma nova janela para análise.

4.3. O MODELO LER-DIGITAR.LISP

O segundo modelo apresentado neste trabalho simula o sistema de percepção e o sistema motor humano. O modelo faz a leitura de uma letra no campo visual, delimitado por uma pequena janela, e pressiona a tecla no teclado virtual correspondente aquela letra.

Diferente do primeiro modelo, este é acompanhado de um experimento, ou seja, um conjunto de códigos Lisp responsável por criar o cenário de atuação do modelo, como por exemplo, a janela que servirá de campo visual. Detalhes sobre isto serão abordados no decorrer da explicação do código, que é mostrado a seguir.

```
(defvar *resposta* nil)

(defmethod rpm-window-key-event-handler
        ((win rpm-window) key)
        (setf *resposta* (string key))
        (clear-exp-window)
        (when *actr-enabled-p*
                (proc-display)
        )
)

; Início do código da função experimento
(defun experimento ()
        (reset)
        (let*    (
                        (lis
                                (permute-list '("A" "B" "C" "D" "E" "F" "G" "H" "I"
                                                "J" "K" "L" "M" "N" "O" "P" "Q" "R"
                                                "S" "T" "U" "V" "W" "X" "Y" "Z")
                                )
                        )
                        (text1 (first lis))
                        (window (open-exp-window "Letter recognition"))
                )
                (add-text-to-exp-window :text text1 :x 125 :y 150)
                (setf *resposta* nil)
                (if *actr-enabled-p*
                        (progn
                                (install-device window)
                                (proc-display)
                                (run 10 :real-time t)
```

```
                    )
                    (while
                        (null *resposta*)
                        (allow-event-manager window)
                    )
                )
        *resposta*
        )
) ; Fim do código da função experimento

(clear-all)

; Início do código do modelo Ler-digitar.lisp
(define-model ler-digitar
        ; Configuração dos parâmetros do modelo
        (sgp
                :show-focus t
                :trace-detail high
        )

        ; Declaração dos tipos de chunk
        (chunk-type ler-letra estado)
        (chunk-type array letra)

        ; Criação do chunk
        (add-dm
                (meta ISA ler-letra estado “iniciar”)
        )

        ; Produções
        (P encontrar-letra
                =goal>
                        ISA     ler-letra
                        estado “iniciar”
        ==>
                +visual-location>
                        ISA     visual-location
                        :attended nil
                =goal>
                        estado “procurar”
        )
        (P atender-letra
                =goal>
                        ISA     ler-letra
                        estado “procurar”
                =visual-location>
                        ISA     visual-location
                ?visual>
                        state   free
        ==>
```

```
        +visual>
            ISA     move-attention
            screen-pos =visual-location
        =goal>
            estado “atender”
    )
    (P codificar-letra
        =goal>
            ISA     ler-letra
            estado “atender”
        =visual>
            ISA     text
            value   =letra
    ==>
        =goal>
            estado “responder”
        +imaginal>
            ISA     array
            letra   =letra
    )
    (P responder
        =goal>
            ISA     ler-letra
            estado “responder”
        =imaginal>
            ISA     array
            letra   =letra
        ?manual>
            state   free
    ==>
        =goal>
            estado “concluido”
        +manual>
            ISA     press-key
            key     =letra
    )

    (goal-focus meta)
    (setf *actr-enabled-p* t)
) ; Fim do código do modelo Ler-digitar.lisp
```

4.3.1. O Código da Função *Experimento*

Os detalhes de cada comando utilizado nesta parte do código não serão discutidos aqui, pois são comandos Lisp e este não é o foco do trabalho. Será apenas explicado o funcionamento desta função responsável por criar a janela do experimento e permitir com que o modelo trate-a como seu campo visual.

Primeiramente, é definido uma variável de nome **resposta**, que receberá a resposta da tarefa, ou seja, a tecla que foi pressionada, e retornará seu valor através da função *experimento*, mais adiante. O método *rpm-window-key-event-handler* define a janela do experimento e a deixa pronta para o experimento e o campo visual do modelo.

Já dentro do código da função *experimento*, iniciado por *defun*, o modelo é configurado para seu estado inicial, cria uma lista contendo todas as letras do alfabeto para serem sorteadas e uma delas lançadas na janela do experimento, configura a janela e a letra na posição 125 e 150 (coordenadas X e Y) e aguarda resposta, seja ela do modelo ou do usuário participante.

Para maiores informações sobre tal código, Seibel (2005) e Wangenheim (2007) descrevem com detalhes os comandos em Lisp.

4.3.2. O Início do Código do Modelo

Após executar o comando *clear-all* e iniciar a descrição do modelo Ler-digitar.lisp, os parâmetros são configurados utilizando o comando *sgp*, como no modelo anterior.

```
(sgp
        :show-focus t
        :trace-detail high
)
```

O parâmetro *:show-focus* determina se o foco da atenção visual do modelo será destacado quando ocorrer. Por exemplo, quando o experimento deste modelo for realizado, ao focar a visão na letra existente no campo visual, o módulo *vision* irá informar ao usuário para onde ele está apontando, circulando a letra lida. O valo padrão do parâmetro é *nil*, que significa que o foco não será exibido; porém neste modelo será configura para o valor para que faz com que o foco seja exibido, *t*.

O outro parâmetro, *:trace-detail*, como já foi abordado na seção anterior, determina o nível de detalhe do traço do modelo. Neste utilizaremos um alto nível de detalhes, como no anterior.

4.3.3. A Declaração dos *Chunk-types*

Neste modelo existirão dois tipos de *chunk*.

```
(chunk-type ler-letra estado)
(chunk-type array letra)
```

O tipo *ler-letra* possui apenas um atributo chamado *estado*, e servirá para guardar o atual estado do modelo, ou seja, o que ele deve fazer ou qual produção é mais apropriada a ser disparada no momento. O *buffer goal* carregará somente *chunks* deste tipo, mantendo controle da atual posição do modelo e permitindo o correto andamento do mesmo.

Já o outro tipo, *array*, também possui somente um atributo, de nome *letra*, e terá a função de guardar uma representação da letra que será vista pelo modelo. O *buffer imaginal* será responsável por gerar o *chunk* deste tipo para guardar a letra lida.

4.3.4. A Criação do *Chunk*

Como pode ser visto no código, apenas um *chunk* é predeterminado pelo programador neste modelo. Ele é chamado meta e é do tipo ler-letra e servirá para manter o *buffer goal* carregado, guardando informação do atual estado do modelo em sua execução.

```
(add-dm
        (meta ISA ler-letra estado “iniciar”)
)
```

Em sua criação é atribuído o valor “iniciar” ao seu atributo *estado*. Assim, o modelo já iniciará com uma produção disponível a ser disparada, a produção *encontrar-letra*.

4.3.5. As Produções

4.3.5.1. A Produção *Encontrar-letra*

Esta é a produção que faz o modelo iniciar o processo de leitura da letra no campo visual:

```
(P encontrar-letra
        =goal>
                ISA     ler-letra
                estado “iniciar”
==>
        +visual-location>
                ISA     visual-location
                :attended nil
        =goal>
                estado “procurar”
)
```

A parte de testes verifica se o *buffer goal* contém um *chunk* do tipo *ler-letra* e com valor do atributo *estado* igual a “iniciar”. Isto será verdadeiro, pois o *chunk* meta, criado inicialmente, foi colocado manualmente no *buffer* (através do comando *goal-focus* no final do código) para que o objetivo fosse traçado, e ele possui este valor de atributo.

Enfim, quando a produção é disparada, o *buffer visual-location* do módulo *vision*, faz uma solicitação de um novo *chunk* do tipo *visual-location* e valor de *:attended* igual a *nil*. Note que *chunks* deste tipo servem para guardar informações sobre a localização de um objeto no campo visual, como visto no capítulo anterior; logo, possuem a seguinte estrutura, os seguintes atributos (seus respectivos significados se encontram entre parênteses):

VISUAL-LOCATION
- SCREEN-X (posição no eixo X do objeto no campo visual)
- SCREEN-Y (posição no eixo Y do objeto no campo visual)
- DISTANCE (distância em que o objeto se encontra do foco visual)
- KIND (tipo do objeto)
- COLOR (cor do objeto)
- VALUE (valor do objeto)
- HEIGHT (altura do objeto)
- WIDTH (largura do objeto)
- SIZE (tamanho do objeto)

Agora, repare que na solicitação é mencionado o atributo *:attended*, mesmo ele não sendo um atributo válido para o tipo de *chunk visual-location*. Isso porque *:attended* não é um atributo e sim um parâmetro de pedido.

Parâmetros de pedidos são usados para suprir informação geral ao módulo sobre um pedido, uma solicitação, para evitar que um *chunk* não desejável seja carregado no *buffer*. Estes parâmetros são específicos para cada *buffer* em particular e sempre tem seu nome precedido de dois pontos “:”, o qual os diferenciam dos atributos do tipo do *chunk*.

No caso deste modelo, é solicitado um *chunk* do tipo *visual-location* que possua *:attended* igual a *nil*, ou seja, uma localização no campo visual que o modelo ainda não atendeu, a localização de um objeto que ainda não recebeu o foco de visão. Se o valor fosse *t*, seria solicitado um local que já foi atendido; se fosse *new*, seria

solicitado um local que recentemente recebeu um novo objeto e ainda não estava constado no módulo *vision*.

Quando a produção é disparada e o *buffer visual-location* recebe um *chunk* do módulo *vision* informando a localização de um objeto existente no campo visual que ainda não foi atendido (focado), o *chunk* no *buffer goal* tem seu atributo estado modificado para “procurar”, fazendo com que a próxima ação do modelo seja focar o objeto existente na localização retornada, prosseguindo com a execução.

4.3.5.2. A Produção *Atender-letra*

Esta produção fará o modelo focar sua atenção visual na localização adquirida na produção *encontrar-letra*.

```
(P atender-letra
        =goal>
                ISA     ler-letra
                estado “procurar”
        =visual-location>
                ISA     visual-location
        ?visual>
                state   free
==>
        +visual>
                ISA     move-attention
                screen-pos =visual-location
        =goal>
                estado “atender”
)
```

Como mostra o LHS desta produção, ela será executada quando o *buffer goal* possuir um *chunk* do tipo *ler-letra* e seu atributo *estado* tiver valor igual a “procurar”. Depois, no *buffer visual-location*, testerá existe um *chunk* do tipo *visual-location*. Também exige que o módulo *vision* esteja livre, como pode ser visto nas seguintes linhas:

```
?visual>
        state   free
```

Este tipo de teste é chamado de teste de estado de módulo. Neste caso, o *buffer* visual está perguntando ao seu módulo, o módulo *vision*, se ele está disponível para uma nova solicitação de *chunk*, ou seja, para codificar um objeto no campo visual. Os possíveis estados de módulo são *free*, *busy* ou *error*, que significa livre, ocupado ou erro, respectivamente. Caso o módulo esteja ocupado, a o sistema central de produção espera até que fique livre e a produção possa ser disparada. Este teste é recomendado, pois o módulo *vision* leva um tempo a mais para produzir resultado, gerando as representações dos objetos codificados, o que o torna mais lento que o sistema central de produção. Caso não seja testado seu estado e o modelo faça muitas requisições de leitura de objetos, uma seguida da outra, fará com que o módulo fique sobrecarregado, pois ele só é capaz de atender a uma solicitação por vez. Esta sobrecarga pode causar erro e tornar o modelo ineficiente.

Dando continuidade na produção, quando ela é disparada, uma solicitação de um novo *chunk* é feita ao módulo *vision* pelo *buffer* visual. Este *chunk* será do tipo *move-attention* e terá o atributo *screen-pos* igual ao valor da variável *=visual-location*, que atribui automaticamente o *chunk* presente no *buffer visual-location*. Uma solicitação do *buffer* visual de um *chunk* do tipo *move-attention* ao módulo *vision* fará com que o módulo direcione sua atenção (foco) até a posição especificada no atributo *screen-pos* e crie um *chunk* que represente o objeto existente naquele local, codificando suas características para o novo *chunk* no *buffer* visual. Isso tomará 85 milissegundos de execução, pois como já foi citado, o módulo leva um tempo para processar o objeto.

Após este processo, o *chunk* presente no *buffer* visual terá esta estrutura (os valores são apenas um exemplo de representação de uma letra “v” no campo visual):

```
TEXT0-0
        ISA             TEXT
        SCREEN-POS      VISUAL-LOCATION0-0-0
        VALUE           "v"
        STATUS          NIL
        COLOR           BLACK
        HEIGHT          10
        WIDTH           7
```

Note que, como no exemplo acima, o atributo *screen-pos* guarda um *chunk* do tipo *visual-location*, deixando explicito que os *buffers* visual e *visual-location* trabalham juntos para o módulo *vision* ter sucesso ao enxergar um objeto.

Por último, no RHS da produção, o *chunk* no *buffer goal* tem seu atributo estado modificado para “atender”, permitindo ao modelo dar o próximo passo no processo.

4.3.5.3. A Produção *Codificar-letra*

Esta produção fará a leitura do objeto focado pela produção anterior e o transformará em informações, criando um *chunk* na memória declarativa que represente o objeto codificado.

```
(P codificar-letra
        =goal>
                ISA     ler-letra
                estado “atender”
        =visual>
                ISA     text
                value   =letra
==>
        =goal>
                estado “responder”
        +imaginal>
                ISA     array
                letra   =letra
)
```

Primeiramente, a produção testa se o *buffer goal* possui um *chunk* do tipo *ler-letra* e atributo *estado* igual a “atender”. Depois, verifica se o *buffer* visual contém um *chunk*

que represente um objeto que é uma letra, ou seja, um *chunk* do tipo texto (*text*), e possua o atributo *value* diferente de *nil*, atribuindo seu valor na variável =letra.

No RHS, quando disparada modifica o atributo estado do *chunk* no *buffer goal* para “responder”. Além disso, faz uma solicitação de criação de *chunk* através do *buffer imaginal* do tipo *array* (criado no início do código) e atribui o valor da variável =letra ao seu atributo letra. Em outras palavras, caso o *buffer* visual esteja guardando um *chunk* que representa um texto (ou um caractere), o *buffer imaginal* irá solicitar a criação de um *chunk* do tipo *array* contendo o valor daquele texto. Isto faz com que a representação no sistema visual se torne memória declarativa, podendo ser manipulada por outros módulos além do *vision*.

4.3.5.4. A Produção *Responder*

Finalizando a seqüência de produções do modelo, esta faz com que o modelo faça uso do módulo motor e pressione a tecla no teclado virtual correspondente à letra identificada e codificada nas produções anteriores.

```
(P responder
        =goal>
                ISA     ler-letra
                estado “responder”
        =imaginal>
                ISA     array
                letra   =letra
        ?manual>
                state   free
==>
        =goal>
                estado “concluido”
        +manual>
                ISA     press-key
                key     =letra
)
```

Iniciando seu LHS, testa se o *buffer goal* possui um *chunk* do tipo *ler-letra* e valor do atributo *estado* igual a “responder”. Em seguida, testa se o *buffer imaginal* possui um

chunk do tipo *array*, atribuindo o valor de seu atributo *letra* à variável =letra. Como terceiro e último teste, verifica se o módulo motor está livre, fazendo um teste de estado através do *buffer* manual.

Então, no RHS da produção, o *chunk* no *buffer goal* tem o valor de seu atributo *estado* alterado para “concluido”. Por fim, o *buffer* manual faz uma solicitação de um novo *chunk* do tipo *press-key*, cujo valor do atributo *key* seja igual ao valor da variável =letra.

Uma solicitação do tipo *press-key* faz com que uma tecla, informada no atributo *key*, seja pressionada. Ela assume que as mãos do modelo estejam na posição inicial padrão para digitação, ou seja, com o dedo indicador esquerdo sobre a tecla “F” e o dedo indicador direito sobre a tecla “J”, com os demais dedos devidamente posicionados. Quando realizada, faz o dedo mais próximo e apropriado para a tecla solicitada mover-se até a tecla e pressioná-la, retornando à posição anterior logo em seguida, como se fosse o comando *Peck-recoil*. Ao realizar esta ação, o modelo terá atingido seu objetivo com sucesso, restando apenas a impressão da letra correspondente à tecla pressionada no fim do traço (ação realizada pela função *experimento*, como resultado de seu retorno).

4.3.6. O Final do Código do Modelo

Após a definição das produções, no código, é possível visualizar estes dois comandos:

```
(goal-focus meta)
(setf *actr-enabled-p* t)
```

O primeiro, como já foi citado no modelo Contador.lisp, serve para inserir o *chunk* de nome *meta* no *buffer goal*, tornando-o a informação base que o modelo irá seguir para atingir seu objetivo, guiando o modelo e informando o que fazer após cada passo.

O segundo, *setf*, é um comando está configurando a variável global **actr-enabled-p** para o valor *t*. Esta variável é usada para informar se o modelo vai ou não realizar a tarefa sozinho, ou seja, se ele irá agir por conta própria executando as produções, ou terá um participante para fazer isto, um usuário humano. Se o valor for igual a *t*, como usado neste modelo, ele irá realizar a tarefa e cumprir seu objetivo, gerando o traço mostrado na Figura 25. Caso o valor seja igual a *nil*, o modelo irá parar após a criação da janela com a letra para digitar e irá esperar que o usuário faça a tarefa por ele, ou seja, digite a letra que aparece na janela, gerando o traço por ele.

4.3.7. O Traço do Modelo

Ao executar o modelo utilizando o comando *run*, o traço será exatamente como mostrado na Figura 24. Desta forma o objetivo não é atingido pelo modelo. Isto ocorre porque não há um campo visual configurado para o modelo poder enxergar.

```
0.000   GOAL                 SET-BUFFER-CHUNK GOAL META REQUESTED NIL
     0.000   PROCEDURAL            CONFLICT-RESOLUTION
     0.000   PROCEDURAL            PRODUCTION-SELECTED ENCONTRAR-LETRA
     0.000   PROCEDURAL            BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL            PRODUCTION-FIRED ENCONTRAR-LETRA
     0.050   PROCEDURAL            MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL            MODULE-REQUEST VISUAL-LOCATION
     0.050   PROCEDURAL            CLEAR-BUFFER VISUAL-LOCATION
     0.050   VISION                Find-location
     0.050   VISION                FIND-LOC-FAILURE
     0.050   PROCEDURAL            CONFLICT-RESOLUTION
     0.050   ------                Stopped because no events left to process
```

Figura 24. Traço do modelo Ler-digitar.lisp ao executar o comando *run*

Por isso, para obter um resultado que atinja o objetivo do modelo, é necessário utilizar a chamada da função *experimento*, definida no mesmo arquivo do modelo, antes do código do mesmo. Executando a função, uma janela será criada, o campo de visão para o modelo será definido nesta janela, a letra a ser exibida será sorteada

e o modelo será executado, podendo assim disparar suas produções e atingir seu objetivo. O traço do modelo ao realizar o experimento fica parecido com o exibido na Figura 25.

```
     0.000   GOAL                 SET-BUFFER-CHUNK GOAL META REQUESTED NIL
     0.000   VISION               SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0 REQUESTED NIL
     0.000   PROCEDURAL           CONFLICT-RESOLUTION
     0.000   PROCEDURAL           PRODUCTION-SELECTED ENCONTRAR-LETRA
     0.000   PROCEDURAL           BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL           PRODUCTION-FIRED ENCONTRAR-LETRA
     0.050   PROCEDURAL           MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL           MODULE-REQUEST VISUAL-LOCATION
     0.050   PROCEDURAL           CLEAR-BUFFER VISUAL-LOCATION
     0.050   VISION               Find-location
     0.050   VISION               SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0
     0.050   PROCEDURAL           CONFLICT-RESOLUTION
     0.050   PROCEDURAL           PRODUCTION-SELECTED ATENDER-LETRA
     0.050   PROCEDURAL           BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL           BUFFER-READ-ACTION VISUAL-LOCATION
     0.050   PROCEDURAL           QUERY-BUFFER-ACTION VISUAL
     0.100   PROCEDURAL           PRODUCTION-FIRED ATENDER-LETRA
     0.100   PROCEDURAL           MOD-BUFFER-CHUNK GOAL
     0.100   PROCEDURAL           MODULE-REQUEST VISUAL
     0.100   PROCEDURAL           CLEAR-BUFFER VISUAL-LOCATION
     0.100   PROCEDURAL           CLEAR-BUFFER VISUAL
     0.100   VISION               Move-attention VISUAL-LOCATION0-0-1 NIL
     0.100   PROCEDURAL           CONFLICT-RESOLUTION
     0.185   VISION               Encoding-complete VISUAL-LOCATION0-0-1 NIL
     0.185   VISION               SET-BUFFER-CHUNK VISUAL TEXT1
     0.185   PROCEDURAL           CONFLICT-RESOLUTION
     0.185   PROCEDURAL           PRODUCTION-SELECTED CODIFICAR-LETRA
     0.185   PROCEDURAL           BUFFER-READ-ACTION GOAL
     0.185   PROCEDURAL           BUFFER-READ-ACTION VISUAL
     0.235   PROCEDURAL           PRODUCTION-FIRED CODIFICAR-LETRA
     0.235   PROCEDURAL           MOD-BUFFER-CHUNK GOAL
     0.235   PROCEDURAL           MODULE-REQUEST IMAGINAL
     0.235   PROCEDURAL           CLEAR-BUFFER VISUAL
     0.235   PROCEDURAL           CLEAR-BUFFER IMAGINAL
     0.235   PROCEDURAL           CONFLICT-RESOLUTION
0.435   IMAGINAL             CREATE-NEW-BUFFER-CHUNK IMAGINAL ISA ARRAY
     0.435   IMAGINAL             SET-BUFFER-CHUNK IMAGINAL ARRAY0
     0.435   PROCEDURAL           CONFLICT-RESOLUTION
     0.435   PROCEDURAL           PRODUCTION-SELECTED RESPONDER
     0.435   PROCEDURAL           BUFFER-READ-ACTION GOAL
     0.435   PROCEDURAL           BUFFER-READ-ACTION IMAGINAL
     0.435   PROCEDURAL           QUERY-BUFFER-ACTION MANUAL
     0.485   PROCEDURAL           PRODUCTION-FIRED RESPONDER
     0.485   PROCEDURAL           MOD-BUFFER-CHUNK GOAL
     0.485   PROCEDURAL           MODULE-REQUEST MANUAL
     0.485   PROCEDURAL           CLEAR-BUFFER IMAGINAL
     0.485   PROCEDURAL           CLEAR-BUFFER MANUAL
     0.485   MOTOR                PRESS-KEY y
     0.485   PROCEDURAL           CONFLICT-RESOLUTION
0.735   MOTOR                PREPARATION-COMPLETE
     0.735   PROCEDURAL           CONFLICT-RESOLUTION
     0.785   MOTOR                INITIATION-COMPLETE
     0.785   PROCEDURAL           CONFLICT-RESOLUTION
     0.885   MOTOR                OUTPUT-KEY #(6 3)
     0.885   PROCEDURAL           CONFLICT-RESOLUTION
     0.970   VISION               Encoding-complete VISUAL-LOCATION0-0-1 NIL
     0.970   VISION               No visual-object found
     0.970   PROCEDURAL           CONFLICT-RESOLUTION
     1.035   MOTOR                FINISH-MOVEMENT
     1.035   PROCEDURAL           CONFLICT-RESOLUTION
     1.035   ------               Stopped because no events left to process

"w"
```

Figura 25. Traço do modelo Ler-digitar.lisp ao executar a função *experimento*

Diferente do modelo Contador.lisp, este modelo não resultará sempre no mesmo traço, pois a letra a ser utilizada para ser exibida e lida nunca será a mesma, portanto, sempre haverá diferença nos eventos relacionados ao valor deste objeto. Note que o módulo motor leva 885 milissegundos para pressionar a tecla, desde a preparação do movimento, o início da ação, o pressionar e o retorno do dedo à posição anterior. Não é uma velocidade de um grande digitador, mas é uma velocidade razoável muito útil para diversos modelos.

5. CONCLUSÃO

Conclui-se com este trabalho que a arquitetura cognitiva ACT-R é realmente capaz de simular, de maneira clara, o funcionamento da cognição humana. Este fato está comprovado pela extensa lista de publicações produzida pelo grupo de pesquisa, multidisciplinar, do ACT-R. Ela demonstra a utilização dos sistemas cognitivo, perceptivo e motor de modo eficiente, fazendo o leitor aprender com facilidade as funções de sua estrutura, seus códigos e conceitos básicos. Se analisarmos cuidadosamente a maneira que realizamos uma contagem (como no modelo Contador.lisp) e pressionamos uma tecla percebida na tela (como no modelo Ler-digitar.lisp), veremos que as produções e o traço destes modelos descrevem um passo a passo do nosso subconsciente nos dizendo o que deve ser feito naquele momento, pois é exatamente assim que a memória procedural trabalha: nós fazemos, mas não sabemos como fazemos.

Observando os resultados deste trabalho, é possível imaginar inúmeras ideias para futuros trabalhos, como uma extensão deste estudo, fazendo maior utilização dos módulos, tratando detalhes que não foram citados aqui. Enfim, é possível criar um grande modelo, muito mais complexo, capaz de realizar uma grande e duradoura tarefa, mostrando, por exemplo, como agiria a cognição humana ao jogar um jogo de atenção no computador que exige, além dos olhos, as mãos para pressionar certas teclas agilmente para completar o objetivo. Ou até mesmo, estender o estudo do módulo motor para outros dispositivos, como por exemplo, o volante e o câmbio de um carro, gerando um campo visual que simula uma estrada com curvas e obstáculos, e fazendo as mãos do modelo realizarem movimentos de curva, simulando um volante virtual, e troca de marcha, simulando um câmbio virtual.

Espera-se que este trabalho sirva de inspiração e conteúdo para acadêmicos que desejam aceitar o desafio de estudar a Inteligência Artificial e a arquitetura cognitiva ACT-R, expandindo estas áreas de estudo e tornando-as mais influentes nas disciplinas de cursos de tecnologia.

REFERÊNCIAS BIBLIOGRÁFICAS

ACT-R. 1998. Department of Psychology – Carnegie Mellon University – Pittsburgh – Pensilvania – EUA. Disponível em: < http://act-r.psy.cmu.edu/>. Acesso em: 21 abr. 2009.

ANDERSON, John R.; BOTHELL, Daniel; BYRNE, Michael D.; DOUGLASS, Scott; LEBIERE, Christian; QIN, Yulin. **An Integrated Theory Of The Mind**. 2004. 25 p.

ANDERSON, John R.; BYRNE, Michael D. **Enhancing ACT-R's Perceptual-Motor Abilities**. Department of Psychology – Carnegie Mellon University – Pittsburgh – Pensilvania – EUA.

BEGOSSO, Luiz Carlos. **ACT-R, Atomic Components of Thought – Rational**. Notas de aula, 2006. 12 p.

BOTHELL, Daniel. **ACT-R 6.0 Reference Manual**. 2007a.

BOTHELL, Daniel. **ACT-R Environment Manual**. 2007b.

BOOSE, J. H. Personal Construct Theory and the Transfer of Human Expertese. **Proceedings of AAAI**. California, American Association for Artificial Inteligente, 1984.

CASTRO, Leandro de; ZUBEN, Fernando Von. **“Hill Climbing” e “Simulated Annealing”**. DCA / FEEC / Unicamp. Disponível em: <ftp://ftp.dca.fee.unicamp.br/pub/docs/vonzuben/ia707_02/topico1_02.pdf>. Acesso dia: 22 jun. 2009.

FEIGENBAUM, A. E. Expert Systems, Principles and Pratice. In: **The Encyclopedia of Computer Science and Engeneering**. 1992.

FERNANDES, Anita Maria da Rocha. **Inteligência Artificial – Noções Gerais**, 3. imp. / Anita Maria da Rocha Fernandes. Florianópolis: VisualBooks, 2005.

GANASCIA, J. G. **Inteligência Artificial**. São Paulo: Ática, 1993.

GATTASS, Ricardo. **Fundamentos da Ressonância Magnética Funcional**. UFRJ (Universidade Federal do Rio de Janeiro), 2000. Disponível em: <http://www.cerebromente.org.br/n13/tecnologia/ressonancia.htm>. Acesso em: 4 nov. 2009.

GURNEY, K. **An Introduction To Neural Networks**. University College London (UCL) Press. 1997. 234 p.

HOLLAND, J. **Adaptation in Natural and Artificial Systems**, University of Michigan Press, Ann Arbor, 1975.

JOHN, Bonnie E.; PREVAS, Konstantine; SALVUCCI, Dario D.; KOEDINGER, Ken. Predictive Human Performance Modeling Made Easy. **In:** ACM CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS, CHI 2004.

KIERAS, D. E.; MEYER, D. E. **The EPIC architecture: Principles of operation**, University of Michigan, Department of Electrical Engineering and Computer Science. 1996.

LEE, R. W. **Pesquisa Jurisprudencial Inteligente**. Florianópolis, 1998. Tese (Doutorado em Engenharia de Produção) – Centro Tecnológico, Universidade Federal de Santa Cantarina.

LIPPMANN, R. P. **Review Of Research On Neural Nets For Speech**. Neural Computation 1(1), 1989.

MCCULOCH, Warren; PITTS, Walter. **A Logical Calculus of Ideas Immanent in Nervous Activity**, 1943.

MITCHELL, M. **An Introduction to Genetic Algorithms**. MIT Press, 1996.

NEWELL, A. **The Knowledge Level**. Artificial Intelligence 18(1), 1982.

RICH, E.; KNIGHT, K.. **Inteligência Artificial** / Elaine Rich, Kevin Knight; tradução Maria Cláudia Santos Ribeiro Ratto; revisão técnica Alvaro Antunes. São Paulo: Makron Books, 1993.

SABBATINI, R. M. E. Uso do Computador no Apoio ao Diagnóstico Médico. **Revista Informédica**, 1(1): 5-11, 1993.

SEIBEL, Peter. **Practical Common Lisp**. Apress, 2005.

SIMONS, G. T. **Introdução a Inteligência Artificial**. Classe, 1998.

SIMÕES, Alexandre da Silva. **Inteligência Artificial – Aula 5: Busca com Informação**. UNESP – Sorocaba. Disponível em: <http://www.sorocaba.unesp.br/professor/assimoes/ia/ia_aula05.pdf>. Acesso em: 22 jun. 2009.

VIEIRA, Luis Eduardo. **Resolução de Problemas: Métodos de Busca**. 2006. CCEI (Centro de Ciências da Economia e Informática) / URCAMP – São Gabriel. Disponível em: <http://gabriel.sg.urcamp.tche.br/eduardo/IA/Aula_03_IA.pdf>. Acesso em: 21 jun. 2009.

WANGENHEIM, Aldo Von. **Métodos e Técnicas de Programação Funcional: Linguagem Lisp**. 2007. UFSC (Universidade Federal de Santa Catarina) – Florianópolis. Disponível em: <http://www.inf.ufsc.br/~func/lisp1.html>. Acesso em: 26 out. 2009.