Silvio Alexandre Porto

Planejamento em Redes de Tarefas Hierárquicas com Aplicação em Jogos

CURITIBA

SETEMBRO 2006

Silvio Alexandre Porto

Planejamento em Redes de Tarefas Hierárquicas com Aplicação em Jogos

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Marcos Alexandre Castilho

CURITIBA

SETEMBRO 2006

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Lista de Siglas

HTN	*Hierarchical Tasks Network*
UMCP	*Universal Method Composition Planner*
IA	Inteligência Artificial
NPCs	*Non-Player Characters*
FPS	*frames per second*
DOM	*Degree of Membership*
FLV	*Fuzzy Linguistic Variable*
FAM	*Fuzzy Association Matrix*
FFLL	*Free Fuzzy Logic Library*
PDDL	*Planning Domain Definition Language*
MTP	*Multi-Timeline Preprocessing*
FF	*Fast Forward*
MIPS	*Model-Checking Integrated Planning System*
OBDDs	*Ordered Binary Decision Diagrams*
STL	*Standard Template Library*
YACC	*Yet Another Compiler-Compiler*
CPU	*Computer Processing Unit*

Resumo

Neste trabalho nós investigamos a utilização de planejadores com a finalidade de equipar motores de inteligência artificial para jogos de computador. O maior problema neste tipo de aplicação de planejamento é o desempenho exigido pelos jogos. Nesta aplicação é necessário que os planos sejam compilados em tempos na ordem de milissegundos, o que não é possível de alcançar utilizando a maioria dos algoritmos de planejamento.

Com o objetivo de contextualizar o cenário em que nosso trabalho se insere, revisamos o estado atual da área de jogos mostrando as principais técnicas utilizadas atualmente para vencer os requisitos impostos pelos jogos aos seus desenvolvedores. Atualmente o mecanismo de decisão de alto nível, que seria o papel dos planejadores em jogos, em geral é preenchido por esquemas utilizando sistemas de regras e árvores de decisão para a escolha de planos pré-definidos.

Este trabalho propõe utilizar algoritmos baseados em redes de tarefas hierárquicas (HTN) (sigla em inglês), os quais nos parecem os mais adequados para esta aplicação. Em geral os algoritmos baseados em HTN são semi-automáticos. Estes algoritmos, embora dependam mais do conhecimento do especialista humano para produção do domínio do que os automáticos, são os mais adequados em termos de tempo de execução para o uso em jogos.

Dentre estes planejadores existe SHOP2, o qual é implementado neste trabalho na forma de uma biblioteca de *software* orientada a objetos. Esta biblioteca se diferencia de outras implementações de SHOP2 por atender a algumas necessidades particulares dos desenvolvedores de jogos, entre elas está a possibilidade de execução simultânea de vários domínios e problemas.

Palavras-chave: SHOP2, Planejamento, Jogos de Computador.

Abstract

We analyse SHOP2, a HTN based planner and implement it in terms of a object oriented software library. The difference betweem our implementation and other ones is to get some particular facilities to game developers, among them the possibility to allow the paralel execution of several instances of domains and problems.

In this work we investigate how planners can improve artificial engines for computer games. The main problem is the performance needed by the games. In this kind of application it is necessary to compile the plans in the order of milisecounds which seens to be difficult for the most know algorithms for planners.

We present an state of the art view of the games area, by showing the main techniques in use nowadays in order to improve computer games. In the present time the decision high level decision procedure, which should be the role of planners in games, is in general done by rule based systems schemmas or by decision trees aiming to find predefined plans.

In our work we propose to use algorithms based on hierarchical tasks network (HTN), which seen soon the most adequate for the application. In general planners for this kind of application are semi-automatic and, although they depend more of the knowledge of a human expert and less in relation with the automatic ones, they are the most adequate in terms of the runtime at games.

Key-words: SHOP2, Planning, Computer Games.

1 Introdução

Podemos conceituar inteligência artificial como a capacidade de uma máquina, munida de sistemas e programas de computação, imitar o comportamento humano e desempenhar funções normalmente associadas à inteligência humana, como a aprendizagem, a adaptação, a auto-correção e o poder de decisão.

A inteligência artificial atualmente é considerada pelos desenvolvedores de jogos como parte essencial de um jogo moderno. Mas nem sempre foi assim. No princípio o tempo destinado para o desenvolvimento desta era bem limitado e a inteligência não passava de umas poucas regras ou ações baseadas em estatísticas. Estas regras ou ações geralmente eram escritas diretamente no código do jogo. Desta forma não havia como adicionar mais inteligência ao programa depois de compilado. Também havia escassez de recursos para executar os algoritmos de inteligência artificial devido a cada quadro da animação ter um tempo limite para ser montado, o que obrigava os desenvolvedores a sacrificar a IA em favor dos gráficos.

Embora os desenvolvedores tenham usado técnicas melhores desde então, os agentes implementados em jogos são essencialmente reativos. Agentes reativos têm como características principais a facilidade em obter a próxima ação que deverá ser executada e o baixo custo de implementação e execução. Mas, entre os novos desafios, está tornar os agentes deliberativos, ou seja, inserir neles um mecanismo que lhes permita fazer uma análise mais complexa do mundo do que a feita pelos agentes reativos.

Este mecanismo geralmente consiste em um planejador (Seção 3). O agente equipado com um planejador é capaz de antecipar passos em vez de apenas reagir a estímulos com base no estado atual, acarretando que o agente demonstre ter consciência de seus objetivos e busque vencer suas metas.

Entre os planejadores existentes, os mais adequados para esta aplicação são os semi-automáticos baseados em HTN (*Hierarchical Tasks Network*). Como os planejadores HTN fazem reduções de planos mais complexos para um ou mais planos simples, isto

mantém a consistência do plano durante o processo de planejamento, o que não acontece no caso dos planejadores baseados em estado.

Planejadores baseados em HTN já foram usados com sucesso por Hawes (2003) em uma pesquisa envolvendo jogos. O pesquisador modificou o algoritmo UMCP (*Universal Method Composition Planner*) (EROL; HENDLER; NAU, 1994b) criando sua versão *anytime*, o A-UMCP (HAWES, 2003). Algoritmos *anytime* podem ser interrompidos a qualquer tempo e retornar a solução atual para o problema. A qualidade dessa solução deve ser monótona em relação ao tempo, ou seja, deve sempre melhorar conforme o tempo de processamento do problema cresce.

No planejamento HTN o planejador reduz tarefas com maior nível de abstração em um conjunto de zero ou mais tarefas menos abstratas, até o plano conter apenas tarefas primitivas (HAWES, 2003). Tarefas primitivas são tarefas que podem ser executadas diretamente pelo módulo de execução de planos. As tarefas compostas, ao contrário das primitivas, precisam ser reduzidas para um conjunto de tarefas primitivas para serem executadas (EROL; HENDLER; NAU, 1994a).

Um planejador semi-automático parece uma escolha sensata para ser usado em um módulo deliberativo de um agente concebido para esse tipo de aplicação. Estes agentes precisam tomar decisões muito rapidamente para não afetar a dinâmica do jogo. Existem planejadores desse tipo que fazem bons planos em um tempo finito e curto, um desses planejadores é o SHOP2 (NAU et al., 2001).

SHOP2 é um sistema de planejamento baseado em redes de tarefas hierárquica que se diferencia dos demais planejadores HTN por fazer planejamento para frente, ou seja, as ações são adicionadas ao plano na ordem em que serão executadas. Isto não acontecia em algoritmos como, por exemplo, o UMCP, o qual trabalhava reduzindo as tarefas e ordenando suas reduções de acordo com algumas restrições descritas no domínio. Isto era feito sem necessariamente fixar uma ordem para a seleção das tarefas a serem reduzidas.

Desta forma SHOP2 se tornou um algoritmo simples, principalmente por conhecer o estado atual em qualquer passo do processo de planejamento, o que permitiu um aumento considerável em seu poder de expressão. Assim foi possível incluir no sistema diversas funcionalidades, incluindo avaliação numérica e simbólica e chamada a funções,

procedimentos e programas externos.

O objetivo do nosso trabalho é implementar o algoritmo de planejamento SHOP2 (apresentado na Sessão 3.2) adaptando-o para que possa ser usado como parte do módulo deliberativo de um agente de jogo. Pelos testes realizados com a implementação de JSHOP2 (ILGHAMI, 2005), acreditamos que não é necessário transformar o algoritmo em *anytime* como Hawes (2003) fez com UMCP. Mas, para otimizações futuras, o fato de SHOP2 construir os planos adicionando os passos na ordem em que serão executados facilitaria o emprego desta técnica de especificação de algoritmos.

A implementação do algoritmo SHOP2 ocorrerá na forma de uma biblioteca orientada a objetos. Essa deve possibilitar a fácil gerência através do uso futuro de um escalonador de processos que venha a ser construído para gerenciar o tempo disponível para cada agente. Assim, o planejador deve ser capaz de salvar o estado atual para que este seja retomado em um momento posterior. A especificação ou instanciação de uma arquitetura de agentes está fora do escopo deste trabalho.

Para este propomos uma solução que viabilizará o uso de SHOP2 em jogos. Esta é uma biblioteca codificada em C++ que deverá ser ligada à aplicação (jogo). A biblioteca codificará os domínios e problemas em uma representação intermediária ao contrário do que acontece em JSHOP2 onde as descrições desses são compiladas em um programa executável.

Esta representação poderá ser carregada diretamente nas estruturas de dados do planejador e assim o processo poderá começar rapidamente. Segundo Ilghami (2005) uma grande vantagem da implementação do JSHOP2 é o sistema saber antecipadamente o tamanho necessário para otimizar o uso das estruturas de dados. Em nossa análise também é possível fazer o mesmo em tempo de execução do planejador devido aos domínios serem pré-codificados.

Existem duas implementações disponíveis de SHOP2, uma codificada em Java (JSHOP2) e uma em LISP, ambas de Ilghami (2005). JSHOP2 tem o inconveniente de exigir uma compilação de um código Java para cada domínio e para cada problema, o que inviabiliza sua utilização em um jogo, pois na maioria das vezes os problemas são formulados de forma parcial ou total em tempo de execução. Se este fosse adotado seria necessário que a cada novo problema houvesse uma chamada ao compilador Java para fazer o código para resolver aquele determinado problema. Por outro lado LISP está muito distante do paradigma imperativo, que predomina na programação de jogos, o que

cria dificuldades na integração do planejador com as outras estruturas do motor de IA (Seção 2.6).

O problema mais grave que encontramos na idéia de utilizar JSHOP2 em jogos foi que a aplicação não teria flexibilidade nenhuma para controlar o planejador e seria extremamente custosa a interação do planejador com os outros componentes do motor de IA. Desta forma seria muito difícil cumprir os requisitos relacionados ao tempo de execução, pois na implementação do agente o desenvolvedor não teria como parar os processos para dividi-los em cotas. Também seria custosa a compilação do problema e o retorno do plano devido às conversões que deveriam ser efetuadas.

Para nossa implementação foi escolhida a linguagem C++ que é a mais difundida entre os desenvolvedores de jogos. A maioria dos compiladores construídos para esta linguagem faz bibliotecas dinâmicas. Estas bibliotecas são acessíveis tanto em outros programas escritos em C++, quanto em programas escritos em outras linguagens, incluindo Java. Além disso, C++ é uma linguagem portável e existem implementações para as mais diversas plataformas.

Em nosso problema o planejador não é uma entidade isolada. Este faz parte de um motor de IA e deve poder interagir com outras técnicas tradicionalmente utilizadas para fazer inteligência para jogos. A essência da engenharia de um jogo é o equilíbrio de prioridades, ou seja, deve-se utilizar a ferramenta de IA certa para a atividade certa para obter um comportamento adequado com um tempo aceitável de processamento.

O texto está estruturado como segue. No Capítulo 2 é feita uma abordagem geral do problema de IA para jogos, mostrando seus objetivos e são apresentadas as principais ferramentas que atualmente são utilizadas para desenvolver inteligência nessas aplicações. O Capítulo 3 revisa o tema planejamento, orientando o foco para a busca das soluções que mais sejam viáveis em termos de tempo de processamento para o uso em jogos. No Capítulo 4 é descrita a estrutura de nossa implementação e no Capítulo 5 nossos experimentos e resultados. O Capítulo 6 conclui e apresenta propostas para trabalhos futuros.

2 Inteligência artificial em jogos

Dentre os objetivos da inteligência artificial (IA) está desenvolver programas de computador que tenham um comportamento inteligente semelhante ao do ser humano. Isto tem sido realizado em vários campos do conhecimento com relativo sucesso, incluindo o desenvolvimento de jogos de computador. Em especial, os jogos oferecem uma vantagem sobre outras aplicações que utilizam IA. As ações produzidas por esses programas não necessitam ser exatamente as mesmas que um jogador humano produziria, mas o comportamento gerado deve ser coerente sob o ponto de vista de um observador humano.

O objetivo de utilizar a inteligência artificial em jogos é propiciar diversão, assim não é necessário (nem desejado) que o computador acerte sempre. Se o agente sempre vencer, como no caso do *Deep Blue*[1], os jogadores poderiam perder a motivação por nunca ganharem da máquina e assim desistiriam do jogo rapidamente.

A grande maioria dos jogos de computador podem ser classificados como sistemas multi-agentes. Segundo Russell e Norvig (2003) um agente é "algo que percebe seu ambiente através de sensores e age sobre o mesmo através de atuadores". Estes agentes podem ser classificados quanto ao modo em que geram seu comportamento como reativos, deliberativos ou híbridos (HAWES, 2003). Os agentes desenvolvidos para jogos também são conhecidos como NPCs (*Non-Player Characters*) (BLY, 2004).

Jogos podem ser vistos como um meio não linear de representar uma história. Nos jogos, a história toma um novo rumo a cada ação do jogador, ao contrário dos filmes e livros onde a história segue um único fluxo que vai do seu início até o seu fim. Existem diversos tipos de jogos e cada tipo tem suas particularidades no modo em que deve ser implementado. Entre os tipos mais comuns de jogos estão os de simulação, tais como os de esportes, os de tabuleiro, os de estratégia e os de aventura (PEDERSEN, 2003).

Um fator determinante no projeto de um jogo é a jogabilidade. Esta pode ser vista como o grau de interatividade do jogo, isto é, o quanto o jogador está acoplado no mundo e

[1] Computador da IBM que venceu o campeão mundial de xadrez Kasparov

a forma com que esse responde ao usuário (ROUSE, 2001). Segundo Crawford (2003) existe uma escola entre os projetistas de jogos que enxerga o computador como uma ferramenta que oferece várias funções, entre elas, gráficos e IA. A mais importante destas funções é a interatividade que deve ser arquitetada de acordo com as necessidades de cada projeto por se tratar do principal requisito para a obtenção da jogabilidade. Assim o projetista deverá utilizar as outras funções de forma a maximizar a interatividade juntamente com a formulação de uma interface entre o jogador e o personagem, adequada ao tipo de jogo e à história.

Um exemplo de aplicação da jogabilidade é o paralelo entre um jogo de tabuleiro e um simulador. No jogo de tabuleiro o usuário faz um movimento e espera o computador fazer o dele. Desta forma se o computador demorar um minuto para fazer a jogada não fará diferença para a jogabilidade, pois jogadores humanos também param por alguns instantes para pensar no próximo movimento. Por outro lado em um simulador o computador não pode se dar ao luxo de parar para analisar por muito tempo, assim como no caso do piloto humano, o computador é obrigado a tomar decisões sob pressão e cabe ao desenvolvedor adotar esquemas de IA que viabilizem isto.

Nesse capítulo serão apresentados os desafios enfrentados pela indústria de jogos e as técnicas de IA utilizadas para tentar vencê-los. O maior objetivo, sob o ponto de vista de um desenvolvedor de jogos, é tentar fazer o jogo rodar, em um ritmo rápido e contínuo, em qualquer computador pessoal de uma determinada configuração ou superior.

A Seção 2.1 faz uma análise sobre este problema e as seções 2.4 e 2.5 demonstram as técnicas de nível de detalhe e balanceamento de carga, respectivamente. Estas técnicas, se aplicadas sabiamente no gerenciamento de agentes no motor de IA (Seção 2.6), podem fazer a diferença em tempo de execução. Existem duas abordagens para atualizar os agentes no motor de IA. Estas são tratadas na Seção 2.3 e consistem em tratamento de eventos ou a execução de um laço onde são atualizados todos os agentes, essa técnica é denominada *pooling* (RABIN, 2000a).

Cada tipo de problema tem suas particularidades e o motor de IA deve implementar técnicas que resolvam os problemas da melhor maneira possível. Esta maneira não precisa nem ser a mais sofisticada nem a mais complexa, mas a mais acertada para obter uma boa razão entre o custo de utilização da técnica e o benefício oferecido por esta para o agente. Algumas vezes é possível utilizar artifícios que façam o agente "trapacear", isto é detalhado na Seção 2.2.

Entre as técnicas mais comuns estão as máquinas de estado (Seção 2.7.3) e a

lógica *fuzzy* (Seção 2.7.2). Os motores de IA mais modernos oferecem também a técnica de planejamento (Capitulo 3), isto é útil para implementar agentes deliberativos (HAWES, 2003). Estes agentes, ao contrário dos reativos, fazem planos que buscam prever passos futuros que serão necessários para atingir determinado objetivo.

As técnicas detalhadas na Seção 2.7 ou citadas neste trabalho consistem em apenas uma pequena parcela das técnicas utilizadas pelos desenvolvedores de jogos. A maior parte destas técnicas e seus fins estão amplamente discutidos na literatura referenciada neste trabalho.

2.1 Restrições no tempo de execução

Até o surgimento das placas gráficas aceleradoras o processamento da IA teve menor prioridade do que o gráfico, pois os aspectos visuais da animação influenciam muito mais na jogabilidade do que a inteligência dos adversários. Esta inteligência em geral era gerada com o uso de técnicas de IA simples que eram melhoradas com o uso de trapaça (Seção 2.2).

O motivo para tão pouco investimento em termos de tempo de processamento no motor de IA era devido ao motor gráfico influenciar mais na jogabilidade do que o de inteligência artificial. Com a renovação constante dos computadores domésticos, substituídos por máquinas cada vez mais velozes e que contam com placas de vídeo com aceleração gráfica surgiu a oportunidade de investir mais na inteligência dos jogos.

A idéia da animação em tempo real é mostrar quadros na tela a uma taxa tal que transmita ao usuário uma noção de continuidade em suas interações com o mundo. Se a velocidade de apresentação dos gráficos cair muito serão apresentados vários quadros de imagem individuais que farão o jogador se aborrecer rapidamente com o jogo.

A taxa com que as imagens são geradas é medida em quadros por segundo, ou *frames per second*(FPS) (sigla em inglês). Uma aplicação gerando quadros começa a ser considerada interativa quando opera a uma taxa de 6 FPS, mas ainda é muito lenta para ser considerada em tempo real. Uma boa faixa de velocidade para gráficos em tempo real está entre 15 e 25 quadros por segundo, sendo que após 72 FPS a diferença entre os quadros é imperceptível, criando muita redundância (AKENINE-MÖLLER; HAINES, 2002).

Tradicionalmente os jogos de computador dividem o tempo de processamento em cotas de tamanho fixo referentes a cada componente do jogo. Podemos citar, entre

outras componentes a entrada de dados, o cômputo da física, o sombreamento dos gráficos e a inteligência artificial. Destas componentes as mais pesadas, do ponto de vista do processamento em tempo de simulação, são o sombreamento de imagens, o cômputo da física e o processamento da IA.

A quantidade de recursos computacionais dedicados a cada uma destas componentes é definida de acordo com a quantidade mínima exigida por elas e sua contribuição para o bom andamento do jogo.

Os gráficos, por exemplo, tem prioridade alta e geralmente é fixada uma taxa mínima de quadros por segundo que devem ser exibidos. A idéia por trás da divisão do tempo é manter um compromisso entre o ritmo do jogo e a jogabilidade. Durante o tempo destinado à inteligência artificial o motor de IA deverá ser executado e os vários agentes devem ser atualizados. O tempo destinado ao cômputo da IA geralmente está entre 10% e 20% do tempo total de processamento disponível para o jogo. Este intervalo foi encontrado de forma empírica através de resultados práticos durante o desenvolvimento da maioria dos jogos (BLY, 2004).

Alguns jogos mais sofisticados definem apenas quantas vezes cada componente deve ser atualizado em um segundo, com exceção do sombreamento, que é executado por quantas vezes a máquina conseguir no restante do tempo (BUCKLAND, 2005).

Cada componente, dentro de sua cota de tempo, deverá ser explorada pelo jogo de forma a priorizar a atualização dos elementos mais importantes do mundo no ponto de vista do usuário. Para isto é interessante a construção de um motor de IA com a finalidade de agrupar ferramentas e gerenciar recursos. A construção de motores para jogos já é uma prática bastante difundida, principalmente na área gráfica.

O motor de IA deve implementar o nível de detalhe (Seção 2.4). Este permite, com base em heurísticas, definir qual é a prioridade de cada entidade do mundo para definir quanto tempo cada uma terá para ser processada, e define a forma na qual serão executadas. Assim, por exemplo, se a heurística for a distância entre o jogador e o agente inimigo, então inimigos mais longe terão menor prioridade para usar a cota de processamento.

É desejável que o motor de IA implemente balanceamento de carga (Seção 2.5). Isto é necessário para distribuir a execução das cotas dos diversos componentes entre os vários quadros que serão sombreados dentro de uma fatia de tempo. Um exemplo seria um jogo gerando uma animação com a velocidade de 20 FPS. Se simplesmente a IA fosse

atualizada dentro de 3 dos 20 quadros, o tempo de compilação destes quadros retardaria a exibição desses e faria com que houvessem "socos" na animação tirando sua continuidade.

Outro fator importante para obter uma animação de ótima qualidade é a construção de um jogo bem estruturado. O jogo é responsável por combinar e gerenciar os motores específicos, assim como o motor gráfico e o de IA.

Dentre as responsabilidades do jogo estão a criação dos objetos baseados em tipos especificados em tempo de desenvolvimento e modificação de suas propriedades conforme ocorrem operações no mundo. Estas operações podem ser tanto efetuadas pelo jogador quanto pelos agentes artificiais e devem estar dentro das regras do mundo. Em geral os tipos destes objetos são especializações de tipos definidos na elaboração dos motores.

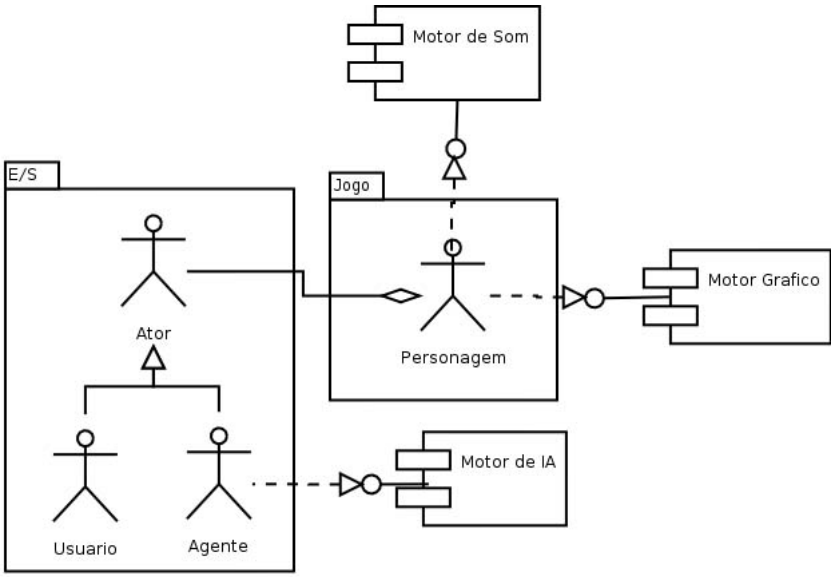

Figura 1: Motores.

A Figura 1 ilustra um exemplo de implementação hipotética de um jogo utilizando três motores. Estes motores são o de IA, o gráfico e o de som. Cada um destes fornece serviços para os outros módulos do sistema através de *interfaces* construídas sob a forma de classes abstratas. No módulo de entrada e saída existe a classe "Agente" que implementa o agente fornecido pelo motor de IA. Como cada instância da classe "Personagem" implementada no jogo tem um "Ator", então se esse for um robô o motor de IA formulará as decisões que o agente deverá executar para modificar o mundo à seu favor. A classe "Personagem" cria uma representação física para o mesmo, por isso também interage com os motores gráficos e de som. Mais detalhes sobre a representação física do agente podem ser encontrados em Bly (2004).

2.2 Trapaça

Muitas vezes o desenvolvedor poderá "trapacear" usando uma série de artifícios que dão ao usuário a falsa impressão de que os agentes são oponentes espertos. Isto é útil para simplificar a simulação, e assim economizar recursos.

Um exemplo de trapaça é o cálculo da navegação dentro de um ambiente 3D. Esta é uma atividade caríssima que não pode ser efetuada facilmente em tempo de execução, por isto os desenvolvedores preferem fazê-la em tempo de projeto. Desta forma, em tempo de execução, a navegação é feita através de pontos e linhas pré-calculados, invisíveis ao usuário, que guiam o agente entre os pontos desejados (SURASMITH, 2002).

Outra tática que ajuda a diminuir a sobrecarga no processamento é colocar a inteligência no mundo e não nos algoritmos de IA. Esta abordagem foi utilizada no jogo *The SIMS*[2]. Neste jogo os objetos são especificados de forma que possam instruir os agentes sobre o seu funcionamento. Devido a estas instruções os agentes não precisam conhecer o funcionamento dos outros agentes e objetos. Para haver a interação entre os objetos e agentes basta utilizar algumas trocas de mensagens entre eles. Um exemplo disso pode ser um agente faminto. De posse da informação de fome, a geladeira mais próxima irá instruí-lo da maneira de abri-la e se houver um bolo dentro dela esse deverá instruir o personagem do modo que ele deve proceder para comê-lo.

Segundo Laird e Lent (2000), os jogos que utilizam uma abordagem exclusivamente reativa, como *DOOM II*[3], equilibram a eficiência de seus agentes através da superioridade numérica e parâmetros de resistência maiores para os atores em comparação com os do jogador humano equivalente ao agente.

Buckland (2005) relata um caso de teste onde o jogo *Halo*[4] foi submetido a duas turmas de testadores. Uma delas recebeu uma versão com os valores definidos pelos projetistas como bons, mas a maior parte dos jogadores achou a inteligência fraca. À outra turma foi submetida uma outra versão do jogo cuja única diferença era os valores de resistência aumentados em relação a versão original. Os jogadores que utilizaram a segunda versão apontaram a inteligência dessa como muito boa, pois era difícil vencer os agentes.

[2] *Copyright Maxis*, 2004
[3] *Copyright ID Software*, 1994
[4] *Copyright Microsoft Corporation*, 2003

2.3 Eventos *versus pooling*

A cada ciclo de simulação, cada objeto necessita verificar sua condição no mundo e gerar as ações convenientes e executá-las. Existem dois modos de um objeto fazer isto. Estas formas podem ser por *pooling* ou dirigida por eventos (RABIN, 2000a). Na primeira abordagem é necessário que a cada ciclo todos os objetos atualizem sua lógica. Com a direção por eventos os objetos precisam se responsabilizar por receber os eventos gerados por outros objetos e enviados a ele e executar as ações pertinentes.

Em geral, dado o número de objetos envolvidos em um ambiente 3D, é conveniente implementar um sistema de troca de mensagens. Isto ocorre porque quando um objeto é afetado por um evento gerado por outro objeto, ele é avisado do mesmo e não é necessário verificar a ocorrência do evento e seu envolvimento no fato. Simplesmente o objeto que gera o evento verifica quais são os objetos envolvidos e trata de avisá-los. Um exemplo de aplicação é o acionamento do método "explodir" de um objeto "bomba", assim os objetos que estão em seu raio de ação serão avisados da explosão e deverão se comportar como é apropriado. Desta forma o problema é reduzido, nesse caso, para o cálculo da distância de todos os objetos contra o objeto que representa o artefato.

Se este mesmo sistema fosse implementado usando *pooling*, a cada ciclo seria necessário que todos os objetos buscassem por consequências para si vindas das ações de todos os outros objetos. Desta forma o manipulador teria de checar todo o estado do mundo, a fim de verificar se algum objeto executou alguma modificação no mesmo e se esta influenciou outros objetos.

Basicamente uma mensagem é um objeto que contém cinco campos: o nome da mensagem, o nome do remetente, o nome do destinatário, o horário em que deve ser entregue e a informação relevante para o agente referente a esta mensagem.

Além de otimizar o uso dos recursos de uma forma muito mais eficiente do que é possível com *pooling*, ainda é possível salvar todas as mensagens em um arquivo. Isto é uma ótima ferramenta de depuração, dado que é possível verificar todas as interações entre os agentes e o ambiente. Se houver um erro será fácil verificar qual agente engatilhou o evento que gerou a condição de erro.

A Figura 2 ilustra um exemplo de sistema de troca de mensagens. Neste sistema o roteador de mensagens recebe eventos enviados pelos dispositivos de entrada usados pelo jogador, juntamente com os eventos gerados por mudanças no mundo que criaram interações entre objetos. Dentre estas mudanças no mundo estão as colisões geradas pela

física e eventos gerados por relógios ou qualquer outro objeto ou agente. Os objetos do mundo recebem estes eventos em um determinado instante de tempo especificado em seu cabeçalho e processam utilizando as técnicas de IA disponíveis, neste caso a máquina de estados. Baseado em suas decisões, o agente envia mensagens para os objetos afetados por suas ações, e assim por diante até não haver mais efeitos que influenciem outros objetos.

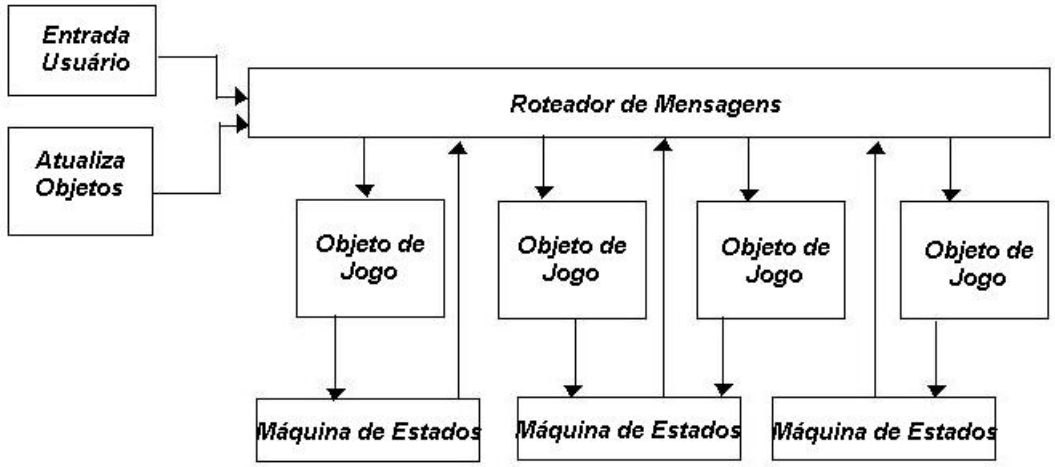

Figura 2: Exemplo de Roteamento de Mensagens.
Fonte: (RABIN, 2000a).

Dentre as criaturas e objetos do mundo real, em geral há tempos de reação diferentes. Para simular tal fato os sistemas de trocas de mensagens permitem que se agende um horário de entrega para cada mensagem. Se este horário for a hora atual ou inferior, a mensagem é entregue no ato; senão, ela vai para um limbo onde espera o horário em que deve ser recebida pelo objeto destinatário.

Existem duas utilidades para a entrega agendada de mensagens. Isto simula uma característica presente em todos os objetos e criaturas do mundo real, o tempo de retardo entre a percepção de uma ação que mudou o mundo e sua reação a ela. O agendamento das mensagens também pode ser usado para criar um espaçamento entre as ações do agente. O agente também pode enviar para si mesmo mensagens especificando as ações que deve executar, mas com o tempo agendado com uma data aleatória. Desta forma o agente vai fazendo as coisas que deve fazer em intervalos variáveis, adicionando caos e tornando o agente mais próximo das criaturas do mundo real.

2.4 Nível de detalhe

Devido ao tempo de processamento ser escasso e existirem muitos agentes para serem executados são necessários mecanismos para assegurar que esse recurso seja bem gasto.

Um destes mecanismos é o nível de detalhe. Esta técnica é aplicada de forma parecida com a que ocorre na computação gráfica. Isto consiste em uma heurística que relaciona o agente ao ponto de interesse no mundo. O ponto de interesse geralmente é relacionado à visão que o usuário tem do mundo, ou seja, objetos escondidos não precisam ser exibidos e nem animados. A função heurística define se um agente deve ganhar tempo de processamento para ser atualizado e, em algumas vezes, quanto tempo deverá ser disponibilizado para executar o processo de atualização.

Esta heurística pode ser feita com base em certos parâmetros. Alguns destes são a distância entre o agente e o jogador, a presença ou não do agente na cena, se o agente foi atualizado ou mesmo se esteve na cena nas últimas iterações, se o personagem morreu ou não é essencial para a cena. Caso o personagem não seja essencial para a cena ele pode ser atualizado somente quando há sobras de recursos de computação.

A distância entre o jogador e o agente é a mais simples, mas também poderosa heurística se usada em conjunto com outras. É muito interessante poder descartar certo agente se ele está fora do campo de ação do jogador. Mas também pode ser interessante que agentes em posição estratégica interajam com o jogador. Assim cada função heurística deve ser construída levando em conta as regras do jogo específico para a qual será desenvolvida.

Um exemplo de função heurística para resolver este problema é inicialmente definir áreas circulares em volta do jogador estabelecendo assim o ponto de interesse. Considerando um agente de um jogo em que o jogador é colocado em primeira pessoa, ou seja, ele tem a visão correspondente a de um personagem do jogo. Este agente terá maior prioridade se estiver dentro do ângulo de visão do usuário ou atrás dele, mas perto. Se o agente estiver dentro do disco formado pelas áreas circulares ele deve ser considerado, e quanto mais interna for a trilha do disco em que o agente está contido, maior será sua prioridade. Quanto maior for a prioridade do agente, maior será o tempo alocado para sua execução.

2.5 Balanceamento de carga

Há um grande esforço direcionado para encaixar técnicas de IA com qualidade para jogos na pequena cota destinada ao processamento desta. Mas além do requisito de compilação de vários quadros de imagem por segundo de execução, estes quadros devem formar uma animação contínua e o usuário não deve perceber grandes mudanças nas imagens de quadros consecutivos.

A execução da IA em um jogo de tempo real deve ser dividida em ciclos correspondentes à execução do laço que faz a atualização dos agentes. É neste laço que deve ser implementado o balanceamento de carga. Este laço é dividido em duas partes, ou seja, a IA que necessita ser recalculada a cada quadro de animação, ou IA constante, e a IA que necessita ser recalculada eventualmente.

Dentre os algoritmos que precisam ser processados a cada quadro de animação estão os de animação do agente e os de "busca ao alvo". O primeiro destes algoritmos executa animações especificadas em tempo de projeto, ajustando-as com a ajuda do segundo. O segundo algoritmo faz pequenos ajustes, como por exemplo, mirar outro agente com a arma. Apenas os agentes mais importantes, de acordo com as heurísticas como a do nível de detalhe, pode se dar ao luxo de gastar tempo de processamento fazendo atualizações constantes precisas, e consequentemente complexas, de seu posicionamento. A maioria apenas tem sua posição modificada através de cálculos simples.

A IA eventual não necessita ser chamada a cada quadro de animação. São exemplos desta a visão e audição do agente, bem como planejamento de ações e outras técnicas de decisão. Existem alguns efeitos colaterais positivos e negativos para esta abordagem. Entre os positivos está que um atraso da IA para detetar certa mudança no mundo, o que pode ser contado como um tempo de reação normal para as criaturas do mundo real. Dentre os negativos está o agente não detetar que uma condição do mundo mudou, como por exemplo, o inimigo já morreu, e continuar fazendo determinada ação que no caso seria atirar. Mesmo assim este efeito colateral negativo não é tão grave neste caso do atirador, pois os atiradores do mundo real quando estão em uma situação de *stress* continuam atirando mesmo depois do oponente tombar. Mas em alguns casos este efeito negativo pode trazer consequências desagradáveis. Por isto em tempo de projeto deve ser definido, caso a caso, qual das abordagens usar, se será usada a abordagem da IA atualizada eventualmente ou a atualizada a cada quadro de animação.

Frequentemente os algoritmos de IA calculados a cada quadro utilizam informações

compiladas pelos algoritmos que executam apenas eventualmente, ajustando-as para serem aplicadas no mundo.

Seria extremamente desejável que o jogo pudesse distribuir as cotas de processamento de todas as componentes (Seção 2.1) dentro do tempo de construção de cada quadro ao invés de distribuir estas cotas dentro de um segundo. Isto é necessário para garantir que os quadros sejam exibidos na tela do usuário a uma taxa constante e desta forma evitando "socos" na animação que tiram sua continuidade.

Distribuir as cotas desta maneira é uma tarefa bastante difícil, o que pode ser feito é distribuir a carga de uma forma aproximada, fazendo assim a variação da taxa construção de quadros oscilar muito pouco. Uma abordagem para realizar isto é ajustar a carga sob demanda. O ajuste da carga sob demanda pode ser feito através da distribuição dos agentes para cada quadro de animação de acordo com o poder de processamento da máquina e da política de distribuição de tempo do jogo. Estes agentes terão sua IA eventual atualizada de tempos em tempos, mas a parte constante de sua IA será atualizada a cada quadro.

O meio mais simples de fazer a distribuição é processar apenas alguns agentes por quadro de animação, sendo que este número de agentes varia de acordo com alguns requisitos. Um destes requisitos é a jogabilidade. Como somente alguns agentes estarão sendo atualizados por vez, o intervalo de tempo entre as atualizações influencia diretamente na jogabilidade. Por isto é importante restringir a variação no tamanho do intervalo de tempo entre as atualizações de determinados agentes para valores que não afetem aqueles requisitos.

Preferencialmente o sistema deve escolher os agentes com maior influência no contexto em que o jogador está. Para isto pode ser utilizada a técnica de nível de detalhe (Seção 2.4). A escolha correta dos agentes que devem ter maior oferta de processamento tende a dar maior qualidade à simulação.

Alguns cuidados devem ser tomados na implementação do esquema de balanceamento de carga. Para isto deve ser assegurado que nenhum agente inicie um ciclo infinito de execução. Também é importante verificar se nenhum agente está sendo atualizado mais de uma vez por ciclo. Isto poderia acontecer se o número de agentes processados por ciclo fosse maior do que o número de agentes presentes no mundo. Em geral, executar as atualizações duas vezes seguidas em um mesmo agente não é uma falta grave, aliás, em alguns casos isto é desejável, mas executar múltiplas atualizações em objetos que estão fora da área de atuação do jogador é desperdício de processamento.

O Quadro 1 mostra um esquema simples de balanceamento de carga. Neste, a cada ciclo, é executada toda a IA constante e depois é executada a parte periódica da IA em um número determinado de agentes. Este laço é equivalente ao utilizado tradicionalmente (Quadro 2) quando o número de agentes no mundo for igual ao número de agentes por ciclo de animação.

Se implementado adequadamente, este esquema pode realizar os seus dois maiores objetivos que são distribuir o trabalho entre os quadros de animação e ao mesmo tempo compilar um comportamento interessante para os vários agentes.

```
1   void Agent::UpdateAI(){
2                   pAgent = m_agents->begin();
3                   //Processa IA constante
4                   while (pAgent){
5                             pAgent->ProcessConstantAI();
6                             pAgent++;
7                   }
8                   //Processa IA eventual
9                   while (m_agents_processed < m_agents_frame){
10                            m_agent_ciclic->ProcessPeriodicIA();
11                            m_agent_ciclic++;
12                            m_agents_processed++;
13                  }
14  }
```

Quadro 1: Exemplo de laço de IA com carga balanceada.

```
1   Agent::UpdateAI(){
2                   pAgent = m_agents->begin();
3                   while (pAgent){
4                             pAgent->ProcessAI();
5                             pAgent++;
6                   }
7   }
```

Quadro 2: Exemplo de laço de IA tradicional.

Existem esquemas mais complexos como os apresentados em Alexander (2002). Este autor publicou uma arquitetura baseada em distribuição de carga em seu artigo. Nesta arquitetura o sistema agenda processos de IA, tais como, planejamento, busca de caminhos, máquinas de estado, entre outros. Estes processos são colocados em espera na forma de tarefas e essas são executadas conforme haja tempo de processamento disponível.

Desta forma para cada tarefa é dado um prazo máximo de espera por recursos computacionais. Conforme os prazos das tarefas se esgotam, se ainda não foram executadas, estas ganham prioridade sobre as que ainda estão dentro do prazo estipulado. Desta forma é evitada a ocorrência de tarefas que esperam indefinidamente por recursos para executar.

2.6 Motor de IA

Alguns pesquisadores, como Funge (1998) esperam que os avanços em *hardware* tragam para a IA aplicada em jogos os mesmos frutos que a aceleração gráfica trouxe para os gráficos. Mas infelizmente isso não tem se provado verdadeiro. Alguns problemas de IA têm se mostrado mais relacionados a *software* do que a *hardware*, então estes não seriam resolvidos simplesmente aumentando a capacidade de processamento.

Desta forma a lógica de cada jogo deve ser construída com as ferramentas de IA que melhor se adequem aos subproblemas que devem ser resolvidos, assim a escolha da ferramenta correta para cada subproblema específico é a chave para assegurar o bom desempenho do sistema por inteiro.

Akenine-Möller e Haines (2002) definem o motor gráfico como uma coleção de técnicas não organizadas que consistem em algo mais complexo do que apenas uma camada responsável por gerenciar o sombreamento de triângulos. O motor também deve oferecer facilidades, como por exemplo, o agrupamento destes triângulos em malhas e das malhas em modelos. Estas entidades que são representadas por objetos na memória podem facilmente receber transformações físicas e geométricas de acordo com a simulação. Assim o motor pode ser visto como a aplicação dos princípios da orientação a objetos.

Em geral esta definição é generalizada e o conceito de motor é também aplicado à parte de inteligência artificial do jogo. Desta forma as entidades "inteligentes" do jogo são mapeadas para agentes e conforme seja o problema a ser resolvido é necessário escolher a técnica mais apropriada para sua solução. A escolha das técnicas é definida na própria lógica do jogo, na especificação de cada agente, pelo desenvolvedor utilizando a *interface* de classes do motor.

No motor de IA também existe o ambiente dos agentes. A função deste é gerenciar os agentes que determinam e executar as ações necessárias para modificar o mundo conforme as decisões de um cada deles dentro da cota de tempo destinada à inteligência artificial. Os ambientes para agentes artificiais geralmente são construídos levando em consideração uma arquitetura multi-agente (BLY, 2004).

Algumas das técnicas que devem ser implementadas em um motor de inteligência artificial são explicadas na Seção 2.7 e outras são citadas na mesma e são amplamente abordadas na literatura referenciada neste texto.

Entre as propriedades desejáveis de um motor de IA estão a facilidade em permitir

a comunicação entre os objetos do jogo, possibilitar uma forma de especificar o comportamento da IA de uma forma clara e concisa e assim facilitar a depuração da aplicação. Assim os motores são bibliotecas de *software* cuja grande vantagem é a diminuição da necessidade de refazer código já escrito.

Existem vários tipos de jogos tipificados de acordo com suas características. Dentre esses tipos estão, os de tabuleiro, os de corrida, os de "atirador em primeira pessoa" e os de esportes. Cada jogo que pertence a uma determinada categoria apresenta subproblemas parecidos com os problemas dos demais jogos de seu tipo. Isto torna possível para o desenvolvedor instanciar o motor de IA, adaptando-o aos problemas apresentados pelo jogo a ser desenvolvido.

Hoje o problema maior que impede a implementação das técnicas de IA mais sofisticadas é o pouco tempo destinado ao desenvolvimento destas dentro do cronograma do projeto do jogo. As empresas desenvolvedoras tendem a reservar alguns poucos meses no final do cronograma para esta atividade e muitas vezes é necessário deixar muitos requisitos de lado, priorizando o lançamento do produto em uma determinada data.

Atualmente, com a grande concorrência no setor, as produtoras se encontram em um dilema entre o corte na qualidade ou aumento do tempo e custos de desenvolvimento. Uma solução para o problema é a elaboração de um motor de IA ou o licenciamento de um motor de IA de outra empresa. Desta forma um mesmo código poderá servir como base para vários programas.

2.7 Técnicas de IA utilizadas em jogos

Para que a implementação do motor de IA seja útil, as técnicas que fazem parte deste devem ser implementadas de uma forma que se permita a interação entre elas. Além do que, também é necessário que sejam escaláveis, ou seja, possam trabalhar tanto com problemas simples e pequenos quanto com grandes e complexos.

Dentre as técnicas implementadas nos motores de IA estão as máquinas de estado (Seção 2.7.3), sistemas de regras (CHRISTIAN, 2002), árvores de decisão (BUCKLAND, 2005), vida artificial (WOODCOCK, 2000), robótica (movimento) (LAMOTHE, 1999), redes neurais (LAMOTHE, 2000), lógica *fuzzy* (Seção 2.7.2) e planejamento (Capítulo 3).

Todas estas técnicas já foram implementadas em jogos com relativo sucesso, mas somente as mais simples estão presentes na maioria dos jogos. As técnicas mais imple-

mentadas, e que serão detalhadas nesta seção, são as máquinas de estado, os sistemas de regras e a lógica *fuzzy*.

Devido aos agentes de jogos em geral tomarem muitas decisões sobre dados numéricos, tais como, distâncias entre objetos, valores destes, probabilidades, decisões através de variáveis nebulosas e avaliação de recursos baseada na teoria dos jogos é desejável que as técnicas utilizadas nas tomadas de decisão possam avaliar expressões. Por outro lado estas técnicas devem poder diferenciar os objetos do mundo e os contextos ao qual se inserem, assim também é necessária a capacidade de avaliação simbólica dos modelos.

A maioria dos jogos atuais utiliza apenas agentes reativos e uma pequena minoria tem algum mecanismo de predição. Quando são utilizados agentes reativos as decisões desses dependem apenas do estado atual, ao contrário dos deliberativos que procuram antecipar passos futuros do agente fazendo planos. A especificação de agentes deliberativos é possível através do uso de um planejador. Planejadores são abordados no Capítulo 3.

Afim de ilustrar a aplicação dos conceitos explicados nesta seção e o uso de planejamento para auxiliar o agente de jogos na tomada de decisão (Capitulo 5), será tomado como exemplo um motorista automático.

Basicamente um motorista automático deve se comportar como um motorista do mundo real, ou seja, deve conduzir o seu carro de um ponto "A" para um ponto "B" utilizando o menor tempo possível para vencer o percurso, evitando receber multas e colidir o veículo com os obstáculos.

Em geral jogos de simulação de transito utilizam rotas pré-calculadas, isto consiste em um dos tipos mais comuns de "trapaças" usadas pelos desenvolvedores de jogos (Seção 2.2). Assim, em tempo de execução as rotas possíveis estão à disposição do agente que utiliza estas como se houvessem "trilhos" invisíveis indicando por onde é possível navegar pelo terreno e assim passar com o veículo (SURASMITH, 2002).

Infelizmente há um efeito colateral neste caso, ou seja, perde-se a noção de realidade pelos carros aparentarem estar andando sobre trilhos ou fazer curvas bruscas (PINTER, 2002). Para corrigir isto existem técnicas que permitem fazer correções de rota a fim do agente não dirigir exatamente em cima do trilho, mas adicionando algum grau de variação na posição do carro com relação às bordas da pista.

Embora a navegação sobre o terreno seja uma parte bastante importante na implementação do motorista, ela é somente a camada de baixo nível da inteligência do

agente e não será discutida aqui. Os algoritmos e estruturas de dados utilizados pela IA para especificar a parte do controle de navegação de veículos em um ambiente é abordada com certa profundidade em Biasillo (2002) e Adzima (2002) em seus artigos.

Existe a necessidade de um controle em mais alto nível. Esta se dá pois não é suficiente que os carros andem sobre os trilhos, mesmo de forma realística, sem considerar o contexto em que se encontram. Os motoristas em um simulador de trafego enfrentam congestionamentos, são obrigados a fazer ultrapassagens de veículos mais lentos, necessitam desviar de veículos acidentados e vencer outros desafios que fazem parte do cotidiano dos motoristas reais.

2.7.1 Sistema de regras

Sistemas de regras, também conhecidos como sistemas de regras de produção consistem em uma das formas mais simples e diretas de representar conhecimento em um agente para jogos.

O sistema geralmente é dividido em três partes: a memória de trabalho, a memória de regras e a lógica do sistema. A memória de trabalho contém os fatos que consistem no conhecimento armazenado no sistema. As regras são aplicadas na base de conhecimento a fim desta gerar mais conhecimento. A lógica do sistema é responsável pela inferência, assim esta utiliza os fatos da memória de trabalho para raciocinar sobre as regras e escolher qual deverá ser executada. As regras costumam ter a seguinte forma:

$$if \quad (premissa) \quad then \quad (consequente)$$

Onde a premissa é uma expressão booleana que pode ser verdadeira ou falsa. Se a premissa for verdadeira, o consequente é considerado verdadeiro. Assim se existirem os fatos que validam a premissa, a regra poderá ser executada e o consequente alterará os fatos na memória de trabalho (JONES, 2003).

O conjunto de fatos na memória de trabalho também é conhecido como base de conhecimento. Na maior parte dos problemas os fatos podem ser representados pela lógica convencional, mas algumas vezes o raciocínio requer que sejam considerados vários graus de verdade, assim é possível utilizar lógica *fuzzy* (Seção 2.7.2) para representar os fatos. Utilizar a lógica *fuzzy* é menos eficiente do que utilizar a lógica convencional, então é adequado que o sistema possa trabalhar com as duas formas de representação e

usar cada uma de acordo com as necessidades do sub-problema que está sendo resolvido (CHAMPANDARD, 2003).

O mecanismo que aplica a lógica para selecionar as regras pode ser implementado de várias formas. Os métodos variam em complexidade e vão desde regras escritas diretamente no código do jogo utilizando as construções da linguagem de programação até o uso de uma máquina de inferência como o Prolog.

Durante a execução do mecanismo, o sistema busca pela ação na base de regras tentando casar as condições de cada regra com a situação atual do mundo. Se houver este casamento a ação é executada. Como apenas um comportamento pode ser selecionado e executado para cada regra, são evitados efeitos colaterais gerados pela interação entre estas.

Embora um sistema de regras ofereça bons resultados para domínios restritos, conforme esses crescem, surgem problemas sérios de escalabilidade devido ao número de regras. Os maiores problemas são a complexidade crescente de depuração do sistema e o desempenho degradado durante o tempo de execução devido ao número de condições que devem ser testadas para a escolha do comportamento correspondente à situação (HAWES, 2003).

Há um consenso entre os desenvolvedores de que devem manter os dados do programa separados da lógica que os trabalha. Isto porque se perde muita flexibilidade por estes dados estarem estáticos e imutáveis dentro do código compilado (RABIN, 2000b). Assim é recomendável que as regras sejam especificadas dentro de arquivos que possam ser lidos pelo jogo e colocados em uma estrutura de dados na memória do computador para serem utilizados pela lógica do sistema de regras (CHAMPANDARD, 2003).

Para efetuar a busca por regras existem, basicamente, duas abordagens que podem ser usadas em um sistema de regras de produção, ou seja, com encadeamento para frente ou para trás (BOURG; SEEMAN, 2004).

O encadeamento para trás inicia o processamento tendo o estado objetivo como hipótese e tenta encontrar as regras cujo consequente "case" com esse estado e a condição da regra é consultada para verificar a validade dessa. A partir daí estabelece uma outra hipótese e este processo é repetido recursivamente pelas regras até que seja atingido o estado inicial. Caso um caminho não leve até este estado o sistema faz reversão (*backtracking*) e tenta um outro caminho alternativo.

A inferência por encadeamento para frente é feita inicialmente sobre os fatos

conhecidos sobre o mundo. As regras são consultadas para que sejam selecionadas aquelas que cuja condição "case" com estes fatos, as quais poderão incluir novos fatos na base de conhecimento. Destas regras é escolhida uma através de um critério de desempate utilizado com a finalidade de resolver conflitos entre várias regras que são aplicáveis a partir daqueles fatos. Selecionada a regra, essa é aplicada e o processo reinicia e prossegue até que seja encontrado o estado final ou novos fatos não possam ser derivados da base de conhecimento.

O sistema de regras de produção como visto acima é uma das formas mais simples e comuns de efetuar planejamento em jogos. Mas isto é bom para planos pequenos e simples, pois para problemas complexos de planejamento é mais adequado usar um planejador como um daqueles vistos no Capítulo 3. Isto se dá devido ao custo da busca exaustiva usada no sistema de regras crescer demais em problemas complexos. Assim, muitas vezes é mais barato utilizar um planejador semi-automático para resolver os mesmos problemas.

Caso o sistema encontre duas ou mais regras aplicáveis é necessário um critério de desempate, para isso é assumida alguma heurística. Neste caso a heurística pode variar em complexidade. Pode ser tanto algo complexo como descobrir o número de antecedentes (ou consequentes) da regra, ou mesmo pegar a primeira regra aplicável que for encontrada (JONES, 2003).

O exemplo do Quadro 3 mostra uma das formas com que as regras podem ser escritas em um sistema de regras de produção simples. Neste sistema as regras são implementadas através de objetos de uma classe que são criados através do emprego de quatro macros. A macro "if_rule_begin" inicia um bloco onde está representada uma regra e recebe como parâmetro o nome desta. Esse bloco termina com o comando "if_rule_end" que recebe um objeto da classe "TIFAIRule" onde deverá ser atribuída a regra. As macros "if_rule_antecedent" e "if_rule_consequent" declaram, respectivamente, a premissa e o consequente da regra. Estes, no exemplo, também recebem um valor que pode ser falso ou verdadeiro. Em aplicações do mundo real é possível estender este sistema usando, por exemplo, variáveis em lógica de primeira ordem e *fuzzy*.

A maioria das regras de um sistema real requer também que sejam especificadas ações que mudem o mundo físico ao invés de apenas modificar o conhecimento do sistema. Isto pode ser alcançado através da ampliação das funcionalidades da macro "if_rule_consequent" a fim dela ter um código associado que execute algo como por exemplo uma animação.

As regras no exemplo do Quadro 3 estão escritas no próprio código do programa

```
1   class TIFAIRule{
2         private:
3         public:
4                 TIFAIRule(char * rulename){}
5                 void Antecedent(char * rule_assert){}
6                 void Consequent(char * rule_assert){}
7   };
8
9   #define if_rule_begin(rule_name)\
10          {\
11                TIFAIRule * tmp = new TIFAIRule(#rule_name);
12
13  #define if_rule_antecedent(rule_assert, val) \
14                tmp->Antecedent(#rule_assert);
15
16  #define if_rule_consequent(rule_assert, val) \
17                tmp->Consequent(#rule_assert);
18
19  #define if_rule_end(addr_rule) \
20                addr_rule = tmp; \
21          }
22
23  void Sistema_de_regras(){
24          TIFAIRule * test, * test2, * test3;
25
26          if_rule_begin (liga_carro);
27               if_rule_antecedent(tem_chave, true);
28               if_rule_antecedent(motorista_esta_sentado, true);
29               if_rule_antecedent(tem_combustivel, true);
30
31               if_rule_consequent(motor_funcionando, true);
32          if_rule_end(test);
33
34          if_rule_begin (desliga_carro);
35                if_rule_antecedent(tem_chave, true);
36                if_rule_antecedent(motorista_esta_sentado, true);
37                if_rule_antecedent(motor_funcionando, true);
38
39                if_rule_consequent(motor_funcionando, false);
40          if_rule_end(test2);
41
42          if_rule_begin (acabou_combustivel);
43                if_rule_antecedent(have_gas, false);
44                if_rule_antecedent(motor_funcionando, true);
45
46                if_rule_consequent(motor_funcionando, false);
47          if_rule_end(test3);
48  }
```

Quadro 3: Exemplo de sistema de com 3 regras.

e tratam o funcionamento do motor de um carro. Este pode ser ligado ou desligado pelo agente e deve ser desligado automaticamente no caso de término do combustível. Para isto o sistema verifica e altera os fatos no mundo através das macros responsáveis por manipular as premissas e os consequentes.

2.7.2 Lógica *fuzzy*

A lógica *fuzzy* é amplamente utilizada em jogos para resolver problemas que não poderiam ser resolvidos diretamente pela lógica proposicional ou pela de lógica de primeira ordem. Estes problemas geralmente tratam da decisão sobre variáveis não exatas que oferecem vários graus de verdade.

A lógica convencional permite a verificação da existência ou não de um objeto dentro de um determinado conjunto, bem como fazer operações sobre este conjunto ou operações deste com outros conjuntos. Desta forma esta lógica é muito apropriada para situações onde existam somente duas possibilidades, ou seja, está ou não está no con-

junto. Os conjuntos da lógica convencional também são conhecidos como conjuntos *crisp* (precisos). Mas para a inteligência de jogos é interessante existir a figura do "talvez".

A lógica *fuzzy*, ao contrário das outras duas lógicas, não se preocupa se o objeto está ou não no conjunto. Mas o foco desta lógica está no grau de pertinência do objeto em um determinado conjunto. Este grau de pertinência é conhecido como DOM *Degree of Membership* (sigla em inglês). Este é representado por um valor no intervalo entre zero e um. Isto permite ao desenvolvedor flexibilizar a fronteira entre estar dentro do conjunto ou estar fora deste.

A *fuzzyficação* é o processo de conversão dos valores numéricos de um conjunto *crisp* para os DOM de uma variável *fuzzy*.

Podem ser representados por variáveis *fuzzy* quantidades geralmente representadas por intervalos tais como temperatura e distância. Estas variáveis também são referenciadas como *Fuzzy Linguistic Variable* (FLV) (sigla em inglês). Entre os inúmeros exemplos de aplicações deste conceito estão as variáveis saúde do agente e munição.

Cada variável pode ter certo número de conjuntos associados. Um conjunto *fuzzy* é definido formalmente como:

$$A = (x, \mu_a(x) | x \in X)$$

Onde $\mu_a(x)$ (DOM) é a função que quantifica o grau de participação dos elementos x em um conjunto X. Um elemento mapeado para o valor 1 indica que esse está totalmente dentro do conjunto enquanto se o valor for 0 ele não é membro desse. Valores entre 0 e 1 caracterizam um elemento *fuzzy*.

Assim como na lógica convencional, a lógica *fuzzy* define algumas operações[5], entre essas estão a união, a interseção e o complemento. A união de dois conjuntos *fuzzy* é igual a aplicação da função matemática máximo em todos os elementos dos conjuntos. A interseção em *fuzzy* é feita da mesma forma que a união, mas aplicando a função mínimo em vez da máximo. O complemento é dado pela função $(1 - X)$, onde X é o valor da variável *fuzzy*.

A figura 3 (a) mostra um gráfico representando uma variável *fuzzy* com dois conjuntos, em (b) este mesmo gráfico aparece tendo sido efetuada uma operação de união e em (c) interseção. Na figura 3 (d) é mostrado o resultado da operação complemento.

[5]As operações tais como a união e a interseção possuem classes de funções denominadas *t-norma* e *t-conorma*. Aqui, para fins de simplificação, serão abordadas apenas as funções mais citadas na literatura referente a jogos.

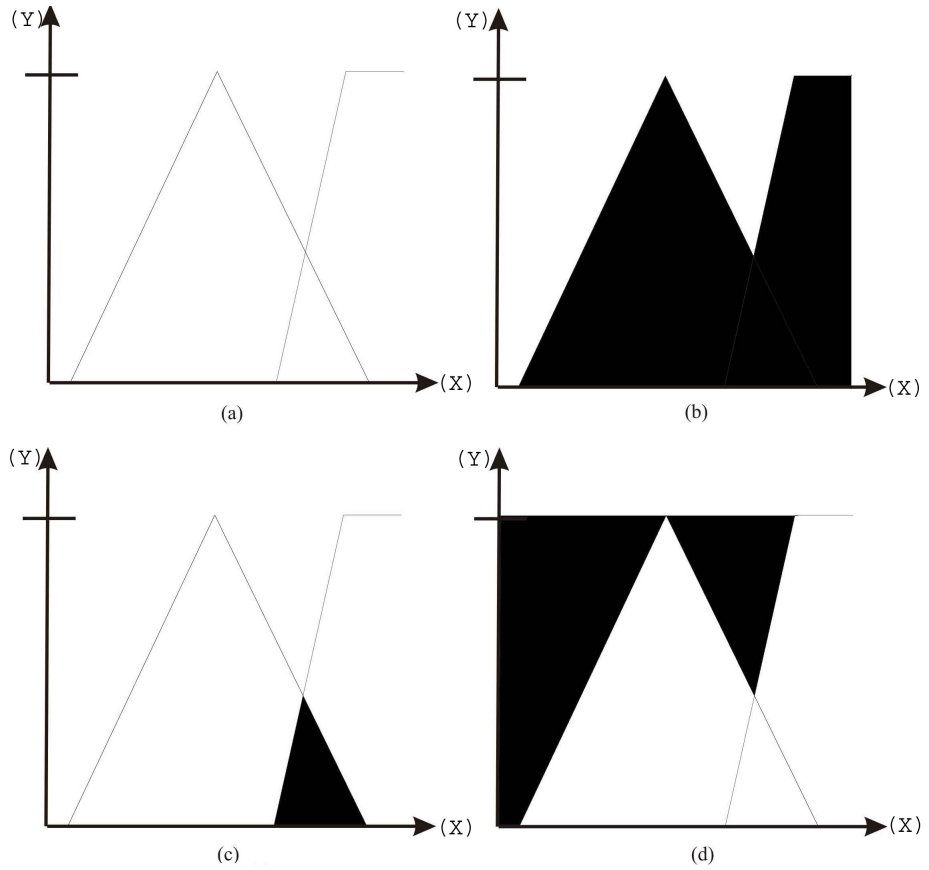

Figura 3: Exemplo de aplicação de operadores utilizando lógica *fuzzy*.

A inferência é um processo que verifica quais regras são válidas. A partir da seleção destas regras são combinados os DOM dos conjuntos de entrada que validam a regra e é criado um único DOM de saída. Estas regras também são conhecidas coletivamente como FAM *Fuzzy Association Matrix*. Um exemplo de aplicação destas regras pode ser visto no Quadro 5.

O número máximo de regras é dado pela equação s^v, onde s é o número de conjuntos e v o de variáveis. Este costuma ficar muito grande conforme o número de variáveis ou o de conjuntos cresce causando uma explosão combinatorial no tamanho da matriz FAM. Isto pode tornar o sistema *fuzzy* um tanto lento e difícil de manter. Mas existe uma técnica para contornar este problema chamada *Combs Method* que é explicada em detalhes em Zarozinski e Than (2001).

Através destas regras é feita também a *desfuzzyficação* que consiste em um processo que produz um resultado quantitativo na lógica *fuzzy*. Estas regras transformam um grupo de variáveis em um resultado.

Um dos métodos de *desfuzzyficação* mais simples é escolher o conjunto em que a variável tem maior valor de participação. O grande problema com este método é a perda de informação, pois as regras que dizem respeito aos outros conjuntos são desconsideradas.

Uma técnica útil de *desfuzzyficação* precisa considerar os resultados das regras de todos os conjuntos envolvidos de alguma maneira. A função mais comum para *desfuzzyficação* tem o gráfico de um triângulo. Este triângulo é recortado em linha reta em alguma parte entre seu topo e a sua base indicada pelo grau de pertinência do conjunto que ele representa. Deste triângulo, sem a parte de cima, resta um trapezóide. A *desfuzzyficação* agrupa os trapezóides do gráfico, onde a união das várias partes formam uma figura geométrica. Assim é calculado o centróide desta figura que é conhecido como centróide *fuzzy* e sua coordenada x é o valor do DOM de saída. Podem ser utilizadas outras figuras geométricas para fazer a *desfuzzyficação*, bastando adaptar o processo.

O processo de *desfuzzyficação* utiliza as mesmas regras da inferência para converter os conjuntos em valores lógicos. Assim estes valores podem ser utilizados em condicionais, que em última instância efetuam a decisão que foi avaliada.

O grande problema da lógica *fuzzy* está na complexidade de sua implementação. Mas existem várias opções de bibliotecas prontas, inclusive otimizadas para ter maior velocidade de processamento. Entre as bibliotecas que tem esta otimização, que é a mais desejável para jogos, está a *Free Fuzzy Logic Library* (FFLL) (sigla em inglês). Esta biblioteca tem licença *open source* BSD que permite inclusive que a biblioteca seja utilizada em programas comerciais de código fechado (ZAROZINSKI, 2002).

Para ilustrar o funcionamento do controle por lógica *fuzzy* será dado prosseguimento ao desenvolvimento do motorista automático apresentado na Seção 2.7. Muitos problemas relacionados a trânsito e a motoristas podem ser modelados através desta técnica. Entre esses estão à personalidade e as habilidades do motorista, controle do consumo de combustível do carro e decisão se pára, ou passa, ao se deparar com semáforos.

Em especial, nesta seção será tratado o problema do tráfego urbano, onde os carros andam próximos uns dos outros, evitando colisões na traseira do veículo imediatamente à sua frente e mantendo uma distância razoável deste. Esta distância deve ser de cerca de dois carros e isto fará com que o sistema em si (a fila) tenha um comportamento complexo, já que o controle de um carro depende do controle dos carros à sua frente e pelo controle não ser centralizado.

Adotamos como exemplo a solução de McCuskey (2000) e implementamos a

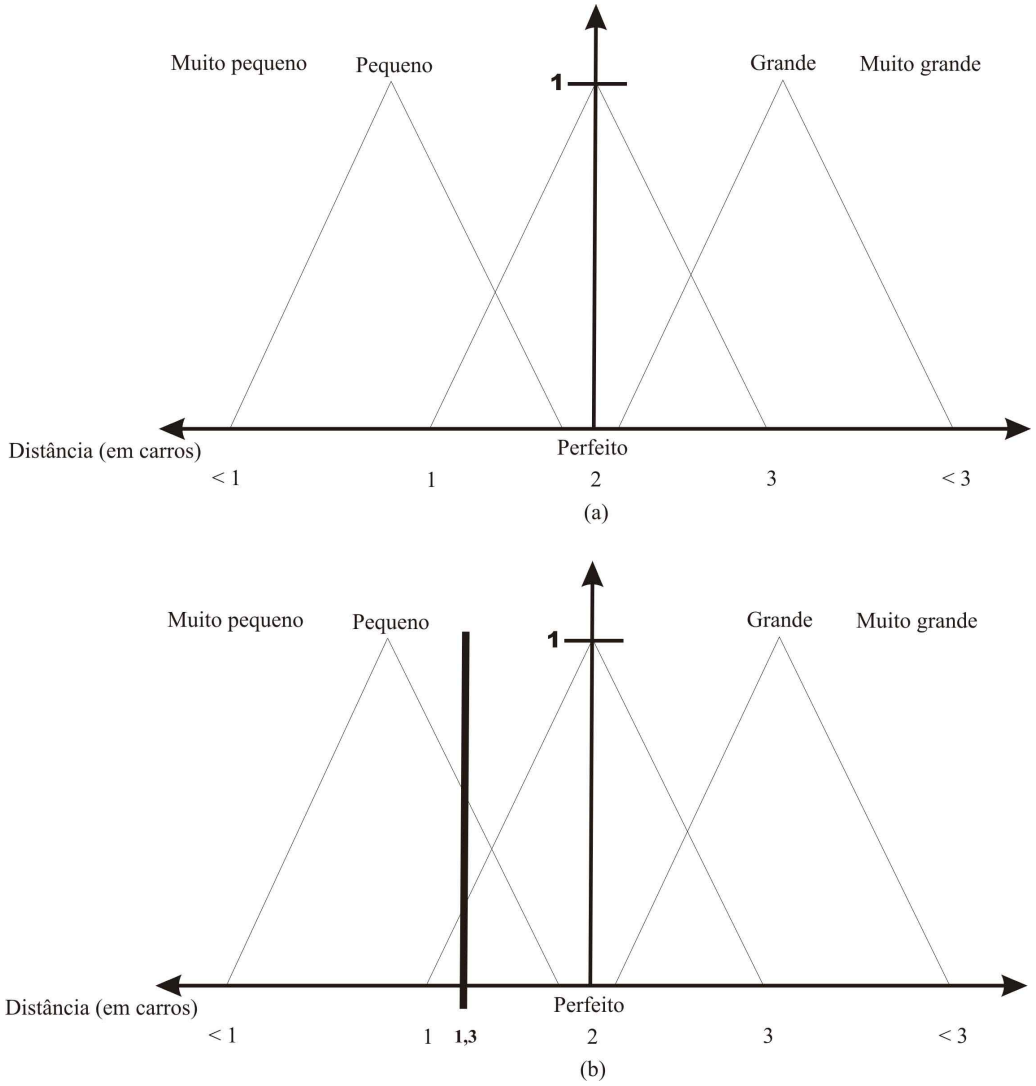

Figura 4: Variável distância.
Fonte: McCuskey (2000).

mesma utilizando a biblioteca FFLL como é apresentado no Quadro 4 e utilizando as regras encontradas no Quadro 5.

Neste problema há duas variáveis: a distância entre o carro do agente e o veículo imediatamente à sua frente e a variação desta distância. Se esta variação delta é positiva, então significa que o espaço entre os carros está aumentando e o agente deverá aumentar a velocidade afim de acompanhar a fila. Se delta for negativo o carro deverá diminuir a velocidade ou até parar em segurança. No caso de um delta igual a zero o agente deve manter a velocidade.

A variável distância (Figura 4 (a)) pode ser modelada tendo cinco conjuntos *fuzzy*, cada um representando uma distância relativa entre os carros. Estas distâncias represen-

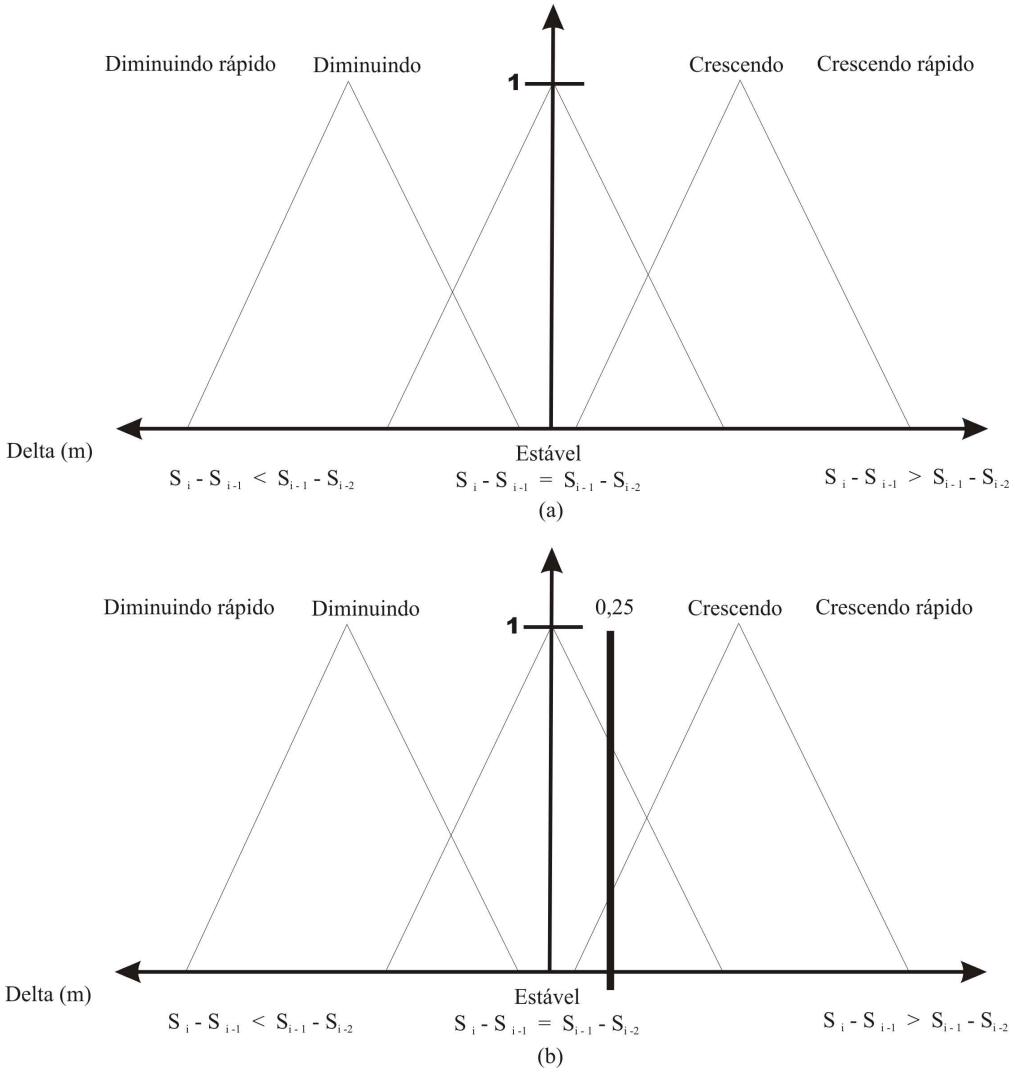

Figura 5: Variável distância delta.
Fonte: McCuskey (2000).

tam espaço “muito pequeno” (distância menor do que o tamanho de um carro), “pequeno” (aproximadamente um carro), “perfeito” (aproximadamente dois carros), “grande” (perto de três carros) e “muito grande” (mais de três carros). É possível aplicar a técnica de nível de detalhe aqui considerando que distâncias infinitas ignoram o controle *fuzzy*. Para isto pode ser adotado como infinito, por exemplo, o comprimento de vinte carros.

A variação delta da distância (Figura 5 (a)) por sua vez utiliza outros cinco conjuntos que representam o quanto o carro do agente se aproxima ou se afasta do carro da frente. São estes conjuntos o “diminuindo rápido” (delta muito negativo), o “diminuindo” (delta negativo, mas com valor absoluto pequeno), o “estável” (delta próximo de zero), o “crescendo” (delta positivo) e o “crescendo rápido” (delta positivo com valor alto).

```
1   FUNCTION_BLOCK
2
3   VAR_INPUT
4           Distancia       REAL; (* RANGE(0 .. 20) *)
5           Delta           REAL; (* RANGE(-20 .. 20) *)
6   END_VAR
7
8   VAR_OUTPUT
9           Acao    REAL; (* RANGE(0 .. 5) *)
10  END_VAR
11
12  FUZZIFY Distancia
13          TERM Muito_Pequeno := (0, 1) (0.5, 0) (1, 0) ;
14          TERM Pequeno := (0.5, 0) (1, 1) (2, 0) ;
15          TERM Perfeito := (1, 0) (2, 1) (3, 0) ;
16          TERM Grande := (2, 0) (3, 1) (4, 0) ;
17          TERM Muito_Grande := (3, 0) (4, 1) (20, 0) ;
18  END_FUZZIFY
19
20  FUZZIFY Delta
21          TERM Diminuindo_Rapido := (-20, 0) (-1, 1) (-0.5, 0);
22          TERM Diminuindo := (-1, 0) (-0.5, 1) (0, 0);
23          TERM Estavel := (-0.5, 0) (0, 1) (0.5, 0);
24          TERM Crescendo := (0, 0) (0.5, 1) (1, 0);
25          TERM Crescendo_Rapido := (0.5, 0) (1, 1) (20, 0);
26  END_FUZZIFY
27
28  FUZZIFY Acao
29          TERM Frear_Forte := 1 ;
30          TERM Frear := 2 ;
31          TERM Manter := 3;
32          TERM Acelerar := 4 ;
33          TERM Acelerar_Forte := 5;
34  END_FUZZIFY
35
36  DEFUZZIFY valve
37          METHOD: MoM;
38  END_DEFUZZIFY
39
40  RULEBLOCK first
41            ...
42  END_RULEBLOCK
43
44  END_FUNCTION_BLOCK
```

Quadro 4: Exemplo de controle *fuzzy* aplicado em um motorista automático.

A variável de saída é uma combinação das outras duas. Esta é mostrada na Figura 6 e tem cinco conjuntos que representam a ação que o motorista deve tomar e a intensidade desta. Os conjuntos desta variável são “frear forte”, “frear”, “manter”, “acelerar” e “acelerar forte”.

Estabelecendo como exemplo uma distância entre o carro e o veículo que segue a frente igual ao comprimento de 1.3 carros (Figura 4 (b)), obtemos que esta é considerada “quase pequena” (ou um DOM de cerca de 0.75 para “pequena” e 0.1 para “perfeita”). Também adotando uma distância delta de 0.25 (Figura 5 (b)), esta pode ser vista como “timidamente crescendo” (ou um DOM de 0.3 para “crescendo” e 0.6 para “estável”).

Como é visto acima, os valores escolhidos para este exemplo interceptam dois conjuntos cada. Assim são necessárias quatro regras para tomar a decisão baseadas na combinação das variáveis distância e distância delta. O Quadro 6 mostra as quatro regras utilizadas neste caso.

O próximo passo é verificar o quanto cada uma das regras é verdadeira. Para isto devem ser selecionadas as áreas dos gráficos das duas variáveis onde ambas são verdadeiras simultaneamente, produzindo um gráfico de saída como o da Figura 6. Um exemplo é a

```
1   RULEBLOCK first
2           AND:MIN;
3           ACCUM:MAX;
4           RULE 0: IF Muito_Pequeno AND Diminuindo_Rapido THEN Frear_Forte;
5           RULE 1: IF Muito_Pequeno AND Diminuindo THEN Frear_Forte;
6           RULE 2: IF Muito_Pequeno AND Estavel THEN Frear;
7           RULE 3: IF Muito_Pequeno AND Crescendo THEN Frear;
8           RULE 4: IF Muito_Pequeno AND Diminuindo THEN Manter;
9
10          RULE 5: IF Pequeno AND Diminuindo_Rapido THEN Frear_Forte;
11          RULE 6: IF Pequeno AND Diminuindo THEN Frear;
12          RULE 7: IF Pequeno AND Estavel THEN Frear;
13          RULE 8: IF Pequeno AND Crescendo THEN Manter;
14          RULE 9: IF Pequeno AND Diminuindo THEN Acelerar;
15
16          RULE 10: IF Perfeito AND Diminuindo_Rapido THEN Frear;
17          RULE 11: IF Perfeito AND Diminuindo THEN Frear;
18          RULE 12: IF Perfeito AND Estavel THEN Manter;
19          RULE 13: IF Perfeito AND Crescendo THEN Acelerar;
20          RULE 14: IF Perfeito AND Diminuindo THEN Acelerar;
21
22          RULE 15: IF Grande AND Diminuindo_Rapido THEN Frear_Forte;
23          RULE 16: IF Grande AND Diminuindo THEN Manter;
24          RULE 17: IF Grande AND Estavel THEN Acelerar;
25          RULE 18: IF Grande AND Crescendo THEN Acelerar;
26          RULE 19: IF Grande AND Diminuindo THEN Acelerar_Forte;
27
28          RULE 20: IF Muito_Grande AND Diminuindo_Rapido THEN Manter;
29          RULE 21: IF Muito_Grande AND Diminuindo THEN Acelerar;
30          RULE 22: IF Muito_Grande AND Estavel THEN Acelerar;
31          RULE 23: IF Muito_Grande AND Crescendo THEN Acelerar_Forte;
32          RULE 24: IF Muito_Grande AND Diminuindo THEN Acelerar_Forte;
33  END_RULEBLOCK
```

Quadro 5: Regras aplicadas no motorista automático do exemplo de controle *fuzzy*.

```
1   if (distância é pequena e delta é crescente) manter;
2   if (distância é pequena e delta é estável) freiar;
3   if (distância é perfeita e delta é crescente) acelerar;
4   if (distância é perfeita e delta é estável) manter;
```

Quadro 6: As quatro regras utilizadas para o DOM 1.3 para distância e 0.25 para distância delta.

regra exibida na linha 1 do Quadro 6. Esta tem um grau de verdade 0.3 no gráfico de saída, pois entre o grau 0.75 da variável distância e o 0.3 da variável distância delta, o maior DOM possível simultaneamente.

A partir deste gráfico é possível fazer a *desfuzzyficação* calculando o centróide, ou centro de massa, da figura formada pelas áreas cobertas pelas regras. Neste caso foi obtido como resultado no gráfico de saída "frear" com 25% de redução da velocidade atual do carro. Dado que a regra "frear" para o grau 1.0 equivalente a 0.75, então a velocidade do carro deve ser multiplicada por 0.81.

2.7.3 Máquinas de estado

A máquina de estados é um formalismo muito utilizado na modelagem de comportamento de programas de computador, projeto de *hardware*, engenharia de *software* e em várias outras aplicações das ciências da computação.

Nos jogos, em especial na IA para jogos, este formalismo também é muito aplicado na especificação do comportamento dos agentes. Isto se deve ao fato da máquina de estado

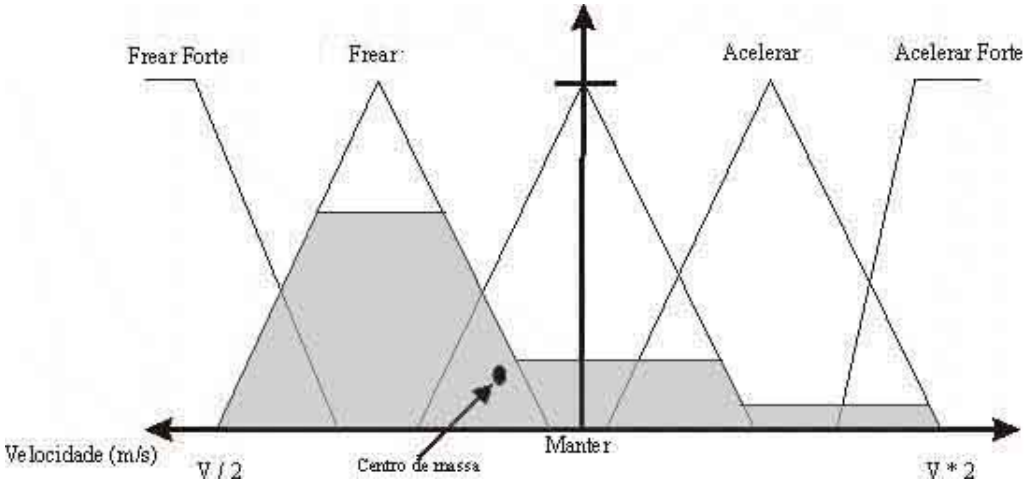

Figura 6: *Desfuzzyficação* do problema da velocidade.
Fonte: McCuskey (2000).

oferecer grande poder com pouquíssima complexidade se comparada com outras técnicas utilizadas para especificar agentes inteligentes para jogos. Além disto, estas máquinas são intuitivas e fáceis de implementar, o que torna esta técnica a mais utilizada neste tipo de aplicação.

A máquina de estados é composta por três elementos, ou seja, estados, transições e ações. Tem ela um número finito de estados que armazenam informações sobre o passado, refletindo as mudanças ocorridas no sistema desde o início de sua execução até o estado atual. Uma transição indica a mudança de estado e obedece a uma condição que precisa ser verdadeira para que a transição ocorra, levando o sistema para outro estado. Uma ação é a descrição de uma atividade que deve ser realizada em determinado momento. Os tipos possíveis de ação são a entrada ou saída de um estado, a entrada de dados ou uma ação associada à uma transição, em última instância são as mudanças efetuadas no mundo pelo agente.

Formalmente a máquina de estados é uma quíntupla $FSM = \{\Sigma, Q, Z, \delta, \lambda\}$, onde Σ é um alfabeto de entrada, Q é um conjunto de estados, Z é um alfabeto de saída, δ é a função de transição e λ é uma função que determina as saídas tomando como base as transições. As transições da máquina FSM são disparadas quando as entradas são validadas pela função δ como verdadeiras. Com base no disparo desta transição, a função λ produz as saídas.

Uma máquina de estado pode ser vista também como um dispositivo, ou modelo de dispositivo, que tem um número finito de estados em que o dispositivo pode estar em

um determinado instante de tempo. Esta pode receber entradas para mudar do estado atual para outro estado através de transições, fazer uma saída de dados ou executar uma ação. Uma máquina de estados pode estar apenas em um estado em um dado instante de tempo.

A representação das máquinas de estado pode ocorrer de duas formas, ou seja, graficamente (Figura 7) ou utilizando tabelas (Tabela 1). Na representação gráfica os nodos do grafo representam os estados e as arestas representam as transições. As transições têm uma condição associada que comanda a mudança de estado e faz com que o estado atual mude para o estado destino que é ligado ao atual pela aresta. Na forma de tabelas as linhas dessa representam as possíveis transições e as colunas os respectivos estados. A célula da tabela endereçada pelo cruzamento do estado atual e da transição que foi executada guarda a referência para o estado destino que indica para que estado a máquina deve ir após a transição ser disparada.

Tabela 1: Exemplo de máquina de estados representada em forma de tabela.

	Estados				
Transições		Aborta	Dirigindo	Ultrapassando	Tira carro
	C1	-	Ultrapassando	-	-
	C2	-	Tira carro	-	-
	C3	-	-	Dirigindo	-
	C4	-	-	Aborta	-
	C5	-	-	-	Dirigindo
	C6	Ultrapassando	-	-	-
	C7	Dirigindo	-	-	-

Na maioria das situações o comportamento do agente pode ser dividido em partes e essas podem ser representadas por estados. Um grande problema das máquinas de estado é o crescimento da complexidade conforme são adicionados mais estados nas mesmas. Para contornar o problema, as partes também podem ser subdivididas, hierarquicamente, e assim gerar outras máquinas de estados que tratam de subproblemas da máquina principal (BLY, 2004).

Além das máquinas hierárquicas, existem outras variantes das máquinas de estados. Uma destas é a FUSM (DYBSAND, 2001) que consiste em uma máquina de estado que se diferencia das outras pelas condições das transições a serem representadas por lógica *fuzzy*. A associação desta lógica com a máquina de estados tira um pouco do determinismo do modelo original, fazendo com que comportamento do agente seja mais

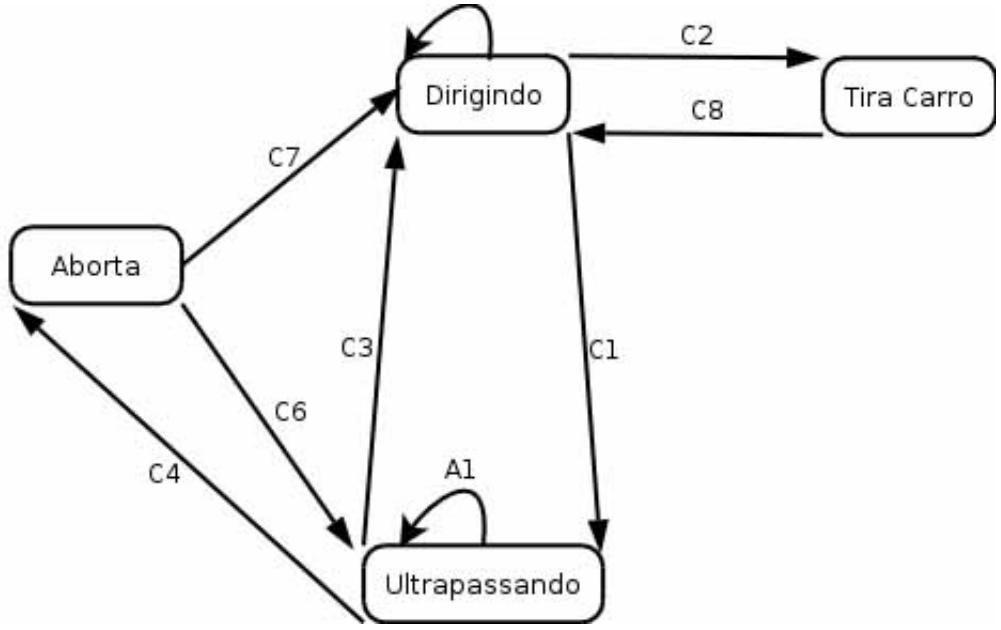

Figura 7: Representação gráfica da máquina de estados listada no Quadro 7.

interessante para o uso em jogos.

Entre os jogos mais conhecidos que utilizaram as máquinas de estados está o exemplo clássico do jogo *Pack-Man*[6]. O comportamento dos fantasmas em *Pack-Man* é implementado através do uso deste formalismo.

Em tempo de execução a máquina de estados é descrita como um conjunto de condições, assim não há um processamento real de raciocínio. Na verdade existe uma busca pelo estado ideal descrito dentro pela máquina, às vezes diretamente dentro do próprio código do jogo, outras em arquivos de configuração carregados em tempo de execução (RABIN, 2000b). Este formalismo permite a fácil expansão do escopo de um agente simplesmente adicionando novas regras e estados aos já existentes. As máquinas de estados permitem facilmente que se utilize outras técnicas junto à mesma, assim como lógica *fuzzy* e redes neurais (LAMOTHE, 2000).

Este formalismo apresenta algumas desvantagens no uso em jogos. Entre estas está a dificuldade em expandir o conhecimento do agente durante o tempo de execução. Se este for totalmente definido através de uma máquina de estados, isto torna a inteligência artificial estática. Outra desvantagem frente a outras técnicas é que estes agentes apresentam um comportamento de certo modo previsível. Assim é interessante que pelo menos os agentes que influenciam mais no jogo mesclem o uso da máquina de estados com

[6]Copyright Midway, 1980

outras técnicas para ter um comportamento mais apurado, embora mais custoso.

A Figura 7 apresenta uma máquina de estados que implementa um controle de alto nível para o agente motorista automático apresentado nas seções anteriores. Esta máquina tem quatro estados (Figura 7) e sua implementação é mostrada no Quadro 7. Os estados são descritos a seguir:

- "Dirigindo" - o agente trafega em uma pista de rolamento sem problemas. Na ocorrência de outros veículos seguindo a sua frente este utiliza controle *fuzzy* (Seção 2.7.2), caso contrário aplica a heurística de nível de detalhe e deixa o carro trafegar livremente. Se existir um carro à frente e esse estiver indo muito abaixo da velocidade máxima, o agente tem a opção de ultrapassá-lo e assim manter certa velocidade. A transição para o estado "Ultrapassando" é acionada pelo segmento de código identificado no quadro como $C1$ (Quadro 7). Se o motorista automático se deparar com um obstáculo à sua frente pode chamar a transição para o estado "Tira carro" usando o código identificado como $C2$.

- "Tira carro" - caso haja um veículo muito lento ou parado a frente, ou carro vindo na contra mão de direção e não haja condições de frear, então este estado aplica a condição de contorno que faz com que o agente tente tirar o carro para o acostamento ou para a outra pista. Isto não leva em conta apenas a boa prática no trânsito, mas também as reações naturais dos motoristas reais. Quando o problema com o obstáculo é resolvido o sistema volta ao estado "Dirigindo" através da transição $C8$. Este código não é detalhado no Quadro 7.

- "Ultrapassando" - o carro do agente invade a pista contrária para ultrapassar outro veículo mais lento, as condições para este estado devem levar em conta o tráfego na pista contrária tomando o cuidado de não colidir frontalmente com outro veículo. Ao entrar neste estado existe a passagem para a outra pista que é realizada pela ação $A1$ (Quadro 7). Dentro deste estado há duas transições possíveis, ou seja, voltar ao estado "Dirigindo" ($C3$) ao final da operação ou passar para o estado "Aborta" ($C4$).

- "Aborta" - aborta uma ultrapassagem no caso do agente estar fazendo esta operação. Neste caso é possível tentar aumentar a velocidade e passar pelo veículo que está sendo ultrapassado ou frear o carro e voltar para a posição em que estava antes de iniciar a operação.

Se o agente decidir forçar a ultrapassagem, deve aumentar a velocidade e executar a transição para o estado “Ultrapassando” ($C6$). Se, mesmo assim, a velocidade escolhida for insuficiente para terminar a ultrapassagem, será executada nova transição para “Abortar” ($C4$).

No caso do agente optar por desistir da ultrapassagem e voltar atrás, “Abortar” leva o carro para a posição atrás do veículo que estava sendo ultrapassado e passa a máquina para o estado “Dirigindo” ($C7$). O estado “Abortar” também não é detalhado no código fonte do Quadro 7.

A Tabela 1 exibe a representação tabular da máquina de estados mostrada na Figura 7 e codificada no Quadro 7.

```
1   BeginStateMachine
2                   OnEnter{
3                           SetState(DIRIGINDO);
4                   }
5                   OnMsg(HOUVE_COLISAO){
6                           ...
7                   }
8                   State(DIRIGINDO){
9                           OnEnter{
10                                  ...
11                          }
12                          OnUpdate{
13                                  //C1
14                                  if (v' = v + (20 * v) / 100 &&
15                                       v' = (velocidade_maxima < v') ? velocidade_maxima : v' &&
16                                       pista[i + 1] == NULL &&
17                                       dist(me, obstaculo(pista[i + 1]) > comprimento(carro[j + 1]) * 3 &&
18                                       (v' - v) / 3.6 >= comprimento(carro[j + 1]) * 3 &&
19                                       dist(carro[j + 1], carro[j + 2]) > 3 * mysize())
20                                                  SetState(ULTRAPASSANDO);
21                                  //C2
22                                  if (dist(me, obstaculo(pista[i]) < 2 * mysize())
23                                                  SetState(TIRA_CARRO);
24                          }
25                          OnExit{
26                                  ...
27                          }
28                  }
29                  State(ULTRAPASSANDO){
30                          OnEnter{
31                                  //A1
32                                  me->pista = i + 1;
33                                  v = v';
34                          }
35                          OnUpdate{
36                                  //C3
37                                  if (dist(me, pista[i - 1]->carro[j + 1], me) > mysize() &&
38                                         afrente(me, pista[i - 1]->carro[j + 1]){
39                                              pista[i - 1]->carro[j - 1] = pista[i - 1]->carro[j + 1];
40                                              pista[i - 1]->carro[j + 1] = me
41                                              SetState(DIRIGINDO);
42                                  }
43                                  //C4
44                                  if (dist(me, obstaculo(pista[i + 1]) < 2 * mysize())
45                                              SetState(ABORTA);
46                          }
47                          OnExit{
48                                  ...
49                          }
50                  }
51                  State(ABORTA){
52                          ...
53                  }
54                  State(TIRACARRO){
55                          ...
56                  }
57  EndStateMachine
58
```

Quadro 7: Exemplo de máquina de estados aplicada em jogos.

2.7.4 Conclusões

Esta seção analisou os principais problemas encontrados pelos desenvolvedores de jogos, bem como as técnicas mais utilizadas para resolvê-los. Dentre os problemas um dos mais significativos é o tempo que um agente leva para atualizar a sua inteligência, a qualidade das decisões deste e quantos agentes devem ser atualizados. Assim a escolha entre uma técnica ou outra depende de vários fatores heurísticos que definem a importância de determinado agente no mundo. Técnicas que dão resultados mais precisos e consequentemente propiciam um melhor resultado no jogo tendem a ser mais complexas e a exigir mais tempo de processamento. Entre estas heurísticas estão o nível de detalhe e o balanceamento de carga.

Outro desafio para os desenvolvedores é o reaproveitamento de código e tentativa de reduzir a complexidade da implementação. A cada dia é exigido mais poder do motor de IA devendo esse oferecer máquinas de estado, sistemas de regras de produção, lógica *fuzzy*, entre outras técnicas prontas para o uso e acopladas, ou seja, deve ser possível para a implementação de cada uma das técnicas trocar mensagens entre as demais e delegar tarefas umas às outras conforme for conveniente de acordo com as heurísticas.

3 Planejamento baseado em redes de tarefas hierárquicas para jogos

Planejamento consiste em construir sequências de ações simples que formem um caminho entre o estado atual e o objetivo. Essa sequência de ações é denominada plano e pode ser processada por um módulo que aplica essas operações no mundo a fim de atingir o efeito desejado. Este módulo é denominado executor de planos (NILSSON, 1980).

Um exemplo prático de problema de planejamento em jogos é o desejo do agente de ler um jornal. Para isto ele deve ter um exemplar do jornal, ou providenciar um. No caso dele não ter o jornal precisa pegar a carteira com seu dinheiro e ir até a banca para comprá-lo. Neste problema o agente também precisa decidir a maneira que usará para se deslocar até a banca. Entre as maneiras com que ele poderia ir estão caminhando, de carro, de ônibus ou de táxi.

Cada maneira tem fatores prós e contras à sua aplicação. Por exemplo, se o agente decidir ir de carro ele gastará mais do que de ônibus, mas menos do que de táxi no deslocamento. Mas terá de lidar com estacionamentos ou "flanelinhas". Por outro lado se ele for de ônibus terá um preço fixo que é muito mais baixo do que o do táxi, mas o agente demorará bem mais para ir até a banca e voltar para a sua casa para ler o jornal.

Assim o planejamento consiste na solução ideal para as situações em jogos que exijam o gerenciamento de recursos e atividades, em especial aquelas situações onde o agente tenha muitas opções para desempenhar o mesmo papel.

STRIPS (*Stanford Research Institute Problem Solver*) apresentado por (FIKES; NILSSON, 1971) é considerado o precursor dos sistemas de planejamento modernos. Este introduziu o operador que consiste em uma forma de representação das ações que é mantida na maioria dos planejadores atuais.

Um operador de STRIPS é distinto dos demais por seu nome e este consiste em três partes, são essas:

- a pré-condição expressa uma condicionante que deve ser atendida para que o operador possa ser aplicado e assim efetuar as mudanças no mundo;
- a lista de adições representa os fatos que devem ser adicionados ao estado atual do mundo após a execução da ação. Esta representa os efeitos positivos, ou seja, o que passou a ser verdade após a aplicação do operador;
- a lista de exclusões é o inverso da lista de adições, ou seja, representa os efeitos negativos ou o que passou a ser falso depois da execução do operador (NILSSON, 1980).

As listas de efeitos positivos e negativos podem ser agrupadas em uma única lista como acontece na linguagem de descrição de domínio PDDL (*Planning Domain Definition Language*) (LONG; FOX, 2002). Para isso basta utilizar o operador de negação antes dos átomos que deixarão de ser verdadeiros no mundo, ou seja, aqueles que estariam na lista de exclusões.

Em jogos, planejamento é a técnica de IA para tomada de decisões de mais alto nível. O plano gerado pelo planejador contém um conjunto de ações, que mesmo primitivas do ponto de vista do planejamento, geralmente necessitam ser retrabalhadas através de outras técnicas e algoritmos implementados no motor de IA. Desta forma o executor de planos pode ser visto como o "comandante" das outras técnicas de IA que resolvem os subproblemas na ordem estipulada pelo planejador.

O planejamento em jogos também difere muito de um programa para outro quanto à forma em que é implementado. A maior parte dos jogos oferece planos prontos associados a estímulos ou faz alguma análise do estado atual utilizando regras que implicam em planos prontos ou regras associadas a arvores de decisão (LAMOTHE, 1999).

Alguns jogos mais modernos utilizam planejamento hierárquico afim de construir agentes orientados a objetivos. Estes agentes são definidos em termos de tarefas que devem ser executadas para atingir o objeto de sua ação.

Os planejadores podem ser classificados como clássicos ou hierárquicos. Os planejadores clássicos utilizam busca no espaço de estados associada com análise meio-fim para construir um plano que ligue o estado inicial ao objetivo. A análise meio-fim verifica quais ações devem ser adicionadas ao plano, em ordem, para que depois da execução do mesmo seja obtido o efeito desejado. Por outro lado, os hierárquicos, fazem decomposição de tarefas, ou seja, partem de uma tarefa complexa que é reduzida em uma ou mais tarefas com menor grau de complexidade.

Nos planejadores hierárquicos são utilizados métodos para reduzir as tarefas de maior nível de abstração. Os métodos especificam as possíveis reduções destas tarefas para conjuntos de zero ou mais tarefas abstratas ou primitivas. Cada redução tem uma condição associada que define se ela é aplicável ou não em determinada situação. As tarefas primitivas são realizadas por operadores que efetivam as mudanças no estado do mundo. Os operadores utilizados para realizar as tarefas primitivas têm uma estrutura parecida com os operadores de STRIPS e cada um deles está associado a uma determinada tarefa primitiva. A realização das tarefas através dos operadores tem o mesmo efeito da instanciação das ações de um plano na abordagem clássica.

Tanto nos planejadores clássicos quanto nos hierárquicos, o processo termina quando existir no plano apenas ações ou tarefas primitivas que possam ser executadas diretamente pelo executor de planos. Entre as operações que um executor de planos poderia executar estão andar, virar, pular, etc. Embora o método clássico seja efetivo para resolver problemas de domínios pequenos e estáticos, este falha ao tentar tratar domínios dinâmicos e mais complexos. Os planejadores clássicos são também conhecidos como baseados em estado e os hierárquicos como baseados em abstração.

Os objetivos atômicos representam uma única tarefa, comportamento ou ação que pode ser diretamente realizada por outro mecanismo de IA ou mesmo modificar o mundo diretamente. Os objetivos compostos têm vários subobjetivos que podem ser atômicos ou não e são definidos como uma hierarquia. Segundo Buckland (2005) o planejamento hierárquico é intuitivo para os engenheiros de conhecimento, pois, é estruturado de uma forma similar à forma usada pelas pessoas para reduzir problemas.

O planejamento pode ser encadeado para frente ou para trás como pode ser visto na Figura 8 (HAWES, 2003). No primeiro caso o plano é construído adicionando ações, uma a uma, do estado inicial ao objetivo. A segunda alternativa é adotar o processo inverso e construir o plano do estado final ao inicial.

Um plano consistente é uma lista de ações que leva o mundo do estado inicial até o estado desejado. O estado inicial de um problema consiste em um conjunto de átomos que devem ser verdadeiros no mundo no começo do processo de planejamento. O objetivo, no planejamento clássico, é um conjunto de átomos que devem ser verdadeiros no mundo após a execução do plano. Utilizando a abordagem baseada em abstração o objetivo é dado por uma lista de tarefas que devem ser realizadas.

A Figura 8 mostra um plano sendo construído por um planejador clássico. Conforme são adicionadas ações ao plano a consistência desse varia. A Figura 8(a) exibe

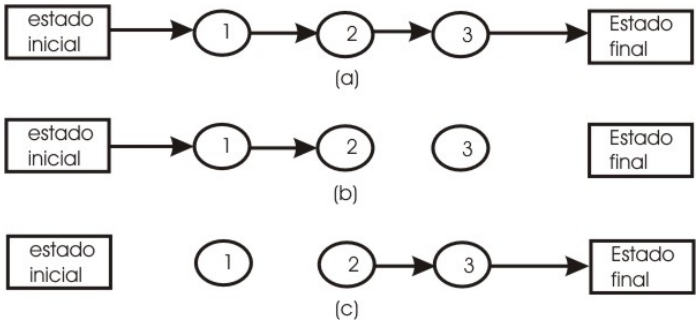

Figura 8: Planejamento baseado em estados: (a) um plano completo, (b) encadeamento para frente e (c) encadeamento para trás.

Fonte: Hawes (2003).

um plano completo e consistente, mas se o plano for necessário antes de estar completo este provavelmente não conterá todos os elementos que formam um caminho que vai do estado inicial ao final. Tanto faz a forma escolhida para construir o plano, na Figura 8(b) um plano inconsistente foi construído por um planejador que utiliza encadeamento para frente e em (c) um plano, também inconsistente, foi feito por um planejador que adota encadeamento para trás. Isto se dá devido à consistência do plano depender das ações que serão futuramente adicionadas.

No planejamento baseado em abstração o plano é sempre consistente devido às tarefas serem consistentes individualmente. Dessa maneira, o plano sempre contém todas as "ações" que levarão o mundo de um estado para o outro, assim, o que muda é o nível de abstração das tarefas conforme o plano é refinado (HAWES, 2003). Hawes (2003) atribui o insucesso dos planejadores implementados como algoritmo *anytime* baseados em busca no espaço de estados para uso em aplicações de tempo real ao problema provocado por parar o algoritmo e não haver um plano consistente para resolver o problema parcialmente. Esse autor propôs resolver o problema com uma abordagem mista, utilizando planejamento para as tarefas de mais alto nível e sistemas reativos para as demais. Segundo Nau e colegas (1999) utilizar encadeamento para a frente oferece uma série de vantagens, incluindo a simplificação do processo de planejamento.

Existe um caso especial de planejamento em jogos, o planejamento de caminhos (*path planning*), que consiste na formulação de um caminho entre um ponto A e um ponto B de forma que o agente desvie dos obstáculos e atinja seus objetivos de locomoção de forma realista. Isto é a parte de baixo nível das tarefas do tipo $Ir(A, B)$ e geralmente é resolvido através de variações do algoritmo A* (MATTHEWS, 2002).

Em mundos muito complexos como os que apresentam ambientes contínuos, com

muitos obstáculos é conveniente definir uma rede de pontos cujas ligações representam os caminhos transitáveis pelo agente. Quando o agente precisa de um caminho ele utiliza o algoritmo A* para encontrá-lo dentro das possibilidades representadas pela rede (BUCKLAND, 2005).

O algoritmo dedicado de busca de caminhos torna o processo de busca pelos caminhos que o agente deve percorrer muito mais barato do que se fosse utilizado um planejador de uso geral para a mesma aplicação. Desta forma, quando surgir uma tarefa primitiva do tipo $Ir(A, B)$, essa será encarada como uma ação que deverá ser executada no mundo. Neste caso o algoritmo de busca de caminhos deverá ser acionado para solucionar o problema. Esta chamada é implementada no executor de planos.

Todas as tarefas primitivas que não se limitam a apenas mudar o estado atual, mas executam também alguma modificação no mundo físico do jogo tem uma ação associada. Esta ação pode ser a chamada para algum algoritmo que produza alguma inteligência, um plano pré-definido ou mesmo uma animação. Em Hawes (2003), devido ao processo de planejamento parar antes do plano todo estar instanciado, o autor também implementou ações para os métodos.

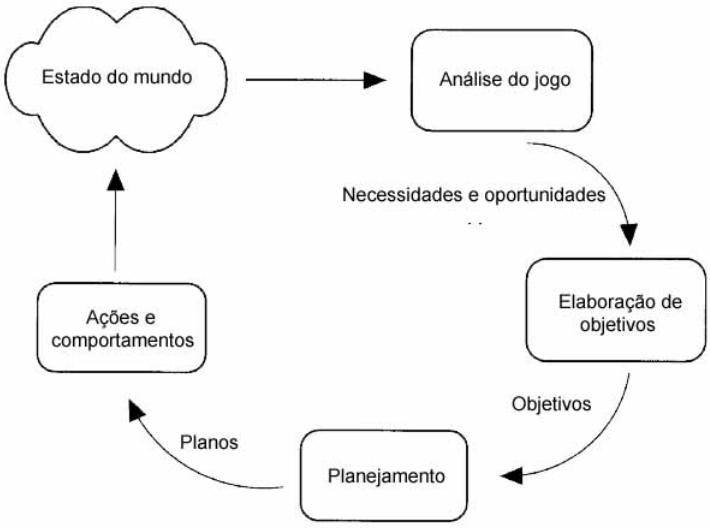

Figura 9: Arquitetura de planejamento para jogos.
Fonte: O´Brien (2002).

A Figura 9 mostra a arquitetura baseada em agentes orientados a objetivos apresentada em O´Brien (2002). Nesta arquitetura o agente percebe o mundo através de um conjunto de sensores implementados na forma de funções. Com base nestas percepções, o agente define os objetivos e necessidades que têm em determinado momento.

Destes objetivos são selecionados os que têm maior prioridade para a elaboração do problema e submissão deste para o planejador. A partir deste passo o plano é construído e o executor de planos transforma o plano abstrato em um conjunto de ações que se propagam de técnica em técnica até o nível mais baixo da hierarquia de algoritmos de decisão. Em geral, neste nível há os algoritmos de movimentação e animações (LAMOTHE, 1999).

3.1 Redes de tarefas hierárquica (HTN)

O planejamento através de redes de tarefas hierárquica (HTN) é uma abordagem ao planejamento automático baseado em abstração na qual a dependência entre as ações podem ser dadas na forma de uma rede.

O objetivo do planejamento através de redes de tarefas hierárquica é produzir sequências de ações que executem algumas atividades ou tarefas (NAU et al., 2003). Na Figura 10 a tarefa "Ir ao Shopping" é reduzida, ou seja, substituída por uma lista de tarefas contendo "Obter Dinheiro", "Fazer Lista", "Vá ao Shopping" e "Comprar Itens". Segundo o diagrama as tarefas "Fazer Lista" e "Obter Dinheiro" podem ser executadas em paralelo ou intercaladas.

As HTN foram implementadas em muitos trabalhos práticos desde meados da década de setenta, mas somente em Erol, Hendler e Nau (1994a) esta técnica recebeu uma formalização através da atribuição de uma sintaxe e semântica ao modelo. Em Erol, Hendler e Nau (1994b) foi apresentado o algoritmo UMCP. Esse é um algoritmo baseado em HTN que tem as propriedades de ser completo e correto.

O planejamento utilizando as redes de tarefas hierárquica é semelhante ao planejamento clássico em muitos aspectos. Em ambos cada estado do mundo é representado por um conjunto de átomos e cada ação corresponde à uma ação determinística no espaço de estados. Mas as principais diferenças entre os planejadores HTN e os baseados em STRIPS (NILSSON, 1980) está nos objetivos que eles buscam e em como acontece o planejamento para atingir esses objetivos. Os baseados em STRIPS buscam por sequências de ações que levem o mundo do estado atual até um que satisfaça certas condições. O algoritmo HTN procura por operadores que tenham o efeito desejado para atingir os sub-objetivos, até que isso leve ao objetivo final. Em HTN o planejador procura realizar tarefas, decompondo e resolvendo conflitos entre elas (EROL; HENDLER; NAU, 1994a) (NAU et al., 2003).

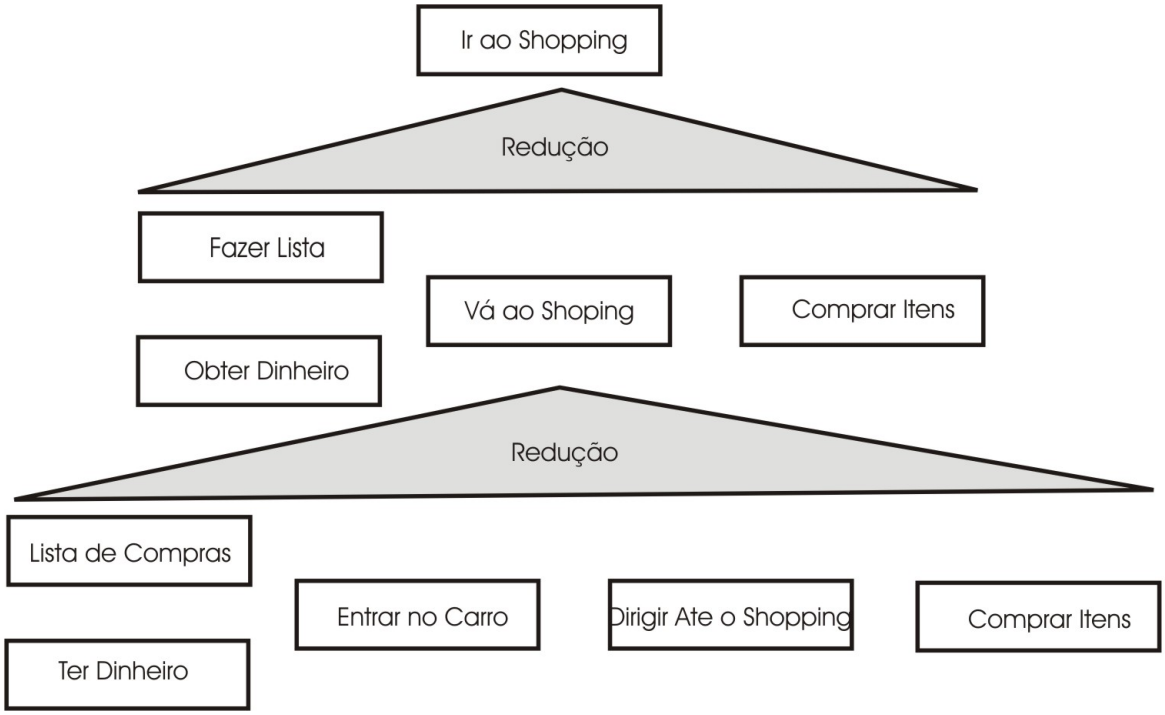

Figura 10: Um exemplo de redução em uma HTN.
Fonte: Hawes (2003).

Uma das principais funções de um algoritmo de planejamento é detetar e resolver conflitos. Uma rede de tarefas é uma coleção de tarefas que devem ser executadas obedecendo às suas restrições de ordenação. Essas restrições definem a forma com que as variáveis devem ser instanciadas e quais literais devem ser verdadeiras para que as tarefas sejam executadas. Em UMCP (EROL; HENDLER; NAU, 1994b) são usadas funções chamadas críticas. Essas servem para manipular restrições, limitações de recursos e prover heurísticas dependentes de domínio. Em SHOP (Nau, Muñoz-Avila e Lotem (1999)) essas foram encapsuladas na forma de axiomas e na própria estrutura dos métodos e formulas condicionais (EROL; HENDLER; NAU, 1994b) (NAU; MUñOZ-AVILA; LOTEM, 1999).

A representação do mundo em HTN é similar à de STRIPS (NILSSON, 1980), onde cada estado do mundo é representado por um conjunto contendo os átomos que devem ser verdadeiros no estado atual. A descrição do domínio de planejamento inclui um conjunto de operadores similares aos utilizados pelos planejadores clássicos e um conjunto de métodos, cada um desses com a descrição de como reduzir um tipo de tarefa em subtarefas. O problema é descrito como um conjunto de átomos representando o estado inicial e um conjunto de tarefas a serem realizadas representando o estado final (NAU et al., 2003). Em HTN as ações são chamadas de tarefas primitivas. Essas são transições

entre estados, ou seja, cada tarefa primitiva representa a passagem de um conjunto de estados para outro. As tarefas primitivas são executadas através do uso de operadores. Esses tem a forma $(!h, D, A)$, onde $!h$ é uma tarefa primitiva, D é a lista de átomos que se tornaram falsos no mundo e A a lista dos que se tornaram verdadeiros (EROL; HENDLER; NAU, 1994a) (NAU; MUñOZ-AVILA; LOTEM, 1999).

A rede de tarefas formada apenas por tarefas primitivas é chamada de rede de tarefas primitiva. Essa rede pode ocorrer, por exemplo, em problemas de agendamento. Nesse caso o planejador montará uma agenda (ordenação de tarefas e instanciação de variáveis) que satisfaça as restrições (EROL; HENDLER; NAU, 1994a).

A rede de tarefas pode conter tarefas não primitivas. Tarefas desse tipo não podem ser executadas diretamente, assim o planejador precisa determinar como estas serão realizadas. As tarefas não primitivas representam atividades que envolvem a execução de várias tarefas menos abstratas do que ela. O planejador deverá utilizar os métodos afim de reduzir as tarefas não primitivas até que haja apenas tarefas primitivas no plano.

Métodos descrevem os meios pelos quais as tarefas podem ser reduzidas. Esses tem a forma (h, T), onde h é uma tarefa não primitiva e T é uma lista de tarefas que juntas realizam h (EROL; HENDLER; NAU, 1994a) (NAU; MUñOZ-AVILA; LOTEM, 1999). Formalmente uma rede de tarefas é descrita como $[(n_1 : h_1), ..., (n_n : h_n), F]$, onde:

- cada h_i é uma tarefa;
- n_i é um rótulo que distingue determinada instância de hn_i de qualquer outra ocorrência dessa tarefa na rede;
- F - é uma formula booleana construída da ligação de restrições sobre variáveis. Essas restrições são a igualdade entre variáveis $(v = v')$ ou de uma variável com uma constante $(v = c)$; restrições de ordenação $(n < n')$; e restrições de estado, tais como, $(initially \quad l)$, (n, l), (l, n) e (n, l, n'). Os símbolos v e v' são variáveis, l é um literal, c uma constante e n representa rótulos. Intuitivamente $(n < n')$ significa que a tarefa rotulada com n precisa estar antes da denominada n'; (n, l), (l, n) e (n, l, n') significam que l precisa ser verdadeiro no estado imediatamente após a n, imediatamente antes de n, e em todos os estados entre n e n', respectivamente; $(initially \quad l)$ significa que l precisa ser verdadeiro no estado inicial. Tanto a negação quanto a disjunção são permitidas na formula de restrições (EROL; HENDLER; NAU, 1994a).

Erol, Hendler e Nau (1994a) provam que a linguagem HTN é provavelmente mais expressiva que a sua precursora, STRIPS. A principal diferença entre as linguagens está na inclusão da representação do "desejo de mudança do mundo". As tarefas, que podem ser de três tipos, são divididas em:

- as tarefas primitivas: são tarefas que podem ser diretamente efetivadas pela execução de uma ação correspondente (ex: pagar a casa);
- as tarefas de objetivo: são como os objetivos de STRIPS; propriedades que devem ser verdadeiras no mundo (ex: ter uma casa);
- as tarefas compostas: denotam o "desejo de mudança" do mundo que envolvem vários objetivos e tarefas primitivas (ex: construir uma casa).

As tarefas "ter uma casa" (objetivo) e "construir uma casa" (composta) são diferentes em HTN. Enquanto para satisfazer a primeira basta um ação atômica (comprar a casa) na outra são necessárias várias atividades, tais como, comprar terreno, material, contratar pedreiros, etc (EROL; HENDLER; NAU, 1994b).

3.2 SHOP e SHOP2

SHOP é um planejador baseado em HTN que utiliza encadeamento para frente e ordenação total das tarefas. Segundo Nau, Muñoz-Avila e Lotem (1999), embora convencionou-se na pesquisa da inteligência artificial que o uso dessa abordagem é considerada ruim por gerar excessivo *backtracking*, muitas pesquisas prosseguem nesse sentido. Entre os planejadores que se utilizam de encadeamento para frente se encontram TLPLAN (BACCHUS; KABANZA, 2000) e TALPLANNER (MAGNUSSON; KVARNSTRÖM, 2003).

SHOP (NAU; MUñOZ-AVILA; LOTEM, 1999) surgiu de uma abordagem apresentada em Smith, Nau e Throop (1998). Esta era baseada em reduções HTN e em buscas da esquerda para a direita, como no Prolog, utilizando *backtracking* quando necessário. Embora esta abordagem tenha sido bem sucedida, até o desenvolvimento de SHOP, não foi possível confrontá-la com os algoritmos independentes de domínio devido a essa ser dependente de domínio. A abordagem de Smith, Nau e Throop (1998) foi utilizada com sucesso em aplicações como o jogo Bridge Baron (NAU; MUñOZ-AVILA; LOTEM, 1999).

SHOP é a generalização do trabalho de Smith, Nau e Throop (1998). Este difere de outros planejadores baseados em HTN por gerar cada passo do plano na ordem em

que esse será executado. Desta maneira, SHOP conhece o estado atual do mundo a cada passo do processo de construção do plano. Isto simplifica o planejamento e evita interações indesejáveis entre tarefas. Assim é possível aumentar o poder de expressão do modelo tornando possível incorporar à SHOP computação numérica e simbólica, inferência utilizando axiomas e interações com agentes externos e outras fontes de informação (NAU; MUñOZ-AVILA; LOTEM, 1999), (LONG; FOX, 2003).

Devido ao poder de expressão de SHOP é possível criar representações de domínio muito eficientes. Segundo testes de Nau, Muñoz-Avila e Lotem (1999), SHOP foi muitas ordens de magnitude mais rápido do que Blackbox (KAUTZ; SELMAN, 1998) e muitas vezes mais rápido do que TLPLAN (BACCHUS; KABANZA, 2000) nos testes utilizando os domínios Mundo de Blocos e Logística. Nas competições o algoritmo TLPLAN foi mais rápido que SHOP2 (NAU et al., 2001)(que contém e é mais veloz que SHOP) em todos os domínios, exceto no Satélites Numérico Difícil. Isso provavelmente ocorre devido a influência exercida pelo conhecimento do especialista humano no desempenho do sistema ou a natureza de certos problemas se adaptar melhor a uma das duas técnicas (NAU; MUñOZ-AVILA; LOTEM, 1999) (LONG; FOX, 2003).

Segundo Nau, Muñoz-Avila e Lotem (1999) SHOP é polinomial para a maior parte dos domínios e tanto ele quanto SHOP2, aparentemente, pertencem à mesma classe de complexidade do algoritmo de planejamento TLPLAN (BACCHUS; KABANZA, 2000).

Tanto SHOP quanto SHOP2 tem as propriedades de ser completo e correto (NAU et al., 2001). Os planejadores semi-automáticos dependem do conhecimento do especialista e por isso apresentam um desempenho maior do que os automáticos, tal como o planejador Blackbox. Infelizmente este é o preço pago pela diferença de desempenho entre os automáticos e os semi-automáticos.

Embora Nau, Muñoz-Avila e Lotem (1999) compare SHOP com TLPLAN, utilizando testes próprios, este foi desclassificado em uma das etapas da competições de 2000 devido aos pesquisadores não terem conseguido terminar a depuração da base de conhecimento dentro do tempo estabelecido. A grande fragilidade de SHOP é justamente a complexidade de sua base de conhecimento. Isso se deve em muito à obrigatoriedade de ordenação das tarefas (NAU; MUñOZ-AVILA; LOTEM, 1999) (NAU et al., 2001).

Em Nau et al. (2001) foi apresentado o algoritmo SHOP2. Esse é uma nova versão do algoritmo original que mantém a abordagem de planejar os passos na ordem em que serão executados (da esquerda para a direita), mas relaxa a restrição da ordenação de tarefas. SHOP2 permite que uma tarefa seja decomposta em um conjunto parcialmente

ordenado de sub-tarefas. Isto possibilita a criação de planos que intercalem sub-tarefas que são originarias de várias tarefas (NAU et al., 2001).

SHOP2 é quase totalmente compatível com SHOP, ou seja, é possível rodar bases de conhecimentos geradas para SHOP no novo algoritmo apenas fazendo algumas modificações sintáticas. A maior motivação para o desenvolvimento de SHOP2 foram os problemas gerados pela demasiada complexidade da base de conhecimento requerida por SHOP. Em certos problemas (como o Logística) era necessário escrever métodos e axiomas auxiliares para descrever conhecimento global do mundo. Isso acontecia devido ao requisito de ordenação total de tarefas. Com o desenvolvimento de SHOP2 o tamanho da base de conhecimento para o domínio logístico caiu para 26% do tamanho original. Isso não traz vantagens para todos os domínios. Um exemplo disso é o domínio Mundo de Blocos, onde, ao tentar modificar a descrição de domínio afim de obter alguma vantagem do relaxamento da ordenação total de tarefas a base de conhecimento acabou aumentando de tamanho. Mesmo rodando bases de conhecimento de SHOP em SHOP2 se consegue alguma vantagem. Isso é possível devido a algumas otimizações feitas no novo algoritmo (NAU et al., 2001).

SHOP2 é capaz de manipular domínios de planejamento que envolvam tempo. Em Nau et al. (2003) é apresentado o MTP (*Multi-Timeline Preprocessing*). O MTP consiste em uma técnica de tradução dos operadores de SHOP2 são pelo menos tão expressivos quanto as ações de nível 2 de PDDL, porém, SHOP2 não oferece explicitamente um suporte para as ações com duração (oferecidas pelo nível 3) e nem tem um mecanismo de avaliação para as ações com duração e para as ações concorrentes. Mesmo assim, SHOP2 tem poder de expressão para representar estas ações. Devido aos operadores de SHOP2 permitirem atribuir valores á variáveis e fazer operações numéricas sobre elas é possível manter a informação temporal, para isso basta traduzir o operador PDDL para um operador SHOP2 utilizando o algoritmo MTP. Em princípio essa técnica pode ser utilizada em qualquer planejador temporal, bastando que esse disponha de poder de expressão suficiente para isso.

Devido à SHOP ser um subconjunto de SHOP2 será assumido daqui para frente que os recursos descritos existem nos dois algoritmos, ressaltando as diferenças entre os algoritmos.

O algoritmo SHOP2 recebe como entrada um domínio e um problema de planejamento e retorna um plano que resolve esse problema. Em SHOP esse plano é uma sequência totalmente ordenada de operadores instanciados, enquanto em seu sucessor

é apenas parcialmente ordenada devido ao relaxamento do requisito de ordenação que existia no algoritmo original. O estado em SHOP2 é um conjunto de átomos lógicos instanciados e um conjunto de axiomas (NAU et al., 2003), (ILGHAMI, 2005).

Um predicado é identificado por seu nome e é associado à uma lista de termos. O termo pode ser qualquer variável, constante, número, lista de termos ou chamada para função. Uma lista de termos é uma lista na forma $(t_1, t_2, ..., t_n)$, onde, os elementos t_i são termos. Qualquer variável, tarefa, constante, predicado ou função é dita instanciada se todas suas variáveis estiverem instanciadas (ILGHAMI, 2005).

A sintaxe das estruturas que definem os componentes do domínio e do problema apresenta diversos elementos, esses são definidos como se segue:

- variáveis - pode ser qualquer símbolo cujo nome comece com ponto de interrogação;
- tarefa primitiva - pode ser qualquer símbolo cujo nome comece com ponto de exclamação;
- constante, predicado, função[1] e tarefa composta - qualquer símbolo cujo nome comece com uma letra ou sublinhado.

Variáveis em SHOP2 podem armazenar valores ou objetos. Em SHOP2 a atribuição tem a forma $(assign \quad VT)$, onde V é uma variável e T é um termo. O objetivo da atribuição é ligar o valor de T à variável V. O escopo das variáveis em SHOP2 é limitado aos métodos, operadores ou axiomas onde elas aparecem. Dessa forma duas variáveis homônimas não têm o mesmo significado em instâncias diferentes, mesmo se tratando do mesmo elemento de domínio. Esses objetos e valores também podem ser expressos por constantes. A constante especial *nil* representa uma lista vazia.

Uma chamada para função é uma expressão na forma $(call \quad ft_1t_2...t_n)$, onde, f é uma função interna de SHOP2 ou uma externa definida no ambiente, e t_i são termos que representam os argumentos da função. A função é instanciada quando sua chamada for substituída por SHOP2 pelo seu resultado.

Os problemas consistem em um estado inicial composto por átomos lógicos e uma lista de tarefas que devem ser realizadas. Em SHOP2 uma tarefa é descrita como uma lista na forma $([: immediate] \quad st_1t_2t_3...t_n)$, onde o símbolo S representa o nome da

[1]Na implementação JSHOP2 foi assumido que a função tem a mesma sintaxe do identificador na linguagem de programação Java. Aqui foi adotada a mesma utilizada pelas constantes e outros elementos de SHOP2

tarefa e os t_i são seus argumentos definidos como termos. As tarefas também podem ser primitivas. Essas são diferenciadas das compostas por ter um caractere de exclamação ("!") no início do nome S. As tarefas primitivas são resolvidas por operadores enquanto as não primitivas são reduzidas por métodos (NAU; MUñOZ-AVILA; LOTEM, 1999).

Uma tarefa sem o modificador :*immediate* é chamada tarefa ordinária. O objetivo desse modificador é dar alta prioridade para a tarefa (NAU et al., 2001). Uma lista de tarefas é uma lista na forma ([: *unordened*] $tl_1, tl_2, ..., tl_n$), onde, os elementos tl_i são listas de tarefas. O modificador : *unordened* especifica que não há nenhuma ordem entre as listas de tarefas. Ambas palavras-chave foram adicionadas em SHOP2, assim, quando omitido o modificador : *unordened* SHOP2 trata as listas de tarefas da forma que SHOP trataria. O modificador : *immediate* faz com que a tarefa seja priorizada quando, na lista de tarefas onde está inserida, não há outra tarefa que requisite ser executada antes desta. Se existirem duas tarefas usando o modificador, nessas condições, o comportamento dessas tarefas é indefinido.

O domínio é um conjunto de axiomas, métodos e operadores. Todos os elementos do domínio envolvem expressões lógicas que combinam os predicados de várias formas. É possível utilizar nas expressões lógicas conjunção, disjunção, negação, implicação, quantificação universal, atribuição e chamada a expressões[2]. Em última instância uma expressão lógica será um predicado (NAU; MUñOZ-AVILA; LOTEM, 1999) (ILGHAMI, 2005).

A conjunção tem a forma ([*and*] $[L_1 L_2 ... L_n]$), onde, a palavra chave *and* é opcional e os L_i são parâmetros que consistem em expressões lógicas, também opcionais. Quando os parâmetros são omitidos a expressão equivale ao valor *nil*. A disjunção é escrita da mesma forma que a conjunção, mas utiliza a palavra chave *or* e a os parâmetros são obrigatórios. A negação por sua vez tem a forma ($not \quad L$), onde, L é uma expressão lógica.

A implicação é semanticamente equivalente à expressão $\neg Y \cup Z$. Esta é escrita como ($imply \quad Y Z$), onde Y e Z são expressões lógicas. Desta forma Y deve ser instanciado antes da avaliação para que o resultado não se torne ambíguo.

O quantificador universal tem a forma ($forall \quad V Y Z$), onde, Y e Z são expressões e V é uma lista de variáveis em Y. O objetivo do quantificador é satisfazer essas variáveis para cada possível substituição de variável em V. Se Y^u é satisfeita, então Z^u também deve ser satisfeita no atual estado do mundo (NAU et al., 2003).

[2]Tem a mesma forma da chamada à função, mas semanticamente é falso para uma lista vazia ou nil e verdadeiro no caso contrário

Um axioma é uma expressão na forma (: $-aL_1L_2...L_n$), onde, a é a cabeça (representada por um átomo) e cada L_i é uma expressão lógica. Um axioma oferece várias maneiras de provar sua cabeça, caso alguma dessas seja satisfeita, a cabeça será considerada verdadeira e assim o axioma será verdadeiro.

Um operador é uma expressão (: *operator* $hPDAc$), onde, h é uma tarefa primitiva, P é uma pré condição lógica e D (lista de exclusão) e A (lista de adição) são listas de átomos. Essas listas somente contêm variáveis que estejam em h (NAU et. al., 1999) e c é um custo associado, cujo valor padrão é 1 (ILGHAMI, 2005). Por outro lado P pode ter variáveis que estão em h ou não, assim, deve ter uma condição que satisfaça P em determinado estado, em um dado ponto do processo de planejamento.

SHOP2 permite o uso de proteções para impedir que determinada condição seja excluída. Para isso é usado o predicado especial : *protection* nas listas de adição e de exclusão. Quando uma proteção para determinada condição é adicionada um contador associado a ela é incrementado e no caso contrário é decrementado. Qualquer condição somente pode ser excluída do estado do mundo quando esse contador tiver o valor zero (NAU et al., 2001).

O quantificador universal é também utilizado nas listas de adição e de exclusão dos operadores de SHOP2 afim de expressar um conjunto de efeitos em vez de uma expressão lógica. Este quantificador é escrito como (*forall* VYZ), onde, V é uma lista de variáveis em Y, Y é uma expressão e Z é uma lista de predicados que não tem variáveis que não estejam na cabeça h, na lista V e nas condições protegidas P (NAU; MUñOZ-AVILA; LOTEM, 1999) (ILGHAMI, 2005).

Um método é uma expressão na forma (: *method* hCT), onde, h é uma tarefa não primitiva, C é um conjunto de condição que devem ser atendidas pelo mundo para que o método possa ser aplicado e T é uma lista de tarefas para qual a tarefa t vai ser reduzida. Um método pode ser utilizado como uma estrutura se-então-senão de uma linguagem de programação procedural, bastando para isso, criar pares C e T para cada opção de decomposição. A decomposição escolhida será a primeira de cima para baixo onde a condição for avaliada como verdadeira. Caso não seja encontrado nenhum C verdadeiro, então, é escolhido outro método que reduz a tarefa t em questão, caso não haja nenhum disponível, não há redução para essa tarefa. A escolha entre os vários métodos que reduzem a tarefa t deve ser feita de forma não determinística.

Uma pré-condição lógica é uma expressão lógica. Em SHOP2 é possível aplicar modificadores a essa pré-condição de forma a ordenar as várias alternativas existentes

que satisfazem a ela de acordo com algum critério especificado na definição do método ou fazer que o planejador apenas considere a primeira das alternativas que satisfaça a pré-condição. Isto é conveniente para permitir que o planejador explore certas partes do domínio antes de outras. Esta técnica, a princípio, pode ser utilizada por qualquer planejador encadeado para frente.

Utilizando (: *first* L), onde, L é uma expressão lógica SHOP2 considera apenas o primeiro conjunto de ligações que satisfaça L. Outras ligações não serão consideradas mesmo que esta não leve a um plano válido.

A ordenação, por sua vez, é especificada na forma (: *sort-by* $v[f]L$), onde, v é uma variável, f é o nome do comparador e L é uma expressão lógica. O comparador f pode ser os internos oferecidos por SHOP2, ou um personalizado. No caso dos comparadores internos, são oferecidos dois, menor ($<$) ou maior ($>$). Caso o comparador for omitido SHOP2 assumirá o comparador menor.

3.2.1 JSHOP2

JSHOP2 é uma implementação de SHOP2 desenvolvida em Java. A maior parte das implementações de planejadores são feitas na linguagem LISP (incluindo SHOP2) ou C++. Essas implementações geralmente usam interpretadores para ler as especificações do domínio e do problema a fim de, posteriormente, fazer o planejamento. JSHOP2 se diferencia dos demais por não ser um interpretador, mas um compilador de descrições de domínios. Ou seja, JSHOP2 é um compilador de planejadores dependentes de domínios.

JSHOP2 consiste em um conjunto de modelos[3] de algoritmos e um analisador sintático que transforma as regras de sua linguagem (descrições de domínios e problemas) em um código fonte na linguagem Java. Este código consiste em um planejador que é uma instância dos modelos de algoritmos. A descrição do problema pode ser compilada separada da do domínio. Isto permite que o código Java feito para o domínio possa ser ligado a vários problemas.

O código, após ser compilado por Java, poderá ser usado para resolver o problema de planejamento rapidamente. Existem certas razões que levou Ilghami (2005) a justificar essa abordagem. Segundo ele existem certas otimizações que podem ser feitas se é conhecido de antemão detalhes de implementação, tais como, tamanhos de vetores. Outra razão alegada foi a facilidade de implementação de chamadas externas e procedimentos

[3] *Templates*

remotos (dentro do ambiente Java) e funções comparadoras.

Mas, infelizmente no caso dos jogos há duas complicações ao se tentar usar JSHOP2.

A primeira complicação diz respeito ao tempo de compilação do problema e ligação com o código do domínio. O domínio pode ter seu código gerado durante o tempo de desenvolvimento, mas os problemas a serem resolvidos são formulados dinamicamente nos jogos. A necessidade do problema ser gerado em tempo de simulação vem do fato desse ser baseado no estado atual do mundo, o que impossibilita a criação de bibliotecas de problemas para esta aplicação.

Desta forma seria necessário gerar o código Java referente ao problema, e em seguida compilar esse juntamente com o código do domínio obtendo um código objeto. Somente depois desse processo o planejador estaria apto a rodar. Isto provavelmente violaria o requisito de tempo real exigido durante a simulação.

A segunda limitação ocorre quando o jogo for escrito em uma linguagem que não seja Java. É inviável manter uma troca de mensagens, de forma eficiente, com a maquina virtual a fim de controlar o planejador que roda nela e assim solicitar e obter planos. Além disto, se desenvolvido desta forma, não seria possível parar a construção de um determinado plano para priorizar a construção de outro. Isto é necessário devido a muitas vezes ser desejável o poder de postergar o planejamento no caso de uma escassez de tempo de processamento disponível.

3.3 Planejadores automáticos vs. semi-automáticos

SHOP2 esteve entre os competidores inscritos na 3ª Competição Bienal de Planejadores (LONG; FOX, 2003). Esse é um algoritmo semi-automático, mas independente de domínio. Esse tipo de algoritmo permite ao modelador de domínios criar métodos HTN específicos para cada domínio e assim o planejador pode ser adaptado para obter vantagens de particularidades do domínio. O conhecimento dependente de domínio pode aumentar em muito o desempenho, isso pode significar a diferença entre o algoritmo semi-automático ser executado em tempo polinomial e o automático em exponencial para resolver o mesmo problema (NAU; MUñOZ-AVILA; LOTEM, 1999) (NAU et al., 2003).

SHOP2 (NAU et al., 2001) concorreu na categoria de semi-automáticos juntamente com outros dois sistemas - TLPLAN (BACCHUS; KABANZA, 2000) e TALPLANNER

(MAGNUSSON; KVARNSTRÖM, 2003). Esses concorrentes de SHOP2 utilizam uma tática de encadeamento para a frente associada com regras de controle escritas declarativamente em lógica temporal. Essas regras oferecem à TLPLAN e TALPLANNER o conhecimento dependente de domínio necessário para podar más escolhas e promover boas (NAU et al., 2003) (LONG; FOX, 2003).

Esses três planejadores resolveram, além dos problemas destinados exclusivamente aos semi-automáticos, também a maior parte dos destinados aos planejadores automáticos. SHOP2 aceitou problemas das classes STRIPS, numérica, numérica difícil, tempo simples, tempo e complexos (NAU et al., 2003).

Long e Fox (2003) compararam FF(HOFFMAN; NEBEL, 2000) (*Fast Forward*) e MIPS(EDELKAMP, 2003) (*Model-Checking Integrated Planning System*) (automáticos) com SHOP2 e TLPLAN (semi-automáticos). Para isto utilizaram os problemas dos domínios numérico difícil e satélites da competição. Embora os objetivos lógicos destes domínios sejam triviais (ex: cada local de observação deve estar no fim do plano), os problemas utilizam uma métrica complexa para avaliar o plano. Essa métrica utiliza informações coletadas pelo satélite durante a execução e gera os planos através de objetivos implícitos (LONG; FOX, 2003).

FF é um planejador automático baseado em busca com encadeamento para frente. Este planejador utiliza heurísticas que estimam a distância entre os objetivos e os estados afim de escolher caminhos dentro do espaço de estados. MIPS é um sistema de planejamento baseado em uma variedade de técnicas, mas no fundo é um verificador de modelos baseados em OBDDs (*Ordered Binary Decision Diagrams*). Esse é utilizado para gerar estados alcançáveis. O espaço de estados é compactado e são feitas buscas heurísticas baseadas em planos relaxados (LONG; FOX, 2003).

Estes planejadores foram selecionados para essa comparação devido a todos eles aceitarem resolver os problemas dos domínios numérico difícil e satélites, sendo que dentre os automáticos, apenas FF e MIPS aceitaram esses problemas. Nessa comparação fica evidente o aumento na velocidade de planejamento entre os algoritmos semi-automáticos e automáticos que mostram grande vantagem em termos de desempenho em favor da primeira categoria. Por outro lado, a produção da base de conhecimento para algoritmos semi-automáticos se torna mais complexa devido ao agregamento do conhecimento dependente de domínio. O teste mostrou que existe uma proximidade entre os resultados dos dois algoritmos semi-automáticos (Figura 11), isso sugere que ambos estão aplicando heurísticas similares e aparentemente produzem planos semi-ótimos.

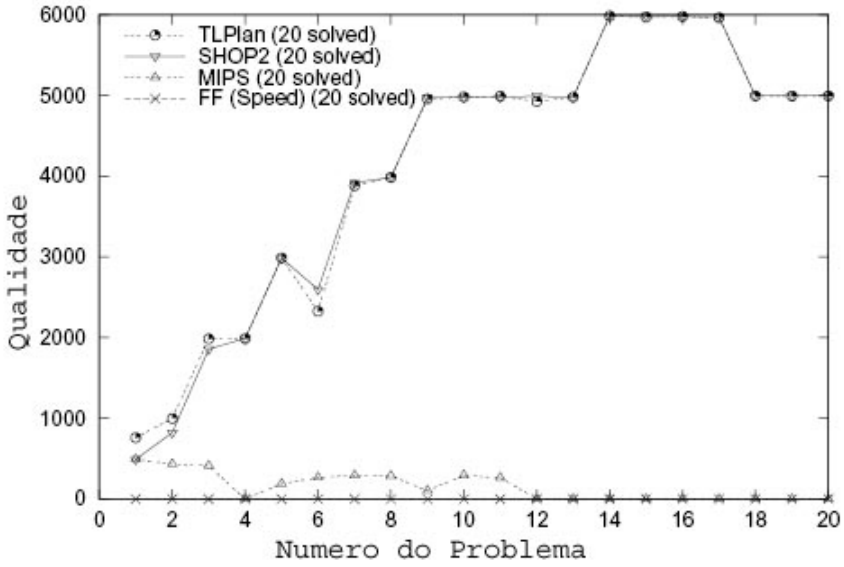

Figura 11: Gráfico comparativo entre os planejadores automáticos e semi-automáticos.
Fonte: Long e Fox (2003).

Para compilação desse conhecimento, nas competições, as regras da categoria de semi-automáticos alocam um determinado tempo para que os engenheiros de conhecimento possam reformular as descrições do domínio. Segundo Long e Fox (2003) existem alguns fatores que podem ser úteis na comparação qualitativa dos algoritmos de planejamento semi-automáticos. Entre esses fatores estão o tempo necessário para reformular o domínio, número de componentes da equipe e a experiência deles. Esses fatores não foram levados em conta até o momento, mas poderão ser explorados em futuras edições da competição (LONG; FOX, 2003).

TLPLAN ganhou o melhor prêmio da categoria devido a ter resolvido rapidamente todos os 894 problemas tentados por ele, tendo produzido planos de ótima qualidade. Por outro lado SHOP2 ganhou um prêmio na mesma categoria por ter tentado resolver todos os 904 problemas destinados aos planejadores semi-automáticos e resolvido a maior parte deles (899). Outro fato que contribuiu para a atribuição desse prêmio à SHOP2 foi a possibilidade desse de criar mais planos que qualquer um dos planejadores inscritos na competição. O prêmio destinado a SHOP2 foi um dos quatro melhores da competição. Os planos produzidos por SHOP2 são de boa qualidade e a velocidade do sistema de planejamento é bem competitiva (LONG; FOX, 2003) (NAU et al., 2003).

Em geral TLPLAN e TALPLANNER foram mais rápidos do que SHOP2 em quase todos os domínios, exceto no de satélites numérico difícil onde SHOP2 foi o mais rápido entre todos os planejadores. A diferença entre os três planejadores semi-automáticos

sempre se manteve em níveis polinomiais, talvez, pela base de conhecimento do domínio permitir que busquem a solução sem fazer muito *backtracking*. Todos foram extremamente mais velozes do que os planejadores automáticos (NAU et al., 2003).

3.4 Conclusões

Nesta seção foram abordados os principais métodos de planejamento, bem como as motivações para o uso desta tecnologia em jogos de computador.

Embora existam boas soluções utilizando sistemas de regras para fazer algum planejamento em jogos, a utilização de um planejador de propósito geral é uma solução cada vez mais atraente devido à demanda por técnicas mais eficientes e que tragam melhores resultados com o uso de bases de conhecimento menores.

Entre os algoritmos vistos, os que melhor se encaixam neste uso são os semi-automáticos baseados em redes de tarefas hierárquica. Estes algoritmos apresentam um tempo de execução curto em relação aos seus concorrentes automáticos.

Apesar do custo de desenvolvimento do domínio ser maior para os planejadores semi-automáticos, isto não representa um problema para os jogos devido aos domínios desses serem elaborados por especialistas.

Entre os algoritmos de planejamento baseados em HTN está SHOP2 que parece apropriado para o uso dos desenvolvedores de jogos. Este algoritmo oferece um tempo de processamento muito bom, associado à computação numérica e simbólica que são fundamentais para gerenciar os recursos de um agente para jogos.

Embora existam duas boas implementações deste algoritmo, infelizmente ambas não são apropriadas para o uso em jogos. Pela forma com que foi arquitetado JSHOP2 ofereceria problemas em tempo de execução do jogo por precisar compilar seus domínios e problemas com o uso de um compilador Java. A outra implementação, feita em LISP, acarretaria problemas de acoplamento com o código do motor de jogos, devido a dificuldades em implementar técnicas como balanceamento de carga (Seção 2.5).

Por isto, no próximo capítulo apresentaremos uma implementação que entendemos apropriada para esta finalidade.

4 Uma implementação de planejamento HTN apropriada para jogos

Neste capítulo apresentamos o *SHOP2 4 Games* que é uma implementação do algoritmo SHOP2 (Seção 3.2) com características desejáveis para seu uso em jogos de computador.

Nossa implementação foi baseada na de JSHOP2 (Seção 3.2.1), mas diferencia-se dela principalmente por ser interpretada e por permitir que vários problemas e domínios sejam explorados paralelamente pelo agente, cabendo a esse escalonar os recursos de processamento.

Outra diferença significativa da nossa abordagem é que o *software* foi desenvolvido em C++. Isto é mais adequado para a aplicação em jogos, pois, a grande maioria deles é produzida nesta linguagem. Além disto, a parte mais significativa dos compiladores atuais é capaz de importar funções de bibliotecas dinâmicas produzidas em C++, o que aumenta em muito a flexibilidade da nossa biblioteca.

Na construção de jogos de computador é imperativo que se respeite os requisitos de tempo real dos mesmos (Seção 2.1). Os jogos, principalmente os de simulação onde o jogador está imerso em um ambiente virtual, exigem que as respostas dos agentes sejam quase imediatas para não atrapalhar o ritmo do jogo. O planejamento, mesmo utilizando um algoritmo rápido como SHOP2, não é para ser usado o tempo todo, mas em situações muito especiais onde ele agregue valor às decisões do agente.

A maior deficiência encontrada na aplicação da abordagem usada em JSHOP2 na implementação de jogos está na compilação dos domínios. JSHOP2 depende da geração do domínio e do problema em código Java e posterior compilação em código objeto naquela linguagem para cada novo problema criado. Somente após este processo o código estará apto a ser executado e resolver aquele problema específico para o qual foi compilado.

Em nossa abordagem foram criados os moldes para domínios e problemas. Estes consistem na versão não instanciada dos domínios e problemas. Assim o código fonte

destes é lido dos arquivos fonte e guardado em listas que serão posteriormente adicionadas às listas de domínios e problemas disponíveis para instanciação. Desta forma, quando um molde para domínio ou problema é requisitado ele pode ser imediatamente instanciado e assim é economizado muito tempo não fazendo compilações.

Os moldes também podem ser guardados diretamente em arquivos na forma de sua estrutura de dados. Assim é possível economizar também o tempo de tradução dos arquivos fonte quando esses dados forem necessários. Esta representação também oferece como vantagem a proteção dos direitos autorais do domínio, por se tratar de uma forma de representação binária esta desobriga o autor de distribuir o código fonte que especifica o domínio.

Outra possibilidade permitida pelo nosso trabalho é formular o problema em tempo de execução. Assim, o agente pode criar um molde de problema e descrever, de acordo com sua situação atual, o problema que deve ser resolvido de acordo com determinado domínio. O agente para isso deverá descrever o estado inicial do planejamento e atribuir os objetivos que devem ser alcançados através de uma tarefa ou lista de tarefas de alto nível que será reduzida pelo algoritmo de planejamento. Essas tarefas devem ser definidas de acordo com as necessidades do agente.

Quando a formulação do problema é deixada a cargo do agente, o estado inicial é compilado à partir do estado atual do mundo. O estado inicial consiste em um conjunto de átomos que eram verdadeiros no mundo no início do planejamento e são relevantes para resolver o problema dentro de seu domínio. Esta relevância deve ser levantada pelo agente que irá gerar o problema a ser resolvido.

Para haver a instanciação devem existir um molde de problema (classe “TIFAIProblemScheme”) e um do domínio (classe “TIFAIDomainScheme”) correspondente que deverá ser resolvido.

Neste ponto basta selecionar o molde do problema, que já está associado à um domínio e executar a instanciação através de sua função membro “InitializeCode()”. A estrutura de dados do problema e do domínio passará a estar instanciada e esta função retornará um objeto da classe “TIFAIProblem” que não será mais um simples molde como era “TIFAIProblemScheme”, mas uma estrutura instanciada que poderá ser submetida ao planejador para execução.

Após a instanciação, deve ser criado um contexto (classe “TIFAISHOP2Context”) que servirá de plataforma de ligação entre os vários elementos do planejador tais como,

estado, domínio, problema, variáveis, etc. Chamando a função membro do contexto "StartPlanning()" o problema será processado e um plano será retornado, ou nulo se não existir plano para aquele problema.

Nossa abordagem permite que várias instâncias de problemas sejam resolvidas simultaneamente com o uso de *threads* pelo agente. Isto traz vantagens, pois a arquitetura de agentes pode gerenciar vários pedidos de resolução de problemas e colocá-los para rodar com prioridades diferentes. Há duas restrições para o uso de paralelismo nesta implementação e são essas o *parser* e o mecanismo de instanciação. Para assegurar que o código de leitura e instanciação de domínios e problemas não seja executado simultaneamente por duas *threads* foram utilizados semáforos de proteção para estas duas seções críticas. Mas como a leitura dos arquivos acontece fora do tempo de simulação do jogo e o processo de instanciação é rápido isto não irá acarretar problemas.

O plano retornado pela instância de problema processada contém uma sequência de objetos do tipo "TIFAIPredicate" que representam as cabeças das tarefas primitivas que se executadas nessa ordem resolvem o problema.

Nas próximas seções apresentaremos mais detalhes sobre o material utilizado para o desenvolvimento da implementação, convenções e diagrama componente e de classes. O diagrama componente dará uma visão geral da interação entre o código do usuário e o da biblioteca. A estrutura de classes, tanto a de tempo de compilação dos dados do domínio e do problema quanto à de tempo de execução do planejamento, dará uma visão específica do funcionamento da biblioteca.

4.1 Ferramental e convenções

Um programa utilizando a biblioteca que implementamos foi rodado tanto no Debian GNU/Linux quanto no Microsoft Windows. Em geral nosso programa requer para ser compilado um compilador C++ equipado com a biblioteca STL (*Standard Template Library*), um criador de analisadores léxicos *Flex*, e um criador de analisadores sintáticos *Bison*.

Para compilar a versão Windows foram usados os seguintes programas utilitários:

- Compilador C++ GCC 3.4.2 (mingw-especial);
- Bison 1.875;

- Flex 2.5.4;
- Code::Blocks v1.0rc2 (desenvolvimento do código);
- ArgoUML 0.2 (diagramas de classes);
- Dia 0.94 (diagrama de componentes).

Para compilar a versão para o sistema Debian Linux foram usados:

- Compilador C++ GCC 4.0.3 para Linux (Debian 4.0.2-5);
- Bison 2.1;
- Flex 2.5.31.

Afim de assegurar a qualidade do código do programa e a sua portabilidade foram tomadas algumas medidas que são descritas a seguir.

Nossa biblioteca foi dividida em três pacotes, são esses o do planejador, o do *parser* e o do sistema. Cada um destes tem um espaço de nomes e prefixos próprios.

O planejador está no espaço de nomes "NMSPC_IFAIEngine" e suas classes começam com o prefixo "TIFAI", no espaço de nomes "NMSPC_NovilinguaCompiler" está o *parser* e esse tem apenas uma interface simples de algumas funções sem prefixo e "NMSPC_SYSTEM" contém classes de estruturas de dados que são usadas pelos outros módulos tais como vetores, listas, tratamento de exceções, depuração, etc. As classes e funções do espaço de nomes "NMSPC_SYSTEM" levam o prefixo "t".

Existem também classes com o sufixo "Scheme" que identificam classes que representam problema, domínio e seus elementos na forma não instanciada e são usadas em tempo de compilação. Estas classes estão alocadas dentro do espaço de nomes "NMSPC_IFAIEngine". As demais classes deste espaço de nomes, sem sufixo, são usadas em tempo de execução do planejamento ou em ambos os tempos.

Os tipos básicos de dados da linguagem também tiveram seus nomes modificados para esta implementação recebendo o prefixo "IF" antes do nome original. Isto facilita no caso de haver necessidade de estabelecer uma nova semântica para um determinado tipo de dados em alguma plataforma ao portar o programa.

Como optamos por usar a biblioteca STL achamos prudente encapsular suas classes utilizadas na implementação em novas classes com prefixo "t", assim se em alguma

plataforma ocorrer problemas de transporte da aplicação, basta fazer os devidos consertos na classe correspondente, sem afetar todo o programa. Isto foi útil, por exemplo, para a re-implementação do método "string::erase(start, num)" que apresentava problemas na versão da STL utilizada pelo compilador no Microsoft Windows.

Em nossa implementação cada classe tem o seu próprio arquivo de cabeçalho (extensão ".h") e esse tem o seu respectivo arquivo fonte (extensão ".cpp"). No arquivo de cabeçalho é declarada a classe e as dependências desta, enquanto no fonte são implentados seus métodos. Para resolver os problemas de dependências cíclicas entre as classes adotamos a seguinte estratégia:

- Apenas há declarações usando outros tipos no arquivo cabeçalho e não uso de suas instâncias;
- É feito o *forward* do tipo que foi declarado na classe se apresentar problemas de tipo não encontrado durante a compilação;
- Se na compilação do arquivo fonte houver problemas por causa de tipo incompleto, inclua nesse arquivo o cabeçalho correspondente à classe que está incompleta.

Os arquivos fonte e de cabeçalhos, exceto os do *parser*, tem o seu nome iniciado por "cs" seguido do nome da classe sem prefixos. Os arquivos que pertencem ao espaço de nomes "NMSPC_IFAIEngine" são diferenciados dos demais pelo prefixo "csai".

Todos os arquivos cabeçalhos de cada uma das bibliotecas estão referenciados em um arquivo mestre daquela biblioteca juntamente com a inclusão dos arquivos necessários para o seu funcionamento. Isto torna prático o uso da biblioteca tendo de haver apenas a inclusão de um único cabeçalho no programa usuário. Os arquivos mestres para os pacotes do planejador, do *parser* e o do sistema são, respectivamente, "csaiengine.h", "novilingua.h" e "cssystem.h".

4.2 Interação entre componentes

O diagrama da Figura 12 mostra o relacionamento entre o código do usuário e o de nossa biblioteca. Neste aparecem como componentes o código do usuário e o do *parser*. Estes fazem solicitação de serviços para os nodos planejador, disco e analisador. Além disso, são mostradas as interações destes componentes com as três classes mais importantes do ponto de vista do usuário, ou seja, "TIFAISHOP2", "TIFAIProblemScheme" e

"TIFAISHOP2Context".

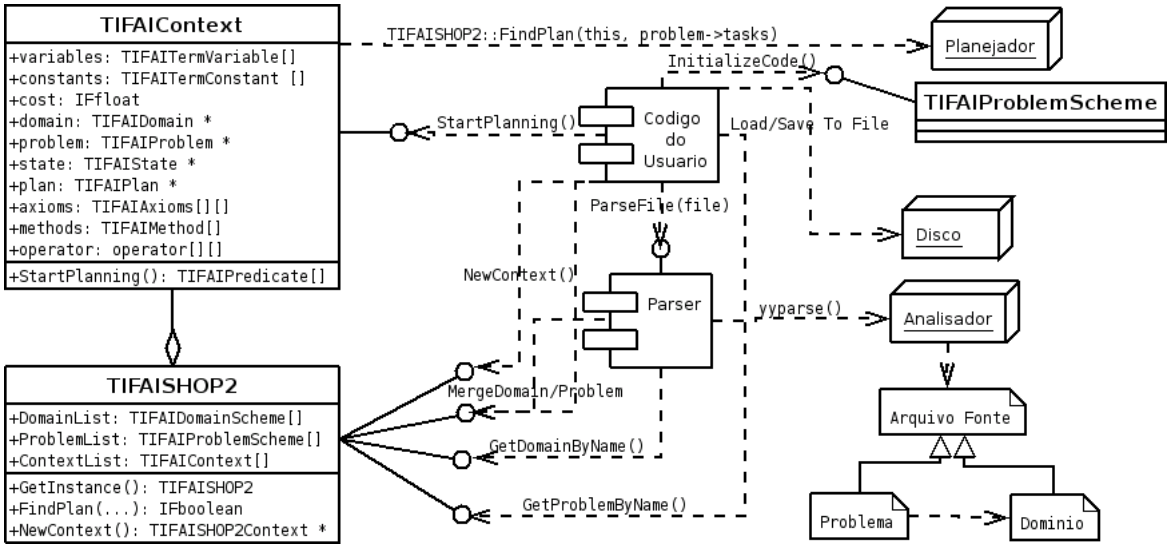

Figura 12: Diagrama de componentes com a visão do usuário.

A Tabela 8 mostra um exemplo de código utilizado para ler o domínio *logistics* e um problema resolvível com este. Identificaremos na descrição a seguir do diagrama as linhas de código correspondentes a cada ação.

O analisador implementa praticamente a mesma linguagem de JSHOP2, mas com algumas diferenças sintáticas. Aqui por conveniência colocamos o analisador como o processamento do *parser* e de problemas e esse como a interface com o usuário. O compilador inicia o processamento do arquivo fonte com a chamada à função "ParseFile()" (linhas 1 e 3) lotada no espaço de nomes "NMSPC_NovilinguaCompiler", que entre outras coisas ativa a função "yyparse()" do YACC (*Yet Another Compiler-Compiler*).

Existe uma dependência por parte do problema quanto a existência de um domínio correspondente em memória. Desta forma, caso não exista o domínio, será retornada uma exceção tanto na leitura do problema quanto na instanciação dele.

Após terminada a leitura, o *parser* coloca os moldes de domínios e de problemas em listas que podem ser acessadas pelo código do usuário. Opcionalmente estes moldes podem ser salvos em arquivos binários em disco através da função "SaveToStream()" e carregados posteriormente através de "LoadFromStream()".

Em seguida o problema deve ser selecionado entre os disponíveis. Automaticamente este irá selecionar o domínio correspondente e estarão prontos para serem instanciados.

Os moldes de domínio e de problema são colocados em listas pelo usuário através do método “MergeDomain()” e “MergeProblem” respectivamente (linhas 2 e 4). Estes podem ser selecionados através dos métodos “GetDomainByName()” e “GetProblemByName()” (linha 5). Os métodos que manipulam as listas de moldes disponíveis, bem como as referidas listas, são membros estáticos da classe “TIFAISHOP2”. Desta forma existe apenas uma lista de cada por instância do programa carregada em memória.

```
1       NMSPC_NovilinguaCompiler::ParseFile("logistics");
2       TIFAISHOP2::MergeDomains(NMSPC_NovilinguaCompiler::GetDomainSchemeList());
3       NMSPC_NovilinguaCompiler::ParseFile("logistics_problem");
4       TIFAISHOP2::MergeProblems(NMSPC_NovilinguaCompiler::GetProblemSchemeList());
5       TIFAIProblemScheme * pbs = TIFAISHOP2::GetProblemByName("logistics_problem");
6           TIFAIProblem * pb = pbs->InitializeCode();
7       TIFAISHOP2Context * context = TIFAISHOP2::NewContext(pb);
8       tlist <TIFAIPredicate *> * plan = context->StartPlanning();
9       if (plan){
10                  tlist<TIFAIPredicate*>::iterator e = plan->begin();
11                  while (e != plan->end()){
12                        printf("\%d", (*e)->Head.get());
13                e++;
14            }
15      }
```

Quadro 8: Exemplo de código para uso da biblioteca.

Em seguida o problema selecionado é instanciado através da função membro “InitializeCode()” (linha 6). A classe “TIFAIProblemScheme” se encarrega de instanciar toda a estrutura, incluindo o domínio que resolve aquele problema, e retorna um objeto do tipo “TIFAIProblem”.

O próximo passo é criar um objeto do tipo contexto (linha 7) que recebe como parâmetro para sua construção o objeto que representa o problema. Para criar o objeto contexto utiliza-se o construtor da classe “TIFAISHOP2Context”. O contexto é a ligação de todos os elementos de um domínio, com um problema, com o plano que será gerado e com o estado atual do planejamento. Este funciona como uma solicitação para processamento de planejamento.

De posse do contexto basta chamar o método “StartPlanning()” (linha 8) e o processo de planejamento é iniciado. Este procedimento passa o contexto atual, juntamente com a lista de tarefas que deverá ser realizada para o algoritmo de planejamento SHOP2 através da chamada “TIFAISHOP2::FindPlan()”. Se existir um plano que resolva o problema será retornada uma lista contendo uma sequência de cabeças de tarefas primitivas que representam os passos do plano, caso contrário retornará nulo. Esta lista, se existir, pode ser percorrida por um iterador do tipo “tlist <TIFAIPredicate*>” (linhas de 9 à 14).

4.3 Estrutura de Classes

A estrutura desta implementação é dividida em duas fases, ou seja, de tempo de compilação e de tempo de execução. Existem classes que são exclusivas de um dos "tempos" e outras participam tanto da compilação dos dados quanto da execução do planejador. O diagrama da Figura 13 representa o tempo de compilação, ou seja, a estrutura na forma não instanciada enquanto o da Figura 14 representa a estrutura depois da instanciação, ou seja, em tempo de execução.

Fazem parte da especificação do domínio o axioma, o operador e o método. Estes são considerados os três tipos de elementos do domínio.

Um axioma fornece várias maneiras de provar sua cabeça, assim basicamente é formado por um conjunto de pré-condições. No momento da prova do axioma a lista de pré-condições, como padrão, é varrida de frente para trás e se encontrada uma pré-condição verdadeira o axioma é provado verdadeiro, caso contrário, se achar o final da lista sem encontrar uma pré-condição verdadeira é provado falso.

Um método faz a decomposição de tarefas compostas. Ou seja, um determinado método é escolhido tendo como base a sua cabeça, que consiste em uma tarefa composta, e este dá várias maneiras de fazer sua decomposição. A implementação do método consiste em uma lista de pré-condições associada a uma lista de tarefas que devem substituir a tarefa composta no caso da pré-condição ser verdadeira. Assim como nos axiomas, as ramificações do método são avaliadas, como padrão, do início da lista em diante e é escolhida a primeira em que a pré-condição casar com o estado atual do mundo.

Um operador tem como função realizar tarefas primitivas. Basicamente o operador tem uma cabeça que consiste em uma tarefa primitiva que deve ser realizada, uma pré-condição que deve ser atendida pelo estado atual para que o operador possa ser utilizado e duas listas, uma de adição e uma de exclusão de átomos. Estas listas são as responsáveis pelas mudanças no estado do mundo. Também é nessas listas que são determinadas as proteções dos átomos que não devem ser modificados em determinado momento. Todo operador tem um custo associado, sendo esse definido como um termo.

As pré-condições existem em tempo de execução e servem para validar o estado atual ou substituição e realização de tarefas. Estas são geradas a partir das expressões lógicas ("TIFAILogicalExpression") do tempo de compilação no momento da instanciação. As pré-condições também podem ter sua ordem de avaliação modificada de acordo com uma função pré-definida pelo especificador do domínio. Os comparadores maior e menor

estão definidos para fazer esta ordenação através da classe “TIFAIComparator”.

Uma expressão lógica contém uma expressão e uma função de ordenação que deve ser utilizada para ordenar as pré-condições.

Uma expressão (“TIFAIExpression”) é uma classe abstrata que define formas de combinar seus termos e dar-lhes uma semântica. Esta pode ter vários formatos, são esses, conjunção, disjunção, negação, *forall*, *nil*, atribuição, chamada a expressão e átomo. As variações conjunção e disjunção são listas, muitas vezes recursivas, que contêm outras expressões e aplicam respectivamente a semântica “e” e “ou” sobre estas listas. O operador “*not*” e o comando “*forall*” implementam respectivamente a negação e a quantificação universal, bem como “*nil*” indica uma lista vazia. A chamada a termo é semelhante à chamada à função, diferenciando-se apenas por retornar falso para uma lista de termos vazia e verdadeiro para o caso contrário. Uma atribuição em SHOP2 torna possível atribuir valores ou objetos a uma variável, esta operação liga a variável a um termo. A expressão atômica por sua vez representa os objetos simbólicos nas expressões.

A classe abstrata “TIFAITerm” define os terminais de uma expressão. Estes terminais podem ser do tipo constante, variável, número, chamada à função ou lista de termos. Constantes e variáveis quando instanciadas são referenciadas em um vetor de acordo com seu índice em sua respectiva tabela, assim quando há a instanciação o procedimento “InitializeCode()” retorna a instância naquela posição do vetor. No caso de uma chamada a função é retornado pelo procedimento de inicialização um termo do tipo número com o resultado da chamada. A avaliação de funções é feita juntamente com a classe “TIFAICalculate” e que implementa dois tipos de função - as internas do SHOP2 e as externas definidas pelo usuário. A lista de termos é utilizada pela guardar os parâmetros de predicados (classe “TIFAIPredicate”) que representam os átomos em SHOP2.

As tarefas são alocadas em listas e estas listas podem conter outras listas de tarefas. As listas de tarefas são definidas na classe “TIFAITaskList” e contém ou uma lista de listas de tarefas ou uma tarefa atômica que é definida por “TIFAITaskAtom”. As listas de tarefas por padrão são ordenadas, mas SHOP2 permite que uma lista possa ser tratada como não ordenada. Se uma lista estiver marcada como não ordenada e ela contiver uma tarefa atômica marcada como imediata, esta tarefa será a próxima a ser escolhida. A Tabela 9 mostra o código que percorre as listas obtendo o nome das listas atômicas.

Os objetos da classe (“TIAIDelAddElement”) contêm duas listas, são essas a de

```
1   IFvoid TaskValidate(TIFAITaskList * tl){
2       if (tl->SubTasks.get() == NULL){
3            cout<<Tarefa atômica Labels[tl->Task.get()->Head.get()->Head.get()];
4       }else{
5            for (IFuint i = 0; i < tl->SubTasks.get()->size(); i++)
6                 TaskValidate((*tl->SubTasks.get())[i]);
7       }
8   }
```

Quadro 9: Exemplo usando a definição recursiva das listas de tarefas.

adição e a de exclusão. Estas listas contém, respectivamente, os elementos que serão incluídos e os que serão retirados do estado atual mundo quando o operador associado a elas for aplicado.

Aquela é uma classe abstrata que é base para quatro tipos de elemento, ou seja, variável, *forall*, atômico e átomo de proteção. O elemento atômico representa os átomos que serão incluídos ou excluídos no mundo e o átomo de proteção associa um contador ("TIFAINumberedPredicate") que impede que o um determinado átomo seja excluído até que todos os pedidos de proteção sejam revogados.

O tipo de elemento variável define se uma lista é uma lista de fato, ou uma variável que aponta para uma determinada lista através de seu índice. O operador de quantificação universal também está disponível para os elementos destas listas, ou seja, através de *forall* é possível definir as condições para adicionar átomos ao mundo baseada numa fórmula.

O estado atual é definido por um objeto da classe "TIFAIState". O estado consiste em três listas, uma de átomos, uma de axiomas e uma de átomos protegidos. A lista de átomos guarda o que é verdadeiro no mundo em um dado instante de tempo.

A descrição do problema consiste em uma lista de átomos que representam o que é verdade no estado inicial do mundo e uma lista de tarefas a serem realizadas. Esta lista está definida da mesma forma que é usada na descrição do método e é mostrada na Tabela 9.

A forma não instanciada dos domínios, elementos de domínios e problemas são baseadas em esquemas. Esses esquemas são convertidos no momento da instanciação para a forma instanciada e isto representa a transição entre o diagrama da Figura 13 para o da Figura 14.

Por conter as variáveis de SHOP2 instanciadas os objetos das classe "TIFAIDomain" e a "TIFAIProblem" precisaram ser definidos como escopos. Assim foi definida uma classe comum para ambas, ou seja, o tipo "TIFAIScope". A definição deste escopo facilitou vários detalhes de implementação do processo de instanciação, dentre eles as

referências dos termos dos tipos variável e constante para o domínio ou problema a que pertencem.

4.4 Conclusões

Neste capítulo foi apresentada *SHOP2 4 Games*, uma biblioteca que implementa o algoritmo de planejamento SHOP2 para ser acoplado em um motor de IA (Seção 2.6). Embora SHOP2 tenha muitas qualidades, ele é um algoritmo bastante complexo e não é viável, assim como acontece com outras técnicas, codificá-lo a cada novo projeto de jogo. Em nossa implementação foi adotada a linguagem C++ que permite construir códigos portáveis entre plataformas e com grande flexibilidade quanto à ligação com códigos gerados por outras linguagens.

Para a construção desta biblioteca foi observada a modularidade da mesma, bem como foram adotadas abordagens e ferramentas que permitam levar rapidamente o programa para outras plataformas. Basicamente esta depende apenas de uma implementação da biblioteca STL, um compilador C++ e um gerador de analisadores sintáticos para ser compilada. Todas estas ferramentas são amplamente disponíveis nas principais plataformas de computação.

Esta implementação diferencia-se das outras de SHOP2 por permitir um melhor escalonamento dos trabalhos de planejamento por parte dos agentes. Utilizando nossa implementação é possível iniciar vários planejamentos simultâneos graças aos contextos de planejamento. Desta forma se torna possível implementar nível de detalhe (Seção 2.4) e balanceamento de carga (Seção 2.5).

Depois do fim do processo de planejamento e do algoritmo SHOP2 entregar o plano, esse deverá passar pelo executor de planos. O executor faz parte da implementação do motor de IA e implementa para cada tarefa primitiva do plano uma ação que pode ser tanto a execução de uma animação quanto o comando de inicialização para que o problema seja resolvido em mais um nível mais baixo por alguma outra técnica como uma daquelas descritas na Seção 2.7.

No próximo capítulo são apresentados os experimentos que fizemos para aferir a eficiência de nossa implementação e a sua utilidade para jogos.

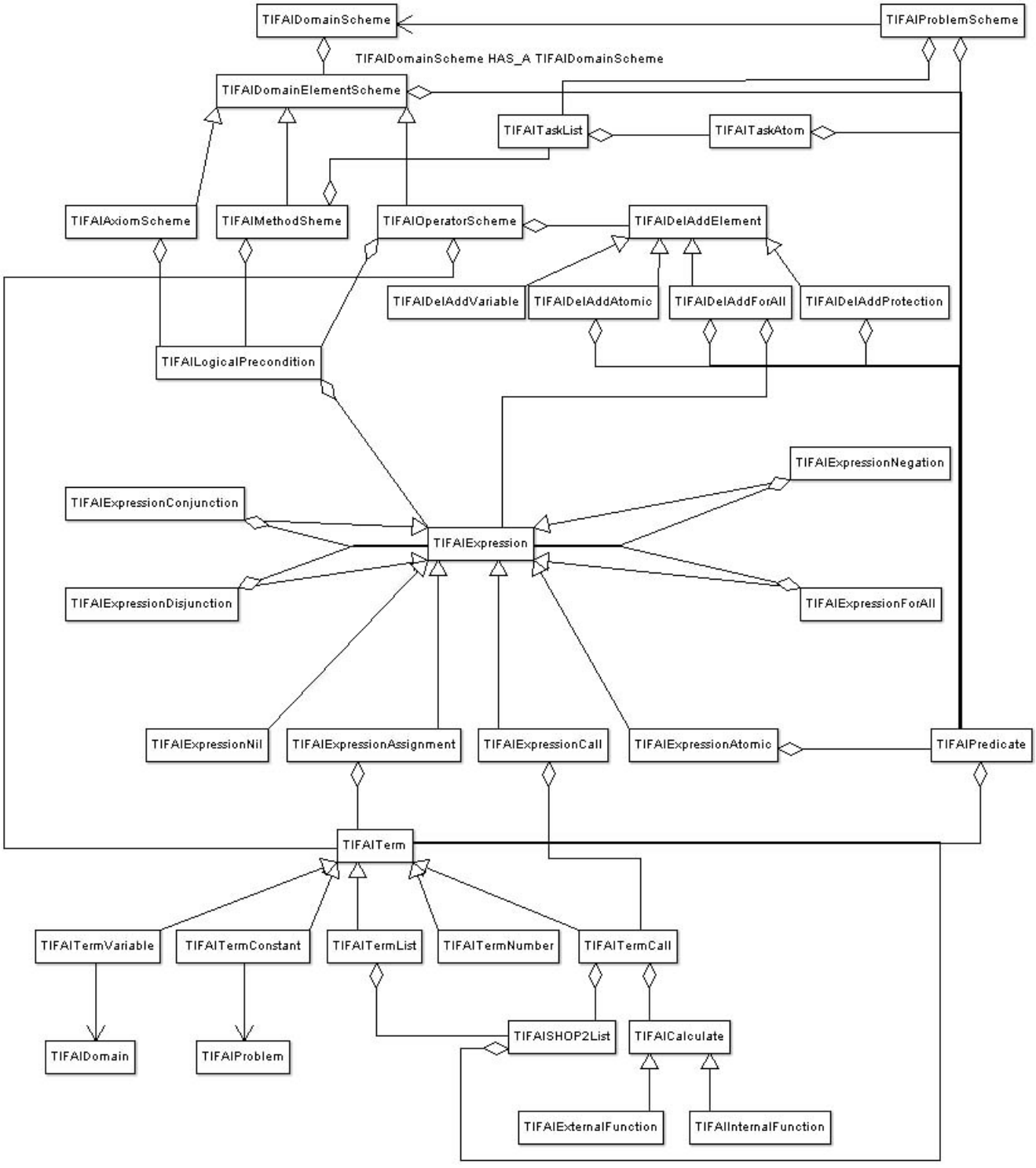

Figura 13: Estrutura de tempo de compilação.

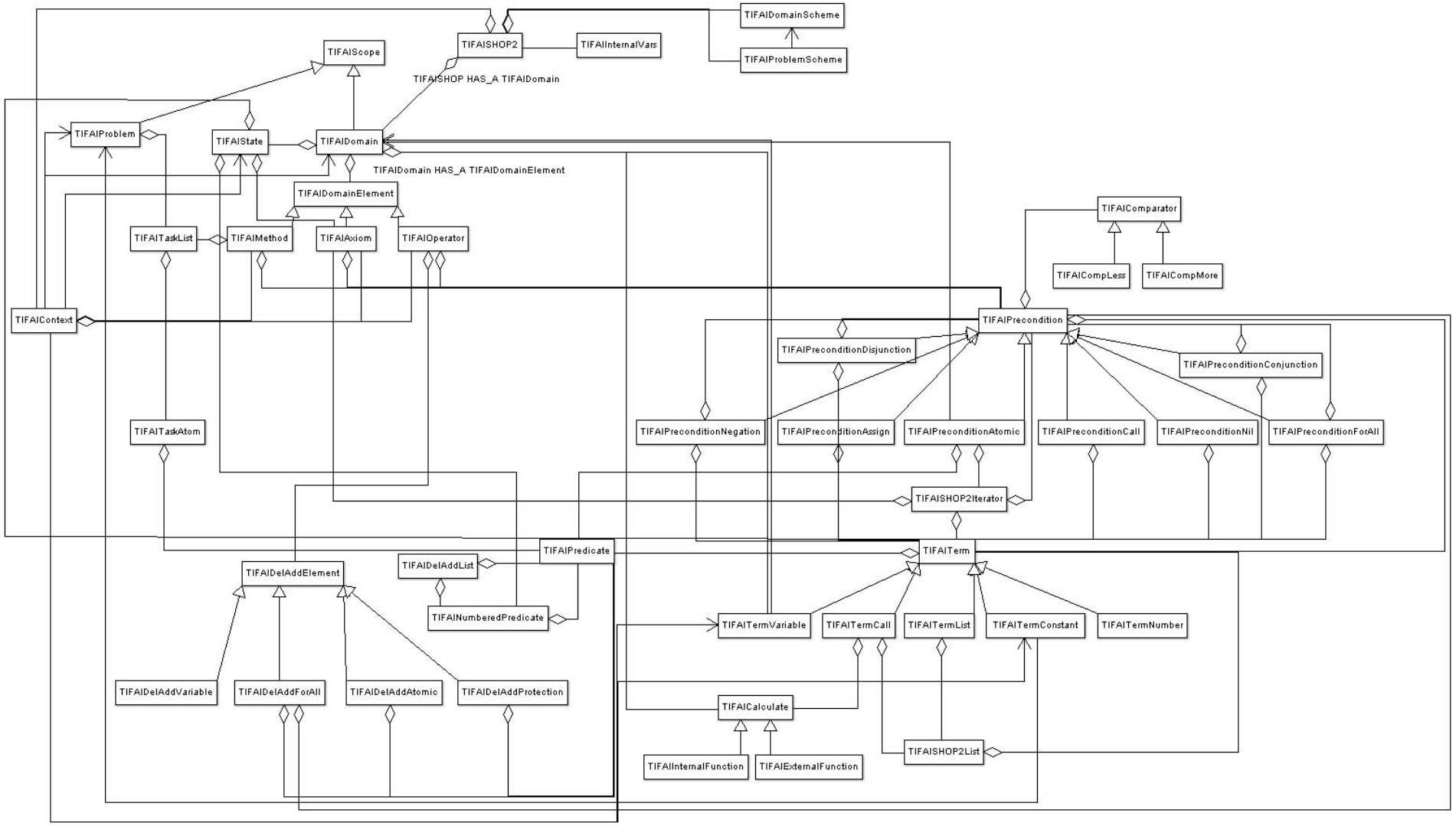

Figura 14: Estrutura de tempo de execução.

5 Experimentos

Neste capítulo é demonstrado como domínios para resolver dois problemas comuns em jogos podem ser especificados em SHOP2. Aqui também fica evidente a eficiência de SHOP2 para solucionar problemas em tempo hábil quando aplicado em jogos. Por fim são comparados nossos resultados aos de JSHOP2 (ILGHAMI, 2005) nesta aplicação.

Afim de ilustrar a especificação de domínios e problemas em SHOP2 retornamos ao exemplo do motorista automático iniciado na Seção 2.7, mas em um nível bem mais alto de abstração do que o apresentado naquela seção. Este exemplo é usado agora para demonstrar o uso de SHOP2, em especial o uso de nossa biblioteca. Para isto adotamos o domínio *travel* apresentado em Nau, Muñoz-Avila e Lotem (1999).

Neste capítulo também comparamos o domínio *travel* com o domínio *logistics*. O domínio *logistics* trata problemas de transporte de objetos entre dois pontos. Geralmente um veículo tem uma capacidade máxima de transporte e o ator precisa determinar as melhores maneiras que podem ser utilizadas para transportar seus objetos.

O domínio *travel*, por sua vez, trata da escolha de um meio de transporte por um agente que não é motorista.

O ator principal pode escolher entre ir à pé, de táxi ou de ônibus ao centro, ao subúrbio ou ao parque. A decisão de "como" ir é tomada pelo agente baseando-se nas tarefas que devem ser realizadas por ele. Alguns fatos também influenciam na decisão do agente, entre eles estão o clima (se está bom ou ruim) e a quantia de dinheiro que o ator tem para gastar na viagem.

O custo da viagem de ônibus é de $1.00 e a de táxi custa $1.50, no segundo caso acrescido de $1.00 por quilômetro rodado. Embora a viagem à pé seja gratuita, esta é condicionada ao clima. Se o clima estiver ruim, então a distância que ele pode caminhar é bem menor do que a que poderia num dia de sol.

O agente interage com outros atores no mundo afim de realizar seus objetivos.

Esta interação é realizada em nível de arquitetura de agentes e não é coberta por este trabalho. Cada agente segue os planos especificados pelos domínios que ditam seu comportamento. Dois casos de interação são evidentes neste domínio.

No primeiro caso é quando o agente toma um táxi. Neste o motorista automático do táxi está em um estado de espera por um plano que deve ser entregue pelo passageiro. A partir do recebimento deste plano o motorista deve aplicar uma técnica de busca de caminhos (BUCKLAND, 2005) para escolher a melhor rota a ser seguida para fazer a ligação entre os dois pontos. A espera pelo passageiro pode ser implementada como uma máquina de estados e o comportamento do motorista em tráfego é descrito pela máquina de estados da Seção 2.7.3.

O segundo caso é quando o veículo escolhido é o ônibus. Neste, o motorista automático de ônibus segue indefinidamente um plano pré-fixado, ou seja, independente da vontade do ator principal os ônibus vão circular, cabendo a esse apenas a decisão de usar o transporte ou não. Neste caso a rota do veículo é pré-definida e muda somente por motivo de força maior, quando deve haver um re-planejamento do caminho com o algoritmo de busca de caminhos. O tráfego deste motorista também é governado pela máquina de estados da Seção 2.7.3.

Nesta seção também adotamos uma variação do domínio *logistics* disponível entre os exemplos de domínios que acompanham JSHOP2 e esse trata os problemas relacionados ao transporte de pacotes utilizando caminhões e aviões. Neste domínio o plano determina como o veiculo deve ser carregado e os deslocamentos que este deverá efetuar. Na execução do plano em mais baixo nível o trafego dos caminhões e aviões pode ser governado por máquinas de estado (Seção 2.7.3) de forma semelhante a utilizada nos veículos do domínio *travel*.

A metodologia escolhida para comparar nossa abordagem com JSHOP2 é considerar o caso em que vários problemas são compilados sequencialmente utilizando o mesmo domínio. Neste caso, utilizando JSHOP2, seria necessário fazer n compilações do código do domínio ligado ao código de cada um dos n problemas. Isto causa uma sobrecarga muito grande na linha de montagem de planos, tomando na maioria das vezes mais tempo para fazer a compilação do planejador do que esse utiliza para fazer o planejamento. O tempo total usado pelas implementações que compreende a leitura do domínio e do problema, a carga destes e a execução do planejamento também é considerada para fins de comparação entre as abordagens.

5.1 Especificação do domínio *travel*

O quadro 10 mostra a estrutura da definição do domínio *travel* em SHOP2. Este é composto por um nome e uma coleção de axiomas, métodos e operadores.

```
1   (defdomain travel
2      (
3          ; Axiomas
4          ; Métodos
5          ; Operadores
6      )
7   )
```

Quadro 10: Codificação do domínio *travel*.

O axioma em SHOP2 é uma estrutura que oferece vários caminhos que podem provar sua cabeça. Cada um destes caminhos é representado por um conjunto de expressões. Se todas as expressões de um caminho forem verdadeiras, este é verdadeiro e consequentemente o axioma é verdadeiro e sua cabeça é tida como válida. No Quadro 11 são mostrados os dois axiomas implementados neste domínio e esses são descritos à seguir:

- "distancia_caminhada" (da linha 1 até a 11) - Verifica se o agente pode ir caminhando, caso as condições sejam as desejáveis para este tipo de transporte o axioma se torna verdadeiro. No caso, existem duas possibilidades do ator ir á pé, ou seja, com o clima bom ele pode andar até 3 km (linha 2), caso contrário apenas meio quilômetro (linha 7);
- "tem_dinheiro_para_taxi" (da linha 12 até a 17) - Este axioma verifica se o agente tem dinheiro suficiente para percorrer determinado trecho do caminho de táxi. Para isto testa o valor total da corrida com a quantia que o ator tem de dinheiro, se esta quantia for maior ou igual ao custo da viagem o axioma é verdadeiro.

```
1       (:- (distancia_caminhada ?u ?v)
2             (
3                 (tempo_esta bom)
4                 (distancia ?u ?v ?w)
5                 (call <= ?w 3)
6             )
7             (
8                 (distancia ?u ?v ?w)
9                 (call <= ?w 0.5)
10            )
11      )
12      (:- (tem_dinheiro_para_taxi ?distancia)
13            (
14                (tem_dinheiro ?m)
15                (call >= ?m (call + 1.5 ?distancia))
16            )
17      )
```

Quadro 11: Codificação dos axiomas do domínio *travel*.

```
1       (:method (pague_motorista ?taxa)
2               (
3                   (tem_dinheiro ?m)
4                   (call >= ?m ?taxa)
5               )
6               (
7                   (!atualiza_dinheiro ?m (call - ?m ?taxa))
8               )
9       )
10
11      (:method (viaje_ate ?q)
12              (
13                  (at ?p)
14                  (distancia_caminhada ?p ?q)
15              )
16              {
17                  (!caminhe ?p ?q)
18              )
19      )
20
21      (:method (viaje_ate ?y)
22              (
23                          (at ?x)
24                  (no_ponto_taxi ?t ?x)
25                  (distancia ?x ?y ?d)
26                  (tem_dinheiro_para_taxi ?d)
27              )
28              (
29                  (!suba ?t ?x) (!va_de ?t ?x ?y)
30                  (pague_motorista (call + 1.50 ?d))
31              )
32
33              (
34                  (at ?x)
35                  (rota_onibus ?onibus ?x ?py)
36              )
37              (
38                  (!espere_por ?onibus ?x)
39                  (pague_motorista 1.00)
40                  (!va_de ?onibus ?x ?y)
41              )
42      )
```

Quadro 12: Codificação dos métodos do domínio *travel*.

Os métodos descrevem as formas possíveis de decompor as tarefas compostas. Estas estão em um nível maior de abstração do que as tarefas primitivas e podem ser decompostas em outras tarefas compostas ou primitivas. Os métodos do domínio *travel* (Quadro 12) são descritos à seguir:

- "pague_motorista" (da linha 1 até a 9) - Esta tarefa é decomposta em uma tarefa primitiva "!atualiza_dinheiro", tendo como pré-condição para essa substituição o agente disponibilizar a quantia necessária para pagar o transporte;

- "viaje_ate" (da linha 11 até a 19) - A implementação de "viaje_ate" é dividida em dois métodos. Ao contrário do que acontece no caso dos operadores que devem ser únicos para cada tarefa primitiva, os métodos podem ter várias versões para uma determinada tarefa composta em um mesmo domínio. Durante o planejamento, a ordem em que estas versões são escolhidas é definida ao acaso. Na versão deste método que inicia na linha 11 do código do Quadro 12 implementada no domínio *travel* o agente tenta fazer o percurso à pé. Para isto utiliza o axioma "distancia_caminhada", se esse for verdadeiro então há a substituição da tarefa composta "viaje_ate" pela tarefa primitiva "!atualiza_dinheiro";

- "viaje_ate" (da linha 21 até a 42) - No segundo método são tratados dois casos: Em sua primeira ramificação é verificado se o ator está no ponto de táxi e tem dinheiro para pagar a corrida. Esta verificação é feita através do axioma "tem_dinheiro_para_taxi" e se for confirmada a tarefa composta é trocada pela sequência de tarefas "!suba" e "pague_motorista", que são respectivamente primitiva e composta. No segundo caso o agente tenta ir de ônibus. Caso exista uma rota de ônibus que ligue o ponto onde o ator está e onde deseja estar e ele tenha $1.00 para pagar a passagem. As rotas são representadas através de constantes "rota_onibus" que associam um determinado veículo a dois pontos. Se existir a rota a tarefa composta é substituída pela sequência "!espere_por", "pague_motorista" e "!va_de". A primeira e a última tarefas desta sequência são primitivas e a do meio é composta.

Os operadores realizam as tarefas primitivas. O plano consiste em uma sequência de operadores instanciados que descrevem as ações que o agente deve tomar para realizar o comportamento que foi planejado. Os operadores do domínio *travel* (Quadro 12) são descritos à seguir:

- "!va_de" - Modifica a posição no mundo onde está o veículo e o agente que está a bordo deste. Uma forma de executar esta ação é passar o controle sobre ela para uma máquina de estados. Esta máquina chama um algoritmo de busca de caminhos e obtém o trajeto que deve ser percorrido pelo ator. Com a descrição deste trajeto e as técnicas descritas no Capítulo 2 o agente é capaz de executar o comportamento de forma satisfatória;

- "!atualiza_dinheiro" - Atualiza a quantia em dinheiro que o agente possui. Esta é armazenada em um termo do tipo "numérico" que faz parte da descrição do estado atual do mundo;

- "!espere_por" - O agente espera por um ônibus. Este operador apenas assume que o ônibus chegou ao ponto onde o agente se encontra para que o restante do planejamento se desenrole. Quanto à implementação da ação para este operador basta manter a máquina de estados do agente que controla o ônibus funcionando, assim esse vai acabar parando nos pontos onde o agente embarca e desembarca;

- "!caminhe" - Esta ação é idêntica á "!va_de", mas diferencia-se por não haver um veículo, assim é mudada apenas a posição do agente no mundo. O procedimento da ação é o mesmo do outro caso, mas sem carro;

```
1       (:operator (!va_de ?veiculo ?a ?b)
2             (
3
4             )
5             (
6                   (at ?a)
7                   (at ?veiculo ?a)
8             )
9             (
10                  (at ?b)
11                  (at ?veiculo ?b)
12            )
13      )
14
15      (:operator (!atualiza_dinheiro ?old ?new)
16            (
17
18            )
19            (
20                  (tem_dinheiro ?old)
21            )
22            (
23                  (tem_dinheiro ?new)
24            )
25      )
26
27      (:operator (!espere_por ?onibus ?location)
28            (
29
30            )
31            (
32
33            )
34            (
35                  (at ?onibus ?location)
36            )
37      )
38
39      (:operator (!caminhe ?here ?there)
40            (
41
42            )
43            (
44                  (at ?here)
45            )
46            (
47                  (at ?there)
48            )
49      )
50
51      (:operator (!suba ?veiculo ?location)
52            (
53
54            )
55            (
56
57            )
58            (
59                  (at ?veiculo ?location)
60            )
61      )
```

Quadro 13: Codificação dos operadores do domínio *travel*.

- "!suba" - Sobe em um veículo que se encontra em determinado lugar. O operador tem uma implementação parecida com a do operador "!espere_por" e sua ação consiste em fazer o ator buscar o veículo naquela área e anima-lo para adentrar a esse.

5.2 Especificação dos problemas do domínio *travel*

O problema em SHOP2 é composto pelo nome do mesmo, uma referência ao domínio que o resolve, um estado inicial descrito através de um conjunto de constantes e uma rede de tarefas que devem ser realizadas.

```
(defproblem problem travel
  (
     (at centro)
     (tempo_esta bom)
     (tem_dinheiro 12)
     (distancia centro parque 2)
     (distancia centro centro_comercial 8)
     (distancia centro suburbio 12)
     (no_ponto_taxi taxi1 centro)
     (no_ponto_taxi taxi2 centro)
     (rota_onibus onibus1 centro parque)
     (rota_onibus onibus2 centro centro_comercial)
     (rota_onibus onibus3 centro suburbio)
   )
   (
     (viaje_ate suburbio)
   )
)
```

Quadro 14: Codificação do problema *travel*.

O Quadro 14 mostra um problema que deve ser resolvido com o uso do domínio *travel*. Neste o agente começa posicionado no centro da cidade e o clima estando bom. Além disto, o ambiente também é descrito pelo estado inicial. Esta descrição inclui distâncias entre as zonas da cidade, rotas de ônibus e pontos de táxi. Na rede de tarefas daquele quadro há apenas uma, ou seja, o agente deverá ir até o subúrbio.

```
(defproblem problem travel
   (
     (at centro)
     (tempo_esta bom)
     (tem_dinheiro 12)
     (distancia centro parque 2)
     (distancia centro centro_comercial 8)
     (distancia centro suburbio 12)
     (no_ponto_taxi taxi1 centro)
     (no_ponto_taxi taxi2 centro)
     (rota_onibus onibus1 centro parque)
     (rota_onibus onibus2 centro centro_comercial)
     (rota_onibus onibus3 centro suburbio)
   )
   (:unordered
     (viaje_ate suburbio)
     (viaje_ate centro)
     (viaje_ate parque)
     (viaje_ate centro_comercial)
   )
)
```

Quadro 15: Codificação do mesmo problema com outras tarefas.

```
(!espere_por onibus1 centro)
(!atualiza_dinheiro 12.0 11.0)
(!espere_por onibus1 centro)
(!va_de onibus1 centro suburbio)
(!atualiza_dinheiro 11.0 10.0)
(!va_de onibus1 centro centro)
(!suba taxi1 centro)
(!caminhe centro parque)
(!va_de taxi1 centro centro_comercial)
(!atualiza_dinheiro 10.0 0.5)
```

Quadro 16: Plano gerado pelo domínio *travel* com o problema do Quadro 15.

O Quadro 16 mostra um plano gerado utilizando o domínio *travel* juntamente com o problema listado no Quadro 15. Neste problema foi utilizada a palavra chave *unordered* que faz com que SHOP2 trate a lista de tarefas como não ordenada. Este é um exemplo de problema que quando submetido sem esta palavra, ou seja, é utilizado planejamento compatível com SHOP que é sempre ordenado, não é encontrado plano.

A especificação tanto do domínio quanto do problema exige conhecimento de um especialista, a segunda podendo ser delegada ao sistema. Mesmo assim o gerador de problemas que deve ser implementado no agente depende do escopo de cada agente, utilizando para isto heurísticas próprias para cada caso.

5.3 Especificação do domínio *logistics*

A estrutura do domínio *logistics* é semelhante a apresentada no Quadro 10 utilizada para especificar o domínio *travel*. Neste caso o domínio é formado por dois axiomas, cinco métodos e oito operadores.

Os axiomas são "*same*" e "*different*" (Quadro 17). Estes determinam, respectivamente, se os dois parâmetros que cada um recebe são iguais ou diferentes. Estes axiomas são utilizados pelos métodos e operadores com a finalidade de distinguir entre duas instâncias de objetos ou duas localizações.

```
 1       (:- (same ?x ?x)
 2             nil
 3       )
 4       (:- (different ?x ?y)
 5             (
 6                 (not
 7                         (same ?x ?y)
 8                 )
 9             )
10       )
```

Quadro 17: Codificação dos axiomas do domínio *logistics*.

Uma diferença significativa entre a especificação do domínio *logistics* e a do domínio *travel* é que no primeiro domínio são usadas proteções (através da palavra chave ":protected") para impedir que átomos sejam excluídos no mundo. O mecanismo de proteção é um dos maiores avanços introduzidos em SHOP2 (NAU; MUñOZ-AVILA; LOTEM, 1999) se comparado com SHOP (NAU et al., 2001), do qual foi derivado.

Quando um átomo recebe uma proteção, um contador é associado a ele e é incrementado se tornando impossível excluir o átomo até que o valor do contador seja igual a zero. Isto é útil em situações como ocorre no domínio *logistics* onde é necessário carregar diversos pacotes entre dois pontos em uma só viagem compartilhando o veículo. O veículo não pode ser movido do ponto de origem até que todos os pacotes estejam a bordo.

Os métodos que fazem parte do domínio *logistics* são mostrados nos quadros 18 e 19, sendo os mesmos descritos à seguir:

- "*object-at*" (da linha 1 até a 27 do Quadro 18) - Há duas formas para fazer a redução

desta tarefa. Se for possível fazer o transporte por terra este será feito diretamente por caminhões. Caso contrário a carga é levada de caminhão para o aeroporto e esta é embarcada em um avião. Assim a carga é transladada por via aérea e em seguida é colocada em outro caminhão que leva a mesma até o destino por terra;

- "*in-city-delivery*" (da linha 29 até a 47 do Quadro 18) - Há duas formas de reduzir está tarefa composta. Se o objeto já está no destino então não faça nada, mas não falhe o método. No caso do objeto estar em um local diferente na mesma cidade e existir um caminhão nesta cidade envie o caminhão para o local onde está o objeto, carregue e leve este para o destino onde deve ser descarregado;

- "*truck-at*" (da linha 49 até a 65 do Quadro 18) - Há duas maneiras de reduzir está tarefa composta. Se existe um caminhão em um local e esse local é diferente do local onde ele deveria estar para ser carregado com os objetos, então este caminhão deve ser movido para o local onde estão os objetos a serem transportados. Se o caminhão já está no local onde se encontram os objetos é adicionada uma proteção que impede que o caminhão saia dali;

- "*air-deliver-obj*" (da linha 1 até a 22 do Quadro 19) - Existem duas formas de realizar está tarefa composta. Caso o avião esteja no aeroporto onde se encontram os objetos que devem ser transportados, então é adicionada uma proteção para o avião não ser movido deste aeroporto e o avião é carregado, movido para o aeroporto de destino e em seguida descarregado. Caso o avião esteja em qualquer outro aeroporto ele é levado para o aeroporto onde se encontram os objetos, este é carregado e segue para o destino;

- "*fly-airplane*" (da linha 24 até a 40 do Quadro 19) - Há duas formas de realizar está tarefa composta. Se o avião estiver no aeroporto para onde ele deveria ir sua posição deve ser mantida e deve ser adicionada uma proteção no átomo para impedir que este seja movido para outro aeroporto. Caso o avião esteja em qualquer outro aeroporto este é conduzido através da substituição desta tarefa por "!*fly-airplane*" com parâmetros para conduzi-lo ao aeroporto em que ele deveria estar.

Os operadores do domínio *logistics* são mostrados no Quadro 21 e detalhados à seguir:

- "*!load-truck*" (da linha 1 até a 11 do Quadro 21) - Efetua a carga de um objeto em um caminhão. Este operador retira um átomo do mundo que representa a associação

da posição do objeto ao lugar onde se encontra no momento e adiciona um átomo que representa que o mesmo agora está a bordo do caminhão. Uma proteção ao átomo "truck-at" é retirada significando que se houver zero proteções para este átomo o caminhão pode ser movido;

- "*!unload-truck*" (da linha 13 até a 23 do Quadro 21) - Descarrega o caminhão associando a posição do objeto ao local onde o veículo se encontra no momento da descarga. Átomos de proteção são utilizados da mesma forma da usada no operador "*!load-truck*", mas nesse o número de proteções remanescentes indica o número de pacotes que permanecem no caminhão;
- "*!load-airplane*" (da linha 25 até a 35 do Quadro 21) - Idêntico ao "*!load-truck*", mas para aviões;
- "*!unload-airplane*" (da linha 37 até a 47 do Quadro 21) - Idêntico ao "*!unload-truck*", mas para aviões;
- "*!drive-truck*" (da linha 49 até a 59 do Quadro 21) - Move um caminhão de uma localidade até outra trocando o átomo "truck-at" que associa determinado veículo ao lugar de origem por um outro átomo de mesmo nome que associa o mesmo veículo ao lugar de destino. Uma proteção é adicionada ao átomo para que o caminhão não seja movido sem que seja totalmente carregado/descarregado;
- "*!fly-airplane*" (da linha 61 até a 71 do Quadro 21) - Idêntico ao "*!drive-truck*", mas para aviões;
- "*!add-protection*" (da linha 73 até a 81 do Quadro 21) - Adiciona uma proteção a um átomo;
- "*!delete-protection*" (da linha 83 até a 91 do Quadro 21) - Exclui uma proteção de um átomo.

5.4 Especificação dos problemas do domínio *logistics*

O Quadro 20 mostra a especificação de um problema de logística que pode ser resolvido utilizando o domínio *logistics*. Devido à complexidade da especificação do estado inicial cada tipo de átomo será detalhado a seguir:

- "AIRPLANE" - Átomo com um parâmetro que define a existência de um avião que é identificado por uma constante do tipo $plane_k$, onde k é o número que identifica o avião;

- "TRUCK" - Átomo com dois parâmetros que definem a existência de um caminhão e a cidade em que esse se encontra. A declaração deste átomo tem a seguinte forma:

$$TRUCK \quad truck_i -_j \quad city_i$$

  onde j identifica um determinado caminhão dentro os disponíveis na cidade com índice i;

- "LOCATION" - Átomo com um parâmetro que define uma localidade que consiste em uma subdivisão de uma cidade. A associação de uma localidade com a cidade em que está inscrita é feita pelo átomo do tipo "IN-CITY". "LOCATION" é declarado na forma $LOCATION \quad LOC_i{-}_j$, onde i identifica a cidade e j a localidade;

- "CITY" - Átomo com um parâmetro que define a existência de uma cidade. Este tem a forma $CITY city_i$, onde i é o índice usado para distinguir as cidades;

- "obj-at" - Associa um objeto a um local ou veículo. Este tem a seguinte forma:

$$obj - at \quad package_k \quad LOC_i{-}_j$$

  onde k é o índice que distingue cada pacote e $LOC_i{-}_j$ é uma localidade;

- "airplane-at" - Define a localização de um determinado avião. Este átomo tem a forma:

$$airplane - at \quad plane_k \quad LOC_i{-}_j$$

  onde $plane_k$ é um dos aviões e $LOC_i{-}_j$ é uma das localidades;

- "truck-at" - O mesmo que "airplane-at", mas para caminhões;

- "IN-CITY" - Associa uma localidade á cidade onde ela se encontra. Este atomo tem a forma:

$$IN - CITY \quad LOC_i -_j \quad city_k$$

onde $LOC_i{-}_j$ é uma determinada localidade e $city_k$ representa a cidade onde a localidade se encontra;

- "AIRPORT" - Declara a existência de um aeroporto em uma determinada localidade. Este átomo tem a forma $AIRPORT \quad LOC_i{-}_j$ onde $LOC_i{-}_j$ é uma localidade.

A rede de tarefas a realizar é formada por um conjunto de tarefas do tipo $obj-at$. Estas são descritas da mesma forma que os átomos de mesmo nome (que descrevem a situação atual dos objetos no mundo), mas diferenciam-se por representarem os objetivos que devem ser alcançados através do plano. Esta lista, através do uso da palavra reservada "unordered" é tratada por SHOP2 adotando qualquer ordem para a realização das tarefas.

5.5 Planejando com *SHOP2 4 Games* x JSHOP2

De posse de um problema e do domínio relacionado a esse é possível efetuar o planejamento em ambas as abordagens. Em JSHOP2 o problema e o domínio são compilados e deste é gerado um código objeto que pode ser executado. Isto se diferencia de *SHOP2 4 Games* que é interpretado e permite que múltiplos arquivos contendo descrições de domínios e problemas sejam carregados e eventualmente sejam eleitos para serem instanciados e sirvam de subsídio para uma busca por planos.

Na Tabela 2 aparece o resultado de um confronto entre as duas abordagens. Neste confronto foram utilizadas 10 variações do problema mostrado no Quadro 14. Para gerar estas variações foram modificados o estado inicial e as tarefas a serem realizadas pelo agente. Basicamente no estado inicial foi modificado o lugar onde o agente se encontra no início da execução do planejamento. Já na reformulação de tarefas foram escolhidos novos destinos para o agente.

Os planos neste caso côntem os itinerários e as formas com que o agente deve se conduzir para o outro ponto. A idéia por trás de utilizar diversos problemas neste teste é mostrar uma situação mais próxima da realidade dos jogos onde há muitas especificações de problemas para um único domínio.

Para fazer os testes apresentados na Tabela 2 o tempo total de planejamento foi dividido em três partes, são essas:

- Compilação - O tempo que o sistema de planejamento leva para ler os arquivos fonte

Tabela 2: Desempenho *SHOP2 4 Games* x JSHOP2 no domínio *travel*.

Problemas	JSHOP2			*SHOP2 4 Games*		
	Compilação	Carga	Planejamento	Compilação	Carga	Planejamento
01	2,844	0,350	0,050	0,035	0,0003	0,00158
02	2,794	0,370	0,040	0,005	0,0003	0,00062
03	2,794	0,360	0,040	0,005	0,0003	0,00030
04	2,784	0,360	0,040	0,005	0,0003	0,00030
05	2,744	0,360	0,040	0,005	0,0003	0,00030
06	2,744	0,350	0,040	0,005	0,0003	0,00124
07	2,744	0,360	0,040	0,005	0,0003	0,00030
08	2,734	0,360	0,040	0,005	0,0003	0,00030
09	2,734	0,350	0,050	0,005	0,0003	0,00030
10	2,734	0,360	0,040	0,005	0,0003	0,00126

com o domínio e o problema que deve ser resolvido. Neste processo em JSHOP2 é feito um programa executável na linguagem Java e em *SHOP2 4 Games* a descrição dos domínios e problemas são colocadas em estruturas na memória do computador denominadas moldes. Estes moldes são utilizados na fase de carga para gerar a estrutura instanciada de planejamento.

- Carga - Este é o tempo que o sistema de planejamento leva para estar pronto para iniciar o planejamento propriamente dito. Em JSHOP2 este consiste no tempo que a máquina virtual demora em carregar o executável gerado pelo processo de compilação. Em *SHOP2 4 Games* este é o tempo levado pelo sistema para instanciar a estrutura de moldes para que o processo de planejamento possa começar.

- Planejamento - É o tempo que o algoritmo SHOP2 leva para planejar em cada uma das abordagens.

Embora todos os dez problemas tenham propositalmente o mesmo custo de compilação, carga e planejamento, JSHOP2 tem um custo de compilação constante enquanto *SHOP2 4 Games* tem o custo de compilação do domínio no início e um custo para cada problema. Na Tabela 2 o custo da compilação do domínio foi agrupado no custo da compilação do primeiro problema, sendo que nos próximos problemas foi computado apenas o custo da leitura do problema. Mesmo assim os custos de JSHOP2 para está fase foram no mínimo 81 vezes mais altos.

A fase de carga é bem mais rápida que a de compilação. Nesta tivemos problemas

para medir o tempo de execução por limitações da função “clock()” do cabeçalho “time.h” de C++. Isto se deu por esta função possuir resolução de aproximadamente 10 ms nas máquinas atuais e isto não ser suficiente para determinar os tempos de *SHOP2 4 Games* onde foi acusado zero, ou próximo a zero.

Para efetuar a medição dos tempos de carga de cada par de domínio e de problema, fizemos 50 testes consecutivos com cada um desses. Em seguida dividimos o tempo total por 50 e desta forma obtivemos os tempos de cada carga de problema como é mostrado na Tabela 2.

O mesmo experimento utilizado para determinar o tempo de carga foi repetido para encontrar os tempos de solução do plano para cada um dos problemas que são mostrados na mesma tabela.

Também testamos a solução de um problema utilizando o domínio *logistics* obtendo o tempo de execução da mesma maneira descrita acima. Tanto este domínio quanto o problema que foi escolhido para testá-lo (Quadro 20) são bem mais complexos do que o domínio *travel* e seus os problemas que foram resolvido resultando nos tempos mostrados na Tabela 2.

Devido à maior complexidade do domínio *logistics*, os tempos de compilação e carga foram um pouco maiores do que o tempo utilizado para as mesmas operações com o domínio *travel*. Mesmo assim a diferença entre os tempos utilizados em um domínio e no outro é insignificante.

Mas as diferenças gritantes aparecem mesmo quando comparadas as abordagens JSHOP2 e *SHOP2 4 Games*. Para compilar o problema e o domínio *logistics* na primeira abordagem são gastos 3,775 segundos, enquanto utilizando a segunda abordagem são gastos 0,04 segundos para compilar o domínio e 0,01 segundo para o problema. Somente no tempo equivalente a diferença entre o tempo de compilação do domínio *logistics* e o utilizado para compilar o domínio *travel* com JSHOP2 daria para processar os dois domínios e seus respectivos problemas em *SHOP2 4 Games*.

A diferença aparece também no tempo de carga do problema e do domínio. Para o domínio *logistics* JSHOP2 gasta 0,421 segundos para ser carregado pela máquina Java enquanto nossa abordagem que é instanciada diretamente na memória por um programa que já se encontra em execução (o próprio jogo) gasta apenas 0,01 segundo.

Acreditamos que as estas diferenças nos tempos de compilação e carga entre as duas abordagens estão na necessidade de utilizar um compilador durante o processo e

os tempos de carga de executáveis do sistema operacional e da máquina Java. Também tivemos alguns problemas ao tentar cancelar a produção de um plano ou parar o planejador quando usado JSHOP2, o que às vezes é útil pelo fato de, nem sempre, os planos saírem em um tempo viável nesta aplicação.

O tempo total de processamento do domínio *travel* e de seus dez problemas foi de cerca de 31 segundos para JSHOP2 e cerca de 9 milissegundos utilizando a nossa abordagem. Acreditamos que isto justificou o esforço despendido na construção do planejador para jogos.

Durante o tempo de execução do jogo os agentes deverão fazer chamadas ao planejador de acordo com suas necessidades de obtenção de planos de mais alto nível que ditam as estratégias que cada um deles deve seguir. Voltando ao exemplo do motorista automático, seria possível colocar uma chamada ao planejador no evento "OnEnter" do estado "DIRIGINDO" na máquina de estados mostrada no Quadro 7 da seção 2.7.3.

Assim toda vez que o agente retornasse a este estado o plano seria refeito e distorções do plano anterior provocadas por modificações no mundo seriam corrigidas. Estas distorções vão desde a impossibilidade de chegar ao destino dentro do prazo, ou falta de combustível, ou mesmo mudanças no plano que permitam contornar obstáculos.

Existe dezenas de pontos como este no código fonte de um jogo onde seria interessante solicitar um plano de ação para que o agente realize seus objetivos. Tendo um planejador em seu conjunto de ferramentas, cabe ao desenvolvedor do jogo utilizá-lo racionalmente no desenvolvimento da aplicação juntamente com as outras técnicas de IA mostradas no Capítulo 2.

```
1       (:method (obj-at ?obj ?loc-goal)
2               (
3                       (in-city ?loc-goal ?city-goal)
4                       (obj-at ?obj ?loc-now)
5                       (in-city ?loc-now ?city-goal)
6                       (truck ?truck ?city-goal)
7               )
8               (
9                       (in-city-delivery ?truck ?obj ?loc-now ?loc-goal)
10              )
11
12              (
13                      (in-city ?loc-goal ?city-goal)
14                      (obj-at ?obj ?loc-now)
15                      (in-city ?loc-now ?city-now)
16                      (different ?city-goal ?city-now)
17                      (truck ?truck-now ?city-now)
18                      (truck ?truck-goal ?city-goal)
19                      (airport ?airport-now) (in-city ?airport-now ?city-now)
20                      (airport ?airport-goal) (in-city ?airport-goal ?city-goal)
21              )
22              (
23                      (in-city-delivery ?truck-now ?obj ?loc-now ?airport-now)
24                      (air-deliver-obj ?obj ?airport-now ?airport-goal)
25                      (in-city-delivery ?truck-goal ?obj ?airport-goal ?loc-goal)
26              )
27      )
28
29      (:method (in-city-delivery ?truck ?obj ?loc-from ?loc-to)
30              (
31                      (same ?loc-from ?loc-to)
32              )
33              (
34
35              )
36
37              (
38                      (in-city ?loc-from ?city)
39                      (truck ?truck ?city)
40              )
41              (
42                      (truck-at ?truck ?loc-from)
43                      (:immediate !load-truck ?obj ?truck ?loc-from)
44                      (truck-at ?truck ?loc-to)
45                      (:immediate !unload-truck ?obj ?truck ?loc-to)
46              )
47      )
48
49      (:method (truck-at ?truck ?loc-to)
50              (
51                      (truck-at ?truck ?loc-from)
52                      (different ?loc-from ?loc-to)
53              )
54              (
55                      (:immediate !drive-truck ?truck ?loc-from ?loc-to)
56              )
57
58              (
59                      (truck-at ?truck ?loc-from)
60                      (same ?loc-from ?loc-to)
61              )
62              (
63                      (:immediate !add-protection (truck-at ?truck ?loc-to))
64              )
65      )
```

Quadro 18: Codificação dos métodos do domínio *logistics*.

```
    (:method (air-deliver-obj ?obj ?airport-from ?airport-to)
             airplane-at-the-current-airport
             (
                    (airplane-at ?airplane ?airport-from)
             )
             (
                    (:immediate !add-protection (airplane-at ?airplane ?airport-from))
                    (!load-airplane ?obj ?airplane ?airport-from)
                    (fly-airplane ?airplane ?airport-to)
                    (!unload-airplane ?obj ?airplane ?airport-to)
             )

             (
                    (airplane-at ?airplane ?any-airport)
             )
             (
                    (:immediate !fly-airplane ?airplane ?any-airport ?airport-from)
                    (!load-airplane ?obj ?airplane ?airport-from)
                    (fly-airplane ?airplane ?airport-to)
                    (!unload-airplane ?obj ?airplane ?airport-to)
             )
    )

    (:method (fly-airplane ?airplane ?airport-to)
             airplane-already-there
             (
                    (airplane-at ?airplane ?airport-to)
             )
             (
                    (:immediate !add-protection (airplane-at ?airplane ?airport-to))
             )

             fly-airplane-in
             (
                    (airplane-at ?airplane ?airport-from)
             )
             (
                    (:immediate !fly-airplane ?airplane ?airport-from ?airport-to)
             )
    )
```

Quadro 19: Codificação dos métodos do domínio *logistics* (continuação).

```
(defproblem logistics_problem logistics
    (
        (AIRPLANE plane1)           (airplane-at plane1 LOC1-1)     (AIRPLANE plane2)           (airplane-at plane2 LOC4-1)
        (AIRPLANE plane3)           (airplane-at plane3 LOC5-1)     (CITY city1)                (AIRPORT LOC1-1)
        (TRUCK truck1-1 city1)      (truck-at truck1-1 LOC1-1)      (LOCATION LOC1-1)           (IN-CITY LOC1-1 city1)
        (LOCATION LOC1-2)           (IN-CITY LOC1-2 city1)          (LOCATION LOC1-3)           (IN-CITY LOC1-3 city1)
        (CITY city2)                (AIRPORT LOC2-1)                (TRUCK truck2-1 city2)      (truck-at truck2-1 LOC2-1)
        (LOCATION LOC2-1)           (IN-CITY LOC2-1 city2)          (LOCATION LOC2-2)           (IN-CITY LOC2-2 city2)
        (LOCATION LOC2-3)           (IN-CITY LOC2-3 city2)          (CITY city3)                (AIRPORT LOC3-1)
        (TRUCK truck3-1 city3)      (truck-at truck3-1 LOC3-1)      (LOCATION LOC3-1)           (IN-CITY LOC3-1 city3)
        (LOCATION LOC3-2)           (IN-CITY LOC3-2 city3)          (LOCATION LOC3-3)           (IN-CITY LOC3-3 city3)
        (CITY city4)                (AIRPORT LOC4-1)                (TRUCK truck4-1 city4)      (truck-at truck4-1 LOC4-1)
        (LOCATION LOC4-1)           (IN-CITY LOC4-1 city4)          (LOCATION LOC4-2)           (IN-CITY LOC4-2 city4)
        (LOCATION LOC4-3)           (IN-CITY LOC4-3 city4)          (CITY city5)                (AIRPORT LOC5-1)
        (TRUCK truck5-1 city5)      (truck-at truck5-1 LOC5-1)      (LOCATION LOC5-1)           (IN-CITY LOC5-1 city5)
        (LOCATION LOC5-2)           (IN-CITY LOC5-2 city5)          (LOCATION LOC5-3)           (IN-CITY LOC5-3 city5)
        (CITY city6)                (AIRPORT LOC6-1)                (TRUCK truck6-1 city6)      (truck-at truck6-1 LOC6-1)
        (LOCATION LOC6-1)           (IN-CITY LOC6-1 city6)          (LOCATION LOC6-2)           (IN-CITY LOC6-2 city6)
        (LOCATION LOC6-3)           (IN-CITY LOC6-3 city6)          (CITY city7)                (AIRPORT LOC7-1)
        (TRUCK truck7-1 city7)      (truck-at truck7-1 LOC7-1)      (LOCATION LOC7-1)           (IN-CITY LOC7-1 city7)
        (LOCATION LOC7-2)           (IN-CITY LOC7-2 city7)          (LOCATION LOC7-3)           (IN-CITY LOC7-3 city7)
        (CITY city8)                (AIRPORT LOC8-1)                (TRUCK truck8-1 city8)      (truck-at truck8-1 LOC8-1)
        (LOCATION LOC8-1)           (IN-CITY LOC8-1 city8)          (LOCATION LOC8-2)           (IN-CITY LOC8-2 city8)
        (LOCATION LOC8-3)           (IN-CITY LOC8-3 city8)          (obj-at package1 loc8-3)    (obj-at package2 loc1-1)
        (obj-at package3 loc2-2)    (obj-at package4 loc6-3)        (obj-at package5 loc5-1)    (obj-at package6 loc2-3)
        (obj-at package7 loc1-2)    (obj-at package8 loc6-3)        (obj-at package9 loc5-2)    (obj-at package10 loc7-1)
        (obj-at package11 loc2-3)   (obj-at package12 loc2-2)       (obj-at package13 loc7-2)   (obj-at package14 loc7-1)
        (obj-at package15 loc8-2)
    )
    (:unordered
        (obj-at package1 loc8-1)
        (obj-at package2 loc2-1)
        (obj-at package3 loc2-3)
        (obj-at package4 loc6-2)
        (obj-at package5 loc1-1)
        (obj-at package6 loc6-2)
        (obj-at package7 loc6-3)
        (obj-at package8 loc1-1)
        (obj-at package9 loc4-2)
        (obj-at package10 loc8-3)
        (obj-at package11 loc3-2)
        (obj-at package12 loc3-3)
        (obj-at package13 loc3-2)
        (obj-at package14 loc6-3)
        (obj-at package15 loc5-1)
    )
)
```

Quadro 20: Codificação do problema *logistics*.

```
(:operator (!load-truck ?obj ?truck ?loc)
           (
           )
           (
                (obj-at ?obj ?loc)
                (:protection (truck-at ?truck ?loc))
           )
           (
                (in-truck ?obj ?truck)
           )
)

(:operator (!unload-truck ?obj ?truck ?loc)
           (
           )
           (
                (in-truck ?obj ?truck)
                (:protection (truck-at ?truck ?loc))
           )
           (
                (obj-at ?obj ?loc)
           )
)

(:operator (!load-airplane ?obj ?airplane ?loc)
           (
           )
           (
                (obj-at ?obj ?loc)
                (:protection (airplane-at ?airplane ?loc))
           )
           (
                (in-airplane ?obj ?airplane)
           )
)

(:operator (!unload-airplane ?obj ?airplane ?loc)
           (
           )
           (
                (in-airplane ?obj ?airplane)
                (:protection (airplane-at ?airplane ?loc))
           )
           (
                (obj-at ?obj ?loc)
           )
)

(:operator (!drive-truck ?truck ?loc-from ?loc-to)
           (
           )
           (
                (truck-at ?truck ?loc-from)
           )
           (
                (truck-at ?truck ?loc-to)
                (:protection (truck-at ?truck ?loc-to))
           )
)

(:operator (!fly-airplane ?airplane ?airport-from ?airport-to)
           (
           )
           (
                (airplane-at ?airplane ?airport-from)
           )
           (
                (airplane-at ?airplane ?airport-to)
                (:protection (airplane-at ?airplane ?airport-to))
           )
)

(:operator (!add-protection ?g)
           (
           )
           (
           )
           (
                (:protection ?g)
           )
)

(:operator (!delete-protection ?g)
           (
           )
           (
                (:protection ?g)
           )
           (
           )
)
```

Quadro 21: Codificação dos operadores do domínio *logistics*.

6 Conclusões

Jogos de computador são sistemas de computação que ao longo de sua história sempre demandaram muito esforço de desenvolvimento, tanto na área de computação gráfica quanto na área de inteligência artificial. Esta segunda área, em especial sua subárea nomeada planejamento, que é o foco deste trabalho, durante anos teve de ser um pouco limitada em prol da outra. Isto se deu pelos gráficos serem considerados prioritários para obtenção de uma boa interação entre o jogador humano e o jogo.

Embora o desenvolvimento de um planejador seja uma tarefa bastante cara e difícil de ser realizada, este tem-se mostrado cada vez mais adequado para o uso em jogos. A grande vantagem do uso de um planejador é que o comportamento do agente do jogo se torna orientado a objetivos. De certa forma sempre houve alguma orientação a objetivos nos agentes para jogos, mas esta varia de um simples plano de ação traçado por um programador humano e executado diretamente no código do jogo a um planejador simplificado baseado em um sistema de regras ou árvores de decisão que escolhe entre conjuntos de planos prontos.

Mas não basta inserir no código do jogo um sub-programa ou uma chamada a um programa externo que faça planos a partir de descrições de domínios e problemas. Os jogos apresentam restrições duras de limite de tempo para sua execução, o que exige que atividades complexas sejam executadas em milésimos de segundo. Além do que é importante que o planejador possa ser utilizado junto a outras técnicas que são utilizadas atualmente para resolver problemas de IA em outros níveis de abstração. Para que isto seja possível é necessário haver algum acoplamento entre os algoritmos que implementam cada técnica. Promover este acoplamento é a função do motor de IA.

O processamento de algumas destas atividades pode ser feito antes do início da partida, onde não há limites de tempo tão severos. Entre estas atividades estão a leitura de arquivos de domínios e de alguns problemas. Isto é oportuno, afinal entre as fases que antecedem a execução do planejamento a mais custosa é a de leitura dos domínios.

Outras atividades, tais como, o planejamento propriamente dito e a compilação de planos em tempo de execução dependem em muito da coleta de dados em tempo de execução, assim estas atividades não podem ser pré-processadas.

Nos últimos anos foram feitas algumas tentativas interessantes de utilizar um planejador automático ou semi-automático em jogos, o que se mostrou um caminho promissor. Entre estas duas categorias de planejadores a que melhor se adequou aos requisitos impostos pelos jogos foi a de semi-automáticos, em especial os baseados em redes de tarefas hierárquicas.

Neste contexto apresentamos *SHOP2 4 Games*. Esta é uma biblioteca de *software* que consiste em uma implementação do sistema de planejamento SHOP2 adaptada para ser anexada em um motor de IA para jogos. SHOP2 se destaca dentre os algoritmos de planejamento baseado em HTN pelo poder de expressão de sua linguagem de descrição e pela rapidez com que faz planos. Seus domínios podem inclusive trabalhar com dados numéricos, o que é desejável pelos agentes em jogos geralmente tomarem decisões sobre recursos que são tratados de uma forma mais natural se quantificados.

Nossa biblioteca diferencia-se das outras implementações de SHOP2 pela maneira com que foi arquitetada para resolver questões comuns à implementação de jogos de computador.

Para alcançar os requisitos de tempo real, juntamente com o uso de problemas descritos em tempo de simulação a biblioteca foi codificada como um interpretador de problemas e domínios. Os domínios são lidos de arquivos antes do tempo de compilação e ficam em listas na memória prontos para serem instanciados. Desta forma os tempos de análise sintática e semântica não influenciam na qualidade do jogo. Além disto, nossa implementação permite que vários problemas sejam instanciados com seus respectivos domínios simultaneamente. Isto é útil, pois vários agentes poderão estar compartilhando a CPU (*Computer Processing Unit*) (sigla em inglês) em dado instante de tempo.

Hoje já está amplamente disponível para os jogadores *hardware* com múltiplas CPUs ou similares que permite que mais de um processo seja efetivamente executado ao mesmo tempo. Assim é importante que exista uma independência entre os problemas instanciados, o que é atendido por nossa implementação.

Nossa biblioteca foi codificada na linguagem C++ que é tradicionalmente a mais utilizada na construção de jogos. Além da possibilidade de utilização desta biblioteca diretamente nos jogos produzidos em C++, esta também pode ser facilmente importada

por programas descritos em outras linguagens cujos compiladores que geram seu código objeto aceitam importar bibliotecas dinâmicas. Além disto, foram observados vários aspectos ligados à portabilidade da biblioteca e esta atualmente roda tanto no Microsoft Windows quanto no Linux. Acreditamos que não seja difícil fazer portes desta para outros sistemas que tenham versões dos *softwares* descritos na Seção 4.1.

Existem algumas coisas por fazer que ficam para trabalhos futuros. Entre estas está a formulação dos objetivos e a do estado inicial de planejamento em tempo de simulação. Também há a questão da serialização da execução de planos. Esta deve tratar a disputa dos recursos pelos agentes, ou seja, a maneira com que as mudanças especificadas no plano são efetivadas no mundo. Finalmente deve ser feita a integração da biblioteca com o motor de IA e este com o motor de jogos. Outra coisa que nos parece interessante é a elaboração de um tipo de termo que represente variáveis para representação de lógica *fuzzy*.

Referências

ADZIMA, J. Competitive AI Racing under Open Street Conditions. *AI Game Programming Wisdom*, Charles River Media, p. 460–471, 2002.

AKENINE-MÖLLER, T.; HAINES, E. *Real-Time Rendering*. 2ª. ed. [S.l.]: A K Peters, 2002.

ALEXANDER, B. An Architecture Based on Load Balancing. *AI Game Programming Wisdom*, Charles River Media, p. 298–304, 2002.

BACCHUS, F.; KABANZA, F. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, v. 116, n. 1-2, p. 123–191, 2000. Disponível em: <citeseer.ist.psu.edu/bacchus00using.html>.

BIASILLO, G. Training an AI to Race. *AI Game Programming Wisdom*, Charles River Media, p. 455–459, 2002.

BLY, P. B. de. *Programming Believable Characters for Computer Games*. 1ª. ed. [S.l.]: Charles River Media, 2004.

BOURG, D. M.; SEEMAN, G. *AI for Game Developers*. 1ª. ed. [S.l.]: O'Reilly, 2004.

BUCKLAND, M. *Programming Game AI by Example*. 1ª. ed. [S.l.]: Wordware Publishing Inc, 2005.

CHAMPANDARD, A. J. *AI for Game Developers*. 1ª. ed. [S.l.]: New Riders Publishing, 2003.

CHRISTIAN, M. A Simple Inference Engine for a Rule-Based Architecture. *AI Game Programming Wisdom*, Charles River Media, p. 305–320, 2002.

CRAWFORD, C. *Chris Crawford on Game Design*. 1ª. ed. [S.l.]: New Riders Publishing, 2003.

DYBSAND, E. A Generic Fuzzy State Machine in C++. *Game Programming Gems 2*, Charles River Media, p. 337–341, 2001.

EDELKAMP, S. Taming Numbers and Durations in the Model Checking Integrated Planning System. *Journal of Artificial Intelligence Research, 3rd International Planning Competition*, 2003.

EROL, K.; HENDLER, J.; NAU, D. S. *Complexity Results for HTN Planning*. citeseer.ist.psu.edu/article/erol95semantics.html, 1994a. v. 18, n. 1, 69-93 p. Technical report CS-TR-3240.

EROL, K.; HENDLER, J.; NAU, D. S. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. *In Proceedings of 2nd International Conference on AI Planning Systems*, p. 249–254, 1994b.

FIKES, R. E.; NILSSON, N. J. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence 2*, p. 189–208, 1971.

FUNGE, J. D. *Making Them Behave: Cognitive Models for Computer Animation*. Phd thesis. University of Toronto, 1998.

HAWES, N. *Anytime Deliberation for Computer Games Agents*. Phd thesis. The University of Birminghan, 2003.

HOFFMAN, J.; NEBEL, B. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, p. 253–302, 2000.

ILGHAMI, O. *Documentation for JSHOP2*. University of Maryland, 2005. Technical Report, CS-TR-4694.

JONES, M. T. *AI Application Programming*. 1ª. ed. [S.l.]: Charles River Media, 2003.

KAUTZ, H.; SELMAN, B. Blackbox: A New Approach to the Application of Theorem Proving to Problem Solving. *Working Notes of the Workshop on Planning as Combinatorial Search*, p. 58–60, 1998.

LAIRD, J. E.; LENT, M. van. Human-Level AI's Killer Application Interactive Computer Games. 2000.

LAMOTHE, A. *Things of Windows Game Programming Gurus*. 1ª. ed. [S.l.]: Sams, 1999.

LAMOTHE, A. A Neural-Net Primer. *Game Programming Gems*, Charles River Media, p. 330–350, 2000.

LONG, D.; FOX, M. *An Extension to PDDL for Expressing Temporal Planning Domains*. http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/fox03a.pdf, 2002. Manual de Referência da Linguagem PDDL.

LONG, D.; FOX, M. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research 20*, 2003. 3rd International Planning Competition.

MAGNUSSON, M.; KVARNSTRÖM, J. Talplanner in the 3rd International Planning Competition: Extensions and Control Rules. *Journal of Artificial Intelligence Research*, 2003.

MATTHEWS, J. Basic A* Pathfinding Made Simple. *AI Game Programming Wisdom*, Charles River Media, p. 105–113, 2002.

MCCUSKEY, M. Fuzzy Logic and Video Games. *Game Programming Gems*, Charles River Media, p. 319–329, 2000.

NAU, D. S. et al. Total-Order Planning with Partially Ordered Subtasks. *In IJCAI-2001*, p. 968–973, 2001.

NAU, D. S.; MUñOZ-AVILA, H.; LOTEM, A. SHOP: Simple Hierarchical Ordered Planner. *In IJCAI-99*, p. 968–973, 1999.

NAU, D. S. et al. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, n. 20, p. 379–404, Dezembro 2003.

NILSSON, N. J. *Principles of Artificial Intelligence*. [S.l.]: Morgan Kaufmann Publishers, 1980. pp. 275-414 p.

O´BRIEN, J. A Flexible Goal-Based Planning Architecture. *AI Game Programming Wisdom*, Charles River Media, p. 375–383, 2002.

PEDERSEN, R. E. *Game Design Foundations*. 1ª. ed. [S.l.]: Wordware Publishing, 2003.

PINTER, M. Realistic Turning between Waypoints. *AI Game Programming Wisdom*, Charles River Media, p. 186–192, 2002.

RABIN, S. Designing a General Robust AI Engine. *Game Programming Gems*, Charles River Media, p. 221–236, 2000.

RABIN, S. The Magic of Data-Driven Design. *Game Programming Gems*, Charles River Media, p. 3–7, 2000.

ROUSE, R. *Game Design: Theory & Practice*. 1ª. ed. [S.l.]: Wordware Publishing, 2001.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 2ª. ed. [S.l.]: Prentice-Hall, Englewood Cliffs, NJ, 2003.

SMITH, S.; NAU, D.; THROOP, T. Computer Bridge: A Big Win for AI Planning. *AI Magazine*, v. 19, n. 2, p. 93–106, 1998.

SURASMITH, S. Preprocessed Solution for Open Terrain Navigation. *AI Game Programming Wisdom*, Charles River Media, p. 161–170, 2002.

WOODCOCK, S. Flocking: A Simple Techinique for Simulating Group Behavior. *Game Programming Gems*, Charles River Media, p. 330–350, 2000.

ZAROZINSKI, M. An Open-Source Fuzzy Logic Library. *AI Game Programming Wisdom*, Charles River Media, p. 90–101, 2002.

ZAROZINSKI, M.; THAN, L. Imploding Combinatorial Explosion in Fuzzy Logic. *Game Programming Gems 2*, Charles River Media, p. 342–350, 2001.