

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ICE
Kernel de Comportamento
Baseado em Agentes para Jogos

por

FELIPE RECH MENEGUZZI
PAULO HENRIQUE DE SOUZA SCHNEIDER
THAIS CHRISTINA WEBBER DOS SANTOS

Relatório Final de Trabalho de Conclusão
Do Curso de Bacharelado em Ciência da Computação

Prof. Dr. Michael da Costa Móra
Orientador

Porto Alegre, dezembro de 2001

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Meneguzzi, Felipe Rech
Santos, Thais Christina Webber dos
Schneider, Paulo Henrique de Souza

ICE Kernel de Comportamento Baseado em Agentes para Jogos – PUCRS. Felipe Rech Meneguzzi, Paulo Henrique de Souza Schneider e Thais Christina Webber dos Santos. – Porto Alegre: Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul, 2001.

156p.: il.

Relatório de Trabalho de Conclusão - Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática, Porto Alegre, 2001. Orientador: Prof. Dr. Michael da Costa Mora.

Relatório de Trabalho de Conclusão: Inteligência Artificial, Agentes, BDI, Reativo, Jogos, Comportamento, Objetos, Multi-Agente.

All that is necessary for the triumph of evil is that good men do nothing.

Edmund Burke

AGRADECIMENTOS

Antes e durante o curso da elaboração deste trabalho, diversas pessoas contribuíram para que este se trabalho evoluísse até o ponto onde chegou, desta forma o grupo gostaria de prestar agradecimento a algumas pessoas.

Em primeiro lugar ao orientador deste trabalho, o professor Michael da Costa Móra, que proporcionou a serenidade tão necessária nos momentos de impasse que ocorreram no curso deste trabalho.

Ao professor Eduardo Henrique Pereira de Arruda e seus orientandos Cristiano Rech Meneguzzi, André Gobbi Farina e Leandro Puricelli Pires respectivamente pela criação e disponibilização do modelo utilizado como base para a elaboração deste documento.

Às orientandas da professora Lúcia Maria Martins Giraffa, Ângela Cristina Mazzorani, Luciana de Araújo Spagnoli e Sabrina dos Santos Marczak pela pronta disponibilização do seu trabalho e materiais utilizados.

A Marcelo Barbosa por ter gentilmente aberto mão do seu tempo oferecendo seus dons artísticos utilizados na criação dos jogos exemplo.

SUMÁRIO

LISTA DE ABREVIATURAS.....	IX
LISTA DE FIGURAS.....	X
LISTA DE TABELAS.....	XI
RESUMO.....	XII
ABSTRACT.....	XIII
1 INTRODUÇÃO.....	1
2 OBJETIVOS.....	3
3 AGENTES.....	5
3.1 Inteligência Artificial.....	5
3.2 Introdução aos Agentes em IA.....	6
3.2.1 Origens dos Agentes.....	6
3.2.2 Aplicações atuais.....	7
3.2.3 Definições Básicas sobre Agentes.....	8
3.3 Taxonomia.....	11
3.3.1 Agentes.....	11
3.3.2 Ambientes.....	12
3.4 Modelos de Agentes.....	14
3.4.1 Agentes Reativos.....	15
3.4.2 Agentes BDI.....	16
3.4.3 Arquiteturas Híbridas.....	17
3.5 Arquiteturas de Agentes.....	18
3.5.1 IRMA.....	18
3.5.2 PRS.....	21
3.5.3 COSY.....	24
3.5.4 GRATE.....	25
3.5.5 InteRRaP.....	27
3.5.6 Comparação entre as arquiteturas.....	29
4 PROGRAMAÇÃO DE JOGOS.....	32
4.1 Histórico do desenvolvimento de jogos.....	32
4.1.1 O Começo.....	32
4.1.2 Evolução de Hardware.....	34
4.1.3 Gráficos.....	34
4.1.4 Som.....	35
4.1.5 Interface.....	35

4.1.6 Jogos Multi-Jogador	35
4.1.7 Evolução de Software	36
4.1.8 Evolução dos Profissionais	37
4.1.9 Inteligência Artificial nos Jogos	39
4.2 O Estado da Arte da IA na indústria	40
4.2.1 Unreal Tournament	40
4.2.2 Baldur's Gate	41
4.2.3 The Sims	41
4.2.4 Black and White	41
4.3 Jogos Exemplo	42
4.3.1 Tcheco Balls	42
4.3.2 FormiguinhaS	43
5 A ARQUITETURA ICE	45
5.1 Funcionalidades Propostas	45
5.1.1 Definição dos Agentes no Jogo	45
5.1.2 Camada de Visualização	46
5.1.3 Ambiente de comunicação multi-agentes	46
5.2 Descrição da arquitetura ICE	47
5.2.1 Visão geral	47
5.2.2 Os Agentes do ICE	48
5.2.3 Ether: o ambiente dos Agentes	52
5.3 Revisões no projeto original	53
6 O PROJETO ICE	55
6.1 Fundamentos do projeto	55
6.1.1 A Separação entre Lógica e Dados	55
6.1.2 Modelando comportamento fora do código	56
6.1.3 Por que utilizar Agentes	58
6.2 Considerações sobre a Linguagem	58
6.2.1 O Agente	59
6.2.2 Tipos	60
6.2.3 Ações	61
6.2.4 Crenças	63
6.2.5 Objetivos	63
6.2.6 Planos	64
6.2.7 Reações	64
6.3 Exemplos de utilização da linguagem	65
6.3.1 Soldado Hans	65
6.3.2 Base de comando	67
6.3.3 Bola do Tcheco Balls	68
6.3.4 AntBot	70

6.3.5 Formiga.....	73
7 MODELAGEM E IMPLEMENTAÇÃO	76
7.1 Compilador.....	76
7.1.1 Compilação versus Interpretação	77
7.1.2 Ferramentas	78
7.1.3 Módulos do compilador.....	78
7.1.4 Analisador léxico e sintático	79
7.1.5 Construtor de nodos da árvore sintática	79
7.1.6 Gerador de código	81
7.2 Ambiente ICE	82
7.2.1 Arquitetura Básica	83
7.2.2 Ciclo de funcionamento	85
7.3 Ether e Game Engine.....	86
7.3.1 Classe GameEngine.....	86
7.3.2 Classe Ether	87
7.4 Agente.....	88
7.4.1 Diagrama de classes	88
7.4.2 Classe GenericAgent.....	89
7.4.3 Classe Sensor	91
7.4.4 Classe Actuator	91
7.4.5 Classe GenericReflexes	92
7.4.6 Classe GenericBeliefs	93
7.4.7 Classe GenericObjectives	93
7.4.8 Classe GenericObjective	94
7.4.9 Classe GenericPlanner.....	95
7.4.10 Classe GenericPlan	96
7.5 Geração de código	97
7.5.1 O agente.....	97
7.5.2 Ações.....	99
7.5.3 Reações.....	101
7.5.4 Crenças, objetivos e planos.....	102
8 ATIVIDADES	115
8.1 Atividades e Responsáveis no TC1	115
8.1.1 Revisão Bibliográfica	115
8.1.2 Projeto da Arquitetura.....	116
8.1.3 Redação do Volume	116
8.2 Atividades e Responsáveis no TC2	117
8.2.1 Implementação do Projeto.....	117
8.2.2 Revisão Bibliográfica	118
8.2.3 Redação do Volume Final de TC2.....	119

8.2.4 Apresentação do Trabalho de Conclusão II	119
8.3 Dependências entre as Atividades	120
8.4 Recursos Necessários	120
9 CONCLUSÕES	121
9.1 Trabalhos Futuros	122
ANEXO I – CRONOGRAMA.....	123
ANEXO II – IADL GRAMÁTICA COMPLETA	124
ANEXO III – MANUAL DO ICEC E DA IADL	127
GLOSSÁRIO.....	139
REFERÊNCIAS BIBLIOGRÁFICAS.....	141

LISTA DE ABREVIATURAS

ABDI- Componente BDI dos agentes da arquitetura ICE

AI - *Artificial Intelligence*

API - Application Programming Interface

AR - Componente Reativo dos agentes da arquitetura ICE

CD - *Compact Disk*

DAI - *Distributed Artificial Intelligence* (Inteligência Artificial Distribuída)

GE - *Game Engine*

IA - Inteligência Artificial

IADL - *ICE Agent Description Language*

ICE - *Intelligence Control Engine*

ICEC - ICE Compiler

IP - Internet Protocol

IPX - Internetwork Packet Exchange

MP3 - (MPEG 3) *Movie Pictures Expert Group Layer 3*

MSDOS - *Microsoft Disk Operating System*

PC - *Personal Computer*

SO - Sistema Operacional

TC1 - Trabalho de Conclusão I

TC2 - Trabalho de Conclusão II

TCP - *Transmission Control Protocol*

LISTA DE FIGURAS

Figura 1	Contexto da arquitetura ICE dentro do jogo	3
Figura 2	Arquitetura básica de um agente	9
Figura 3	Visão abstrata da interação agente-ambiente [WOO99]	12
Figura 4	Estrutura da arquitetura IRMA, extraído de [DHE00].....	19
Figura 5	Estrutura da arquitetura PRS [DHE00]	22
Figura 6	Arquitetura COSY [HAD96]	25
Figura 7	Arquitetura funcional da GRATE [HAD96].....	26
Figura 8	A arquitetura InteRRaP [FIS98].....	28
Figura 9	Importância da IA nos jogos.	39
Figura 10	Tela do protótipo do jogo Tcheco Balls	43
Figura 11	Tela do protótipo do jogo FormiguinhaS.....	44
Figura 12	Descrição em alto nível da arquitetura do ICE	47
Figura 13	Visão geral dos Agentes ICE.....	49
Figura 14	Estrutura interna dos Agentes ICE	50
Figura 15	Modelo ideal de interação entre os desenvolvedores e o jogo.....	56
Figura 16	Visão geral do ambiente ICE.....	77
Figura 17	Nodos da árvore sintática.....	80
Figura 18	Classes do gerador de código	81
Figura 19	Diagrama de classes do ambiente ICE	83
Figura 20	Ciclo de funcionamento do ICE caso não ocorra reconsideração ..	85
Figura 21	Ciclo de execução do ICE caso ocorra reconsideração	86
Figura 22	Classe <code>Ether</code>	87
Figura 23	Diagrama das classes internas do agente ICE.....	89
Figura 24	Classe <code>GenericAgent</code>	89
Figura 25	Classe <code>Sensor</code>	91
Figura 26	Classe <code>Actuator</code>	91
Figura 27	Classe <code>GenericReflexes</code>	92
Figura 28	Classe <code>GenericBeliefs</code>	93
Figura 29	Classe <code>GenericObjectives</code>	93
Figura 30	Classe <code>GenericObjective</code>	94
Figura 31	Classe <code>GenericPlanner</code>	95
Figura 32	Classe <code>GenericPlan</code>	96
Figura 33	Utilização do ICEC	136

LISTA DE TABELAS

Tabela 1: Evolução dos jogos	38
------------------------------------	----

RESUMO

Recentemente tem-se visto um aumento significativo na importância da IA no processo de desenvolvimento de jogos na medida em que o esforço de programação tem se deslocado dos gráficos e do som para a “jogabilidade”.

Além do mais, a evolução dramática nas plataformas de hardware desde o nascimento dos jogos de computador tornou possível a inclusão de comportamento complexo em entidades de programa.

Levando em consideração que transformar um programa em um jogo divertido é um processo experimental por natureza, é necessário dotar a definição dos módulos de IA de uma interface de fácil utilização, em particular uma interface que não é *hard-coded* no programa. Para este fim, a interface de definição deve atingir níveis de abstração mais altos de modo a permitir que não-programadores experimentem com as possibilidades. Uma abstração possível é a noção de agentes de IA.

O objetivo deste trabalho é definir um *kernel* para um Gerador de Controle de Inteligência (em inglês *Intelligence Control Engine*) usando a noção de agentes da IA. A interface de definição onde o comportamento é definido será uma linguagem orientada a agentes, externa. O uso de agentes é motivado pelo seu alto nível de abstração, o que permite que não-programadores possam fazer experiências com possibilidades de comportamento.

Como a programação de IA é inerentemente complexa, este *kernel* deve ser um módulo facilmente anexável de modo a reusar código confiável.

ABSTRACT

Recently, the importance of AI in the process of game development has seen a very noticeable increase as the programming effort is shifting from sound and graphics to game play.

Also, the dramatic evolution in hardware platforms since the birth of computer games has made possible the inclusion of complex behavior into program entities.

Taking into account that turning a program into an enjoyable game is an experimental process by nature, it is necessary to provide an easy interface to define AI entities, in particular one that is not hard-coded in the program. To this point, higher levels of abstraction must be attained by the definition interface in order to allow non-programmers to experiment possibilities. One such abstraction is the notion of AI Agents, as their definition tries to mimic mental structures.

The objective of this work is to define a *kernel* for an Intelligence Control Engine using the notion of AI Agents. The definition interface where behavior is defined will be an external, agent-oriented language. The usage of agents is due to its high level of abstraction, which allows non-programmers to experiment with behavior possibilities.

As AI programming is inherently complex, this *kernel* should be an easily attachable module in order to reuse reliable code.

1 INTRODUÇÃO

A indústria dos jogos tem passado por grandes evoluções desde seu início, aproximadamente na década de 80. Nos seus primórdios, os jogos eram extremamente simples do ponto de vista técnico. Com a evolução de processadores e o aumento da capacidade de memória, os jogos ganharam uma nova dimensão e atraíram maior público. A primeira grande evolução foi na área dos gráficos, que hoje em dia é uma parte fundamental de qualquer produto. Em segundo lugar, veio a evolução dos efeitos sonoros e periféricos, que adicionaram uma maior imersão do jogador no mundo imaginário de cada jogo. Atualmente a mais nova evolução tem se dado na área da jogabilidade, ou seja, apresentar para o jogador um ambiente virtual com maiores desafios e credibilidade. Esta é a parte onde o campo de Inteligência Artificial se introduz nos jogos. Dentro deste campo, um dos principais desafios é a utilização e modelagem da IA de modo a permitir maior flexibilidade para a experimentação de possibilidades.

Uma das principais áreas que vem tomando impulso neste processo de evolução é a definição de comportamentos para as entidades contidas nos mundos virtuais dos jogos. Diversas correntes de projetistas de jogos acreditam que o limite de eficácia para o modelo atual de desenvolvimento e definição de comportamentos foi atingido. Para sanar esta deficiência, diversas alternativas estão sendo pesquisadas pelas companhias produtoras de jogos. Infelizmente existe uma certa distância entre a pesquisa em IA em nível acadêmico e a pesquisa realizada nas empresas, desta forma apenas técnicas já estabelecidas estão em uso ou em vias de serem aplicadas em produtos. Dentro do contexto de pesquisa acadêmica, a utilização de agentes computacionais vem ganhando destaque nos últimos anos, uma vez que estes são vistos com entidades definidas com um alto grau de abstração facilitando a modelagem de sistemas.

Neste contexto é apresentado ao leitor um resumo da história desta indústria, a evolução do processo de desenvolvimento de jogos, de seu início humilde até a indústria de bilhões de dólares dos dias atuais, apresentando o que é considerado estado da arte deste mercado atualmente. Desta forma espera-se vislumbrar mais claramente os rumos que estão sendo tomados para a melhoria do desenvolvimento de jogos. Ao mesmo tempo para que melhor se compreenda o que são agentes, informações sobre sua taxonomia, comunicação e algumas noções sobre os diversos tipos de agentes são apresentadas neste trabalho.

Ao se observar a necessidade de um novo paradigma para definir comportamentos na modelagem de programas dentro da indústria de entretenimento eletrônico, e considerando as características positivas da utilização do conceito agentes computacionais, define-se o ICE, sigla para *Intelligence Control Engine*. O ICE consiste de um *kernel* (núcleo) de comportamento baseado em agentes para jogos, com o propósito de facilitar o desenvolvimento de entidades inteligentes em um jogo.

O relatório a seguir representa o requisito final de conclusão da disciplina de Trabalho de Conclusão 2 do curso de Informática da Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul. A principal área de pesquisa contemplada no seu conteúdo é a Inteligência Artificial, dando enfoque a aplicação da mesma na área de entretenimento eletrônico, em especial aos jogos de computador.

O conteúdo do texto apresentado neste trabalho assume conhecimentos em nível de graduação em informática, são fornecidas explicações mais detalhadas quando algum conteúdo mais avançado é apresentado.

2 OBJETIVOS

O objetivo deste trabalho é definir um *kernel* que permita a definição de comportamento de entidades a serem utilizadas no âmbito de jogos de computador. A definição deste comportamento é abstraída utilizando agentes definidos através de uma linguagem que visa facilitar o trabalho do desenvolvedor, seja ele da área técnica ou não.

Para este fim, foi criado um protótipo de núcleo, ou *kernel*, de inteligência artificial. Dentre os objetivos principais do mesmo está um alto grau de modularidade, de modo que ele possa ser conectado com projetos de novos jogos facilmente, e desta forma poupar tempo para os programadores. Além disto, um módulo externo ao jogo pode ser mantido separadamente minimizando defeitos.

Como consequência da necessidade de modelagem externa dos dados, foi criada uma linguagem de definição comportamental orientada a agentes. Esta linguagem tem como objetivo principal proporcionar um alto nível de abstração para quem a utiliza.

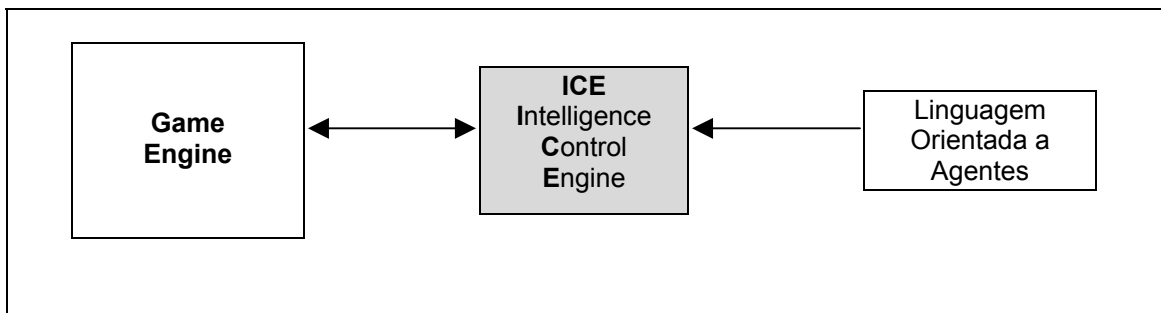


Figura 1 Contexto da arquitetura ICE dentro do jogo

Deste modo, pretendeu-se centrar toda a tarefa de IA em um núcleo, oferecendo o poder de modelagem do ICE para os desenvolvedores e ao mesmo tempo cortando o tempo de desenvolvimento, como mostra a arquitetura da Figura 1. O projeto ICE tem como característica o código aberto e uma arquitetura projetada para ser modificável, portanto, poderá ser

modificado de acordo com as necessidades peculiares de possíveis desenvolvedores.

É interessante salientar neste ponto que, apesar de extremamente importante no mercado atualmente, o desenvolvimento da apresentação de um jogo, especialmente a parte gráfica, não foi o foco deste trabalho. Este desvio de atenção se deveu principalmente a restrições de tempo, uma vez que como já citado, o desenvolvimento completo de jogos é uma tarefa que envolve um grande número de pessoas por um longo período de tempo.

3 AGENTES

3.1 Inteligência Artificial

Uma das maiores áreas dentro da Ciência da Computação atualmente é a Inteligência Artificial. Esta área é abrangente de tal modo que há quem diga poderia ser ela mesma uma ciência independente [MIN92]. IA é uma área onde não há muito consenso em relação à solução dos problemas, sendo um dos principais motivos disto à extrema dificuldade de caracterizar-se o termo Inteligência, e mais ainda adicionando-se o termo Artificial, uma vez que quanto mais se desenvolve a IA, mais restritivo o termo Inteligência se torna [DAV96]. Além do mais, dos diversos problemas que a IA se propõe a resolver não existe uma maneira universal de tratá-los, mas sim um conjunto muitas vezes vasto de soluções, dentre as quais sobressaem algumas como mais ou menos adequadas, sendo desejável ao pesquisador em IA conhecer diversas maneiras de resolver o problema para poder selecionar a mais adequada para um determinado caso [MIN92]. Este fato torna esta área extremamente multidisciplinar, fazendo com que freqüentemente cientistas de outras áreas migrem gradualmente para a IA, pois nela encontram as ferramentas e o vocabulário para sistematizar e automatizar as tarefas intelectuais nas quais têm trabalhado.

Dentre as diversas soluções da IA pesquisadas atualmente figura o conceito de Agente, inserido no contexto de Inteligência Artificial Distribuída, ou DAI¹. Este conceito é, na opinião deste grupo, uma das soluções com potencial de ser a mais adequada para a questão proposta neste trabalho, ou seja, a modelagem de jogos. Não é interessante se aprofundar neste momento nos motivos que levaram o grupo a escolher este tipo de solução, uma vez que isto será tratado no capítulo 5. Porém, é interessante notar que agentes se prestam principalmente a representar entidades individuais, auto-contidas, e inseridas em ambientes heterogêneos. Portanto, o moderno conceito de agentes

¹ Do inglês Distributed Artificial Intelligence

inteligentes tem raízes que se estendem desde os fundamentos de Inteligência Artificial, computação orientada a objetos e sistemas distribuídos, até áreas da filosofia e ciências sociais e econômicas [RUS94] e [WEI99].

3.2 Introdução aos Agentes em IA

3.2.1 Origens dos Agentes

Nos primórdios do desenvolvimento da computação, o computador não era muito mais do que uma máquina de calcular bastante poderosa. Naquela época, utilizava-se o computador para fazer inúmeros cálculos em seqüência para aplicações críticas, cálculos estes que demorariam tempo demais se efetuados por seres humanos, de modo que quando prontos eles não mais seriam úteis. Um exemplo clássico desta utilização é que os primeiros computadores eram utilizados primariamente para computação de trajetória de projéteis, mais tarde para cálculo da trajetória balística de mísseis estratégicos intercontinentais.

Ainda não se falava em diferenciação entre software e hardware, pois os dispositivos computacionais eram máquinas dedicadas a funções específicas. Quando os dispositivos finalmente foram divididos entre máquina e programa, começou a noção de flexibilidade. Este conceito expandiu-se ainda mais com o advento das primeiras linguagens de programação, pois com elas o computador se aproximou do ser humano, deixando de ser simplesmente um processador de código binário. Com as facilidades das linguagens de programação, o desenvolvimento de programas acelerou-se gradativamente, tanto em tamanho como em complexidade. Este aumento muitas vezes ultrapassa a capacidade do ser humano em termos de organização e compreensão dos meandros de funcionamento de um programa. Desde então a busca por flexibilidade e aproximação com o ser humano têm se tornado uma constante principalmente na área de engenharia de software. O desenvolvimento de jogos, como pode ser visto no capítulo seguinte, tomou um curso muito semelhante em sua evolução [WOO99].

Mesmo com este grau de evolução no campo de engenharia de software, nossa relação com o computador essencialmente ainda se mantém a mesma. Ao se desenvolver um software, tanto projetistas como programadores devem ser extremamente meticulosos, ou seja, devem pensar em todas as possibilidades de situação que o programa irá encontrar. Caso surja uma situação inesperada este programa irá, na melhor das hipóteses “pendurar”, podendo na pior das hipóteses causar um acidente de conseqüências catastróficas [WOO99], ou no caso de jogos, acabar totalmente com o desafio de um jogo deixando a sua “inteligência artificial” perdida.

Nestes, e em um número crescente de casos, a solução ideal seria que o programa pudesse decidir por si mesmo, bastando especificar de uma forma genérica o que ele deve fazer, de certa forma, de uma maneira semelhante ao que se faz com seres humanos [WOO99]. No caso de jogos de computador, arriscamo-nos a dizer que é ainda mais importante a capacidade de lidar com situações inesperadas, uma vez que quem está gerando entradas para o programa é um ser humano, que têm reações inerentemente complexas.

Estes fatores tornaram os agentes o foco de um enorme conjunto de interesses de comunidades de pesquisa acadêmica e industrial, desenvolvedores de software, principalmente os voltados para o entretenimento, entre outros. Os avanços em arquitetura de software, representação de métodos e tecnologias de resolução de problemas têm proporcionado o aumento crescente da capacidade das plataformas de computação e uma ampla gama de novos conteúdos, visões e possibilidades de agentes inteligentes serem integrados a sistemas, no intuito de torná-los mais independentes.

3.2.2 Aplicações atuais

Os ambientes de informações modernos têm se tornado amplos, abertos e largamente autônomos e distribuídos. Recentes estudos introduzem agentes como ambientes que partilham estas características. Sua maior

dificuldade para prover uniformidade e consistência é o dinamismo dos ambientes de informação, que impulsionam as interfaces a ter assistentes pessoais ativos e adaptativos – em outras palavras, *agentes*.

Aplicações que acessam e/ou filtram informações, comércio eletrônico, gerenciamento de *workflow*, produção inteligente, educação, e entretenimento (mais especificamente, jogos) estão cada vez mais se tornando predominantes. O que estas aplicações têm em comum é a necessidade de mecanismos de busca, unificação, aproveitamento, apresentação, gerenciamento, propaganda e atualização de informações. Desde que a base do ambiente seja aberta - onde as fontes de informação são autônomas e heterogêneas e podem ser adicionadas ou removidas dinamicamente – os mecanismos associados devem ser extensíveis e flexíveis. Visivelmente, mais pessoas estão chegando à conclusão de que agentes são partes integrais desses mecanismos, e que sua aplicabilidade na indústria de jogos é conseqüentemente essencial devido às exigências cada vez maiores de seus consumidores.

3.2.3 Definições Básicas sobre Agentes

Muitas são as tentativas de definição para agentes e de especificação de suas características. Entretanto há um ponto em comum entre conceitos e opiniões. Podendo-se sintetizá-las em uma única definição: *agentes são ativos, são componentes persistentes (software) com percepção, objetivos, ação e capacidade de comunicação* [GAS98].

Um agente é qualquer entidade que *percebe* seu ambiente através de sensores e *age* sobre ele através de atuadores. Um exemplo de modelo e sua arquitetura básica podem ser facilmente compreendidos observando a Figura 2.

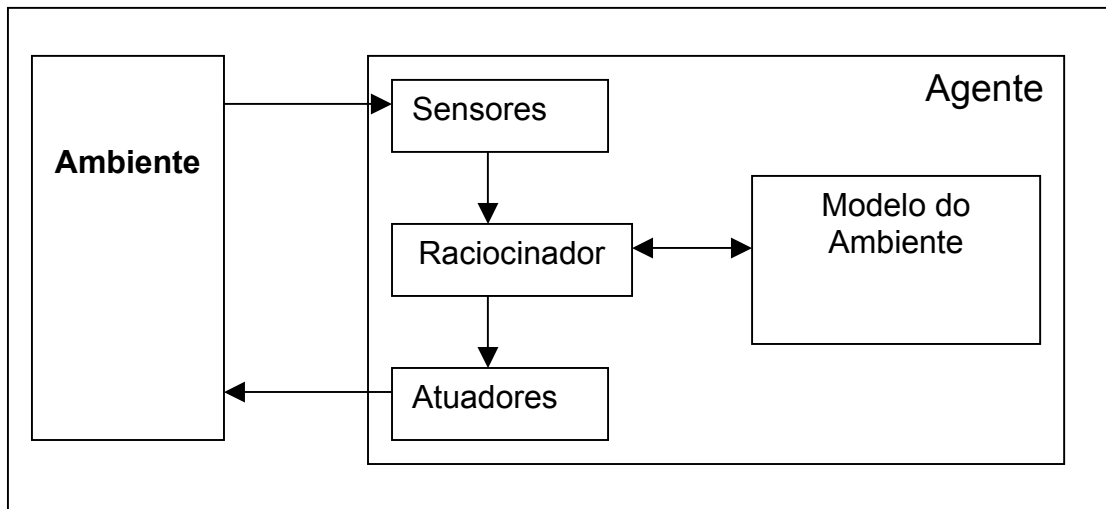


Figura 2 Arquitetura básica de um agente

Através dos sensores o agente capta as informações e estímulos provenientes do meio externo. O raciocinador, através das regras definidas em seu modelo de ambiente, pode validar e atualizar suas informações de forma que seus atuadores tenham condições de interagir com o ambiente externo.

Há duas visões de agentes distintas, segundo [GAS98]:

- Uma, considera os agentes essencialmente conscientes, entidades cognitivas que têm sentimentos, percepções, e emoções como seres humanos.
- Outra visão alternativa é que agentes são meramente autômatos e comportam-se exatamente tal qual foram programados. Esta visão admite uma ampla variedade de computações, incluindo computação de agentes.

A segunda visão de [GAS98] vem ao encontro da definição de [WOO99] que define agente como qualquer entidade que perceba o ambiente, “raciocine” sobre o mesmo e tenha condições de agir sobre ele. Como exemplo desta definição, o autor cita o termostato de um ar-condicionado, que percebe que a temperatura ambiente está no patamar definido e desliga o mecanismo de resfriamento do ar. Outro exemplo desta definição seria um *dæmon* de UNIX, que percebe ocorrências no ambiente do SO, processa o que deve ser feito e atua sobre o mesmo.

[WOO99] estende sua definição para incluir o que ele chama de “agentes inteligentes”, que vem ao encontro da primeira visão de [GAS98]. De acordo com Wooldridge, o que caracteriza inteligência é a flexibilidade, logo agentes inteligentes devem, acima de tudo serem capazes de ação autônoma e flexível de modo a cumprir com suas metas. Neste contexto flexibilidade inclui três características:

- Reatividade: o agente percebe o ambiente e responde no momento correto a mudanças no ambiente de modo a cumprir com seus objetivos;
- Pro Atividade: agentes inteligentes comportam-se norteados por objetivos e devem tomar a iniciativa para cumpri-los;
- Habilidade social: agentes inteligentes são capazes de interagir com outros agentes e possivelmente, também com seres humanos de modo a chegar aos seus objetivos.

Em um jogo, o jogador normalmente prefere que seus oponentes sejam mais imprevisíveis, que eles analisem o ambiente em volta e desenvolvam uma estratégia mais realista, ou seja, que simulem o comportamento humano. Dentre as visões citadas anteriormente a que mais corresponderia seria a segunda, pois esta espelha com maior fidelidade o raciocínio humano, porém esta é computacionalmente limitada devido aos recursos tecnológicos disponíveis nos dias de hoje. Apesar disto, a pró-atividade pode ser complementada pelas alternativas apresentadas.

Os agentes representam diferentes entidades que colaboram para encontrar e unificar soluções, competindo por recursos e fontes de informações e são melhor aproveitados como partes de um sistema multi-agente, não tanto quando isolados. A filosofia multi-agente poderia ser transposta para um ambiente de jogo onde, por exemplo, os oponentes podem combinar estratégias de modo a armar ciladas contra jogador. Porém estes oponentes também deverão combinar a divisão de recursos, tais como armas ou qualquer outro tipo de itens entre eles.

Uma propriedade interessante dos agentes é que eles provêm um meio natural de executar tarefas sobre ambientes incontroláveis. Quanto mais agentes, mais simples (subdividido) fica o problema. No entanto, mais complexa fica a comunicação e a coordenação entre eles.

3.3 Taxonomia

3.3.1 Agentes

A taxonomia dos agentes requer características consideradas chave no que diz respeito à identificação do próprio agente, e do total de agentes no sistema multi-agente ao qual ele participa.

Segundo [FRA96] as características dos agentes estão fundamentalmente interligadas não somente as suas propriedades intrínsecas, que são definidas para o agente em particular, mas também suas propriedades extrínsecas, que são definidas para um agente no contexto de outros agentes, ou seja, no meio em que ele está inserido.

Podemos citar como exemplo de características intrínsecas [GAS98]:

- Duração (transitória ou permanente)
- Nível de cognição (reativo ou deliberativo)
- Construção (declarativa ou procedural)
- Mobilidade (fixa ou móvel)
- Adaptabilidade (capacidade de aprender)
- Modelagem (do ambiente, deles mesmos, ou de outros agentes)

Como exemplo de características extrínsecas [GAS98]:

- Localidade (local ou remota)
- Autonomia social (independente ou controlada)
- Sociabilidade (autística – alienado de seu ambiente; ciente – prevenido, sabedor; responsável; atrelado ao grupo)

- Grau de integração (cooperativa, competitiva ou antagônica)
- Interações (logística – direta ou via promovedores, mediadores ou não-agentes; estilo / qualidade / natureza – com agentes / mundo / ambos; nível semântico – comunicações declarativas ou procedurais).

Considerando que agentes são estruturas computacionais, e as características dos mesmos utilizam uma terminologia humano-cognitiva, estes proporcionam uma interface de definição computacional mais compreensível. Ou seja, definem características como sociabilidade, tipo de integração, adaptabilidade, que são características essencialmente humanas. Tem-se então uma aproximação cada vez maior do comportamento dos agentes com o comportamento humano. De certa forma, isso tornaria um jogo muito mais atraente e interativo, pois seus personagens agiriam como seres pensantes, ativos. Para a maioria dos jogadores a emoção de um jogo está principalmente no fato de poder “sentir” a realidade mostrada com o maior grau de fidelidade possível do mundo real.

3.3.2 Ambientes

Segundo [WOO99], um agente é um sistema computacional que está situado em algum ambiente, e que recebe a saída dos atuadores e fornece a entrada dos sensores. De acordo com a Figura 3:

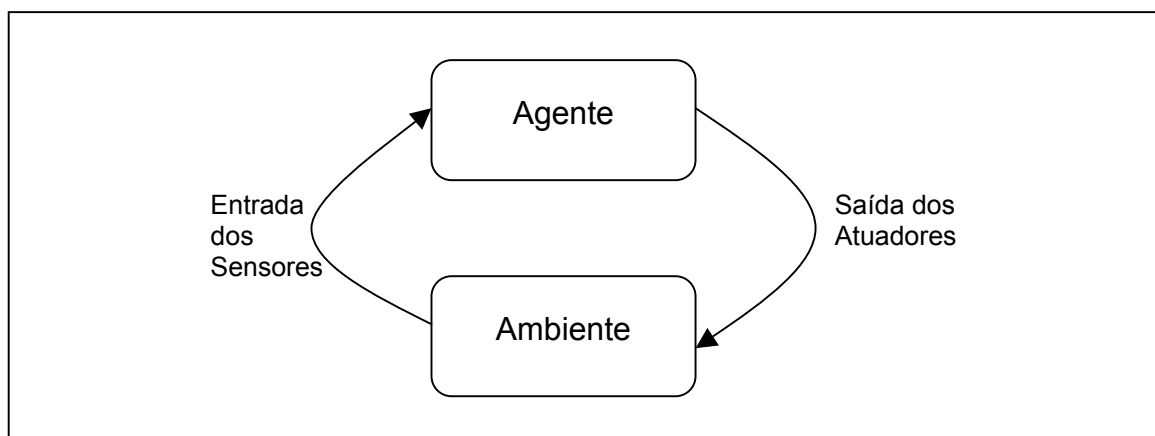


Figura 3 Visão abstrata da interação agente-ambiente [WOO99]

Em geral o agente não tem controle total sobre o ambiente onde está situado, de acordo com [WOO99], na melhor das hipóteses ele terá controle parcial sobre o ambiente, isto é, ele poderá influenciar o mesmo. Neste contexto, a capacidade do agente influenciar o ambiente está limitada pelo conjunto de ações que o agente pode executar sobre o mesmo. Estas ações podem falhar, e nem sempre podem ser executadas, por exemplo, um soldado não pode realizar uma ação do tipo atirar caso sua arma não esteja carregada.

Logo, a complexidade de um agente é proporcional à complexidade do ambiente em que ele está inserido, e, portanto, o processo decisório é afetado por diversas características do ambiente. [RUS94] sugere a seguinte classificação das propriedades do ambiente, retirada de [WOO99]:

- Acessível versus inacessível: refere-se à capacidade de se obter informações completas sobre o ambiente. Neste sentido, o mundo real em geral é inacessível para os seres humanos. Quanto mais acessível o ambiente, mais simples será construir agentes para operar no mesmo;
- Determinístico versus não-determinístico: um ambiente determinístico reage como uma função, ou seja, existe um resultado garantido para cada ação. O mundo físico, para todos os efeitos, é não-determinístico;
- Episódico versus não-episódico: em um ambiente episódico, a performance do agente é baseada apenas em um número de eventos discretos, ou seja, o agente não precisa levar em consideração a interação de um episódio com episódios futuros;
- Estático versus dinâmico: num ambiente estático pode-se assumir que este não irá mudar exceto pelas ações do agente. Neste contexto, o mundo real é um ambiente altamente dinâmico;
- Discreto versus contínuo: um ambiente é discreto caso exista um número fixo e finito de ações e percepções nele. Como exemplo de ambiente discreto, [RUS94] cita um jogo de xadrez, e como contínuo, a condução de um táxi.

3.4 Modelos de Agentes

É possível dividir os agentes quanto a sua implementação em quatro classes, de acordo com [WOO99]:

- Agentes baseados em lógica: onde a tomada de decisão é feita através de dedução lógica;
- Agentes reativos: onde a tomada de decisão é feita através de algum tipo de mapeamento direto de situação para ação;
- Agentes BDI: onde a tomada de decisão é resultado da manipulação de estruturas de dados representando as crenças, desejos e intenções do agente;
- Arquiteturas em camadas: onde a tomada de decisão é feita através de diversas camadas de software que raciocinam em algum nível sobre o ambiente.

Dentre estas classes, os agentes podem ser classificados essencialmente em reativos, com apenas um representante óbvio nesta classificação, ou deliberativos, representados pelas arquiteturas BDI e baseadas em lógica.

As primeiras implementações de agentes se encaixam na primeira classe de agentes. Estas implementações representam o conhecimento do agente através de um conjunto de afirmações em lógica e o comportamento dos agentes é obtido através de provas de teoremas sobre o conjunto de premissas que representam o conhecimento do agente. Este tipo de aproximação na criação de agentes possui dois problemas segundo [WOO99]: a velocidade de decisão é demorada, e a definição deste tipo de agente é excessivamente complexa. Uma vez que um dos objetivos deste trabalho é facilitar a definição de comportamento, e proporcionar um modelo de agentes que possa se adequar às necessidades de velocidade para a aplicação em jogos, este modelo foi considerado incompatível com os objetivos propostos pelo grupo.

A classe de arquiteturas em camadas também é bastante interessante, pois utiliza deliberação e reatividade para atingir comportamento inteligente, sendo considerada híbrida. A arquitetura proposta neste volume utiliza-se da aproximação híbrida para combinar agentes reativos e BDI. Desta forma este trabalho irá se concentrar nos agentes reativos, para a obtenção de respostas rápidas, e nos agentes BDI para a definição de comportamento pró-ativo e planejamento. Estes modelos serão descritos a seguir.

3.4.1 Agentes Reativos

Rodney Brooks em [BRO86] deu voz a um grupo de pesquisadores que era contra a corrente simbólica da IA, corrente esta que foi a fonte da aproximação em lógica dada inicialmente aos agentes. Este artigo lançava as bases para os agentes reativos, que diferem essencialmente dos agentes deliberativos na ausência de uma representação interna do mundo, segundo Brooks, “O mundo é seu próprio modelo”. Também de acordo com os defensores da aproximação reativa, o comportamento racional é visto como intimamente ligado ao ambiente que o agente ocupa, ou seja, o comportamento inteligente não é disjunto de um corpo, mas sim o resultado da interação do agente com o ambiente que ele ocupa, e da interação e cooperação de diversos agentes [WOO99].

Essencialmente os agentes reativos são aqueles cujo comportamento consiste em reagir aos estímulos do meio, o que se constitui em uma vantagem quando se fala na atuação em ambientes dinâmicos, pois os sinais oriundos dos sensores geram uma ação imediata nos atuadores. As arquiteturas que se baseiam neste modelo de agente visam construí-los para que decidam suas ações em tempo de execução. Ao invés de atuarem em função de uma representação interna do mundo, estes agentes atuam em função de seu estado interno e do estado do ambiente a cada momento. As decisões são tomadas em tempo real, baseando-se em poucas informações e regras simples, que definem a ação em função de uma situação. Apesar desta vantagem, os agentes reativos são deficientes em relação à realização de

tarefas que exigem um comportamento mais complexo e dirigido a determinado objetivo, porque eles não apresentam a capacidade de elaborar previamente uma seqüência de ações que os leve até isso. Entretanto, conceitos semelhantes como, por exemplo, regras de reação a eventos, individualizadas por entidade¹, são largamente utilizados atualmente na indústria de jogos [RAB00b], dada a sua rapidez de resposta, característica necessária caso se tenha como objetivo que diversas entidades respondam em tempo real.

3.4.2 Agentes BDI

O segundo modelo de agentes considerado neste trabalho ultrapassa a falta de pró-atividade dos agentes reativos: são os agentes deliberativos, aqui representados pelos agentes BDI. Estes são autônomos e mantêm uma representação simbólica do mundo e das ações, sobre a qual pode fazer raciocínios lógicos e planejamento de um curso de ações para alcance de um objetivo. Como o próprio nome já diz, estes agentes são capazes de realizar deliberação, sendo uma vantagem na solução de tarefas que exigem coordenação de ações, pois são capazes de elaborar planos prevendo o resultado de suas ações e, portanto, possuem um comportamento mais racional, no sentido de agir em busca de um determinado objetivo. Contudo estes agentes têm certa dificuldade no tratamento de alterações no ambiente, principalmente aquelas que exigem resposta em tempo real.

Os ambientes dinâmicos exigem dos agentes deliberativos uma constante monitoração do ambiente e atualização de suas representações simbólicas. Isso pode levar o agente a tomar decisões quando estas já não são mais necessárias. Para acrescentar reatividade aos agentes deliberativos não é exigida nenhuma estrutura ou característica que já não esteja presente em sua arquitetura. Entretanto há a necessidade de sensores adequados para que os agentes deliberativos possam ser capazes de gerar respostas ao que ocorre no ambiente em que se encontra. Um fator determinante no desempenho desse tipo de agente acrescido de reatividade é o processo de deliberação.

¹ Na opinião do grupo este é apenas um uso informal de agentes reativos

Este processo pode dificultar a geração de respostas em tempo real quando estas são assim exigidas, pois a deliberação na escolha de uma ação pode demorar tempo suficiente para que a ação já não seja mais necessária naquele momento em que será executada.

Agentes BDI (*Belief-Desire-Intention*) são também sistemas deliberativos, na medida em que baseiam sua atuação em uma representação simbólica e em um raciocínio lógico sobre esta. Eles apresentam estados mentais (crenças, desejos e intenções) que são processados internamente e ações que são baseadas nestes estados. Crenças são um estado mental representando o conhecimento do agente a respeito do mundo e sobre si mesmo. Desejos são predisposições a determinadas realizações. Intenções são um novo estado que irá representar o comprometimento do agente com algumas destas realizações. Claro que a respeito deste modelo de agentes tem-se muitos fatores relevantes, como por exemplo, inconsistências nestas bases de conhecimento. Na teoria de Bratman, descrita em [BRA90], a inconsistência entre desejos é permitida e o agente não precisa acreditar que seus desejos sejam possíveis. Neste estado não há comprometimento com sua realização. Mas a partir do momento que este se torna uma intenção não pode haver inconsistências. Pois as intenções norteiam a atuação.

Este tipo de arquitetura é, no escopo dos jogos interessante para a definição de entidades cujo comportamento deve se assemelhar a comportamento inteligente. A definição de comportamentos complexos é bastante limitada no modelo reativo, uma vez que regras estáticas de condição e ação não representam exatamente a maneira como seres inteligentes conduzem seu raciocínio. Desta forma a utilização de abstrações mentais como crenças e desejos facilita o processo de definição deste tipo de comportamento.

3.4.3 Arquiteturas Híbridas

Pensando nas facilidades providas pela reatividade e na complexidade intrínseca ao processo de deliberação, pode-se pensar em uma

arquitetura que una os princípios da arquitetura reativa e os mecanismos da arquitetura deliberativa, compondo-se uma arquitetura híbrida. Em arquiteturas híbridas têm-se um processo de geração de ações em resposta a estímulos do meio independente do processo de deliberação que gera as ações necessárias para satisfação dos anseios do agente através de um planejamento das mesmas. Isto eliminaria em parte os problemas citados anteriormente em relação à combinação destas duas abordagens. Na verdade, a arquitetura do ICE não propõe uma nova forma de implementar uma funcionalidade ou outra, mas uma nova forma de integrar um módulo reativo a um processo de planejamento, no intuito de melhorar a performance global do agente na solução de tarefas complexas em um mundo dinâmico.

3.5 Arquiteturas de Agentes

Diversas arquiteturas utilizando os diversos modelos de agente já foram desenvolvidas com diversos objetivos. Desta forma, algumas delas foram estudadas de modo a analisar as dificuldades e limitações encontradas por outros pesquisadores. Também é interessante que algumas delas sejam citadas a título de comparação. A utilização de uma arquitetura híbrida de agência como na arquitetura ICE não é de forma alguma inédita, tendo sido utilizada, por exemplo, na arquitetura IRMA, citada a seguir e em [DHE00].

3.5.1 IRMA

IRMA (*Intelligent Resource-bounded Machine Architecture*) baseia-se na modelagem de “agentes limitados pelos recursos”, onde esta limitação refere-se primariamente ao poder de processamento. Ela apresenta uma especificação de alto nível do componente “raciocínio prático”, ou seja, inclui direta representação de crenças, desejos e intenções na arquitetura. Além do mais, a arquitetura IRMA apresenta diferentes bancos de informações (crenças, desejos, biblioteca de planos, intenções estruturadas em planos) e processos [HAD96]. Veja Figura 4 abaixo:

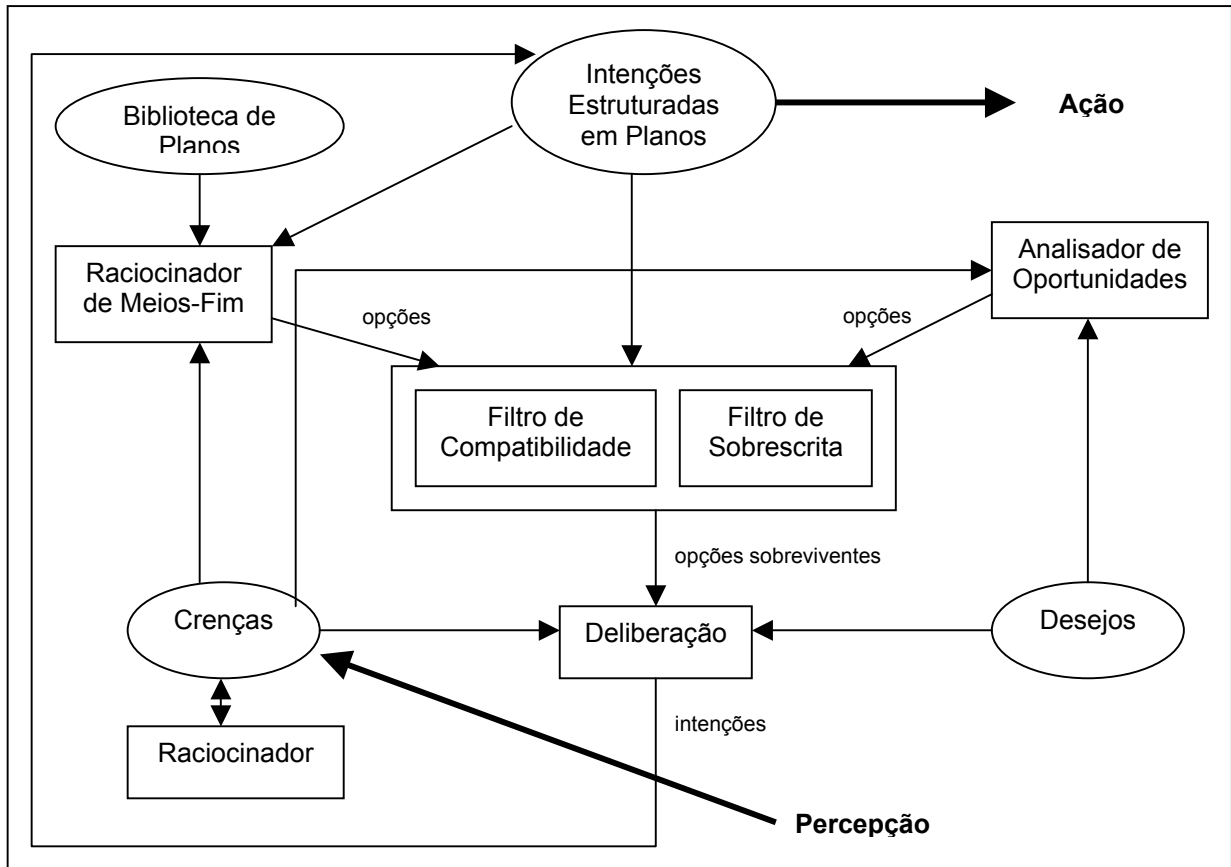


Figura 4 Estrutura da arquitetura IRMA, extraído de [DHE00]

Há uma distinção entre os planos existentes: têm-se alguns em forma de “receitas” (procedimentos) e outros, que são escolhidos por um agente para executar. Uma biblioteca de planos contém seus planos em forma procedural e pode ser vista como um subconjunto de crenças dos agentes (qual plano é aplicável para resolver esta ou outra tarefa, e em que circunstâncias). O último tipo de planos é aquele que o agente frequentemente segue, correspondendo às intenções que estão armazenadas no componente das “intenções estruturadas em planos”. Neste ponto é que o agente realmente se compromete com um fim, mesmo sem ter feito a deliberação sobre todos os meios para atingi-lo. Uma vez que uma intenção é formada, o componente “raciocinador meios-fim” é invocado para cada plano parcial existente, propondo sub-planos que o completem – este componente pode propor inúmeras opções para um mesmo objetivo; estas, juntas, passam pelo filtro do Analisador de Oportunidades, um componente que também propõe opções, mas estas são o resultado das mudanças no ambiente. Tais opções

podem ser boas ou nem tanto, dependerá do ponto de vista do agente; por exemplo, se forem opções inesperadas, mas que mesmo assim satisfazem seus desejos, não deixam de ser também uma forma de tratamento possível para determinada situação.

O Filtro de Compatibilidade recebe estas opções, oriundas do Raciocinador de Meios-fim e do Analisador de Oportunidades e testa sua compatibilidade com relação às intenções já seguidas pelo agente. É necessária esta compatibilidade, pois às vezes são inseridas novas crenças, fazendo com que planos com que o agente tenha se comprometido anteriormente, devam ser reconsiderados, ou por outro lado, até mesmo abandonados. As opções que não apresentarem incompatibilidade passam para o processo de deliberação, onde seus conflitos e relacionamentos com novas crenças são avaliados, passando assim, a tornarem-se intenções para o agente. As intenções geradas neste processo ainda podem servir como planos parciais para o Raciocinador Meios-fim fazendo-o gerar novas opções; ao mesmo tempo, o Analisador de Oportunidades também gera opções em função das alterações nas crenças, repetindo, desta forma, o processo.

Existe ainda um mecanismo que atua em paralelo com o Filtro de Compatibilidade que é o Filtro de Sobrescrita; ele “reaproveita” as opções descartadas por incompatibilidade analisando a possibilidade de ser descartada como intenção ou aproveitada em relação a outras opções, se estas, claro, considerarem as condições codificadas no Filtro de Sobrescrita. Segundo [DHE00], é aconselhável que este mecanismo não seja tão suscetível a reavaliar planos, pois isto aumentaria muito a quantidade de deliberação no agente reduzindo o papel do comprometimento, apesar de que o agente também não pode ser muito preso ao comprometimento deixando de reagir a mudanças importantes. Logo, tem-se como um dos resultados práticos da arquitetura IRMA.

3.5.2 PRS

PRS significa *Procedural Reasoning System* e é uma arquitetura de raciocínio e execução de tarefas em ambientes dinâmicos. Por esta característica que PRS é classificada como uma arquitetura híbrida em muitas bibliografias [DHE00]. Inicialmente ela foi desenvolvida para um sistema de controle reativo de ônibus espaciais da NASA, o que indica que o ambiente de atuação é sujeito inevitavelmente a constantes alterações.

Nela, as atitudes como crenças, desejos e intenções são representadas explicitamente, e juntas determinam as ações do sistema para alguma dada instância. Assim como a arquitetura clássica que conhecemos para agentes, com atuadores e sensores, PRS também se utiliza destes conceitos na modelagem de seus agentes. Através dos sensores o agente adquire novas crenças sobre o mundo, onde seus sinais são traduzidos para uma representação simbólica. Já as ações que compõem os planos são traduzidas em comandos para os executores. Como veremos a seguir, a arquitetura PRS é composta basicamente de cinco módulos, conforme a Figura 5.

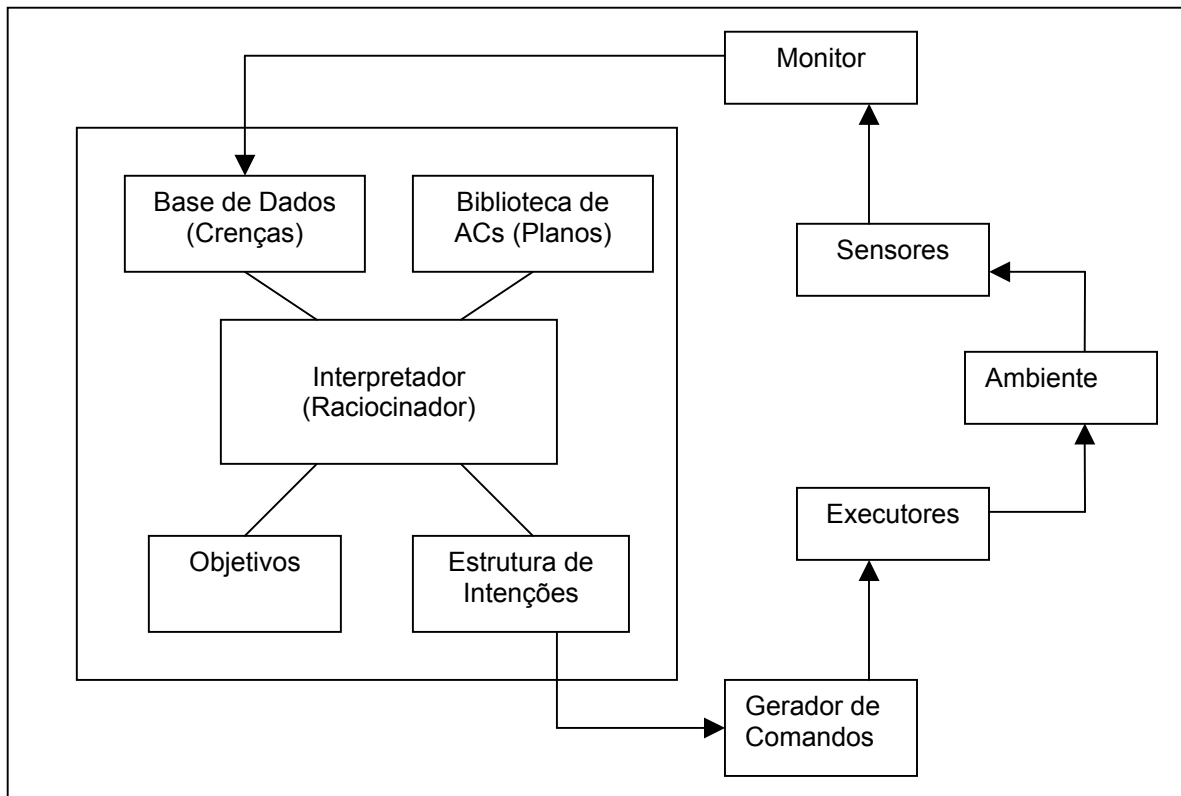


Figura 5 Estrutura da arquitetura PRS [DHE00]

Os componentes constituintes desta arquitetura são: uma Base de Dados (contendo as crenças, fatos sobre o mundo), um conjunto de objetivos (representam o comportamento desejado do sistema e do agente – substituem o os desejos previstos na arquitetura BDI), uma Biblioteca de Áreas de Conhecimento (ACs – encapsula o conhecimento sobre como cumprir tarefas e como reagir em determinadas circunstâncias), uma estrutura de intenções (planos que o sistema tem que escolher para executar imediatamente ou em alguma situação futura) e um interpretador (controla o funcionamento do agente, selecionando e colocando na estrutura de intenções os planos de acordo com as sentenças e os objetivos, executando-os).

A Base de Dados contém fatos que o agente conhece a respeito de seu ambiente e a respeito de si mesmo, chamando de crenças de meta-nível. Assim como existem alguns objetivos do agente em relação ao seu próprio comportamento interno, que são os chamados objetivos de meta-nível.

As ACs exprimem as formas que o agente tem para atingir seus objetivos ou reagir a certas condições especificadas declarativamente em

procedimentos. Cada AC tem duas partes distintas: um corpo e uma condição de invocação. Um conjunto de situações deve ser satisfeito para que determinada AC seja aceitável constituindo então a condição de invocação que é formada basicamente de crenças e objetivos. O corpo das ACs pode ser visto como um plano, pois à medida que um nó fim é atingido, o objetivo responsável por ativar o plano foi realizado [DHE00].

Na estrutura de intenções tem-se aquelas tarefas que foram selecionadas pelo sistema para execução imediata ou em algum instante futuro. Basicamente, cada intenção é formada por um AC em conjunto com outras sub-ACs (são ACS designadas a realizar objetivos parciais da AC principal) necessárias para execução da mesma, organizadas em uma pilha. As intenções apresentam três estados possíveis: ativa (próxima a ser executada ou imediatamente dependendo da estrutura de intenções), suspensa (é adotada a intenção, porém nenhuma decisão é tomada para utilizá-la, devendo ocorrer uma ativação explícita) e suspensa condicionalmente (espera satisfação de determinada condição para passar para o estado ativo).

A execução de uma intenção ocorre através da realização de ações primitivas modificando o mundo externo ou o estado interno do sistema, ou através da criação de novos objetivos. É importante ter-se procedimentos fixos de decisão, pois a partir do momento que o interpretador não mais alcança informação que indique o melhor a fazer, ele deve ter uma alternativa em forma de procedimento para sair do impasse. Esta característica facilita, segundo [DHE00] a implementação deste componente já que o sistema não ficará preso somente ao comportamento ditado por ele. Por outro lado, é importante que o Interpretador seja capaz de lidar com situações mais corriqueiras, de forma que não seja necessário um acesso constante aos ACs de meta-nível, pois isto prejudicaria fortemente o desempenho do sistema.

Como PRS também se trata de uma arquitetura BDI, o comprometimento com as intenções previamente adotadas é de fundamental importância, sendo que o raciocínio é efetuado baseando-se no conjunto de intenções existentes. Um agente comprometido em alcançar determinado

objetivo seguirá o plano estabelecido mesmo que alterações no mundo proporcionem formas melhores de alcançá-lo.

3.5.3 COSY

COSY (*Cooperating Systems*) distingue-se por ser um estudo sobre um sistema baseado em conhecimento cooperativo. Seus agentes, ditos “agentes racionais”, são capazes de trabalhar em conjunto, ordenadamente, de forma a atingirem um objetivo em comum. Esta é a maior diferença conceitual quando comparamos esta arquitetura com as outras vistas anteriormente. Seu modelo de agentes descreve o agente por suas intenções (definindo dois estados mentais distintos que veremos a seguir – táticos ou estratégicos), comportamentos (percepções, ações e comunicação) e recursos requeridos para satisfazer as intenções executando os comportamentos tais como eles o são.

A arquitetura COSY tem uma estrutura modular, contando basicamente com os seguintes módulos: Atuadores, Sensores, Comunicação, Motivações e Cognição. Os três primeiros são os componentes básicos do domínio de atuação, de forma que sua implementação fica a cargo do construtor do agente. As funções inerentes a cada um destes módulos são bastante evidentes, sendo os Atuadores responsáveis pelas ações sobre o meio, os Sensores fornecem informações atualizadas sobre o estado e os eventos do ambiente, e o módulo de Comunicação é responsável pela troca de informações com os outros agentes (através de mensagens, por exemplo).

As intenções, como citado anteriormente, podem ser estratégicas ou táticas. O módulo de Motivações compreende as intenções estratégicas, aquelas que levam em conta além dos objetivos futuros e das preferências, os atributos que definem as características do próprio agente. [HAD96] considera este conteúdo semelhante aos desejos da teoria dos BDI's. Para melhor entendermos como a arquitetura COSY está estruturada, segue abaixo um esquema básico da mesma:

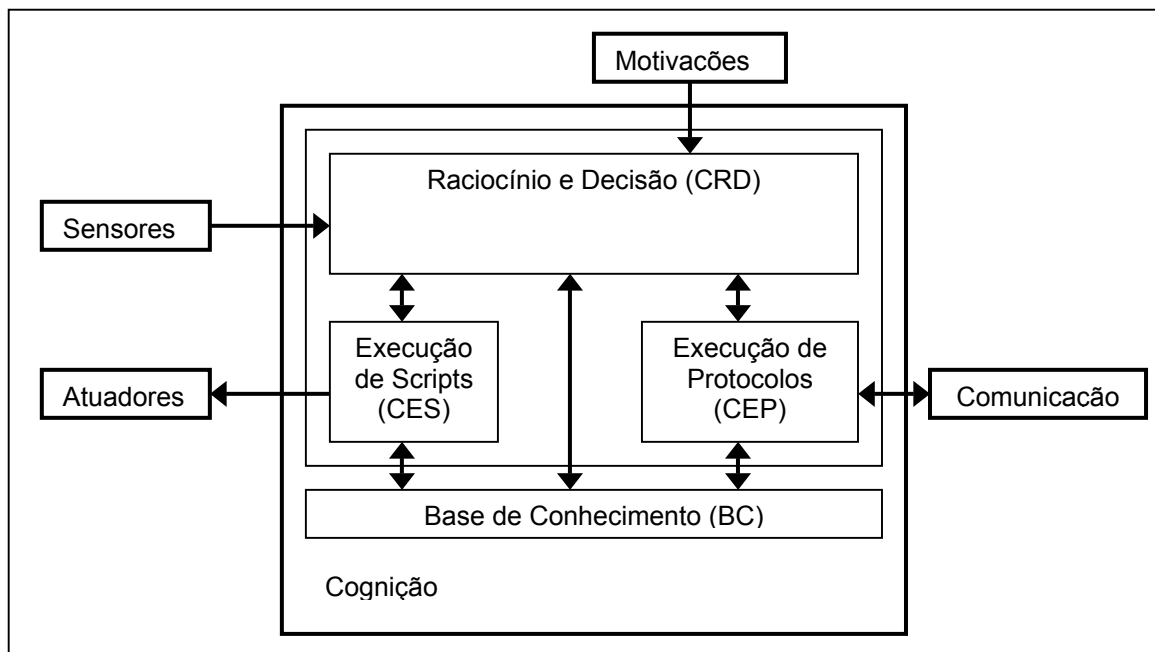


Figura 6 Arquitetura COSY [HAD96]

Um script é um procedimento estereotipado ou um plano para realizar uma tarefa específica. Este pode conter chamadas para outros scripts, protocolos e comportamentos primitivos que os atuadores executam. O componente de cognição (S) administra a execução de scripts, delegando a execução de comportamentos primitivos para os atuadores e a execução de protocolos de cooperação para o componente respectivo (P).

A tripla que caracteriza as arquiteturas BDI (*belief, desire, intention*) é representada no COSY diretamente. Assim como os protocolos de cooperação representam um “diálogo” pré-estabelecido para uma cooperação específica. O D será então responsável pelo raciocínio sobre o mundo, reagindo se a situação demanda, e deliberando sobre qual a melhor forma de realizar objetivos e satisfazer intenções.

3.5.4 GRATE

A arquitetura funcional deste agente assemelha-se com a arquitetura IRMA vista anteriormente. Porém GRATE tem componentes adicionais que incorporam a resolução de problemas de forma colaborativa. Ele utiliza os conceitos de combinação de intenções e responsabilidades para estabelecer

uma atividade colaborativa e monitorar sua execução. Como mostra a Figura 7, esta arquitetura também se utiliza dos conceitos de arquitetura BDI.

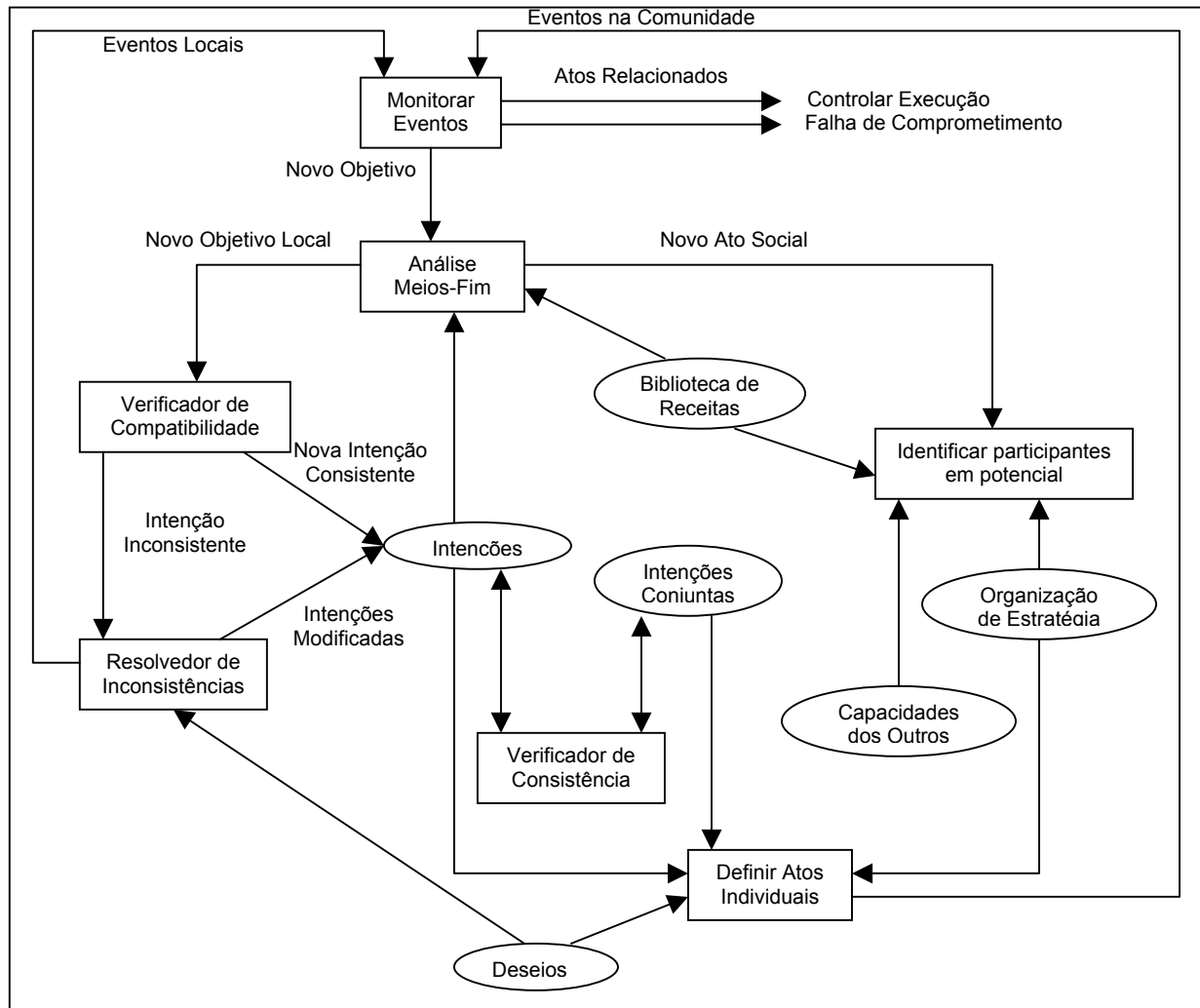


Figura 7 Arquitetura funcional da GRATE [HAD96]

Esta arquitetura considera que eventos ocorrem como resultado de uma ação local para resolver determinado problema ou por mudanças no ambiente, adicionalmente, se o evento ocorrer em meio à comunidade (ambiente de cooperação entre os agentes) é controlado por um processo Monitor de Eventos.

Eventos são a força motriz para iniciar uma atividade, e então alcançar um novo objetivo. Este servirá como entrada para o processo de Análise de Meios-fim. Este processo buscará, na Biblioteca de Procedimentos, planos apropriados para completar o objetivo em questão. Estes planos é que

indicarão se o objetivo pode ser alcançado localmente, colaborativamente, ou ainda, uma escolha entre estas duas opções pode ser necessária. As intenções aparecem aqui como recursos para priorizar determinados objetivos ou não, quem faz esta deliberação é o processo de Análise de Meios-fim. No caso do agente decidir buscar o objetivo localmente o Avaliador de Compatibilidade verificará se o objetivo e o meio de alcançá-lo são compatíveis com as intenções existentes. Em caso negativo, uma revisão é feita pelo Solucionador de Inconsistências até remover o conflito. Se o agente decidir alcançar o objetivo contando com a colaboração de outros agentes, uma ação social é estabelecida.

Em uma primeira fase, que se inicia com o estabelecimento da ação social citada anteriormente, é necessário identificar os agentes da comunidade que estão potencialmente envolvidos. Esta fase é denominada *Team Formation*. A segunda fase é a geração do plano social propriamente dito. É importante salientar que nesta arquitetura as intenções não somente provêm meios para coordenar as ações como também agem como um guia para execução das tarefas e monitoração.

3.5.5 InteRRaP

A arquitetura InteRRaP combina reatividade com deliberação e capacidades de cooperação, exemplificando com este modelo uma arquitetura híbrida. Utiliza-se das vantagens do estilo BDI, dividindo-se em três níveis de conhecimento e controle que interagem entre si, estruturados em camadas diferentes de abstração, complexidade de representação de conhecimentos e raciocínio. Veja Figura 8 abaixo:

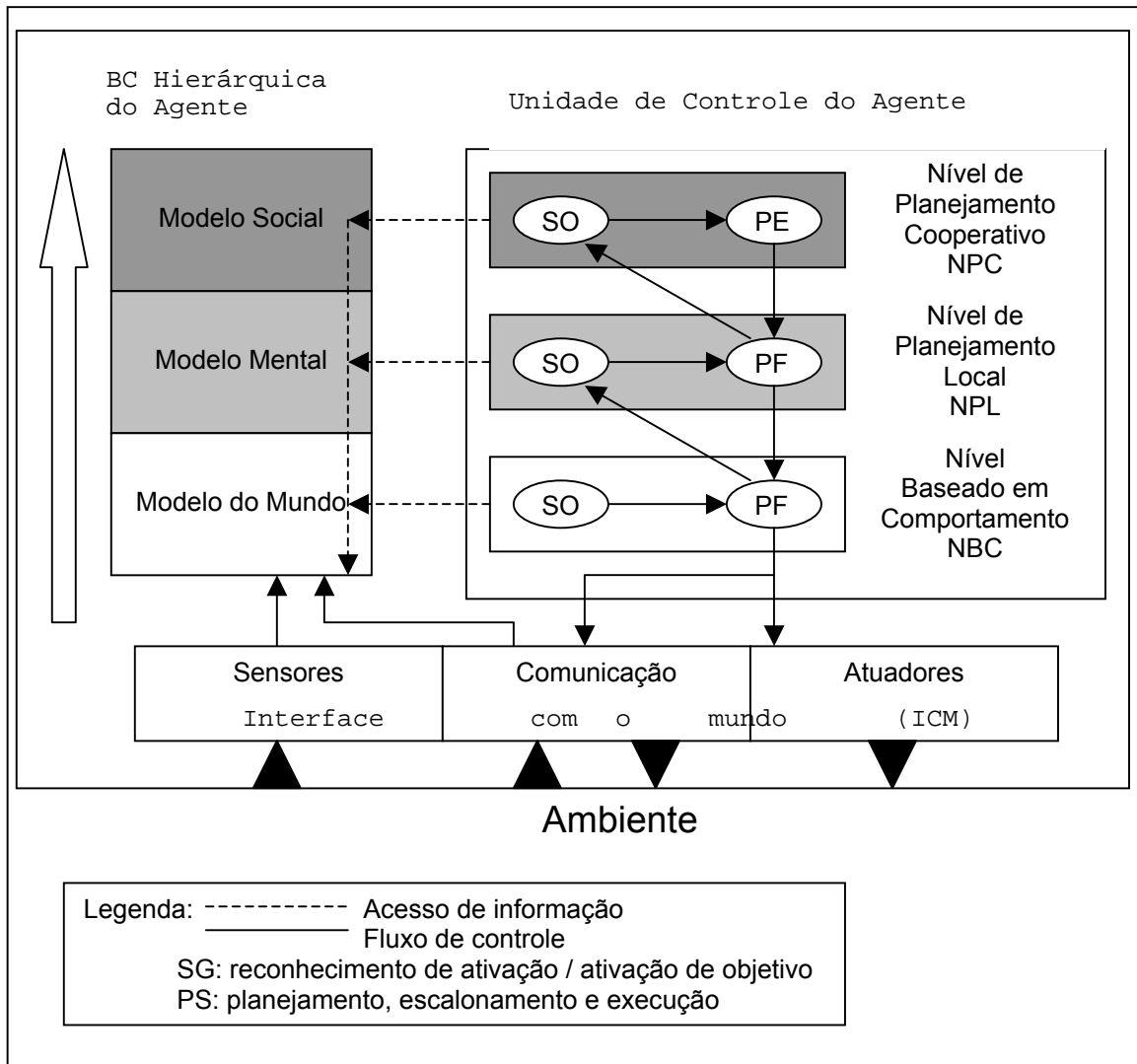


Figura 8 A arquitetura InteRRaP [FIS98]

É uma arquitetura que consegue explicar o comportamento dos seus agentes, sendo estes ditos “sociais”, ou seja, na presença de outros agentes atuam de forma cooperativa para alcançar seus objetivos, utilizando-se, é claro, de suas habilidades sociais.

Esta arquitetura descreve o agente por três módulos basicamente:

- Pela sua interface com o mundo, esta pode ser através do *sensoriamento* (capacidade de habilitar ou desabilitar os sensores, e também ler seus valores em dado instante ou o tempo todo), *ação* (execução de ações físicas do agente no ambiente) e *comunicação* (enviar e receber mensagens de outros agentes).

- Uma base de conhecimento (modelo do mundo, um modelo mental e um modelo social);
- Uma unidade de controle (UC) com três níveis de abstração, cada um com dois processos:
- Um de reconhecimento de situações e ativação (SO);
- Outro de planejamento, escalonamento e execução (PE);

Esta unidade fica, então, subdividida em três níveis: o NBC (Nível baseado em Comportamento), o NPL (Nível de Planejamento Local) e o NPC (Nível de Planejamento Cooperativo). Sendo que, em cada um desses níveis, pode-se perceber funções distintas; por exemplo, no NBC tem-se procedimentos para tarefas corriqueiras suportando também reatividade; o NPL fica responsável pelos raciocínios do agente, sendo que o mesmo deve ter comportamentos totalmente direcionados ao objetivo; já o NPC tratará de questões mais complexas, ou pelo menos de questões ainda não abordadas tão profundamente até agora, como é o caso da cooperatividade entre agentes, ou seja, utilizar-se das vantagens desta característica para a realização de tarefas em conjunto e coordenadamente.

A UC acessa constantemente a Base de Conhecimentos, podendo modificá-la com novas crenças, objetivos, e até mesmo planos hipotéticos. Vale ressaltar que a Base de Conhecimento é um recurso que pode ser acessado por cada um dos níveis da UC, porém com algumas restrições estabelecidas. Cada nível pode acessar o nível da Base de Conhecimentos referente ao seu, ou inferior a ele, porém não superior. Assim, o NBC pode verificar somente o Modelo do Mundo, o NPL pode acessar não só o Modelo do Mundo como também o Modelo Mental, e o NPC, por ser o nível mais sofisticado, tem acesso livre a todos os níveis de conhecimento.

3.5.6 Comparação entre as arquiteturas

Arquiteturas, de um modo geral, são especificações de como podemos construir agentes a partir de módulos e como estes módulos vão

interagir entre si. Isto é o que gerará as ações dos agentes e modificará o estado interno deles a partir das percepções.

Cada problema deve ser tratado de forma particular, pois cada um tem como requisitos um conjunto de características diferentes, necessitando que tenhamos, de certa forma, uma separação em tipos de arquiteturas conforme o modelo de agente que melhor se adequar ao problema. Como visto na §3.4 neste trabalho são consideradas três perspectivas: reativa, deliberativa ou híbrida.

A abordagem deliberativa compõe uma ampla variedade de conceitos relativos à construção de agentes autônomos. Todas as arquiteturas explanadas anteriormente encaixam-se nesta prerrogativa, visto que encontram a solução do problema da mesma forma: executando uma seqüência de ações que transformam o mundo em seu estado inicial para um estado final (objetivo). Elas buscaram, também, aprimorar seus sistemas de planejamento, superando as diversas limitações impostas pelo ambiente (que inicialmente tinha que ser projetado), incluindo aos agentes deliberativos a capacidade de atuar em ambientes reais. Esses agentes mais tarde receberam características baseadas nos estudos do comportamento humano desenvolvidos por filósofos e psicólogos sendo designadas de estados mentais – conceituando o que chamamos de arquitetura BDI (*Belief-Desire-Intention*).

A modelagem de IRMA foi pioneira em se tratando de arquiteturas BDI, mas com o intuito de modelar um agente considerando que este possui recursos limitados em termos computacionais, dando ênfase ao processo de filtragem das informações processadas pelo agente, e ao custo de execução das várias estratégias de deliberação dentro de um ambiente de constantes modificações. Não há nesta arquitetura uma distinção nítida de crenças, desejos e intenções, bem como não se percebe uma base lógica que apóie suas teorias e especificações.

Como IRMA, a arquitetura PRS foi desenvolvida para suportar simples agentes locais; apesar de se ter uma extensão formal para o contexto multi-agente, esta arquitetura não implementa cooperatividade entre os

agentes dentro do ambiente. Entre todos os sistemas descritos, PRS é um dos que possui certo fundo teórico em sua especificação.

COSY utilizou-se de elementos das arquiteturas IRMA e PRS, inclusive nomenclatura de módulos e componentes (e funcionalidades). Mas sobre esta implementação, estendeu seus conceitos básicos para também suportar protocolos de cooperação, que segundo [HAD96] habilitaram um alto nível de comunicação multi-agente e interações dentro desse ambiente.

GRATE foi a primeira arquitetura BDI implementada que trata de estados mentais e utiliza agentes sociais como alternativa para solução de problemas, apesar de se basear em muitos aspectos de IRMA.

Apesar da arquitetura PRS ser considerada para alguns autores como uma arquitetura híbrida segundo [DHE00], ou seja, aquela que consegue unir as vantagens da deliberação com as facilidades de resposta da reatividade, quem veio solidificar este conceito foi a arquitetura InteRRaP.

Em nenhuma destas arquiteturas apresentadas está a formação de intenções, baseadas nos desejos, satisfatoriamente descritas. Isto se deve, provavelmente, porque “desejos” têm diferenças semânticas de um autor para outro. Em IRMA e GRATE isto realmente não está bem descrito e não há distinção clara na implementação destes módulos. Em PRS isto acontece no meta-nível da Área do Conhecimento, sem qualquer especificação adicional. Similarmente, em COSY isto é de responsabilidade do módulo de intenções que é um conceito largamente dependente dos outros questionados (crenças, desejos).

4 PROGRAMAÇÃO DE JOGOS

Este capítulo visa introduzir um breve histórico do desenvolvimento dos jogos, como a inteligência artificial foi aproveitada neste contexto e alguns estudos de caso.

4.1 Histórico do desenvolvimento de jogos

4.1.1 O Começo

O desenvolvimento de jogos para computador começou e evoluiu através de um caminho relativamente diferente de outras aplicações da área da informática, como banco de dados e outros. De acordo com [ROL00], enquanto o desenvolvimento de aplicações comerciais e científicas começou com computadores de grande porte (os *mainframes*) e evoluiu para sistemas mais domésticos (os computadores pessoais), o desenvolvimento de jogos começou principalmente em sistemas de pequeno porte (como o *Commodore 64*) e evoluiu para os sistemas domésticos previamente citados. A evolução de ambos tornou possível a utilização de computadores tanto para entretenimento quanto para trabalho, como por exemplo, operar um aplicativo de planilha eletrônica e jogar um jogo de última geração, ambos aplicativos no mesmo computador.

Uma característica interessante do desenvolvimento de jogos em sua fase de ascensão (aproximadamente 1980) era a certeza de que o computador (ou seja, o hardware) onde o jogo seria executado era idêntico ao computador em que era desenvolvido. É interessante comparar este modelo com o desenvolvimento atual para videogames (onde o hardware é específico e conhecido). Isto traz alguns fatores desejáveis aos desenvolvedores, pois como a plataforma é imutável, os testes de compatibilidade de hardware eram nulos. Assim sendo, o jogo não corria o risco de executar lentamente, pois era

desenvolvido pelo mesmo tipo de computador (e não por um mais rápido), garantindo a precisão em testes de performance [ROL00].

Em relação à programação, inicialmente o *Assembly* era utilizado. O fato de que as plataformas possuíam recursos restritos de hardware, principalmente memória, tornava necessário o desenvolvimento de um código enxuto, que possuía (e ainda possui) a rapidez como requisito primário.

Como na época os compiladores C falhavam em gerar um código tão otimizado, a solução era mesmo o *Assembly*. Essa medida tornou os jogos menos portáteis, pois a linguagem de cada plataforma era diferente e os compiladores de linguagens de alto-nível (que além de não gerar um código aceitável) não possuíam formas de compilação para diferentes plataformas alvo.

Os ambientes de programação eram rústicos comparados com os de hoje. Eles eram apenas editores de *Assembly*, e os programadores necessitavam um bom conhecimento das instruções da plataforma, aritmética binária e transformações de base.

Os jogos desenvolvidos naquela época possuíam um “espírito” diferente. Como os recursos de hardware eram limitados, não era possível inovar muito graficamente. A jogabilidade, termo relacionado com a interação do usuário ao ambiente do jogo, deveria ser extremamente simples, pois interações mais complexas poderiam confundir o usuário. Todavia, esta simplicidade era a fórmula do sucesso de um jogo naquela época, e até hoje.

Entretanto, os jogos não eram necessariamente simplórios. Como a área de gráficos e sons estava apenas em estágio inicial, o que existia visualmente não era o suficiente para agradar por muito tempo o público. Portanto, os jogos dependiam da criatividade para possuírem alguma chance de sucesso.

4.1.2 Evolução de Hardware

Com o tempo, a evolução dos computadores começou a se refletir na indústria dos jogos. Com o aumento do processamento gráfico, os jogos melhoraram a sua aparência e as formas eram mais definidas. Com a melhoria no aparato de som, os jogos se tornaram mais agradáveis ao ouvido, e com o surgimento do mouse e *joysticks*, os jogos aproveitaram estes periféricos para obter uma interface melhor. Porém, a jogabilidade não acompanhou a onda de melhorias e este fator ganhou menos importância comparada com gráficos, som e interface.

4.1.3 Gráficos

A evolução gráfica foi espantosa. De enormes blocos coloridos formados por *pixels*, que construíam rústicos cenários 2D, passamos para ambientes modelados e exibidos completamente em 3D.

Inicialmente os programadores manejavam *pixels* para formar personagens, ambientes e arriscar animações (e.g. Pacman). O passo seguinte da evolução foi à manipulação de arquivos de imagens para representar cenários e personagens, que poderiam ser trabalhadas por desenhistas profissionais e carregadas para o jogo (graças ao aumento da memória, que permitiu o armazenamento de tais imagens). Deste modo surgiram os populares *sprites*, que são a denominação de um conjunto de imagens que, quando exibidas seqüencialmente, formam uma animação (e.g. Super Mario Bros).

Finalmente, com a evolução das placas gráficas (que atualmente possuem módulos de memória próprios, para carregar as imagens) foi possível obter a rapidez necessária para desenvolver ambientes completamente 3D, com texturas e iluminação, adicionando uma dose de realismo aos jogos. Com isso, os atuais jogos topo de linha são jogos em 3D, normalmente com uma perspectiva de primeira pessoa onde o jogador pode navegar por um imenso ambiente iterativo (e.g. Unreal Tournament).

4.1.4 Som

A evolução na parte sonora dos jogos levou um certo tempo, se comparada com a gráfica. Os sons começaram apenas com *bips* dos alto falantes internos dos computadores, e as variações de sons eram obtidas através de modulação de frequências.

O desenvolvimento de um hardware específico (que era conectado em caixas de som) adicionou mais tons e eram, sem dúvida, mais agradáveis do que os predecessores. Com o tempo, vários formatos de sons, inclusive *mp3*, começaram a ser suportados pelas placas de som. Hoje em dia, a moda nesta área é o som ambiental em 3D. Com uma aparelhagem relativamente cara é possível utilizar esta nova tecnologia e obter um efeito de imersão maior no jogo.

4.1.5 Interface

A evolução da interface também foi mais lenta do que a gráfica. No início, a interface era feita através de *joysticks* básicos (com um, no máximo dois, botões) no caso de videogames, e teclado no caso dos primeiros computadores. A evolução dos jogos forçou a adição de mais botões aos *joysticks*, e a migração destes para os computadores. Com o advento do mouse, este periférico foi aproveitado como um método fácil de interface em jogos, e continua extremamente popular.

Os últimos avanços nesta área são os *joysticks Force Feedback*, que possuem um mecanismo vibratório que é ativado em resposta ao ambiente do jogo, e em mouses mais sensíveis, específicos para jogadores. Também é comum encontrar volantes e pedais como periféricos de interfaces, designados especialmente para jogos de corrida (e.g. Nascar).

4.1.6 Jogos Multi-Jogador

Uma área relativamente nova é a de jogos multi-jogador em rede. O jogo multi-jogador é suportado desde o videogame através, por exemplo, do

uso de dois *joysticks*. Porém, jogos deste tipo em rede de multi-computadores são relativamente novos, pois seguem a evolução da telecomunicação e dos protocolos de rede. Um fato interessante é que os primeiros jogos (aproximadamente em 1970) operavam em ambientes de rede e rodavam em *mainframes*. Esta área teve início desde quando foi possível conectar dois computadores através de um cabo serial. Desde então, a tecnologia evoluiu para suportar protocolos como IPX, TCP/IP e alguns outros. Outro fator de influência foi a Internet, o desenvolvimento de modems e, mais recentemente, acesso de banda larga.

O suporte a multi-jogador tornou-se um requerimento primário para todos os jogos de última geração, pois este fator proporciona que o jogo seja aproveitado por mais tempo, com a possibilidade de enfrentar outros jogadores, ou cooperar com estes. O fato da necessidade de multi-jogador é tão verdade que existem alguns jogos, além de vários a serem lançados, que possuem o objetivo de serem jogados exclusivamente na rede (e.g. Asheron's Call), onde existem diversas comunidades voltadas a estes fins.

4.1.7 Evolução de Software

Um benefício proporcionado pelas evoluções de hardware foi a evolução na área de software. Com o aumento da capacidade de processamento e da quantidade de memória, foi possível desenvolver Sistemas Operacionais com mais recursos e a criação de compiladores mais otimizados, assim possibilitando o uso de linguagens de alto-nível.

Portanto, como os compiladores atuais geram código muito mais otimizado, hoje é quase desnecessário o uso do *Assembly* para a geração de um código enxuto. A maioria dos jogos atuais são feitos utilizando-se a linguagem C/C++. Os ambientes de programação também melhoraram. Agora eles suportam uma vasta série de opções para auxiliar o programador em sua tarefa.

O sistema operacional MS-DOS foi a primeira plataforma de jogos para o PC. Porém, o desenvolvimento para este sistema possuía seus

contratempos. O relacionamento entre o MS-DOS e os componentes gráficos e de som não era ideal. Considerando o fato de que as imagens necessitavam do acesso direto à placa de vídeo, era necessário conhecer os padrões das mesmas, e isto se mantinha verdade em relação às placas de som [ROL00]. Portanto, os programadores precisavam escrever os *drivers* para utilizar tais periféricos.

Também de acordo com [ROL00], com o lançamento do Windows 3.1, os programadores de jogos cogitaram a possibilidade de mudar a plataforma de desenvolvimento, porém esta mostrou uma perda de performance acentuada, se comparada ao MS-DOS, portanto a idéia foi abandonada. O Windows 95 passou pelo mesmo efeito.

No entanto, em 1995, a Microsoft lançou uma biblioteca gráfica e de áudio chamada *DirectX*. Inicialmente ela não correspondia à velocidade do MS-DOS, mas após a terceira versão ela passou a trazer melhores resultados. Hoje, esta biblioteca está na sua oitava versão e possui componentes gráficos (2D e 3D), de áudio, interface, rede e outros. Ela apresenta ao desenvolvedor de jogos uma camada que abstrai a especificação de hardware, possibilitando que este não precise se preocupar em fornecer suporte a determinados periféricos, pois esta função deve ser desenvolvida pelos fabricantes do hardware através de *drivers* para a biblioteca. O *DirectX* tornou-se o novo padrão de desenvolvimento de jogos.

4.1.8 Evolução dos Profissionais

Outro fator marcante no histórico do desenvolvimento de jogos é como as funções desta indústria mudaram durante o tempo e como elas responderam as evoluções. A Tabela 1 mostra uma relação entre o ano, número de desenvolvedores e estágio de evolução dos jogos.

Ano	Pessoas	Produtos
1971	1	Jogos texto em <i>mainframes</i>
1981	1,3	Cartuchos de videogame de 4k

1985	1,5 – 3	Jogos em disco para <i>Apple II</i> e <i>Commodore 64</i>
1989	4	Jogos em disco para o PC
1993	8	Primeiro jogo em CD para o PC
1995-2001	25	Grandes jogos em CD

Tabela 1: Evolução dos jogos

Nota-se um grande aumento de pessoal desde 1971. Mas, o que mudou?

Nos primórdios do desenvolvimento de jogos, os programadores eram a alma do negócio. Eram eles que desenvolviam a parte gráfica, o som, a interface com o usuário e até a história do jogo, a qual não necessitava tanta ênfase na época.

Com mais recursos de hardware disponíveis, foi possível aprimorar a parte artística, pois desenhistas capacitados poderiam produzir as imagens do jogo, músicos poderiam compor uma trilha sonora, escritores poderiam elaborar um roteiro para o jogo, e os programadores poderiam se ater às partes realmente importantes, que constituem o desenvolvimento do jogo em si.

Surge também um novo perfil de funcionário, o designer. O perfil deste cargo não é necessariamente o de um programador, mas é preciso ter noções técnicas. Também não é de um artista ou escritor, mas é preciso ser criativo. O designer é encarregado de “criar” o jogo, ele basicamente modela as fases e cria o ambiente onde o jogador irá interagir. Para isso, ele utiliza um editor de fases (comum na maioria dos jogos), seguindo o roteiro definido por um escritor e utilizando a arte, tanto 2D como 3D, desenvolvida por um artista. Com os jogos em 3D, existem equipes de desenvolvimento que têm empregado inclusive arquitetos.

No estágio de evolução atual, a criação de um jogo passa a ser um processo incrivelmente elaborado, que mistura a parte técnica com a parte criativa e artística. É virtualmente impossível criar um jogo (com tecnologia de ponta) com um time de menos de quatro pessoas. A evolução das tecnologias,

tanto de hardware como de ferramentas para a elaboração do jogo, requer profissionais especializados nas diversas áreas que o compõem, como a de gráficos, som e, sem dúvida, a de inteligência artificial.

4.1.9 Inteligência Artificial nos Jogos

Segundo [HOW01] um dos primeiros e principais jogos a utilizar noções de inteligência artificial foi o Pacman. Neste, o jogador controla um avatar, ou seja, o representante do jogador dentro do mundo virtual do jogo, que precisava fugir de quatro “fantasmas” por um labirinto. O fato que tornava o jogo mais interessante é que estes fantasmas não possuíam todos o mesmo comportamento, cada um tinha um plano de como perseguir e atacar o jogador. Este é um exemplo de uso de algoritmos de busca e comportamento em um jogo.

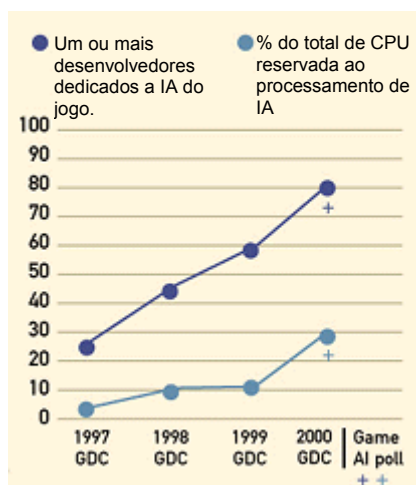


Figura 9 Importância da IA nos jogos.

Desde o Pacman até os dias de hoje, a inteligência artificial tem evoluído e cada vez tem ganhado mais força e importância no desenvolvimento dos jogos. Atualmente, algumas equipes de desenvolvimento têm empregado programadores especializados em IA desde o começo do processo de desenvolvimento. Segundo [WOO00] os desenvolvedores têm reservado cada vez mais tempo de processamento para as rotinas de IA (vide Figura 9), e isto mostra a mudança de foco das empresas e uma nova evolução na indústria de jogos. Estas mudanças favoráveis em relação à IA têm se dado principalmente

devido à interação dos usuários com as empresas, requisitando que os oponentes possuam comportamentos realistas e reações inesperadas.

Com isso, várias técnicas de IA têm sido empregadas no contexto dos jogos. Uma das mais fundamentais é a de algoritmos para percorrer ambientes. É de vital importância que os elementos móveis de um jogo consigam navegar pelo seu terreno de uma forma adequada. O mais popular algoritmo desta classe, o A*, possui diversas implementações e otimizações nestes jogos.

Outra noção que tem se tornado cada vez mais popular é a da *A-Life*, que modela o comportamento de uma vida. Um jogo que ficou famoso por utilizar esta técnica é o *The Sims*, que simula a vida de um ser humano, ou uma família, através de uma interação social simulada entre personagens virtuais, assim então desenvolvendo uma personalidade.

A técnica de agentes é normalmente explorada com algumas restrições. Poucos jogos têm sido desenvolvidos utilizando abertamente esta técnica.

4.2 O Estado da Arte da IA na indústria

4.2.1 Unreal Tournament

Neste produto, o jogador batalha contra outros jogadores ou com oponentes controlados pelo computador (denominados *bots*). Estes *bots* possuem uma boa autonomia, pois eles sabem como percorrer os cenários, coletando armamentos e munições, e atacar os jogadores com eficiência (ou, dependendo do nível de dificuldade, ineficiência simulada). Estes *bots* são completamente configuráveis, ou seja, é possível determinar pesos em seus atributos e necessidades, tais como perícia com a arma, movimentação e medo.

Como o computador controla estes *bots*, é razoável assumir que quanto mais destes existem no ambiente mais lento e prejudicado seria o processamento mental deste. Por isso, foi realizado um projeto de distribuição

de *bots* em máquinas diferentes, onde cada máquina se dedicaria a controlar um *bot* e enviar seus movimentos para um servidor, assim obtendo uma melhor performance.

4.2.2 Baldur's Gate

Este é um jogo de RPG (*Role Playing Game*) onde o jogador assume o papel de um herói em um mundo medieval fantasioso (*Dungeons & Dragons*). O jogo possui uma linguagem de script, relativamente poderosa, para definir comportamentos dos personagens. É possível criar várias reações realistas como, por exemplo, quando uma personagem está prestes a morrer ele pode começar a correr de seus inimigos em direção a algum companheiro que possa prover proteção.

4.2.3 The Sims

É um jogo de simulação de pessoas. O jogador toma conta de uma ou mais pessoas (os *Sims*), moradoras de uma casa, e pode influenciar estes *Sims* a fazerem ações comuns como almoçar, assistir televisão e limpar o banheiro. Porém, os *Sims* possuem desejos próprios (definidos por suas personalidades) e se o jogador decidir não controlar estes, eles irão agir por conta própria, simulando de uma maneira interessante o comportamento humano. Este jogo, como descrito anteriormente, utiliza técnicas de *A-Life* para o comportamento dos *Sims*.

4.2.4 Black and White

Neste jogo, o jogador assume o papel de Deus em um mundo paralelo. Em determinado momento, o jogador obtém uma criatura, que será o seu avatar no mundo. Ele deverá alimentar a criatura, para que ela cresça, e ensiná-la magias, para que ela ganhe autonomia. A criatura observa o jogador em suas ações e tenta imitá-las, assim moldando a sua personalidade. Ao obter uma certa autonomia, é possível notar comportamentos imprevisíveis e surpreendentes, validando o aprendizado da criatura.

4.3 Jogos Exemplo

Na fase de implementação do trabalho de conclusão, alguns jogos que utilizam a arquitetura de agentes do ICE foram implementados. Estes jogos foram implementados acoplando-se as classes do *kernel* ICE junto com agentes gerados através da linguagem IADL e estão descritos em linhas gerais abaixo:

4.3.1 Tcheco Balls

Este jogo tem como propósito demonstrar as capacidades reativas do agente. O jogo é constituído de um tabuleiro, com dimensões variadas, que possui algumas casas coloridas e unidades coloridas, que podem se movimentar e são controladas pelos agentes.

O objetivo do jogo é que as unidades coloridas achem suas casas de respectiva cor. Para isso elas devem percorrer o tabuleiro (vide Figura 10) até achar alguma casa colorida. Ao achar uma casa com cor, as unidades “dormem” (ou seja, elas não fazem mais movimentos) não importando a cor da casa. Porém, se uma unidade estiver dormindo em uma casa que não a pertence, ela pode ser desalojada pela unidade com cor igual a da determinada casa (Ex. uma unidade verde está dormindo em uma casa amarela. Se uma unidade amarela tentar entrar em sua casa, ela pode expulsar a unidade verde). Uma unidade que dorme em uma casa com a cor respectiva não pode ser desalojada.

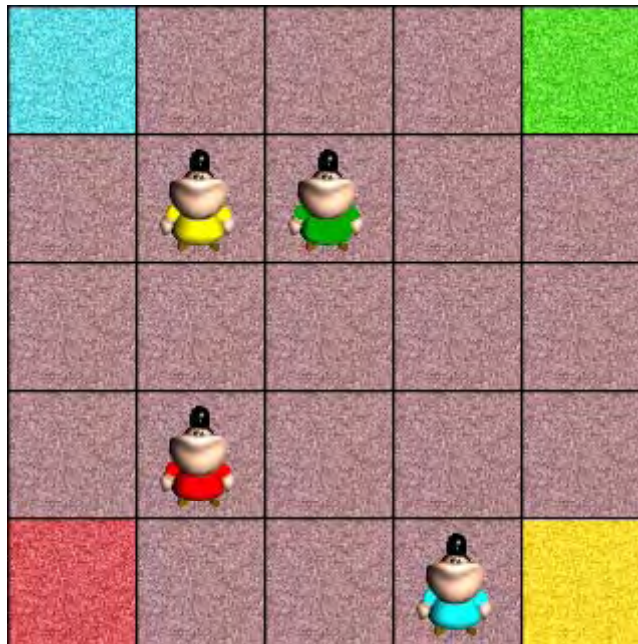


Figura 10 Tela do protótipo do jogo Tcheco Balls

O jogo termina quando todas as casas coloridas estiverem preenchidas com unidades das cores respectivas.

4.3.2 FormiguinhaS

FormiguinhaS é um jogo em que o objetivo principal é obter a maior quantidade possível de comida em um ambiente onde o jogador compete com um agente Formiga que tem o mesmo objetivo – procurar comida. Acontece que este agente ao perceber a presença do inimigo foge dele – neste caso o jogador poderá aproximar-se do agente para afastá-lo das comidas ainda existentes. Ambos terão de percorrer um labirinto onde aleatoriamente encontram-se espalhadas as comidas.

O jogador e o agente iniciarão o jogo em posições aleatórias também, o que demonstrará com mais eficácia a atuação do agente neste ambiente. Ambos são representados por um avatar que os diferencia dentro do labirinto. O agente é representado por uma formiga vermelha, e o jogador, por uma formiga marrom, visível na Figura 11.



Figura 11 Tela do protótipo do jogo FormiguinhaS

O labirinto é formado por folhas verdes espalhadas pela tela do jogo, as quais nem o jogador nem o agente podem ultrapassar, tendo então que contorná-las.

O jogo terminará quando não mais houverem comidas no labirinto, mostrando ao final o vencedor (quem obteve o maior número de comidas capturadas).

5 A ARQUITETURA ICE

Neste capítulo serão descritas as funcionalidades propostas da arquitetura ICE bem como seus componentes principais.

5.1 Funcionalidades Propostas

5.1.1 Definição dos Agentes no Jogo

A funcionalidade principal a ser oferecida pelo ambiente ICE será a definição dos agentes imersos no jogo. Esta definição será feita através de uma linguagem orientada a agentes, onde o usuário poderá definir as entidades utilizando dois níveis de abstração. No primeiro e mais baixo nível, o usuário irá especificar reações a situações simples para serem realizadas assim que percepções específicas forem detectadas pelo agente. No segundo nível de abstração, o usuário irá especificar os objetivos da entidade, que então irá se responsabilizar por planejar e escolher as ações de baixo nível necessárias para o cumprimento destes objetivos. No protótipo implementado, o planejamento consiste apenas em selecionar planos pré-definidos.

É interessante notar que a diferenciação nos níveis de abstração pode ser encarada da mesma forma que a diferenciação entre seres dotados de capacidades cognitivas mais abstratas (i.e. seres humanos, animais inteligentes) e seres dotados de capacidades de ação e reação (i.e. objetos inanimados, insetos).

Esta diferenciação também é importante, pois nem todas as entidades do jogo necessitam apresentar comportamentos complexos. Por exemplo, uma formiga que caminha aleatoriamente em busca de alimento. No momento em que encontrar um obstáculo ela apenas desvia deste sem qualquer reflexo mais importante em seu comportamento.

Entidades mais complexas podem ser dotadas de objetivos que irão nortear as ações tomadas pró ativamente. Por exemplo, imaginemos um

soldado, que chamaremos de Hans. Hans tem como objetivo encontrar o jogador e eliminá-lo. Para este fim, ele deverá criar um plano de ação de como encontrar o jogador de modo a poder atacá-lo. Este plano pode incluir uma busca aleatória pelo mundo, que deverá levar em conta o ambiente e os obstáculos (e.g. paredes e buracos) nele encontrados. Hans também poderá encontrar Friedrich, que tem como objetivo impedir Hans de encontrar seu objetivo. Logo Hans deverá tentar evitá-lo. Após encontrar o jogador, Hans deverá planejar como eliminá-lo. Este plano pode incluir cooperação com outros soldados facilitando esta tarefa, pois o jogador possivelmente é mais hábil do que Hans.

5.1.2 Camada de Visualização

É importante lembrar que no contexto de um jogo de computador, é necessária a percepção pelo jogador do que ocorre dentro do mundo virtual do mesmo. Para tanto o jogo possui o componente GE (*Game Engine*), que pode compreender diversos *engines* mais específicos relacionados com a percepção do jogador tais como: gráficos, som, percepção de tato, impulsos neurais, etc.

Assim sendo, uma das *engines* do GE pode ser responsável pela parte da interface com o usuário. Ela captará os estímulos do jogador, através de periféricos como o teclado e mouse, e exibirá a este o resultado no jogo, através do vídeo.

Porém, estes estímulos devem ser transportados para a camada dos agentes, na arquitetura ICE, tanto quanto as respostas dos agentes devem ser transportadas para o GE. Desta forma, foi confeccionada uma camada de visualização dos agentes, que tem como objetivo primário comunicar o GE com os agentes. Esta camada reside basicamente no componente chamado *Ether*, que será explicado a seguir.

5.1.3 Ambiente de comunicação multi-agentes

Uma vez que os agentes estão definidos, estes não podem permanecer isolados no ambiente. Logo deve existir um ambiente onde os

agentes possam agir, e de onde os agentes recebem percepções para alimentar suas crenças, ou seja, onde aconteça a interação entre os agentes e dos mesmos com o mundo.

Este ambiente também irá implementar as regras do “universo” do jogo. Estas podem ser vistas, por exemplo, como as leis da natureza tais quais a gravidade e a resistência do ar. Elas devem ser definidas pelo usuário do ambiente ICE, proporcionando máxima flexibilidade na definição do mundo.

5.2 Descrição da arquitetura ICE

5.2.1 Visão geral

O ambiente ICE será essencialmente um sistema multi-agentes. De acordo com as definições mais genéricas, agentes percebem o mundo através de sensores e agem sobre ele através de atuadores. Desta forma a principal, e em teoria única interface necessária entre os agentes são ações geradas pelos atuadores e percepções recebidas pelos sensores.

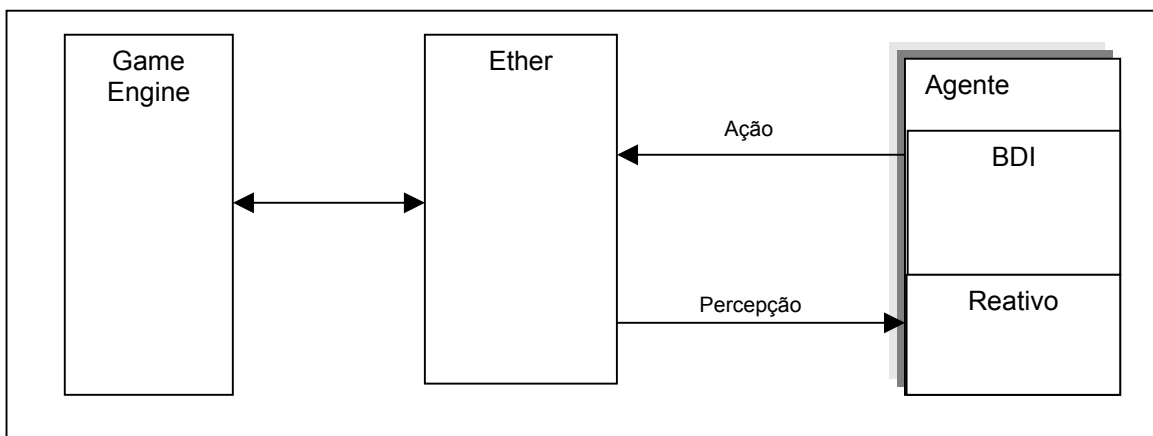


Figura 12 Descrição em alto nível da arquitetura do ICE

Sob uma análise simplificada esta interface pode ser vista como um sistema de troca de mensagens, cuja implementação se assemelha à maioria dos simuladores de sistemas. Assim sendo, a arquitetura ICE será composta de um componente responsável pela troca deste tipo de mensagens. Este componente será chamado de *Ether* em referência a substância fantasiosa que permeia e dá consistência ao universo. Este componente será utilizado como

ambiente por todos os agentes do jogo, de acordo com a Figura 12, e dada a sua importância, será detalhado na §5.2.3.

O mapeamento das ações em percepções realizado pelo *Ether* será definido pelo usuário do ICE na criação do ambiente do jogo. Em resumo este será o *kernel* do ICE, pois será este componente o por onde toda a comunicação entre os agentes irá ocorrer.

Imersos no *Ether* estão os agentes que representam as entidades do jogo. Estes agentes são compostos essencialmente de duas partes, um componente de mais baixo nível, que deverá corresponder ao modelo de agente reativo, e um componente de alto nível que deverá corresponder ao modelo de agente BDI.

Estas entidades do jogo existentes apenas na sua representação lógica até o momento deverão ser representadas de alguma forma para o jogador. Apesar de a cognição e interação dos agentes serem os focos principais do ambiente ICE, uma interface com o GE será criada. Esta interação é necessária para que possa existir uma representação “visível” (este adjetivo aqui empregado transcende a representação visual dos componentes) ao usuário do estado das entidades do jogo, pré-requisito fundamental para a interação do usuário com o ambiente interno do ICE. De modo a permitir uma comunicação da parte lógica do jogo (i.e. o ambiente ICE) com a parte de visualização (aqui englobando toda a percepção do jogador, não só o visual), o *Ether* deverá prover uma interface que permita ao *Game Engine* extrair as informações relevantes à geração da visualização.

Este componente será apresentado com mais detalhes nas seções subsequentes, também apresentando a implementação realizada no TC2.

5.2.2 Os Agentes do ICE

A arquitetura de agência proposta como modelo para criação do ICE irá consistir de um modelo híbrido, contendo componentes baseados em Agentes Reativos, daqui por diante chamados de AR, e componentes baseados em Agentes BDI (*Belief-Desire-Intention*), doravante chamados

ABDI, como descrito na Figura 13. A representação de cada entidade no mundo de jogo descrito para utilização com o ICE irá consistir de um componente AR, de mais baixo nível, e opcionalmente por um componente ABDI.

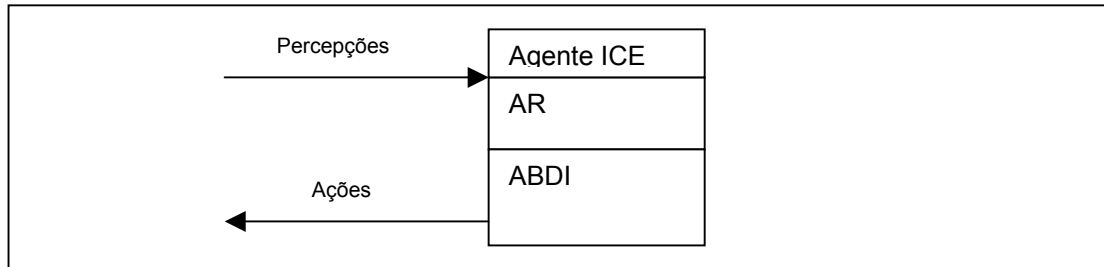


Figura 13 Visão geral dos Agentes ICE

O motivo desta composição das entidades do jogo se torna mais aparente após a consideração das características de cada componente. O componente AR, dada a sua característica puramente reativa, torna possível uma implementação bastante rápida, já que seu comportamento será definido de forma semelhante a cláusulas condicionais (se, então). E será utilizado para a descrição de efeitos cujo comportamento pode ser facilmente descrito a priori pelo desenvolvedor. Este comportamento em geral é apresentado por entidades inanimadas, por entidades que não apresentam comportamento deliberativo, ou até mesmo para descrever reações involuntárias. Por exemplo, depois de perceber 20 unidades de dano, uma cadeira irá se quebrar, ou também um zumbi que anda da direita para a esquerda até encontrar um ser vivo, para então atacá-lo.

O componente ABDI, baseado na arquitetura BDI descrita em [WOO99], terá como objetivo a descrição de comportamentos deliberativos. Este componente poderá apresentar capacidade de planejamento, e será descrito em termos de seus objetivos, desta forma possibilitando um alto grau de abstração por parte do desenvolvedor. Evidentemente, o componente ABDI apresentará uma implementação mais custosa em termos de tempo de processamento, porém como este componente é opcional, poderá ser reservado a entidades de maior importância dentro do jogo como, por exemplo, o soldado Hans. Isto irá permitir o dimensionamento do gasto computacional, e

ao mesmo tempo permitir o balanceamento das capacidades cognitivas dos componentes do jogo.

Internamente, os agentes poderão ser dotados de uma estrutura que permita com que o hibridismo da arquitetura não cause conflitos. Os fluxos de comunicação desta estrutura podem ser vistos na Figura 14.

Assim que o agente receber uma percepção do ambiente, esta será entregue ao componente AR, que irá verificar nas suas regras se há alguma reação definida para a mesma. Quando o agente executar uma ação sobre o ambiente, esta será entregue a função de revisão de crenças.

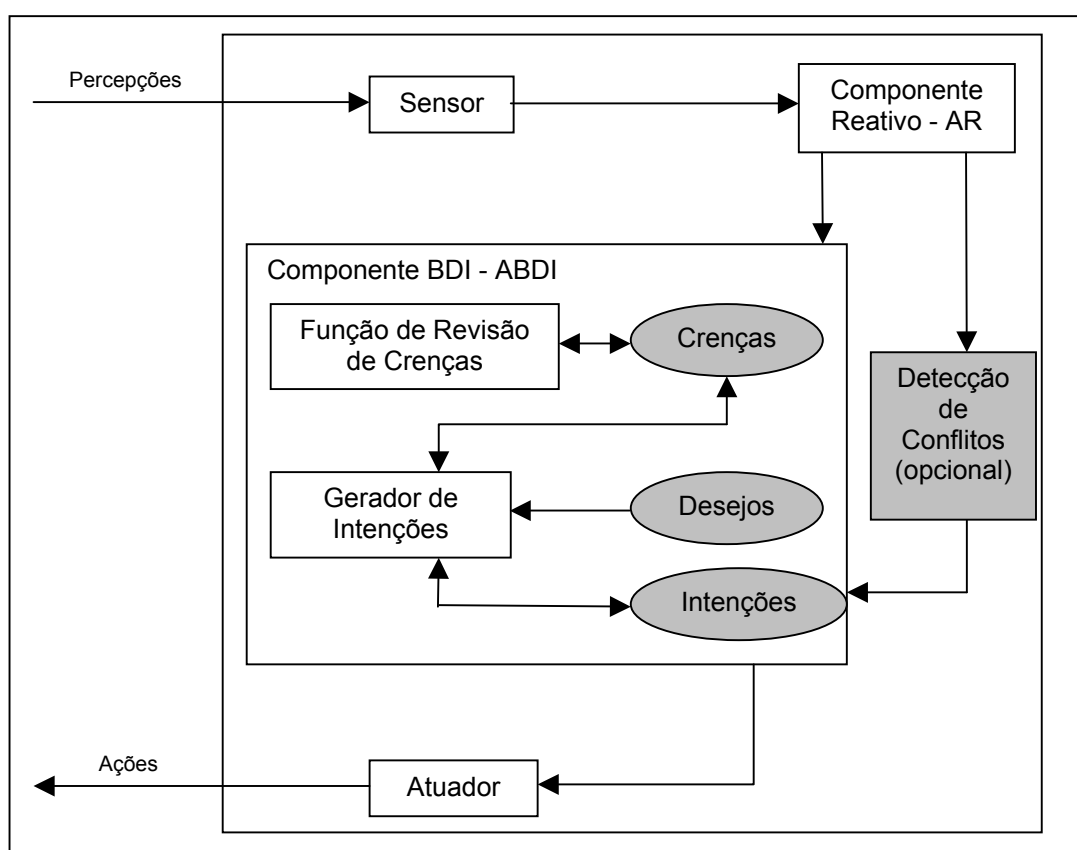


Figura 14 Estrutura interna dos Agentes ICE

O componente ABDI irá levar em consideração a base de crenças atualizadas para decidir se o objetivo atual continua viável ou já foi cumprido. A seguir caso haja necessidade de reconsideração o gerador de intenções irá, baseado nos objetivos definidos pelo usuário, representados pelo repositório de desejos, analisar as crenças e selecionar os desejos que são viáveis para serem perseguidos, ou seja, dados os objetivos de um agente, o gerador de

intenções irá verificar suas crenças e selecionar os desejos viáveis. Tendo os desejos selecionados, o gerador de intenções irá organizar os desejos com base em alguma função de prioridade definida pelo usuário e iniciar o planejamento para a concretização dos mesmos. Nos agentes ICE as intenções equivalem aos passos do plano, ou seja, ações concretas.

Como as intenções equivalem aos passos do plano, elas podem ser representadas como uma fila de ações para execução. Desta forma quando o componente reativo AR da arquitetura gerar uma ação, esta será colocada na frente da fila de intenções, dotando este componente de uma maior prioridade em relação ao componente ABDI, esta prioridade é apropriada para a semântica esperada para o AR, ou seja, a capacidade de reações rápidas a mudanças no ambiente. Por exemplo, o soldado Hans acaba de ter planejado e decidido as ações necessárias para a defesa de sua base, e é atingido por um inimigo. Hans deve reagir diminuindo seu atributo de energia, porém é necessário que esta modificação seja efetivada imediatamente e não somente após a realização de todo o plano de defesa contido nas intenções.

Esta arquitetura também permite que o agente realize seu planejamento em paralelo com a execução das ações previamente planejadas. Isto é possível, pois o Gerador de Intenções, que realiza o planejamento pode inserir seus resultados na fila de execução enquanto o agente extrai e executa as ações na frente da fila. O componente AR também possibilita que se definam reações para situações críticas, onde a pausa para re-consideração da situação e re-planejamento seria inviável do ponto de vista de tempo de espera.

Um aspecto que deve ser levado em consideração devido a esta dualidade na origem das ações do agente é o fato de ser possível a ocorrência de conflitos entre os passos de um determinado plano e uma reação. Para resolver este problema, algumas arquiteturas híbridas são dotadas de um detector de conflitos. O projeto da arquitetura ICE permite que tal componente seja utilizado, porém a implementação realizada no TC2 não irá prover um componente detector de conflitos, uma vez que sua devida implementação

seria inviável do ponto de vista de tempo disponível. Desta forma fica a cargo do usuário a previsão e eliminação de possíveis conflitos causados por conjugação de planos e reações.

Uma das características fundamentais dos agentes e que deverá ser preservada em sua implementação no ICE é a independência dos mesmos em relação ao ambiente, uma vez que não haverá maneira de se determinar o comportamento do agente através de sua interface. De um modo geral, os agentes ICE poderão apresentar ou não objetivos, sendo que o processo de planejamento e cognição deverá ser exclusivo das entidades que os apresentarem. É importante lembrar que a escolha de objetivos a serem cumpridos, a seleção de comprometerimentos, e o planejamento de sua execução é um processo bastante custoso e não deverá ser executada pela maioria dos agentes imersos no mundo.

5.2.3 Ether: o ambiente dos Agentes

Como indicado na Figura 12, e mencionado anteriormente, a arquitetura do ICE divide claramente o funcionamento do jogo do funcionamento dos agentes. Esta divisão foi concebida a fim de prover um nível de abstração maior ao desenvolvedor do jogo (ou seja, o desenvolvedor do *Game Engine*), pois este não precisa, ou deve, ter o conhecimento imediato de como manipular os agentes. Por isso o *Ether* se torna uma camada tão importante na arquitetura.

O *Ether*, basicamente o cerne do sistema multi-agentes, corresponde ao sistema de troca de mensagens entre os agentes da arquitetura ICE. Uma vez que a única interface conceitual dos agentes são as ações dos atuadores e as percepções recebidas pelos sensores, o *Ether* é responsável por receber como entrada as ações de todos os agentes do sistema, processá-las e distribuir percepções de volta aos agentes. Em termos de ambiente de agentes, o *Ether* pode ser classificado, de acordo com §3.3.2, como inacessível, não-determinístico, episódico, dinâmico e discreto.

Além da funcionalidade de “roteador” das percepções dos agentes, o *Ether* funciona como a representação do ambiente. Neste estará mapeada as constantes do mundo que se deseja representar.

O *Ether* implementa a troca de duas mensagens com os agentes ICE, que são *ação* e *percepção*. A mensagem de *ação*, que é disparada pelo agente com destino ao *Ether*, visa informar o sistema multi-agente da função que o agente pretende realizar. Ao receber esta mensagem, o *Ether* deve descobrir quais agentes podem ser afetados pela ação e, para estes, enviar a mensagem *percepção* indicando o que aconteceu no ambiente.

O *Ether* terá seu funcionamento ligado diretamente com o da *Game Engine*, ou seja, ele terá seu tempo de execução limitado pela GE. Como os jogos possuem um *loop* central, localizado na GE, onde o processamento crítico ocorre (como exibição das imagens, processamento de periféricos e lógica do jogo), será designada uma fatia de tempo para que o *Ether* processe. Com isso, será possível escalonar facilmente o tempo de processamento lógico.

Da mesma forma que a GE, o *Ether* pode alterar o tempo de processamento dos agentes ICE, designando a eles fatias de tempo para cumprir suas tarefas. Com isso, o processamento de ações por parte dos agentes pode ser balanceado de forma que elementos que estiverem fora da linha de visão do jogador, assim considerados “fora da cena”, ocupem menos processamento do que elementos próximos ao jogador.

5.3 Revisões no projeto original

A modelagem física e a melhor consideração da arquitetura realizada no início do TC2 levaram a algumas mudanças na arquitetura proposta inicialmente. Estas alterações serão descritas a seguir.

Após a reconsideração da arquitetura causada pelo estágio de modelagem física, chegou-se a algumas conclusões a respeito da arquitetura proposta inicialmente.

A princípio a arquitetura ICE previa um componente para a representação dos agentes, que se basearia no estado interno do agente para a geração de estímulos perceptíveis para o usuário, a *Persona*. Como o componente *Ether*, definido na §5.2.3, teria a possibilidade de omitir dados corretos para simular algum tipo de empecilho no sensoriamento, o *Ether* então deve guardar consigo os dados reais para que possa informar o agente da realidade no momento em que tal empecilho não mais existisse. Desta forma, insere-se no agente a possibilidade de não haver os dados mais fidedignos sobre a realidade no mesmo. Este fato invalidaria uma representação do agente baseada no estado interno do mesmo, forçando o componente *Persona* a ter de buscar este estado junto ao *Ether*.

Além disto, observou-se que diversos dados deveriam ser enviados para o *Game Engine*, e alguns destes dados não seriam necessariamente úteis para a simulação do comportamento do agente. Um exemplo típico seria no caso de um que agente disparasse com uma arma contra outro agente, e se no jogo em questão existisse uma simulação de trajetória da bala. Neste exemplo, a posição exata da arma do atacante seria necessária para computar a trajetória descrita pelo tiro no componente que calcula a física do jogo, usualmente chamado de *Physics Engine*. Este exemplo mostra um dado que seria de pouca utilidade para o ICE, a não ser que se deseje criar uma definição de comportamento desproporcionalmente complexa. Assim, caso a totalidade de variáveis sobre uma entidade estivesse contida no ICE, criar-se-ia a necessidade de um *overhead* enorme, e possivelmente desnecessário de comunicação entre ICE e *Game Engine*. Para sanar este problema optou-se por adotar um modelo de representação onde apenas as informações relevantes para a representação do comportamento de uma agente sejam armazenadas no Agente ICE. Estes dois fatores motivaram a remoção do componente *Persona* da arquitetura ICE.

6 O PROJETO ICE

6.1 Fundamentos do projeto

6.1.1 A Separação entre Lógica e Dados

Jogos, e programas em geral, são compostos por duas partes: lógica e dados. Existe uma diferença marcante entre as duas: a lógica define as regras básicas e os algoritmos do *game engine*; enquanto os dados dão os detalhes do conteúdo e do comportamento do jogo. Sozinhos eles são inúteis, porém quando juntos estas partes fazem um jogo evoluir rapidamente. É necessário desviar-se do assunto principal por um momento para se esclarecer o que é *game engine*; a palavra *engine*, além de seu significado clássico na língua inglesa também é utilizada no contexto de software, especialmente na indústria de jogos para definir os componentes principais da API mais básica de um programa, isto é, aquela que dá suporte aos componentes de mais alto nível. Fazendo um paralelo com sistemas operacionais, o *game engine* corresponderia ao *kernel* do jogo.

De acordo com [RAB00a], a motivação básica da separação entre a lógica e os dados é que o jogo evolui dramaticamente desde a concepção até o produto acabado, uma vez que o desenvolvimento de bons jogos (não em um sentido técnico) é muito mais uma arte do que uma ciência. O processo de desenvolvimento de jogos envolve muita experimentação em relação a valores de diversos componentes do programa, desde regras do próprio jogo até posições de câmera em um ambiente. Considerando o fato de que a equipe de desenvolvimento do jogo não é composta apenas de programadores, e que em diversas situações quem irá realizar a experimentação e sugerir as modificações não faz parte do pessoal técnico (apesar de ter algumas noções), um alto nível de parametrização do jogo permite que os designers do jogo possam tentar diversas combinações de parâmetros sem a necessidade do auxílio de um programador, como descrito na Figura 15.

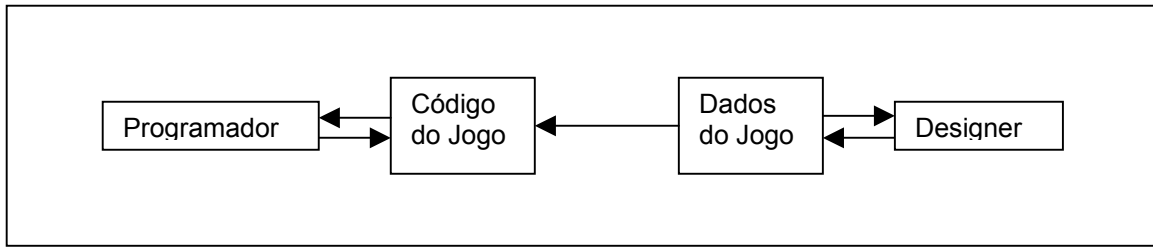


Figura 15 Modelo ideal de interação entre os desenvolvedores e o jogo.

Uma prerrogativa básica neste sentido é evitar o *hard-coding*, ou seja, inserir de forma estática no código fonte parâmetros de comportamento do jogo, deixando todas as constantes do jogo carregáveis através de arquivos-texto externos. Evidentemente, por questões de performance, estes arquivos poderão se tornar arquivos binários na versão final do jogo. Desta forma pode-se experimentar diversos valores sem a necessidade de re-compilação, usando-se apenas o Bloco de Notas, ou algum tipo de editor especificamente projetado para tanto. Desta forma, projeta-se um programa mais flexível, e como muitas vezes se observa, programar uma solução genérica faz com que se pense melhor na solução do problema.

Por exemplo, se a especificação inicial do jogo pede a existência de três tipos de inimigos, o programador pode facilmente (e talvez até mais rapidamente) programar três inimigos diretamente no código com as características descritas na especificação. Porém, se for feito um programa que carrega os inimigos de um arquivo externo, é possível fazer experiências com diversas características dos mesmos e até mesmo colocar uma variedade maior de entidades antagônicas.

6.1.2 Modelando comportamento fora do código

Na medida em que os jogos foram se tornando mais complexos e começaram a envolver mais pessoas de diferentes áreas de conhecimento, a separação da lógica e dos dados tornou-se um dos objetivos do desenvolvimento de um jogo. Porém, este processo é longe de ser trivial e existem diversas questões que se apresentam ao se desenvolver jogos desta forma.

Além de se colocar as constantes fora do código, é importante que se descreva comportamentos em scripts externos [HUE97], pois estes são excelentes para se descrever passos seqüenciais como, por exemplo, descrever o que acontece quando o jogador termina uma determinada tarefa do jogo. Como o objetivo destes scripts é a sua manipulação por pessoal que não faz parte da equipe de programação, a linguagem utilizada nos mesmos deve ser mantida o mais simples possível, pois não se quer transferir a responsabilidade da programação para o pessoal não-técnico [RAB00a]. Em decorrência da limitação de complexidade nas linguagens de script dos jogos, é uma prática difundida não se utilizar scripts para a descrição de comportamentos de maior complexidade, salvo quando a personalização da mesma faz parte do projeto do jogo. A criação de uma linguagem complexa para os scripts também incute um grande aumento de esforço por parte dos projetistas e programadores.

Porém, atualmente se observa um aumento de importância da IA dentro do contexto de desenvolvimento de jogos. Desta forma, é importante que se possa modelar experimentalmente o comportamento inteligente das entidades de um jogo [LAM95]. Este fato torna a prática de se incluir especificidades da IA no código altamente restritiva. Além do mais, o uso de scripts não é desejável para a criação de procedimentos complexos, uma vez que o programador do comportamento deve prever a priori uma grande maioria das possibilidades de percepção das entidades comandadas por um script, senão todas, fato que o torna extremamente grande e sujeito a erros. Assim sendo, torna-se necessária uma abstração diferente para a modelagem de comportamentos mais inteligentes. Uma maneira de se modelar tais comportamentos de uma forma mais compreensível é a utilização de agentes, uma vez que sua descrição envolve elementos como estados mentais, objetivos e comportamentos entre outros [HAD96] [RAO98] [FIS98], dependendo de qual modelo de agência é tomado como base.

6.1.3 Por que utilizar Agentes

Utilizando-se a definição de [SHO93] “Um agente é uma entidade cujo estado é visto como consistindo de componentes mentais tais como crenças, capacidades, escolhas e comprometerimentos. Estes componentes são definidos de uma maneira precisa, e ficam em parca correspondência com seus homônimos consuetudinários”. Pode-se concluir que agentes são definidos de uma forma não ambígua - o que os torna utilizáveis em um contexto computacional -, porém em termos de definições mentais - o que os torna mais inteligíveis para quem os define. Além disso, como mencionado na §3, existem algumas características dos agentes, tais como autonomia, adaptabilidade e sociabilidade, que proporcionam um grau ainda mais elevado de abstração, facilitando a modelagem, por conseguinte.

Este aumento do grau de abstração tende a ser visto por correntes mais conservadoras na indústria com certa desconfiança, uma vez que torna mais difícil fazer um relacionamento direto do que foi definido com a complexidade final do programa. Porém, se for considerado o aumento de complexidade nos jogos produzidos atualmente, e as mudanças que tal evolução ocasionou, ver-se-á que a utilização de uma abstração de mais alto nível é bastante oportuna, principalmente em projetos que visam a melhoria da “jogabilidade”.

6.2 Considerações sobre a Linguagem

Na §6.1.1 a importância da separação dos elementos mutáveis e imutáveis da implementação de um jogo é destacada. Esta separação se realizará na arquitetura ICE através de uma linguagem de definição de comportamento baseada em agentes híbridos, que irá conjugar elementos de definição de agentes BDI e reativos. Logo a linguagem deverá apresentar construções adequadas a descrever características das duas arquiteturas. Esta linguagem irá se chamar IADL¹ e será descrita ao longo deste item utilizando uma gramática da linguagem cujas regras de derivação relativas a cada

¹ ICE Agent Description Language

característica serão apresentadas no item correspondente. Algumas regras da gramática não estão descritas neste capítulo estando a gramática completa listada no Anexo II, no final deste trabalho.

A descrição da linguagem foi inicialmente apresentada no volume final do TC1. Esta descrição previa mudanças que seriam introduzidas após uma consideração mais precisa, resultado de testes de funcionamento realizados no curso do TC2. A descrição da linguagem apresentada neste capítulo consiste apenas da sua forma final, sendo que as mudanças sobre o protótipo original de linguagem são salientadas quando pertinente.

6.2.1 O Agente

A sintaxe definida para a descrição de um agente é bastante semelhante à utilizada em orientação a objetos para a descrição de uma classe, apenas observando-se a mudança de paradigma.

De modo a contemplar as características BDI do agente, para ele deverá ser definido um conjunto de crenças que são relevantes para o mesmo, além de um conjunto de objetivos, que correspondem aos desejos. As intenções do agente não serão definidas na linguagem, uma vez que estas serão obtidas através dos desejos e das crenças em tempo de execução. Como planejamento não é o foco deste trabalho, o conjunto de planos deverá ser especificado através da linguagem.

Quanto à característica reativa, esta será contemplada através de um conjunto de reações definidas em relação a percepções. Comum aos dois modelos de agente utilizados será a definição do conjunto de ações passíveis de execução. Estas características se materializam na seguinte regra:

```
<agent> →
  agent <agent_name>
  {
    <beliefs>
    <actions>
    <objectives>
    <plans>
    <reactions>
  }
```


6.2.2 Tipos

Como explicado na §3.4, a especificação utilizando lógica é complexa e excessivamente lenta, este último fato especialmente importante no contexto de jogos de computador. Desta forma o grupo optou pela definição utilizando tipos simples semelhantes aos existentes em linguagens de programação contemporâneas como C++ e Java. Além dos tipos simples, duas possibilidades de tipos complexos foram adicionadas a linguagem: a construção de tipos compostos, semelhantes a *structs* em C, e a construção de listas de tipos.

Os tipos compostos chamam-se *composite*, e serão disponibilizados de modo a aumentar a legibilidade de crenças sobre uma mesma entidade. As listas foram adicionadas na definição da linguagem para facilitar a expressão de conjuntos de entidades semelhantes, porém apesar de a linguagem prover este tipo de construção, não gera código no protótipo desenvolvido no TC2. Esta funcionalidade não foi implementada, pois o tempo de implementação seria proibitivo dado o objetivo de finalização antecipada do compilador visando a experimentação nos jogos de demonstração. O grupo concluiu que a falta de listas não diminui a expressividade da linguagem, consistindo apenas de uma limitação do ponto de vista prático. Estas definições resultam nas seguintes regras:

```

<simple_type> → int | float | bool | string
<type> → <simple_type> | IDENTIFIER
<type_declaration> → <composite> | <list>
<composite> → composite IDENTIFIER
                {<fields>}
<fields> → <field> ; <fields> |
            <field> ;
<field> → <simple_type> IDENTIFIER
<composite_reference> → IDENTIFIER.IDENTIFIER
<list> → IDENTIFIER list of <type>;
<constant> → <mathematical_constant> |
                STRING_CONSTANT |
                BOOL_CONSTANT
<mathematical_constant> → INTEGER_CONSTANT |
                            FLOAT_CONSTANT

```

Além da definição dos tipos são necessários operadores para os tipos simples e compostos. Novamente, a não inclusão das listas na

implementação implica que as operações com listas não estão disponíveis para uso no protótipo desenvolvido no TC2. Tais operadores estão definidos a seguir:

```

<list_statement> → IDENTIFIER.<list_operation> |
                  <list_iteration>
<list_iteration> → iterate IDENTIFIER with IDENTIFIER
                  {<action_effects>}
<list_operation> → add(IDENTIFIER) | remove()
<list_condition> →
                  <type> IDENTIFIER <list_logical_operator> IDENTIFIER
                  where (<regular_condition>)
<list_logical_operator> → in | forall

<expression> → <expression><binary_operator><expression2> |
               <unary_expression> |
               <constant> |
               <belief_reference> |
               (<expression>)
<unary_expression> → <unary_operator><belief_reference> |
                    <unary_operator><constant>
<expression2> → <constant> |
                <belief_reference> |
                (<expression>)

<unary_operator> → ++ | -- | + | -
<binary_operator> → <logical_operator> |
                  <mathematical_operator> |
                  <comparison_operator>
<logical_operator> → and | or
<mathematical_operator> → + | - | * | /
<comparison_operator> → > | >= | < | <= | ==
<assignment_operator> → = | += | -= | *= | /=

<condition> → <list_condition> | <regular_condition>
<regular_condition> → <expression>

```

6.2.3 Ações

Um elemento comum aos dois modelos utilizados nesta arquitetura híbrida é a noção de ação, que define a capacidade de afetar o ambiente de uma entidade. Ações serão definidas através da seção *action*: na definição do agente.

Para cada ação são especificados parâmetros, que podem modificar o resultado da mesma. Além disto serão especificados os efeitos desta ação nas crenças relacionadas. Quanto aos efeitos das ações foram consideradas duas alternativas de semântica.

Na primeira alternativa o agente executa a sua ação e espera um retorno do ambiente sobre os efeitos da mesma. Nesta alternativa os efeitos de uma ação não são descritos no agente deixando apenas as percepções advindas do ambiente informar os resultados da mesma. Esta alternativa implica que a função de mapeamento de ações em percepções localizada no *Ether* seja mais complexa. Isto torna a implementação mais difícil, porém os resultados são mais interessantes.

Na segunda, que faz com que o agente se comporte como um “agente esquizofrênico” o agente assume que os efeitos de suas ações são sempre verdadeiros e imediatamente após a execução das mesmas, altera sua base de crenças refletindo tais efeitos. Nesta alternativa o agente irá descobrir se falhou caso uma percepção o informe deste fato atualizando suas crenças.

A alternativa escolhida para implementação no ICE foi a do agente esquizofrênico, pois é mais simples de implementar e deixa o componente *Ether* mais simplificado, favorecendo a velocidade do mesmo. Além disso, permite que o agente trabalhe pró-ativamente, levando em consideração a expectativa dos resultados de suas ações. Em caso de detecção de falhas ele pode escolher um novo objetivo.

As construções providas na gramática da linguagem são as seguintes:

```

<actions> → action: <action_declaration_list>
<action_declaration_list> →
    <action_declaration><action_declaration_list> |
    <action_declaration> | ε
<action_declaration> → IDENTIFIER(<parameter_declaration_list>)
    {
        <action_effects>
    }
<action_effects> → <action_effects><action_effect> | ε
<action_effect> → <belief_assignment>; | <unary_expression>; |
    <list_statement>;
<action_call_list> → <action_call_list><action_call>; |
    <action_call>; | ε
<action_call> → IDENTIFIER(<parameters>)

```

6.2.4 Crenças

A primeira estrutura referente ao componente ABDI¹ a ser descrita pela linguagem é a definição das crenças. Estas formam o modelo de mundo do agente e são definidas na forma de identificadores que utilizam os tipos definidos na §6.2.2. As crenças representam, portanto, o que o agente em questão sabe, ou acredita sobre o estado do mundo a sua volta. O estado destas crenças não representa necessariamente a verdade sobre os fatos, podendo estar defasadas, ou inclusive completamente divergentes da realidade. Isto pode acontecer quando o mundo muda e o agente não percebe a mudança. Ou ainda em situações onde o *Ether* foi deliberadamente construído para introduzir informações errôneas.

As regras gramaticais referentes às crenças são as seguintes

```

<beliefs> → belief: <belief_declaration_list>
<belief_declaration_list> →
  <belief_declaration>; <belief_declaration_list> |
  <belief_declaration>; | ε
<belief_declaration> → <type> <belief_declarator_list>
<belief_declarator_list> →
  <belief_declarator_list>, <belief_declarator> |
  <belief_declarator>
<belief_declarator> → IDENTIFIER = <expression> | IDENTIFIER
<belief_assignment> →
  <belief_reference><assignment_operator><expression>
<belief_reference> → IDENTIFIER | <composite_reference>

```

6.2.5 Objetivos

O comportamento pró-ativo do componente ABDI é definido pelos objetivos do mesmo. Na IADL, os objetivos serão definidos através de um conjunto de valores desejados para as crenças. Estes valores serão declarados pela construção “pre” na forma de uma expressão onde os estados desejados para as crenças são ligados por operadores lógicos, quando esta expressão for avaliada como verdadeira então o objetivo em questão terá sido atingido.

Os objetivos serão definidos através das seguintes construções da gramática:

¹ Veja §5.2.2

```

<objectives> → objective: <objective_declaration_list>
<objective_declaration_list> →
  <objective_declaration_list><objective_declaration> |
  <objective_declaration> | ε
<objective_declaration> →
  IDENTIFIER(<parameter_declaration_list>)
  pre(<conditions>)
  pos(<conditions>)
<objective_list> → IDENTIFIER | ε

```

6.2.6 Planos

Considerando o fato de que a construção de um planejador¹ completo não é o objetivo deste trabalho, o planejamento de cada agente deverá ser realizado através de uma biblioteca pré-estabelecida de planos.

Cada agente irá possuir um conjunto de planos onde cada plano terá um objetivo a ser atingido, um conjunto de condições sobre as crenças que determinam a viabilidade de execução do plano e um valor de prioridade que determina a ordem de preferência com que o planejador irá escolher o plano a ser adotado caso mais de uma maneira de se realizar um objetivo seja possível.

A linguagem definida provê as seguintes construções para a definição de planos:

```

<plans> → <plans><plan_declarator> | <plan_ declarator> | ε
<plan_declarator> →
  plan IDENTIFIER(<objective_list>)
  if(<conditions>)
  priority INTEGER_CONSTANT
  {
    <action_call_list>
  }

```

6.2.7 Reações

A funcionalidade do componente AR² será definida por um conjunto de reações. Cada reação será definida por um identificador para a reação, o nome da percepção que irá ativá-la e a ação de resposta caso a percepção

¹ Vide glossário.

² Veja §5.2.2

especificada seja ativada. A palavra reservada *reconsider* serve para denotar a não necessidade de se reconsiderar os objetivos caso a reação seja ativada.

Esta funcionalidade será definida na linguagem através das seguintes construções:

```
<reactions> → <reactions><reaction>; |
              <reaction>; | ε
<reaction > →
              <reaction_modifier> reaction IDENTIFIER
              if (IDENTIFIER)
                  <action_call>
<reaction_modifier> → reconsider | ε
```

6.3 Exemplos de utilização da linguagem

De modo a validar a expressividade da linguagem especificada na §6.2, alguns exemplos de uso serão apresentados nesta seção junto com sua semântica.

6.3.1 Soldado Hans

O soldado Hans é um soldado alemão que tem dois objetivos principais: sobreviver ao combate (*saveLife*) e eliminar o jogador inimigo quando estiver a seu alcance (*killPlayer*). Na sua representação de mundo o outro jogador é representado através de sua saúde, a distância que ele se encontra do jogador, e de sua condição de vivo.

Hans considera importante para a sua representação do mundo o quanto de saúde ele possui (*health*), a sua quantidade de munição (*ammo*) e os dados sobre os jogadores que ele já detectou. Ao ser criado ele acredita que sua saúde vale 10 unidades e que tem 50 unidades de munição.

Para realizar seus objetivos Hans conta com dois planos, uma para manter-se vivo caso algum jogador se aproxime dele quando ele estiver em uma situação crítica (*flee*), e outro para caçar se o jogador estiver ao seu alcance (*chase*).

A sua capacidade de atuação se resume a aproximar-se (*moveTo*) e afastar-se (*moveAway*) do jogador e atirar no mesmo (*shoot*).

Este comportamento seria atingido através da seguinte definição em

IADL:

```

composite Player
{
    bool alive;
    int health;
    float distance;
};

agent Hans
{
    belief:
    int health=10;
    int ammo=50;
    Player player;
    action:
        shoot(Player player)
        {
            ++player.health;
        }
        moveAway(Player player)
        {
            ++player.distance;
        }
        moveTo(Player player)
        {
            --player.distance;
        }
    objective:
        saveLife()
        pre(health < 4)
        pos(false)

        killPlayer(Player player)
        pre((player.alive==true) and (player.distance < 10))
        pos(false)

    plan flee(saveLife)
        if(player.distance < 10)
        priority 1
        {
            moveAway(player);
        }

    plan chase(killPlayer)
        if(player.distance > 10)
        priority 1
        {
            moveTo(player);
            shoot(player);
        }

    reaction getHit if (shooting)
        shoot(player);
}

```

6.3.2 Base de comando

A base de comando pode ser considerada um centro de operações de um jogo de estratégia. Ela estará encarregada de ordenar a unidade para colher recursos, ou defender a base, e ainda poderá cuidar da unidade se necessário.

Ao notar que os recursos da base estão escassos, a base pode ordenar a unidade a colher recursos para abastecimento. Quando a base estiver sob ataque, ela poderá pedir auxílio à unidade que está sob seu comando para combater os atacantes. Caso a unidade sofra muito dano, a base pode ordenar que ela retorne para se recuperar.

Vejamos como este tipo de agente seria descrito utilizando a IADL:

```

composite Unit{
  bool orderResource;
  bool orderDefend;
  bool orderRecover;
  int health;
};

agent Base
{
  belief:
    int resources=100;
    int damage=0;
    bool defend = false;
    Unit unit;
    Unit soldier

  action:
    orderResource(Unit collector)
    {
      collector.orderResource = true;
      collector.orderDefend = false;
      collector.orderRecover = false;
    }

    orderDefend(Unit soldier)
    {
      soldier.orderResource = false;
      soldier.orderDefend = true;
      soldier.orderRecover = false;
    }

    orderRecover(Unit soldier)
    {
      soldier.orderResource = false;
      soldier.orderDefend = false;
    }
  }

```



```

        soldier.orderRecover = true;
    }

    objective:
        saveLife()
            pre(defend == true)
            pos(false);
        getResources()
            pre(resources < 20)
            pos(false);
        saveUnit(Unit unit)
            pre(unit.health < 5)
            pos(false);

    plan defense(saveLife)
        if(true)
            priority 1
    {
        orderDefend(soldier);
    }

    plan resources(getResources)
        if(true)
            priority 1
    {
        orderResource(collector);
    }

    plan help(saveUnit)
        if(true)
            priority 1
    {
        orderRecover(unit);
    }
}

```

6.3.3 Bola do Tcheco Balls

O agente abaixo representa as unidades coloridas do jogo. O objetivo da unidade é chegar em sua casa (*getHome*) e, após isso, dormir (*sleep*). Para isso, ela quer saber, através das suas crenças, quando ela está em alguma casa (*inAnyHome*) ou quando ela está na casa de sua respectiva cor (*inHome*).

O planejamento que ela possui para atingir seus objetivos é bem simples, enquanto ela não está em casa (ou seja, o objetivo *getHome* é o corrente) ela continua se movendo (*findHome*).

Para conseguir informações do ambiente (e atualizar suas crenças) ela possui três reações para captar percepções: *gotHome*, *gotAnyHome* e

gotKicked. *GotHome* indica quando ela chegou em alguma casa colorida qualquer. *GotAnyHome* indica quando ela chegou em sua casa colorida. *GotKicked* indica quando ela foi expulsa da casa onde estava dormindo. Esta ultima acontece quando a unidade está dormindo em uma casa que não a pertence e a unidade respectiva da casa aparece e a desaloja.

```
agent Ball
{
  belief:
    bool inHome=false;
    bool inAnyHome=false;

  action:
    move()
    {
    }

    foundHome()
    {
      inHome = true;
      inAnyHome = true;
    }

    foundAnyHome()
    {
      inAnyHome = true;
    }

    kicked()
    {
      inAnyHome = false;
    }

  objective:
    getHome()
      pre(inAnyHome==false and inHome==false)
      pos(inAnyHome==true)
    sleep()
      pre(inAnyHome==true)
      pos(false)

  plan findHome(getHome)
    if(true)
      priority 1
      {
        move();
      }

  reaction gotHome if(foundHome)
    foundHome();

  reaction gotAnyHome if(foundAnyHome)
    foundAnyHome();

  reaction gotKicked if(wasKicked)
```

```

        kicked();
    }

```

6.3.4 AntBot

A AntBot é uma formiga que tem em sua base de conhecimento *hungry* indicando seu nível de fome, *health* indica a saúde da AntBot e tem relação direta com o atributo *hungry*, pois a saúde dela depende do fato de estar bem alimentada, ou seja, a fome não pode ultrapassar o limite médio de 50 unidades.

Existe um predador que poderá atacá-la ou não, e este terá como característica o grau de ameaça que representa para a AntBot (*treat*), seu nível de vida (*life*) e a que distância se encontra (*distance*).

A AntBot sabe a distância existente entre ela e o formigueiro (*antHill*) e tem conhecimento da necessidade de armazenar alimento ou não através do atributo booleano *store*.

A AntBot não necessita ir ao formigueiro somente para armazenar alimento, mas também pode ir para lá após um combate, pois precisará descansar (atributo *rest*) para recuperar suas forças. Ela sabe que se comer o alimento existente no formigueiro terá de sair em busca de mais para reposição. Seus objetivos primordiais são alimentar-se, buscar alimento e armazenar no formigueiro, manter-se viva e de preferência bem nutrida (com um nível de fome inferior a 50 unidades) e, ainda, fugir do predador/ameaça.

Este exemplo está descrito em IADL logo abaixo:

```

composite Predator {
    bool treat;
    int distance;
    int smell;
    int life;
    bool detect;
};

composite Food {
    int distance;
    int smell;
    int weight;
};

```

```

agent AntBot
{
  belief:
    int hungry=0;
    int health=100;
    int antHill=150;
    bool store=true;
    bool rest=true;

    Predator predator;
    Food food;

  action:

    goFood(Food food)
    {
      --food.distance;
      ++food.smell;
    };

    goAway(Predator predator)
    {
      ++predator.distance;
    };

    goTo(Predator predator)
    {
      --predator.distance;
    };

    attack(Predator predator)
    {
      --predator.life;
    }

    goAntHill()
    {
      --antHill;
    }

    leaveFood(Food food)
    {
      ++food.distance;
      --food.smell;
      ++antHill;
    }

    lostHealth()
    {
      --health;
    }

    storeFood()
    {
      store=true;
    }

    goRest()
    {

```

```

        rest=true;
    }

    objective:

        feed()
        pre((hungry>50) and (foods.distance<20) and
foods.smell>80))
        pos(false)

        getFood()
        pre((store==true) and (foods.distance==0) and
(foods.weight<50))
        pos(store==false)

        foodInPantry()
        pre(antHill==0)
        pos(false)

        stayAlive()
        pre(health<50)
        pos(false)

        stayWellNourished()
        pre(hungry>80)
        pos(false)

        goToRest()
        pre(rest==true)
        pos(false)

        isThreaten()
        pre(predator.distance<50)
        pos(false)

    plan eatInAntHill(feed)
        if ((foods.distance<20) and (foods.smell>80))
        priority 1
        {
            goFood ();
            storeFood();
            goRest();
        }

    plan eat (feed)
        if ((foods.distance<20) and (foods.smell>80))
        priority 1
        {
            goFood();
        }

    plan runAway (isThreaten)
        if ((predator.distance<50) and (predator.treat>5))
        priority 1
        {
            goAway ();
        }

```

```

plan goAttack (isThreaten)
  if ((predator.distance<50) and (predator.treat<5))
  priority 1
  {
    goTo();
    attack();
    goRest();
  }

plan live(stayAlive)
  if(true)
  priority 1
  {
    goAway();
  }

plan fat (stayWellNourished)
  if(true)
  priority 1
  {
    goFood();
  }

plan goPantry (getFood)
  if ((foods.distance==0) and (foods.smell==100))
  priority 1
  {
    goAntHill ();
  }

plan goOutPantry(foodInPantry)
  if ((foods.distance==0) and (foods.smell==100))
  priority 1
  {
    leaveFood();
  }

reaction getTeethmark if (attack)
  lostHealth();

reaction emptyStomach if (stayWellNourished)
  lostHealth();
}

```

6.3.5 Formiga

O Agente Formiga tem em sua base de conhecimento as informações necessárias para que possa caminhar no labirinto e cumprir seus objetivos. Por exemplo, considerando que seu campo de visão se restringe a quatro espaços (acima, abaixo, esquerda e direita) ela perceberá do ambiente:

- quando existir um inimigo próximo, o que a obrigará a fugir (`temInimigo=true`);
- quando houver comida em seu campo de visão ela poderá comê-la (`temComida=true`);
- ou ainda, quando não existir nada em seu campo de visão, isto a fará andar para encontrar comida (`soAndar=true`).

Quando perceber a presença de um inimigo, seu objetivo será fugir dele. Não importando se há comida disponível. O plano para cumprir este objetivo contém a ação `foge()` que deverá fazê-la andar mais rápido na direção oposta ao inimigo. Da mesma forma, que se encontrar comida, deverá comê-la através da ação `moveC()`. Outro de seus objetivos é percorrer o labirinto sempre que não encontrar nada ao seu redor executando a ação `moveA()`. A cada ciclo de processamento a Formiga receberá percepções do ambiente que lhe indicarão o que há ao seu redor, partindo destas, virão suas decisões em detrimento de seus objetivos.

Logo abaixo, segue a descrição do agente Formiga na IADL proposta:

```
agent Formiga
{
  belief:
    bool temComida=false;
    bool temInimigo=false;
    bool soAndar=true;

  action:
    comer()
    {
      temComida=true;
      soAndar=false;
    }

    moveC()
    {
      temComida=false;
      soAndar=true;
    }

    acheiInimigo()
    {
      temInimigo= true;
      soAndar=false;
    }
}
```

```

    }

    semInimigo()
    {
        temInimigo=false;
        soAndar=true;
    }

    foge()
    {
    }

    moveA()
    {
    }

objective:
    fugirInimigo()
        pre(temInimigo==true)
        pos(temInimigo==false)
    buscarComida()
        pre(temComida==true)
        pos(false)
    andarLabir()
        pre(soAndar==true)
        pos(false)

plan correrInimigo(fugirInimigo)
    if(true)
    priority 1
    {
        foge();
    }

plan comerComida(buscarComida)
    if(true)
    priority 1
    {
        moveC();
    }

plan andarLabirinto(andarLabir)
    if(true)
    priority 1
    {
        moveA();
    }

reconsider reaction fugir if (inimigo)
    acheiInimigo();
reaction semPerigo if (nInimigo)
    semInimigo();
reaction comer if (comida)
    comer();
}

```


7 MODELAGEM E IMPLEMENTAÇÃO

A utilização do ambiente ICE será baseada em dois componentes de software distintos: um *framework* de objetos que será utilizado no código do *Game Engine* do jogo, e um compilador para os agentes deste ambiente. Este *framework* de objetos irá suportar a utilização dos agentes ICE, que serão gerados pelo compilador ICE, que irá gerar agentes que estendem a funcionalidade de um agente básico que conterà os algoritmos de comunicação com o *framework*.

Estes componentes serão apresentados nas seções a seguir descrevendo as alternativas de implementação adotadas além de sua relação com a arquitetura conceitual apresentada na §5.

7.1 Compilador

Uma das características fundamentais do ambiente ICE é a possibilidade de pessoal não-técnico criar e modificar comportamentos definidos por uma linguagem de mais alto nível do que as linguagens de programação tradicionais. Para este fim foi implementado um compilador que traduz descrições de agentes feitas na linguagem IADL, que está definida na §6.2, estas descrições são transformadas em um conjunto de classes que estendem um conjunto de classes básicas, como descrito na Figura 16.

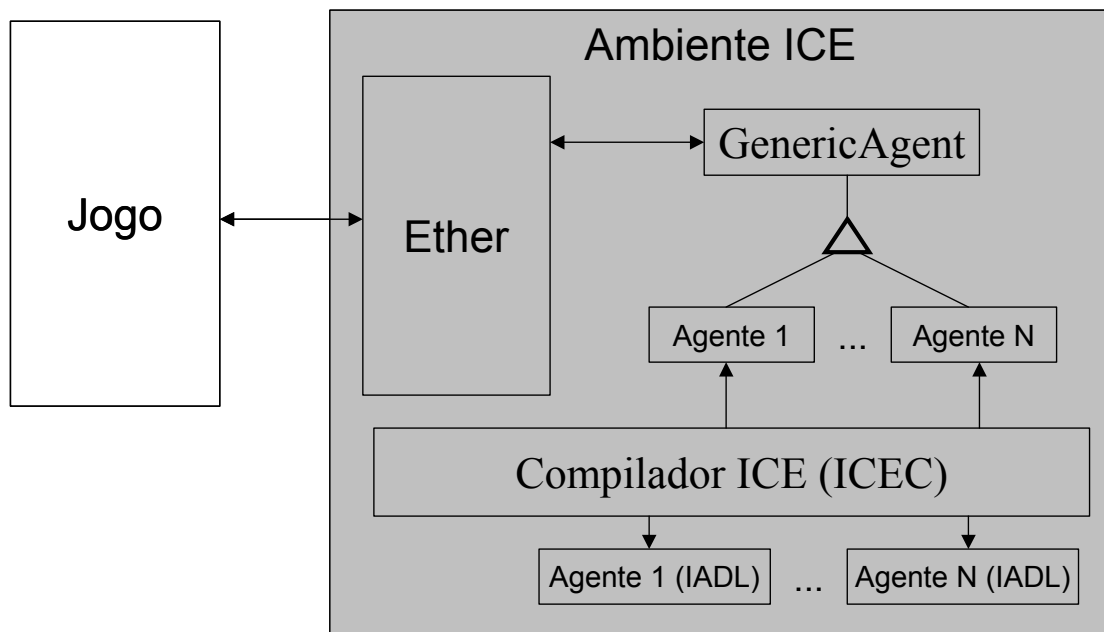


Figura 16 Visão geral do ambiente ICE

7.1.1 Compilação versus Interpretação

Na §6.1.2 foi apresentada a importância de se parametrizar o comportamento das entidades de um jogo e desta forma evitar codificar este tipo de informação nos fontes do programa. Logo a idéia de um compilador cuja saída é um conjunto de classes está, de certa forma, em conflito com os objetivos delineados no capítulo anterior. Entretanto é necessário ter em mente que um dos mais importantes requisitos de um jogo de computador é a velocidade e a inserção do comportamento no código é uma alternativa aceitável nas fases finais de desenvolvimento de um jogo. E foi a alternativa escolhida no curso deste trabalho por ser de implementação mais rápida e de depuração mais simples.

Não obstante, é perfeitamente possível a criação de um interpretador da IADL, permitindo desta forma a utilização do ambiente ICE da forma proposta na §6.1.2, bastando para isto a criação de uma extensão dinâmica das classes genéricas descritas no restante deste capítulo.

7.1.2 Ferramentas

Para a criação do compilador da IADL foram utilizadas variantes do par de ferramentas Lex e YACC chamados ALex e AYACC. Estas ferramentas acompanham uma interface de desenvolvimento chamada *Parser Generator* que é distribuída na forma de *shareware* além de ser de distribuição gratuita para a comunidade acadêmica.

As ferramentas do *Parser Generator* têm como vantagem a geração de código orientado a objetos, de acordo com a filosofia de projeto utilizada neste trabalho. Esta ferramenta faz uso de uma classe abstrata básica de analisador léxico chamada “ylexer”, que contém os algoritmos de caminhamento nas tabelas geradas pela gramática regular que define os *tokens* da linguagem. Estas tabelas são introduzidas em uma classe filha do analisador léxico básico.

A ferramenta de geração de *parsers* do *Parser Generator* chamada AYACC também faz uso de uma classe abstrata básica contendo algoritmos de caminhamento nas tabelas geradas a partir da gramática especificada e introduzidas em uma classe filha. Este programa é capaz de gerar *parsers* dos tipos *Simple LR*, *Lookahead LR* e *Canonical LR*, sendo que o tipo de *parser* utilizado na implementação do compilador da IADL é *Lookahead LR*.

7.1.3 Módulos do compilador

O compilador da IADL foi dividido em três módulos:

- Analisador léxico e sintático: a estrutura deste módulo foi gerada através das ferramentas descritas na §7.1.2, utilizando a gramática descrita na §6.2. Este módulo utiliza o módulo construtor de nodos, explicado em detalhes na §7.1.5 para criar uma representação em forma de árvore da descrição do agente;
- Construtor de nodos da árvore sintática: este módulo é responsável pela criação e junção dos componentes da árvore sintática gerada ao longo do processo de *parsing*, esta árvore sintática equivale ao

código intermediário da definição do agente. O construtor foi inspirado no padrão *Builder* em [GAM94]. Este padrão foi levemente modificado devido ao modo de funcionamento das classes geradas pelas ferramentas utilizadas;

- Gerador de código: após a árvore sintática ter sido criada, esta é transformada em uma representação em termos de uma estrutura de classes de objetos em conformidade com a especificação da arquitetura ICE. Ao final do processo de compilação a representação Orientada a Objetos é convertida em arquivos fonte na linguagem alvo, que no caso deste trabalho é C++.

7.1.4 Analisador léxico e sintático

O processo de compilação no ICEC é encabeçado pelos analisadores léxico e sintático. Estes analisadores são criados utilizando-se as ferramentas já descritas, tendo como fonte a gramática da IADL. Devido à forma com que as ferramentas utilizadas operam, foi necessária a criação de um módulo que isolasse a geração de código intermediário do módulo de análise sintática. As regras especificadas na definição da gramática da linguagem para o analisador sintático são acompanhadas de regras da gramática de atributos utilizada para a geração do código intermediário. Nas regras da gramática de atributos são feitas chamadas para o módulo construtor de nodos da árvore sintática, de modo a gerar o código intermediário.

7.1.5 Construtor de nodos da árvore sintática

Este módulo é responsável pela criação dos nodos da árvore sintática, que corresponde à representação em código intermediário do agente sendo compilado. Ele é composto basicamente por métodos para a criação de tipos específicos de nodos, cada um representando um componente lógico da definição de um agente. Estes nodos podem ser vistos na Figura 17 e são todos derivados de uma classe comum, chamada de *Node*, que contém as funcionalidades referentes a estrutura básica da árvore sintática. Ao final do

7.1.6 Gerador de código

O módulo gerador de código é responsável pela transformação do código intermediário nas classes que compõem um agente. Este módulo é composto pelos algoritmos de geração de código, suportados por meta-classes, ou seja, classes que descrevem outras classes, como visto na Figura 18.

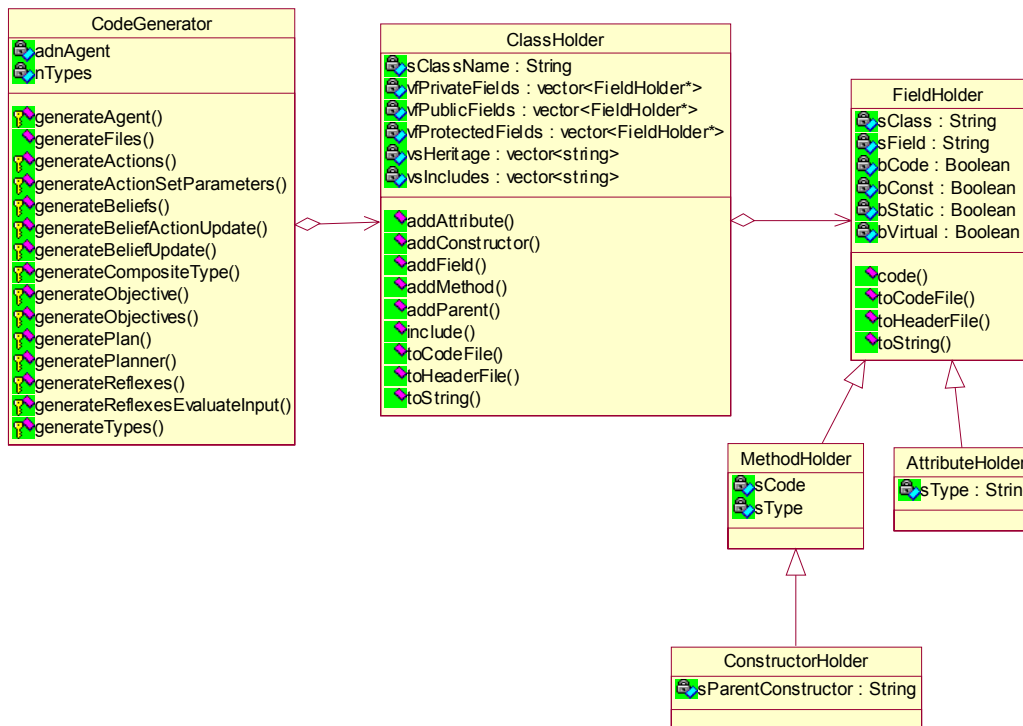


Figura 18 Classes do gerador de código

As meta-classes definidas para o Gerador de código contêm elementos genéricos comuns a definições de classes em orientação a objetos. A principal delas é a classe `ClassHolder`, que representa uma classe. Os elementos são adicionados à definição da mesma na forma de campos, onde cada um é especializado representando uma parte da definição de uma classe, estes campos são codificados na classe `FieldHolder`.

A classe `FieldHolder` contém descritores para características comuns a definições de campos em diversas linguagens orientadas a objetos, permitindo que se defina campos de classe e de instância, campos constantes

e campos que podem ser sobrecarregados em especializações da classe. As especializações para os campos de uma classe, definidas para o ICEC, são:

- `AttributeHolder`: representa um atributo dentro de uma classe;
- `MethodHolder`: representa um método dentro da classe, este campo contém o tipo de retorno de um método, e o código do mesmo;
- `ConstructorHolder`: é uma especialização de `MethodHolder` que contém um construtor, nele é possível especificar um construtor da classe pai para ser invocando concomitantemente com o mesmo;

Esta estrutura do gerador de código permite que o ICEC seja facilmente adaptado para a geração de código em outra linguagem orientada a objetos, como, por exemplo, Java.

7.2 Ambiente ICE

A arquitetura ICE é dividida em três componentes básicos, cada um desempenhando uma função distinta no ambiente do jogo. O modelo deste ambiente será primeiramente exposto de maneira geral e em seguida seus detalhes serão explorados.

7.2.1 Arquitetura Básica

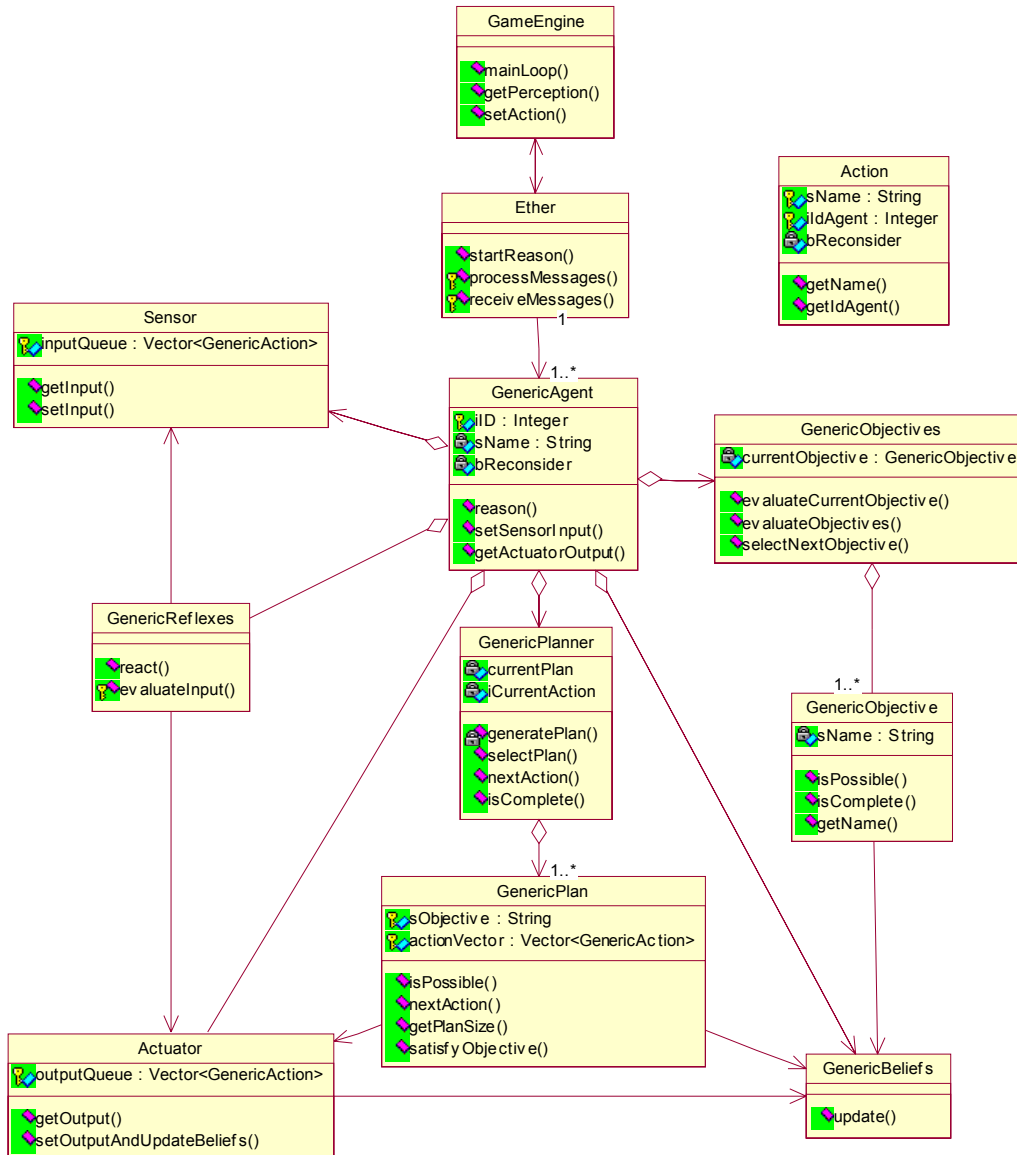


Figura 19 Diagrama de classes do ambiente ICE

O ICE foi modelado para ser, em seu centro, um conjunto de classes genéricas que fornecem a funcionalidade básica do sistema. Portanto, para adicionar as características específicas de cada agente, estas classes são estendidas com atributos e métodos sobrecarregados pela versão compilada da descrição do agente, descrita pela IADL.

O ponto de ligação entre o ICE e o jogo desenvolvido se dá através da classe *Ether*. Ela deve ser estendida manualmente (ou seja, o compilador da IADL não gera uma versão desta classe) e seu funcionamento está diretamente ligado ao jogo. Esta é a única classe do ICE cujo *Game Engine* necessite possuir algum conhecimento.

A seguir, uma breve descrição de cada classe da arquitetura:

- *Game Engine*: classe que representa o jogo. Não faz parte do ICE, mas é o ponto de contato com o jogo.
- *Ether*: classe que armazena os agentes, ordena sua execução e se comunica com o ambiente.
- *GenericAgent*: classe central do agente. Mantém referência para suas classes auxiliares e possui o algoritmo de pensamento do agente.
- *Sensor*: classe de entrada de ações do agente.
- *Actuator*: classe de saída de ações do agente.
- *GenericReflexes*: classe que representa a parte reativa do agente ICE.
- *GenericBeliefs*: classe que armazena as crenças e as ações que o agente pode realizar.
- *GenericObjectives*: classe que armazena e escolhe objetivos.
- *GenericObjective*: classe que representa um objetivo.
- *GenericPlanner*: classe que armazena, escolhe e executa planos.
- *GenericPlan*: classe que representa um plano.

Todas as classes que começam com *Generic* são estendidas pelo código gerado pelo compilador ICE.

7.2.2 Ciclo de funcionamento

O ciclo completo de funcionamento da arquitetura começa pelo *Game Engine* e passa por todos os componentes do agente, incluindo as partes estendidas. Existem duas possibilidades de caminhamento no fluxo de execução do algoritmo básico utilizado pela arquitetura ICE, vide Figura 20 e Figura 21. Estas possibilidades estão diretamente relacionadas à necessidade de reconsideração, ou seja, a necessidade de se escolher um novo objetivo. A necessidade de reconsideração pode ser causada por dois motivos: o objetivo atual ter sido completado, ou ter se tornado inviável.

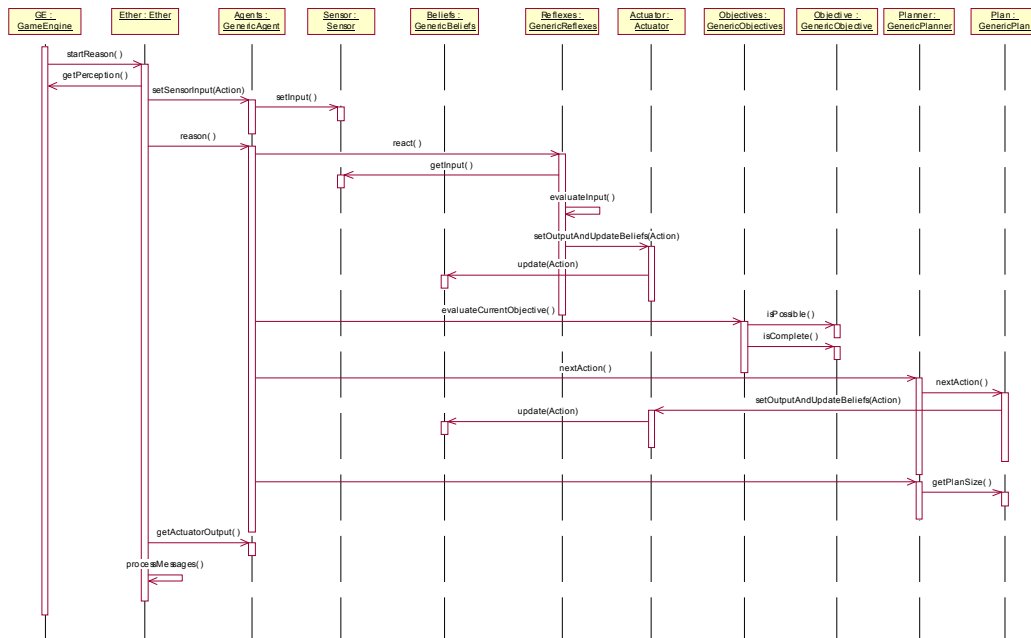


Figura 20 Ciclo de funcionamento do ICE caso não ocorra reconsideração

Este ciclo será descrito, passo a passo, a seguir:

- O *Game Engine*, em seu laço principal, passa o processamento para o *Ether*;
- O *Ether*, para cada agente, coleta suas percepções do ambiente e as registra no agente, e transfere a execução para este;
- O agente, por sua vez, realiza primeiro seu processamento reativo;

- A seguir, verifica se o objetivo escolhido (caso exista) ainda é valido;
- Verifica a necessidade de reconsideração de objetivos e planos (baseado em resultados anteriores);
- Caso seja necessário reconsiderar, escolhe um novo objetivo e plano (vide Figura 21 e Figura 20);
- A seguir, executa um passo do plano selecionado;

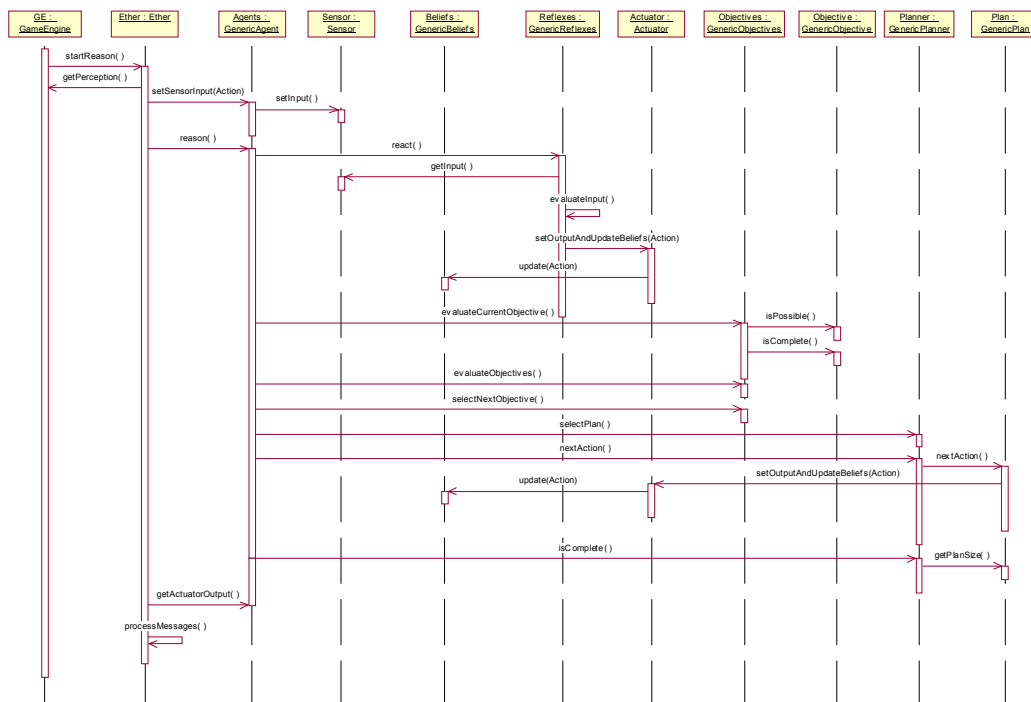


Figura 21 Ciclo de execução do ICE caso ocorra reconsideração

7.3 Ether e Game Engine

7.3.1 Classe GameEngine

Classe principal de um jogo. Ela possui, normalmente, um *loop* principal onde o processamento geral do jogo (como atualização de tela, entrada de periféricos, etc) é realizado. Este *loop* é o ponto de contato entre o

jogo e o ICE. Nele será chamado o método `startReason` da classe `Ether` para que os agentes processem.

7.3.2 Classe Ether

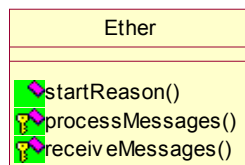


Figura 22 Classe `Ether`

Esta é a classe que controla a execução dos agentes e faz a representação do ambiente para estes. Possui uma referência para todos os agentes que devem realizar alguma execução.

O *Ether* é ativado a partir do *Game Engine*, através da chamado ao método `startReason`. Possui métodos virtuais (que devem ser estendidas por uma implementação filha desta classe) para a captura e envio de ações em relação ao ambiente. Isto pois estas informações são muito dependentes do ambiente, portanto foram relegadas para a implementação específica do jogo.

Métodos principais:

- `startReason`: faz com que, para todos os agentes no sistema, seja realizados a captura de percepções do ambiente, a inserção destas percepções no agente, a transferência de processamento para o agente e, finalmente, a captura e o envio das ações processadas pelo agente para o ambiente.
- `receiveMessages` e `processMessages`: são, respectivamente, os métodos de recebimento de percepções e envio de ações para o ambiente. Eles devem ser implementados por uma versão estendida da classe `Ether`, pois são dependentes do ambiente do jogo.

7.4 Agente

As unidades comportamentais do ambiente ICE são agentes descritos nos capítulos anteriores. Diversas classes de objetos foram criadas para representar as abstrações apresentadas na definição teórica do agente de modo a reproduzir com a máxima fidelidade os conceitos estudados pelo grupo. As classes que perfazem o agente ICE estão descritas no diagrama de classes da Figura 23, sendo que as classes representadas no diagrama estão explicadas nas seções subseqüentes.

7.4.1 Diagrama de classes

De modo a modularizar o funcionamento interno do agente os diversos componentes dos modelos de agência utilizados foram modelados na forma de classes, e dependendo da granularidade escolhida, alguns de seus componentes internos também. O resultado desta modelagem pode ser visto na Figura 23.

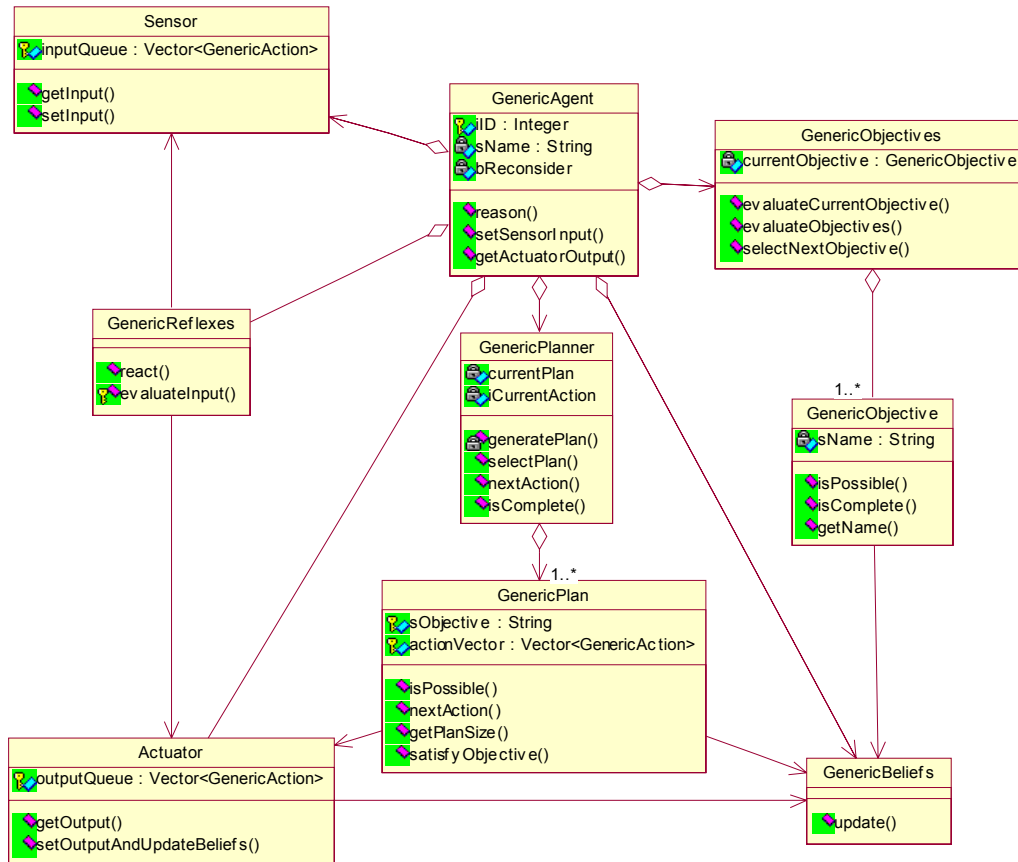


Figura 23 Diagrama das classes internas do agente ICE

7.4.2 Classe GenericAgent

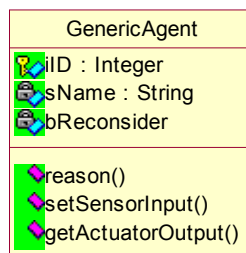


Figura 24 Classe `GenericAgent`

A classe `GenericAgent` personifica a abstração do agente para o *Ether*, de forma com que o algoritmo de passagem de mensagens contido no *Ether* tenha uma interface uniforme para os agentes ICE qualquer que seja a representação e organização interna dos mesmos. Esta classe foi modelada

tendo em mente o padrão *Facade* de [GAM94], sendo sua principal função reduzir o acoplamento entre o sistema do *Ether* e os sistemas internos de cada agente.

Duas das principais tarefas que o *Ether* realiza sobre cada agente são a entrega e recolhimento de mensagens representado as ações e percepções, dois métodos são providos na interface do agente: `setSensorInput`, para a inserção de percepções no sensor do mesmo, e `getActuatorOutput`, para a remoção das ações marcadas para execução pelo atuador.

Apesar de os sistemas multi-agente possuírem conceitualmente múltiplos processos, na arquitetura ICE optou-se por apenas emular o comportamento de um sistema multi-processo. Esta opção foi feita de modo a evitar o *overhead* causado pelo uso de *threads*, que apesar de serem processos ditos leves, incorrem um certo gasto de memória para manutenção e processamento para a troca de contexto [DAW01]. Este gasto é bastante elevado especialmente quando se leva em consideração que seriam necessárias dezenas ou talvez centenas de *threads* caso fosse utilizada uma por agente [DAW01]. Logo, o resultado desta alternativa de implementação é o aumento do desempenho geral dos jogos que utilizam a arquitetura ICE. Esta emulação é atingida utilizando um método que passa a linha de execução para os sistemas internos do agente para que o mesmo possa ter uma fatia do tempo de processamento para realizar seus processos de reação e deliberação, este método é chamado `reason`.

O ciclo de execução de um agente individual no método `reason` consiste em ativar o componente reativo, e em seguida ativar o componente BDI. A versão estendida desta classe muda apenas o seu construtor, assim referenciando as classes também estendidas.

7.4.3 Classe Sensor

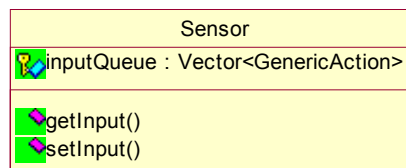


Figura 25 Classe `Sensor`

Esta classe armazena a entrada de percepções do agente. Ela possui uma fila de `GenericAction` e sua lógica é a de manipulação de entrada e saída em filas.

Quem insere nesta fila é o `GenericAgent`, ao receber percepções do ambiente, e quem retira é o `GenericReflexes`, ao tratar as reações.

7.4.4 Classe Actuator

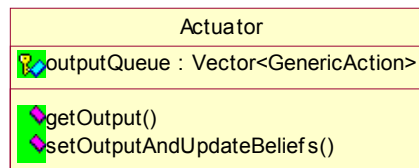


Figura 26 Classe `Actuator`

Esta classe armazena a saída de ações do agente. Como o `Sensor`, ela possui uma fila de `GenericAction` e sua lógica é a de manipulação de entrada e saída em filas, apenas com uma pequena diferença na entrada.

O método de inserção no `Actuator` possui uma funcionalidade extra, a de “executar” o resultado da ação no agente, antes de inserir esta na fila. Esta semântica foi adicionada para facilitar o controle da atualização das crenças, pois, como o agente é esquizofrênico (ou seja, ele acredita que suas ações obtiveram resultados), o melhor momento para esta atualização é quando a ação é enfileirada no `Actuator`, onde ela está prestes a ser retirada do agente. O nome deste método é `setOutputAndUpdateBeliefs`, e a atualização as crenças é feita através do método `update` do `GenericBeliefs`.

Quem insere na fila do `Actuator` é o `GenericReflexes`, ao gerar a ação reativa e o `GenericPlan`, ao executar a ação planejada.

7.4.5 Classe `GenericReflexes`

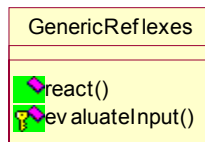


Figura 27 Classe `GenericReflexes`

Esta classe representa a parte reativa do agente ICE. Ela é ativada pelo `GenericAgent` como o primeiro passo do processo de pensamento, através do método `react`. Esta classe tem conexão direta com o `Sensor`, pois é ela que recebe a entrada de percepções do agente, e com o `Actuator`, pois as reações geram ações a serem executadas.

Métodos Principais:

- `react`: método que começa o processamento das reações do agente. Ele é encarregado de retirar todas as percepções do `Sensor` e tratá-las chamando o método `evaluateInput` (descrito abaixo). Este método retorna um booleano que indica se recebeu alguma reação que necessita uma reconsideração de objetivos do agente.
- `evaluateInput`: método que recebe uma percepção (na forma de `GenericAction`) e a trata, ou seja, caso exista uma reação para esta percepção, gera uma ação de resposta correspondente. Como as informações de reações e ações são dependentes do agente, este método deve ser sobrecarregado pela versão compilada da IADL respectiva.

7.4.6 Classe GenericBeliefs

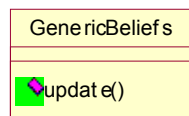


Figura 28 Classe `GenericBeliefs`

O modelo do mundo que cerca cada agente é representado no modelo BDI pelas crenças (*Beliefs*), como visto na §3.4.2. Este componente é representado pela classe `GenericBeliefs`, cuja principal função é atualizar o modelo do mundo de acordo com as ações que foram despachadas para execução no `Actuator`. Os efeitos de cada ação estarão descritos em extensões desta classe específicas para cada agente de modo com que no momento da submissão de uma ação, seus resultados sejam refletidos no modelo de mundo interno do agente.

O único método componente da interface padrão desta classe é chamado `update`, que é responsável pelo processamento de mensagens que encapsulam uma ação e a ativação dos resultados da mesma.

7.4.7 Classe GenericObjectives

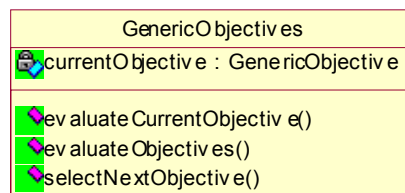


Figura 29 Classe `GenericObjectives`

Os objetivos de um determinado agente que utilize o modelo BDI, como visto na §3.4.2, são representados pelo componente *Desires*, que equivalem aos objetivos do mesmo, e está representado no componente `GenericBeliefs`.

Esta classe armazena os objetivos do agente e possui o algoritmo de escolha de objetivos. Seus métodos são sempre invocados pelo algoritmo de pensamento do `GenericAgent`.

Além de possuir uma lista com todos os objetivos do agente, ao chamar o método `evaluateObjectives` esta classe cria uma lista auxiliar que contém todos os objetivos possíveis no ciclo atual. Esta lista tem o objetivo de auxiliar na escolha do plano de execução, pois o `GenericPlanner` poderá achar um plano que satisfaça o objetivo principal porém não satisfaça a condição a ser executada. Neste caso, outro objetivo possível deve ser escolhido.

Métodos Principais:

- `evaluateCurrentObjective`: método que avalia se o objetivo corrente (caso exista) não pode ser executado (pré-condição é inválida) ou se ele já está completo (pós-condição é válida). Caso de alguma das premissas anteriores for verdadeira, o método retorna a necessidade da reconsideração de objetivos.
- `evaluateObjectives`: método que cria a lista auxiliar de objetivos avaliados.
- `selectNextObjective`: método que pega da lista auxiliar um objetivo que não foi ainda considerado pelo `GenericPlanner`. Caso não encontre nenhum objetivo que possa ser executado, retorna `NULL` e então o algoritmo de pensamento do `GenericAgent` saberá que não existirá um plano neste ciclo.

7.4.8 Classe `GenericObjective`

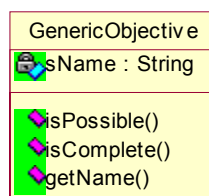


Figura 30 Classe `GenericObjective`

Classe que representa um objetivo do agente. Dois componentes perfazem a definição de um objetivo, uma pré-condição que define quando um objetivo é viável, e uma pós-condição que define quando um objetivo foi atingido. Cada um destes componentes é representado na forma de uma

expressão lógica em termos do conjunto de crenças do agente, que são avaliadas antes da seleção de um objetivo e durante o processo de execução de um plano que visa completar o objetivo selecionado. É armazenada pelo `GenericObjectives` e é um dos parâmetros para a escolha de um plano, no `GenericPlanner`. Possui uma ligação com o `GenericBeliefs` para a verificação da pré-condição e da pós-condição.

Métodos Principais:

- `isPossible`: método que verifica se a pré-condição é válida, ou seja, se o objetivo é válido de ser perseguido.
- `isComplete`: método que verifica se a pós-condição é válida, ou seja, se o objetivo esta completado.

7.4.9 Classe `GenericPlanner`

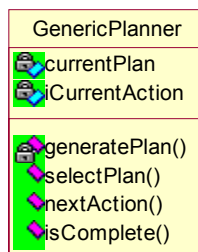


Figura 31 Classe `GenericPlanner`

A escolha em alto nível do curso de ação que um determinado agente irá tomar é feita através dos Objetivos. A fim de concretizar o objetivo selecionado é necessário que ações mais específicas sejam tomadas. A seleção destas ações é responsabilidade de um componente planejador, representado na arquitetura ICE pela classe `GenericPlanner`, que conterà a interface básica para seleção de planos.

No escopo deste trabalho o planejamento consistirá apenas da execução de um conjunto pré-definido de planos para cada tipo de agente, planos estes especificados na definição do agente em IADL, logo esta é uma classe de armazenamento, escolha e execução de planos. O funcionamento

desta classe está ligado com o algoritmo de pensamento do `GenericAgent`, pois ele determina quando um novo plano deve ser escolhido.

Métodos Principais:

- `selectPlan`: método que seleciona um plano baseado em um objetivo (no caso, `GenericObjective`). Ele verifica, na lista de todos os planos, qual satisfaz o objetivo e se este pode ser executado (dependendo de sua condição). Caso encontre, este será o plano escolhido.
- `nextAction`: método que executa a próxima ação do plano selecionado. Ele chama o método de mesmo nome do `GenericPlan` selecionado.

7.4.10 Classe `GenericPlan`

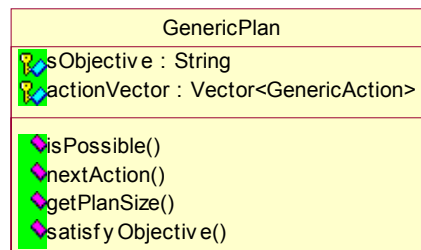


Figura 32 Classe `GenericPlan`

Classe que representa um plano do agente. Estes planos são armazenados pelo `GenericPlanner`. Esta classe possui uma lista de ações, geradas pelo compilador, que descrevem o plano. Possui também uma ligação com o `Actuator`, para a execução das ações, e com o `GenericBeliefs`, para verificar a possibilidade de execução do plano.

Métodos Principais:

- `nextAction`: método que executa a ação corrente. Ele retira uma referência da ação da lista e envia para o `Actuator`.
- `satisfyObjective`: método que verifica se o plano tem o objetivo de satisfazer o `GenericObjective` passado por parâmetro.

7.5 Geração de código

No intuito de facilitar a compreensão do processo de geração de código, utilizaremos duas descrições de agentes como base para esta seção. Como um primeiro exemplo, suponhamos que se deseja modelar um agente capaz de comportar-se como um soldado.

Como se trata de um agente híbrido, conforme a arquitetura ICE propõe na seção §5.2.2, este agente possui características puramente reativas e características que também o classificam como BDI, ou seja, ele teria uma base de crenças com atributos como direção, munição, distância, entre outros, além de ser capaz de atirar contra um inimigo virando-se para a esquerda ou para a direita, podendo mover-se nestas direções inclusive. Seus objetivos seriam basicamente caçar seus inimigos para eliminá-los, e esconder-se quando estivesse sem munição. Apresentaria reações imediatas a determinados acontecimentos como, por exemplo, ouvir um barulho à esquerda, ou à direita, ou mesmo à sua frente, desta vez indicando a presença de um inimigo visível (o inimigo é considerado visível quando estiver no campo de visão do agente, ou seja, neste caso a sua frente).

7.5.1 O agente

Esta seria a descrição textual de um agente simples, que pode ser facilmente transcrita para a IADL. Porém, quando se deseja gerar código acerca desta descrição, muitos são os detalhes a serem considerados na busca de uma boa performance do agente além da garantia de que o código gerado será fiel à descrição dada pelo desenvolvedor do jogo para o agente em questão. Logo abaixo temos o “agente Soldado”, como o chamaremos daqui por diante, descrito pela IADL:

```
agent Soldado
{
  belief:
    int direction=0;
    int ammo=5;
    boolean cover = false;
    int distance = 0;
    int lastTurn = -1;
```

```

action:
    shoot()
    {
        ammo--;
    }

    turn(int dir)
    {
        direction += dir;
        lastTurn = dir;
    }

    move(int size)
    {
        distance += size;
    }

    takeCover()
    {
        cover = true;
    }
objective:
    kill()
        pre(ammo>0)
        pos();
    hide()
        pre(ammo==0)
        pos();
plan huntLeft(kill)
    if(lastTurn == 1)
    {
        move(1);
        turn(-1)
    }
plan huntRight(kill)
    if(lastTurn == -1)
    {
        move(1);
        turn(1)
    }
plan run(hide)
    {
        move(5);
        takeCover();
    }

reaction turnLeft if noiseLeft
    turn(-1);
reaction turnRight if noiseRight
    turn(1);
reaction shootFront if noiseFront
    shoot();
}

```

Com base nesta descrição, será explicitada a maneira que o código deve ser gerado pelo compilador ICE (ICEC) na medida em que for identificando blocos de comandos passíveis de interpretação gerando código

em C++. É importante salientar que, conforme a arquitetura genérica proposta na §7.2, serão geradas extensões das classes que a constituem, inclusive sobrecarregando alguns métodos cuja semântica não pode ser definida genericamente. Começaremos pelo bloco de comandos que representam as ações que o agente pode executar, o bloco *action*.

O agente Soldado tem como ações definidas atirar contra o inimigo, virar-se em duas direções, mover-se e proteger-se. Sabendo que na arquitetura proposta, a classe `GenericAction` conta com alguns atributos e operações, ao criar-se uma classe derivada desta, tem-se uma nova classe que absorverá estes atributos e comportamentos, redefinindo recursos que esta requer.

A partir da descrição do agente serão gerados ao todo, dois arquivos (na linguagem C++) para cada extensão das classes genéricas realizada: um arquivo de cabeçalho e um arquivo com a implementação propriamente dita.

7.5.2 Ações

O construtor da classe `GenericAction` citado logo abaixo, apresenta três parâmetros distintos: um inteiro que representará o identificador do agente, um string que receberá o nome, e um booleano que acionará a reconsideração por parte do agente.

```
GenericAction(int,string,bool);
```

Consideremos a primeira ação descrita para o agente Soldado, a `action shoot()`. Um arquivo de cabeçalho “ShootAction.h” seria gerado com a inclusão das diretivas propostas pela classe `GenericAction` componente da arquitetura ICE, onde uma classe derivada denominada, por exemplo `ShootAction`, herdaria as especificações da classe base em questão. Na descrição da ação em IADL não se tem menção da necessidade de novos parâmetros para sua inicialização, então veja como ficaria a implementação de cada um destes arquivos, começando pelo arquivo de cabeçalho, que não constará de nenhum atributo:

```
// ShootAction.h
```



```
#include "GenericAction.h"

class ShootAction : public GenericAction
{
public:
    ShootAction(int, string, bool);
};
```

O arquivo de programa para esta sub-classe com extensão “cpp” terá simplesmente a inclusão do arquivo de cabeçalho criado e a implementação do construtor para este tipo de objetos, que neste caso específico não terá bloco de comandos, pois não declara nenhum atributo na especificação desta classe.

```
#include "ShootAction.h"

ShootAction::ShootAction(int id, string name, bool recon) :
GenericAction(id, name, recon)
{
}
```

Em se tratando de ações que requeiram passagem por parâmetro de alguma outra variável, tem-se o acréscimo deste novo parâmetro no construtor da ação. Considerando o exemplo do agente Soldado, tem-se a ação move, que requer a passagem por parâmetro do valor do deslocamento que se quer dar na posição do agente. Veja no exemplo de código gerado abaixo:

```
//MoveAction.h
#include "GenericAction.h"

class MoveAction : public GenericAction
{
public:
    MoveAction(int, string, bool, int);
    int size;
};
//MoveAction.cpp
#include "MoveAction.h"
MoveAction::MoveAction(int id, string name, bool recon, int auxsize)
:GenericAction(id, name, recon)
{
    size = auxsize;
}
```

Como se pode notar, o parâmetro `auxsize` foi incluído na implementação do construtor desta ação sendo atribuído à variável `size` (esta é um atributo da classe `MoveAction` declarada no arquivo de cabeçalho gerado pela ação `move`).

7.5.3 Reações

Passemos agora para os blocos de comandos da IADL que representam as reações do agente, os blocos *reaction*, pois devido à ordem de raciocínio imposta pela arquitetura ICE, somente com a implementação dos arquivos relativos às ações já se pode compor o componente reativo que o agente comporta.

Seguindo o exemplo do agente Soldado em IADL, seguem as três percepções que o sensor do agente seria capaz de notar: virar à esquerda, virar à direita e atirar à frente:

```
reaction turnLeft if noiseLeft
    turn(-1);
reaction turnRight if noiseRight
    turn(1);
reaction shootFront if noiseFront
    shoot();
```

Para estas reações ocorrerem espera-se que determinadas condições sejam satisfeitas. Para o agente Soldado existem três condições para que suas reações imediatas sejam acionadas conforme sua especificação.

Considerando que o componente `GenericReflexes` da arquitetura tem por finalidade retirar estas percepções do Sensor do agente e processá-las de forma a gerar ações compatíveis. Cada agente criado terá uma extensão deste componente que possibilita a sobrecarga do método de avaliação destas entradas, que sempre será específico para cada agente. Veja o código gerado para o nosso exemplo:

```
//SoldadoReflexes.h
#include "GenericAction.h"
#include "GenericReflexes.h"
#include "TurnAction.h"
#include "ShootAction.h"

class SoldadoReflexes : public GenericReflexes
{
public:
    SoldadoReflexes(Sensor*, Actuator*, int);
    void evaluateInput(GenericAction*);
};
```

A geração deste cabeçalho leva em conta as ações pertinentes à reatividade do agente e os componentes genéricos envolvidos neste processo. O agente Soldado terá o seu componente de reflexos específico, recebendo indicações para o `Sensor` e o `Actuador`.

```
// SoldadoReflexes.cpp
#include "SoldadoReflexes.h"

SoldadoReflexes::SoldadoReflexes(Sensor* sen, Actuador* act, int
id):GenericReflexes(sen, act, id)
{
}

void SoldadoReflexes::evaluateInput (GenericAction *action)
{
    string name = action->getName();
    GenericAction *newAction;
    if(name == "noiseLeft")
        newAction = new TurnAction(iIdAgent,"turn",false,-1);
    else if(name == "noiseRight")
        newAction = new TurnAction(iIdAgent,"turn",false,1);
    else if(name == "noiseFront")
        newAction = new ShootAction(iIdAgent,"shoot",false);
    actuator->setOutputAndUpdateBeliefs(newAction);
}
```

Para o método `evaluateInput` a ação é passada por parâmetro na forma de percepção, representada no exemplo por `noiseLeft`, para o agente reverter em ação no ambiente ou em seu estado interno. Seguindo a descrição do agente Soldado, esta percepção faria com que ele acionasse a ação `Turn`. Neste ponto do código, a estrutura da ação é criada para posteriormente ser adicionada ao `Actuator`.

Neste ponto, já identificamos as estruturas de código necessárias para criação de um agente puramente reativo. Mas como citado anteriormente, a arquitetura proposta pelo grupo tem natureza híbrida, contando com outros componentes que a torna passível de deliberação: *beliefs*, *objectives* e *planner*.

7.5.4 Crenças, objetivos e planos

Para melhor explicar a geração de código para estes novos componentes, utilizaremos um segundo exemplo de descrição de agentes utilizando a IADL, um agente Formiga, capaz de visualizar alimento,

predadores e outras formigas, seguindo objetivos como procurar alimento, ajudar outras formigas e fugir de predadores quando ameaçada.

```

agent Formiga
{
  belief:
    int fome = 0;
    int posicao = 0;
    boolean amiga = false;
    boolean predador = false;
    boolean comida = false;
  action:
  anda(int passo)
  {
    posicao += passo;
  }
  fuga(int passoLargo)
  {
    posicao -= passoLargo;
    fome++;
  }

  come()
  {
    fome--;
    comida=false;
  }

  alerta()
  {
    predador=true;
  }

  protegida()
  {
    predador=false;
  }

  ajuda()
  {
    fome++;
    amiga=false;
  }

  acheiAmiga()
  {
    amiga=true;
  }

  acheiComida()
  {
    comida=true;
  }

  objective:
  procurarComida()
    pre(comida==true and fome>5)
    pos()

```

```

    ajudarFormiga()
        pre(amiga==true)
        pos()

    fugirPredador()
        pre(predador==true)
        pos()

plan faminta(procurarComida)
if(true)
priority 0
{
    anda(5);
    come();
}

plan ajudaAmiga(ajudarFormiga)
if(true)
priority 0
{
    anda(2);
    ajuda();
}

plan fuga(fugirPredador)
if(true)
priority 0
{
    fuga(10);
    protegida();
}

reaction perigo if vePredador
    alerta();

reaction oba if veComida
    acheiComida();

reaction oiAmiga if veAmiga
    acheiAmiga();
}

```

Um agente sempre terá conhecimento de seu mundo interno, e esse conhecimento será representado na forma de crenças para contemplar o modelo BDI estudado. Assim como o componente dos reflexos foi estendido para o agente Soldado, a classe `GenericBeliefs` presente na arquitetura ICE deverá ser estendida para cada agente criado, pois representará as crenças de cada agente especificamente. Para o novo exemplo, onde um agente Formiga tem em sua base de conhecimento atributos como sua quantidade de fome, sua posição, *flags* que representam a visualização de outras formigas no

ambiente, de predadores ou mesmo de alimento, vejamos os arquivos e as estruturas criadas:

```
//FormigaBeliefs.h
#include "GenericBeliefs.h"
#include "GenericAction.h"
#include "AndaAction.h"
#include "FugaAction.h"
#include "ComeAction.h"
#include "AlertaAction.h"
#include "ProtegidaAction.h"
#include "AjudaAction.h"
#include "AcheiAmigaAction.h"
#include "AcheiComidaAction.h"

class FormigaBeliefs : public GenericBeliefs
{
public:
    FormigaBeliefs();
    void update(GenericAction*);
    int fome;
    int posicao;
    bool amiga;
    bool predador;
    bool comida;
private:
    void come();
    void alerta();
    void protegida();
    void ajuda();
    void acheiAmiga();
    void acheiComida();
    void anda(int passo);
    void fuga(int passoLargo);
};
```

O arquivo de cabeçalho conterà referências aos componentes genéricos dos quais as crenças da formiga irão derivar, além de incluir todas as classes geradas para representar as ações da Formiga, pois o agente terá presente não só seus atributos em termos de estado interno, como também as ações que consegue realizar no ambiente e em si mesmo, estes últimos na forma de métodos.

No arquivo de implementação propriamente dita, o construtor da classe `FormigaBeliefs` deverá inicializar todos os atributos inerentes ao agente em questão e, ainda, sobrescrever o método `update` provido pela classe base.

```
//FormigaBeliefs.cpp
#include "FormigaBeliefs.h"

FormigaBeliefs::FormigaBeliefs()
```

```

{
    fome=0;
    posicao=0;
    amiga = false;
    predador = false;
    comida = false;
}

void FormigaBeliefs::update(GenericAction *action)
{
    string name = action->getName();

    if(name == "anda")
    {
        AndaAction *a1 = static_cast<AndaAction*>(action);
        anda(a1->passo);
    }
    else if(name == "fuga")
    {
        FugaAction *a2 = static_cast<FugaAction*>(action);
        fuga(a2->passoLargo);
    }
    else if(name == "come")
    {
        ComeAction *a3 = static_cast<ComeAction*>(action);
        come();
    }
    else if(name == "alerta")
    {
        AlertaAction *a4 = static_cast<AlertaAction*>(action);
        alerta();
    }

    else if(name == "protegida")
    {
        ProtegidaAction *a5 =
static_cast<ProtegidaAction*>(action);
        protegida();
    }
    else if(name == "ajuda")
    {
        AjudaAction *a6 = static_cast<AjudaAction*>(action);
        ajuda();
    }
    else if(name == "acheiAmiga")
    {
        AcheiAmigaAction *a7 = static_cast< AcheiAmigaAction
*>(action);
        acheiAmiga();
    }
    else if(name == "acheiComida")
    {
        AcheiComidaAction *a8 =
static_cast<AcheiComidaAction*>(action);
        acheiComida();
    }
}

void FormigaBeliefs::anda(int passo)

```

```

{
    posicao+=passo;
}

void FormigaBeliefs::fuga(int passoLargo)
{
    posicao-=passoLargo;
    fome++;
}

void FormigaBeliefs::come()
{
    fome--;
    comida=false;
}

void FormigaBeliefs::alerta()
{
    predador = true;
}

void FormigaBeliefs::protegida()
{
    predador = false;
}

void FormigaBeliefs::ajuda()
{
    fome++;
    amiga = false;
}

void FormigaBeliefs::acheiAmiga()
{
    amiga = true;
}

void FormigaBeliefs::acheiComida()
{
    comida = true;
}

```

O método `update` identificará cada ação que receber como parâmetro, através do nome da mesma. Neste momento, o método correspondente à execução daquela ação em particular pode ser invocado com ou sem parâmetros, dependendo da descrição dada para o agente em IADL.

Da mesma forma, voltando ao exemplo do agente Soldado, teremos o mesmo princípio de geração de código relativo às crenças do mesmo. O arquivo de implementação da classe `SoldadoBeliefs` também deverá inicializar todos os seus atributos e sobrecarregar o método `update` provido pela classe base, a classe `GenericBeliefs`.


```

//SoldadoBeliefs.cpp
#include "SoldadoBeliefs.h"
SoldadoBeliefs::SoldadoBeliefs()
{
    direction=0;
    ammo=5;
    cover = false;
    distance = 0;
    lastTurn = -1;
}

void SoldadoBeliefs::update(GenericAction *action)
{
    string name = action->getName();

    if(name == "shoot")
    {
        ShootAction *sa = static_cast<ShootAction*>(action);
        shoot();
    }
    else if(name == "turn")
    {
        TurnAction *ta = static_cast<TurnAction*>(action);
        turn(ta->dir);
    }
    else if(name == "move")
    {
        MoveAction *ma = static_cast<MoveAction*>(action);
        move(ma->size);
    }
    else if(name == "takeCover")
    {
        TakeCoverAction *tca =
static_cast<TakeCoverAction*>(action);
        takeCover();
    }
}

```

Além disso, os procedimentos referentes à execução das ações são aqui também implementados, pois fazem parte do conhecimento que o agente possui de si mesmo, do que ele é capaz de realizar, exatamente como vimos no exemplo do agente Formiga.

```

//Continuação do código de implementação SoldadoBeliefs.cpp
void SoldadoBeliefs::shoot()
{
    ammo--;
}

void SoldadoBeliefs::turn(int dir)
{
    direction += dir;

    lastTurn = dir;
}

```

```

void SoldadoBeliefs::move(int size)
{
    distance += size;
}

void SoldadoBeliefs::takeCover()
{
    cover = true;
}

```

Após definir a base de conhecimento do agente com seus atributos e ações que realiza, é necessário definir os objetivos do agente de forma que ele possa decidir suas ações em termos de seus desejos. Para prover este tipo de funcionalidade, duas classes genéricas foram implementadas compondo não só estruturas que armazenam cada objetivo em particular, como também uma estrutura capaz de armazenar todos os objetivos que o agente é capaz de buscar, as classes `GenericObjective` e `GenericObjectives`.

O agente Formiga proposto tem três objetivos distintos: procurar alimento, ajudar outra formiga e fugir de predadores. Para cada objetivo declarado na descrição do mesmo, tem-se uma extensão da classe `GenericObjective`, para que todos disponham de métodos de avaliação dos mesmos em termos de possibilidade de execução e cumprimento do mesmo.

Um dos objetivos primordiais do agente Formiga, segundo a descrição, seria buscar alimento no ambiente. Denominado `ProcurarComidaObjective`, é sobre este objetivo que exemplificaremos a geração de código, começando pela declaração do cabeçalho:

```

//ProcurarComidaObjective.h
#include "GenericObjective.h"
#include "FormigaBeliefs.h"

class ProcurarComidaObjective : public GenericObjective
{
public:
    ProcurarComidaObjective(string, FormigaBeliefs*);
    inline bool isPossible();
    inline bool isComplete();
};

```

Há necessidade de inclusão das crenças do agente neste cabeçalho, pois a verificação realizada em tempo de execução sobre cada objetivo depende do seu estado interno naquele dado momento.

O agente estará sempre buscando algum objetivo a cumprir, e sua decisão dependerá do estado de cada um deles. Nota-se que cada objetivo em particular terá dois estados possíveis durante o período de deliberação do agente: ou o objetivo será possível, ou ele já estará completo, ou seja, não necessita mais ser atingido.

```
//ProcurarComidaObjective.cpp
#include "ProcurarComidaObjective.h"

ProcurarComidaObjective::
ProcurarComidaObjective(string name,FormigaBeliefs*
bel):GenericObjective(name,bel)
{
}

bool ProcurarComidaObjective::isPossible()
{
    FormigaBeliefs *b1 = static_cast<FormigaBeliefs*>(beliefs);
    if((b1->comida == true)&&(b1->fome>5))
        return true;
    else
        return false;
}
bool ProcurarComidaObjective::isComplete()
{
    FormigaBeliefs *b1 = static_cast<FormigaBeliefs*>(beliefs);
    if(false)
        return true;
    else
        return false;
}
```

A possibilidade de eleição de um objetivo vai depender das pré-condições impostas na sua modelagem. No exemplo acima, temos uma estrutura de decisão que retornaria verdadeiro caso as condições preestabelecidas (a formiga ter visto alimento além de estar com fome) sejam satisfeitas, ou caso contrário, retornaria falso não respeitando as condições impostas.

Tendo todos os objetivos sido criados em termos de geração dos arquivos acima descritos, há a necessidade de reuni-los em uma estrutura que facilite a seleção dos mesmos. Esta estrutura é representada pela classe `GenericObjectives`, cuja finalidade é reunir todos os objetivos do agente em uma única estrutura de dados, neste caso, uma lista de objetivos. Para o agente Formiga, teríamos:

```

//FormigaObjectives.h
#include "GenericObjectives.h"
#include "ProcurarComidaObjective.h"
#include "AjudarFormigaObjective.h"
#include "FugirPredadorObjective.h"

class FormigaObjectives : public GenericObjectives
{
public:
    FormigaObjectives(FormigaBeliefs *);
};

```

Quando estendida, a classe derivada possuirá objetivos ordenados por prioridade (dada pela ordem em que aparecem na descrição do agente em IADL), provendo recursos de análise dos mesmos em situações específicas, podendo descartar e eleger cada um conforme a necessidade e prioridade do momento. Na implementação do construtor desta classe cada objetivo seria instanciado e enfileirado, como mostrado logo abaixo:

```

//FormigaObjectives.cpp
#include "FormigaObjectives.h"

FormigaObjectives::FormigaObjectives(FormigaBeliefs
*beliefs):GenericObjectives()
{
    ProcurarComidaObjective *o1 = new
ProcurarComidaObjective("procurarComida",beliefs);
    AjudarFormigaObjective *o2 = new
AjudarFormigaObjective("ajudarFormiga",beliefs);
    FugirPredadorObjective *o3 = new
FugirPredadorObjective("fugirPredador", beliefs);
    allObjectives.push_back(o3);
    allObjectives.push_back(o2);
    allObjectives.push_back(o1);
}

```

Logicamente, a partir do momento que se têm objetivos a alcançar, necessita-se de meios para tanto, que na arquitetura ICE foram chamados de “planos do agente” abstraídos pela classe `GenericPlan`. Cada plano será representado por uma classe derivada desta classe genérica, contendo as ações necessárias para cumprir o objetivo ao qual o plano se destina, e também terá dependência das crenças do agente no momento de ser selecionado e executado. Segundo a descrição do agente Formiga, ele teria apenas três planos, um para cada um dos seus objetivos. Vejamos como exemplo o plano `Faminta` referente ao objetivo de procurar alimento:

```

//FamintaPlan.h
#include "GenericPlan.h"
#include "FormigaBeliefs.h"
#include "Actuator.h"
#include "AndaAction.h"
#include "ComeAction.h"

class FamintaPlan : public GenericPlan
{
public:
    FamintaPlan(FormigaBeliefs*,Actuator*,int);
    inline bool isPossible();
};

//FamintaPlan.cpp
#include "FamintaPlan.h"

FamintaPlan::
    FamintaPlan(FormigaBeliefs *fbeliefs,Actuator *act,int
id):GenericPlan(fbeliefs,act)
{
    AndaAction *a1 = new AndaAction(id,"anda",false,5);
    ComeAction *a2 = new ComeAction(id,"come",false);
    actionVector.push_back(a1);
    actionVector.push_back(a2);
    sObjective = "procurarComida";
}

bool FamintaPlan::isPossible()
{
    FormigaBeliefs *b1 = static_cast<FormigaBeliefs*>(beliefs);
    if(true) return true;
    else return false;
}

```

Os planos terão, além de referência para as crenças do agente, também para o `Actuator`, onde poderão colocar as ações que solicitam para execução. O plano acima tem duas ações para executar `anda` e `come`, e estas são criadas e colocadas ao fim da fila do atuador para serem executadas cumprindo o objetivo `procurarComida`.

Assim como cada agente tem uma lista de objetivos, ele terá também uma lista de planos para alcançá-los. Esta lista de planos será constituinte do planejador do agente, que será uma derivação da classe `GenericPlanner` provida pela arquitetura ICE. O planejador do agente dependerá desta lista de planos bem como das crenças do mesmo, tendo ligação com o `Actuator`. Exemplificando, veremos o planejador do agente Formiga:

```

//FormigaPlanner.h

```

```

#include "GenericPlanner.h"
#include "FormigaBeliefs.h"
#include "Actuator.h"
#include "FamintaPlan.h"
#include "AjudaAmigaPlan.h"
#include "FogePlan.h"

class FormigaPlanner : public GenericPlanner
{
public:
    FormigaPlanner(FormigaBeliefs*,Actuator*,int);
};

//FormigaPlanner.cpp
#include "FormigaPlanner.h"

FormigaPlanner::
    FormigaPlanner(FormigaBeliefs *beliefs,Actuator *act,int
id):GenericPlanner()
    {
        FamintaPlan *p1 = new FamintaPlan(beliefs,act,id);
        AjudaAmigaPlan *p2 = new AjudaAmigaPlan(beliefs,act,id);
        FogePlan *p3 = new FogePlan(beliefs,act,id);
        allPlans.push_back(p1);
        allPlans.push_back(p2);
        allPlans.push_back(p3);
    }

```

Até o momento, descrevemos como cada um dos componentes do agente que criarmos usando a IADL deveria ser implementado, ou ainda, como o código referente à descrição do agente poderia ser gerado pelo compilador da linguagem. Porém, agora é necessário reforçar a abstração de agente como uma entidade capaz de perceber e atuar no ambiente, reativamente ou deliberativamente, estendendo-se a classe `GenericAgent` provida pela arquitetura ICE. Para exemplificar, mostraremos os dois agentes abordados neste capítulo, o agente Soldado e o agente Formiga. Eis as interfaces para as classes em que estão representados:

```

//SoldadoAgent.h
#include "GenericAgent.h"
#include "SoldadoBeliefs.h"
#include "SoldadoReflexes.h"
#include "SoldadoPlanner.h"
#include "SoldadoObjectives.h"

class SoldadoAgent : public
GenericAgent
{
public:
    SoldadoAgent(int,string);
}

//FormigaAgent.h
#include "GenericAgent.h"
#include "FormigaBeliefs.h"
#include "FormigaReflexes.h"
#include "FormigaPlanner.h"
#include "FormigaObjectives.h"

class FormigaAgent : public
GenericAgent
{
public:
    FormigaAgent(int,string);
}

```

```
};
```

A implementação de suas classes segue logo abaixo:

```
//SoldadoAgent.cpp
#include "SoldadoAgent.h"
SoldadoAgent::SoldadoAgent(int i,string s):GenericAgent(i,s)
{
    beliefs = new SoldadoBeliefs();
    actuator = new Actuator(beliefs);
    sensor = new Sensor();
    reflexes = new SoldadoReflexes(sensor,actuator,iID);
    objectives = new
SoldadoObjectives(static_cast<SoldadoBeliefs*>(beliefs));
    planner = new
SoldadoPlanner(static_cast<SoldadoBeliefs*>(beliefs),actuator,iID);
}

//FormigaAgent.cpp
#include "FormigaAgent.h"
FormigaAgent::FormigaAgent(int i,string s):GenericAgent(i,s)
{
    beliefs = new FormigaBeliefs();
    actuator = new Actuator(beliefs);
    sensor = new Sensor();
    reflexes = new FormigaReflexes(sensor,actuator,iID);
    objectives = new
FormigaObjectives(static_cast<FormigaBeliefs*>(beliefs));
    planner = new
FormigaPlanner(static_cast<FormigaBeliefs*>(beliefs),actuator,iID);
}
```

8 ATIVIDADES

8.1 Atividades e Responsáveis no TC1

As principais atividades desenvolvidas no Trabalho de Conclusão I podem ser divididas em:

8.1.1 Revisão Bibliográfica

A revisão bibliográfica visou embasar teoricamente o projeto a ser desenvolvido e coletar material para a preparação do volume final de TC1. A revisão bibliográfica pode ser dividida em três áreas relativamente distintas, e portando pôde ser delegada a diferentes componentes do grupo, como segue:

- Pesquisa sobre arquiteturas de agentes: consiste em analisar arquiteturas e modelos de agência pré-existentes, analisando seus pontos fortes e fracos bem como a viabilidade de inclusão do modelo escolhido como base na arquitetura geral do jogo. Esta tarefa foi de responsabilidade de Thais Webber
- Pesquisa sobre arquiteturas de IA em jogos: consiste em analisar as principais arquiteturas de IA atualmente usadas na indústria, novamente verificando pontos fortes e fracos e sua viabilidade de inclusão no projeto a ser desenvolvido. Esta tarefa foi de responsabilidade de Paulo Schneider.
- Pesquisa sobre projeto arquitetural de software: consiste em pesquisar técnicas de projeto e implementação de software adequadas à criação do *kernel*. Este projeto levou em consideração a integração do paradigma orientado a agentes e contemplar seus objetivos de modularização. Além de definir a arquitetura, foi definida uma linguagem orientada a agentes adequada ao desenvolvimento de jogos. Ao final desta pesquisa foi projetado o

produto final do Trabalho de Conclusão II. Esta tarefa foi de responsabilidade de Felipe Meneguzzi.

Apesar de terem sido enumerados responsáveis pelas diversas atividades, todos os componentes do grupo adquiriram uma visão dos conhecimentos exercitados no decorrer do trabalho, de modo a diversificar as opiniões em cada assunto e promover a interação do grupo.

8.1.2 Projeto da Arquitetura

Esta tarefa visou a criação de um projeto do sistema proposto para a implementação a ser realizada como parte do Trabalho de Conclusão II. Como foi objetivo e interesse de todos os componentes do grupo o aprimoramento dos conhecimentos sobre projeto e implementação de software, os componentes desta tarefa não foram delegados individualmente, e sim executados pelo grupo como um todo. Esta tarefa pode ser subdividida em:

- Funcionalidades: consiste na análise de outras implementações semelhantes e na definição de um conjunto de funcionalidades a serem contempladas no projeto e posterior implementação.
- Interface: consiste em estabelecer um padrão de interface de entrada (i.e. linguagem orientada a agentes) para a definição de comportamento. Também consiste em estabelecer uma interface (API) a ser utilizada pelo ambiente externo dentro do jogo que abrigará o ICE.
- Desenho da Arquitetura: consiste na definição diagramada em alto nível da arquitetura a ser projetada posteriormente.
- Projeto: consiste na definição do projeto físico a ser implementado durante o TC2.

8.1.3 Redação do Volume

Esta tarefa consistiu na reunião do material textual produzido ao longo do semestre para a elaboração do volume final de TC1. Este volume

contemplou os assuntos estudados na fase revisão bibliográfica bem como o projeto elaborado na fase de projeto de arquitetura. A responsabilidade de redigir o embasamento teórico foi da pessoa correspondente ao estudado na fase de revisão bibliográfica, e a responsabilidade de documentação do projeto foi de responsabilidade do grupo todo.

8.2 Atividades e Responsáveis no TC2

As principais atividades a desenvolvidas no Trabalho de Conclusão 2 podem ser divididas em três grandes grupos:

8.2.1 Implementação do Projeto

Este grupo de tarefas está relacionado com a implementação real do sistema apresentado no TC1, e testes relativos ao ajuste técnico da IADL, e inclui as seguintes tarefas:

- Conclusão da modelagem física do sistema a ser implementado: devido a restrições no tempo disponível, a conclusão da modelagem física teve de ser postergada para o período do TC2. Esta modelagem consistiu da modelagem de objetos para a arquitetura ICE, o compilador/interpretador¹ da IADL e o jogo de demonstração a ser implementado. Esta tarefa foi concluída nas três primeiras semanas de agosto e foi de responsabilidade de Felipe Meneguzzi e Paulo Schneider.
- Implementação do *kernel* ICE: depois de concluída a modelagem da arquitetura ICE, teve início a implementação do cerne da arquitetura ICE, possibilitando a criação do jogo de demonstração. Um protótipo que permita testes deverá foi concluído no início de outubro e foi de responsabilidade de Paulo Schneider.
- Implementação do compilador/interpretador da IADL: uma vez concluída a modelagem física dos agentes da ICE, a linguagem

¹ A modelagem também definiu se a linguagem seria compilada ou interpretada

definida no TC1 passou por testes de praticidade e viabilidade técnica no início do TC2. Após a linguagem ter recebido os ajustes necessários um compilador da mesma foi ser implementado. Esta tarefa foi concluída na primeira semana de setembro e foi de responsabilidade de Felipe Meneguzzi.

- Implementação dos jogos de demonstração de conceito do ICE: após ter sido concluída a modelagem do *kernel*, foi possível iniciar a implementação de jogos para serem encaixados no mesmo. Este jogos não apresentaram tecnologias avançadas de visualização, apenas demonstraram o conceito da arquitetura ICE em funcionamento. Esta tarefa foi concluída na segunda semana de outubro e foi de responsabilidade de Paulo Schneider e Thais Webber.

8.2.2 Revisão Bibliográfica

Este grupo de tarefas visou acrescentar a base teórica para os aspectos utilizados na implementação dos componentes do ICE, além de enriquecer os tópicos abordados no TC1. Este grupo de tarefas foi concluído na segunda semana de novembro e está dividido em:

- Pesquisa sobre tecnologias de implementação de jogos: consiste em pesquisar as mais recentes tecnologias utilizadas na implementação de jogos de modo a embasar a implementação do jogo de demonstração. Esta tarefa foi de responsabilidade de Paulo Schneider.
- Pesquisa sobre engenharia de software: consiste em pesquisar os requisitos necessários para a modelagem e implementação do sistema proposto de modo a facilitar a integração com o jogo de demonstração. Esta tarefa foi de responsabilidade de Felipe Meneguzzi

8.2.3 Redação do Volume Final de TC2

Este conjunto de tarefas consistiu da integração da modelagem do sistema com o material colhido na revisão bibliográfica e as conclusões obtidas da implementação do sistema proposto. Este grupo de tarefas foi concluído na metade de novembro. As seguintes tarefas compõem este grupo:

- Redação dos resultados obtidos na fase de implementação: consistiu em sistematizar as conclusões obtidas na implementação do sistema proposto e incluí-las no volume final. Esta atividade foi de responsabilidade de todos os componentes do grupo.
- Inclusão do material colhido na revisão bibliográfica: esta tarefa consistiu em organizar os conteúdos obtidos na revisão bibliográfica no volume final do trabalho. Esta atividade foi de responsabilidade de todos os componentes do grupo.
- Elaboração do manual do usuário do sistema ICE: esta atividade consistiu em elaborar um guia de referência que facilite a utilização do sistema implementado. Esta atividade foi de responsabilidade de Thais Webber.
- Redação final do volume: esta tarefa consistiu na reunião e organização de todos os materiais colhidos e redigidos ao longo do trabalho para a elaboração deste documento. Esta atividade foi de responsabilidade de Felipe Meneguzzi.

8.2.4 Apresentação do Trabalho de Conclusão II

Esta tarefa consistirá na elaboração da apresentação deste trabalho para a banca avaliadora. Consistirá na elaboração de slides e uma apresentação expositiva descrevendo a experiência e os resultados da realização do Trabalho de Conclusão. Esta tarefa será de responsabilidade de todos os componentes do grupo.

¹ A modelagem também irá definir se a linguagem será compilada ou interpretada

8.3 Dependências entre as Atividades

As atividades de revisão bibliográfica podem ser executadas concorrentemente, visto que são assuntos relativamente disjuntos. A redação do TC2 relativa aos tópicos de implementação e resultados obtidos da mesma só pôde ocorrer após a conclusão pelo menos parcial da fase de implementação.

Quanto à implementação da arquitetura ICE, esta só pôde ocorrer após a modelagem física ter sido devidamente concluída. Após o início da fase de implementação, o compilador/interpretador teve de ser o primeiro módulo a ser concluído, de modo a possibilitar testes de conceito sobre a arquitetura ICE em implementação.

8.4 Recursos Necessários

O produto final do TC2 não requisitará de recursos extraordinários, uma vez que consistirá apenas de um programa. Logo os recursos necessários consistirão de:

- Um PC, utilizando Sistema Operacional Windows 9x/2000;
- Compilador C++, preferencialmente o Microsoft Visual C++;
- Programas Lex e Yacc;
- DirectX 7 instalado.

9 CONCLUSÕES

Durante o curso das pesquisas do grupo para a realização deste trabalho, diversas conclusões foram obtidas:

A grande envergadura que os projetos de jogos de computador em nível comercial possuem inviabilizam o desenvolvimento de um exemplar completo por um grupo pequeno de pessoas como o grupo que realizou este trabalho. Desta forma o grupo teve de eliminar por completo, por exemplo, o objetivo de se projetar uma apresentação gráfica complexa, sendo esta por si só o argumento de um novo trabalho de conclusão.

Além das dificuldades de se criar um jogo completo, o grupo também chegou à conclusão de que as diversas facetas de sistemas de IA possuem podem apenas ser parcamente exploradas no período da disciplina de trabalho de conclusão, sendo que diversas simplificações na idéia original tiveram de ser adotadas para a viabilização do projeto especificado neste trabalho. Por exemplo, no início do TC foi cogitada a implementação de um planejador completo para a arquitetura, porém esta meta teve de ser descartada visto que não seria possível alcançá-la simultaneamente com as demais.

A mais importante conclusão a que se chegou é que a abstração de agentes aparenta ser apropriada para a definição de comportamentos em jogos de computador.

Sobre a execução do projeto definido no TC1, observou-se que a modelagem e implementação do Kernel em si não foi o item mais trabalhoso. Os diversos componentes de suporte a arquitetura, tais como o compilador e o *Game Engine* demandaram mais esforço de implementação, sendo que de seu correto funcionamento dependia a correta implementação e avaliação do Kernel ICE.

9.1 Trabalhos Futuros

Dentre as possibilidades de expansão do ambiente ICE pode-se citar:

- Criação de um interpretador para a IADL: através da interpretação da linguagem serão possíveis o teste e modificação dos agentes em tempo de execução, isto também implica a modelagem de uma classe de agentes dinâmicos, cujo conjunto de atributos pode mudar durante a execução;
- Criação de uma linguagem de definição para funções no Kernel (i.e. *Ether*);
- Inserção de um algoritmo de planejamento no componente planejador: o planejador atualmente depende de planos especificados pela IADL, logo o comportamento dos agentes não pode mudar em tempo de execução. A inserção de planejamento em tempo de execução possibilitaria a mudança de comportamento dos agentes ICE de formas possivelmente não previstas na especificação do agente, capacidade esta bastante interessante em ambientes de jogos;
- Implementação da geração de código para listas de crenças no compilador.

ANEXO I – CRONOGRAMA

MARÇO						
D	S	T	Q	Q	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

- 1 a 25** - Pesquisa de Material (embasamento proposta)
26 - Indicação do Prof. Avaliador
27 a 31 - Redação da Proposta TC1

ABRIL						
D	S	T	Q	Q	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

- 1 a 3** - Cont. Redação da Proposta TC1
4 - Entrega Proposta TC1
16 a 26 - Revisão da Proposta de TC1 (caso indeferida)
5 a 26 - Revisão bibliográfica TC1
27 - Entrega da Proposta alterada (caso indeferida)
28 a 30 - Cont. Revisão bibliográfica TC1

MAIO						
D	S	T	Q	Q	S	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

- 1 a 31** - Projeto da Arquitetura (vol. Final TC1)

JUNHO						
D	S	T	Q	Q	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

- 1 a 9** - Cont. Projeto da Arquitetura
10 a 24 - Redação e fechamento vol. Final TC1
25 - Entrega do volume final de TC1
26 a 30 - Início implementação projeto TC1

JULHO						
D	S	T	Q	Q	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

- 1 a 11** - Sequência implementação TC1
12 a 16 - Estudo do cronograma atividades 01-2

AGOSTO						
D	S	T	Q	Q	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

- 1 a 18** - Projeto e modelagem física dos componentes do sistema
19 a 31 - Implementação do compilador/interpretador da IADL.
 - Início da implementação do *kernel* da arquitetura ICE

SETEMBRO						
D	S	T	Q	Q	S	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

- 1 a 7** - Implementação do compilador/interpretador da IADL
1 a 30 - Implementação do *kernel* da arquitetura ICE
23 a 30 - Implementação do jogo de demonstração de conceito

OUTUBRO						
D	S	T	Q	Q	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

- 1 a 13** - Implementação do jogo de demonstração de conceito
14 a 31 - Revisão bibliográfica TC1

NOVEMBRO						
D	S	T	Q	Q	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

- 1 a 20** - Redação e fechamento do vol. Final TC2

DEZEMBRO						
D	S	T	Q	Q	S	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

ANEXO II – IADL GRAMÁTICA COMPLETA

Formato do Arquivo

```

<ice_file> → <agent_file> | <type_file>
<agent_file> →
    <import_section>
    <type_declaration_section>
    <agent> |
    <type_declaration_section>
    <agent> |
    <import_section>
    <agent> |
    <agent>
<type_file> → <import_section> <type_declarations> |
    <type_declarations>
<import_section> → <imports> | ε
<type_declaration_section> →
    <type_declaration_section><type_declaration>; |
    <type_declaration>; | ε
<imports> → <import> <imports> | <import>; | ε
<import> → import <filename>;

```

Tipos

```

<simple_type> → int | float | bool | string
<type> → <simple_type> | IDENTIFIER
<type_declaration> → <composite> | <list>
<composite> → composite IDENTIFIER
    {<fields>}
<fields> → <field> ; <fields> |
    <field> ;
<field> → <simple_type> IDENTIFIER
<composite_reference> → IDENTIFIER.IDENTIFIER
<list> → IDENTIFIER list of <type>;
<constant> → <mathematical_constant> |
    STRING_CONSTANT |
    BOOL_CONSTANT
<mathematical_constant> → INTEGER_CONSTANT |
    FLOAT_CONSTANT

```

Operações com Tipos

```

<list_statement> → IDENTIFIER.<list_operation> |
    <list_iteration>
<list_iteration> → iterate IDENTIFIER with IDENTIFIER
    {<action_effects>}
<list_operation> → add(IDENTIFIER) | remove()
<list_condition> →
    <type> IDENTIFIER <list_logical_operator> IDENTIFIER
    where (<regular_condition>)
<list_logical_operator> → in | forall

```

```

<expression> → <expression><binary_operator><expression2> |
               <unary_expression> |
               <constant> |
               <belief_reference> |
               (<expression>)
<unary_expression> → <unary_operator><belief_reference> |
                    <unary_operator><constant>
<expression2> → <constant> |
               <belief_reference> |
               (<expression>)

<unary_operator> → ++ | -- | + | -
<binary_operator> → <logical_operator> |
                  <mathematical_operator> |
                  <comparison_operator>
<logical_operator> → and | or
<mathematical_operator> → + | - | * | /
<comparison_operator> → > | >= | < | <= | ==
<assignment_operator> → = | += | -= | *= | /=

<conditions> → <list_condition> | <regular_condition>
<regular_condition> → <expression>

```

Agente

```

<agent> →
  agent <agent_name>
  {
    <beliefs>
    <actions>
    <objectives>
    <plans>
    <reactions>
  }

```

Crenças

```

<beliefs> → belief: <belief_declaration_list>
<belief_declaration_list> →
  <belief_declaration>; <belief_declaration_list> |
  <belief_declaration>; | ε
<belief_declaration> → <type> <belief_declarator_list>
<belief_declarator_list> →
  <belief_declarator_list>, <belief_declarator> |
  <belief_declarator>
<belief_declarator> → IDENTIFIER = <expression> | IDENTIFIER
<belief_assignment> →
  <belief_reference><assignment_operator><expression>
<belief_reference> → IDENTIFIER | <composite_reference>

```

Ações

```

<actions> → action: <action_declaration_list>
<action_declaration_list> →
  <action_declaration><action_declaration_list> |
  <action_declaration> | ε
<action_declaration> → IDENTIFIER(<parameter_declaration_list>)
  {

```

```

    <action_effects>
  }
<action_effects> → <action_effects><action_effect> | ε
<action_effect> → <belief_assignment>; | <unary_expression>; |
    <list_statement>;
<action_call_list> → <action_call_list><action_call>; |
    <action_call>; | ε
<action_call> → IDENTIFIER(<parameters>)

```

Objetivos

```

<objectives> → objective: <objective_declaration_list>
<objective_declaration_list> →
    <objective_declaration_list><objective_declaration> |
    <objective_declaration> | ε
<objective_declaration> →
    IDENTIFIER(<parameter_declarator_list>)
    pre(<conditions>)
    pos(<conditions>)
<objective_list> → IDENTIFIER | ε

```

Planos

```

<plans> → <plans><plan_declarator> | <plan_declarator> | ε
<plan_declarator> →
    plan IDENTIFIER(<objective_list>)
    if(<conditions>)
    priority INTEGER_CONSTANT
    {
        <action_call_list>
    }

```

Reações

```

<reactions> → <reactions><reaction>; |
    <reaction>; | ε
<reaction > →
    <reaction_modifier> reaction IDENTIFIER
    if (IDENTIFIER)
        <action_call>
<reaction_modifier> → reconsider | ε

```

Auxiliares

```

<parameters> → <parameter_list> | <parameter> | ε
<parameter_list> → <parameter_list>, <parameter>
<parameter> → <expression>
<parameter_declarator_list> →
    <parameter_declarator_list>, <parameter_declarator> |
    <parameter_declarator>
<parameter_declarator> → <type> IDENTIFIER

```

ANEXO III – MANUAL DO ICEC E DA IADL

1. Introdução

Este manual visa explicar o funcionamento e uso da linguagem IADL (*ICE Agent Description Language*) utilizada pelo compilador de agentes ICEC implementado como parte do requisito final da disciplina de Trabalho de Conclusão II do curso de bacharelado em Ciência da Computação da Pontifícia Universidade Católica do Rio Grande do Sul.

Inicialmente o manual procura introduzir o usuário aos conceitos propostos na arquitetura ICE bem como exemplificar o uso da linguagem de descrição de agentes para que o mesmo possa criar e desenvolver o comportamento dos agentes ICE.

A IADL é utilizada pelo ICEC (*ICE Compiler*) para transformar uma descrição de agentes em código C++. As palavras reservadas, tipos, operações com tipos, como expressões, operadores unários e binários, lógicos e matemáticos, e operadores de atribuição estarão listados ao final deste manual para consulta por parte dos usuários, visto que os exemplos abordados nem sempre cobrirão todas as funcionalidades previstas pela linguagem.

Como os personagens criados são baseados na noção de agentes em IA, uma breve introdução ao assunto explicitará o que são, e que princípios seguem dentro da arquitetura proposta.

2. Introdução aos Modelos de Agência

A arquitetura ICE foi modelada para prover a possibilidade de implementação de agentes do tipo híbrido, cujas características podem ser tanto reativas, como deliberativas, dependendo somente do que se espera do agente quando inserido no ambiente do jogo. Por exemplo, personagens inanimados ou que apresentam reações involuntárias a determinadas percepções podem ser modelados baseando-se no modelo de agência reativo, onde se pode

estabelecer um mapeamento direto de uma situação a uma ação específica como veremos a seguir.

Personagens mais elaborados, como por exemplo, um soldado, ou qualquer outra entidade que realize planejamento de suas ações para cumprir seus objetivos, podem ser modelados baseando-se em características BDI (*Belief-Desire-Intention*). Este tipo de comportamento, conhecido também como pró-atividade, norteará o personagem dentro do contexto do jogo, dando-lhe uma base de conhecimento com seus atributos e ações (que chamaremos *beliefs*), e ainda objetivos com planos correspondentes.

Como podemos notar, a IADL permite que estes dois modelos de agência possam ser explorados na modelagem de personagens com características reativas e inteligentes ao mesmo tempo.

2.1. Modelando agentes reativos

A modelagem que se utilizado princípio reativo visa construir personagens que possam decidir suas ações rapidamente em tempo de execução. As decisões são tomadas em tempo real, baseando-se em poucas informações e regras simples que definem a ação em função de uma situação.

Sabendo que a definição de agente pressupõe a existência de sensores e atuadores no agente, o personagem então será dotado desta característica abstrata, e a linguagem IADL permitirá que sejam especificadas as percepções que, sendo captadas pelo sensor, resultam em reações. Nas próximas seções veremos como fazer com que o personagem tenha reflexos imediatos através de percepções, apresentando o comando *reaction* provido pela IADL.

É importante lembrar que os agentes reativos são deficientes em relação à realização de tarefas que envolvam diversos passos, pois as construções reativas da linguagem não permitem a elaboração prévia de uma seqüência de ações, somente o mapeamento de uma percepção em uma ação correspondente. Para adicionar um comportamento deliberativo e dirigido a determinado objetivo de um personagem, utilizaremos a noção de BDI.

2.2. Acrescentando características BDI aos agentes

Já citamos anteriormente a importância de se trabalhar com reatividade no comportamento dos personagens. Porém muitas vezes, deseja-se que o personagem possa decidir suas ações sem que isto seja um resultado direto da função de mapeamento. Nesta seção serão explicadas as construções da IADL relativas ao comportamento pró-ativo apresentado pelos agentes BDI.

Na noção de BDI o agente é descrito em termos de estados mentais, dentre eles: uma representação do mundo e de si mesmo (na forma de atributos), ações que poderá realizar e predisposições a determinadas realizações as quais chamaremos de objetivos.

Estes agentes são capazes de realizar deliberação acerca de abstrações como crenças e desejos, podendo coordenar suas ações e selecionar planos conforme seus objetivos.

3. A Linguagem de Descrição de Agentes ICE (IADL)

A IADL linguagem possui um alto nível de abstração quando comparada às linguagens de programação tradicionais. Logo a seguir veremos como utilizar cada comando disponível na linguagem de forma que se possa descrever o comportamento de um personagem.

O primeiro passo para descrever um agente, é definir quais os conhecimentos que ele terá de si mesmo e do mundo, quais as ações que poderá realizar, se ele terá objetivos a cumprir, quais planos ele será capaz de escolher e executar. Então comecemos por uma descrição textual do agente que queremos modelar.

Suponhamos, como exemplo, que queremos modelar um personagem que tenha reações e atitudes semelhantes às de um soldado. Então, o mesmo terá em sua base de crenças:

- a direção em que olha;
- a que distância está do inimigo;
- a quantidade de munição disponível;
- e um indicador de quando deve proteger-se.

Suas principais ações neste contexto serão:

- atirar no inimigo, perdendo munição;
- virar-se para um lado ou outro;
- mover-se modificando sua noção de distância em relação ao inimigo;
- proteger-se quando perceber que não tem mais munição.

Só com estas especificações já poderíamos modelar um agente essencialmente reativo simplesmente conhecendo as percepções que ele é capaz de captar do ambiente do jogo. Passemos então para a prática começando pela definição mais geral – a do agente em si.

3.1. Declarando agentes ICE

A descrição utilizando a IADL pode ser feita utilizando-se de qualquer editor para gerar o arquivo com a descrição. O corpo de uma descrição de agente ICE segue o padrão descrito a seguir:

```
agent Soldado
{
  belief:
  ...
  action:
  ...
  objective:
  ...
  <plans>
  ...
  <reactions>
  ...
}
```

Note que, para cada palavra reservada destacada, teremos uma estrutura diferente a ser definida e entendida que será descrita nas seções subseqüentes.

Como citado anteriormente, os agentes ICE contemplam características BDI ao mesmo tempo em que podem ser definidas reações. Dentro da estrutura de definição, temos então crenças (palavra reservada `belief`), objetivos (palavra reservada `objective`) e planos (palavra reservada `plan`) para modelagem de comportamento deliberativo baseado em BDI, e

reações (palavra reservada `reaction`) para representação dos reflexos, compondo um comportamento reativo.

3.2. Crenças

Primeiramente define-se o que o personagem que queremos modelar conhece sobre si mesmo e sobre seu ambiente, no caso o jogo. As crenças suportarão os atributos do personagem e estas devem possuir um identificador com um tipo associado (`int`, `float`, `bool`, `string` e tipos compostos), podendo ainda ser inicializadas com valores. Vejamos o exemplo descrito anteriormente, agora na linguagem IADL:

```
belief:
  int direction=0;
  int ammo=5;
  bool cover = false;
  int distance = 0;
  int lastTurn = 1;
```

3.3. Ações


Depois de definidas as crenças, podemos passar para a definição das ações que o personagem poderá executar. A execução destas ações poderá modificar as crenças do agente, vai depender da descrição. Seguindo a especificação textual do agente Soldado que estamos modelando, vejamos a sintaxe deste bloco de descrição:

```
action:
  shoot()
  {
    --ammo;
  }

  turn(int dir)
  {
    direction += dir;
    lastTurn = dir;
  }

  move(int size)
  {
    distance += size;
  }

  takeCover()
  {
    cover = true;
  }
```


Podem ser definidas ações com ou sem parâmetros. As ações definidas sem parâmetros como no exemplo as ações `shoot()` e `takeCover()` modificam as crenças do agente atribuindo novos valores constantes a elas. As outras ações `turn(int dir)` e `move(int size)` vão depender do valor que recebem para alterar as crenças que modificam 

3.4. Reações

Nesta próxima etapa definiremos um conjunto de reações para o personagem Soldado, que serão descritas em função das percepções definidas para o agente.

Sabendo que o Soldado perceberá, por exemplo, a existência de um inimigo à sua esquerda, à sua direita ou à sua frente, pode-se modelar algumas reações imediatas como atirar se o inimigo estiver na sua frente ou virar para o lado aonde o inimigo se encontra.

Este bloco de comando é representado pela palavra reservada `reaction` seguida do identificador da reação e a condição para este reflexo ser acionado. A sintaxe deste comando está ilustrada logo abaixo:

```
reaction turnLeft if (noiseLeft)
    turn(2);
reaction turnRight if (noiseRight)
    turn(1);
reaction shootFront if (noiseFront)
    shoot();
```

No caso do agente Soldado, as percepções definidas são as três citadas anteriormente. Logo, para cada percepção pode-se definir um reflexo correspondente e, através dele, invocar a execução de alguma das ações que o personagem implementa.

Usando como exemplo a percepção `noiseFront`, que supõe a presença de um inimigo à frente do Soldado, a reação denominada `shootFront` mapeia esta percepção para execução da ação `shoot()`, ou seja, atirar imediatamente no inimigo. Pode-se definir outras percepções neste bloco de reações (quantas se julgar necessário) concluindo a definição do componente reativo para o personagem em questão.

Vejamos a seguir as estruturas necessárias para que se possa incluir pró-atividade no agente.

3.5. Objetivos

Um grande passo na busca de deliberação é dar ao personagem objetivos a alcançar. Isto pode ser descrito utilizando a palavra reservada `objective`. No agente Soldado modelado até agora, ainda não havíamos inserido tal característica. Para simplificar, definiremos dois objetivos principais ao nosso personagem: matar e esconder-se. Vejamos:

```
objective:
  kill()
    pre(ammo>0)
    pos(false)
  hide()
    pre(ammo==0)
    pos(false)
```

As condições para a busca de um objetivo são modeladas usando-se operadores lógicos e de comparação, definindo-se através da sintaxe ilustrada acima. No exemplo utilizamos apenas operadores de comparação (>, ==), mas poder-se-ia utilizar operadores lógicos como *and* e *or* dentro do mesmo bloco caso mais de uma condição devesse ser obedecida.

Pode-se definir também, opcionalmente, condições que indiquem que o objetivo fora alcançado utilizando a mesma sintaxe para definição das pré-condições. Caso isto não seja desejado, estabelece-se como padrão a palavra reservada `false`.

A ordem de definição dos objetivos é de total relevância, pois ela que definirá a prioridade dos mesmos quando inseridos na arquitetura ICE. Feito isto, os planos para cumpri-los deverão ser especificados.

3.6. Planos

A ordem de definição dos planos é importante, pois definirá a prioridade de cada plano em relação aos outros. Através da palavra reservada `priority`, estabelece-se um critério de desempate quando houver mais de um plano para o mesmo objetivo.

O Soldado do exemplo tem planos para caçar o inimigo à esquerda (`huntLeft`), à direita (`huntRight`) e para fugir caso se sinta ameaçado (`run`). Cada plano especificado deve conter a palavra reservada `plan` e deve receber como parâmetro o objetivo que se destina a cumprir. Nestes planos coloca-se, então, a seqüência de ações que devem ser executadas pelo agente em cada plano. Estas ações devem estar definidas na seção `action`.

Complementando a definição de cada plano pode-se estabelecer mais algumas condições para execução de suas ações através do comando `if`. Estas condições seguem a mesma sintaxe de definição das pré-condições dos objetivos. Caso não se deseje impor estas condições, utiliza-se a palavra reservada `true` como exemplifica o plano `run` mais abaixo.

```
plan huntLeft(kill)
  if(lastTurn == 1)
  priority 1
  {
    move(1);
    turn(2);
  }

plan huntRight(kill)
  if(lastTurn == 2)
  priority 1
  {
    move(1);
    turn(1);
  }

plan run(hide)
  if (true)
  priority 1
  {
    move(5);
    takeCover();
  }
```

Terminada a descrição, o agente poderá ser compilado gerando o código em C++ correspondente, seguindo as diretivas propostas pela arquitetura ICE.

4. Utilizando o Compilador ICE

O Compilador ICE (ICEC) é responsável por traduzir as descrições de agentes feitas na linguagem IADL transformando-as em um

conjunto de classes que estendem as classes básicas que constituem a arquitetura ICE.

O compilador utiliza uma interface do tipo linha de comando. O usuário do ICEC deve executá-lo informando o caminho completo do arquivo que contém a descrição do agente que deseja compilar e o diretório em que será colocado o código gerado.

Temos aqui o exemplo do personagem Soldado que modelamos nas seções anteriores passo a passo:

```
agent Soldado
{
    belief:
        int direction=0;
        int ammo=5;
        bool cover = false;
        int distance = 0;
        int lastTurn = -1;
    action:
        shoot()
        {
            --ammo;
        }

        turn(int dir)
        {
            direction += dir;
            lastTurn = dir;
        }

        move(int size)
        {
            distance += size;
        }

        takeCover()
        {
            cover = true;
        }

    objective:
        kill()
            pre(ammo>0)
            pos(false)
        hide()
            pre(ammo==0)
            pos(false)

    plan huntLeft(kill)
        if(lastTurn == 1)
            priority 1
        {
            move(1);
        }
}
```

```

        turn(-1);
    }

    plan huntRight(kill)
        if(lastTurn == -1)
            priority 1
    {
        move(1);
        turn(1);
    }

    plan run(hide)
        if (true)
            priority 1
    {
        move(5);
        takeCover();
    }

    reaction turnLeft if (noiseLeft)
        turn(-1);
    reaction turnRight if (noiseRight)
        turn(1);
    reaction shootFront if (noiseFront)
        shoot();
}

```

O arquivo Soldado.ice que contém esta descrição agora será então submetido ao compilador ICE para transformar a descrição em código C++. Como a interface disponível é linha de comando, como na Figura 33, a sintaxe será:

icec <caminho do arquivo> <diretório para geração de código>
Exemplo:

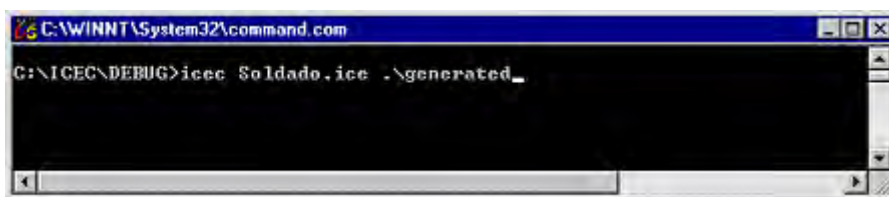


Figura 33 Utilização do ICEC

5. Saída do Compilador

A saída regular do ICEC é um conjunto de classes que correspondem à representação em C++ da especificação do agente em IADL. Estas classes devem ser compiladas e ligadas ao programa alvo juntamente com as classes do *kernel* ICE. Cabe ao programador instanciar tantas vezes quanto julgar necessário cada agente dentro do jogo, uma vez que cada

especificação representa um padrão de comportamento podendo ser livremente replicado para diversas entidades do jogo.

Além disto, o compilador ICEC é capaz de detectar os erros mais comuns possíveis na especificação de um agente em IADL. Dentre os erros mais comuns estão:

- Erros de sintaxe: quando uma construção da linguagem é utilizada incorretamente, ou uma construção inexistente é utilizada o compilador informa ao usuário a linha e o último *token* reconhecido antes da detecção do erro;
- Erros de declaração: quando uma crença não declarada é utilizada ou quando um tipo não declarado é utilizado para uma declaração de crença, o compilador informa a linha e o nome do símbolo não declarado;
- Funcionalidades não implementadas: quando uma construção relativa a uma funcionalidade não implementada é utilizada o compilador informa a linha e o último *token* reconhecido antes da detecção do erro. Este tipo de erro atualmente ocorre quando construções relativas a listas de crenças são utilizadas.

6. Limitações do compilador e da linguagem

- Um objetivo poderá ter vários planos, e, dependendo das condições estabelecidas para executar um plano, este pode sofrer “postergação indefinida”, ou seja, nunca ser executado;
- Cada plano irá se propor a alcançar um objetivo específico, não é possível que um plano sirva uma lista de objetivos;
- Não existe um componente planejador per se, os planos são estabelecidos manualmente. A deliberação se dá na escolha do melhor plano para alcançar o objetivo, e não na elaboração do plano em si;

7. Considerações

Este manual se propõe a fornecer uma noção geral da linguagem IADL e da utilização do compilador ICE. Para melhor aproveitar os recursos providos pela linguagem, a gramática completa da IADL encontra-se disponível no volume final de TC2, bem como uma visão detalhada da arquitetura ICE e seus componentes.

GLOSSÁRIO

Agente: abstração utilizada em IA representando uma entidade autônoma, geralmente pró-ativa, que “sente” o ambiente através de sensores e age sobre o mesmo através de atuadores.

A-Life: ou *Artificial Life*; tentativa de pesquisadores de IA de imitar o comportamento de seres vivos, ou grupos de seres vivos.

Consuetudinário: que é baseado nos costumes, senso comum.

Dæmon: programa residente em memória utilizado em UNIX para a execução de tarefas contínuas. Pode ser visto como um agente, onde o ambiente é o sistema operacional.

DirectX: biblioteca gráfica para desenvolvimento de jogos utilizada na plataforma Windows.

Game engine: parte do jogo que lida com o funcionamento básico do mesmo.

Hard-coding: inclusão de constantes, ou partes dos dados diretamente no código de um programa.

Ether: Componente da arquitetura ICE responsável pela representação do mundo e do entrega de mensagens entre os agentes.

Persona: Componente da arquitetura ICE original responsável pelo mapeamento do estado do agente para uma representação visual, foi removido na arquitetura final.

Jogabilidade: termo normalmente associado ao grau de satisfação do jogador com um jogo eletrônico.

Joystick: periférico de interface com jogos, usualmente um manche.

Kernel: parte do sistema operacional que trata do hardware básico.

Mainframe: computador de grande porte, utilizado principalmente nos anos 70, associado à existência de um centro de processamento de dados.

Pixel: um ponto de iluminação em um monitor de vídeo.

Planejador: Componente de um sistema de inteligência artificial responsável pela criação de planos para a realização de um objetivo pré-definido.

Pró-Atividade: capacidade de tomar a iniciativa, isto é, agir em antecipação a um evento, ao contrário de apenas reagir a ele.

Proxy: do inglês, procurador. Em engenharia de software, especificamente em orientação a objetos, um objeto que tem como função representar outro [GAM94].

Script: declaração procedural de comandos ou ações a serem executadas.

Sprite: conjunto de imagens que representam uma animação.

Taxonomia: teoria da classificação.

REFERÊNCIAS BIBLIOGRÁFICAS

- [BRA90] BRATMAN, M.E. **What is intention?** In **P.R. Cohen, J. Morgan, and M.E. Pollack, editors.** Intentions in Communication, pages 15--31. MIT Press, Cambridge, Mass., 1990
- [BRO86] BROOKS, Rodney A. **A robust layered control system for a mobile robot.** IEEE Journal of Robotics and Automation, 1986.
- [DAV96] DAVIS, Randall **What Are Intelligence? And Why? 1996 AAAI Presidential Address.** AI Magazine Volume 19 Number 1, American Association for Artificial Intelligence.
- [DAW01] DAWSON, Bruce **Micro-Threads for Game Object AI.** Game Programming Gems 2, Charles River Media, Inc., 2001.
- [DHE00] DHEIN, Guilherme **Integrando deliberação e reatividade em uma arquitetura de agentes híbrida homogênea.** Dissertação de mestrado, Faculdade de Informática PUCRS, 2000.
- [FIS98] FISCHER, Klaus; MÜLLER, Jörg P.; PISCHEL, Markus **A Pragmatic BDI Architecture.** Readings in Agents, Morgan Kaufmann Publishers(ed) HUHNS, Michael N.; SINGH, Munindar P., 1998.
- [FRA96] FRANKLIN, Stan; GRAESSER, Art **Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.** Proceedings of the 3rd International Workshop on Agent Theories, Architectures and Languages, Springer Verlag, 1996.
- [GAM94] GAMMA, Erich et al **Design Patterns: elements of reusable object-oriented software.** Addison-Wesley Publishing Company, Inc, 1994.

- [GAS98] GASSER, Les **Agents and Multiagent Systems: Themes, Approaches and Challenges**. Readings in Agents (ed) HUHNS, Michael N.; SINGH, Munindar P., Morgan Kaufmann Publishers, 1998.
- [HAD96] HADDADI, Afsaneh; SUNDERMEYER, Kurt **Belief-Desire-Intention Agent Architectures**. Foundations of Distributed Artificial Intelligence (ed) O'HARE, G. M. P.; Jennings, N. R., John Wiley and Sons, Inc., 1996.
- [HOW01] HOWLAND, Geoff **How do I make games**. Extraído em 7 de Fevereiro de 2001. Online. Disponível na Internet: www.lupinegames.com/articles/path_to_dev.html.
- [HUE97] HUEBNER, Robert **Adding Languages to Game Engines**. Game Developer Magazine September 1997, CMP Media Inc., 1997.
- [LAM95] LAMOTHE, André, **Building Brains into Your Games**. Extraído em 20 de Janeiro 2001. Online. Disponível na Internet www.gamedev.net/reference/articles/article574.asp em .
- [MIN92] MINSKY, Marvin L. **Future of AI technology**. Toshiba Review Vol. 47 No. 7, July 1992.
- [RAB00a] RABIN, Steve **The Magic of data driven Design**. Game Programming Gems, Charles River Media, Inc., 2000.
- [RAB00b] RABIN, Steve **Designing a General Robust AI Engine**. Game Programming Gems, Charles River Media, Inc., 2000.
- [RAO98] RAO, Anand S.; GEORGEFF, Michael P. **Modeling Rational Agents within a BDI-Architecture**. Readings in Agents, Morgan Kaufmann Publishers(ed) HUHNS, Michael N.; SINGH, Munindar P., 1998.

- [ROL00] ROLLINGS, Andrew; MORRIS, Dave **Game Architecture and Design**. The Coriolis Group, LCC, 2000.
- [RUS94] RUSSELL, Stuart D.; NORVIG, Peter **Artificial Intelligence: A Modern Approach**, Prentice Hall, 1994.
- [SHO93] SHOHAN, Yoav **Agent-oriented programming**. Artificial Intelligence 60, Elsevier Science Publishers B. V., 1993.
- [WEI99] WEIß, Gerhard **Prologue**. Multiagent systems: a modern approach to distributed artificial intelligence, The MIT press, 1999.
- [WOO99] WOOLDRIDGE, Michael **Intelligent Agents**. Multiagent systems: a modern approach to distributed artificial intelligence, The MIT press, 1999.
- [WOO00] WOODCOCK, Steve **Game AI: The State of the Industry**. Game Developer Magazine August 2000, CMP Media Inc., 2000.
-