UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Holoparadigma: um Modelo Multiparadigma Orientado ao Desenvolvimento de Software Distribuído

por

JORGE LUIS VICTÓRIA BARBOSA

Tese submetida à avaliação, como requisito
parcial para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, abril de 2002

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Barbosa, Jorge Luis Victória

Holoparadigma: um Modelo Multiparadigma Orientado ao Desenvolvimento de Software Distribuído / por Jorge Luis Victória Barbosa. - Porto Alegre: PPGC da UFRGS, 2002.

213 f. : il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Geyer, Cláudio Fernando Resin.

1. Multiparadigma 2. Sistemas Paralelos e Distribuídos. 3. Mobilidade. 4. Paradigmas de Programação. I. Geyer, Cláudio Fernando Resin. II. Título.

Agradecimentos

Chegamos a uma estação. Os companheiros de viagem foram o principal motivo da jornada. Agradeço pela companhia. O próximo trem sai em breve.

"*Essa habilidade do homem em separar a si próprio do ambiente, bem como em dividir e distribuir as coisas, levou em última instância a um largo espectro de resultados negativos e destrutivos, pois ele perdeu a consciência do que estava fazendo e, deste modo, estendeu o processo de divisão além dos limites dentro dos quais este opera adequadamente. Em essência, o processo de divisão é uma maneira conveniente e útil de pensar sobre as coisas, principalmente no domínio das atividades práticas, técnicas e funcionais (por exemplo, dividir um terreno em diferentes campos onde várias safras serão cultivadas). Todavia, quando este modo de pensamento é aplicado de uma forma mais ampla à noção do homem a respeito de si mesmo e a respeito do mundo todo em que vive (isto é, à sua visão de mundo pessoal), então ele deixa de considerar as divisões resultantes como meramente úteis ou convenientes e começa a ver e a experimentar a si próprio, e ao seu mundo, como efetivamente constituídos de fragmentos separadamente existentes. Guiado por uma visão pessoal de mundo fragmentária, o homem então age no sentido de fracionar a si mesmo e ao mundo, de tal sorte que tudo parece corresponder ao seu modo de pensar. Ele assim obtém uma prova aparente de que é correta a sua visão de mundo fragmentária, embora, é claro, negligencie o fato de que é ele próprio, agindo de acordo com o seu modo de pensar, a causa da fragmentação que agora parece ter uma existência autônoma, independente da sua vontade e do seu desejo.*" **David Bohm**, *A Totalidade e a Ordem Implicada* [BOH 80, p.20]

"*O homem solitário pensa sozinho e cria novos valores para a humanidade. Inventa assim novas regras morais e modifica a vida social. A personalidade criadora deve pensar e julgar por si mesma, porque o progresso moral da sociedade depende exclusivamente da sua independência. A não ser assim, a sociedade estará inexoravelmente votada ao malogro, e o ser humano privado da possibilidade de comunicar. Defino uma sociedade sadia por este laço duplo. Somente existe por seres independentes, mas profundamente unidos ao grupo.*" **Albert Einstein**, *Como vejo o mundo* [EIN 81, p.15]

"*Es preciso que el hombre de ciencia deje de ser lo que hoy es con deplorable frecuencia: un bárbaro que sabe mucho de una cosa.... Todo aprieta para que se invente una nueva integrácion del saber, que hoy anda hecho pedazos por el mundo... Ha llegado a ser un asunto urgentísimo e inexcusable da la humanidad inventar una técnica para habérselas adecuadamente con la acumulación de saber que hoy posee. Si no encuentra maneras fáciles para dominar esa vegetación exuberante quedará el hombre ahogado por ella. Sobre la selva primaria de la vida vendría a yuxtaponerse esta selva secundaria de la ciencia, cuya intención era simplificar aquélla. Si la ciencia puso ordem en la vida, ahora será preciso poner también orden en la ciencia, organizarla - ya que no es posible reglamentarla-, hacer posible su perduración sana. Para ello hay que vitalizarla, esto es, dotarla de una forma compatible con la vida humana que la hizo y para la cual fue hecha. De otro modo - no vale recostarse en vagos optimismos - la ciencia se volatizará, el hombre se desinteresará de ella... Y el movimiento que lleva a la investigácion a disociarse indefinidamente en problemas particulares, a pulverizarse, exige una regulácion compensatoria - como sobreviene en todo organismo saludable - mediante un movimiento de dirección inversa que contraiga y retenga en un rigoroso sistema la ciencia centrífuga.*" **José Ortega y Gasset**, *Mision de la Universidad* [ORT 92, p.71]

"*É bastante provável que na história do pensamento humano os desenvolvimentos mais fecundos ocorram, não raro, naqueles pontos para onde convergem duas linhas diversas de pensamento. Essas linhas talvez possuam raízes em segmentos bastante distintos da cultura humana, em tempos diversos, em diferentes ambientes culturais ou em tradições religiosas distintas. Desta forma, se realmente chegam a um ponto de encontro - isto é, se chegam a se relacionar mutuamente de tal forma que se verifique uma interação real -, podemos esperar novos e interessantes desenvolvimentos a partir dessa convergência.*" **Werner Heisenberg**

"*O cientista vive com a realidade. Tanto melhor: conhecer a realidade é aceitá-la, e, por fim, amá-la. Em certo sentido, o cientista é uma criança culta. Há algo de cientista em toda criança. Outras pessoas ultrapassam essa fase, mas ele permanece criança a vida inteira.*" **George Wald**

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ABI	*AutoBehavior Interaction*
ACG	*Actions Composition Graph*
AHI	*AutoHistory Interaction*
AI	*Action Identifier*
AIG	*Actions Invocation Graph*
AIP	Ação Imperativa Predefinida
AKL	*Andorra Kernel Language*
BBI	*Being Being Interaction*
BCI	*Behavior Component Interaction*
BCP	*Behavior Cloning Policy*
BCR	*Behavior Cloning Rules*
BHI	*Behavior History Interaction*
BI	*Being Identifier*
BII	*Behavior Interface Interaction*
BNF	*Backus Naur Form*
CD	*Cloning Descriptor*
CDL	*Cloning Descriptor List*
CHB	*Class/Holo Bytes*
CHI	*Component History Interaction*
CII	*Component Interface Interaction*
CLEI	Cláusula Lógica com Extensão não Imperativa
CNPq	Conselho Nacional de Pesquisa
CTA	Clonagem de Transição Automática
CVM	Código Virtual Multiparadigma
DHolo	*Distributed Holo*
DHoloTree	*Distributed HoloTree*
DI	*Destiny Identifier*
DLO	*Distributed Logic Objects*
DOBPS	*Distributed, Object Based Programming System*
DOBuilder	*Distributed Object Builder*
DSM	*Distributed Shared Memory*
ECCD	Espaço Computacional Compartilhado Distribuído
ECI	Estrutura de Configuração de Invocação
EHU	*Enriched Herbrand Universe*
ESP	*Extended Shared Prolog*
ET	Espaço de Tuplas
ETA	*Everything buT Assignment*
ETL	Espaço de Tuplas Lógicas
ExEHDA	*Execution Environment for high Distributed Applications*
FA	*Functional Action*
FAFI	*FActs FIrst*
FAPERGS	Fundação de Amparo à Pesquisa no Rio Grande do Sul
GRANLOG	*GRANularity analizer for LOGic programming*
HJBT	*Holo/Java Bytes Time*
HJLT	*Holo/Java Lines Time*
HoloEnv	*Holo Environment*
HoloVM	*Holo Virtual Machine*

IA	*Imperative Action*
INE	Índice de Ente
ISAM	Infraestrutura de Suporte às Aplicações Móveis
JHB	*Java/Holo Bytes*
JHL	*Java/Holo Lines*
KS	*Knowledge Sources*
LA	*Logic Action*
LAN	*Local Area Network*
LPA	*Logic Programming Associates*
MA	*Multiparadigm Action*
MI	Multiparadigma de Integração
MET	Múltiplos Espaços de Tuplas
MIA	*Modular Imperative Action*
MFA	*Modular Functional Action*
MLA	*Modular Logic Action*
MUP	Multiparadigma de Unificação Parcial
MUT	Multiparadigma de Unificação Total
MVM	Máquina Virtual Multiparadigma
NEC	Número de Entes Componentes
NECE	Número de Entes Componentes Elementares
NECO	Número de Entes Componentes Compostos
NI	Nível
NIA	Nível de Abstração
NIC	Nível de Composição
OLI	*Object Logic Integration*
OPM	*Oz Programming Model*
OWB	*Objects With Brain*
RDP	Resolução Distribuída de Problemas
ReMMoS	*Replication Model in Mobility Systems*
PERDIO	*Persistent and Distributed Programming in Oz*
SMA	Sistema MultiAgentes
SP	*Shared Prolog*
TEC	Total de Entes Componentes
TECE	Total de Entes Componentes Elementares
TECO	Total de Entes Componentes Compostos
TONIA	Total de Níveis de Abstração
TONIC	Total de Níveis de Composição
UG	Unidade de Gerenciamento
UML	*Unified Modeling Language*
WAM	*Warren Abstract Machine*
WAN	*Wide Area Network*

Resumo

Este texto apresenta um novo modelo multiparadigma orientado ao desenvolvimento de software distribuído, denominado **Holoparadigma**. O Holoparadigma possui uma semântica simples e distribuída. Sendo assim, estimula a modelagem subliminar da distribuição e sua exploração automática. A proposta é baseada em estudos relacionados com modelos multiparadigma, arquitetura de software, sistemas *blackboard*, sistemas distribuídos, mobilidade e grupos. Inicialmente, o texto descreve o modelo. Logo após, é apresentada a **Hololinguagem**, uma linguagem de programação que implementa os conceitos propostos pelo Holoparadigma. A linguagem integra os paradigmas em lógica, imperativo e orientado a objetos. Além disso, utiliza um modelo de coordenação que suporta invocações implícitas (*blackboard*) e explícitas (mensagens). A Hololinguagem suporta ainda, concorrência, modularidade, mobilidade e encapsulamento de *blackboards* em tipos abstratos de dados. Finalmente, o texto descreve a implementação da **Holoplataforma**, ou seja, uma plataforma de desenvolvimento e execução para a Hololinguagem. A Holoplataforma é composta de três partes: uma ferramenta de conversão de programas da Hololinguagem para Java (ferramenta **HoloJava**), um ambiente de desenvolvimento integrado (ambiente **HoloEnv**) e um plataforma de execução distribuída (plataforma **DHolo**).

Palavras-Chave: Multiparadigma, *Blackboard*, Sistemas Distribuídos, Paradigmas de Programação, Grupos.

TITLE: "HOLOPARADIGM: A MULTIPARADIGM MODEL ORIENTED TO DISTRIBUTED SOFTWARE DEVELOPMENT"

Abstract

This text presents a new multiparadigm model oriented to the development of distributed software, called **Holoparadigm**. Holoparadigm has a simple and distributed semantics. Therefore, it stimulates the subliminal modeling of distribution and its automatic exploitation. The proposal is based on researches related to multiparadigm models, software architecture, blackboard systems, distributed systems, mobility and groups. First of all, the model is described. After that, a programming language (**Hololanguage**) that implements the main concepts of the Holoparadigm is presented. The language integrates logic, imperative and object oriented paradigms. It uses a coordination model that supports implicit (blackboard) and explicit (messages) invocation. Besides that, Hololanguage supports concurrency, modularity, mobility and encapsulation of blackboards in abstract data types. In addition, the text describes the implementation of the **Holoplatform**, i. e., a platform for developing and executing programs using the Hololanguage. The Holoplatform is composed by three parts. The first part is a program conversion tool called **HoloJava**. This tool translates Hololanguage programs into Java. The second part is an integrated environment to the development of programs (**HoloEnv**). Finally, the principles of a distributed execution environment (called **DHolo)** are proposed.

Keywords: Multiparadigm, Blackboard, Distributed Systems, Programming Paradigms, Groups.

1 Introdução

Este capítulo contém uma introdução à tese. As próximas seções apresentam o tema pesquisado, a motivação, o contexto e o histórico do trabalho, seus objetivos e a estrutura do texto.

1.1 Tema

Este texto apresenta a criação de um modelo multiparadigma orientado ao desenvolvimento de software distribuído, denominado **Holoparadigma** (de forma sucinta, **Holo**). Um modelo multiparadigma suporta a integração de paradigmas básicos [BAR 98, BAR 99a, BAR 2000e]. No escopo deste trabalho, são considerados básicos os seguintes paradigmas: imperativo [GHE 98, p.334; SEB 99, p.21], funcional [BAC 78, COM 91], em lógica [KOW 79, KOW 79a, ROB 92], orientado a objetos [MEY 87; MEY 96; MEY 98; GHE 98, p.285; MEY 99; SEB 99, p.435] e orientado a agentes [SHO 93, COM 94]. A gênese e o desenvolvimento do Holoparadigma envolvem, especialmente, os seguintes tópicos de pesquisa: paradigmas de programação [BAR 98, GHE 98, SEB 99], modelos multiparadigma [BAR 2000d], arquitetura de software [IEE 95, SHA 96], sistemas *blackboard* [GAR 95, VRA 95, PFL 97, BAR 2001], sistemas paralelos e distribuídos [BAL 89, AND 91, JOU 97, SKI 98, BAR 99a, BAR 2000, BAR 2001, BAR 2001b], mobilidade [ROY 97, IEE 98, LAN 98, FER 99, FER 2001, FER 2001a] e grupos [LIA 90, BIR 93, NUN 98, LEA 2001].

1.2 Motivação

A abstração é um dos principais instrumentos intelectuais do ser humano. Através dela, a mente humana capta apenas as informações relevantes de uma determinada realidade, concentrando esforços apenas no essencial e abstraindo o restante. No universo dos computadores a abstração tem sido utilizada em larga escala. Neste contexto, sua principal função vem sendo a simplificação do uso dos sistemas computacionais. Por exemplo, a abstração criada através da linguagem *assembly* simplificou o uso da linguagem de máquina. Por sua vez, as linguagens imperativas de alto nível abstraíram o *assembly*, no entanto, mantiveram a essência imperativa da arquitetura *Von Neumann* [GHE 98, p. 8; SEB 99, p.20]. Em uma nova etapa, a criação dos paradigmas declarativos (funcional, em lógica e orientado a objetos) abstraiu a natureza imperativa do hardware e criou linguagens que enfocam a descrição do problema em detrimento da descrição do controle da execução. Estas abstrações demandam a criação de camadas de software que simplifiquem a realidade abstraída. Por exemplo, um montador concretiza a abstração *assembly*. Por sua vez, um compilador concretiza a abstração *linguagem de alto nível*. Em alguns casos, a abstração *linguagem declarativa* necessita ainda de uma máquina abstrata para concretização de outra abstração, ou seja, o *código virtual*. Independentemente dos níveis de abstração utilizados, a força vital do sistema computacional possui sempre a mesma origem, isto é, o hardware. A abstração simplifica o uso dos computadores, mas exige vitalidade computacional para sua concretização.

O surgimento do processamento paralelo trouxe maior vitalidade para os sistemas computacionais. No entanto, trouxe também complexidade adicional para a fonte da vitalidade, ou seja, o hardware. O hardware paralelo permite a exploração do paralelismo em nível de execução. Neste nível, a paralelização do problema já deve ter sido realizada. Sendo assim, o paralelismo deve ser propagado para as camadas de

software criadas para simplificação do uso do computador, isto é, o paralelismo deve ser manipulado através do software básico.

A essência da arquitetura *Von Neumann* consiste em um único fluxo de controle manipulando dados. Essa manipulação gera um único fluxo de dados. Sendo assim, *Von Neumann* criou uma arquitetura para processamento seqüencial. As primeiras abstrações criadas para simplificação do desenvolvimento de software mantiveram a mesma essência. As linguagens imperativas são essencialmente seqüenciais, portanto, dificultam a exploração automática do paralelismo. A abstração do paralelismo é dificultada e sua manipulação explícita estimulada. Por outro lado, as linguagens declarativas abstraem a arquitetura *Von Neumann* e enfocam o domínio dos problemas a serem resolvidos [BAC 78]. A descrição declarativa está permeada do paralelismo existente no domínio modelado. Neste caso, a abstração do paralelismo é facilitada e sua manipulação implícita estimulada. Surge assim, a primeira motivação:

Primeira Motivação - Paralelismo Implícito: a exploração automática do paralelismo implícito nos paradigmas declarativos tem sido indicada como um caminho para simplificação do processamento paralelo.

A tecnologia dos sistemas computacionais tem sofrido fortes mudanças. Os avanços da microeletrônica vêm diminuindo o preço do hardware e aumentando seu poder computacional. Em especial, o advento do microprocessador ocasionou uma revolução na arquitetura dos computadores. Em complemento, o desenvolvimento de soluções eficientes para interconexão dos sistemas computacionais fez com que a área de redes de computadores assumisse uma posição de destaque. Nos últimos anos, o crescimento exponencial da *Internet* vem se destacando como um fenômeno tecnológico e de mercado. Neste contexto, as plataformas computacionais vêm migrando de sua natureza centralizada para uma nova realidade distribuída. Os sistemas distribuídos têm recebido cada vez mais dedicação tanto dos centros de pesquisa quanto das empresas. Este universo distribuído está se tornando a base para o desenvolvimento de sistemas computacionais. Na década de 80, membros da Sun Microsystems Inc. cunharam um *slogan* que cada vez mais se torna uma realidade: "A Rede é o Computador" [FRE 99, p. 2].

As novas características e potencialidades introduzidas pela realidade distribuída deverão ser exploradas. Seguindo as tendências da evolução dos computadores, os sistemas distribuídos necessitarão cada vez mais de abstrações que simplifiquem seu uso. A criação de software distribuído está se tornando uma das principais tarefas da ciência da computação. Neste contexto, o estudo de metodologias para o desenvolvimento de software distribuído ocupa uma posição de destaque. De forma semelhante à exploração automática do paralelismo (primeira motivação), o principal problema para uso adequado da arquitetura distribuída consiste na determinação de um paradigma de desenvolvimento de software que possua na sua semântica o potencial para exploração automática da distribuição (distribuição implícita). Através da distribuição implícita, o desenvolvimento de abstrações distribuídas (software) é simplificado e estimulado. Neste texto, a seção 1.1 do anexo 1 propõe uma organização e uma taxonomia para arquiteturas distribuídas. Neste sentido, surge a segunda motivação:

Segunda Motivação - Arquiteturas Distribuídas: cada vez mais, as arquiteturas distribuídas estão se tornando a plataforma básica dos sistemas computacionais. Desta forma, torna-se importante a criação de paradigmas de software que suportem a exploração automática da distribuição (distribuição implícita).

Conforme discutido durante a apresentação da primeira motivação, o desenvolvimento de software é baseado em abstrações. Estas abstrações possuem como base um paradigma, ou seja, uma forma de perceber e modelar o mundo real. No âmbito do paradigma são desenvolvidas metodologias, linguagens e ferramentas. Atualmente, existem diversos paradigmas considerados básicos [BAR 98].

Nos últimos anos o tema multiparadigma vem sendo pesquisado continuamente [HAI 86, IEE 86, PLA 91, WEG 93, MUL 95, NGK 95, CHA 97, LEE 97, APT 98, PIN 99, BAR 2000c, BAR 2000d, BAR 2000e, TAR 2001, BAR 2001, BAR 2001a, BAR 2001b, BAR 2001c]. Os pesquisadores deste tema propõem a criação de modelos de desenvolvimento de software através da integração de paradigmas básicos. Através dessa proposta eles buscam dois objetivos: a superação das limitações de cada paradigma e a exploração conjunta das suas características consideradas benéficas. Neste contexto, surge a terceira motivação:

Terceira Motivação – Software Multiparadigma: atualmente, a comunidade científica dedica consideráveis esforços para a integração de paradigmas básicos. A pesquisa multiparadigma suporta essa tendência.

O interesse pela exploração do paralelismo nos diferentes paradigmas de programação não é recente [COM 86, ACM 89]. Sabe-se que o paradigma imperativo dificulta a exploração automática do paralelismo [BAR 94, BAR 96, APE 2001, OPE 2001]. Este fato resulta de sua tendência em modelar o universo através de comandos imperativos, resultando assim, em vários níveis de dependências de dados e de controle [BAR 93, BLU 94]. Por outro lado, os paradigmas declarativos estimulam a exploração automática do paralelismo, abstraindo o controle da execução e enfocando a descrição do domínio modelado [BAC 78, BAR 96]. Os paradigmas declarativos tendem a refletir no modelo as fontes naturais de paralelismo existentes no domínio. Por exemplo, o paradigma em lógica transmite para o modelo computacional o potencial de paralelismo que está implícito na abordagem em lógica do universo (paralelismo OU e paralelismo E) [GUP 93, KER 94, BAR 96]. Por sua vez, o paradigma orientado a objetos transmite o potencial de paralelismo que existe no universo quando modelado através da filosofia de objetos (paralelismo inter-objetos - entre objetos; e paralelismo intra-objeto - entre métodos de um objeto) [WIA 92, BRJ 96, BRJ 98]. O mesmo domínio, modelado por paradigmas diferentes, introduzirá no modelo computacional diferentes fontes de paralelismo. Sendo assim, a opção pela forma de *perceber* o domínio durante a modelagem (paradigma) influencia nas fontes de paralelismo que ficarão disponíveis no modelo. A figura 1.1 representa essa realidade para os paradigmas em lógica e orientado a objetos.

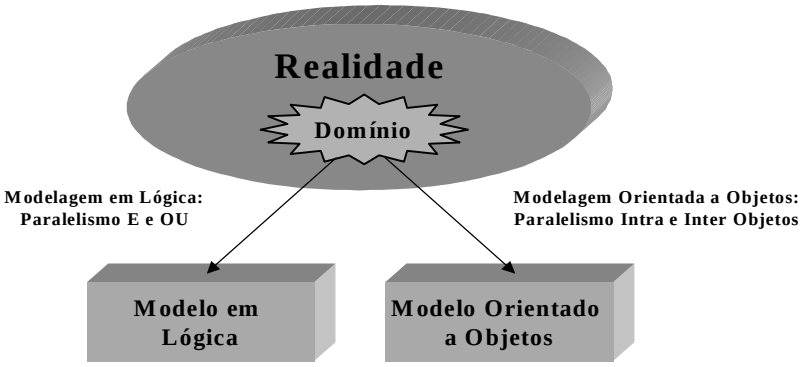


FIGURA 1.1 – Duplicidade no uso dos paradigmas

A integração de paradigmas envolve a integração de suas características. Em complemento, as características de cada paradigma estão relacionadas com suas fontes de paralelismo implícito. Portanto, a integração de paradigmas ocasiona a integração de fontes de paralelismo. Surge assim, o interesse na exploração automática de paralelismo e distribuição no software multiparadigma [NGK 95, CIA 96, ROY 97, BAR 99a, BAR 2000, TAR 2001, BAR 2001, BAR 2001b]. Destaca-se como quarta motivação:

Quarta Motivação – Software Multiparadigma Paralelo e Distribuído: a exploração automática do paralelismo e distribuição é estimulada nos paradigmas declarativos. Em complemento, através da proposta multiparadigma ocorre a integração das características dos paradigmas. Sendo assim, tende a ocorrer também a integração de suas fontes de paralelismo implícito. Neste contexto, torna-se importante o estudo relacionado com software multiparadigma paralelo e distribuído [BAR 99a, BAR 2000, BAR 2001, BAR 2001a, BAR 2001b, BAR 2001c].

As quatro motivações apresentadas estimularam o surgimento do Holoparadigma. Holo é um modelo multiparadigma que possui uma semântica simples e distribuída. Através dessa semântica, Holo estimula a exploração automática da distribuição.

1.3 Contexto e histórico

O modelo proposto está sendo desenvolvido no contexto criado por três projetos: **Opera** [OPE 2001], **Appelo** [APE 2001] e **Holoparadigma** [HOL 2001]. O projeto Opera [BRI 90, BRI 90a, GEY 92, OPE 2001] iniciou suas atividades no Laboratório de Génie Informatique (Universidade de Joseph Fourier em Grenoble / França). Atualmente, encontra-se em desenvolvimento na UFRGS uma ramificação deste projeto. No âmbito do Opera foram desenvolvidas diversas atividades explorando o paralelismo implícito existente na programação em lógica. Entre essas atividades, destaca-se a criação do modelo **Granlog** (*GRanularity ANalyzer for LOGic programming*) [BAR 94a, BAR 95, BAR 96, BAR 2000b]. Este modelo propõe a análise automática de granulosidade para Prolog com a utilização de diversas análises avançadas de programas, tais como Análise Global [BAR 96b, AZE 99] via Interpretação Abstrata, Análise de Grãos [BAR 96a] e Análise de Complexidade [BAR 97]. O Granlog foi prototipado [VAR 95, VAR 95a] e integrado com dois modelos de exploração de paralelismo na programação em lógica (**Plosys** [FER 99a, VAR 2000] e **Andorra** [DUT 99]).

Entre 1996 e 1998, as atividades do Opera foram englobadas por um projeto multi-institucional denominado **Appelo: Ambiente de Programação Paralela em Lógica** [GEY 99, APE 2001]. O Appelo foi aprovado no programa Protem III do CNPq e envolveu cinco instituições de ensino:

- Universidade Federal do Rio Grande do Sul (UFRGS);
- Universidade Federal do Rio de Janeiro (UFRJ);
- Universidade Católica de Pelotas (UCPel);
- Universidade do Porto;
- New Mexico State University.

O Holoparadigma está sendo desenvolvido no contexto criado pelos projetos Opera e Appelo. As pesquisas relacionadas com a exploração do paralelismo implícito existente no paradigma em lógica estimularam o surgimento da proposta. O projeto

Holoparadigma: Software Multiparadigma Distribuído foi aprovado no edital de novos projetos de pesquisa 2000/1 da FAPERGS. Além disso, o projeto **Holoparadigma: Um Modelo Multiparadigma para Desenvolvimento de Software Distribuído** foi aprovado no edital 2000/2 para solicitação de bolsas de iniciação científica e no edital "PROADE – recursos emergenciais 2001/1", ambos da FAPERGS.

Nos últimos dois anos, os pesquisadores envolvidos na criação do Holoparadigma vêm mantendo contato com membros da empresa **Godigital Tecnologia e Informática Ltda** [GOD 2001], visando a criação de produtos oriundos dos resultados da pesquisa. A Godigital é uma empresa especializada em soluções para *datamining* [BER 97] e *marketing* de precisão. Esta empresa desenvolveu o primeiro software de *datamining* da América Latina, ou seja, o **AIRA *Data Mining***. Este software permite a busca e extração automática de informações estratégicas em grandes volumes de dados. A empresa atua através de desenvolvimento de sistemas computacionais "inteligentes", consultoria em *datamining*, cursos e seminários. O *datamining* é uma das principais aplicações previstas para o Holoparadigma (veja seção 6.1). Durante o texto são apresentados exemplos de implementação dessa aplicação. Além disso, são discutidos testes de desempenho de implementações *datamining* seqüenciais e distribuídas usando o Holoparadigma (veja seções 5.1 e 5.3).

1.4 Objetivos

O objetivo geral deste trabalho consiste na criação de um novo modelo multiparadigma orientado ao desenvolvimento de software distribuído. Neste contexto, destacam-se como objetivos específicos:

- descrever os temas de pesquisa envolvidos na criação do modelo, discutindo o estado da arte no escopo pesquisado (trabalhos relacionados);
- apresentar os princípios do **Holoparadigma**, os quais estão guiando sua criação;
- propor uma linguagem que suporte os conceitos do modelo (**Hololinguagem**);
- propor uma plataforma de desenvolvimento e execução (**Holoplataforma**);
- propor e implementar uma ferramenta que converta programas da Hololinguagem para Java (**HoloJava**);
- propor e implementar um ambiente integrado de desenvolvimento de programas, denominado ***HoloEnvironment*** (**HoloEnv**);
- propor os princípios para criação de um ambiente de execução distribuída para o Holoparadigma, denominado ***Distributed Holo*** (**DHolo**);
- submeter a proposta à avaliação como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação;
- difundir a cultura do software multiparadigma paralelo e distribuído;
- suportar o desenvolvimento de relatórios e artigos relacionados com o tema pesquisado.

1.5 Estrutura do texto

O texto está organizado em seis capítulos e seis anexos. A figura 1.2 apresenta sua estrutura. A figura mostra um resumo do conteúdo de cada parte do texto. Os anexos possuem maior afinidade com alguns capítulos. A figura demonstra esta afinidade através de um acoplamento.

O segundo capítulo apresenta um estudo sobre software multiparadigma [BAR 98, BAR 99a, BAR 2000]. Este estudo serve de base para a criação do Holoparadigma. O capítulo contém um resumo do estado da arte no contexto da pesquisa, acompanhado de um estudo comentado de sete modelos multiparadigma (OLI [LEE 97], OWB [AMA 96, AMA 97], I^+ [NGK 95], DLO [CIA 96], ETA [AMB 96], G [PLA 91], Oz [SMO 95, HAR 2001]). Além disso, o capítulo propõe uma taxonomia para classificação de modelos multiparadigma [BAR 98a, BAR 2000d] e discute em detalhes o paralelismo e/ou distribuição em três modelos (I^+ [NGK 95], DLO [CIA 96] e Oz distribuído/Mozart [SMO 95a, ROY 97, HAR 98, HAR 99, ROY 2001]).

O terceiro capítulo apresenta o Holoparadigma [BAR 99, BAR 2000a, BAR 2000c, BAR 2000e]. Os conceitos discutidos neste capítulo norteam o restante do trabalho. O capítulo 4 propõe a Hololinguagem [BAR 2001a, DUB 2001]. Por sua vez, o quinto capítulo descreve a Holoplataforma. Neste capítulo são discutidas as ferramentas HoloJava [BAR 2001c], e os ambientes HoloEnv [BAR 99a, SOA 2000] e DHolo [BAR 2001, BAR 2001b]. Finalmente, o capítulo seis dedica-se as considerações finais do trabalho.

Cada capítulo contém uma seção de encerramento que apresenta considerações finais. Desta forma, as conclusões ficam próximas da sua origem. Por sua vez, o último capítulo aborda os tópicos finais considerados genéricos, ou seja: aplicações previstas para o Holoparadigma, principais contribuições do trabalho, conclusões gerais e trabalhos futuros. Visando enriquecer o texto foram introduzidos seis anexos, os quais são citados quando assumem importância em uma parte específica do trabalho.

O primeiro anexo apresenta tópicos relacionados com a criação do Holoparadigma e da Hololinguagem. Estes tópicos são considerados complementares e, por isso, não fazem parte dos capítulos 3 e 4. Por sua vez, o anexo 2 contém um estudo sobre definições de palavras usadas no escopo do trabalho. Este estudo serviu de suporte para o surgimento de novos termos usados para organização dos resultados e comunicação de novas idéias.

O terceiro anexo contém dez exemplos de programas desenvolvidos com a Hololinguagem. Estes holoprogramas foram utilizados em experimentos apresentados no capítulo 5. Além disso, o anexo 4 mostra a listagem dos arquivos gerados pela conversão de um programa (*datamining.holo*, veja figura 5.5 e anexo 4.1) usando a HoloJava. O *datamining.holo* é explorado em diversos exemplos durante o capítulo 5. Sendo assim, os arquivos gerados pela sua conversão merecem atenção.

O quinto anexo apresenta a listagem do arquivo *HoloJava.jj*. Este arquivo armazena a gramática da Hololinguagem usada para criação da HoloJava. Conforme discutido no capítulo 5 (veja figura 5.2), a criação da HoloJava é realizada pelo JavaCC [JAV 2001]. O *HoloJava.jj* contém a principal formalização da Hololinguagem, pois armazena sua BNF e as ações semânticas usadas para conversão de Holo para Java. Finalmente, o anexo 6 mostra a listagem dos arquivos de definição da Hololinguagem usados pela HoloJava (veja figura 5.8).

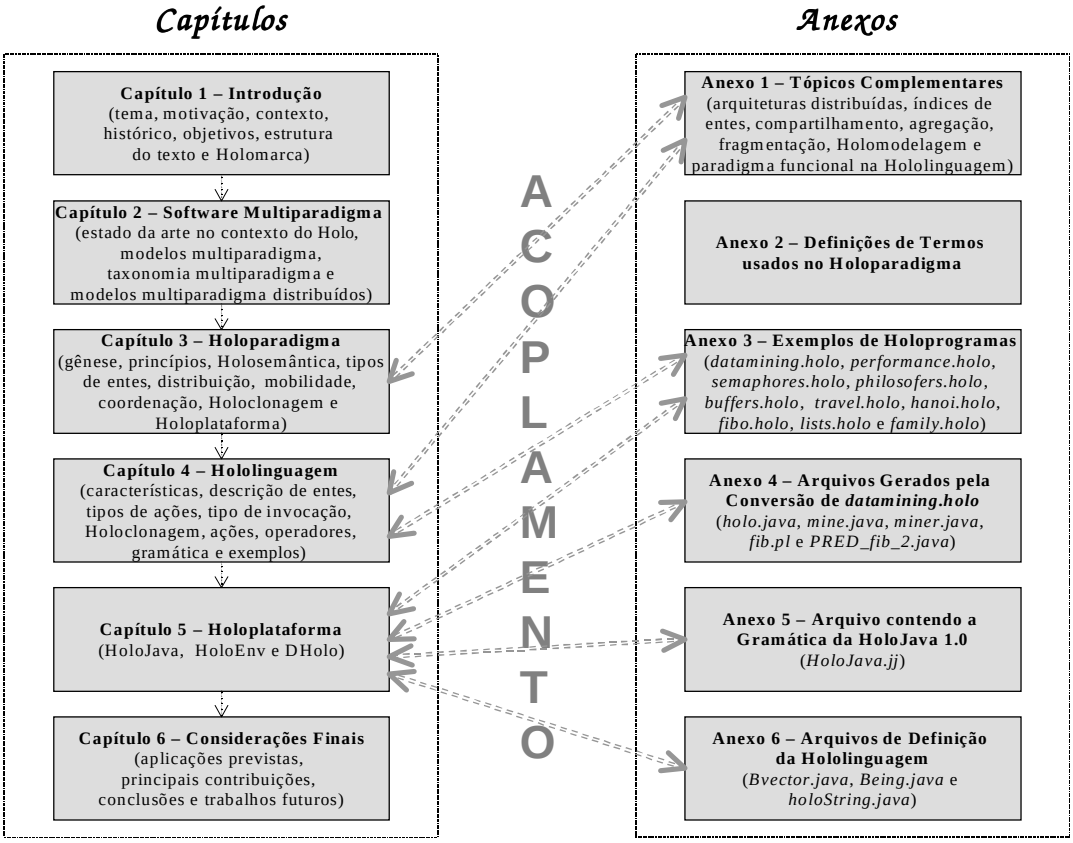


FIGURA 1.2 – Estrutura do texto

1.6 Holomarca e *site* do projeto

Visando a difusão do Holoparadigma como solução para o desenvolvimento de software distribuído, foram criados uma marca e um *site* para o projeto. A **Holomarca** é mostrada na figura 1.3. Esta marca consiste da letra H limitada por um círculo. O endereço do **Holosite** é *www.inf.ufrgs.br/~holo*. O *site* disponibiliza diversas informações relacionadas com o projeto e, em especial, as publicações desenvolvidas no âmbito do Holoparadigma.

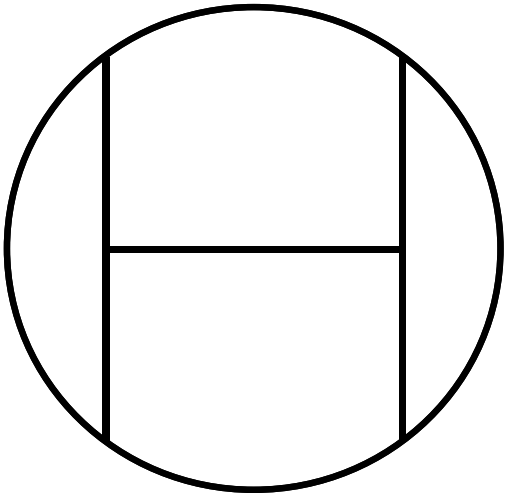


FIGURA 1.3 – Holomarca

2 Software Multiparadigma

Este capítulo dedica-se ao estudo do software multiparadigma. A seção 2.1 contém um resumo do estado da arte no contexto do Holoparadigma. A seção 2.2 descreve e comenta sete modelos multiparadigma [BAR 98, BAR 99a]. Por sua vez, a seção 2.3 propõe uma taxonomia multiparadigma [BAR 98a, BAR 2000d] . Destaca-se nesta seção, a aplicação da taxonomia na classificação dos modelos descritos na seção 2.2. Finalmente, a seção 2.4 apresenta em detalhes a exploração do paralelismo e/ou distribuição em três modelos multiparadigma [BAR 99a, BAR 2000] .

2.1 Contexto do Holoparadigma

Placer [PLA 91] e Muller et al [MUL 95] apresentam resumos da evolução dos trabalhos de pesquisa multiparadigma. Ambos os textos destacam o interesse da comunidade científica na criação de um modelo multiparadigma ideal que unifique os avanços introduzidos pelos paradigmas básicos. Por sua vez, Ng e Luk [NGK 95] apresentam uma interessante tabela que compara vinte e três propostas multiparadigma. Conforme salientado por Placer, grande parte do esforço dedicado à pesquisa multiparadigma enfoca a integração entre dois paradigmas básicos. Entre os primeiros trabalhos que seguiram essa tendência destacam-se as propostas de integração dos paradigmas em lógica e funcional. Hanus [HAN 94] apresenta os conceitos básicos envolvidos nesta integração e descreve diversas propostas. Hanus [HAN 99] propôs a linguagem Curry que permite o uso conjunto da lógica e funções. A compilação de Curry gera *byte code* Java e o uso de *threads* suporta a concorrência e o não-determinismo naturais da linguagem. Neste escopo, merecem destaque ainda os trabalhos de Chakravarty e Lock [CHA 97] e Hanus [HAN 97].

No âmbito da integração dos paradigmas em lógica e imperativo destaca-se a linguagem Alma-0 [APT 98]. Esta linguagem propõe a introdução de suporte para programação declarativa em linguagens imperativas. Alma-0 não suporta processamento simbólico, mas introduz um conjunto de características que permitem o não-determinismo e a realização de *backtracking* em linguagens imperativas.

Bugliesi et al [BUG 94] apresentam os esforços da comunidade científica dedicados à aplicação da modularidade no paradigma em lógica. Atualmente, a maioria dos sistemas de programação em lógica suportam a utilização de módulos (por exemplo, SICStus Prolog [SIC 95, p.51]). Além disso, Cabeza e Hermenegildo [CAD 2000] propuseram um novo sistema de módulos para Ciao-Prolog. A modularização soluciona uma das principais limitações do paradigma em lógica, ou seja, a falta de estrutura para desenvolvimento organizado de grandes sistemas. O surgimento do paradigma orientado a objetos fez com que a comunidade científica percebesse a possibilidade de sua integração com o paradigma em lógica. Davison [DAV 93] apresenta um estudo sobre a integração desses paradigmas destacando as motivações, técnicas e modelos propostos. Wegner [WEG 93, WEG 97] apresenta uma análise dos problemas existente na integração objetos/lógica. Amandi e Price [AMA 96] propõem a aplicação dessa integração na criação de agentes. Por sua vez, Lee e Pun [LEE 97] criaram uma metodologia (OLI) que suporta a modelagem de sistemas com a aplicação conjunta de lógica e objetos. Pineda e Hermenegildo [PIN 99] criaram O'Ciao, um modelo para programação orientada a objetos usando Ciao Prolog. Além disso, a LPA [LPA 2001] distribui o Prolog++ [PRO 2001], uma ferramenta de programação que combina Prolog e orientação a objetos.

Merecem destaque ainda, os esforços para o desenvolvimento de sistemas distribuídos baseados em modelos multiparadigma. Ciampolini et at [CIA 96] descrevem uma proposta para criação de objetos lógicos distribuídos (DLO). Esta proposta é baseada em diversos trabalhos anteriores (Shared Prolog [BRO 91], ESP [CIP 94] e ETA [AMB 96]). Por sua vez, o modelo I^+ [NGK 95] suporta a distribuição de objetos que possuem métodos implementados com funções ou predicados lógicos. Além disso, o modelo multiparadigma Oz [SMO 95, HAR 2001] serve de base para a criação de uma plataforma distribuída denominada Mozart [ROY 97, HAR 98, HAR 99, ROY 2001]. Oz utiliza um espaço de compartilhamento de restrições (*constraint store*) semelhante a um *blackboard* e suporta a programação no estilo de diversos paradigmas básicos [MUL 95, HEN 97]. Carro e Hermenegildo [CAM 99] propõem o uso de *threads* compartilhando uma base de termos Prolog. O acesso à base é realizado com o uso de diretivas semelhantes às utilizadas em Linda [CAR 86]. Tarau et al [BOS 93, BOS 96, TAR 99, TAR 2001] usam Multi-BinProlog [BOS 93, BOS 96] para o desenvolvimento de Jinni [TAR 2001], um sistema que suporta *blackboards* lógicos.

Recentemente, Barbosa e Geyer [BAR 99, BAR 2000a, BAR 2000c, BAR 2001, BAR 2001a, BAR 2001b] propuseram um novo modelo multiparadigma orientado ao desenvolvimento de software distribuído, denominado Holoparadigma (de forma abreviada, Holo). Holo propõe a integração dos paradigmas em lógica, imperativo e orientado a objetos. Além disso, utiliza um modelo de coordenação que suporta invocações implícitas (*blackboard*) e explícitas (mensagens). Holo estimula a modelagem subliminar da distribuição e sua exploração automática. Este estímulo é baseado na utilização de uma nova entidade de modelagem denominada ente. O Holoparadigma serve de base para a criação de uma nova linguagem (denominada Hololinguagem) [BAR 2001a]. Esta linguagem suporta concorrência, modularidade, mobilidade, encapsulamento de blackboards em tipos abstratos de dados. Recentemente, a integração do paradigma funcional na Hololinguagem foi proposta [DUB 2001].

A tabela 2.1 compara sete modelos (I^+, OWB, DLO, OLI, Mozart, Alma-0 e Holo). A tabela está organizada em quatro colunas: nome da proposta, paradigmas integrados, sistema de distribuição e estilo utilizado na integração.

TABELA 2.1 – Comparação entre modelos multiparadigma

Modelo	**Paradigmas integrados**	**Distribuição**	**Estilo**
I^+ *[NGK 95]*	Objetos, Lógica e Funcional	Objetos distribuídos	Programação declarativa orientada a objetos, ou seja, especificação de objetos através da lógica ou funções
OWB *[AMA 96]*	Objetos e Lógica	Não enfoca	Suporte à criação de agentes através da inserção de lógica em objetos (nova classe *LogicKnowledge*)
DLO *[CIA 96]*	Objetos e Lógica	Objetos distribuídos	Processos organizados em Objetos Lógicos implementados através de Cláusulas de Múltiplas Cabeças
OLI *[LEE 97]*	Objetos e Lógica	Não enfoca	Mapeamento de Objeto -> Lógica (classe *Pterm*) e Lógica -> Objeto (*Enriched Herbrand Universe*)
Oz/Mozart *[HAR 98]*	Objetos, Lógica e Funcional	Objetos distribuídos	Tarefas conectadas através de um armazenamento compartilhado (*constraint store*)
Alma-0 *[APT 98]*	Imperativo e Lógico	Não enfoca	Mecanismos para suporte à não-determinismo e *backtracking* em linguagens imperativas
Holo *[HOL 2001]*	Objetos, Lógica e Imperativo	Entes distribuídos	Entes organizados em níveis hierárquicos e comunicação/sincronização implícita (*blackboard* lógico) e explícita (mensagens)

2.2 Modelos multiparadigma

Esta seção descreve e comenta sete modelos multiparadigma (OLI [LEE 97], OWB [AMA 96], I^+ [NGK 95], DLO [CIA 96], ETA [AMB 96], G [PLA 91], Oz [SMO 95]). Este estudo serve de base para a criação da taxonomia proposta na seção 2.3.

2.2.1 OLI (*Object Logic Integration*)

O modelo OLI [LEE 97] foi criado por pesquisadores das universidades de Hong Kong e Manchester. Este modelo propõe a integração da lógica e objetos. Um programa OLI é composto de duas partes: a parte da lógica e a parte dos objetos. A interface entre as partes é baseada na semântica das entidades usadas em cada paradigma. Na lógica, as entidades de programação constituem um Universo de Herbrand Enriquecido (*enriched Herbrand universe- eHu*). O *eHu* resulta do acréscimo de objetos ao universo de Herbrand. Estes objetos devem estar definidos na parte de objetos do programa OLI. No eHu encontram-se dois tipos de entidades: *o-terms* e *h-terms*. *H-terms* são termos comuns encontrados nos programas em lógica. Do ponto de vista semântico, *o-terms* também são termos. No entanto, o estado ou atributo de um *o-term* não está explícito na estrutura do termo. A obtenção dessas informações deve ser realizada através de mensagens ao invés de unificação. Por outro lado, do ponto de vista da orientação a objetos, todas as entidades são um conjunto de objetos. Os termos do universo de Herbrand também são considerados objetos. Em OLI, termos de Herbrand são instâncias de uma classe denominada *PTerm*. Essa classe prove métodos para manipulação das estruturas dos termos.

Comentário: o modelo OLI propõe a integração equilibrada dos paradigmas em lógica e orientado a objetos. A base da integração consiste na definição de uma interface que permita uma clara semântica do ponto de vista de ambos os paradigmas. Os paradigmas em lógica e orientado a objetos ficam inalterados. Existe apenas o acréscimo de algumas novas características em ambos, de forma a permitir a colaboração no desenvolvimento de software. Não é criada nenhuma nova entidade de modelagem. Dependendo da parte do programa, a entidade utilizada é o objeto (parte de objetos) ou o predicado lógico (parte da lógica).

2.2.2 OWB (*Objects With Brain*)

O modelo OWB [AMA 96] foi criado na Universidade Federal do Rio Grande do Sul. Este modelo propõe a integração dos paradigmas em lógica e orientado a objetos como suporte ao desenvolvimento de agentes. OWB integra objetos e lógica permitindo que um objeto tenha parte de seu conhecimento privado expresso como predicados lógicos. OWB estende a orientação a objetos com conceitos da lógica. Essa extensão é suportada por uma nova classe denominada *LogicKnowledge*. A implementação de OWB utiliza *Smalltalk-80* estendida com *Mei-Prolog*, uma versão de Prolog desenvolvida em *Smalltalk*. A orientação a objetos possui as vantagens da modularidade, herança e ocultamento de informações. No entanto, em algumas aplicações é necessária a procura dinâmica de soluções que o paradigma em lógica proporciona. Além disso, torna-se interessante a possibilidade de acréscimo de conhecimento na forma declarativa aos objetos.

Comentário: o principal objetivo de OWB consiste no suporte à programação orientada a agentes. A lógica é introduzida para representação de conhecimento dentro dos objetos. Sendo assim, OWB não possui nenhuma nova entidade. A modelagem é baseada em objetos apesar da possibilidade da existência de objetos com conhecimento privado expresso em forma de predicados lógicos. OWB suporta três entidades distintas: objetos, agentes e predicados lógicos.

2.2.3 I^+

O modelo I^+ [NGK 95] está sendo desenvolvido através de um trabalho conjunto entre a Universidade de Hong Kong e a Universidade de Toronto. Este modelo é uma evolução da linguagem multiparadigma I. O I^+ integra três paradigmas básicos: lógica, funcional e orientado a objetos. A integração desses paradigmas cria uma nova proposta de desenvolvimento de software denominada *programação declarativa orientada a objetos*. No I^+ os métodos não são procedimentos, mas sim, conjuntos de cláusulas ou funções. Dessa forma, os paradigmas declarativos (lógica e funcional) são incorporados nos métodos da orientação a objetos. O modelo permite dois tipos de classes, ou seja, classe lógica (*logic_class*) e classe funcional (*functional_class*). Estas classes diferem na forma com que seus métodos são definidos. Métodos na classe lógica são predicados lógicos e recebem o nome de métodos lógicos (*logic methods*). Por sua vez, métodos na classe funcional são funções e recebem o nome de métodos funcionais (*functional methods*).

Comentário: o I^+ integra os três paradigmas básicos envolvidos. No entanto, a base do modelo consiste na orientação a objetos. A introdução de novos paradigmas pode ser facilmente realizada através da criação de novas classes. O usuário organiza os sistemas em objetos, mas programa os algoritmos de forma declarativa (lógica ou funções). Sendo assim, o modelo suporta três entidades já existentes: objetos, predicados e funções.

2.2.4 DLO (*Distributed Logic Objects*)

O DLO [CIA 96] está sendo desenvolvido através de um trabalho conjunto entre a Universidade de Bologna e a Universidade de Ferrara, ambas localizadas na Itália. O modelo é baseado na união dos paradigmas em lógica e orientado a objetos. DLO estende o paradigma em lógica através de cláusulas de múltiplas cabeças, no entanto, mantém sua capacidade dedutiva e a leitura declarativa dos programas. Os programas DLO são transformados (técnicas de translação) em uma linguagem de programação em lógica concorrente denominada Rose. Essa linguagem possui uma implementação baseada na extensão da máquina abstrata Prolog criada por Warren [AIT 91]. Esta extensão consiste na criação de novas instruções e estruturas de dados que suportem unificação distribuída, criação de processos, comunicação e controle do não-determinismo.

Comentário: o modelo DLO propõe a unificação dos paradigmas em lógica e orientado a objetos. A unificação é suportada através de uma nova entidade de modelagem denominada **Objeto Lógico**. Esta entidade é criada com a aplicação de conceitos da orientação a objetos e através da extensão da lógica com cláusulas de múltiplas cabeças.

2.2.5 ETA (*Everything buT Assignment*)

O modelo ETA [AMB 96] foi proposto por membros do Departamento de Informática da Universidade de Pisa, Itália. Este modelo possui como base a união dos paradigmas em lógica e orientado a objetos. ETA utiliza múltiplos espaços de tuplas como suporte para essa unificação. Nesta proposta, um sistema é composto de um conjunto de espaços de tuplas. As tuplas podem ser enviadas de um espaço para outro. ETA controla o fluxo de tuplas entre objetos (espaços de tuplas) através da unificação. Em complemento, a herança em ETA é baseada na definição de novos espaços de tuplas pela extensão de interfaces e pela redefinição de parte das regras que especificam o comportamento dos objetos. Merece destaque ainda que em ETA o recebimento de uma mensagem faz com que a tupla recebida seja adicionada ao estado do objeto (espaço de tupla). Este comportamento difere da maioria das linguagens orientadas a objeto onde o recebimento de uma mensagem faz com que seja realizada uma operação ou invocado um procedimento.

Comentário: o modelo ETA utiliza a orientação a objetos como estrutura básica para o desenvolvimento de software. ETA possui uma sintaxe semelhante à programação em lógica para codificação dos objetos. A filosofia de múltiplos espaços de tuplas é utilizada como suporte para o fluxo e organização dos dados. Sendo assim, a entidade utilizada é o objeto codificado com sintaxe ETA, ou seja, o **Objeto ETA**. Este tipo de objeto é diferente do objeto lógico (DLO) e do objeto utilizado no paradigma orientado a objetos. Ele possui sintaxe própria criada pelo modelo ETA.

2.2.6 G

O modelo multiparadigma G [PLA 91] foi proposto na tese de doutorado de John Placer desenvolvida na Universidade de Oregon. O modelo G suporta quatro paradigmas básicos: paradigma em lógica, funcional, imperativo e orientado a objetos. G foi projetada para o estudo de estruturas sintáticas e semânticas que suportem a integração de paradigmas básicos. A principal estrutura utilizada em G é a *stream*. Todos os valores são considerados *streams.* Os tipos escalares, tais como *int*, *real*, *type*, *char* e *string* são interpretados como *streams* de um único elemento. Além disso, existem dois tipos compostos denominados *stream* e *relation*. Um tipo composto é uma *stream* que suporta um número variável de elementos. Placer [PLA 91] descreve como o conceito de *stream* e suas qualidades (ambiente, seqüência de valores e protocolo de enumeração) permitem a criação de uma semântica unificada para compreensão das estruturas de dados e estruturas de controle da linguagem G.

Comentário: o modelo não utiliza um paradigma como base, no entanto, suporta os paradigmas em lógica, imperativo, funcional e orientado a objetos. Por outro lado, G estabelece a estrutura *stream* e suas características como sendo a base sintática e semântica para criação de programas. Sendo assim, a entidade de G é a ***stream***.

2.2.7 Oz

O modelo Oz [SMO 95] foi criado no Centro Germânico de Pesquisa em Inteligência Artificial (DFKI), Alemanha. Este modelo é baseado na linguagem AKL [JAN 94] criada no Instituto Sueco de Ciência da Computação. Seus criadores afirmam que Oz substitui linguagens de alto nível sequenciais, tais como, Lisp, Prolog e Smalltalk. A computação em Oz ocorre em um espaço computacional onde existem tarefas conectadas através de um armazenamento compartilhado. A computação avança através da redução de tarefas. A redução de uma tarefa manipula o armazenamento e cria novas tarefas. Uma tarefa desaparece quando reduzida. O armazenamento compartilhado é denominado *constraint store*. A *constraint store* suporta o armazenamento de informações parciais sobre variáveis. Além disso, uma informaçção armazenada nunca se perde (monotônica). Através da *constraint store* é criado um mecanismo declarativo de sincronização entre tarefas. Oz suporta ainda comunicação através de portas (procedimento ligado a uma *stream*) [JAN 93]. Nos últimos anos vários trabalhos de pesquisa vêm sendo desenvolvidos para criação de uma plataforma distribuída do modelo Oz (Mozart [ROY 97, HAR 98, HAR 99, ROY 2001]). Neste escopo estão sendo pesquisados diversos tópicos, tais como, objetos móveis em Oz distribuído [ROY 97] e a utilização de variáveis lógicas em sistemas distribuídos [HAR 99].

Comentário: os criadores de Oz propõem um novo modelo baseado nos conceitos dos paradigmas existentes. O principal conceito para compreensão de Oz é o espaço computacional. As tarefas que reduzidas durante a computação são consideradas unidades de processamento (procedimentos, processos, *threads*, objetos). Sendo assim, a entidade de Oz é a **tarefa** executada no espaço computacional.

2.3 Taxonomia multiparadigma

As propostas multiparadigma buscam a criação de um modelo mais amplo e eficiente, através da união de paradigmas básicos. O modelo criado procura o mantenimento das características benéficas dos paradigmas unificados e a eliminação de suas deficiências. Os paradigmas básicos suportam a modelagem através de comandos imperativos, funções, predicados lógicos, objetos e agentes.

A criação de modelos multiparadigma constitui uma abordagem *bottom-up*, buscando modelos mais genéricos pela unificação de modelos mais específicos. Este processo é natural, pois os modelos mais específicos já existem e servem de base para os novos modelos a serem criados. Os modelos multiparadigma podem ser classificados pela eficiência com que atingem os objetivos da abordagem *bottom-up*. Essa eficiência pode ser medida de acordo com dois índices: número de paradigmas básicos unificados (quantidade) e semântica da unificação (qualidade). O primeiro índice estabelece a amplitude do modelo, ou seja, quantos paradigmas básicos ele abrange? O segundo índice estabelece a qualidade da unificação, ou seja, o quanto a semântica do novo modelo torna implícita a semântica dos paradigmas unificados? Esse índice estabelece o quanto os paradigmas unificados ainda estão visíveis na nova proposta. O primeiro índice é um número inteiro. O segundo índice é mais subjetivo e possui três níveis:

- **Multiparadigma de Integração (MI)**: o modelo não unifica os paradigmas, apenas realiza a integração. Os paradigmas estão presentes, visíveis e independentes. O modelo suporta algum nível de interação entre as entidades dos paradigmas integrados. O desenvolvimento de software envolve o uso de entidades distintas. Normalmente, as linguagens de programação de um paradigma básico suportam a chamada de rotinas desenvolvidas em outro. Este é um caso clássico de modelo MI. Nenhum dos modelos apresentados na segunda seção é MI;

- **Multiparadigma de Unificação Parcial (MUP)**: o modelo unifica parcialmente os paradigmas. Os paradigmas estão presentes, visíveis, mas dependentes. Durante a modelagem são utilizadas entidades distintas. Sendo assim, ainda sobrevivem múltiplas semânticas. Apesar disso, o modelo cria uma semântica multiparadigma de unificação parcial (semântica MUP) que transcende a semântica dos paradigmas unificados. Essa semântica é a base da unificação. Em um modelo MUP, normalmente um paradigma serve de base e os demais são complementares. Entre as propostas apresentadas na seção 2.2, são MUP os modelos OLI, OWB e I^+;

- **Multiparadigma de Unificação Total (MUT)**: o modelo unifica completamente os paradigmas. Os paradigmas estão presentes através de suas características, mas não são visíveis. Durante o desenvolvimento é utilizada apenas uma entidade denominada entidade de unificação total (entidade MUT). O modelo cria uma semântica multiparadigma de unificação total (semântica MUT). Entre os modelos descritos na segunda seção são propostas MUT: DLO, ETA, G e Oz.

Os dois índices (quantidade e qualidade) devem ser aplicados em conjunto para descrição completa de um modelo multiparadigma. Por exemplo, o I^+ é um modelo MUP 3, pois unifica parcialmente três paradigmas (paradigma em lógica, funcional e orientado a objetos). No caso dos modelos MUT, o número de paradigmas unificados é determinado pela apresentação da proposta. Normalmente, os autores salientam em publicações o número de paradigmas suportados. Além disso, os autores costumam demonstrar a aplicação de seus modelos em problemas clássicos dos paradigmas básicos (veja o texto [MUL 95] para o modelo Oz e o texto [PLA 91] para G).

A tabela 2.2 apresenta a classificação dos modelos descritos na seção 2.2. A tabela possui oito colunas: nome do modelo, nível de unificação, quantidade de paradigmas unificados, paradigmas unificados, paradigma que serve de base para a unificação, semântica utilizada como suporte à unificação, entidade do modelo e classificação segundo a taxonomia multiparadigma. Os modelos estão listados na mesma ordem em que foram descritos. Além disso, na quarta linha da tabela é apresentada a classificação do Holoparadigma.

TABELA 2.2 – Classificação de modelos multiparadigma

Mod.	**Nível**	**Quant.**	**Paradigmas Unificados**	**Paradigma Base**	**Semântica de Unificação**	**Entidade**	**Classif.**
OLI	Parcial	2	Orientação a Objetos e Lógica	Não possui	Mapeamento Objeto → Lógica (Classe *Pterm*) e Lógica → Objeto (*Enriched Herbrand Universe - eHu*)	Objeto e Predicado Lógico	MUP 2
OWB	Parcial	3	Orientação a Objetos, Orientação a Agentes e Lógica	Orientação a Objetos	Suporte à criação de agentes através da inserção de lógica em objetos (classe *LogicKnowledge*)	Objeto, Agente e Predicado Lógico	MUP 3
I^+	Parcial	3	Orientação a Objetos, Lógica e Funcional	Orientação a Objetos	Programação Declarativa Orientada a Objetos, ou seja, especificação de objetos através da lógica (*logic_class*) e funções (*functional_class*)	Objeto, Predicado Lógico e Funções	MUP 3
Holo	Parcial	3	Imperativo, Orientação a Objetos e Lógica	Orientação a Objetos	Entes organizados em níveis hierárquicos e comunicação/ sincronização implícita (*blackboard* lógico) e explícita (mensagens)	Ente	MUP 3
DLO	Total	2	Orientação a Objetos e Lógica	Não possui	Processos organizados em Objetos Lógicos implementados através de Cláusulas de Múltiplas Cabeças	Objeto Lógico	MUT 2
ETA	Total	2	Orientação a Objetos e Lógica	Não possui	Múltiplos Espaços de Tuplas	Objeto ETA	MUT 2
G	Total	4	Imperativo, Orientação a Objetos, Lógica e Funcional	Não possui	Universo de *Streams*	*Stream*	MUT 4
Oz	Total	4	Imperativo, Orientação a Objetos, Lógica e Funcional	Não possui	Espaço Computacional, ou seja, tarefas conectadas através de um armazenamento compartilhado	Tarefa	MUT 4

2.4 Modelos multiparadigma distribuídos

Esta seção descreve a exploração da distribuição em três modelos multiparadigma: I^+ [NGK 95], DLO [CIA 96] e Oz [HAR 98].

2.4.1 I^+

Kam e Chi [NGK 95] descrevem duas formas para exploração do paralelismo em I^+: paralelismo **inter-objetos** e **intra-objetos**. No âmbito do paralelismo inter-objetos são utilizados dois modos para invocação de métodos (mensagens): síncrono e assíncrono. No modo síncrono, após o envio de uma mensagem, o objeto origem aguarda a resposta do objeto destino. Em I^+ o envio de uma mensagem síncrona possui a sintaxe:

..., O::p, G,...

A meta *G* não será executada até que o resultado da chamada do método *p* no objeto *O* seja obtido. Em complemento, no modo assíncrono o objeto prossegue sua execução logo após o envio da mensagem. Quando a resposta é recebida, seu conteúdo é armazenado em uma estrutura de dados interna do objeto origem. A tentativa de acesso à resposta pode ser realizada em qualquer momento após o envio da solicitação. No entanto, se a resposta ainda não está disponível, o objeto origem aguarda sua chegada. A sintaxe de uma chamada assíncrona em I^+ é a seguinte:

..., O>>m(X), A, B, O??m(X), c(X), ...

Neste caso, a invocação do método *m* no objeto *O* é assíncrona (representada pelo símbolo >>). O ponto de sincronismo ocorre após a execução das metas *A* e *B*. Portanto, ambas serão executadas em paralelo com o objeto *O*. A resposta é obtida com o uso do operador ***??***. Após o recebimento da resposta, a meta *c* será executada utilizando o valor *X* obtido durante a execução do método *m*. Afirmam os criadores de I^+ que a decomposição de uma chamada de método em duas fases (requisição e resposta) estimula a execução paralela de atividades em múltiplos objetos. Afirmam ainda que, os modos síncrono e assíncrono podem ser utilizados para a chamada de métodos em lógica ou funcionais.

A execução de um método é denominada atividade do objeto. O relacionamento entre atividades e objeto é análoga ao relacionamento entre processos e processador em um ambiente multitarefa. Desta forma, um objeto pode ser percebido como um processador virtual conforme apresentado na figura 2.1. A estrutura mostrada na figura suporta o escalonamento de múltiplas atividades dentro do objeto.

I^+ suporta dois tipos de objetos: objetos passivos e objetos ativos. Os objetos passivos são o padrão do I^+ e somente são ativados quando um dos seus métodos é invocado. O grau de concorrência pode ser aumentado com o uso de objetos ativos. Este tipo de objeto executa suas próprias atividades após a criação. Sendo assim, o objeto não necessita ser ativado por outro objeto. Em I^+ um objeto é ativado pela inclusão de um método de inicialização com o mesmo nome da classe que o define.

A implementação de I^+ foi realizada em duas fases. Na primeira foi construído um protótipo para execução em uma rede de estações de trabalho DEC/ULTRIX. Como ferramentas para construção do protótipo foram utilizadas as linguagens de programação *C*, *Quintus Prolog* e *Lazy ML* (LML). O protótipo possui dois componentes principais: um conversor *I^+/Prolog* (converte as definições de classes em lógica para grupos de módulos em Prolog) e um conversor *I^+/LML* (converte as definições de classes funcionais para grupos de módulos LML). A troca de mensagens é implementada em *sockets* padrão 4.3 BSD. Em complemento, utiliza-se o protocolo TCP/IP. Na segunda fase de construção do I^+, os conversores foram reescritos para geração de programas na linguagem C. Essa mudança foi motivada pela busca de eficiência na execução. Em complemento, foi desenvolvido um ambiente de programação baseado nos padrões X11R5 e Motif. Este ambiente executa em estações de trabalho UNIX.

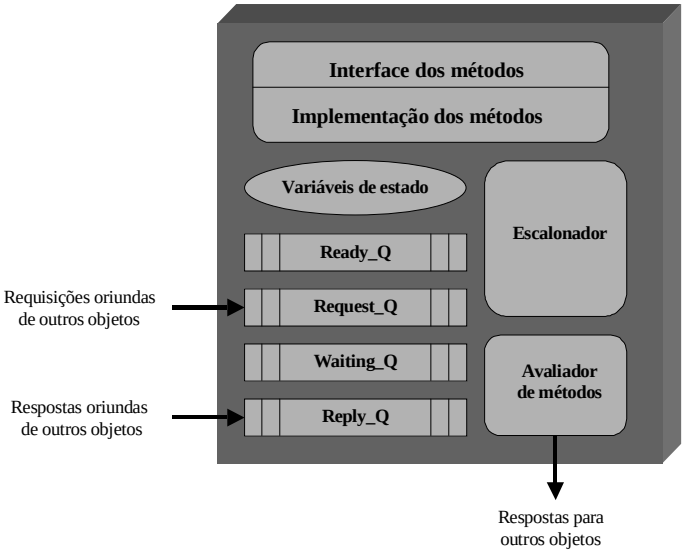


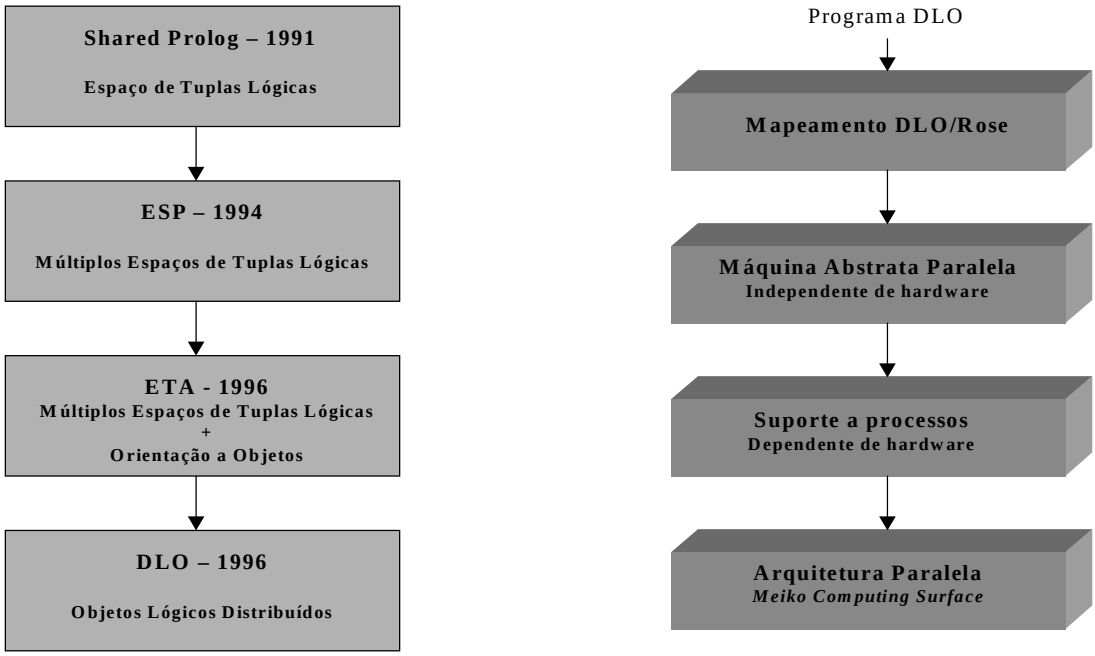
FIGURA 2.1 – Objeto como processador virtual

2.4.2 DLO (*Distributed Logic Objects*)

A evolução da proposta DLO pode ser percebida em quatro etapas. A figura 2.2a mostra que em cada uma das etapas foi criado um modelo com características próprias. Inicialmente, Brogi e Ciancarini [BRO 91] propuseram a primeira linguagem do paradigma em lógica baseada na noção de *Espaço de Tuplas Lógicas* (ETL). Essa linguagem foi denominada *Shared Prolog* (SP). Um sistema em *Shared Prolog* é composto por um conjunto de tarefas paralelas. Estas tarefas são programas *Prolog* estendidos com mecanismos de guarda. Os mecanismos de sincronização e comunicação entre tarefas são baseados no modelo *blackboard*. Este modelo é estendido com o mecanismo de unificação clássico do paradigma em lógica.

Na segunda etapa, Ciancarini [CIP 94] propõe uma extensão para o *Shared Prolog*, denominada ESP (*Extended Shared Prolog*). ESP suporta *Múltiplos Espaços de Tuplas*. Este suporte é baseado em um novo modelo de programação denominado *PoliS*, o qual estende Linda [CAR 86] com o suporte de múltiplos espaços de tuplas.

Na terceira etapa surgiu o modelo ETA [AMB 96]. Neste escopo, encontra-se a primeira referência entre união dos paradigmas em lógica e orientado a objetos usando a noção de múltiplos espaços de tuplas. Em complemento, durante a descrição do modelo torna-se clara a preocupação dos autores quanto a exploração do paralelismo.



(a) Evolução da proposta DLO (b) Implementação de DLO

FIGURA 2.2 – Evolução e implementação do DLO

Finalmente, na quarta etapa surgiu o modelo DLO [CIA 96]. No âmbito do modelo DLO são desenvolvidas as principais atividades relacionadas com a exploração do paralelismo. Os programas em DLO são convertidos para uma linguagem concorrente de programação em lógica denominada Rose. Nesta linguagem a comunicação entre processos é baseada em cláusulas de múltiplas cabeças. Além disso, durante a exploração do paralelismo E não podem ser compartilhadas variáveis (paralelismo E independente). Rose é implementada em uma arquitetura paralela baseada na tecnologia de *transputers*. A plataforma de execução é baseada em uma extensão da máquina abstrata de Warren [AIT 91].

DLO suporta dois tipos de paralelismo:

- **paralelismo inter-objetos**: instâncias de objetos podem ser executadas em paralelo. A exploração inter-objetos possui relação com o paralelismo E suportado pelo paradigma em lógica;
- **paralelismo intra-objetos**: diferentes métodos podem ser executados em paralelo. Os métodos executados em paralelo não podem alterar o valor de uma mesma variável. A exploração intra-objetos possui relação com o paralelismo OU.

A figura 2.2b mostra a implementação de DLO organizada em quatro blocos:

- **mapeamento DLO/Rose**: o processo de tradução converte nomes de objetos em variáveis lógicas. Em complemento, a unificação é utilizada como suporte à troca de mensagens. Um objeto é particionado em um número de metas paralelas em Rose;

- **máquina abstrata paralela**: a execução paralela é baseada na Máquina Abstrata Paralela Rose estendida com suporte ao DLO. Esta máquina é uma extensão da WAM [AIT 91] contendo novas instruções e estruturas de dados com suporte à unificação de múltiplas cabeças, criação de processos, comunicação e controle do não-determinismo;
- **suporte a processos**: imediatamente acima da arquitetura paralela existe uma camada de software, a qual fornece mecanismos para gerenciamento de processos e suporte à comunicação;
- **arquitetura paralela**: o suporte à execução de programas DLO foi implementado em uma arquitetura paralela com memória distribuída denominada *Meiko Computing Surface*. Este equipamento é baseado na tecnologia de *transputers*. Cada nodo da máquina é composto de um processador *transputer* baseado nos conceitos RISC, um co-processador de ponto-flutuante e uma memória RAM estática.

2.4.3 Mozart (*Distributed Oz*)

No ano de 1995, Smolka, Schulte e Van Roy propuseram, através do projeto Perdio [SMO 95a], a criação de uma plataforma de suporte à execução distribuída da linguagem Oz. Esta plataforma é denominada Mozart [ROY 97, ROY 2001].

Mozart possui dois princípios básicos [ROY 97]:

- define a linguagem através de duas semânticas: *semântica da linguagem* e *semântica de distribuição* (regulamenta a execução distribuída);
- incorpora a mobilidade [ROY 97, IEE 98, LAN 98, FER 99, FER 2001a] na semântica distribuída da linguagem.

O Mozart estende a semântica do Oz 2.0 criando uma semântica distribuída que suporta o controle da mobilidade e o posicionamento na rede. Através dessa extensão, as operações de rede tornam-se previsíveis, permitindo ao programador o gerenciamento da comunicação. Existem quatro requisitos considerados básicos para o projeto do Mozart:

- **transparência de rede**: o comportamento do programa é o mesmo independentemente da distribuição na rede. Esse requisito possui como base a criação de um *Espaço Computacional Compartilhado Distribuído* (ECCD). O ECCD cria a ilusão de um único espaço de endereçamento para todas as entidades do sistema. A distinção entre referências locais e remotas é imperceptível para o usuário;
- **previsibilidade de comunicação**: as entidades da linguagem suportam padrões simples de comunicação na rede. Estes padrões permitem a previsão do comportamento da comunicação durante a execução;
- **tolerância à latência**: a influência dos atrasos de comunicação na rede é reduzida ao mínimo;
- **segurança na linguagem**: o processamento e os dados são protegidos entre diferentes usuários que utilizam a plataforma. Neste sentido, o programador possui meios para restringir o acesso aos dados. A segurança é uma das principais características de Mozart [HAR 98].

Os programas em Mozart são transformados para comandos da linguagem de *kernel* conhecida como OPM (*Oz Programming Model* [ROY 97]). Nesta linguagem destacam-se dois aspectos: o sincronismo baseado em fluxo de dados (*dataflow*) controlado por variáveis lógicas e a unificação de funções de alta ordem com programação orientada a objetos.

A organização do Mozart é apresentada na figura 2.3. O Mozart é composto de dois módulos: o compilador Oz e o ambiente de execução. O compilador recebe um programa Oz e gera um programa em *byte code*. Por sua vez, o *byte code* é entregue ao ambiente responsável pela execução distribuída do programa. Cada nodo da rede possui um ambiente Mozart. O usuário usufrui da distribuição de forma transparente e a execução é baseada no espaço computacional compartilhado distribuído. Vários usuários podem compartilhar a utilização do ambiente.

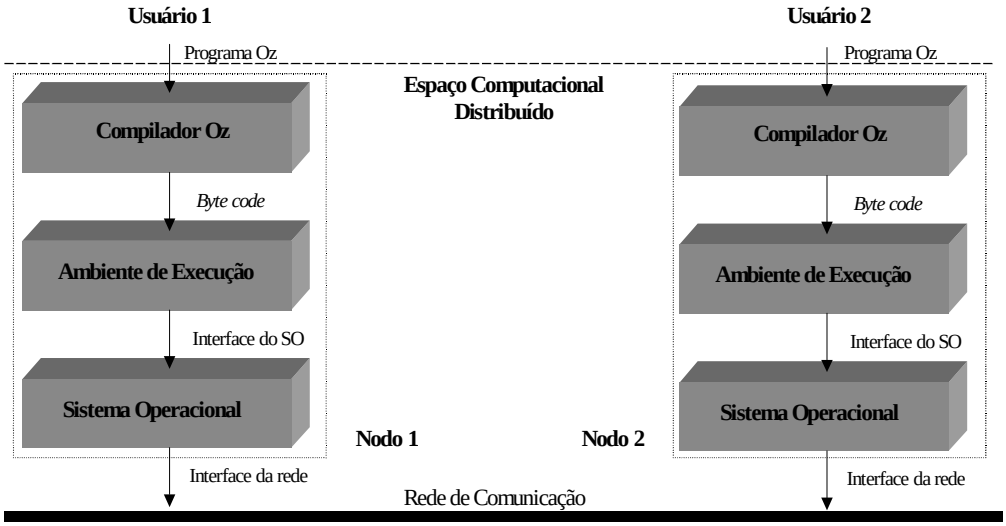


FIGURA 2.3 – Organização básica do Mozart

A figura 2.4 apresenta uma visão interna do ambiente de execução. A distribuição foi integrada de forma conservativa à plataforma centralizada (emulador Oz 2.0). A extensão não afeta o desempenho do suporte centralizado. Este suporte executa as operações locais.

O ambiente de execução distribuída é composto de três camadas:

- **grafo da linguagem**: operações em entidades consideradas distribuídas são tratadas pela camada do grafo da linguagem. Esta camada implementa a semântica distribuída para as entidades do Oz;
- **gerenciamento de memória**: basicamente possui três tarefas. A primeira é o suporte ao ECCD. Neste sentido, são utilizados endereços locais e globais (acesso a nodos remotos). A segunda é a construção e gerenciamento automático de estruturas de acesso. Estas estruturas são utilizadas quando entidades são referenciadas remotamente. A terceira é a coleta de lixo distribuída, a qual é composta de um coletor local e um mecanismo distribuído para gerenciamento de endereçamento global;
- **suporte a mensagens**. implementa a transferência de seqüências de bytes entre nodos.

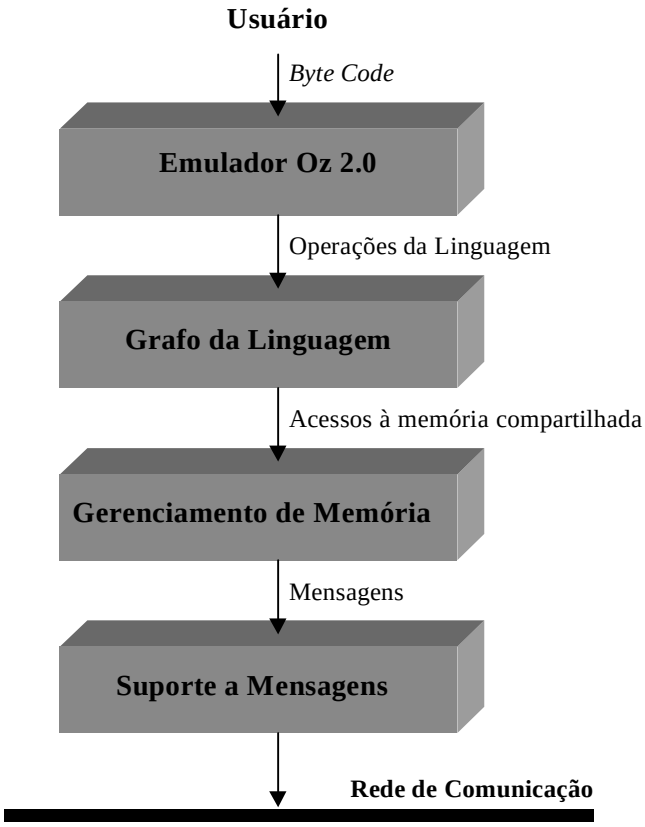


FIGURA 2.4 – Ambiente de execução do Mozart

2.5 Considerações finais

Este capítulo apresentou um estudo sobre software multiparadigma. Uma discussão relacionada com o estado da arte no contexto do Holoparadigma permitiu uma introdução ao tema. Logo após, foram descritos e analisados sete modelos multiparadigma. Em complemento, foi proposta uma taxonomia que possibilitou a organização e o aprofundamento do estudo. A taxonomia foi aplicada na classificação dos modelos analisados. Finalmente, foi discutida a exploração do paralelismo e/ou distribuição em três modelos. As principais conclusões alcançadas neste capítulo são:

- os modelos multiparadigma possuem como base os paradigmas básicos. Estes paradigmas estão presentes com maior ou menor intensidade dependendo do modelo;
- os modelos multiparadigma, da mesma forma que os paradigmas básicos, possuem entidades de modelagem. Um modelo multiparadigma pode utilizar as entidades dos paradigmas que serviram para sua criação ou pode propor uma nova entidade de unificação;
- alguns modelos utilizam uma abordagem semântica como suporte à proposta multiparadigma. Por exemplo, objetos lógicos distribuídos no modelo DLO, múltiplos espaços de tuplas em ETA e espaço computacional em Oz;

- a entidade criada em um modelo multiparadigma é mais genérica do que as entidades unificadas. Sendo assim, existe uma tendência à diminuição de sua eficiência na modelagem dos paradigmas unificados. Este é o principal desafio na criação de novas entidades;
- algumas propostas multiparadigma priorizam a exploração do processamento paralelo e distribuído. Os modelos I^{+}, DLO, ETA e Oz possuem esse enfoque.

Trabalhos futuros poderão aperfeiçoar o estudo apresentado neste capítulo. Existe uma grande variedade de modelos multiparadigma. A classificação de outros modelos permitirá a expansão da tabela 2.2. Além disso, novas taxonomias multiparadigma podem ser criadas.

3 Holoparadigma

Este capítulo apresenta o Holoparadigma (de forma abreviada, Holo). Holo é um modelo multiparadigma que possui uma semântica simples e distribuída. Através dessa semântica, o modelo estimula a exploração automática da distribuição (distribuição implícita). O estímulo à distribuição implícita é seu principal objetivo. Os princípios gerais do modelo foram publicados em [BAR 99a], [BAR 2000a], [BAR 2000e] e [BAR 2001]. Além disso, os aspectos específicos relacionados com paralelismo e distribuição foram analisados em [BAR 2000c] e [BAR 2001b]

3.1 Gênese do Holoparadigma

A criação do Holoparadigma pode ser percebida em três níveis de temas de pesquisa (figura 3.1): básicos, intermediários e final. Cada nível abrange os temas de pesquisa abordados durante a gênese. O primeiro nível contém os temas básicos. Neste nível, cada par de temas origina um tema intermediário no segundo nível. Por sua vez, os temas intermediários servem de base para o surgimento do Holoparadigma. Essa seção discute os dois primeiros níveis.

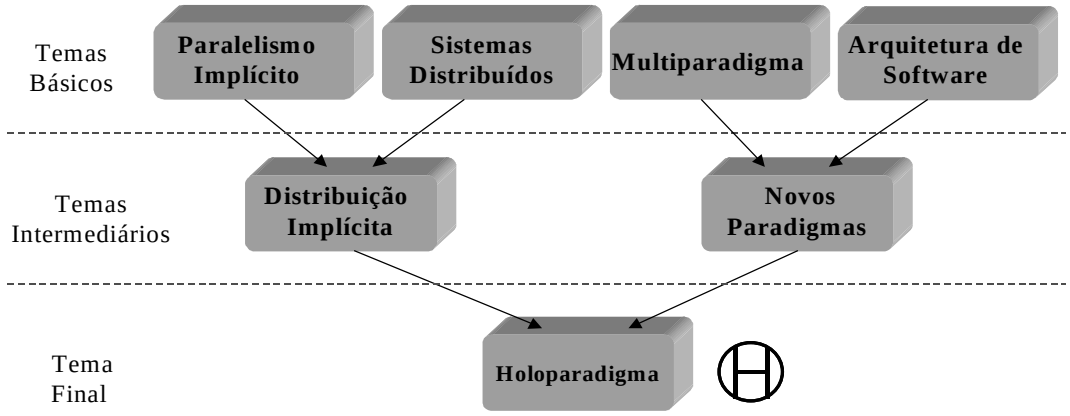


FIGURA 3.1 – Gênese do Holoparadigma

O **paralelismo implícito** propõe a detecção automática do paralelismo e sua exploração através de mecanismos inseridos no software básico dos sistemas computacionais. Diversos estudos mostram que a exploração do paralelismo implícito em paradigmas declarativos é mais simples do que sua exploração no paradigma imperativo [BAR 96, GEY 99, BAR 2000b, VAR 2000, OPE 2001, APE 2001]. Por sua vez, nos últimos anos os **sistemas distribuídos** têm recebido cada vez mais atenção tanto dos centros de pesquisa quanto das empresas. Entre os novos tópicos de pesquisa merecem destaque: uso de redes de computadores como arquiteturas paralelas [JOU 97, IEE 98], mobilidade de código e dados [ROY 97, IEE 98, LAN 98, FER 99, FER 2001a] e tratamento da heterogeneidade de hardware nas redes de computadores [CAB 97, BAR 2001b]. Uma análise da situação atual e das perspectivas futuras permite a previsão de que em breve os sistemas computacionais serão compostos por uma grande variedade de estações de alto desempenho conectadas por redes de alta velocidade organizadas em topologias diversas (**arquiteturas distribuídas**, o anexo 1.1 propõe uma classificação para estas arquiteturas). Neste contexto, torna-se interessante a integração dos temas paralelismo implícito e sistemas distribuídos em um tópico de pesquisa denominado **distribuição implícita** (tema intermediário). Este tópico busca a exploração automática da distribuição através de mecanismos inseridos no software

básico. No âmbito da distribuição implícita devem ser considerados temas que não fazem parte dos estudos no paralelismo implícito, tais como, tratamento automático da mobilidade e heterogeneidade de hardware.

O tema **multiparadigma** propõe a criação de modelos de desenvolvimento de software através da integração de paradigmas básicos. Além disso, na medida em que aumentam o tamanho e a complexidade dos programas, o projeto e a especificação da estrutura dos sistemas tornam-se mais importantes do que a escolha de algoritmos e estruturas de dados [SHA 96]. Entre os tópicos relacionados com o projeto estrutural de sistemas destacam-se: organização do sistema em componentes; protocolos de comunicação, sincronização e acesso de dados; distribuição física dos componentes; desempenho e evolução dos sistemas. Neste contexto, surge a **arquitetura de software** [IEE 95, GAR 95, SHA 96]. Este tema de pesquisa dedica-se à descrição de componentes, as interações entre eles e os padrões que guiam sua composição. Os temas multiparadigma e arquitetura de software podem ser unidos em uma única abordagem de pesquisa dedicada à criação de **novos paradigmas** de software (tema intermediário). Um paradigma orienta todo o desenvolvimento de sistemas, desde a modelagem até a implementação. A semântica do paradigma permea todos os instrumentos utilizados na criação de sistemas. A unificação dos temas de pesquisa no nível intermediário conduz à criação do **Holoparadigma** (tema final).

3.2 Princípios filosóficos

Desde seus primórdios a humanidade busca a compreensão do mundo em que está inserida. Neste sentido, nos últimos séculos a ciência vem alcançando contínuos sucessos. Em grande parte, o sucesso da ciência basea-se na estratégia de fragmentação do conhecimento, ou seja, na criação de diversas áreas de especialização. A fragmentação é uma solução natural para resolução de problemas complexos. No entanto, na medida em que a ciência estimula a fragmentação como instrumento de compreensão, estimula também a visão fragmentária como paradigma (modelo, padrão) do universo. Thomas Kuhn [KUH 70, KUH 77] discute a importância dos paradigmas no âmbito da ciência.

Albert Einstein no discurso pronunciado por ocasião do sexagésimo aniversário de Max Planck analisa os princípios da pesquisa. Naquela abordagem Einstein salienta que, no âmbito da física, a extrema nitidez, a clareza e a certeza só se adquirem à custa de um imenso sacrifício, ou seja, a perda da visão de conjunto da realidade [EIN 81, p.139]. Por sua vez, Ortega y Gasset [ORT 92, p.72], analisando o mesmo tema, salienta que todo esforço criador é uma especialização. No entanto, salienta ainda a necessidade da criação de sínteses do conhecimento. O autor defende que o movimento que leva a investigação a dissociar-se indefinidamente em problemas particulares exige uma regulagem compensatória. Essa regulagem deve ser realizada através de um movimento de direção inversa que contraía e retenha a tendência centrífuga da ciência. Merece destaque ainda, a abordagem de Ortega y Gasset, sob o título “A barbárie da especialização”, que enfoca o impacto da especialização/fragmentação do conhecimento nas distorções psicológicas do homem do século XX [ORT 87]. Torna-se interessante ressaltar ainda, que tanto Einstein como Ortega y Gasset defenderam idéias holísticas também no âmbito da política. Ambos foram ativos defensores da criação da união européia, a qual gradualmente torna-se uma realidade. Neste escopo, as idéias de Einstein podem ser encontradas no artigo intitulado *Pan-Europa* [EIN 95]. Por sua vez, Ortega y Gasset iniciou a análise dessa questão na década de 30 [ORT 87] e complementou suas opiniões na década de 60 [ORT 85].

David Bohm afirma que a ciência carece de uma visão de mundo não-fragmentária [BOH 80, p.12]. Afirma Bohm, que a abordagem que analisa o mundo em partes independentes não funciona bem na física moderna. Ele salienta que tanto na teoria da relatividade como na teoria quântica, noções que impliquem a totalidade indivisa do universo proporcionam um modo mais ordenado para a natureza da realidade. Durante o primeiro capítulo do livro "A Totalidade e a Ordem Implicada" [BOH 80], o autor compara a fragmentação e a totalidade do ponto de vista de paradigmas para compreensão do universo. No âmbito desse tema, salientam-se ainda os diálogos publicados por Weber [WEB 86], em especial, os textos "A Ordem Implícita e a Ordem Superimplícita" [BOH 86] e "Criatividade: a assinatura da natureza" [BOH 86a]. Vários outros textos abordam a questão fragmentação *versus* totalidade [CAP 82, WIL 82, TAL 91].

Na ciência da computação o paradigma fragmentação/especialização vem sendo bastante utilizado. O desenvolvimento de sistemas computacionais destaca a fragmentação como instrumento de domínio da complexidade. A máxima "dividir para conquistar" atua como um arquétipo para a engenharia de software. A fragmentação /especialização vem estimulando a criação de diversos paradigmas de desenvolvimento de software e, em especial, a criação de uma quantidade enorme de linguagens de programação [BAR 98]. A diversidade gerada pela proliferação de paradigmas e linguagens é simbolizada pela "torre de babel". Este símbolo pode ser encontrado na capa de todas as revistas *Computer Languages* e no título do artigo *From Babbage to Babel and Beyond: A Brief History of Programming Languages* [FRI 92].

Ortega e Gasset [ORT 92, p.72] afirma que a tendência centrífuga da ciência naturalmente cria uma fragmentação do conhecimento e defende a regulagem dessa tendência através da criação de sínteses e sistematizações. Na ciência da computação os temas multiparadigma e arquitetura de software atuam como sínteses em duas áreas de conhecimento: linguagens de programação e modelagem de software. No âmbito do Holoparadigma, ambas são sintetizadas em apenas uma frente de estudos, ou seja, novos paradigmas. Além disso, a distribuição implícita também faz parte da síntese. A regulagem da fragmentação do conhecimento no âmbito da ciência da computação é um importante estímulo para a criação do Holoparadigma.

3.3 Holosemântica

Um paradigma suporta a modelagem de sistemas computacionais. Com esse intuito, estabelece um conjunto de abstrações que serão utilizadas para criação dos modelos. A eficiência do paradigma está relacionada com a distância entre o significado dessas abstrações e os conceitos existentes no domínio modelado. No âmbito da engenharia de software, essa distância é denominada ***gap semântico*** (figura 3.2). Um pequeno *gap* semântico significa que as abstrações são adequadas.

A semântica está relacionada com o significado. No caso dos paradigmas, a semântica estabelece o significado de suas abstrações. Por exemplo, o paradigma orientado a objetos possui como principal abstração o objeto. No âmbito deste paradigma, objeto é a unidade de modelagem e significa qualquer realidade que possa ser organizada em uma unidade que encapsule atributos e métodos [GHE 98, p.287; SEB 99, p.436].

Esta seção introduz informalmente a semântica do Holoparadigma. As próximas duas seções complementam esta abordagem. Por sua vez, a seção 3.6 cria uma equivalência entre os principais conceitos do Holoparadigma e o **Cálculo de Ambientes** [CAL 97, CAL 98, CAL 99, CAL 99a, CAL 99b, CAL 2000, CAL 2000a, CAL 2001].

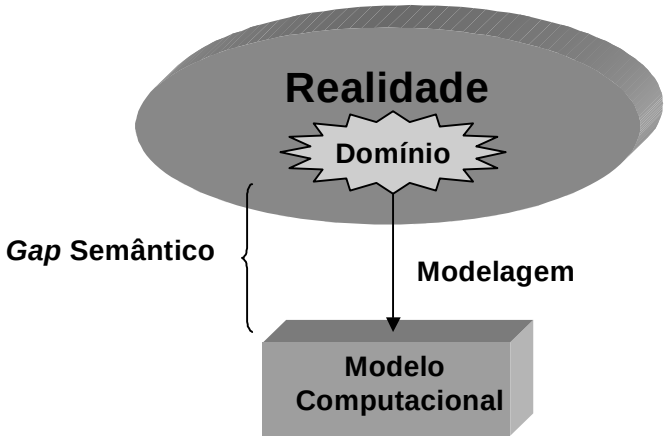


FIGURA 3.2 – *Gap* semântico

A semântica do Holo é denominada Holosemântica. A Holosemântica estabelece a utilização de duas unidades de modelagem:

- **unidade de existência - Ente:** a existência é modelada através de um ente;
- **unidade de informação - Símbolo:** a informação é modelada através de símbolos.

A modelagem em Holo utiliza somente essas unidades. Sendo assim, a criação de modelos computacionais é simplificada. A figura 3.3 mostra a aplicação da Holosemântica.

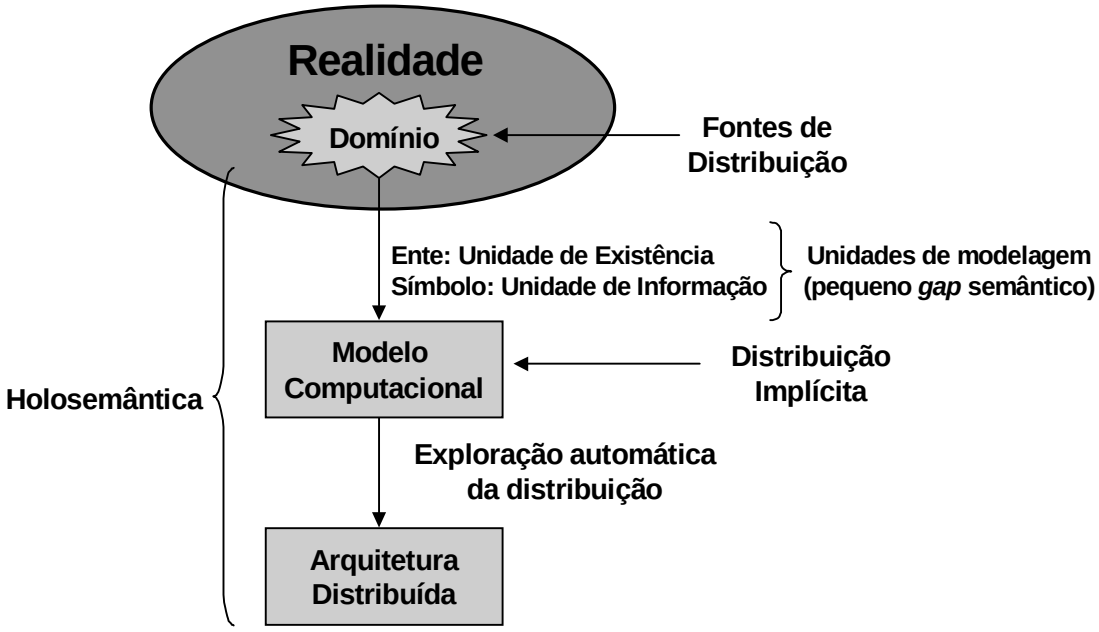


FIGURA 3.3 – Holosemântica aplicada à modelagem

Destacam-se como principais objetivos da Holosemântica:

- **pequeno *gap* semântico**: a distância semântica entre as unidades de modelagem e a realidade é pequena. O termos *ente* e *símbolo* são usados no cotidiano, tornando-se assim, agradáveis para o uso humano;

- **semântica distribuída**: a Holosemântica é uma semântica distribuída, isto é, transmite de forma subliminar o potencial de distribuição existente no domínio modelado. Esta característica estimula a exploração automática da distribuição.

O ente é a principal abstração do Holoparadigma. Toda existência é um ente, desde a mais simples até a mais complexa. O ente possui duas características básicas: distinção e história (figura 3.4). Encontra-se a seguir uma descrição dessas características:

- **Distinção**: um ente existe de forma distinta, ou seja, pode ser distinguido do restante da realidade. Sendo assim, a existência de um ente depende da percepção de quem distingue. Se uma distinção é percebida, existe um ente. A distinção está relacionada com níveis de abstração. Por exemplo, um texto é um ente. No entanto, em níveis mais baixos de abstração existem vários outros entes (capítulos, seções, subseções, parágrafos, frases e caracteres). A existência deles depende do nível de abstração com que o leitor percebe o texto. Um ente pode ser um objeto, um ser vivo, um software, um modelo computacional, uma organização, um grupo de entes e outras existências;
- **História**: um ente possui história. Sendo assim, o ente é uma existência temporal, possuindo passado, presente e futuro. Desde sua criação inicia a construção de sua história, a qual evolui até sua extinção. A história é o passado de um ente. Algumas constatações realizadas no presente entram para a história. Além disso, alterações na sua história o afetam de diversas formas.

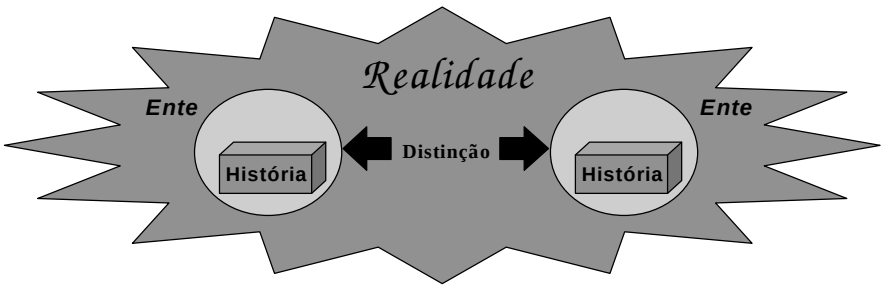


FIGURA 3.4 – Características dos entes

3.4 Tipos de entes

O Holoparadigma estabelece duas **classificações básicas** para organização dos entes: organizacional e funcional. A **classificação organizacional** distingue os entes, de acordo com a sua estrutura, em dois tipos:

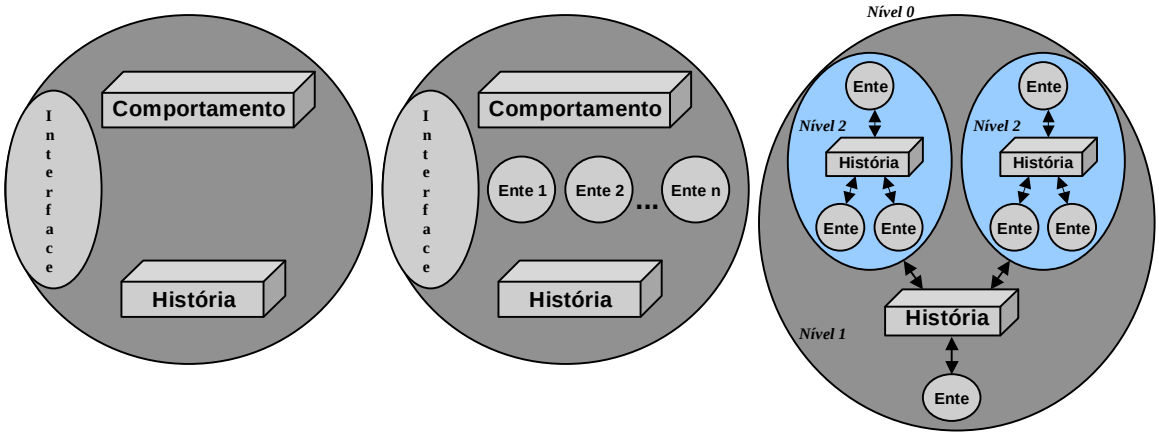
- **Ente elementar:** ente sem níveis de composição;
- **Ente composto:** ente formado pela composição de outros entes. Não existe limite para níveis de composição.

Um **ente elementar** (figura 3.5a) é organizado em três partes: interface, comportamento e história. A **interface** descreve suas possíveis relações com os demais entes. O **comportamento** contém **ações** que implementam a funcionalidade de um ente. Holo não estabelece os tipos de ações a serem utilizadas, no entanto, estabelece que existem dois tipos básicos de comportamento:

- **Imperativo**: o comportamento imperativo é composto de ações imperativas que descrevem os caminhos para solução de um problema (enfoque no controle, ou seja, como deve ser realizada a ação). Uma ação imperativa possui uma natureza determinista. O paradigma imperativo [GHE 98, p.259; SEB 99, p.21] é uma alternativa para descrição do comportamento imperativo;
- **Lógico**: o comportamento lógico é composto de ações lógicas que expressam um problema de forma declarativa (enfoque na lógica, ou seja, o que deve ser realizado). Uma ação lógica possui uma natureza não-determinista. O paradigma em lógica [KOW 79, KOW 79a, ROB 92] é uma alternativa para descrição do comportamento lógico.

A **história** é um espaço de armazenamento compartilhado no interior de um ente. O símbolo é o átomo de informação no Holoparadigma. Holo propõe a utilização do processamento simbólico como principal instrumento para o tratamento de informações. Esta característica é herdada do paradigma em lógica. Neste sentido, a variável lógica e a unificação são consideradas a base do tratamento de símbolos. Holo estabelece que a história deve ser direcionada para armazenamento e gerenciamento de símbolos. Portanto, o paradigma em lógica torna-se uma alternativa adequada para sua implementação.

Um **ente composto** (figura 3.5b) possui a mesma organização de um ente elementar, no entanto, suporta a existência de outros entes na sua composição (**entes componentes**). Cada ente possui uma história. A história fica encapsulada no ente e, no caso dos entes compostos, é compartilhada pelos entes componentes. Os entes componentes participam do desenvolvimento da história compartilhada e sofrem os reflexos das mudanças históricas. Sendo assim, podem existir vários níveis de encapsulamento da história. Os entes acessam somente a história em um nível. A composição varia de acordo com a mobilidade dos entes em tempo de execução (seção 3.5). A figura 3.5c mostra dois níveis de história encapsulada em um ente composto organizado em três níveis. Os comportamentos e interfaces foram omitidos para simplificação da figura.



(a) Ente elementar (b) Ente Composto (c) Exemplo de composição (3 níveis)

FIGURA 3.5 – Tipos de entes

Um ente elementar assemelha-se a um **objeto** [GHE 98, p.287; SEB 99, p.436] do paradigma orientado a objetos. Do ponto de vista estrutural, a principal diferença consiste na história, a qual atua como uma forma alternativa de comunicação e sincronização (modelo de coordenação, discutido na seção 3.7). Um ente composto assemelha-se a um **grupo** [BIR 93, LIA 90, NUN 98, LEA 2001]. Neste caso, a história

atua como um espaço compartilhado vinculado ao grupo. Doug Lea [LEA 2001] salienta que os conceitos da orientação a objetos, tais como objetos e classes, tornam-se limitados quando o tratamento de composição envolve aspectos dinâmicos (mudam durante a execução). Doug propõe a utilização de grupos para solução desta limitação. Por sua vez, Nierstrasz [NIE 93, p.152] destaca que as linguagens orientadas a objeto enfocam aspectos de programação em detrimento de aspectos de composição. Vários estudos relacionados com arquitetura de software seguem a mesma argumentação [IEE 95, GAR 95, SHA 96]. Os entes compostos aliados à mobilidade (seção 3.5) permitem a composição dinâmica no Holoparadigma. Além disso, a utilização de uma mesma unidade (ente) para manipulação de elementos e grupos, simplifica o modelo e sintetiza conceitos já existentes na ciência da computação.

A **classificação funcional** distingue os entes de acordo com suas funções:

- **Ente estático**: definição estática de um ente. Esta definição estabelece um padrão estático que pode ser utilizado para criação de outros entes através da clonagem (seção 3.9). Um ente estático é criado no nível de modelagem e programação. Modelos e programas são compostos de descrições de entes (entes estáticos), as quais estabelecem interfaces, comportamentos e histórias;
- **Ente dinâmico**: ente em execução. Um programa em execução é composto de entes dinâmicos. Estes entes executam ações e interagem (seção 3.8) de acordo com seus comportamentos e histórias.

A única distinção entre entes estáticos e dinâmicos consiste na sua função. Os entes estáticos são utilizados como matrizes estáticas para criação de outros entes. Além disso, estabelecem um estado inicial para execução de programas. Por sua vez, os dinâmicos representam o estado corrente de uma execução.

3.5 Distribuição e mobilidade

O Holoparadigma busca a distribuição implícita através da Holosemântica. Neste escopo, um ente assume dois **estados de distribuição**:

- **Centralizado:** um ente está centralizado quando se localiza em apenas um nodo de um sistema distribuído. Entes elementares estão sempre centralizados. Um ente composto está centralizado se todos os seus entes componentes estão localizados no mesmo nodo;
- **Distribuído:** um ente está distribuído quando se localiza em mais de um nodo de um sistema distribuído. Entes elementares não podem estar distribuídos. Um ente composto está distribuído se um ou mais entes componentes estão distribuídos.

A figura 3.6 exemplifica uma possível distribuição para o ente apresentado na figura 3.5c. O ente encontra-se distribuído em dois nodos da arquitetura distribuída. A história de um ente distribuído é denominada **história distribuída**. A distribuição da história pode ser baseada em técnicas de **memória compartilhada distribuída** [LIK 89, PRJ 96, PRO 99] ou **espaços distribuídos** [CIP 94, FRE 99, CIP 2001].

A mobilidade [ROY 97, IEE 98, LAN 98, FER 99, FER 2001a] é a capacidade que permite o deslocamento de um ente. No Holoparadigma existem dois tipos de mobilidade:

- **Mobilidade lógica:** a mobilidade lógica relaciona-se com o deslocamento em nível de modelagem, ou seja, sem considerações sobre a plataforma de execução. Neste contexto, um ente se move quando cruza uma ou mais fronteiras de entes;
- **Mobilidade física:** a mobilidade física relaciona-se com o deslocamento entre nodos de uma arquitetura distribuída. Neste contexto, um ente se move quando desloca-se de um nodo para outro.

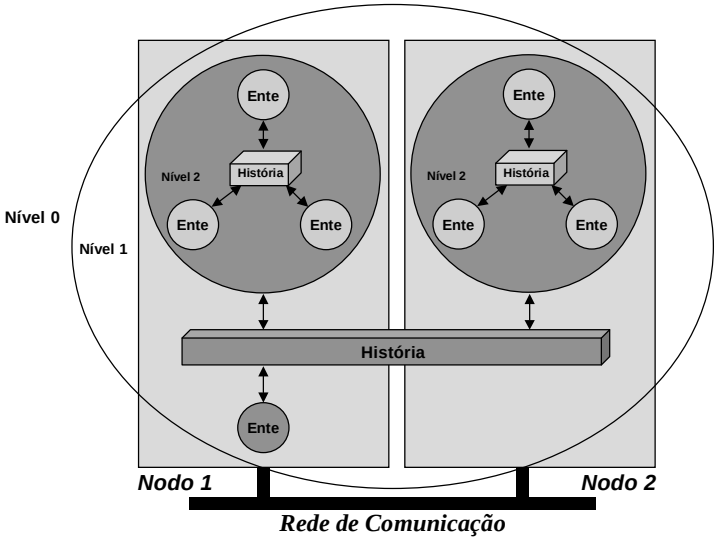


FIGURA 3.6 – Ente distribuído

A figura 3.7a exemplifica uma possível mobilidade lógica no ente apresentado na figura 3.5c. Conforme exemplificado, após o deslocamento, o **ente móvel** não possui mais acesso à história no **ente origem**. No entanto, passa a ter acesso à história no **ente destino**. Neste caso, somente ocorrerá mobilidade física se os entes origem e destino estiverem alocados em diferentes nodos de uma arquitetura distribuída.

As mobilidades lógica e física são independentes. A ocorrência de um tipo de deslocamento não implica a ocorrência do outro. Merece atenção o caso da mobilidade física não implicar obrigatoriamente a mobilidade lógica. Considere-se o ente distribuído mostrado na figura 3.6. Se o ente elementar localizado no nível um, realizar um deslocamento físico conforme mostrado na figura 3.7b, não haverá mobilidade lógica apesar de ter ocorrido mobilidade física. Neste caso, o ente movido continua com a mesma visão da história.

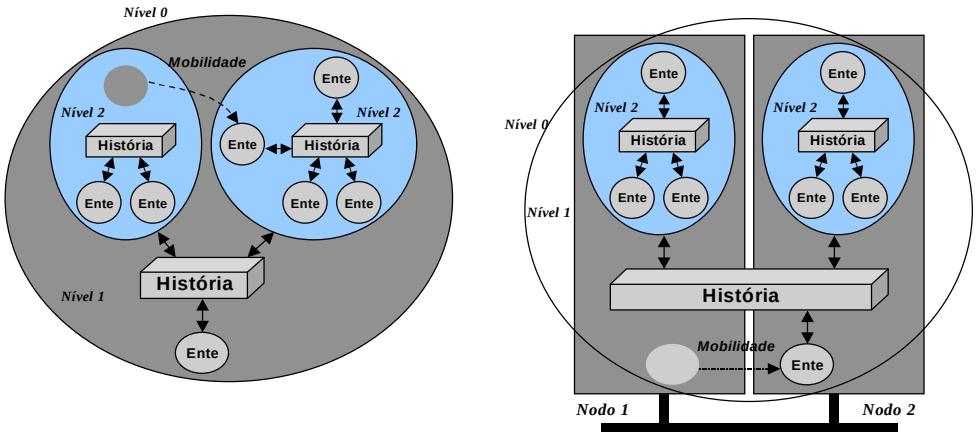


FIGURA 3.7 – Mobilidade no Holoparadigma

3.6 Holosemântica e Cálculo de Ambientes

Nos últimos anos os estudos sobre mobilidade em sistemas distribuídos [CAL 98, IEE 98, CAL 2000, CAL 2000a, FER 2001, FER 2001a] foram impulsionados pelo surgimento da Internet e da *World-Wide-Web* e pela ploriferação de dispositivos eletrônicos portáteis (celulares, *notebooks*, *palmtops*, etc). Cardelli [CAL 2000, p.8] destaca que os sistemas distribuídos estáticos baseados em LANs e WANs serão suplantados por um novo paradigma baseado em mobilidade, onde os nodos (equipamentos e redes inteiras) serão móveis. Sendo assim, tornam-se relevantes os estudos relacionados com formalismos para modelagem de concorrência, distribuição e, principalmente, mobilidade. Neste contexto, Cardelli discute as propostas existentes [CAR 99b, p.62] e cria o Cálculo de Ambientes [CAL 97, CAL 98, CAL 99, CAL 99a, CAL 99b, CAL 2000, CAL 2000a, CAL 2001]. Este cálculo serve como modelo abstrato para computação em WANs [CAL 2000, p.11] e possui como principal conceito a noção de **barreiras** [CAL 99, p.21]. Cardelli destaca que a maioria dos aspectos relacionados à mobilidade envolvem barreiras:

- **Localidade:** a noção de posicionamento é induzida pela existência de diversas localizações físicas ou virtuais. Esta noção depende de uma topologia de barreiras;
- **Mobilidade:** deslocamento entre posições envolve a travessia de barreiras;
- **Segurança:** está relacionada com a habilidade de cruzar barreiras;
- **Comunicação:** controlada pelas barreiras. Comunicações locais ocorrem no escopo de uma barreira e remotas dependem de mobilidade e comunicações locais. Cardelli salienta que ações à distância são proibidas quando utilizadas barreiras [CAL 99, p.21; CAL 2000, p.9];

Um **ambiente** é um local delimitado por uma barreira, onde ocorrem determinadas computações. Um ambiente possui as seguintes características [CAL 98, p. 142; CAR 2000, p.11]:

- limitado por uma barreira;
- suporta outros ambientes aninhados, formando assim uma estrutura em árvore;
- possui uma coleção de processos locais, os quais estão localizados diretamente no ambiente (não em subambientes);
- movimenta-se de forma atômica, ou seja, carrega consigo todos os processos e subambientes;
- possui um nome utilizado para seu controle [CAL 2001].

O Cálculo de Ambientes é baseado em um conjunto de primitivas (figura 3.8), as quais estabelecem a sintaxe das construções usadas para formalização da mobilidade. A primeira primitiva estabelece que os ambientes possuem nomes. As demais estão organizadas em duas categorias: processos e capacidades. As quatro primeiras primitivas relacionadas com processos, normalmente fazem parte do cálculo de processos [CAL 2000, p.14]. As outras duas permitem a definição de ambientes e a realização de ações (capacidades). As capacidades suportadas pelo ambientes são descritas na segunda categoria (entrar, sair e abrir).

n	**nomes**
P,Q ::=	**processos**
(ν n) P	restrição
0	inatividade
P | Q	composição
!P	replicação
n[P]	ambiente
M.P	ação
M ::=	**capacidades**
in n	entrar em *n*
out n	sair de *n*
open n	abrir *n*

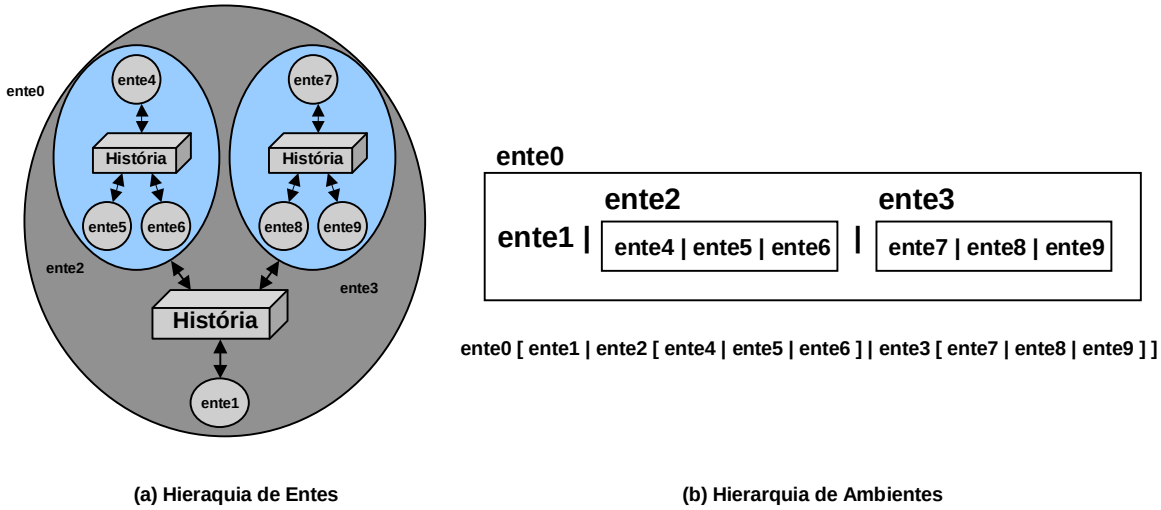
FIGURA 3.8 – Primitivas do Cálculo de Ambientes

A tabela 3.1 mostra uma relação entre os conceitos do Holoparadigma e o Cálculo de Ambientes. Um ente equivale a um ambiente. Entes elementares são ambientes sem níveis de composição. Por sua vez, um ente composto equivale a um ambiente contendo subambientes. Processos correspondem às ações no comportamento dos entes. Além disso, nomes podem ser manipulados através de símbolos (unidade de informação do Holoparadigma, veja figura 3.3).

TABELA 3.1 – Equivalência entre conceitos do Holoparadigma e ambientes

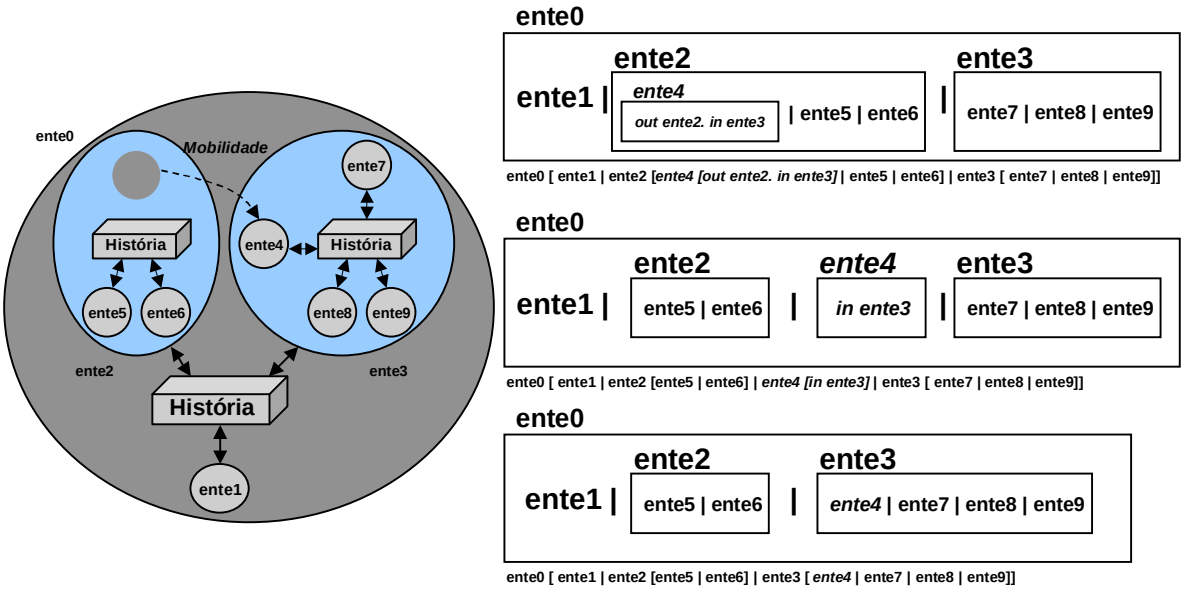
Ambientes	Holoparadigma
Ambiente	Ente
Ambiente sem composição	Ente elementar
Ambiente composto de ambientes	Ente composto
Processos	Ações do comportamento
Nomes	Símbolos
Mobilidade de ambientes	Mobilidades de entes
Barreiras	Limites de entes
Hierarquia de ambientes	Hierarquia de entes

A figura 3.9a mostra uma hierarquia de entes (veja figura 3.5c). Os entes receberam nomes, permitindo assim, sua identificação. A figura 3.9b mostra uma hierarquia de ambientes usando as notações gráficas e textuais propostas por Cardelli [CAL 2000, p.15]. O símbolo "|" corresponde à composição (figura 3.8). Conforme mostra a figura 3.9, as hierarquias são equivalentes. Sendo assim, as notações do Cálculo de Ambientes são adequadas para a especificação da composição de entes no Holoparadigma. A figura 3.10a mostra a mobilidade de um ente (veja figura 3.7a). Por sua vez, a figura 3.10b apresenta a evolução da notação do Cálculo de Ambientes para representação da mesma mobilidade. Merece atenção, a especificação do ambiente *ente4*. A notação *out ente2*. *in ente3* significa que duas ações serão executadas em sequência por um processo no ambiente (tabela 3.1, primitiva *ação*). A primeira (*out ente2*) move *ente4* para fora de *ente2*. A segunda (*in ente3*) desloca *ente4* para o interior de *ente3*. A figura 3.10 mostra que o cálculo proposto por Cardelli pode ser usado para formalização da mobilidade no Holoparadigma.



(a) Hieraquia de Entes (b) Hierarquia de Ambientes

FIGURA 3.9 – Equivalência entre hierarquias de entes e ambientes



(a) Mobilidade no Holoparadigma (b) Mobilidade no Cálculo de Ambientes

FIGURA 3.10 – Equivalência entre mobilidade no Holoparadigma e ambientes

3.7 Modelo de coordenação

Um modelo de coordenação define um padrão organizacional para o software [IEE 95, GAR 95, SHA 96, OSS 99]. Shaw e Garlan [SHA 96] discutem os modelos mais utilizados. Neste contexto destaca-se o **repositório**. Este modelo possui dois tipos de componentes: uma estrutura de dados central que representa o estado corrente e uma coleção de componentes independentes que operam no armazenamento central.

A interação entre o repositório e os componentes pode variar de forma significativa. Se o estado corrente do repositório é a principal fonte de controle dos componentes, o repositório pode ser um ***blackboard*** [VRA 95, PFL 97]. Este modelo é composto de três partes (figura 3.11a):

- **fontes de conhecimento (*Knowledge Sources* - KS):** componentes da aplicação que interagem (comunicação e sincronismo) somente através do armazenamento central;

- **armazenamento central (*Blackboard*):** armazena o estado da resolução do problema;

- **controle:** controla o *blackboard* fazendo com que os KSs respondam a suas alterações.

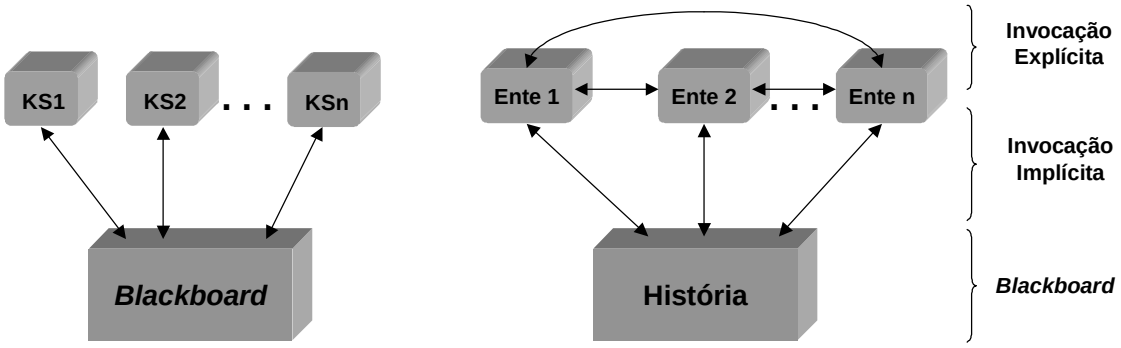


FIGURA 3.11 – Modelo de coordenação

O modelo é baseado na **invocação implícita**. Os componentes usam o *blackboard* para anúncio de eventos e para registro de interesse em um determinado evento. O controle pode invocar de forma implícita vários KSs quando um evento é anunciado.

A implementação do Holoparadigma depende da escolha de um modelo de coordenação que suporte de forma adequada suas principais abstrações. A organização de um ente composto assemelha-se a forma como os repositórios são organizados, ou seja, vários componentes que compartilham um armazenamento. Neste caso, os componentes são entes e o repositório é a história. Em Holo, a história não serve apenas para armazenamento de informações compartilhadas. Parte do controle dos entes é responsabilidade da história. Essa característica é obtida com a utilização do estilo *blackboard* (invocação implícita). Existem várias limitações estabelecidas pelo uso da invocação implícita [SHA 96]. O uso da **invocação explícita** conjuntamente com a invocação implícita é salientado como uma solução para essas limitações. Em Holo ambos os estilos de invocação são utilizados. Os entes influenciam outros entes através da história (invocação implícita), mas também podem trocar informações diretamente (invocação explícita). O modelo do Holoparadigma é mostrado na figura 3.11b.

3.8 Tipos de interação e modos de invocação

Uma **interação** é uma troca de informações. A descrição de uma interação estabelece *quem* interage (tipo de interação) e *como* interage (modo de invocação). No Holoparadigma existem oito **tipos de interação** (*quem* interage, figura 3.12):

1. **Comportamento-História** (*Behavior-History Interaction – BHI*): interação entre uma ação e a história de um ente;
2. **Componente-História** (*Component-History Interaction – CHI*): interação entre uma ação de um ente componente e a história do ente composto;
3. **Comportamento-Componente** (*Behavior-Component Interaction – BCI*): interação entre uma ação de um ente componente e uma ação do composto;
4. **Autocomportamental** (*AutoBehavior Interaction – ABI*): interação entre ações de um ente;
5. **Ente-Ente** (*Being-Being Interaction – BBI*): interação entre ações de entes localizados no mesmo nível;
6. **Autohistórica** (Auto*Historical Interaction – AHI*): interação direta entre o conteúdo da história de um ente;
7. **Comportamento-Interface** (*Behavior-Interface Interaction – BII*): interação entre uma ação e a interface de um ente;
8. **Componente-Interface** (*Component-Interface Interaction – CII*): interação entre uma ação de um ente componente e a interface de um ente composto.

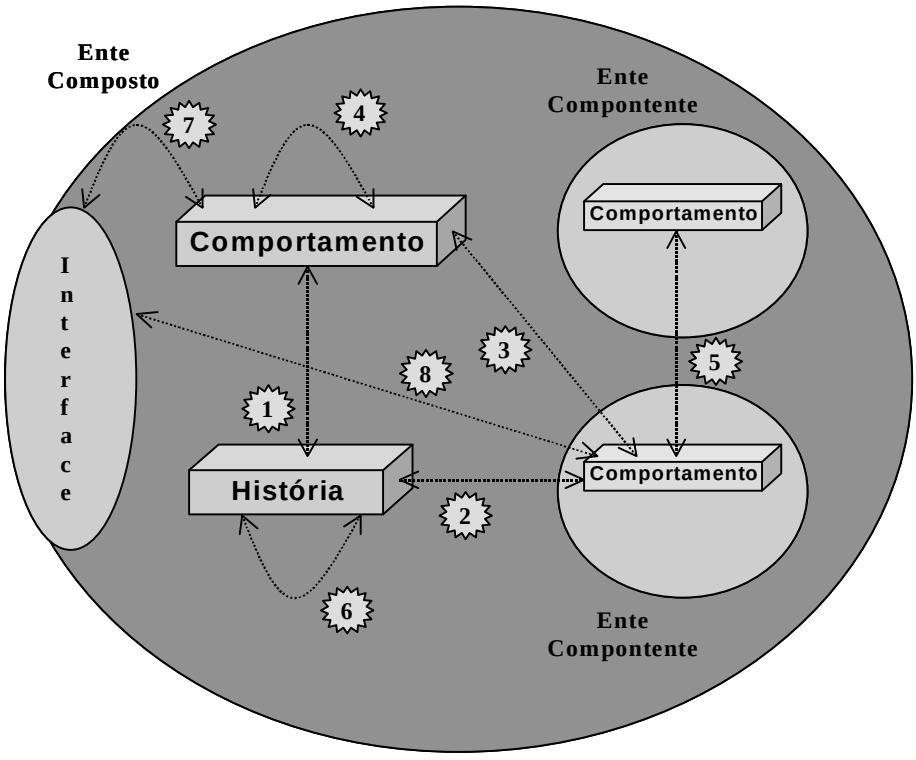


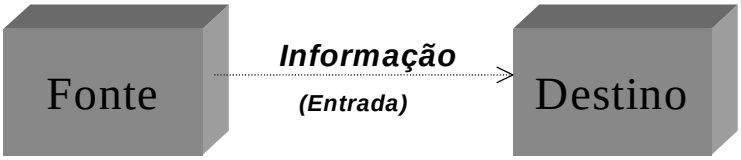
FIGURA 3.12 – Tipos de interação

As interações dos tipos 3, 4, 7 e 8 suportam a alteração dinâmica (tempo de execução) do comportamento e da interface de um ente. Além disso, o modelo de coordenação (figura 3.11b) mostra o relacionamento entre entes componentes no interior de um composto. Naquela figura, as invocações implícitas são do tipo 2 (CHI) e as invocações explícitas são do tipo 5 (BBI).

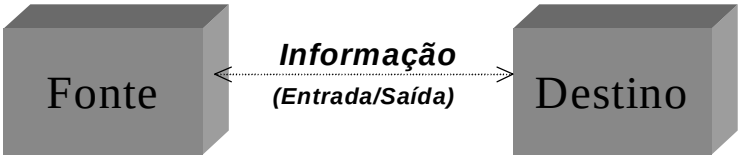
O Holoparadigma propõe dois **modos de invocação** (figura 3.13):

- **Afirmação**: envolve um fluxo de informação unidirecional da fonte para o destino (figura 3.13a). Neste caso, existe somente um fluxo de entrada. Este tipo de invocação não estabelece sincronismo. Quem afirma não aguarda nenhum tipo de resposta. As afirmações podem ser utilizadas para comunicação assíncrona e para inserção de informações na interface, no comportamento e na história;
- **Pergunta**: envolve um fluxo de informação bidirecional entre fonte e destino (figura 3.13b), ou seja, ocorre um fluxo de entrada e/ou saída. Uma pergunta estabelece sincronismo. Quem pergunta aguarda uma resposta. As perguntas podem ser utilizadas para comunicação síncrona e para retirada de informações da interface, do comportamento e da história.

Uma invocação pode ser classificada de acordo com os tipos mostrados na figura 3.12. Por exemplo, um ente pode realizar uma afirmação para outro ente no mesmo nível. Neste caso, ocorre uma afirmação do tipo 5 (afirmação BBI).



(a) Afirmação



(b) Pergunta

FIGURA 3.13 – Modos de invocação

3.9 Holoclonagem

O Holoparadigma estabelece que um software pode ser percebido através de dois domínios (figura 3.14): **domínio estático** (composto pelos entes estáticos) e o **domínio dinâmico** (composto pelos entes dinâmicos). A criação de entes nos domínios estático e dinâmico é realizada através de clonagem. A clonagem é a criação de um ente tendo como matriz outro(s) ente(s). Os entes que servem como matriz são chamados **clonados** e o ente criado recebe o nome de **clone**. O Holoparadigma propõe a existência de três tipos de clonagem:

- **Estática**: a clonagem estática é realizada na descrição dos entes (domínio estático), ou seja, durante a especificação em nível de modelagem e programação. A clonagem estática permite a descrição de entes através da extensão de especificações de outros entes (interface, comportamento e história). Novas características podem ser acrescentadas no clone através de sua especificação. A figura 3.14 mostra uma clonagem estática;

- **Dinâmica**: a clonagem dinâmica ocorre em tempo de execução (domínio dinâmico). Esta clonagem cria uma *cópia de um ente em execução*. Neste caso, o clone possui a estrutura (interface e comportamento) e o estado (história e composição) do clonado. Os entes componentes do clonado também devem ser clonados dinamicamente para composição do clone. Além disso, o clone é criado no mesmo nível do clonado e passa a acessar a mesma história que ele acessa. A figura 3.14 apresenta uma clonagem dinâmica;

- **De Transição**: a clonagem de transição ocorre em tempo de execução. Este tipo de clonagem permite que um ente estático seja utilizado como matriz para criação de um ente dinâmico. Sendo assim, o clone é uma versão dinâmica do clonado. A figura 3.14 exemplifica uma clonagem de transição.

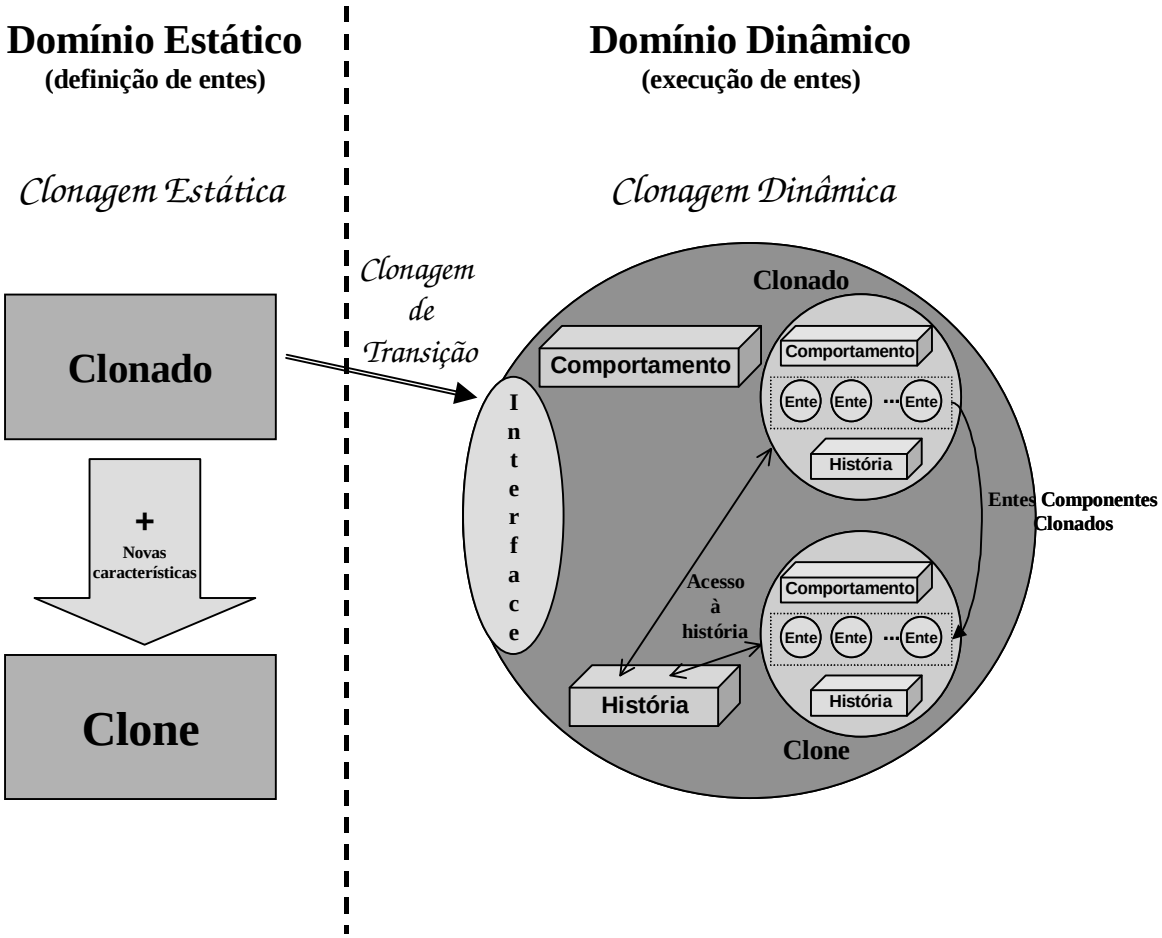


FIGURA 3.14 – Domínios e clonagem

A clonagem sintetiza três operações utilizadas na orientação a objetos. A clonagem estática assemelha-se à herança. A clonagem de transição equivale à instanciação de objetos usando classes. Por sua vez, a clonagem dinâmica corresponde à criação de clones suportada por algumas linguagens orientadas a objetos [FLA 2000, p.390]. A principal vantagem desta síntese consiste no estímulo ao uso de notações semelhantes para os três tipos de operações, simplificando assim, o gerenciamento de entes. O capítulo 4 propõe uma linguagem que explora esta vantagem.

A clonagem é **múltipla e seletiva**. Um clone pode ser originado de múltiplos clonados. Além disso, pode ser realizada uma seleção de quais partes (interface, comportamento, componentes ou história) serão clonadas. Por exemplo, a figura 3.15 mostra a criação de um clone através da clonagem de dois entes compostos. O clone recebe a história (clonagem 3) e a composição (clonagem 4) de ambos, mas a interface (clonagem 1) e o comportamento (clonagem 2) de apenas um deles.

Além disso, a possibilidade da existência de características não-deterministas nos entes (seção 3.4) permite ainda que a clonagem possa ser de dois tipos:

- **Construtiva**: estabelece que as especificações do clone e dos clonados são combinadas de forma construtiva. Esta combinação é uma *união* das especificações não-deterministas (por exemplo, comportamento lógico). A união pode seguir diversas políticas. A seção 4.5 propõe uma política específica para uma linguagem baseada no Holoparadigma;
- **Destrutiva**: estabelece que as especificações do clone e dos clonados são combinadas de forma destrutiva. As especificações mais recentes sobrepõem as anteriores (padrão utilizado na herança da orientação a objetos). A seção 4.5 descreve uma política para clonagem destrutiva. As características deterministas (por exemplo, comportamento imperativo) sempre são clonadas de forma destrutiva.

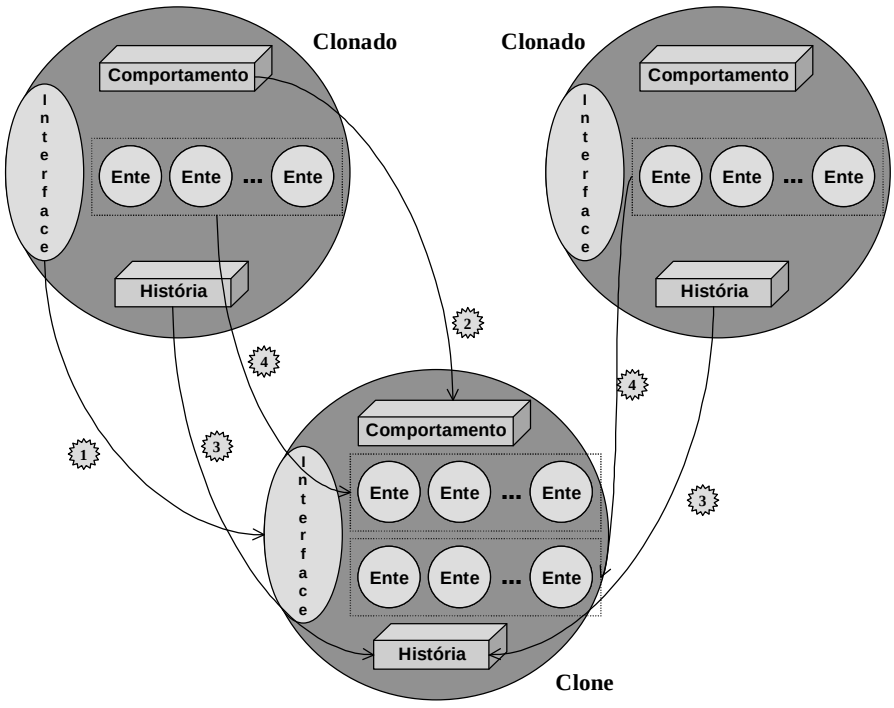


FIGURA 3.15 – Clonagem múltipla e seletiva

3.10 Holoplataforma

Uma plataforma é um conjunto de software e hardware que suporta o desenvolvimento e a execução de sistemas computacionais. A figura 3.16 apresenta a organização da plataforma do Holoparadigma (Holoplataforma). A Holoplataforma é composta de duas partes:

- **Plataforma de desenvolvimento:** conjunto de ferramentas de software utilizadas na criação dos sistemas computacionais. Esta plataforma é composta de duas partes: (1) uma ferramenta CASE que suporte as abstrações propostas pelo paradigma (HoloCASE [SOA 2000]), gerando código em uma linguagem baseada em Holo (Hololinguagem) e (2) um compilador para a Hololinguagem que gere *Código Virtual Multiparadigma* (CVM). Este código será orientado à *Máquina Virtual Multiparadigma* (MVM) projetada para o Holoparadigma (*Holo Virtual Machine* - HoloVM);
- **Plataforma de execução:** conjunto de hardware e software utilizado como suporte à execução de programas. Esta plataforma possui três componentes: (1) uma máquina virtual orientada ao Holoparadigma (HoloVM) para execução do CVM, (2) um ambiente (*Distributed Holo* – DHolo) que suporte processamento distribuído quando for utilizada uma arquitetura distribuída (anexo 1.1) e (3) um conjunto de hardware e sistema operacional. Os dois primeiros componentes formam o ambiente de execução.

As abstrações propostas pelo Holoparadigma são independentes do tipo de sistema computacional utilizado. No entanto, Holo é orientado para arquiteturas distribuídas. As principais características a serem exploradas em ambientes distribuídos são as seguintes:

- **Mobilidade**: a HoloVM cria uma camada de abstração que simplifica a mobilidade física de entes (figura 3.7b), especialmente em sistemas distribuídos heterogêneos;
- **História distribuída hierárquica e dinâmica**: a distribuição envolve o compartilhamento de armazenamento (história) entre entes localizados em nodos diferentes (figura 3.6). Neste contexto, surge a possibilidade de diversos níveis de história compartilhada e a necessidade de adaptação à mobilidade durante a execução (figura 3.7b).

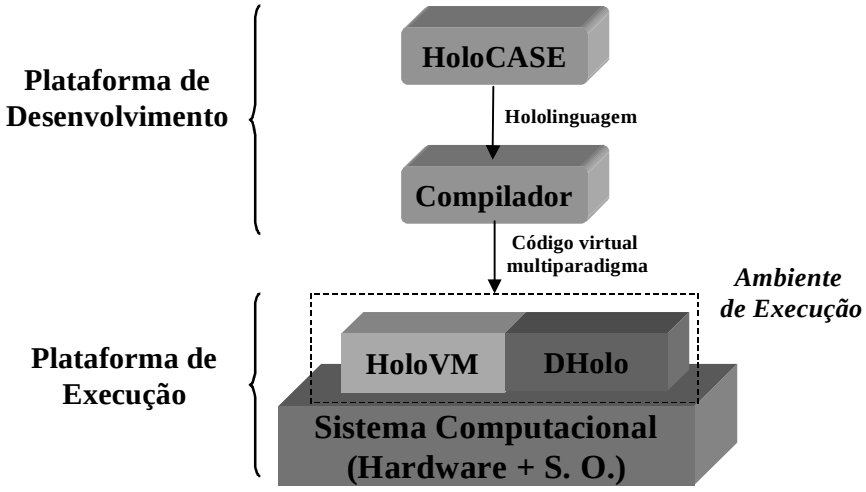


FIGURA 3.16 – Holoplataforma

3.11 Considerações finais

Este capítulo apresentou o Holoparadigma, um modelo multiparadigma orientado ao desenvolvimento de software paralelo e distribuído. Este modelo consiste de um conjunto de idéias que podem ser utilizadas para criação de plataformas de desenvolvimento e execução. O anexo 1 contém tópicos que complementam este capítulo. Merece atenção especial, o anexo 1.1 onde é proposta uma organização hierárquica para arquiteturas distribuídas [BAR 2000, p.23]. Esta organização se adapta aos conceitos do Holoparadigma, podendo assim, ser utilizada para criação de uma plataforma distribuída para o modelo.

Destacam-se como principais conclusões do capítulo:

- a exploração automática da distribuição é estimulada com a utilização de um paradigma de desenvolvimento que possua uma semântica distribuída;
- os temas multiparadigma e arquitetura de software podem ser integrados para a criação de novos paradigmas de desenvolvimento de software;
- os entes possuem dois tipos básicos de comportamento, ou seja, lógico e imperativo;
- a utilização de memória distribuída compartilhada [LIK 89, PRJ 96, PRO 99] ou espaços distribuídos [CIP 94, FRE 99, CIP 2001] permite a implementação da história distribuída (discutida na seção 3.5);
- a mobilidade e os níveis de composição necessitam de uma história hierárquica e dinâmica;
- as mobilidades lógica e física são independentes e podem ser tratadas de forma distinta;
- existe uma equivalência entre os principais conceitos do Holoparadigma e o Cálculo de Ambientes proposto por Cardelli [CAL 97, CAL 98, CAL 99, CAL 99a, CAL 99b, CAL 2000, CAL 2000a, CAL 2001];
- a utilização conjunta das invocações implícita e explícita cria um modelo de coordenação que permite a implementação das principais abstrações (ente e história) propostas pelo Holoparadigma;
- a clonagem atua como uma síntese de três operações usadas na orientação a objetos (herança, instanciação e criação de clones). Além disso, a clonagem pode ser múltipla e seletiva;
- o ente atua como uma síntese para gerenciamento simplificado de elementos e grupos;
- o relacionamento entre entes e suas partes estabelece oito tipos de interação e dois modos de invocação.

O capítulo 4 descreve uma linguagem que implementa os conceitos propostos pelo Holoparadigma. Conforme mostra a figura 3.16, a Hololinguagem constitui um dos principais componentes da Holoplataforma.

4 Hololinguagem

Este capítulo descreve a Hololinguagem (de forma abreviada, Holo). Esta linguagem suporta o desenvolvimento de programas usando os princípios propostos pelo Holoparadigma. Os aspectos gerais da Hololinguagem foram apresentados à comunidade acadêmica através das publicações [BAR 99a] e [BAR 2001a]. Além disso, os aspectos relacionados com a exploração de paralelismo e distribuição foram analisados em [BAR 2001] e [BAR 2001b]. Por sua vez, o anexo 1.7 descreve a integração do paradigma funcional na linguagem [DUB 2001].

4.1 Principais características

Holo é uma linguagem multiparadigma direcionada para o desenvolvimento de sistemas distribuídos. Neste sentido, destacam-se como suas principais características:

- **multiparadigma**: Holo integra múltiplos paradigmas básicos. Atualmente, estão integrados os paradigmas imperativo, em lógica e orientado a objetos. Holo busca não somente o suporte aos paradigmas integrados, mas também as novas oportunidades que surgem através da integração. Entre estas oportunidades destacam-se: clonagem múltipla e seletiva (seção 4.5), clonagem construtiva (seção 4.5), integração das invocações implícita e explícita (seção 4.3) e operadores multiparadigma (seção 4.7);
- **sem tipos e orientada ao processamento simbólico**: conforme descrito na seção 3.3, o símbolo é a unidade de informação do Holoparadigma. A Hololinguagem é baseada no processamento de símbolos e não possui tipos de dados. Neste sentido, Holo utiliza variáveis lógicas. Esta característica é herdada do paradigma em lógica;
- **integração da atribuição e unificação**: as variáveis lógicas são manipuladas através da unificação (paradigma em lógica), mas também existe suporte para atribuição (paradigma imperativo). Holo propõe o uso conjunto destas operações;
- **programação de alta ordem**: Holo suporta programação de alta ordem, ou seja, símbolos representando entes e suas partes (interface, ações, comportamento e história) podem ser manipulados através de variáveis e operações da linguagem;
- ***blackboard* hierárquico**: existe suporte a vários níveis de *blackboard*, conforme proposto pelo Holoparadigma na seção 3.4;
- **suporte à Holoclonagem**: Holo implementa a clonagem múltipla e seletiva proposta pelo Holoparadigma. Além disso, existe suporte para clonagem construtiva e destrutiva;
- **mobilidade**: a mobilidade lógica é suportada explicitamente na Hololinguagem (ação *move*, seção 4.6). A mobilidade física torna-se responsabilidade do ambiente de execução (seção 5.3) projetado para arquiteturas distribuídas (anexo 1.1);
- **suporte à concorrência**: existe suporte para concorrência entre ações (invocação via afirmação na seção 4.3 e ação *spawn* na seção 4.6) e entre entes (ação *clone* na seção 4.6);

- **sintaxe lógica compatível com Prolog**: os aspectos lógicos da sintaxe da linguagem são compatíveis com a linguagem Prolog [STE 86, CAS 87, DER 96];
- **sintaxe imperativa baseada no Pascal**: os aspectos imperativos da sintaxe são baseados no Pascal [WIR 73, p.155; JEN 88]. Esta linguagem foi escolhida pela sua simplicidade e pelas suas semelhanças sintáticas com Prolog, tais como, comandos de entrada e saída (*write* e *read*), operador de atribuição "*:=*" e operador de comparação "*=*".

4.2 Descrição de entes

Um holoprograma (programa em Holo) é formado por descrições de entes (entes estáticos). A figura 4.1 mostra a descrição de um ente (baseada na figura 3.5a). A descrição é formada por *Cabeça* e *Corpo*. A cabeça contém o nome de um ente, seus argumentos e uma descrição de sua clonagem. Os argumentos seguem os mesmos princípios dos parâmetros de classe propostos por Ng e Luk [NGK 45, p.83]. As variáveis utilizadas na composição dos argumentos são visíveis em todo o corpo. Estas variáveis podem receber valores no momento da criação de um ente, mas não podem ser alteradas durante sua existência. O único espaço de armazenamento compartilhado no interior de um ente é a história. Na cabeça são explicitadas as ações que serão exportadas (**interface**), ou seja, ações que podem ser acessadas do exterior de um ente. O corpo é formado por duas partes: **comportamento** e **história**. O comportamento é constituído por um conjunto de **ações** que implementam a funcionalidade. Estas ações estão disponíveis no interior de um ente. Além disso, podem ser exportadas através da interface. A história é um *blackboard* que pode ser acessado pelas ações, pelos entes componentes ou pela própria história (veja figura 3.12, interações dos tipos 1, 2 e 6).

A história armazena **Cláusulas Lógicas com Extensões não Imperativas (CLEI)**. Uma CLEI é uma cláusula lógica (formato Prolog) [CAS 87, p. 285] que suporta apenas extensões extralógicas [CAS 87, p. 341] não imperativas (facilidades aritméticas e/ou facilidades de comparação). A utilização de CLEIs simplifica a implementação do *backtracking* no interior da história, nas ações e entre invocações. Esta simplificação permite a exploração do não-determinismo natural do paradigma em lógica. Além disso, o uso de CLEIs facilita a integração dos paradigmas em lógica e imperativo, conforme será discutido ainda nessa seção. Uma CLEI, quando armazenada na história, pode ser um **fato histórico** (cláusula sem corpo) ou uma **regra histórica** (cláusula com corpo). A história pode ter um conteúdo inicial, ou seja, pode ser inicializada pelo programador.

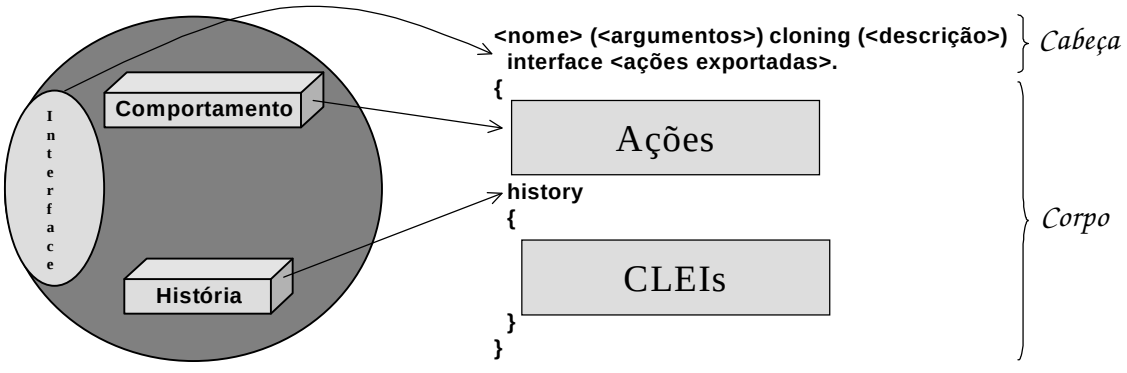


FIGURA 4.1 – Descrição de um ente

O comportamento de um ente é descrito através de ações. As ações suportam invocações com o uso de nomes e argumentos. Além disso, cada ação pode ser identificada univocamente através de um **Identificador de Ação** (*Action Identifier – AI*). Um AI possui o formato <*nome*>/<*aridade*>. Existem cinco tipos de ações:

- **Ação Lógica (*Logic Action* – LA)**: ação formada por um único predicado lógico (uma ou mais cláusulas com mesmo nome e mesma aridade). As LAs usam CLEIs, isto é, permitem o uso limitado de extensões extralógicas [CAS 87, p. 341] (somente facilidades de comparação e facilidades aritméticas). A ação mostrada na figura 4.2 implementa o conhecido predicado *append*. Esta ação possui o identificador *append/3*;

```
append([],L,L).
append([H|L],L1,[H|R]) :- append(L,L1,R).
```

FIGURA 4.2 – Exemplo de LA

- **Ação Imperativa (*Imperative Action* – IA)**: ação formada por uma função imperativa. Uma ação imperativa é encapsulada em uma função, conforme mostra a figura 4.3. A figura 4.4 apresenta uma IA que implementa a soma dos elementos de duas listas (*L1* e *L2*) de mesmo tamanho. O resultado retorna no terceiro argumento e na chamada da ação (comando *return*). A ação *length* é predefinida e determina o tamanho de listas. Holo utiliza comandos imperativos (no exemplo, o comando *for*) semelhantes aos utilizados em Pascal. O símbolo ":=" atua como operador de atribuição e a vírgula é usada como separador de comandos (semelhante ao Prolog). Além disso, o tamanho da lista *R* é definido dinamicamente através das atribuições.

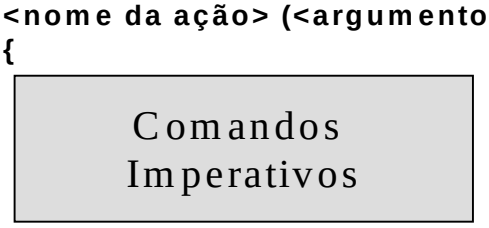


FIGURA 4.3 – Descrição de uma IA

```
add_lists(L1,L2,R)
{
  for C := 1 to length(L1) do
    R[C] := L1[C] + L2[C],
  return(R)
}
```

FIGURA 4.4 – Exemplo de IA

- **Ação Modular Lógica (*Modular Logic Action* – MLA)**: atualmente, a maioria dos sistemas de programação Prolog suportam módulos (por exemplo, SICstus Prolog [SIC 95, p. 51]). Bugliesi et al [BRU 94] discutem as vantagens desta abordagem. Além disso, recentemente Cabeza e Hermenegildo [CAD 2000] propuseram um novo sistema de módulos para Ciao-Prolog. A MLA é um módulo que encapsula LAs. A figura 4.5 mostra sua estrutura. O nome e os argumentos são utilizados na sua invocação. A interface define quais LAs do módulo serão exportadas. O restante da MLA é composto pelo corpo e pelas LAs encapsuladas. Uma MLA com corpo e sem LAs corresponde a uma cláusula do paradigma em lógica (figura 4.6a). A figura 4.6b mostra um programa em lógica sendo convertido para uma MLA com corpo. O corpo da MLA `reverse_hanoi/5` executa `hanoi/5` e inverte a lista resultante. A LA `hanoi/5` é exportada. O acesso a ações exportadas é realizado com o uso do identificador da MLA. Por exemplo, fora do módulo a invocação `reverse_hanoi/5.hanoi(5,p1,p2,p3,Result)` executaria diretamente a ação `hanoi/5`. A MLA `reverse_hanoi` possui dois pontos de entrada (`reverse_hanoi/5` e `hanoi/5`). A limitação dos possíveis pontos de entrada de um programa em lógica simplifica a sua análise global através de interpretação abstrata [AZE 99, BAR 2000b]. Uma MLA pode não ter corpo. Neste caso, os pontos de entrada serão as LAs exportadas. Além disso, uma MLA pode indicar uma de suas LAs como ponto de entrada. A figura 4.7 exemplifica uma MLA (`hanoi/5`) que indica uma LA (`hanoi/5`) como ponto de entrada.

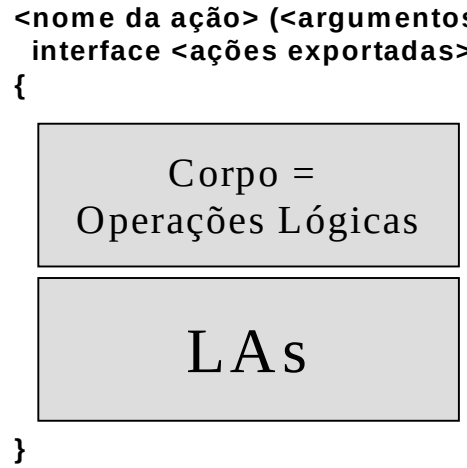
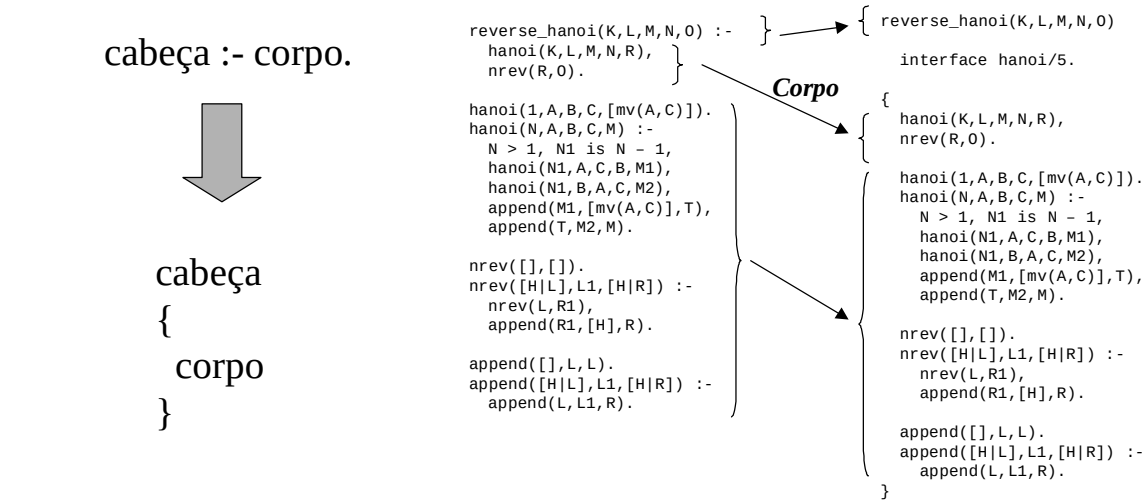


FIGURA 4.5 – Descrição de uma MLA



(a) Conversão de uma cláusula para MLA com corpo (b) Conversão de programa em lógica para uma MLA

FIGURA 4.6 – Exemplo de MLA com corpo

```
hanoi/5()
{
  hanoi(1,A,B,C,[mv(A,C)]).
  hanoi(N,A,B,C,M) :-
    N > 1, N1 is N – 1,
    hanoi(N1,A,C,B,M1),
    hanoi(N1,B,A,C,M2),
    append(M1,[mv(A,C)],T),
    append(T,M2,M).

  append([],L,L).
  append([H|L],L1,[H|R]) :-
    append(L,L1,R).
}
```

FIGURA 4.7 – Exemplo de uma MLA sem corpo

- **Ação Modular Imperativa (*Modular Imperative Action* – MIA)**: ação formada por um módulo que encapsula ações imperativas. A única diferença entre uma MIA e uma MLA é o tipo de comportamento suportado (imperativo na MIA). Toda a descrição apresentada para a MLA é válida para ações modulares imperativas. A figura 4.8 mostra a descrição genérica de uma MIA. O corpo de uma MIA é composto de ações imperativas. A figura 4.9 mostra uma MIA que soma os elementos de duas listas de mesmo tamanho e, logo após, inverte a lista;

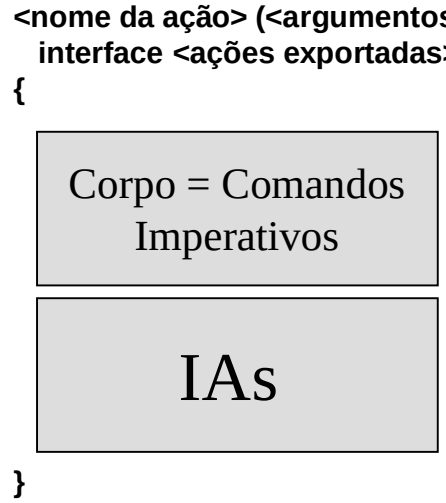


FIGURA 4.8 – Descrição de uma MIA

```
add_reverse_list(L1,L2,R)
  interface add_lists/3.
{
  add_lists(L1,L2,L3),
  reverse_list(L3,R).

  add_lists(L1,L2,R)
  {
    for C := 1 to length(L1) do
      R[C] := L1[C] + L2[C],
    return(R)
  }

  reverse_list(L1,R)
  {
    T := length(L1),
    for C := 0 to T-1 do
      R[T-C] := L1[C+1]
  }
}
```

FIGURA 4.9 – Exemplo de MIA

- **Ação Multiparadigma (*Multiparadigm Action* – MA)** : ação que suporta o encapsulamento de LAs e IAs. A figura 4.10 mostra a organização de uma MA. Em complemento, a figura 4.11 apresenta uma MA que realiza a mesma ação da MIA descrita na figura 4.9. No exemplo, a MA é composta de uma IA e duas LAs. Uma MA segue o mesmo padrão de visibilidade e acesso externo disponíveis nas MLAs e na MIAs. Além disso, o corpo de uma MA é composto de comandos imperativos.

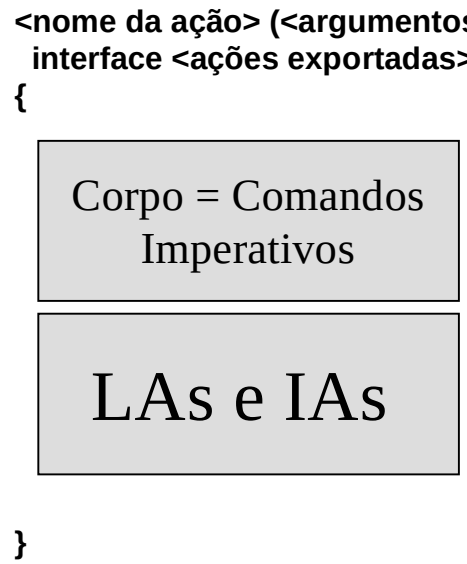


FIGURA 4.10 – Descrição de uma MA

```
add_reverse_list(L1,L2,R)
   interface add_lists/3.
{
  add_lists(L1,L2,L3),
  reverse_list(L3,R).

  add_lists(L1,L2,R)
  {
    for C := 1 to length(L1) do
      R[C] := L1[C] + L2[C],
    return(R)
  }

  reverse_list([],[]).
  reverse_list([H|L],L1,[H|R]) :-
    reverse_list(L,R1),
    append(R1,[H],R).

  append([],L,L).
  append([H|L],L1,[H|R]) :-
    append(L,L1,R).
}
```

FIGURA 4.11 – Exemplo de MA

A figura 4.12 mostra o **Grafo de Composição de Ações (*Actions Composition Graph - ACG*)**. O ACG regulamenta as possíveis composições de ações. Além disso, as invocações entre ações são regulamentadas através do **Grafo de Invocação de Ações (*Actions Invocation Graph* – AIG)** mostrado na figura 4.13. O AIG determina que LAs e MLAs não podem invocar IAs, MIAs e MAs. Os demais tipos de invocações são permitidos. Esta restrição estabelece duas regiões de execução: região lógica e região imperativa (figura 4.13). Após um fluxo de execução entrar na região lógica, a única forma de retorno para a região imperativa é a finalização do fluxo (retornando os resultados solicitados). Além disso, considerando que as LAs e MLAs são compostas de

CLEIs, fica garantido que não existe comportamento imperativo na região lógica. Esta garantia permite uma implementação simples e completa do não-determinismo baseado em *backtracking*. Isolando as LAs e MLAs, o AIG elimina diversos problemas oriundos da integração dos comportamentos imperativo e lógico. Peter Wegner [WEG 93, WEG 97] discute estes problemas e argumenta que esta integração é bastante complexa. O anexo 1.7 descreve as alterações em ambos os grafos para suporte ao paradigma funcional [DUB 2001].

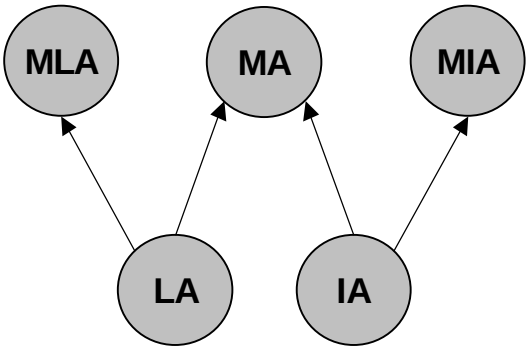


FIGURA 4.12 – Grafo de Composição de Ações (ACG)

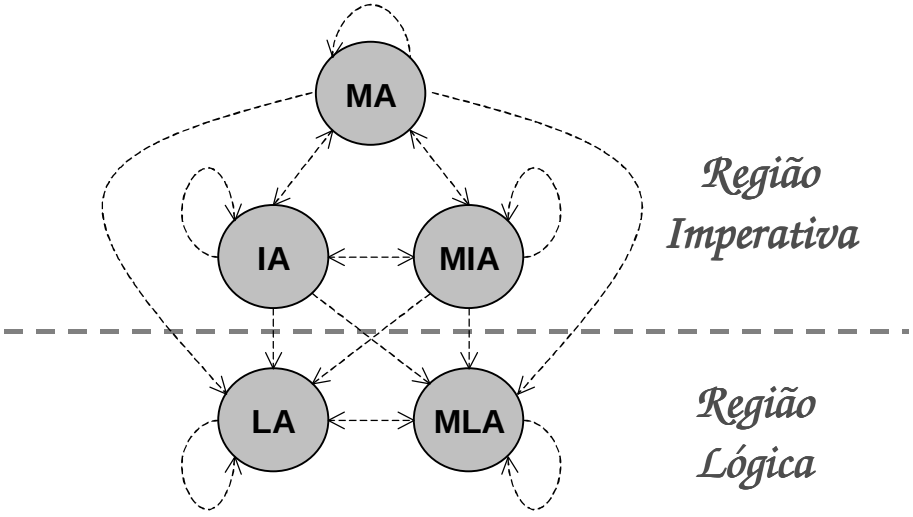


FIGURA 4.13 – Grafo de Invocação de Ações (AIG)

4.3 Tipos de invocação

Na Hololinguagem uma invocação é configurada usando uma **Estrutura de Configuração de Invocação** (**ECI**) composta de três campos (figura 4.14):

- **Identificador de destino (*Destiny Identifier - DI*)**: símbolo identificando o destino da invocação. O destino *default* é o próprio comportamento (interação do tipo 4, na figura 3.12), ou seja, se não houver DI, a invocação está sendo realizada para uma ação especificada no comportamento do próprio ente. O próprio comportamento pode ser representado pelo símbolo *behavior*. O DI pode identificar ainda a própria história (representada pelo símbolo *history*, interação do tipo 1), a própria interface (símbolo *interface*, interação do tipo 7) e um ente componente (símbolo contendo um nome de ente, interação do tipo 3);

- **Modo da invocação**: configura a invocação como uma afirmação (símbolo "***!***") ou uma pergunta (símbolos mostrados na tabela 4.1);
- **Informação simbólica**: informação a ser utilizada na invocação.

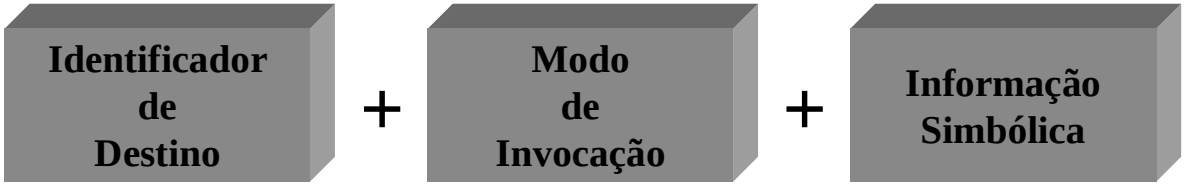


FIGURA 4.14 – Estrutura de Configuração de Invocação (ECI)

A utilização de ECIs é restrita às ações. Neste caso, CLEIs na história não podem usar ECIs. Holo suporta os modos de invocação descritos na seção 3.8 (afirmações e perguntas). Estes modos possuem as seguintes características:

- **Afirmação** (representada pelo símbolo "***!***"): uma afirmação é uma operação imperativa. Sendo assim, somente pode ser executada na região imperativa (figura 4.13). Uma afirmação é sempre não bloqueante, ou seja, não estabelece sincronismo e permite concorrência máxima. A afirmação permite a invocação assíncrona de uma ação (execução concorrente). Além disso, a afirmação pode ser utilizada para inserção de um AI na interface, uma ação no comportamento ou uma CLEI na história (configuração descrita na primeira coluna da tabela 4.2);
- **Pergunta** (resumo na tabela 4.1): uma pergunta pode ser bloqueante (*default*, símbolo "**.**" ou sem símbolo quando invocação de ação própria, tipo 4 na figura 3.12) ou não bloqueante (símbolo "**?**"). Uma pergunta bloqueante estabelece um ponto de sincronismo, isto é, a execução continua somente após a obtenção da(s) resposta(s). Uma pergunta não bloqueante obtém sucesso apenas se o destino possui a capacidade de resposta imediata (por exemplo, a história possui a CLEI solicitada). Além disso, uma pergunta pode ser destrutiva (símbolo "***#***") ou não (sem símbolo, opção *default*). Uma pergunta destrutiva *exige* a destruição no destino, da unidade (ação, AI ou CLEI) usada na sua manipulação. Perguntas destrutivas somente podem ser executadas na região imperativa (figura 4.13). Sendo assim, LAs e MLAs não podem perguntar de forma destrutiva. Finalmente, uma pergunta pode ser configurada com um **quantificador de respostas**. Este quantificador estabelece o número de respostas solicitadas em uma mesma pergunta. O quantificador pode ser configurado usando números e os símbolos "_" e "*". O símbolo "*" significa todas as respostas disponíveis. O símbolo "_" significa um intervalo (letra *I* na tabela 4.1). Por exemplo, "2_4" exige exatamente duas, três ou quatro respostas. Além disso, podem ser utilizados limites abertos. Por exemplo, "_3" exige no máximo três respostas e "3_" exige no mínimo três respostas. O número de respostas *default* é 1. Uma pergunta somente alcança sucesso, se todas as suas condições forem satisfeitas. Por exemplo, uma pergunta bloqueante somente retorna quando o número de respostas exigidas for alcançado. Uma pergunta não bloqueante, retorna sucesso apenas se as respostas exigidas estavam disponíveis imediatamente. Um pergunta destrutiva, destrói apenas se o número de respostas exigidas foi encontrado.

TABELA 4.1 – Tipos de perguntas

Tipos de Perguntas	**Não destrutiva (*default*)**	**Destrutiva #**	**Quantificador de Respostas (*I* ou *, default = 1)**
Bloqueante *(default)* **.**	**.** **(ou não especificado)**	**.#** **(ou somente #)**	***.I.*** **(não destrutiva)** ***#I.*** **(destrutiva)**
Não bloqueante **?**	**?**	**?#**	***?I.*** **(não destrutiva)** ***?#I.*** **(destrutiva)**

Os três campos da ECI podem ser determinados em tempo de execução (dinâmicos). Neste caso, devem ser utilizadas variáveis para configuração da invocação. Quando uma variável surge no primeiro (DI) ou no último campo, o modo de invocação atua como separador. Por exemplo, a seguinte ECI mantém em aberto o destino e a informação simbólica, mas estabelece que a invocação é uma pergunta bloqueante, não destrutiva, que exige somente uma resposta:

Destiny.Symbol

Uma configuração mais avançada pode estabelecer uma pergunta não bloqueante, destrutiva, que exige duas respostas:

Destiny?#2.Symbol

Além disso, o número de respostas pode ser dinâmico:

Destiny?#Quantifier.Symbol

A figura 4.15 mostra um pequeno exemplo em Holo. O programa (domínio estático na figura 3.14) é composto de um ente denominado *holo* (obrigatório em todos os programas). Este ente possui duas ações (LA e IA). A LA é a mesma apresentada na figura 4.2 (*append/3*). A IA contém um conjunto de invocações. A história é inicializada com cinco *fatos históricos*. Além disso, o ente não possui *interface*. A dinâmica de execução de programas em Holo será discutida na próxima seção. No entanto, uma rápida descrição da execução do exemplo torna-se útil. Uma ação que possui o mesmo nome do seu ente é denominada **ação guia**. Esta ação é executada automaticamente quando um ente dinâmico é criado através de uma clonagem de transição (semelhante aos construtores da orientação a objetos).

A execução sempre inicia com uma *clonagem de transição automática* (CTA, figura 3.14) do ente *holo* e, consequentemente, com a execução da sua ação guia. No exemplo, esta ação executa seis invocações (numeradas na figura 4.15), cinco delas direcionadas à própria história (invocações 1, 2, 3, 4 e 6) e uma ao próprio comportamento (invocação 5). A figura 4.15 mostra a evolução da história e o conteúdo

das variáveis, após cada invocação. As invocações atuam como funções, ou seja, estabelecem valores para as variáveis usadas na pergunta e retornam a resposta na própria invocação. Por exemplo, a invocação 1 retorna em *X* o símbolo *clinton* e na própria invocação o símbolo *father(clinton)*. Neste caso, a invocação não foi utilizada como uma função. A terceira invocação utiliza um quantificador de respostas (valor 2). Este caso é uma pergunta destrutiva e bloqueante, que exige duas respostas. Além disso, a invocação é usada como uma função. A invocação retorna uma lista contendo as respostas para a pergunta e a variável *F* recebe o símbolo relacionado com a primeira resposta. A quinta invocação é direcionada para o comportamento de *holo* (invocação da LA). Esta invocação não possui DI e não especifica seu modo. Portanto, é direcionada para o próprio comportamento (*default*), é bloqueante (*default*) e não destrutiva (*default*). Logo após, a sexta invocação insere na história o conteúdo da variável *R*. Este mesmo exemplo é utilizado na figura A1.17 para demonstração da programação funcional na Hololinguagem.

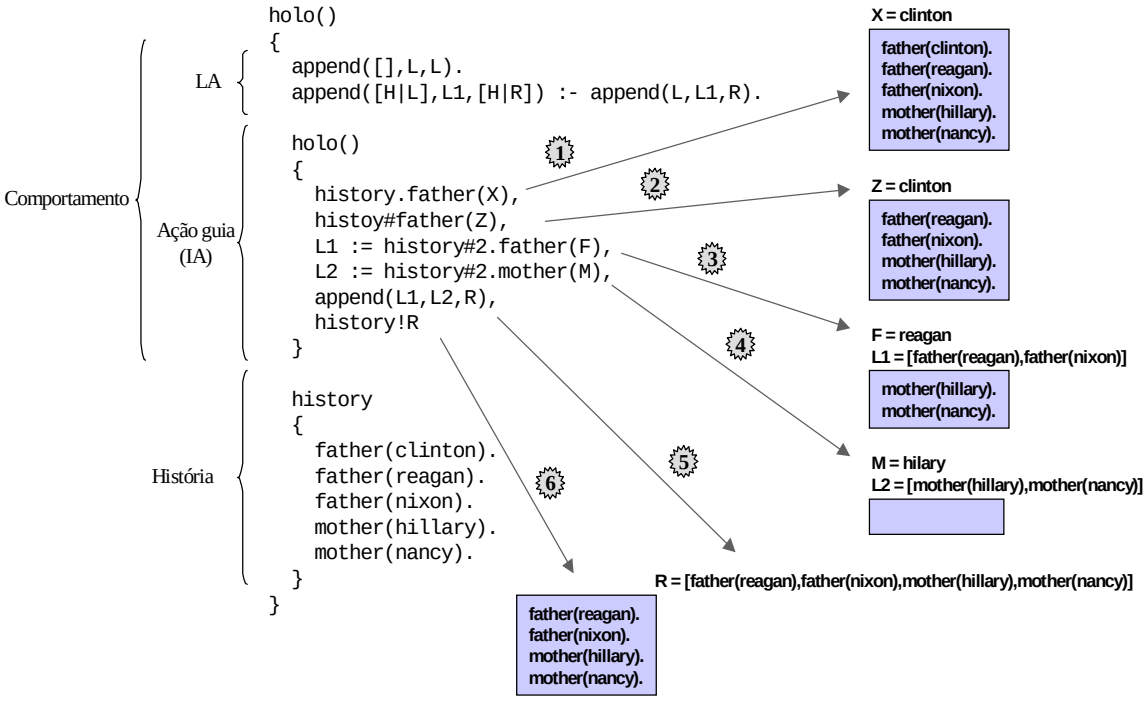


FIGURA 4.15 – Exemplos de configurações de invocações

Todas as configurações de invocação podem ser utilizadas para acesso à história, ao comportamento ou à interface (a tabela 4.2 mostra um resumo das possíveis configurações e suas características). Por outro lado, a **Unidade de Gerenciamento (UG)** da história é a CLEI, do comportamento é a ação e da interface é o AI. Por exemplo, uma afirmação para a história insere uma CLEI, enquanto uma afirmação para o comportamento pode executar ou inserir uma ação. Além disso, uma pergunta destrutiva para a história retira uma CLEI, enquanto uma pergunta destrutiva para o comportamento retira uma ação. Uma pergunta bloqueante direcionada para a história, bloqueia até a disponibilidade de uma CLEI que satisfaça a pergunta. Uma pergunta bloqueante para o comportamento bloqueia até que exista uma ação que responda à pergunta. A manipulação das UGs da interface e do comportamento é simples, pois as ações e os AIs são tratados de forma atômica. No entanto, a manipulação das UGs da história exige uma política. A política utilizada é denominada **FAFI** (***FActs FIrst***) [BAR 2001a]. A FAFI determina que os fatos históricos devem ser manipulados de

forma prioritária. Sendo assim, as CLEIs de um predicado histórico sempre são organizadas com os fatos no início, seguidos pelas regras. A busca de respostas inicia pelos fatos e, somente alcança as regras, se nenhum fato satisfizer a pergunta. Além disso, afirmações seguem a política FAFI, ou seja, os fatos são inseridos entre o último fato e a primeira regra. As regras são inseridas no final do predicado. Neste contexto, uma pergunta destrutiva merece atenção especial. Quanto feita ao comportamento, este tipo de pergunta destrói uma ação. Quando feita à interface, retira uma AI. Quando feita à história, destrói a CLEI que obteve sucesso (mesmo princípio do comando *retract* do Prolog).

TABELA 4.2 – Configuração de invocações

Tipo de Invocação / *Destino*	**Afirmação** **!**	**Pergunta não-destrutiva** **(*default*)**	**Pergunta destrutiva** **#**	**Pergunta bloqueante** **.**	**Pergunta não bloqueante** **?**
Interface	Insere uma AI. Se já existe uma AI idêntica, não realiza nenhuma alteração.	Retorna sucesso se existe uma AI na interface.	Verifica se existe uma AI na interface. Se existe, retira.	Verifica se existe uma AI na interface. Se não existe, aguarda sua inserção.	Verifica se existe uma AI na interface. Se não existe, retorna falha.
História	Insere uma CLEI. Se já existe um predicado histórico com a mesma identificação (nome e aridade), segue a política FAFI. Se não existe, cria um novo predicado.	Invoca um predicado histórico. Este tipo de invocação não altera a história.	Invoca um predicado histórico. Se obtiver sucesso, elimita a CLEI que resultou o sucesso.	Invoca um predicado histórico usando sincronismo imediato. Neste caso, a invocação aguarda até a pergunta ser respondida. Se não existe o predicado invocado, aguarda seu surgimento.	Invoca um predicado histórico, sem sincronismo, ou seja, se não existe o predicado invocado, retorna. Se existe, aguarda a resposta.
Comporta-mento	Executa ou insere uma ação. A execução é assíncrona. No caso da inserção, se existe uma ação com a mesma AI, sobrepõe. Se não existe, cria uma nova ação.	Invoca uma ação. Este tipo de invocação não altera o comportamento.	Invoca uma ação e solicita sua destruição. A destruição somente ocorre se houver sucesso na pergunta. Toda a ação é eliminada.	Invoca uma ação usando sincronismo imediato. Neste caso, a invocação aguarda até a pergunta ser respondida. Se não existe a ação invocada, aguarda seu surgimento.	Invoca uma ação, sem sincronismo. Neste caso, se não existe a ação invocada, retorna. Se existe, aguarda a resposta.

4.4 Ciclo existencial de um ente

O ciclo existencial de um ente consiste nas etapas percorridas durante sua existência. Este ciclo é composto de três etapas: criação, existência e extinção. As etapas existenciais são descritas a seguir e mostradas de forma esquemática na figura 4.16.

- **Criação**: um ente sempre é criado por outro ente. O ente que cria é denominado **ente criador**. Por sua vez, o ente que surge é denominado **ente criado**. Entes são criados através de clonagem (seção 4.5);
- **Existência**: após a criação, inicia a existência de um ente. Durante esse processo ele fica submetido à dinâmica dos entes. No seu interior podem ocorrer vários processos, tais como, criações, extinções e deslocamentos. Além disso, ele interage com os entes que existem ao seu redor e compartilha a história do ente no qual está inserido. Após a criação, um ente pode tornar-se **ativo** ou **reativo**. Se no comportamento existir uma ação com o mesmo nome do ente (**ação guia**), ela é executada automaticamente. Neste caso, o ente é chamado ativo e sua execução é controlada pela ação guia. Uma ação guia é opcional e deve conter apenas comportamento imperativo (IA ou MIA). Se não existir ação guia, o ente é chamado reativo e nenhuma ação é executada automaticamente. Neste caso, o ente aguarda invocações;
- **Extinção**: um ente deixa de existir quando ocorre sua extinção. Um ente pode ser extinto por outro ente (ação predefinida *extinguish*, seção 4.6). O ente que extingue é denominado **ente extintor**. Por sua vez o ente que deixa de existir é denominado **ente extinto**. Além disso, um ente pode se auto-extinguir, neste caso, ele é extintor e extinto. Este processo recebe o nome de **auto-extinção**. A extinção de um ente extingue automaticamente todos os seus entes componentes. Quando um ente é extinto, ocorre a execução automática da **ação de extinção**. Esta ação é opcional e deve conter comportamento imperativo (IA ou MIA). O nome da ação de extinção é padronizado (ação ***extinction***).

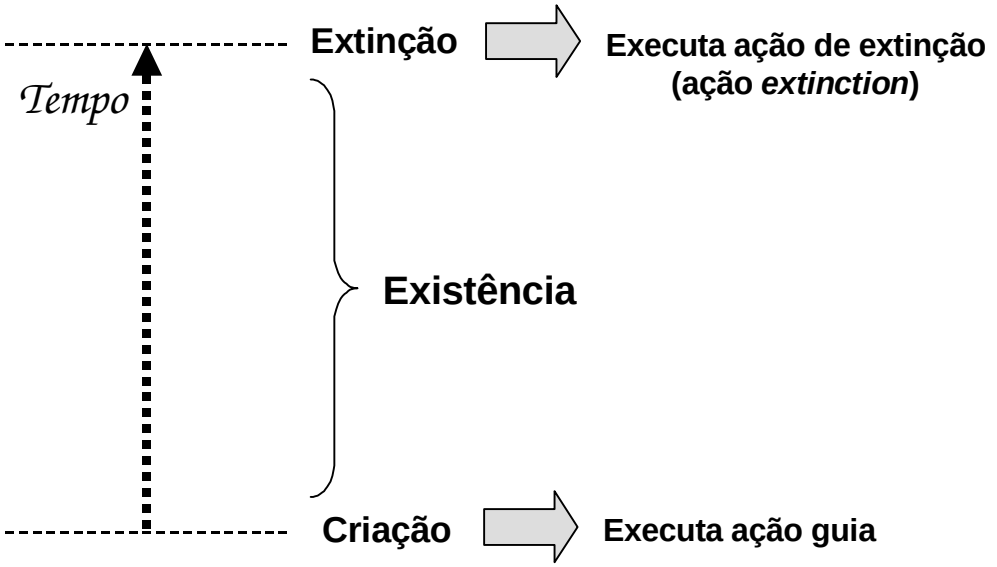


FIGURA 4.16 – Ciclo existencial de um ente

Um programa sempre possui um ente estático denominado *holo*. O ciclo existencial deste ente determina a dinâmica dos programas. Sendo assim, um programa em Holo é executado em três fases:

- **Fase de criação**: um programa sempre inicia com uma clonagem de transição automática (CTA) do ente estático *holo*. Este ente deve estar descrito no domínio estático. Cria-se assim um ente dinâmico que recebe o nome *d_holo* (*dynamic holo*);
- **Fase da existência**: o ente *d_holo* executa sua ação guia. A execução desta ação estabelece a dinâmica de execução do programa;
- **Fase de extinção**: um programa termina quando o ente *d_holo* é extinto.

4.4 Holoclonagem

Conforme descrito na seção 3.9, o Holoparadigma propõe três tipos de clonagem: estática, de transição e dinâmica. As clonagens de transição e dinâmica são realizadas em tempo de execução, com o uso do comando imperativo *clone* (seção 4.6). Por sua vez, a clonagem estática ocorre durante a descrição dos entes. Na cabeça da descrição de um ente (figura 4.1) pode ser inserida a palavra *cloning* acompanhada de um **Descritor de Clonagem** (*Cloning Descriptor – CD*) ou de uma **Lista de Descritores de Clonagem** (*Cloning Descriptor List – CDL*). Um CD indica um ente, seus argumentos (no caso de clonagem estática ou de transição) e, opcionalmente, uma seleção das partes a serem usadas na clonagem. O CD possui a seguinte organização:

```
<nome>(<arg 1>,...,<arg n>,<seleção>)
```

O `<nome>` indica o ente a ser utilizado na clonagem. Os campos `<arg>` são os argumentos. Por definição, a clonagem é completa, ou seja, todo o ente é clonado (interface, comportamento, história e componentes). O campo `<seleção>` é utilizado para seleção da clonagem. A seleção é uma lista contendo a indicação das letras iniciais das partes a serem clonadas (**i** = *interface*, **b** = *behavior*, **h** = *history*, **c** = *composition*). Por definição, toda clonagem é construtiva. Na seleção, o símbolo "#" indica uma clonagem destrutiva. Por exemplo, o seguinte CD estabelece a clonagem de um ente denominado *pessoa* que possui dois argumentos (por exemplo, *Nome* e *Idade*). Conforme indica o último campo, são clonados a interface e a história (de forma destrutiva). O comportamento e os componentes não são clonados.

```
pessoa(jorge,32,[i,h#])
```

A CDL suporta a clonagem múltipla proposta pelo Holoparadigma. Por exemplo, a seguinte CDL indica que o clone deve conter toda a estrutura de *ente1*, acrescida de forma construtiva da história de *ente2*, acrescida ainda do comportamento (de forma construtiva) e da história (de forma destrutiva) de *ente3*. Nenhum dos entes possui argumentos.

```
[ente1,ente2([h]),ente3([b,h#])]
```

A figura 4.17 exemplifica a clonagem múltipla seletiva. A figura mostra três entes (*ente1*, *ente2* e *ente3*). O *ente3* é um clone dos entes *ente1* e *ente2*. O exemplo mostra apenas os identificadores (<nome>/<aridade>) de ações no comportamento e de predicados na história. Um programa real contém descrições completas para ações e predicados (conforme mostrado na figura 4.15). A visualização apenas dos identificadores simplifica a compreensão do exemplo. Além disso, foram acrescidos comentários esclarecendo os tipos de ações envolvidas. A figura mostra ainda uma representação gráfica para visualização dos entes (**Diagrama de Ente**). A representação é composta de quatro campos que descrevem o ente, de acordo com a especificação mostrada na figura 4.18.

```
ente1()
interface a/2, e/2.
{
  a/2    /* ação lógica     */
  b/3    /* ação imperativa */
  c/1    /* ação lógica     */
  d/4    /* ação lógica     */
  e/2    /* ação imperativa */
  f/3    /* ação lógica     */
  history
  {
    g/2
    i/5
  }
 }

ente2()
interface a/2.
{
  a/2    /* ação lógica         */
  c/1    /* ação imperativa     */
  d/4    /* ação modular lógica */
  e/2    /* ação imperativa     */
  f/3    /* ação imperativa     */
  history
  {
    g/2
    h/4
  }
}

ente3() cloning([ente1,ente2([b,h#])])
interface a/2, f/3.
{
 a/2    /* ação lógica     */
 b/3    /* ação imperativa */
 f/3    /* ação lógica     */
 history
 {
   g/2
   i/5
 }
}
```

ente1
a/2, e/2
LA: a/2, c/1, d/4, f/3 IA: b/3, e/2
g/2, i/5

ente2
a/2
LA: a/2 IA: c/1, e/2, f/3 MLA: d/4
g/2, h/4

ente3 – [ente1,ente2([b,h#])]
a/2, e/2, f/3
LA: a/2 {ente1+ente2+ente3}, f/3 {ente1,ente3} IA: b/3 {ente3}, c/1 {ente2}, e/2 {ente1} MLA: d/4 {ente2}
g/2 {ente1 + ente3}, h/4 {ente2}, i/5 {ente1 + ente3}

FIGURA 4.17 – Exemplo de clonagem múltipla seletiva

As seguintes observações esclarecem o exemplo:

- **o clone resulta da união entre os clonados e sua própria especificação (sua descrição)**. Esta união é realizada de forma distinta entre as partes (interface, comportamento e história) dos entes envolvidos na clonagem, ou seja, a interface do clone é a união das interfaces. Por sua vez, o comportamento é a união dos comportamentos. Finalmente, a união das histórias resulta na história do clone;

- **cada parte de um ente pode ser representada por um conjunto de identificadores**. Portanto, a união realizada na clonagem pode ser percebida como uma operação de união de conjuntos [LIP 78, FIL 85]. Por exemplo, a interface especificada no *ente1* é {a/2, e/2} e no *ente3* é {a/2, f/3}. Portanto, a união resulta no conjunto {a/2, e/2, f/3}. Apenas uma cópia de cada identificador aparece no conjunto resultante (neste caso, apenas uma cópia de a/2). Os diagramas mostrados na figura 4.17 exemplificam essa situação;

- **a Hololinguagem suporta completamente a clonagem construtiva da história**. No exemplo, a história de *ente3* é formada pela clonagem construtiva da história de *ente1*, clonagem destrutiva da história de *ente2* e pela sua própria especificação de história. Portanto, sua história contém o predicado *g/2* resultante da integração (simbolizado pelo sinal "+" no diagrama de *ente3* na figura 4.17) de *g/2* de *ente1* (clonagem construtiva) e *g2* de *ente3* (especificação própria). O predicado *g/2* de *ente2* não é clonado, pois já existe uma versão de *g/2* no clone (clonagem destrutiva). Além disso, a história de *ente3* contém *h/4*, clonado de *ente2* (neste caso, não existe nenhuma outra versão de *h/4*, portanto a clonagem é realizada). Finalmente, a história de ente3 contém *i/5* resultante da clonagem construtiva da história de *ente1* e da especificação da história de *ente3*;

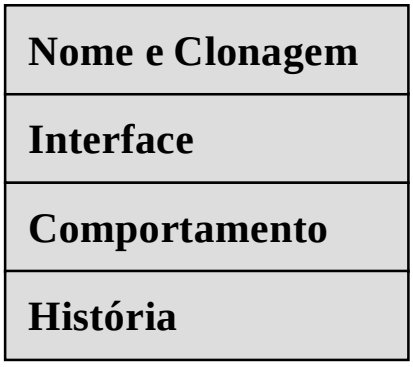


FIGURA 4.18 – Diagrama de ente

- no exemplo, o comportamento de *ente3* é formado pela sua própria especificação, acrescida do comportamento clonado de forma construtiva de *ente1* e *ente2*. **A Hololinguagem suporta apenas clonagem construtiva de LAs**. Por exemplo, a LA *a/2* de *ente3* é formada pela integração das LAs de *ente1*, *ente2* e do próprio *ente3*. A **Política de Clonagem de Comportamento** (*Behavior Cloning Policy – BCP*) regulamenta a clonagem de ações na Hololinguagem. A BCP estabelece quatro **Regras de Clonagem de Comportamento** (*Behavior Cloning Rules – BCR*). Além disso, determina que as BCRs possuem prioridade entre elas, ou seja, a BCR 1 é prioritária em relação a BCR 2 e assim sucessivamente. As BCRs são as seguintes:

1) *Quanto mais específico é o comportamento, mais prioritário ele se torna. Portanto, ações especificadas no clone possuem prioridade sobre ações clonadas.* No exemplo, a ação *b/3* de *ente3* é formada apenas pela especificação existente em *ente3*. A ação *b/3* de *ente1* não é clonada, pois não existe clonagem construtiva de IAs. De acordo com a BCR 1, a versão *b/3* de *ente3* recebe prioridade;

2) *O comportamento misto (MAs) possui prioridade em relação ao comportamento imperativo (IAs e MIAs) e o imperativo possui prioridade em relação ao lógico (LAs e MLAs)*. A figura 4.19 mostra a prioridade entre comportamentos. No exemplo, o *c/1* em *ente3* resulta apenas da clonagem do *c/1* de *ente2*, apesar de existir *c/1* também em *ente1*. No *ente1*, *c/1* é LA e no *ente2*, *c/1* é IA. Portanto, a clonagem construtiva não é possível. Além disso, a BCR 2 estabelece que o comportamento imperativo (*c/1* de *ente2*) é prioritário em relação ao comportamento lógico (*c/1* de *ente1*);

3) *Ações modulares (MIAs e MLAs) possuem prioridade sobre ações não modulares (IAs e LAs)*. A figura 4.19 mostra que no âmbito de cada comportamento, as ações modulares são prioritárias. No exemplo, *d/4* de *ente3* resulta somente da clonagem de *d/4* de *ente2*, apesar de existir *d/4* também em *ente1*. No *ente1*, *d/4* é LA e no *ente2*, *d/4* é MLA. Portanto, a clonagem construtiva não é possível. Além disso, a BCR 3 estabelece que ações modulares possuem prioridade sobre ações não modulares. Neste caso, *d/4* de *ente2* é clonado para o *ente3*;

4) *A ordem de especificação da clonagem estabelece a prioridade de ações*. No exemplo, a ação *e/2* de *ente3* resulta da clonagem da ação *e/2* de *ente1*, apesar de existir *e/2* também em *ente2*. A ação *e/2* é uma IA, tanto em *ente1* quanto em *ente2*. A clonagem construtiva não é possível. A BCR 4 estabelece que a ordem de especificação deve ser respeitada, portanto *e/2* de *ente1* tem prioridade (*ente1* aparece primeiro na especificação da clonagem no *ente3*).

- a organização das CLEIs na clonagem construtiva de LAs e da história, segue as BCRs 1 e 4 e a política FAFI (seção 4.3). Sendo assim, os fatos são colocados antes das cláusulas. Além disso, cada grupo (fatos e cláusulas) é organizado de forma que as CLEIs especificadas no clone são colocadas antes e as CLEIs dos clonados são organizadas de acordo com a especificação (cabeçalho do clonado);

- a formação da ação *f/3* no *ente3* é um exemplo interessante da aplicação conjunta das BCRs. Existem ações *f/3* nos três entes (no *ente1* é LA, no *ente2* é IA e no *ente3* é LA). A BCR 2 estabelece que no processo de clonagem, a IA de *ente2* possui prioridade sobre a LA de *ente1*. No entanto, na especificação de *ente3* existe uma LA, a qual tem prioridade em relação a IA de *ente2* (BCR 1). Neste caso, a clonagem construtiva pode ser aplicada. A *f/3* resultante em *ente3* é formada pela integração da *f/1* de *ente1* e *f/3* de *ente3*.

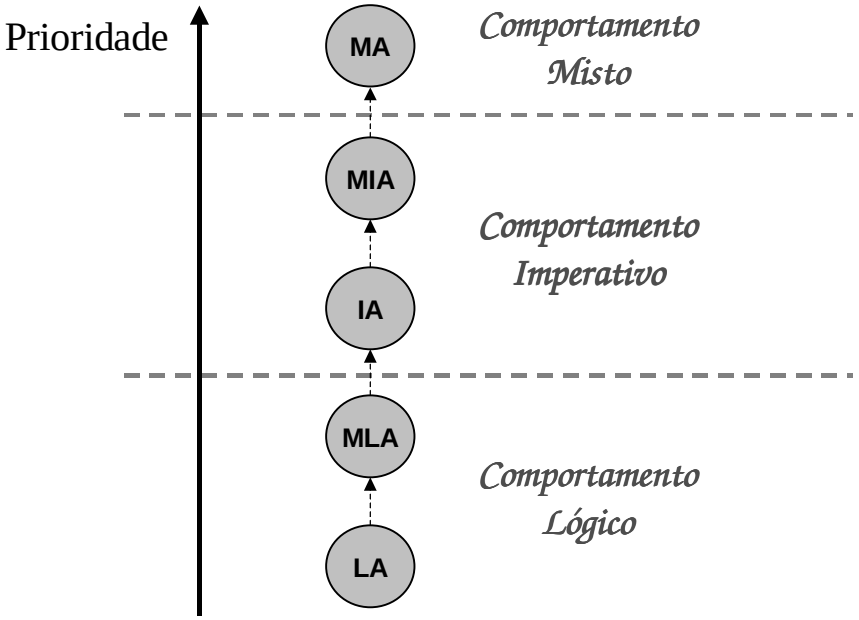


FIGURA 4.19 – Prioridade de ações na clonagem múltipla

4.5 Ações imperativas predefinidas

A Hololinguagem possui várias **Ações Imperativas Predefinidas** (AIP) que realizam atividades básicas. A maioria destas atividades envolve um ou mais entes. Sendo assim, torna-se necessária a utilização de um **Identificador de Ente** (*Being Identifier – BI*). O BI indica univocamente um determinado ente e possui a seguinte estrutura:

/<Nome de um ente>/<Nome de um ente>/.../<Nome de um ente>

O BI indica o caminho para acesso a um ente específico e assemelha-se ao nome de via usado em sistemas operacionais para indicação de arquivos em árvores de subdiretórios. O BI é utilizado apenas no domínio dinâmico. A primeira barra indica o nível zero de composição (figura 3.5c). Portanto, o ente *d_holo* possui um BI igual a "*/d_holo*". Cada barra adicional indica o aprofundamento em um nível. Existem dois tipos de BIs:

- **Absoluto**: indica o caminho completo, desde o nível zero. Por exemplo, na figura 4.20, o caminho "*/d_holo/rgs/pelotas/jorge*" identifica o ente *jorge* localizado no nível 3;
- **Relativo**: indica um caminho parcial, desde o nível onde é utilizado. Por exemplo, na figura 4.20 o caminho "*pelotas/jorge*" identifica o ente *jorge* localizado no interior do ente *pelotas*. Neste exemplo, o BI deve estar sendo utilizado no nível 2, ou seja, no interior de *rgs*.

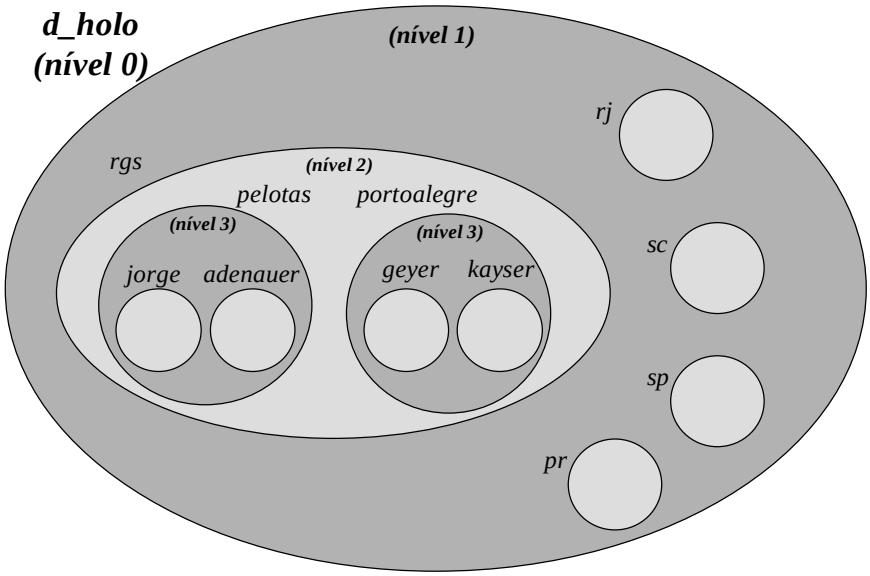


FIGURA 4.20 – Exemplo de identificadores de entes

A linguagem Holo usa dois símbolos predefinidos para composição de BIs e manipulação de entes:

- ***self***: indica o próprio ente onde o símbolo está sendo utilizado;
- ***out***: indica o nível acima do qual o símbolo está sendo utilizado. Por exemplo, na figura 4.20 o BI "*out/adenauer*" usado no comportamento de *jorge*, identifica o ente *adenauer* localizado no nível 3.

Entre as AIPs propostas pela Hololinguagem destacam-se (exemplificadas na seção 4.9):

- **ação *move***: implementa a mobilidade lógica de entes. Esta ação possui a sintaxe:

move(<BI origem>,<BI destino>)

A mobilidade lógica segue os princípios descritos na seção 3.5. As seguintes situações exemplificam o *move* no exemplo mostrado da figura 4.20:

1. ente *pelotas* (nível 2) executa em uma ação do seu comportamento o comando *move(jorge,out/portoalegre)*. Neste caso, *jorge* desloca-se para o interior de *portoalegre*;
2. ente *rgs* executa o comando *move(pelotas,portoalegre)*. Neste caso, o ente *pelotas* desloca-se para o interior de *portoalegre*. Todos os entes componentes de *pelotas* acompanham o deslocamento.

A ação *move* atua como uma função, ou seja, retorna resultados na própria ação. Se a mobilidade foi realizada com sucesso, a ação retorna o símbolo *success*. Se a mobilidade não foi possível retorna *fail*.

- **ação *clone***: permite a clonagem de entes. O comando *clone* é utilizado para clonagem de transição e clonagem dinâmica. A sintaxe do comando é:

clone(<clonado(s)>,<clone>)

O primeiro argumento pode ser um CD ou uma CDL (seção 4.5). Além disso, este argumento pode indicar entes estáticos e/ou dinâmicos. No caso de entes dinâmicos, os CDs podem usar BIs. No caso de entes estáticos, a clonagem deve indicar os argumentos. Se a clonagem envolver apenas entes estáticos, ocorrerá somente clonagem de transição. Se envolver somente entes dinâmicos, ocorrerá somente clonagem dinâmica. Se envolver ambos, ocorrerá uma **clonagem mista**. O argumento <*clone*> indica o nome do clone. As seguintes situações exemplificam o comando *clone* no ente da figura 4.20:

1) ente *pelotas* (nível 2) executa a ação *clone(jorge,clone_jorge)*. Esta é uma clonagem dinâmica. Logo após o comando, existirá um ente denominado *clone_jorge* no interior de *pelotas*, idêntico (interface, comportamento e história corrente) ao ente *jorge*;

2) ente *rgs* (nível 1) executa a ação *clone(pelotas,piratini)*. Após a ação, existirá um novo ente no interior de *rgs* (no nível 2). Conforme descrito na seção 3.9 e mostrado na figura 3.14, o clone possui a mesma composição do clonado, ou seja, os entes *pelotas* e *piratini* possuem a mesma composição. Neste caso, no domínio dinâmico passam a existir entes com o mesmo nome (por exemplo, *jorge* em *pelotas* e em *piratini*). Apesar de terem o mesmo nome, seus BIs são distintos (*/d_holo/rgs/pelotas/jorge* e */d_holo/rgs/piratini/jorge*). Além disso, os entes *jorge* estão encapsulados em *pelotas* e *piratini*. Portanto, possuem contextos de atuação diferentes;

3) ente rgs (nível 1) executa a ação *clone([pelotas([i,b,c]), cidade(100.000,[h])], bage)*. Neste caso, *cidade* seria um ente estático e 100.000 seria seu argumento (número de habitantes). Este exemplo mostra uma clonagem mista que cria um ente *bage* contendo a interface, o comportamento e a composição de *pelotas* (ente dinâmico) e a história de *cidade* (ente estático).

A ação *clone* atua como uma função, ou seja, retorna resultados na própria ação. Se a clonagem foi realizada com sucesso, a ação retorna o nome simbólico do clone. Se a clonagem não foi possível retorna *fail*. Além disso, se o segundo argumento não for indicado (por exemplo, *clone(jorge,_)*, a ação retorna um nome simbólico gerado pelo sistema.

- **ação *extinguish***: permite a extinção de entes. A extinção de um ente ocasiona a extinção de todos os seus entes componentes. A sintaxe da ação é:

extinguish(<extinto(s)>)

O argumento *<extinto(s)>* pode ser o nome ou uma lista de nomes de entes. Somente entes dinâmicos podem ser extintos. Por exemplo, no ente mostrado na figura 4.20, as seguintes situações são possíveis:

1. ente *pelotas* executa a ação *extinguish(jorge)*. Neste caso, o ente *jorge* é extinto. Se a seguir, *pelotas* acionar *extinguish(adenauer)*, o ente *pelotas* passará a ser um ente elementar (sem composição). O mesmo efeito seria causado pela ação *extinguish([jorge,adenauer])*;

2. ente *rgs* executa o comando *extinguish(pelotas)*. Neste caso, o ente *pelotas* e toda a sua composição (entes componentes) são extintos.

A ação *extinguish* atua como uma função, ou seja, retorna resultados. Se a extinção foi realizada com sucesso, a ação retorna o símbolo *success*. Se a extinção não foi possível retorna *fail*.

- **ação *spawn***: executa uma ou mais ações de forma concorrente. A sintaxe é:

spawn(<concorrente(s)>)

O argumento *<concorrente(s)>* pode ser uma ação ou uma lista de ações. Conforme descrito na seção 4.3, uma afirmação para o comportamento pode ser utilizada para execução concorrente de uma ação. A ação predefinida *spawn* é uma versão simplificada de uma afirmação. Afirmações podem ser feitas para comportamentos de outros entes, ou seja, através de uma afirmação pode ser executada uma ação concorrente em outro ente (interações dos tipos 3 e 5). A ação *spawn* está restrita ao comportamento do próprio ente (interação do tipo 4). Portanto, a equivalência é a seguinte:

spawn(<concorrente(s)>) ≡ behavior!<concorrente(s)>

A ação *spawn* retorna sucesso se todas as ações concorrentes foram criadas. Se uma ação não puder ser criada, *spawn* falha e nenhuma ação é criada. A ação *spawn* é assíncrona. Após a criação das ações (sucesso de *spawn*), o fluxo de execução continua. Seguindo o padrão estabelecido por Bosschere e Tarau [BOS 96, p.55], uma chamada de ação concorrente pode ter variáveis, mas no entanto, nenhum resultado é retornado através delas. Esta restrição estabelece que não existirá compartilhamento de variáveis entre ações concorrentes. A história pode ser utilizada na sincronização e comunicação entre ações.

4.6 Operadores multiparadigma

As linguagens utilizam operadores para manipulação de suas entidades de programação (variáveis, constantes, etc). Destacam-se dois grupos de operadores: atribuição/unificação e relacionais. As linguagens imperativas trabalham com operadores de **atribuição** (por exemplo, Pascal usa "**:=**", C e Java usam "**=**"). Uma atribuição é destrutiva, ou seja, substitui o valor já existente pelo valor atribuído. Esta característica é herdada da arquitetura Von Neumann, onde a memória e os registradores suportam apenas atribuições destrutivas. Por outro lado, o paradigma em lógica utiliza **unificação** (algumas vezes chamada atribuição construtiva).

O Holoparadigma utiliza o símbolo como unidade de informação. Sendo assim, a Hololinguagem propõe o uso conjunto dos operadores de atribuição e unificação. Além disso, os operadores relacionados são direcionados para as entidades propostas pelo Holoparadigma. A tabela 4.3 apresenta os operadores usados em Holo.

TABELA 4.3 – Operadores multiparadigma

Nome	Símbolo	Sintaxe	Descrição
Operador de **Atribuição**	:=	`variável := expressão`	A variável recebe, de forma destrutiva, o valor da expressão
Operador de **Unificação**	=:=	`termo =:= termo`	Unifica os termos
Operador Relacional **Idêntico**	==	`termo == termo`	Retorna sucesso se os termos forem idênticos
Operador Relacional **Unifica**	=	`termo = termo`	Retorna sucesso se os termos unificarem
Operador Relacional **Diferente**	\==	`símbolo \== símbolo`	Negação do operador "=="
Operador Relacional **Não Unifica**	\=	`termo \= termo`	Negação do operador "="
Operador Relacional **Maior**	>	`símbolo_1 > símbolo_2`	Retorna sucesso se *símbolo_1* for maior que *símbolo_2*
Operador Relacional **Menor**	<	`símbolo_1 < símbolo_2`	Retorna sucesso se *símbolo_1* for menor que *símbolo_2*
Operador Relacional **Maior ou Igual**	>=	`símbolo_1 >= símbolo_2`	Retorna sucesso se *símbolo_1* for maior ou igual a *símbolo_2*
Operador Relacional **Menor ou Igual**	<=	`símbolo_1 <= símbolo_2`	Retorna sucesso se *símbolo_1* for menor ou igual a *símbolo_2*

4.7 Gramática básica

A figura 4.21 mostra uma versão simplificada da gramática da Hololinguagem [BAR 99a, p.83; BAR 2001a] usando uma BNF. Nesta versão existem construções que foram abstraídas, permitindo uma visão geral e simples da organização da linguagem. Os símbolos não-terminais destacados com letras maiúsculas representam os pontos de abstração. As últimas nove produções descrevem estes símbolos.

Encontra-se no anexo 5 uma versão completa da gramática utilizada para criação da HoloJava (seção 5.1) [BAR 2001c]. Esta gramática é constituída da BNF acrescida de ações semânticas usadas para conversão de Holo para Java.

```
<holoprogram> -> <holodeclaration> ( <static_being> )*

<holodeclaration> -> <holohead> “{“ <holobody> “}”

<holohead> -> “holo” “(“ [ <variable_list> ] “)”
              [ “cloning” “(“ <cdl> “)” ] [ “interface” <ai_list> “.” ]

<holobody> -> <holoaction> ( <action> )* [ <history> ]

<holoaction> -> “holo” “(“ [ <variable_list> ] “)” “{“ [ <COMMANDS> ] “}”

<static_being> -> <head> “{“ <body> “}”

<head> -> <SYMBOL> “(“ [ <variable_list> ] “)”
          [ “cloning” “(“ <cdl> “)” ] [ “interface” <AIList> “.” ]

<body> -> <main_being> ( <action> )* [ <history> ]

<main_being> -> <SYMBOL> “(“ [ <variable_list> ] “)” “{“ [ <COMMANDS> ] “}”

<action> -> <IA> | <LA> | <MIA> | <MLA> | <MA>

<history> -> “history” “{“ ( CLEI )* “}”

<variable_list> -> <VARIABLE> ( “,” <VARIABLE> )*

<cdl> -> <cd> | “[“ <cd> ( “,” <cd> )* “]”

<cd> -> <SYMBOL> [ “(“ <conf_cd> “)” ] | “holo” [ “(“ <conf_cd> “)” ]

<conf_cd> -> <option> ( “,” <option> )*

<option> -> <SYMBOL> | <NUMBER> | “holo” | “[“ <select> (“,” <select>)* “]”

<select> -> “i” | “b” | “b#” | “h” | “h#” | “c” | “c#”

<ai_list> -> <SYMBOL> “/” <NUMBER> ( “,” <SYMBOL> “/” <NUMBER> )*


<SYMBOL> -> { Conjunto de letras e dígitos, iniciando com uma letra minúscula }

<VARIABLE> -> { Conjunto de letras e dígitos, iniciando com uma letra maiúscula }

<NUMBER> -> { Conjunto de dígitos }

<COMMANDS> -> { BNF para avaliação de uma seqüência de comandos (if,for,writeln, etc) }

<IA> -> { BNF para avaliação de uma ação imperativa }

<LA> -> { BNF para avalição de uma ação lógica }

<MIA> -> { BNF para avaliação de uma ação modular imperatica }

<MLA> -> { BNF para avaliação de uma ação modular lógica }

<MA> -> { BNF para avaliação de ação multiparadigma }
```

FIGURA 4.21 – Gramática básica da Hololinguagem

4.8 Exemplos

Esta seção apresenta sete exemplos de programas criados usando a Hololinguagem. O início de cada seção contém comentários sobre os exemplos. Além disso, foram inseridos comentários diretamente nos códigos fonte. Os exemplos estão organizados nas seguintes seções: **semáforos** (seção 4.9.1), ***buffers*** (seção 4.9.2), **jantar de filósofos** (seção 4.9.3), **mineração simples** (seção 4.9.4) e **mineração paralela** (seção 4.9.5).

4.8.1 Semáforos

A figura 4.22 mostra uma implementação simples de semáforos em Holo. Um exemplo semelhante pode ser encontrado em [BOS 96, p.58]. A figura 4.23 aperfeiçoa o exemplo, permitindo que seja utilizado como semáforo, um símbolo escolhido pelo programador. Finalmente, a figura 4.24 mostra um ente gerenciador de semáforos, o qual pode ser utilizado para manipulação de diversos semáforos inicializados pelo programador. Os seguintes comentários auxiliam a compreensão dos exemplos:

- os exemplos mostram apenas trechos de código. O início de cada exemplo apresenta como seria a utilização destes trechos;
- os exemplos mostrados nas figuras 4.23 e 4.24 não necessitam de nenhuma inicialização da história;
- o exemplo da figura 4.24 mostra um ente completo, contendo ações e interface. O ente não possui história inicializada. Além disso, o comentário inserido no cabeçalho mostra a utilização da ação predefinida *clone*;
- nos três exemplos, o sistema de invocação implícita do *blackboard* gerencia o bloqueio para entrada na seção crítica (ação ***p***).

```
/* As acoes podem acessar a secao critica usando “p” e “v”:
       ....
       p,
       “SECAO CRITICA”,
       s,
       ....
*/


holo()
{
  p()
  {
    history#token       // Pergunta bloqueante e destrutiva para a historia.
  }

  v()
  {
   history!token       // Afirmacao para a historia. Insere “token” na historia.
  }

  history
  {
    token.             // Historia inicializada com “token” utilizado como semaforo.
  }
}
```

FIGURA 4.22 – Programa “Semáforo simples”

```
/* As acoes podem inicializar semaforos:
      ....
      init(token),
      ....
   e depois utiliza-los:
      ....
      p(token),
      "SECAO CRITICA",
      v(token),
      ....
*/


holo()
{
  init(Token)
  {
    history!Token
  }

  p(Token)
  {
    history#Token
  }

  v(Token)
  {
   history!Token
  }
}
```

FIGURA 4.23 – Programa "Semáforos inicializados"

```
/* Outros entes podem clonar o ente estatico:
       ....
       clone(semaphore_manager,my_manager),
       ....
   e depois utiliza-lo para inicializacao de diferentes semaforos:
       ....
       my_manager.init(sem1),
       my_manager.init(sem2),
       ....
   e acesso a diferentes secoes criticas:
       ....
       my_manager.p(sem1),
       "SECAO CRITICA 1",
       my_manager.v(sem1),
       ....
       my_manager.p(sem2),
       "SECAO CRITICA 2",
       my_manager.v(sem2),
       ....
*/

semaphore_manager()
  interface init/1,p/1,s/1.
{
  init(Token)
  {
    history!Token
  }

  p(Token)
  {
    history#Token
  }

  v(Token)
  {
   history!Token
  }
}
```

FIGURA 4.24 – Programa "Ente gerenciador de semáforos"

4.8.2 *Buffers*

A figura 4.25 exemplifica um ente gerenciador de *buffers*. O texto [BOS 96, p.58] mostra o mesmo exemplo implementado em Multi-BinProlog. Merecem destaque os seguintes comentários:

- o exemplo contém apenas o ente gerenciador. O comentário inserido no cabeçalho mostra como utilizá-lo;
- o ente não possui história inicializada pelo programador. Além disso, a história é utilizada para armazenamento de todos os *buffers* gerenciados por um ente. Podem ser criados vários gerenciadores;
- o ente exporta suas três ações através da interface;
- o sistema de invocação implícita do *blackboard* é utilizado para gerenciamento do tamanho do *buffer*. Sempre que uma leitura for solicitada a um *buffer* vazio, ocorre um bloqueio até a inserção de um dado. Por outro lado, a inserção em um *buffer* cheio, bloquea até que ocorra um consumo de um dado.

```
/* Outros entes podem clonar o ente estatico:
       ....
       clone(buffers_manager,my_manager),
       ....
   e depois utiliza-lo para inicializacao de diversos buffers:
       ....
       my_manager.init(idade,5),         ---> Cria um buffer "idade" de tamanho 5
       my_manager.init(universidade,3),  ---> Cria um buffer "universidade" de tamanho 3
       ....
   e uso dos buffers:
       ....
       my_manager.put(idade,pedro(32)),     ---> Coloca "pedro(32)" no buffer "idade"
       my_manager.put(idade,maria(24)),     ---> Coloca "maria(24)" no buffer "idade"
       my_manager.put(universidade,ufrgs),  ---> Coloca "ufrgs" no buffer "universidade"
       my_manager.put(universidade,ufsm),   ---> Coloca "ufsm" no buffer "universidade"
       ....
       my_manager.get(idade,pedro(Idade),   ---> Idade = 32
       my_manager.get(universidade,X),      ---> X = ufrgs
       my_manager.get(universidade,ufsm),   ---> Avança sem bloqueio
       my_manager.get(universidade,puc),    ---> Bloquea até "puc" ser inserida
       ....
*/

buffers_manager()
  interface init/1,put/2,get/2.
{
  init(Name,Size)            // Inicializa um buffer com determinado nome e tamanho.
  {
    for Cont := 1 to Size do
    {
      history!Name           // Insere na historia o nome do buffer.
    }
  }

  put(Name,Data)
  {
    history#Name,             // Retira da historia um nome de buffer.
    history!data(Name,Data)   // Insere na historia um simbolo com o nome do bufer,
  }                           // e o dado a ser bufferizado.

  get(Name,Data)
  {
    history#data(Name,Data),
    history!Name
  }
}
```

FIGURA 4.25 – Programa "Ente gerenciador de *buffers*"

4.8.3 Jantar de filósofos

A figura 4.26 mostra o clássico problema *Jantar de Fisólofos* implementado em Holo. O texto [AND 92] apresenta três soluções (centralizada, distribuída e decentralizada) implementadas na linguagem SR. A linguagem SR não suporta *blackboards*. Além disso, o texto [BOS 96, p.59] mostra uma solução em Multi-BinProlog. Merecem destaque os seguintes comentários:

- o exemplo mostra um programa completo;
- o número de filósofos é constante e igual a cinco;
- o programa utiliza uma ação predefinida de saída (*writeln*);
- a ação guia *holo* utiliza o comando *spawn* para execução concorrente de cinco versões da ação *philo*. Além disso, a ação *holo* mostra como pode ser realizada uma pergunta múltipla usando uma lista;
- a ação *random* gera um número aleatório entre 0 e o número especificado no seu argumento. A ação *sleep* bloquea a execução até a passagem de um tempo especificado em milisegundos no seu argumento;
- a história é inicializada com quatorze fatos históricos;
- a ação *philo* mostra uma afirmação múltipla usando uma lista. Esta afirmação insere na história os fatos listados. Além disso, mostra uma pergunta múltipla que solicita quatro fatos diferentes.

```
holo()
{
  holo()
  {
    writeln('O jantar dos cinco filosofos esta comecando... '),        // Escreve na tela uma mensagem.
    spawn([philo(1,4),philo(2,5),philo(3,4),philo(4,10),philo(5,2)]), // Executa cinco acoes concorrentes.
    history#[end,end,end,end,end],                    // Uma forma mais simplificada seria "history#5.end"
    writeln('O jantar terminou...')
  }

  philo(Ident,Times) // Acao imperativa. "Ident" = Identificador,  "Times = Quantas vezes filosofo come.
  {
    writeln('Filosofo ',Ident,' esta jantado!'),
    for Cont := 1 to Times do                              // Conta o numero de vezes que filosofo come.
    {
      sleep(random(10000)),                                   // Filosofo pensando.
      history#[ticket,seat(F1,F2),chopstick(F1),chopstick(F2)], // Filosofo senta e pega talheres.
      sleep(random(10000)),                                   // Filosofo comendo.
      history![chopstick(F2),chopstick(F1),seat(F1,F2),ticket], // Filosofo libera talheres e levanta.
      writeln('Contagem do filosofo ',Ident,': ',Cont)
    }
    history!end                                               // Indica que filosofo terminou de comer.
  }

  history        // Historia inicializada. Neste exemplo, a historia representa a mesa de jantar.
  {
    chopstick(1). chopstick(2). chopstick(3). chopstick(4). chop stick(5). // Talheres
    ticket. ticket. ticket. ticket.                                        // Autorizacao para sentar
    seat(1,2). seat(2,3). seat(3,4). seat(4,5). seat(5,1).                 // Cadeiras
  }
 }
```

FIGURA 4.26 – Programa "Jantar de filósofos"

4.8.4 Mineração simples

A figura 4.27 apresenta um programa que simula uma mineração de dados (*datamining*) [BER 97]. Este exemplo também é utilizado no capítulo 5 para avaliação da ferramenta HoloJava (seção 5.1) e do ambiente DHolo (seção 5.3). Merecem destaque os seguintes comentários:

- o exemplo mostra um programa completo formado por três entes estáticos (*holo*, *mine* e *miner*). No ente *miner* existem duas ações: uma LA (*append/3*) e uma IA (ação guia *miner/1*). Nenhum dos entes possui história inicializada, mas o ente *miner* possui interface (exporta *append/3*);
- o programa cria um determinado número de minas (ente *mine*) e um mineiro (ente *miner*). O número de minas é um parâmetro na linha de comando (parâmetro *Many*). O mineiro entra nas minas (comando *move*), realiza a mineração e armazena o resultado na história de *d_holo* (ente dinâmico criado através da CTA de *holo*). Os dados manipulados são listas criadas aleatoriamente e o processamento realizado pelo mineiro consiste na concatenação das listas;
- o mineiro utiliza a ação *move* para entrar nas minas. O exemplo mostra como a visão de contexto do mineiro muda após cada execução de *move*. O acesso à história do seu composto (interação do tipo 2 na figura 3.12) depende da sua localização;
- listas são criadas através da primeira atribuição a uma variável. O ente *holo* mostra a criação de uma lista vazia (variável *List*). Além disso, listas podem ser criadas com um tamanho estabelecido através da ação predefinida *create_list* (veja ação guia de *mine*). Neste caso, os elementos da lista são inicializados com o símbolo vazio (símbolo *void*);
- no exemplo, o acesso ao exterior de um ente é realizado com a ação predefinida *out* (interações do tipo 2, 3, 5 e 8 na figura 3.12). A notação é `out(<SÍMBOLO DO DESTINO EXTERNO>)`. O ente *miner* utiliza este recurso uma vez para o deslocamento (ação *move*) e duas vezes para acesso a história do composto no qual está localizado (interação do tipo 2). Este tipo de acesso é sensível ao contexto, ou seja, depende da movimentação de um ente. No exemplo, o primeiro acesso externo (*out* no *while*) é direcionado para a história das minas. O segundo acesso externo (*out* no final do *for*) é direcionado para a história de *d_holo*;
- a inserção do símbolo "#" antes de uma variável em um argumento e stabelece que a chamada será realizada com uma variável aberta e o valor obtido será substituído pelo já existente. Este processo equivale ao uso de uma variável temporária seguida de uma atribuição, ou seja:

  `...,chamada(Temp),Argumento := Temp,...` ≡ `...,chamada(#Argumento),...`

- o ente *holo* invoca uma ação lógica localizada em *miner* (ação *append/3*) para concatenação de listas. Esta invocação demonstra como é realizada uma interação do tipo 3, quando a fonte é uma ação do comportamento do composto e o destino é uma ação de um ente componente.

```
holo()                                          // Ente principal
{
  holo(Many)                                    // "Many"=Quantidade de minas. Parametro de linha de comando.
  {
    for Cont := 1 to Many do                    // Cria  "Many" entes dinamicos, baseados no ente estatico
      clone(mine,Mines[Cont]),                  // "mine". Seus nomes sao colocados na lista "Mines".
                                                // A lista e' criada pela primeira atribuicao.
    clone(miner(Mines),my_miner),               // Cria um ente "my_miner" baseado em "miner".
    for Cont := 1 to Many do                    // Aguarda que  "my_miner" insira na historia o resultado
      history.list(Mines[Cont],Result[Cont]),   // da garimpagem. O resultado e colocado na lista "Result".
    List := [],                                 // Cria uma lista vazia.
    for Cont := 1 to Many do                    // Solicita a concatenacao das listas para "my_miner".
      my_miner.append(List,Result[Cont],#List), // Chama a acao "append" em "my_miner". Interacao do tipo 3,
                                                // na figura 3.12.
    history!list(self,List)                     // Insere a lista "List" na historia usando o formato:
  }                                             // "list(<MEU NOME>,<Lista>)" com todos os dados garimpados.
}

miner()                                         // Ente minerador.
  interface append/3.
{
  append([],L,L).                               // LA para concatenacao de listas.
  append([H|L],L1,[H|R]) :- append(L,L1,R).

  miner(Mines)                                  // Acao guia de "miner". "Mines" e uma lista contendo o nome
  {                                             // das minas a serem exploradas. Neste exemplo, os nomes
    for Cont := 1 to length(Mines) do           // foram escolhidos automaticamente pelo acao "clone" no ente
    {                                           // "holo".
      move(self,out(Mines[Cont])),              // Desloca o próprio ente (simbolo "self") para o interior
      List_result := [],                        // de uma mina.
      while out(history)?#list(#List) do        // Repete enquanto forem encontrados dados na historia da
        append(List_result,List,#List_result),  // mina. Invocacao não bloqueante destrutiva (simbolos "?#").
      move(self,out),                           // "out(history)" acessa a historia do composto no qual o
      out(history)!list(Mines[Cont],List_result) // o ente esta inserido (interacao do tipo 2 na figura 3.12).
    }                                           // O segundo "move" retorna o "miner" para um nivel acima
  }                                             // (simbolo "out"). O segundo "out(history)" acessa a
}                                               // historia de d_holo (versao dinamica do ente holo).

mine()
{
  mine()
  {
    for Cont_lists := 1 to random(9)+1 do       // Cria um numero randomico de listas. No minimo uma lista.
    {
      List := create_list(random(9)+1),         // Cria uma lista com tamanho randomico. No minimo tamanho 1
      for Cont := 1 to length(List) do          // Preenche a lista com valores randomicos.
        List[Cont] := random(5),
      history!list(List)                        // Insere a lista na sua historia com o formato padrao:
    }                                           // "list(<LISTA>)"
  }
}
```

FIGURA 4.27 – Programa "Mineração simples"

4.8.5 Mineração paralela

A figura 4.28 mostra uma simulação de mineração semelhante à apresentada na seção 4.9.4. A única diferença consiste na criação de um mineiro para cada mina. Desta forma, torna-se possível a mineração paralela. Destacam-se as seguintes considerações:

- na ação guia *holo*, todas as minas são criadas antes dos mineiros. Assim, quando o primeiro mineiro desloca-se para o interior de uma mina, já estão disponíveis os dados a serem minerados;
- a concatenação final realizada pelo ente *holo* utiliza a LA do primeiro minerador criado;
- se cada mina for colocada em um nodo de uma arquitetura distribuída, a mobilidade lógica realizada pelos mineiros se tornará uma mobilidade física e a mineração ocorrerá em paralelo. A execução paralela ou distribuída depende do ambiente de execução, ou seja, a mobilidade física é transparente. A criação de vários mineiros apenas viabiliza o paralelismo. A seção 5.3 exemplifica essa situação no contexto do DHolo.

```
holo()                                          // Ente principal
{
  holo(Many)                              // "Many" = Quantidade de minas e mineiros. Parametro de linha de comando .
  {
    for Cont := 1 to Many do                       // Cria as minas.
      clone(mine,Mines[Cont]),
    for Cont := 1 to Many do                       // Cria os mineiros apos a criacao das minas. Permite assim
      clone(miner(Mines[Cont]),Miners[Cont]),      // que as minas preencham suas historias antes da mineracao.
    for Cont := 1 to Many do                       // Aguarda que os mineiros insiram na historia os resultados
      history.list(Mines[Cont],Result[Cont]),      // da mineração. Os resultados são agrupados em "Result".
    Final := [],                                   // Cria uma lista vazia.
    for Cont := 1 to Many do
      Miners[1].append(Final,Result[Cont],#Final), // Chama a acao "append" do primeiro mineiro.
    history!list(self,Final)                       // Insere a lista "Final" na historia usando o formato:
  }                                                // "list(<MEU NOME>,<Lista>)" com todos os dados garimpados.
}

miner()                                            // Ente minerador.
  interface append/3.
{
  append([],L,L).                                  // LA para concatenacao de listas.
  append([H|L],L1,[H|R]) :- append(L,L1,R).

  miner(Mine)                                      // Acao guia de "miner". "Mine" contem o nome da mina a ser
  {                                                // explorada.
    move(self,out(Mine)),                          // Desloca o próprio ente (simbolo "self") para o interior
    List_result := [],                             // da mina.
    while out(history)?#list(#List) do              // Repete enquanto forem encontrados dados na historia da
      append(List_result,List,#List_result),       // mina. Invocacao não bloqueante destrutiva (simbolos "?#").
    move(self,out),                                // "out(history)" acessa a historia do composto no qual o
    out(history)!list(Mine,List_result)            // o ente esta inserido (interacao do tipo 2 na figura 3.12).
  }                                                // O segundo "move" retorna o "miner" para um nivel acima
}                                                  // (simbolo "out"). O segundo "out(history)" acessa a
                                                   // historia de d_holo (versao dinamica do ente holo).
mine()
{
  mine()
  {
    for Cont_lists := 1 to random(10)+1 do        // Cria um numero randomico de listas. No minimo uma lista.
    {
      List := create_list(random(10)+1),          // Cria uma lista com tamanho randomico. No minimo tamanho 1
      for Cont := 1 to length(List) do            // Preenche a lista com valores randomicos.
        List[Cont] := random(5),
      history!list(List)                          // Insere a lista na sua historia com o formato padrao:
    }                                             // "list(<LISTA>)"
  }
}
```

FIGURA 4.28 – Programa "Mineração paralela"

4.9 Considerações finais

Este capítulo apresentou a Hololinguagem, uma linguagem de programação baseada nos conceitos propostos no capítulo 3. As principais constatações alcançadas no capítulo foram:

- a Hololinguagem implementa os principais conceitos propostos pelo Holoparadigma (distinção entre mobilidades lógica e física, clonagem múltipla e seletiva, integração dos comportamentos lógico e imperativo, integração das invocações implícita e explícita, níveis de entes encapsulando *blackboards*);
- os cinco tipos de ações (IA, LA, MLA, MIA e IA) propostas suportam a implementação dos comportamentos lógico e imperativo;
- a utilização do ACG regula a composição das ações. Além disso, o AIG regula a invocação entre elas, isolando as LAs e MLAs e criando uma região lógica durante a execução (figura 4.13);
- o uso de CLEIs na história e nas ações que implementam o comportamento lógico, associado ao isolamento de LAs e MLAs através do AIG, simplifica a implementação do não-determinismo via *backtracking*;

- a política FAFI permite o gerenciamento de fatos e regras na história;
- a regulamentação da clonagem múltipla e seletiva do comportamento depende do uso de uma política (BCP) baseada em regras de clonagem (BCRs);
- operadores multiparadigma envolvendo atribuição e unificação, podem ser utilizados para manipulação conjunta de símbolos no âmbito dos comportamentos imperativo e lógico.

O próximo capítulo apresenta a Holoplataforma. Esta plataforma concretiza parte das idéias discutidas nos capítulos 3 e 4.

5 Holoplataforma

Este capítulo descreve a criação da primeira plataforma de desenvolvimento e execução para a Hololinguagem. A Holoplataforma possui três componentes (figura 5.1):

- **HoloJava** (seção 5.1) [BAR 2001a, BAR 2001c]: uma ferramenta para conversão de holoprogramas para programas em Java;
- **HoloEnv** (seção 5.2) [BAR 99a, SOA 2000]: um ambiente integrado de desenvolvimento (gerenciamento de arquivos, edição, compilação e execução);
- **DHolo** (seção 5.3) [BAR 2001, BAR 2001b]: um ambiente para execução distribuída de programas.

Conforme mostra a figura, atualmente a Holoplataforma suporta o ciclo de desenvolvimento e execução em ambientes centralizados. Ainda não existe suporte à execução distribuída. No entanto, foi criado um projeto inicial do DHolo. Além disso, foram realizadas simulações para avaliação das possíveis plataformas a serem utilizadas na sua futura implementação (seção 5.3).

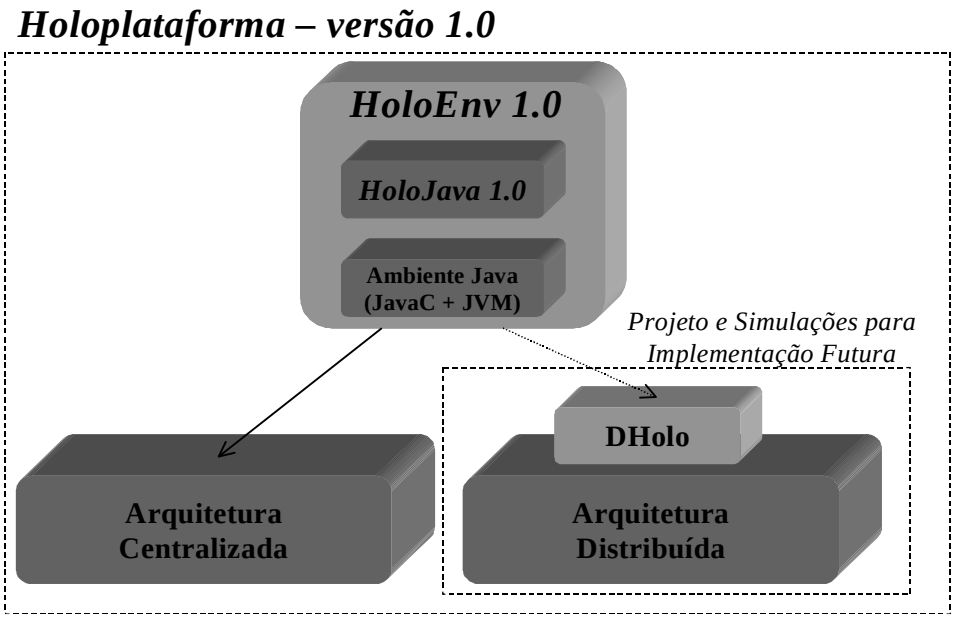


FIGURA 5.1 – Holoplataforma - versão 1.0

5.1 HoloJava

O desenvolvimento de uma plataforma completa (seção 3.10) para a execução de programas em Holo envolve a criação de uma máquina virtual multiparadigma, a especificação de seu código virtual e a construção de um compilador. Esta tarefa demanda recursos (pesquisa, tempo de projeto e implementação, etc) que tornam inviável sua realização neste trabalho. Sendo assim, as diversas partes de uma plataforma completa serão consideradas atividades futuras.

Visando a realização de testes de holoprogramas no menor espaço de tempo possível, foi criada uma ferramenta de **conversão de programas** [PET 96] denominada **HoloJava** [BAR 2001a, BAR 2001c]. Esta ferramenta converte programas em Holo (linguagem origem) para programas em Java (linguagem destino). Esta técnica é bastante usada na avaliação de novas linguagens, pois a utilização de uma linguagem destino que possua uma plataforma (compilação e execução) já disponível, antecipa a realização de testes e a difusão de resultados da pesquisa.

Diversos estudos [HAJ 96, PLJ 2001] indicam que Java pode ser utilizada como linguagem intermediária na construção de compiladores. No âmbito do Holoparadigma, a utilização de Java como linguagem destino está relacionada com as seguintes constatações:

- a linguagem Java vem sendo bastante utilizada em ambientes acadêmicos e tem servido de instrumento para concretização de pesquisas. No grupo de pesquisa onde o Holo está sendo desenvolvido, diversos trabalhos usam Java ([GEY 2001], ReMMoS [FER 2001, FER 2001a], DEPAnalyzer [AZE 1001, AZE 2001a], DOBuilder [MAL 2001]);
- Java suporta diretamente várias abstrações do Holoparadigma (ações imperativas, concorrência, etc). No entanto, algumas características de Holo não são suportadas (ações lógicas, *blackboard*, mobilidade, etc). Por outro lado, existem bibliotecas que complementam a plataforma Java e podem ser utilizadas para implementação dos aspectos não suportados. Por exemplo, ações lógicas em Prolog Café [BAN 99, PRO 2001a], história em Jada [CIP 2001] ou JavaSpaces [FRE 99] e mobilidade física em Voyager [VOY 2001] ou Horb [HIR 98, HOR 2001];
- existem estudos que demonstram que Java é uma solução adequada como linguagem intermediária [HAJ 96, PLJ 2001];
- Java está sendo utilizada na implementação da HoloJava, na construção do ambiente HoloEnv (seção 5.2) e nas simulações do DHolo (seção 5.3).

A figura 5.2 apresenta o contexto de desenvolvimento da HoloJava. A criação da ferramenta está sendo realizada com a utilização de JavaCC [JAV 2001], um gerador de analisadores sintáticos que gera código Java. A entrada do JavaCC é uma gramática Holo acrescida de ações semânticas responsáveis pela conversão (veja o anexo 5). Java não possui suporte para implementação de todas as características propostas pelo Holoparadigma. Portanto, a conversão torna-se limitada e o uso de bibliotecas complementares torna-se necessário.

As principais limitações impostas a versão 1.0 da HoloJava são as seguintes:

- apenas ações imperativas (IAs) e ações modulares lógicas (MLAs) são suportadas. As demais ações (LAs, MIAs e MAs) não podem ser usadas;
- a história gerencia apenas fatos históricos, ou seja, não são permitidas cláusulas;
- apenas a clonagem de transição é suportada (veja figura 3.14). Além disso, não existe suporte a clonagem múltipla e a clonagem seletiva (veja figura 3.15);
- apenas perguntas bloqueantes não destrutivas (símbolo “.”, veja tabela 4.1) podem ser utilizadas para chamada de ações;

- somente concorrência inter-entes (entre entes) é suportada. A concorrência intra-ente (entre ações de um ente) não pode ser usada. Sendo assim, o comando *spawn* e afirmações para ações não são permitidas (veja seção 4.6);
- os tipos de interação 6, 7 e 8 não são suportados (veja figura 3.12);
- não existe controle de interface, portanto, todas as ações de um ente estão disponíveis para os demais.

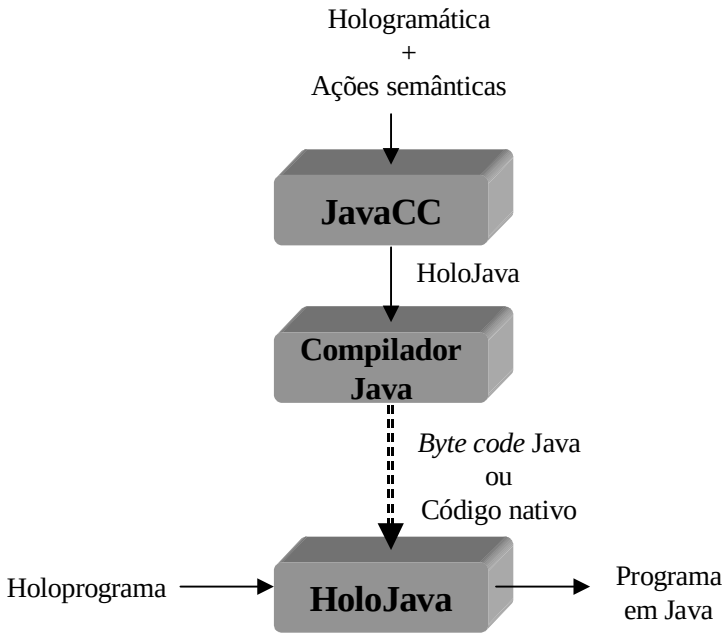


FIGURA 5.2 – Contexto de criação da HoloJava

As seguintes bibliotecas estão complementando a linguagem Java:

- **Prolog Café** [BAN 99, PRO 2001a]: sistema que suporta a conversão de Prolog para Java;
- **Jada** [CIP 2001]: biblioteca que permite a implementação de espaços de objetos (*blackboards*) em Java.

A figura 5.3 mostra a **política de conversão** usada na HoloJava. Esta política consiste no mapeamento de estruturas de holoprogramas para estruturas de um programa em Java. Conforme mostra a figura, Jada e Prolog Café são utilizadas para a concretização da história e ações lógicas. Por sua vez, a concorrência inter-entes é suportada através de *threads*. Entes compostos são tratados como grupos [BIR 93, LIA 90, NUN 98, LEA 2001] e a mobilidade lógica é implementada usando técnicas de *membership* [BIR 93, p. 41; LEA 2001, p.7].

Um ente composto equivale a um grupo de entes e a política de grupo usada pela HoloJava possui as seguintes características:

- **Hierárquica**: um grupo é representado por um ente composto que encapsula os membros (entes componentes). Além disso, um ente pode ser composto de entes compostos. Sendo assim, a política é hierárquica;
- **Dinâmica**: os membros de um grupo mudam durante a execução. As mudanças são guiadas pela clonagem (ação *clone*) e mobilidade (ação *move*). Portanto, a política é dinâmica;
- **Completamente fechada**: a troca de mensagens (interação do tipo 5, na figura 3.12) é restrita aos membros de um grupo, ou seja, a visibilidade de um ente é o seu grupo. A palavra "completamente" foi utilizada para diferenciação dos grupos fechados descritos por Nunes [NUN 98, p.11]. Os grupos fechados suportam troca de mensagens entre todos os membros de todos os grupos. No entanto, somente um membro de um grupo pode mandar uma mensagem para o grupo (*broadcast* implícito para todos os membros do grupo).

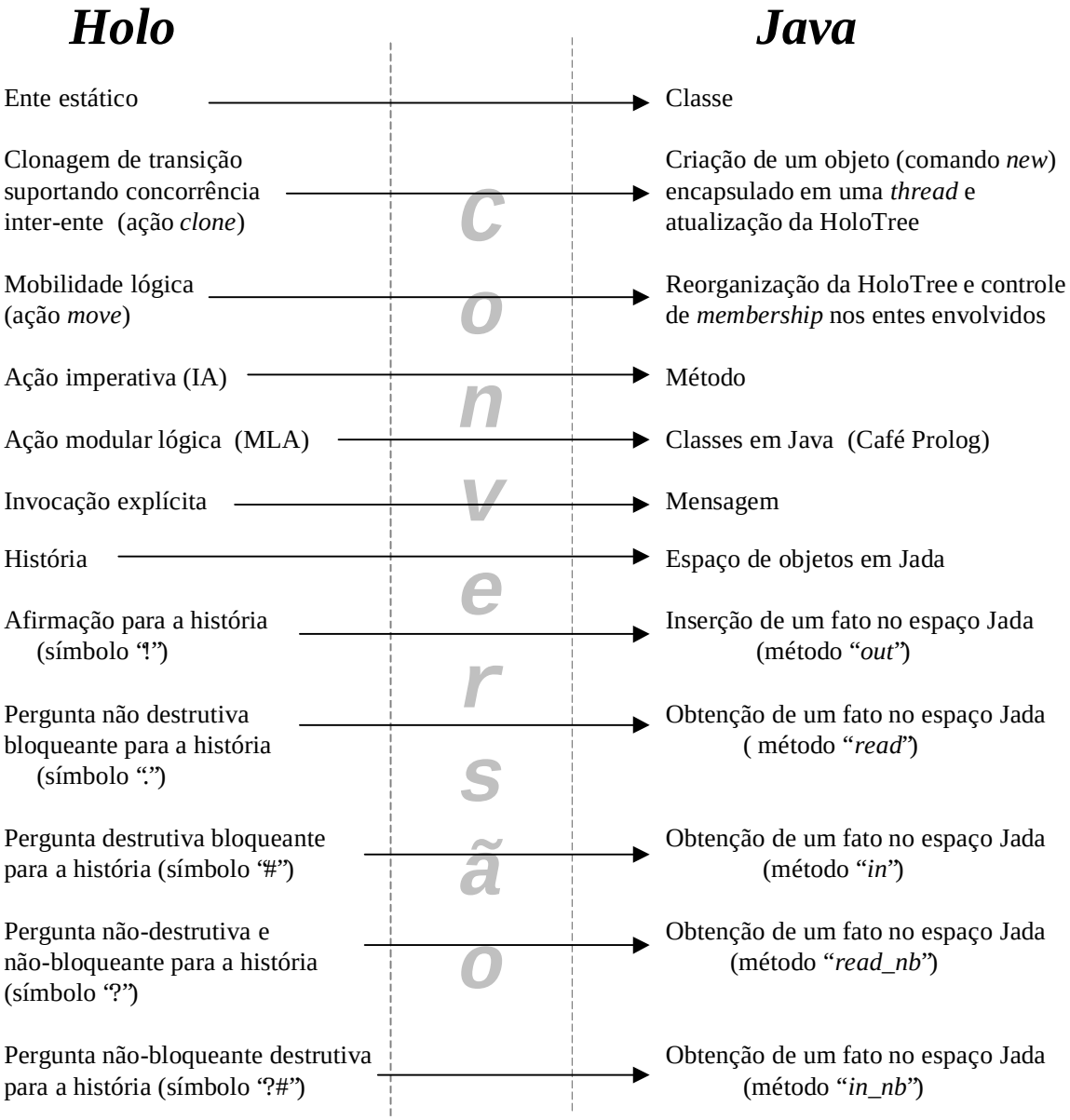


FIGURA 5.3 – Política de conversão de Holo para Java

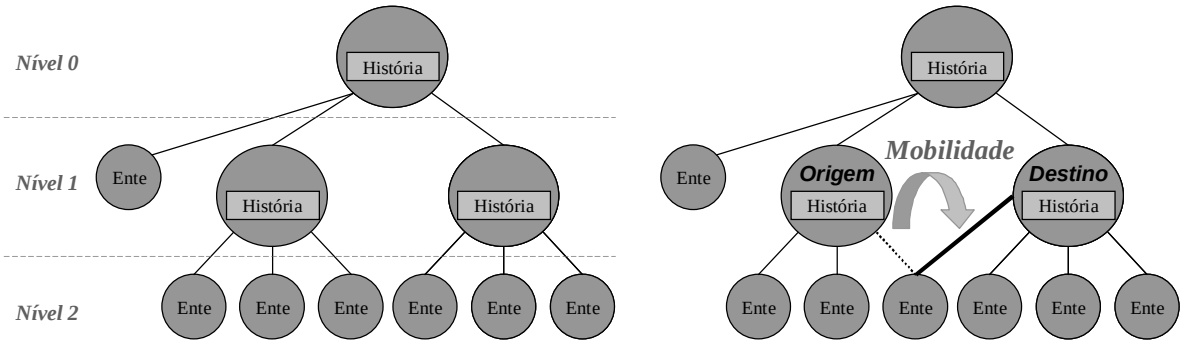
A execução de um Holoprograma cria uma estrutura hierárquica de entes, denominada **Árvore de Entes** (***HoloTree***) [BAR 2001b]. A HoloTree implementa o encapsulamento de entes em níveis de composição, conforme proposto pelo Holoparadigma (veja figura 3.5c). Além disso, a HoloTree suporta o aspecto dinâmico da política de grupos, mudando continuamente durante a execução de um programa. As seguintes ações alteram a HoloTree:

- **Clonagem**: a clonagem de transição cria um novo ente no interior de quem a executa (um nível abaixo na HoloTree);
- **Mobilidade**: a mobilidade altera a HoloTree, deslocando um ente (elementar ou composto) de um ponto para outro da árvore.

A figura 5.4a exemplifica a HoloTree para a organização em níveis mostrada na figura 3.5c. Um ente composto possui ligações com seus entes componentes, os quais ficam localizados no nível abaixo. Os entes componentes possuem acesso à história (interação do tipo 2, figura 3.12) e ao comportamento (tipo 3, figura 3.12) do composto no qual estão inseridos. Por sua vez, o composto possui acesso aos comportamentos dos seus componentes (tipo 3). Além disso, um ente possui acesso ao comportamento dos demais entes no mesmo nível (tipo 5).

A mobilidade é implementada através do gerenciamento de grupo seguindo princípios de controle de *membership* [BIR 93, p.41; LEA 2001, p. 7]. Quando a mobilidade ocorre, torna-se necessária a mudança da **visão do grupo** [NUN 98, p.21] dos entes envolvidos. O ente movido possui uma nova visão (novos entes no mesmo nível e um novo composto acima dele). Se o movido for um ente composto, a visão dos seus componentes não muda.

A mobilidade implica a atualização da composição dos entes origem e destino. Além disso, os componentes de um ente podem ser entes compostos (grupos de grupos [LEA 2001, p. 8]). A mobilidade de um ente elementar equivale a realocação de uma folha da árvore e a mobilidade de um composto transfere um ramo contendo vários entes. A figura 5.4b apresenta a mudança que ocorreria na HoloTree para a mobilidade mostrada na figura 3.7a.



(a) Árvore de Entes (HoloTree) (b) Mobilidade

FIGURA 5.4 – Árvore de entes (HoloTree)

A figura 5.5 mostra um programa utilizado para demonstração da HoloJava. O programa implementa uma mineração (*datamining* [BER 97]) simplificada envolvendo mobilidade. O *datamining* é uma das principais aplicações previstas para o Holo (seção 6.1). Um programa semelhante é utilizado nas simulações do ambiente distribuído (seção 5.3). O programa contém diversos comentários.

```
/********************  ENTE PRINCIPAL ********************
holo()     // Ente principal.
{
  holo()   // Acao guia.
  {
    writeln('HOLO: Vou criar tres minas e um mineiro'),
    clone(mine(1),mine_d1),  // Cria a primeira mina. O parametro identifica a mina.
    clone(mine(2),mine_d2),  // Cria a segunda mina.
    clone(mine(3),mine_d3),  // Cria a terceira mina.
    clone(miner,miner_d),    // Cria o mineiro.
    for X := 1 to 3 do       // Aguarda pelos resultados da mineracao.
    {
      history#list(#Ident,#Num,#Fibo),   // Obtem o resultado de uma mineracao.
      writeln('HOLO: Terminou a mineracao da mina:',Ident),
      writeln('HOLO: Fibonacci de ',Num,' e ',Fibo)
    }
    writeln('HOLO: Terminou a mineracao')
  }
}
//********************  ENTE MINA *******************
 mine()
{
  mine(Ident)
  {
    writeln('MINA ',Ident,': Fui criada')
  }

  history         // A historia da mina de cada mina possui
  {               // o mesmo conteudo. O primeiro numero
    list(1,2).    // identifica a mina. O segundo e o numero
    list(2,4).    // usado para o calculo do Fibonacci.
    list(3,6).
  }
}
//********************  ENTE MINEIRO ******************
miner()
{
  miner()
  {
    writeln('MINEIRO: Inicio da mineracao.'),
    move(self,mine_d1),                      // Passo 1  - Entra na mina 1
    mining(1,Num1,Res1),                     // Passo 2  - Minera a mina 1
    move(self,out),                          // Passo 3  - Sai da mina 1
    out(history)!list(1,Num1,Res1),          // Passo 4  - Salva o resultado 1
    move(self,mine_d2),                      // Passo 5  - Entra na mina 2
    mining(2,Num2,Res2),                     // Passo 6  - Minera a mina 2
    move(self,out),                          // Passo 7  - Sai da mina 2
    out(history)!list(2,Num2,Res2),          // Passo 8  - Salva o resultado 2
    move(self,mine_d3),                      // Passo 9  - Entra na mina 3
    mining(3,Num3,Res3),                     // Passo 10 - Minera a mina 3
    move(self,out),                          // Passo 11 - Sai da mina 3
    out(history)!list(3,Num3,Res3),          // Passo 12 - Salva o resultado 3
    writeln('MINEIRO: Fim da mineracao.')
  }

  mining(Ident,Num,Result)           // IA que realiza a mineracao.
  {
   out(history)#list(Ident,#Num),   // Minera a historia externa.
   fib(Num,#Result)                 // Chama a MLA para determinar Fibonacci.
  }

  fib/2()          // Acao Modular Logica (MLA) que calcula Fibonacci.
  {
    fib(1,1).
    fib(2,1).
    fib(M,N) :-
       M > 2,
       M1 is M-1,
       M2 is M-2,
      fib(M1,N1),
      fib(M2,N2),
      N is N1+N2.
  }
}
```

FIGURA 5.5 – Programa de mineração *datamining.holo*

As seguintes observações descrevem o exemplo:

- o programa é composto de três entes estáticos (*holo*, *mine* e *miner*);
- o ente dinâmico *d_holo* (criado automaticamente para o ente *holo*) cria três minas e um mineiro. Logo após, aguarda os resultados serem colocados na sua história;
- as minas (entes *mine_d1*, *mine_d2* e *mine_d3*) notificam sua criação (ação guia) e aguardam a mineração. A história das minas contém três fatos. Cada fato guarda a identificação da mina e um número que indica qual o *Fibonacci* deverá ser calculado pelo mineiro;
- o mineiro (ente *miner_d*) entra em uma mina, realiza a mineração, sai da mina e insere o resultado na história de *d_holo*. Estes passos são executados para cada mina. A figura 5.6 mostra os doze passos do mineiro, os quais estão destacados no fonte do programa através de comentários. A mineração consiste na busca de um valor para cálculo de *Fibonacci*. O cálculo é realizado pelo mineiro usando uma MLA.

O exemplo possui as seguintes características que merecem atenção especial:

- todos os entes são concorrentes, ou seja, são criadas cinco *threads* (uma para *d_holo*, uma para o mineiro e três para as minas);
- o ente minerador possui três ações, duas IAs e uma MLA. A MLA exemplifica o uso de ações lógica em Holo;
- o ente minerador é responsável pelo controle de sua mobilidade (ação *move*);
- os passos 2, 6 e 10 do mineiro utilizam o acesso à história externa (interação do tipo 2 na figura 3.12). Este acesso é realizado na ação *mining*. Apesar do código em *mining* ser o mesmo nos três passos, o *out(history)* é sensível ao contexto (mina) no qual o mineiro está em determinado momento.

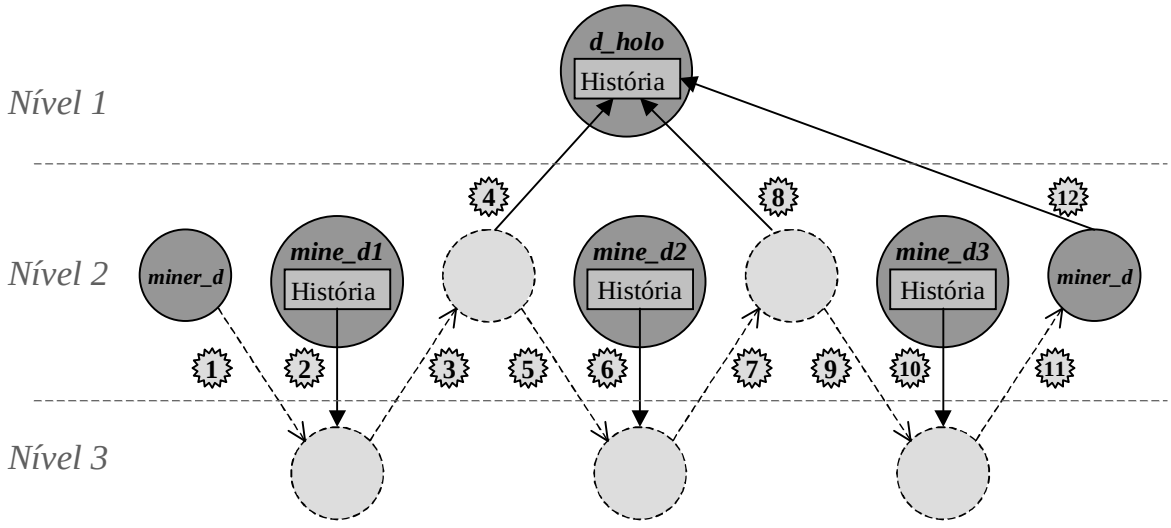


FIGURA 5.6 – Passos de mineração (*datamining.holo*)

A figura 5.7 mostra as quatro etapas envolvidas na conversão e execução de holoprogramas. O programa *datamining.holo* (figura 5.5) é utilizado no exemplo. As etapas são as seguintes:

- **Etapa 1** – **Conversão HoloJava:** esta etapa é realizada pela HoloJava. A conversão de um holoprograma gera um ou mais arquivos Java (*.java*). Um dos arquivos é denominado *holo.java*. Este arquivo contém o método *main*, onde iniciará a execução. Além disso, se houverem MLAs no programa, são gerados arquivos Prolog (*.pl*). A política de geração de arquivos é mostrada na figura 5.8;
- **Etapa 2 – Conversão Prolog Café:** esta etapa é realizada pelo Prolog Café. Os arquivos Prolog são convertidos para arquivos Java;
- **Etapa 3** – **Compilação dos arquivos Java:** nesta etapa, os arquivos Java são compilados. Após a compilação resultam os arquivos *.class*. O arquivo *holo.class* contém o ponto de partida para a execução do programa;
- **Etapa 4 – Execução do programa:** após a conversão e compilação, pode ser realizada a execução.

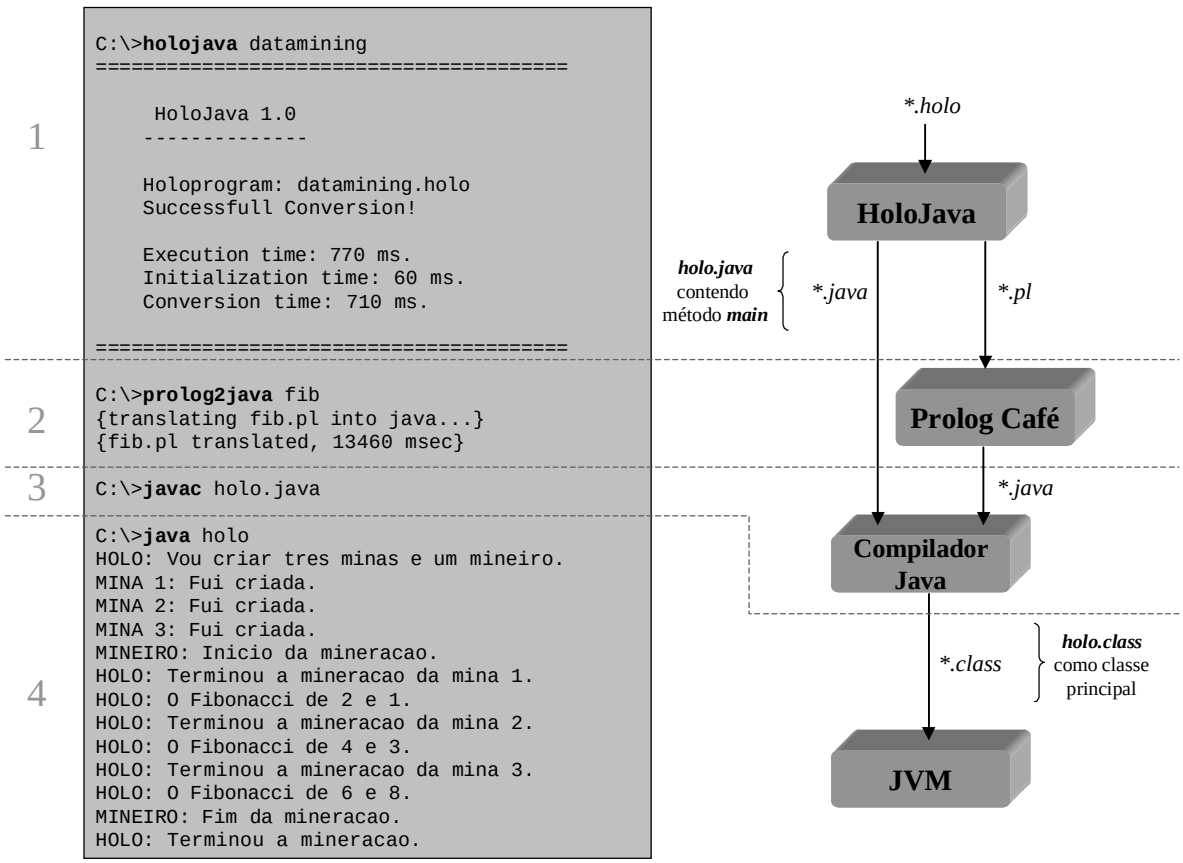


FIGURA 5.7 – Etapas na conversão e execução de holoprogramas

As quatro etapas descritas na figura 5.7 gerenciam diversos arquivos. A figura 5.8 apresenta o gerenciamento para o programa *datamining.holo*. Além disso, a figura mostra as bibliotecas de classe usadas em cada etapa, as quais devem estar disponíveis no ambiente (*classpath*). O anexo 4 contém a listagem dos arquivos gerados durante a conversão de *datamining.holo*. Por sua vez, o anexo 6 possui a listagem das classes de definição da Hololinguagem (*holoj.lang.<*.class*>*).

As seguintes observações sobre cada etapa merecem atenção:

- **Etapa 1:** cada ente estático gera um arquivo *.java* contendo uma classe. Além disso, cada MLA gera um arquivo em Prolog (no exemplo, *fib.pl*). Nesta etapa, o ambiente deve conter o caminho para acesso aos arquivos *.class* gerados pela compilação da HoloJava (veja figura 5.2 e tabela 5.5);
- **Etapa 2:** a conversão realizada pelo Prolog Café gera um arquivo *.java* (no exemplo, *PRED_fib_2.java*) para cada predicado existente no arquivo Prolog. Além disso, o Prolog Café exige o acesso, através do ambiente, ao seu arquivo de classes (*PrologCafe044.jar*);
- **Etapa 3:** os arquivos em Java gerados pelas primeiras etapas devem ser compilados em conjunto. Na compilação, o ambiente deve conter os caminhos para acesso as classes do Jada [CIP 2001], Holo [HOL 2001] e Prolog Café [PRO 2001a];
- **Etapa 4:** a execução do programa necessita dos arquivos *.class* gerados pela terceira etapa. Além disso, o ambiente deve conter o acesso aos mesmos arquivos de bibliotecas de classes da etapa anterior.

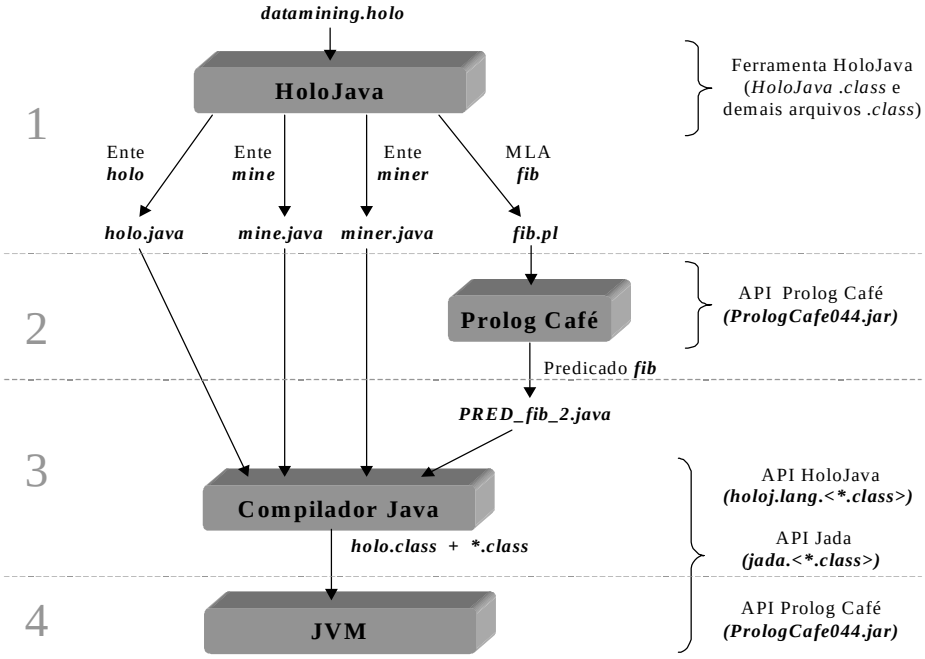


FIGURA 5.8 – Gerenciamento de arquivos na conversão e execução

A figura 5.9 mostra um segundo programa para simulação de *datamining* (*performance.holo*). O programa é semelhante ao apresentado na figura 5.5. No entanto, permite a avaliação de desempenho através de medições em tempo de execução. A mesma simulação é utilizada na seção 5.3 para avaliação do *Distributed Holo*. O programa contém comentários e as seguintes observações servem de complemento:

- a estrutura e comportamento do programa é semelhante ao exemplo mostrado na figura 5.5. Sendo assim, os comentários apresentados naquele contexto são válidos, especialmente, a figura 5.6 mostra os mesmos passos de mineração;
- o programa na figura 5.9 não possui MLAs. A mineração consiste simplesmente em um conjunto de acessos à história das minas. O número de acessos é determinado pelo usuário através da linha de comando.

```
//********************  ENTE PRINCIPAL ********************

holo()
{
  holo(Oper)
  {
    writeln('HOLO: Vou criar tres minas e um mineiro'),
    clone(mine(1),mine_d1),     // Cria as tres minas.
    clone(mine(2),mine_d2),
    clone(mine(3),mine_d3),
    clone(miner(Oper),miner_d), // Cria o mineiro.
    time(Inicio),
    for X := 1 to 3 do
    {
      history#list(#Data),      // Aguarda a escrita dos resultados da mineracao.
      writeln('HOLO: Terminou a mineracao na mina:',Data)
    }
    time(Fim),
    writeln('HOLO: Terminou a mineracao. Tempo = ',(Fim-Inicio),' milisegundos.')
  }
}


//********************  ENTE MINA ******************

mine()
{
  mine(Ident)
  {
    writeln('MINA ',Ident,': Fui criada')
  }

  history
  {
    list(1,2).  // Dado que sera minerado.
  }
}

//********************  ENTE MINEIRO ******************

miner()
{
  miner(Oper)
  {
    writeln('MINEIRO: Inicio da mineracao.'),
    move(self,mine_d1),         // Passo 1  - Entra na mina 1
    mining(1,Oper),             // Passo 2  - Minera na mina 1
    move(self,out),             // Passo 3  - Sai da mina 1
    out(history)!list(1),       // Passo 4  - Escreve o resultado 1
    move(self,mine_d2),         // Passo 5  - Entra na mina 2
    mining(2,Oper),             // Passo 6  - Minera na mina 2
    move(self,out),             // Passo 7  - Sai da mina 2
    out(history)!list(2),       // Passo 8  - Escreve o resultado 2
    move(self,mine_d3),         // Passo 9  - Entra na mina 3
    mining(3,Oper),             // Passo 10 - Minera a mina 3
    move(self,out),             // Passo 11 - Sai da mina 3
    out(history)!list(3),       // Passo 12 - Escreve o resultado 3
    writeln('MINEIRO: Termino da mineracao.')
  }

  mining(Ident,Oper)
  {
    writeln('MINEIRO: Estou minerando ',Oper,' vezes na mina ',Ident),
    for Cont := 1 to Oper do
    {
      out(history).list(1,2)      // Realiza a mineracao.
    }
  }
}
```

FIGURA 5.9 – Programa de mineração *performance.holo*

A figura 5.10 mostra a conversão do programa e sua execução para 5.000 operações. Destaca-se na figura, a linha de comando. O número de minerações consta como um parâmetro. A tabela 5.1 apresenta as medições para um conjunto de execuções. A segunda coluna contém a média de diversas execuções em milisegundos e a terceira o desvio padrão. A figura 5.11 mostra os resultados em um gráfico. Por sua vez, a tabela 5.2 contém as versões de software e a configuração de hardware usados no experimento. Na seção 5.3 os resultados são analisados no contexto dos experimentos do ambiente de execução distribuída (veja figura 5.28 e tabela 5.13).

```
C:\>holojava performance
========================================

     HoloJava 1.0
    --------------

    Holoprogram: performance.holo
    Successfull Conversion!

    Execution time: 770 ms.
    Initialization time: 60 ms.
    Conversion time: 710 ms.

========================================

C:\>javac holo.java

C:\>java holo 5000
HOLO: Vou criar tres minas e um mineiro.
MINA 1: Fui criada.
MINA 2: Fui criada.
MINA 3: Fui criada.
MINEIRO: Inicio da mineracao.
MINEIRO: Estou minerando 5000 vezes na mina 1.
HOLO: Terminou a mineracao na mina 1.
MINEIRO: Estou minerando 5000 vezes na mina 2.
HOLO: Terminou a mineracao na mina 2.
MINEIRO: Estou minerando 5000 vezes na mina 3.
HOLO: Terminou a mineracao na mina 3.
MINEIRO: Termino da mineracao.
HOLO: Terminou a mineracao. Tempo = 2690 milisegundos.
```

Parâmetro na linha de comando

FIGURA 5.10 – Conversão e execução de *performance.holo*

TABELA 5.1 – *Benchmarks* (ms) para *performance.holo*

Número de Operações	**Medições**	
	Média	**Desvio**
1000	378,9	14,9
2000	446,3	17,9
3000	527,5	15,8
4000	608,8	13,9
5000	680,8	14,2
10000	1.055,6	17,8
15000	1.439,0	22,5
20000	1.800,9	25,0
25000	2.137,1	35,2

TABELA 5.2 – Hardware e software usados nos experimentos

Software	**Versão**
Conectiva Linux	versão 6.0
HoloJava	Versão 1.0
Jada	Versão 3.0 beta 7
Java	Versão 1.3.1
Prolog Café	Versão 0.44
Hardware – Configuração	
Intel Pentium II – 233 MHz – 64 MBytes Ram	

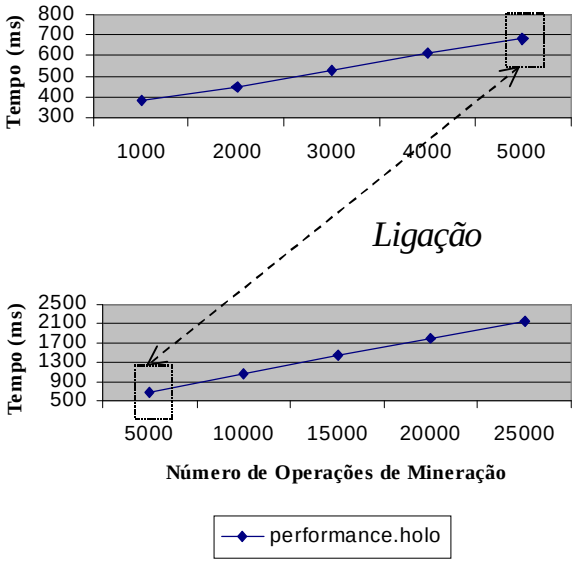


FIGURA 5.11 – Desempenho da execução de *performance.holo*

A tabela 5.3 apresenta as características de dez holoprogramas acompanhadas dos resultados de suas conversões. O anexo 3 contém a listagem dos programas. Por sua vez, a tabela 5.2 mostra as versões de software e a configuração de hardware usados no experimento. Uma descrição sucinta dos programas é apresentada a seguir:

- ***datamining.holo*** (figura 5.5 e anexo 3.1): simula uma mineração usando três minas e um mineiro. O programa utiliza uma MLA para cálculo de números da série de Fibonacci;
- ***performance.holo*** (figura 5.9 e anexo 3.2): simula uma mineração usando três minas e um mineiro. O usuário determina quantas operações de mineração serão realizadas em cada mina. O programa apresenta o tempo necessário para as minerações, permitindo assim, considerações sobre a eficiência do código gerado pela HoloJava;
- ***semaphores.holo*** (anexo 3.3): suporta o gerenciamento de semáforos. Diversos entes concorrem por um recurso compartilhado usando semáforos para controle da exclusão mútua. O número de entes concorrentes é determinado pelo usuário;
- ***philosofers.holo*** (anexo 3.4): implementa o clássico programa concorrente "Jantar de Filósofos". Os filósofos são impl ementados como entes;
- ***buffers.holo*** (anexo 3.5): suporta o gerenciamento de *buffers*. Diversos produtores e consumidores podem acessar de forma concorrente o *buffer*. Os produtores e consumidores são entes e o sincronismo é realizado pela história que implementa o *buffer*. O número de produtores e consumidores pode ser escolhido pelo usuário;
- ***travel.holo*** (anexo 3.6): simula uma viagem por dois estados brasileiros, visitando cinco municípios. Os estado e municípios são modelados como entes. Exemplifica a composição, mobilidade e o uso de ações predefinidas para obtenção de informações de entes (*whereami*, *whoami* e *howmany*);
- ***hanoi.holo*** (anexo 3.7): implementa o programa Torre de Hanói. Utiliza uma MLA para determinação do resultado;

- ***fibo.holo*** (anexo 3.8): implementa o cálculo de Fibonacci. Utiliza uma MLA para obtenção do resultado;
- ***lists.holo*** (anexo 3.9): realiza um conjunto de manipulações (geração, inversão e concatenação) de listas usando uma MLA;
- ***family.holo*** (anexo 3.10): gerenciador de relações familiares. Problema clássico para demonstração de base de dados em Prolog.

A tabela 5.3 está organizada em dez linhas e oito colunas. Cada linha está relacionada com um holoprograma. As linhas são subdivididas de acordo com as informações a serem apresentadas. Merece destaque a subdivisão que totaliza informações (linha **Total**, a partir da quarta coluna). As oito colunas mostradas na tabelas são as seguintes:

- **Holoprograma (*.holo*)**: nome do programa analisado;
- **Tamanho em linhas (*.holo*)**: número de linhas do holoprograma. Os comentários foram retirados para determinação deste valor. Os programas mostrados no anexo 3 permanecem com comentários;
- **Tamanho em bytes (*.holo*)**: número de bytes do programa sem comentários;
- **Arquivos gerados (*.java* e *.pl*)**: nome dos arquivos em Java e Prolog gerados pela conversão. Nesta coluna podem ser encontrados dois tipos de campos. O primeiro tipo representa um arquivo Java gerado diretamente pela HoloJava (por exemplo, o arquivo *holo.java* na conversão de *datamining.holo*). O segundo tipo representa um *.java* resultante da conversão do arquivo Prolog gerado pela HoloJava (por exemplo, o arquivo *PRED_fib_2.java* na conversão de *datamining.holo*). O Prolog Café gera um arquivo Java para cada predicado (veja figura 5.8). No segundo tipo, a primeira linha do campo contém o nome do *.java*. A segunda mostra o programa Prolog origem e o número de arquivos *.class* gerados pela compilação do arquivo Java;
- **Tempo de conversão (milisegundos)**: esta coluna indica o tempo envolvido na conversão de um arquivo *.holo* para os arquivos *.java* e *.pl* (por exemplo, 163,60 ms para geração dos arquivos *holo.java*, *mine.java*, *miner.java* e *fib.pl*, no caso de *datamining.holo*). Além disso, esta coluna mostra o tempo necessário (Prolog Café) para conversão de um programa Prolog para Java (por exemplo, 15.178,80 ms para geração de *PRED_fib_2.java*, no caso de *datamining.holo*). Conforme mostram os exemplos, a HoloJava e o Prolog Café podem gerar diversos arquivos Java. Nos testes foram usados programas sem comentários e o resultado é a média de diversas execuções;
- **Tamanho em linhas (*.java*)**: número de linhas do fonte em Java gerado pela HoloJava ou pelo Prolog Café;
- **Tamanho em bytes (*.java*)**: número de bytes dos arquivos *.java* gerados pela HoloJava ou pelo Prolog Café;
- **Tamanho em bytes (*.class*)**: tamanho em bytes dos arquivos *bytecode* gerados pela compilação dos programas. A compilação de um *.java* gerado pela HoloJava produz somente um *.class*. No entanto, a compilação de um *.java* gerado pelo Prolog Café pode gerar diversos *.class*. Neste caso, esta coluna apresenta a soma dos tamanhos destes arquivos. A quarta coluna indica quantos *.class* resultaram da compilação de um *.java* gerado pelo Prolog Café.

TABELA 5.3 – Informações sobre conversão de holoprogramas

Holoprograma (*.holo*)	**Tam. em linhas (*.holo*)**	**Tam. em bytes (*.holo*)**	**Arquivos gerados (*.java* e *.pl*)**	**Tempo de conver. (*ms*)**	**Tam. em linhas (*.java*)**	**Tam. em bytes (*.java*)**	**Tam. em bytes (*.class*)**
datamining.holo	73	1.391	holo.java	163,60	50	1.461	2.222
			mine.java		31	595	1.279
			miner.java		166	4.813	3.761
			PRED_fib_2.java (fib.pl) (7 x .class)	15.178,80	134	3.961	5.779
			Total	**15.342,40**	**381**	**10.830**	**13.041**
performance.holo	62	1.206	holo.java	146,80	51	1.424	2.186
			mine.java		29	507	1.210
			miner.java		99	2.936	2.766
			Total	**146,80**	**179**	**4.867**	**6.162**
semaphores.holo	50	879	holo.java	126,40	58	1.291	1.967
			competidor.java		45	874	1.508
			Total	**126,40**	**103**	**2.165**	**3.475**
philosofers.holo	51	1.084	holo.java	138,00	54	1.497	2.012
			filosofo.java		56	1.636	1.988
			Total	**138,00**	**110**	**3.133**	**4.000**
buffers.holo	63	1.248	holo.java	151,00	105	2.397	2.373
			produtor.java		49	1.096	1.563
			consumidor.java		49	1.101	1.568
			Total	**151,00**	**203**	**4.594**	**5.504**
travel.holo	112	2.848	holo.java	193,00	44	1.099	1.869
			Estado.java		57	1.541	2.094
			municipio.java		32	719	1.452
			viajante_e.java		102	4.054	3.475
			Total	**193,00**	**235**	**7.413**	**8.890**
hanoi.holo	28	605	holo.java	126,60	121	3.684	2.945
			PRED_hanoi_5.java (hanoi.pl) (5 x .class)	17.238,80	155	5.194	6.009
			PRED_append_3.java (hanoi.pl) (5 x .class)		131	3.990	4.814
			Total	**17.365,40**	**407**	**12.868**	**13.768**
fibo.holo	25	454	holo.java	115,80	85	2.131	2.435
			PRED_fib_2.java (fib.pl) (7 x .class)	15.130,20	134	3.961	5.779
			Total	**15.246,00**	**219**	**6.092**	**8.214**
lists.holo	39	858	Holo.java	131,40	118	3.455	2.815
			PRED_listas_4.java (listas.pl) (1 x .class)	17.126,80	52	1.379	1.389
			PRED_nrev_2.java (listas.pl) (5 x .class)		119	3.483	4.765
			PRED_append_3.java (listas.pl) (5 x .class)		131	3.991	4.814
			PRED_gera_lista_2.java (listas.pl) (5 x .class)		112	3.438	4.790
			PRED_size_2.java (listas.pl) (5 x .class)		117	3.423	4.759
			Total	**17.258,20**	**649**	**19.169**	**23.332**
family.holo	30	577	holo.java	117,20	88	2.231	2.601
			PRED_familia_2.java (familia.pl) (8 x .class)	15.854,40	134	3.721	5.947
			PRED_pai_2.java (familia.pl) (11 x .class)		178	5.433	8.838
			PRED_mae_2.java) (familia.pl) (11 x .class)		178	5.427	8.832
			Total	**15.971,60**	**578**	**16.812**	**26.218**

A análise dos resultados mostrados na tabela 5.3 permite as seguintes conclusões:

- **desempenho da conversão HoloJava x Prolog Café:** existe uma diferença considerável no tempo de conversão da HoloJava em relação ao tempo de conversão do Prolog Café. Por exemplo, no programa *datamining.holo*, 98,93 % do tempo total equivale à conversão de *fib.pl*. O holoprograma possui 73 linhas, das quais apenas 12 compõem a MLA. Os programas *datamining.holo* e *performance.holo* são semelhantes. A principal diferença consiste na MLA existente no primeiro. No entanto, a conversão de *datamining.holo* é aproximadamente 105 vezes mais lenta do que a conversão de *performance.holo*. O mesmo comportamento pode ser constatado nos demais programas que usam MLAs. Conclui-se assim que o uso deste tipo de ações é um ponto crítico no desempenho da conversão;
- **tamanho dos arquivos gerados pela HoloJava x Prolog Café:** o tamanho dos arquivos gerados pelo Prolog Café é consideravelmente maior do que os gerados pela HoloJava. Por exemplo, considerando o programa *datamining.holo*, o Prolog Café gerou 35,17 % do número de linhas em Java, 36,57 % do número de bytes nos arquivos Java e 44,31 % do número de bytes nos arquivos *.class*. Por outro lado, a MLA compõe apenas 16,44 % do tamanho do holoprograma em linhas. Conclui-se assim que as MLAs também são um ponto crítico no tamanho dos arquivos gerados;
- **índices de conversão:** tendo como base a tabela 5.3 surge um conjunto de índices para avaliação da conversão de holoprogramas para Java. A tabela 5.4 mostra os índices criados e sua aplicação em cinco holoprogramas. A tabela é composta de oito colunas. A primeira possui o nome do índice e a segunda sua descrição. As seis colunas restantes contém a aplicação dos índices em cinco holoprogramas e uma média geral. Os cinco holoprogramas escolhidos não possuem MLAs. Busca-se assim uma maior estabilidade para os índices de conversão.

TABELA 5.4 – Índices de conversão da HoloJava

Índice	**Descrição**	**Holoprogramas**					**Média**
		performance	*semaphores*	*philosofers*	*buffers*	*travel*	
JHL (*Java/Holo Lines*)	Número de linhas em Java geradas para cada linha no holoprograma	2,89	2,06	2,16	3,22	2,10	2,49
JHB (*Java/Holo Bytes*)	Número de bytes nos arquivos Java para cada byte no holoprograma	4,04	2,46	2,89	3,68	2,60	3,14
CHB (*Class/Holo Bytes*)	Número de bytes nos arquivos *.class* para cada byte no holoprograma	5,11	3,95	3,70	4,41	3,12	4,10
HJLT (*Holo/Java Lines Time*)	Tempo de conversão (ms) de uma linha em Holo para Java	2,37	2,53	2,70	2,40	1,72	2,34
HJBT (*Holo/Java Bytes Time*)	Tempo de conversão (ms) de um byte em Holo para Java	0,12	0,14	0,13	0,12	0,07	0,12

A tabela 5.5 contém dados técnicos relacionados com a criação da HoloJava (veja figura 5.2). A tabela é composta pelas seguintes colunas:

- nome do arquivo (*HoloJava.jj*, veja o anexo 5) que contém a gramática da Hololinguagem e as ações semânticas para a criação da HoloJava. Este arquivo serve de entrada para a ferramenta JavaCC [JAV 2001];
- tamanho em linhas do arquivo *HoloJava.jj*;
- tamanho em bytes do arquivo *HoloJava.jj*;
- tempo necessário para geração dos arquivos em Java que compõem a HoloJava. O resultado é a média de diversas execuções. Estes arquivos surgem após a aplicação do JavaCC [JAV 2001] no arquivo *HoloJava.jj*;
- nome dos arquivos gerados pelo JavaCC;
- tamanho em linhas dos arquivos *.java*;
- tamanho em bytes dos arquivos em Java;
- tamanho em bytes dos arquivos *.class* gerados pela compilação dos fontes em Java. A compilação do arquivo *HoloJava.java* gera dois arquivos *.class* (notação "*n x .class*" mostrada na tabela).

Os três arquivos contendo as definições de classes usadas pela Hololinguagem também foram inseridos na tabela (veja o anexo 6). Existe uma totalização para os arquivos produzidos pelo JavaCC e uma totalização geral. Os principais dados mostrados na tabela 5.5 para a HoloJava 1.0 são:

- número de linhas da especificação para o JavaCC: 1.832 linhas;
- número de arquivos Java que compõem a HoloJava: 10 arquivos;
- número total de linhas em Java que compõem a HoloJava: 5.701 linhas

TABELA 5.5 – Informações técnicas sobre a HoloJava 1.0

Arquivo fonte do JavaCC (*.jj*)	**Tam. em linhas (*.jj*)**	**Tam. em bytes (*.jj*)**	**Tempo de geração (ms)**	**Arquivos fonte (*.java*)**	**Tam. em linhas (*.java*)**	**Tam. em bytes (*.java*)**	**Tam. em bytes (*.class*)**
HoloJava.jj	1.832	51.324	747,40	HoloJava.java (2 x .class)	3.592	104.214	48.469
				HoloJavaTokenManager.java	1.093	34.791	15.852
				ASCII_CharStream.java	379	10.227	5.179
				ParseException.java	189	6.514	2.791
				HoloJavaConstants.java	146	2.652	4.102
				TokenMgrError.java	131	4.319	2.066
				Token.java	79	2.725	546
				Total gerado pelo JavaCC	**5.609**	**165.442**	**79.005**
				BVector.java	61	1.230	849
				holoString.java	20	314	440
				Being.java	11	202	345
				Total geral da HoloJava	**5.701**	**167.188**	**80.639**

5.2 HoloEnv (*Holo Environment*)

O desenvolvimento de programas em Holo pode ser realizado em um ambiente integrado denominado **HoloEnv**. Este ambiente possui um conjunto de facilidades para gerenciamento e edição de arquivos. Além disso, o HoloEnv gerencia a conversão e a execução de programas através de menus e janelas.

O ambiente é composto pelos seguintes módulos (figura 5.12):

- **Módulo de gerenciamento de arquivos**: suporte para manipulação de arquivos através de janelas e menus;
- **Módulo de edição**: editor de programas com sintaxe *highlight* específica para holoprogramas;
- **Módulo de conversão**: responsável pela conversão de holoprogramas para Java. Este módulo é implementado pela HoloJava (seção 5.1);
- **Módulo de compilação e execução**: este módulo utiliza o ambiente Java para compilação e execução de programas convertidos pela HoloJava.

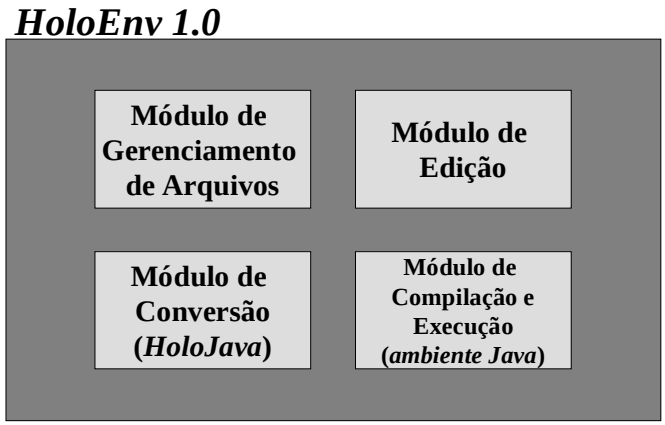


FIGURA 5.12 – Composição do HoloEnv

A figura 5.13 mostra o contexto do HoloEnv. O principal objetivo do ambiente consiste na integração dos módulos que participam da criação de holoprogramas. O uso de janelas, menus e botões simplifica o controle dos módulos e facilita a seqüência de ações necessárias para conversão e execução de programas (seção 5.1).

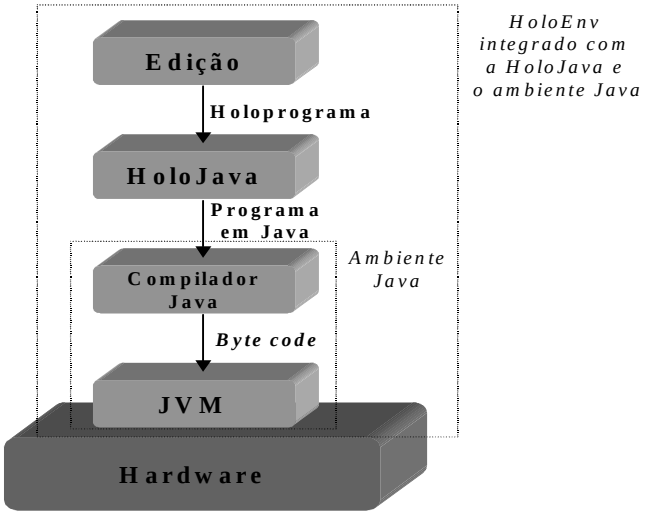


FIGURA 5.13 – Contexto do HoloEnv

A tela principal do HoloEnv é composta por três regiões (figura 5.14):

- **Região de controle:** composta de menus e botões que permitem o gerenciamento e a edição de programas. Além disso, essa região suporta o controle sobre a conversão de programas (HoloJava), compilação (compilador Java) e execução (JVM);
- **Região de edição:** nesta região encontram-se as janelas de edição de programas. Diversos programas podem ser editados simultaneamente;
- **Região de resultados:** essa região contém duas janelas. A janela *Build* mostra mensagens relacionadas com os resultados da conversão e compilação. Por sua vez, a janela *Output* apresenta os resultados da execução.

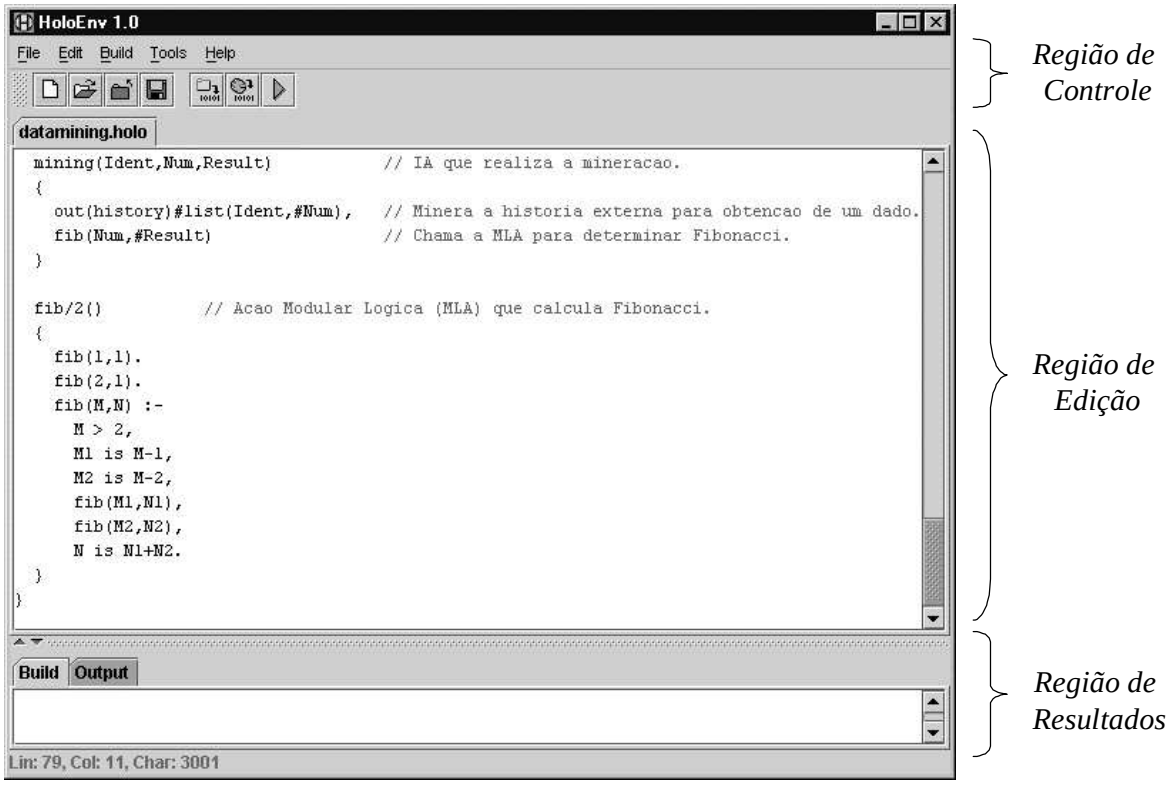


FIGURA 5.14 – Composição da tela principal do HoloEnv

O menu *Help* permite o acesso ao *About* (figura 5.15) do HoloEnv, onde constam informações básicas sobre o ambiente (versão em uso, endereço eletrônico para contato e Holomarca). O menu *File* suporta o gerenciamento de arquivos. A figura 5.16 exemplifica uma de suas principais funções, ou seja, a escolha de arquivos a serem editados.

FIGURA 5.15 – Janela *About*

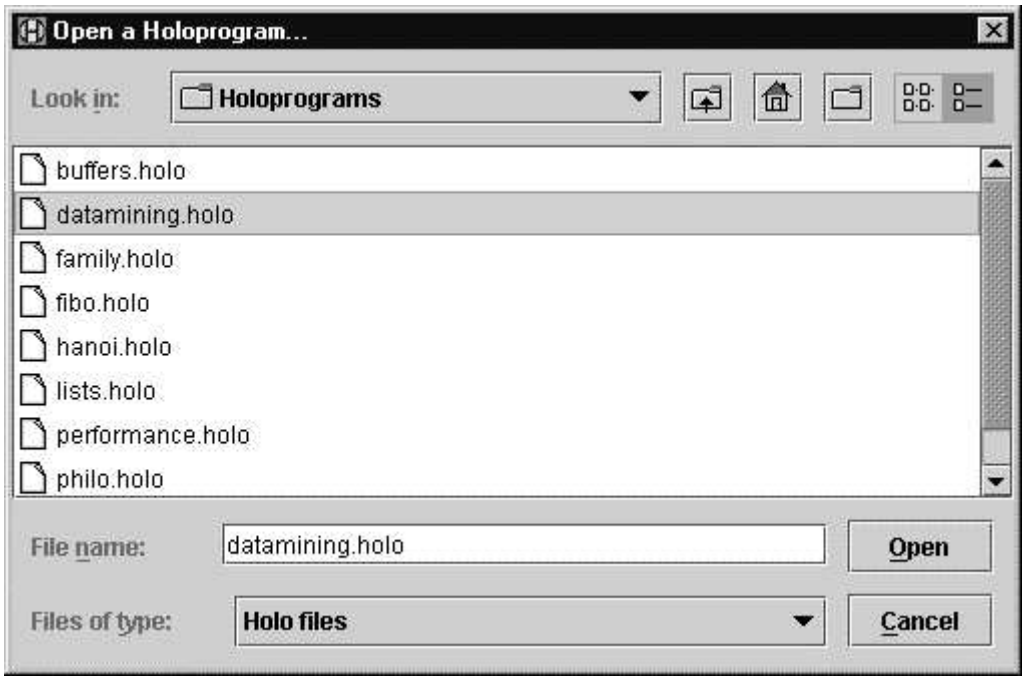


FIGURA 5.16 – Janela para seleção de holoprogramas

No menu *Build* encontram-se os controles para conversão e execução de holoprogramas. A figura 5.17 mostra o menu e destaca as três opções disponíveis:

- ***Compile:*** executa a HoloJava, convertendo o holoprograma para Java;
- ***Build:*** executa a HoloJava (opção *compile*) e compila os programas em Java. Sendo assim, são criados os arquivos *.class*;
- ***Run:*** executa a HoloJava (opção *compile*), compila os programas em Java (opção *Build*) e executa o programa.

Estas opções atuam no holoprograma existente na janela de edição ativa (região de edição). A seção 5.1 discute as quatro etapas envolvidas na conversão e execução de programas (veja as figuras 5.7 e 5.8). As opções do menu *Build* cumprem aquelas etapas. A opção *Compile* executa as duas primeiras. A opção *Build* realiza as três primeiras. Por sua vez, a opção *Run* cumpre as quatro etapas. A figura 5.17 mostra ainda o acesso à janela de parâmetros. Esta janela possibilita a introdução de parâmetros para programas que esperam recebê-los através da linha de comando. A figura mostra a passagem de um parâmetro para o programa *performance.holo* (mesma situação representada na figura 5.10). A região de resultados possui duas janelas:

- ***Build:*** mostra os resultados da conversão e compilação de programas. Erros de conversão são relatados nesta janela. A figura 5.18 mostra um erro de sintaxe detectado pela HoloJava no programa *datamining.holo*. Um duplo clique na linha que relata o problema, coloca o cursor na posição do erro no programa (janela de edição). A figura 5.19 apresenta uma conversão sem erros do programa *datamining.holo*. As mensagens relatam os resultados da conversão realizada pela HoloJava e pelo Prolog Café (veja figura 5.7);
- ***Output:*** mostra o resultado da execução de um programa. A figura 5.20 apresenta a execução do programa *datamining.holo*. A janela mostra as mensagens escritas pelos comandos *writeln* e *write* do holoprograma.

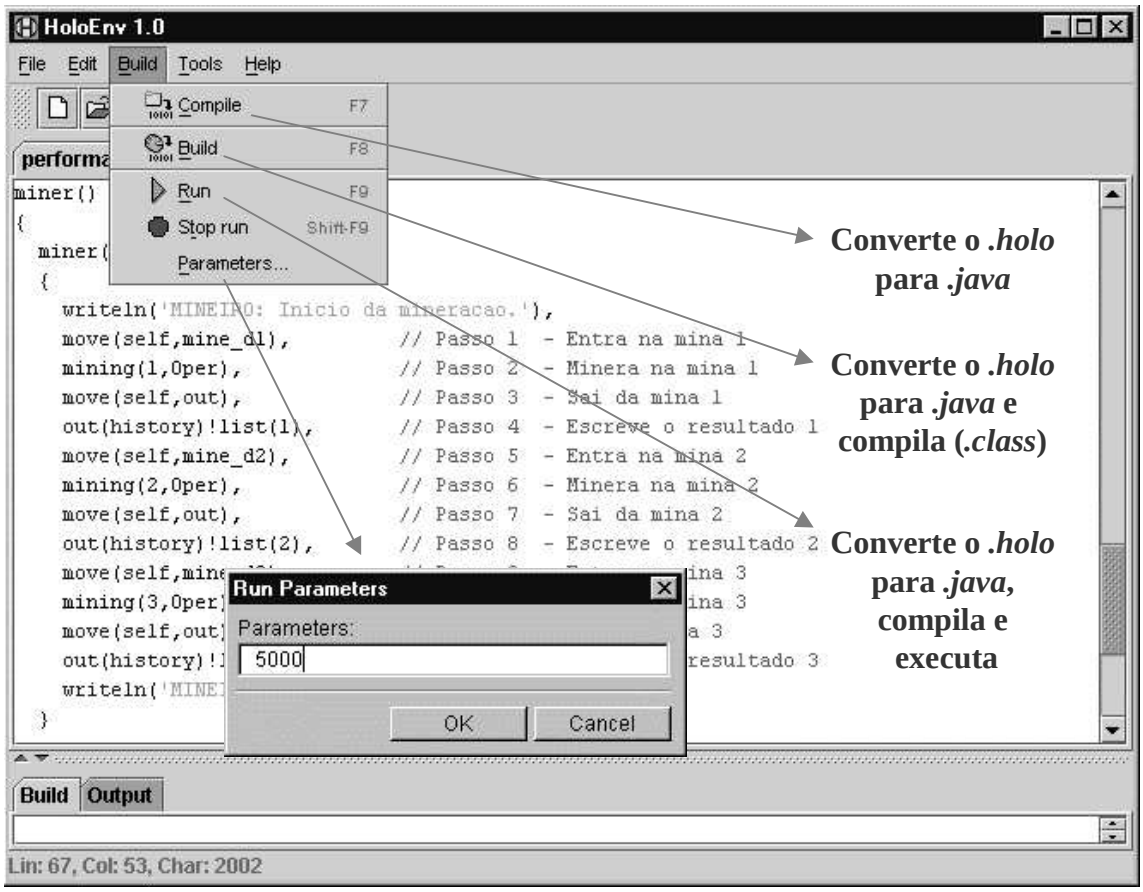


FIGURA 5.17 – Menu *Build*

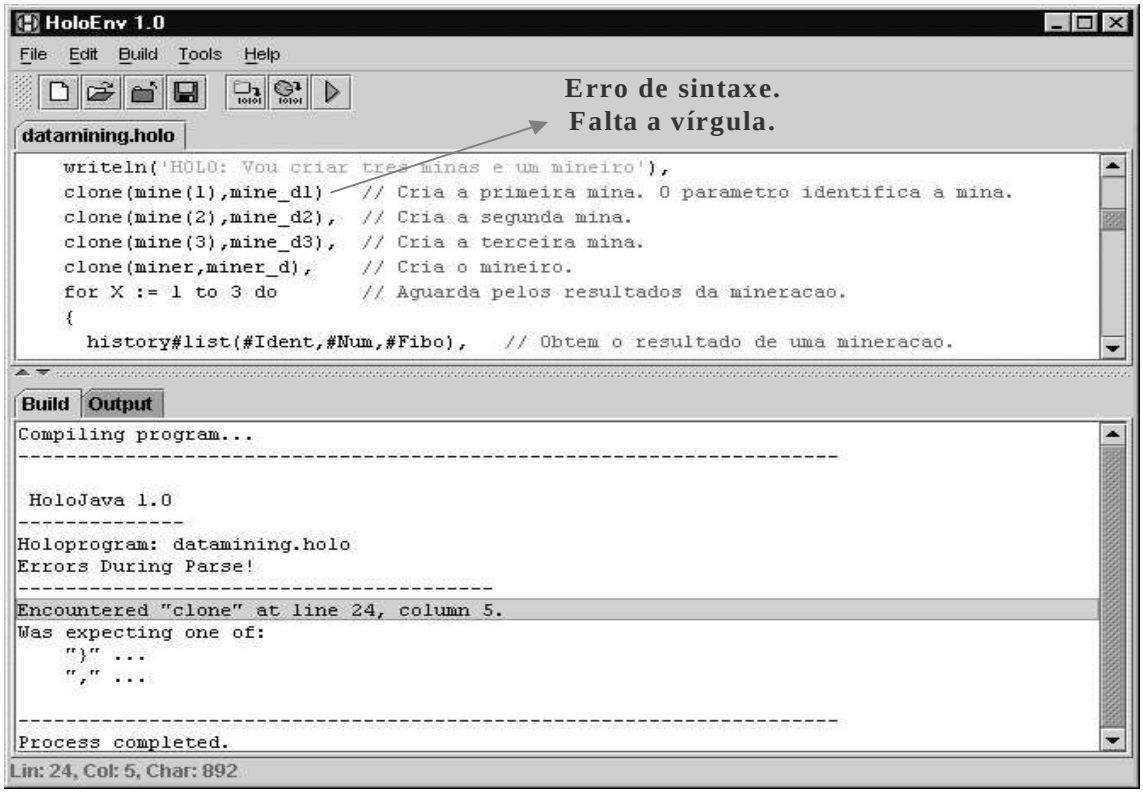


FIGURA 5.18 – Janela *Build* após detecção de erro de sintaxe

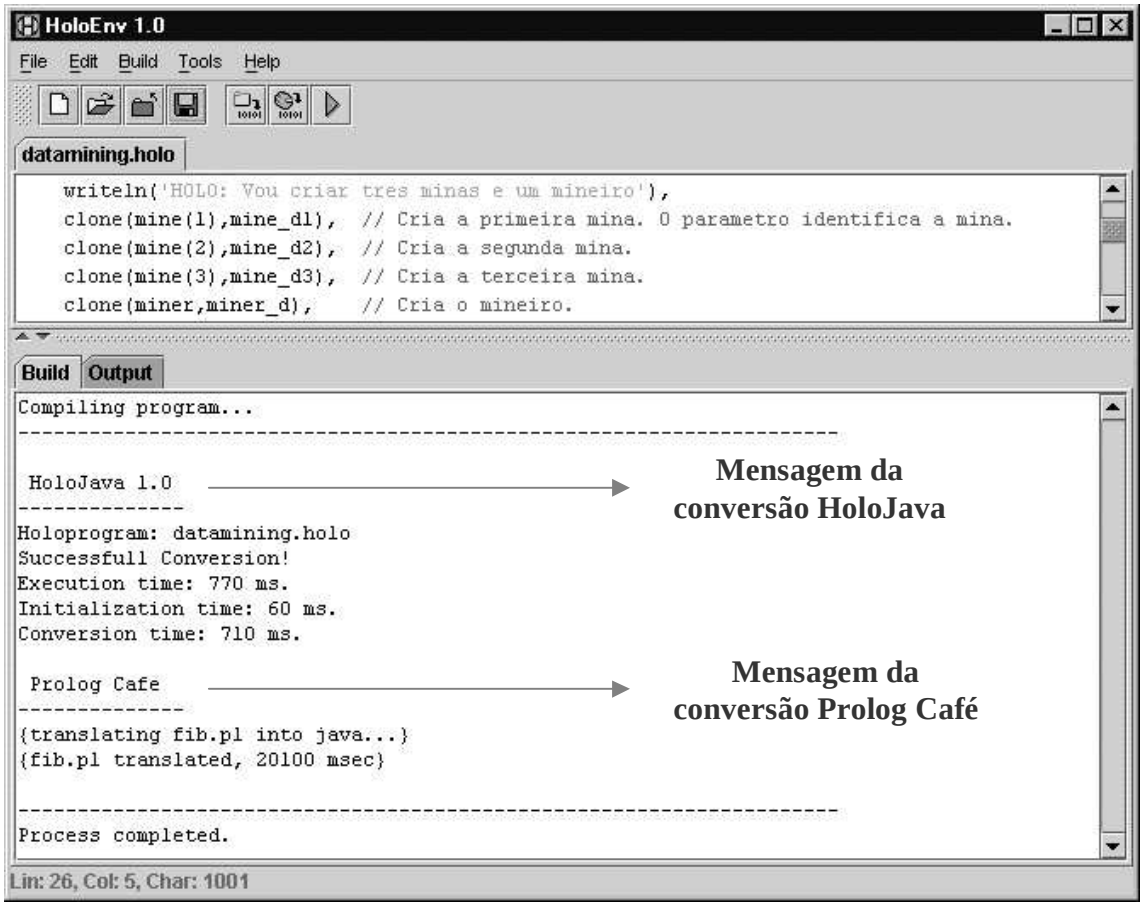


FIGURA 5.19 – Janela *Build* após compilação sem erros

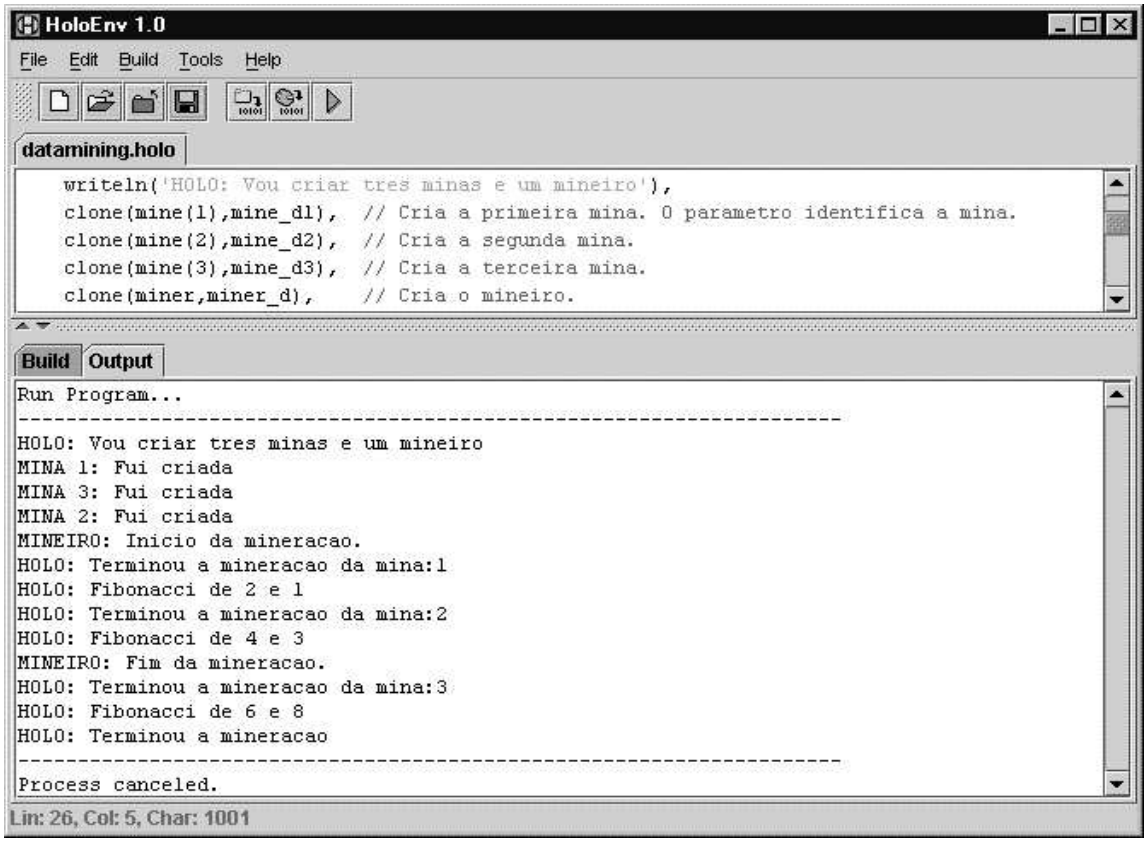


FIGURA 5.20 – Janela *Output* após execução de *datamining.holo*

A tabela 5.6 apresenta informações técnicas sobre o HoloEnv 1.0. A primeira coluna contém o nome dos arquivos Java que compõem o ambiente. A segunda coluna mostra o tamanho dos arquivos em linhas. A terceira apresenta o tamanho dos fontes em bytes e a quarta contém o tamanho dos arquivos *.class* gerados pela compilação de cada arquivo Java. A compilação de alguns *.java* gera mais de um *.class*. Neste caso, a coluna contém a soma dos seus tamanhos e entre parênteses o número de arquivos gerados. A última linha da tabela apresenta o total de cada uma das colunas. Entre as informações mostradas na tabela merecem destaque.

- número de arquivos Java que compõem o HoloEnv 1.0: 27 arquivos
- número total de linhas em Java que compõem o HoloEnv 1.0: 5.749 linhas

TABELA 5.6 – Informações técnicas sobre o HoloEnv 1.0

Arquivos Fontes	**Tam. em linhas (*.java*)**	**Tam. em bytes (*.java*)**	**Tam. em bytes (*.class*)**
HoloEnv.java	920	29.760	17.002 (3)
SyntaxTextAreaHolo.java	795	19.545	17.499 (8)
HoloMenuBar.java	662	22.902	10.724 (2)
TokenMarker.java	314	8.782	2.630 (2)
CtokenMarker.java	300	8.389	3.145
ThreadControl.java	270	10.428	4.778
SyntaxUtilities.java	270	7.853	2.614
SyntaxView.java	216	6.531	2.754
DefaultSyntaxDocumentHolo.java	205	5.593	3.381 (2)
Token.java	182	4.731	1.097
KeywordMap.java	176	4.886	2.207 (2)
HoloToolBar.java	173	5.794	3.279
HoloParameters.java	163	4.309	5.315 (3)
HoloAbout	151	4.057	5.109 (3)
HoloTokenMarker.java	143	5.007	1.098
HoloOutputList.java	114	3.310	2.137
SyntaxDocument.java	112	3.792	381
Language.java	104	5.790	4.684
SyntaxEditorKit.java	99	3.250	684
ThreadExec.java	87	2.522	2.312
TabInfo.java	70	1.286	1.005
ThreadExecError.java	54	1.256	1.499
ThreadExecInput.java	53	1.296	1.557
HoloSettings.java	51	1.372	1.130
HoloFilter.java	36	853	807
FileExtension.java	19	336	486
JMenuItemAdapter.java	10	174	334
TOTAL (27 arquivos)	5.749	173.804	99.648

5.3 DHolo (*Distributed Holo*)

O Holoparadigma possui uma semântica simples e distribuída. O estímulo à modelagem subliminar da distribuição é um dos seus principais objetivos. Este objetivo somente poderá ser alcançado com a criação de um ambiente de execução distribuída que suporte suas principais abstrações. Neste sentido, surge o ***Distributed Holo*** (***DHolo***) [BAR 2001, BAR 2001b]. O DHolo é um ambiente para execução distribuída de holoprogramas. Esta seção apresenta os princípios do ambiente e os resultados parciais que servirão de base para sua criação. A desenvolvimento do DHolo depende da implementação de dois suportes à distribuição discutidos na seção 3.5:

- **Mobilidade física:** gerenciamento da mobilidade de entes entre nodos de uma arquitetura distribuída (veja figura 3.7b). Entre as opções para sua implementação destacam-se as tecnologias para distribuição de objetos, tais como o Voyager [VOY 2001] e o Horb [HOR 2001];

- **História distribuída:** gerenciamento do compartilhamento da história entre nodos de uma arquitetura distribuída (veja as figuras 3.6 e 3.7b). Uma solução para implementação deste aspecto é o uso de suporte para espaços de objetos distribuídos, tais como o Jada [CIP 2001] e o JavaSpaces [FRE 99].

A figura 5.21 mostra o contexto do DHolo. A plataforma de desenvolvimento é a mesma discutida na seção 5.1 (figuras 5.7 e 5.8). A mobilidade física e a história distribuída necessitam de bibliotecas especiais. Estas bibliotecas devem estar disponíveis para a HoloJava. Os experimentos apresentados nessa seção permitem a análise de diversas opções para implementação de ambos os suportes. A plataforma de execução consiste da arquitetura distribuída acrescida do ambiente DHolo. O ambiente é composto basicamente por quatro módulos:

- **módulo de suporte a espaços de objetos distribuídos:** *daemons* que suportam a abstração de um espaço compartilhado de objetos entre nodos da arquitetura (Jada [CIP 2001] e JavaSpaces [FRE 99] usam *daemons*);

- **suporte à mobilidade física:** *daemons* que permitem a mobilidade de objetos entre nodos da arquitetura (Voyager [VOY 2001] e [HORB 2001] precisam de *daemons* durante a execução de programas);

- **suporte à *Distributed HoloTree*** (DHoloTree, figura 5.22): a principal estrutura para controle da execução distribuída de um holoprograma consiste em uma árvore onde os ramos encontram-se distribuídos nos nodos da arquitetura. Esta árvore é denominada *Distributed HoloTree* (DHoloTree) e consiste na HoloTree (veja figura 5.4) com suporte à distribuição;

- **Máquina Virtual Java (JVM):** máquina virtual que permite a execução de programas Java. Conforme mostra a figura 5.21, os *daemons* de suporte à mobilidade e aos espaços de objetos utilizam a JVM. Além disso, o suporte à DHoloTree também necessita da JVM.

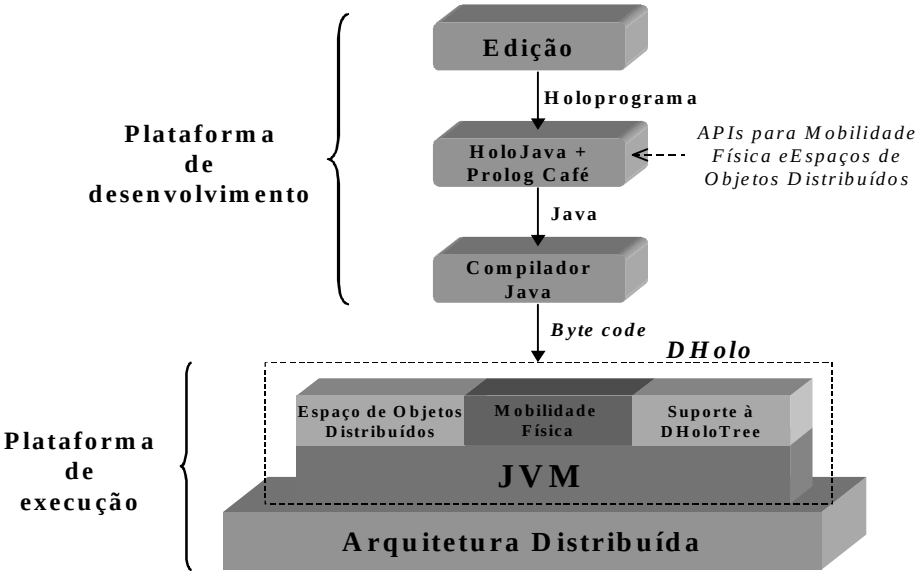


FIGURA 5.21 – Contexto do DHolo

A execução distribuída de um holoprograma consiste na distribuição de sua árvore. A figura 5.22 mostra uma possível distribuição para a árvore inicialmente apresentada na figura 5.4a. Cada nodo possui a JVM e o módulo envolvido no suporte à mobilidade física. Os demais módulos não foram representados.

A figura 5.22 mostra duas mobilidades físicas sendo realizadas:

- **Mobilidade A:** esta mobilidade mostra um ente trocando de posição na HoloTree (veja figura 5.4a). A figura 3.7a apresenta a mesma mobilidade considerando os níveis de encapsulamento de entes. A troca de posição na árvore ocorre através de uma mobilidade lógica (ação *move* na Hololinguagem). Neste caso, a mobilidade física somente ocorre se o ente destino não se encontra no mesmo nodo que o ente origem;
- **Mobilidade B:** neste caso, o ente móvel não troca de posição na HoloTree. Sua visão de contexto continua a mesma. Esta mobilidade não está relacionada com ações executadas pelo programa, ocorrendo assim, uma mobilidade física sem mobilidade lógica (figura 3.7b). Esta situação pode ser explorada pelo ambiente (gerenciamento da árvore) para suporte a uma funcionalidade específica (balanceamento de carga, tolerância a falhas, etc).

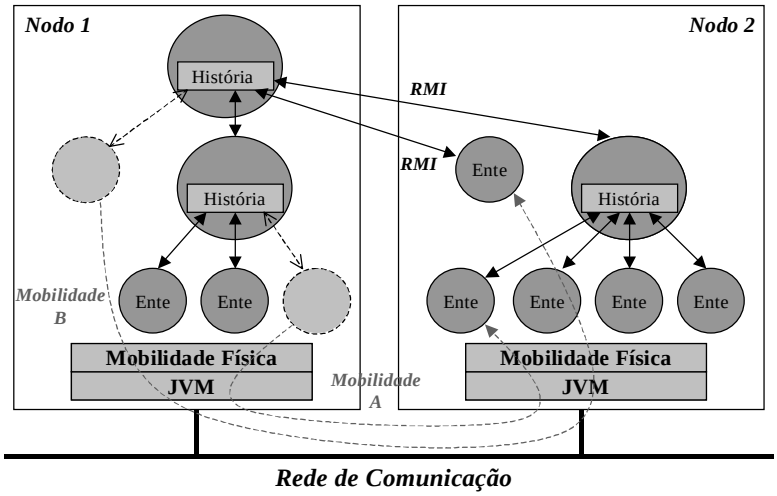


FIGURA 5.22 – HoloTree distribuída (DHoloTree)

O Holoparadigma é um modelo genérico que não estabelece restrições para a configuração da arquitetura distribuída e para as ferramentas a serem usadas na sua implementação. Seguindo este princípio, foram realizados experimentos usando quatro diferentes plataformas (hardware e software). A tabela 5.7 mostra a especificação dos nodos das arquiteturas usadas nos experimentos. Por sua vez, a tabela 5.8 apresenta as versões de software. As plataformas 1 e 2 são baseadas em uma arquitetura heterogênea e as plataformas 3 e 4 em uma homogênea. Além disso, a rede de comunicação em ambas as arquiteturas possui uma banda passante de 10 Mbps.

TABELA 5.7 – Especificação dos nodos

Nodos	**Plataformas 1 e 2**	**Plataformas 3 e 4**
1	Sun SPARCstation 20 – 128 M RAM	Intel Pentium II 233 MHz - 64 M RAM
2	Sun Ultra 10 - 128 M RAM	Intel Pentium II 233 MHz - 64 M RAM
3	Sun Ultra 5 - 192 M RAM	Intel Pentium II 233 MHz - 64 M RAM

TABELA 5.8 – Versões de software usadas nos experimentos

Software	**Plataforma 1**	**Plataforma 2**	**Plataforma 3**	**Plataforma 4**
Sistema Operacional	SunOS Release 5.7	SunOS Release 5.7	Conectiva Linux versão 6.0	Conectiva Linux Versão 6.0
Voyager	Versão 3.3	Versão 3.3	Versão 4.0.1	*Não usado*
Horb	*Não usado*	*Não usado*	*Não usado*	Versão 2.1 b2
Java	Versão 1.2	Versão 1.2	Versão 1.3.1	Versão 1.3.1
Jada	Versão 3.0 beta 7	Não usado	Versão 3.0 beta 7	Versão 3.0 beta 7
JavaSpaces	*Não usado*	Versão 1.1	*Não usado*	*Não usado*

Os experimentos buscaram dois objetivos principais:

- avaliação do modelo DHolo através de uma análise do controle da mobilidade lógica e física usando uma HoloTree distribuída. Este objetivo envolve ainda a avaliação da troca de contexto de entes usando espaços de objetos;
- avaliação de tecnologias disponíveis para implementação da história distribuída (Jada e JavaSpaces) e mobilidade física (Voyager e Horb).

Os testes envolveram uma aplicação específica, ou seja, *datamining*. Foram simulados três casos de mineração em um ambiente distribuído:

A) Mineração sequencial em um nodo (figura 5.23): um mineiro explora três minas no mesmo nodo. O mineiro troca seu nível usando mobilidade e entra na primeira mina (marca 1). Logo após, minera a história usando um número específico de operações de leitura no espaço de objetos (marca 2). O mineiro sai da mina (marca 3) e escreve o resultado na história do ente principal (marca 4). Este comportamento é repetido para as demais minas;

B) Mineração sequencial em três nodos (figura 5.24): um mineiro explora três minas em diferentes nodos. O comportamento é o mesmo do primeiro caso. Entretanto, as três minas estão localizadas em nodos diferentes. Este fato implica em dois fatos importantes. Primeiro, quando o mineiro entra nas minas 2 e 3 (marcas 1), ocorre uma mobilidade física. Segundo, quando o mineiro escreve os resultados na história do principal (marcas 2), torna-se necessário o uso de RMI. Ambos as situações geram custos que devem ser dimensionados;

C) Mineração paralela em três nodos (figura 5.25): neste caso, três mineiros exploram em paralelo três minas localizadas em diferentes nodos. A situação é similar ao caso B. A única diferença consiste na existência de três mineiros trabalhando em paralelo nas minas. Esta situação também envolve a realização de duas mobilidades físicas (marcas 1) e dois RMIs (marcas 2);

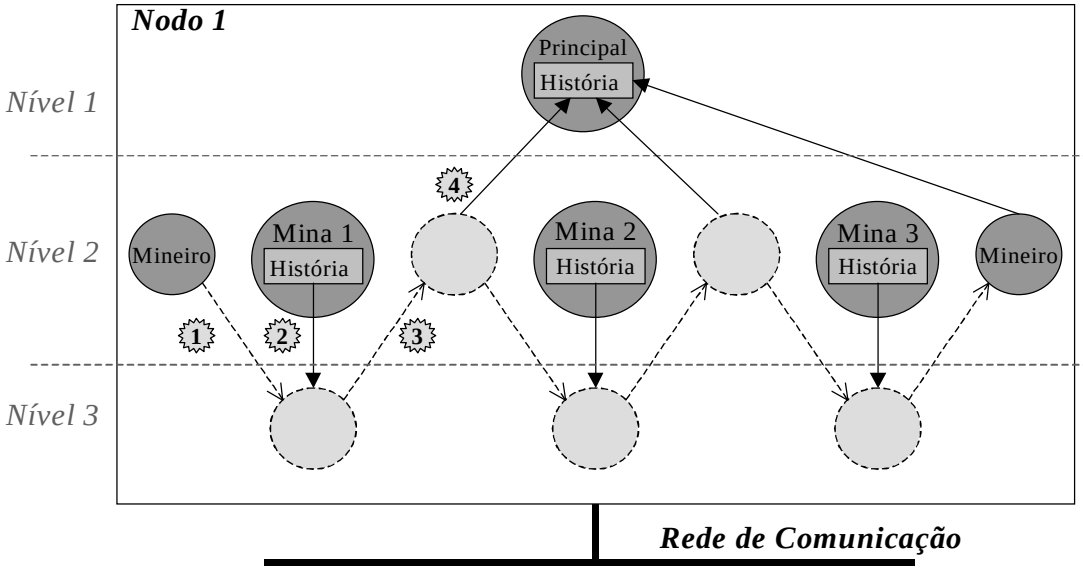


FIGURA 5.23 – *Datamining* seqüencial em um nodo (caso A)

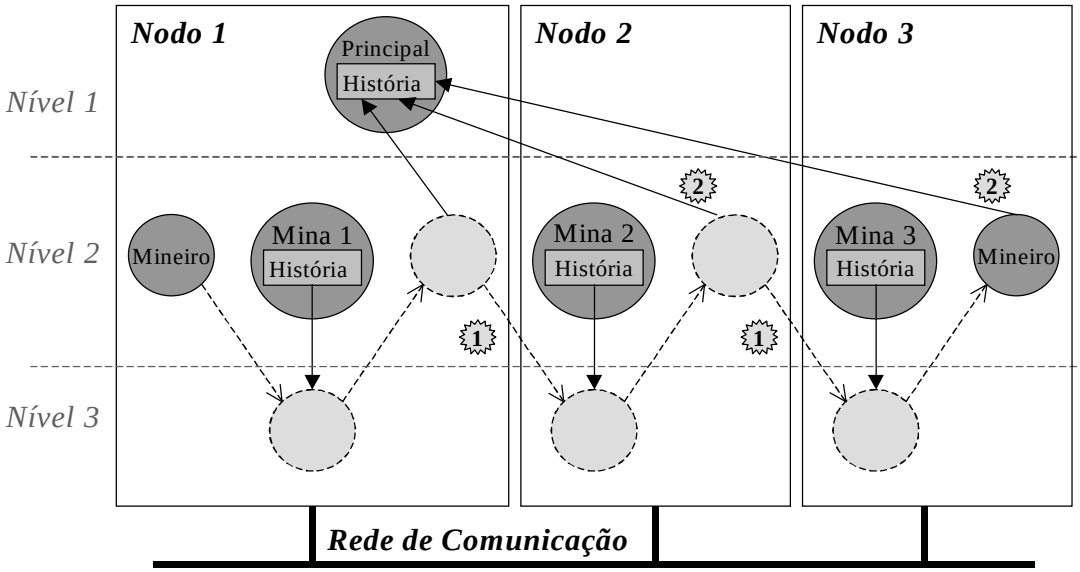


FIGURA 5.24 – *Datamining* seqüencial em três nodos (case B)

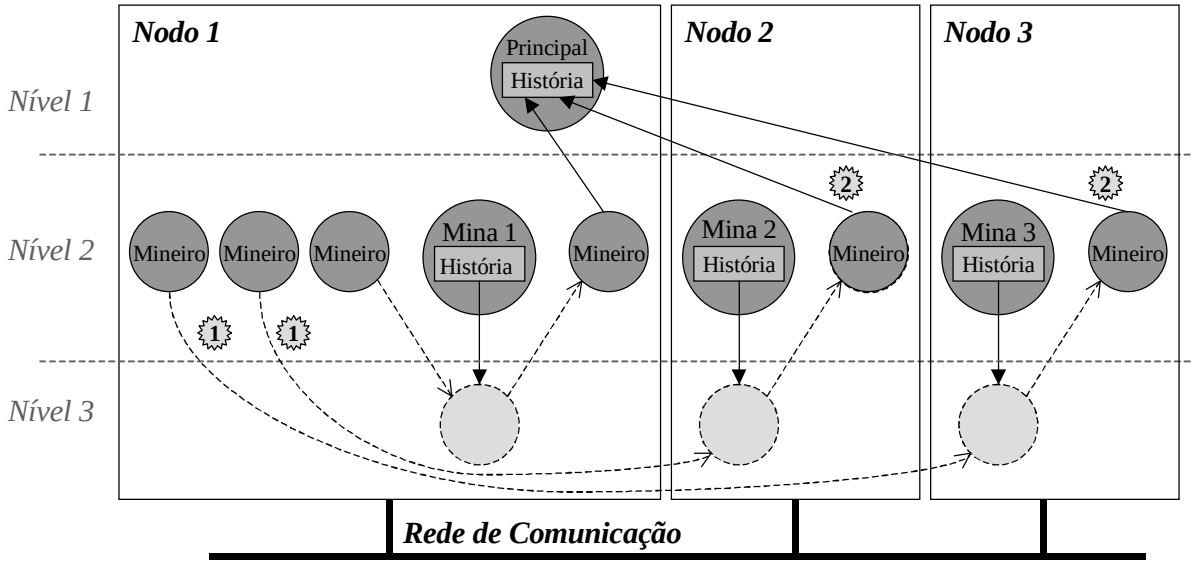


FIGURA 5.25 – *Datamining* paralelo em três nodos (caso C)

A tabela 5.9 contém o custo em cada nodo de uma operação *datamining*. Esta operação consiste em uma leitura (dois campos, uma *string* e um inteiro) no espaço de objetos (por exemplo, marca 2 na figura 5.23). Cada linha da tabela permite as seguintes constatações (os valores de proporção são aproximados):

- Plataforma 1: A diferença de custo entre os nodos resulta da heterogeneidade das estações usadas nesta plataforma. Os nodos 2 e 3 apresentam uma pequena diferença, mas a leitura no nodo 1 é 4 vezes mais lenta;
- Plataforma 2 - casos A e B: Mesma arquitetura da plataforma 1 usando JavaSpaces ao invés de Jada. O uso de JavaSpaces tornou a leitura 500 vezes mas lenta. Esta diferença considerável merece atenção, pois pode inviabilizar o uso de JavaSpaces na implementação do DHolo;
- Plataforma 2 - caso C: Mesma configuração anterior, mas com mineração paralela. O desempenho de uma leitura foi reduzido a metade. Este fato resulta da concorrência ao servidor JavaSpaces durante a mineração paralela. Nos casos A e B o acesso foi sequencial, evitando assim, o acesso concorrente.
- Plataformas 3 e 4: Arquitetura homogênea usando Jada em ambos os casos. Os valores obtidos são praticamente os mesmos.

TABELA 5.9 – Custo de uma operação de mineração (ms)

Operação (uma leitura na história)	**Nodo 1**	**Nodo 2**	**Nodo 3**
Plataforma 1	0,129	0,032	0,040
Plataforma 2 - casos A e B	32	17	20
Plataforma 2 – caso C	42	37	39
Plataformas 3 e 4	0,039	0,036	0,037

A tabela 5.10 apresenta os custos envolvidos no uso da rede de comunicação, ou seja, escrita remota (por exemplo, marcas 2 nas figuras 5.24 e 5.25) e mobilidade física (por exemplo, marcas 1 nas figuras 5.24 e 5.25). Os seguintes comentários sobre cada linha merecem atenção:

- Escrita remota: esta operação consiste basicamente em uma invocação remota de método (RMI) em Java. O custo desta invocação não depende das tecnologias usadas para mobilidade física e espaço de objetos. Nas plataformas 1 e 2 o custo foi aproximadamente 5 vezes maior do que nas plataformas 3 e 4;
- Mobilidade física: esta operação consiste no uso do suporte à mobilidade física para deslocamento de um objeto entre nodos da arquitetura distribuída. Conforme esperado, não houve diferença significativa no desempenho do Voyager nas plataformas 1 e 2. O Voyager foi 3 vezes mais rápido na plataforma 3. Esta diferença é devida à mudança de configuração (hardware e software). Por outro lado, o uso do Horb foi 15 vezes mais rápido do que o Voyager usando a mesma configuração (plataformas 3 e 4). Este fato deve-se a diferenças de tecnologia. O Voyager realiza uma mobilidade física completa, transferindo código e atributos do objeto. Por sua vez, o Horb transfere apenas atributos, exigindo que o código esteja disponível no destino através de algum suporte complementar.

TABELA 5.10 – Custos de comunicação na rede (ms)

Operação	**Plataforma 1**	**Plataforma 2**	**Plataforma 3**	**Plataforma 4**
Escrita remota	193	190	37	36
Mobilidade física	594	580	196	13

Os três casos foram executados em cada plataforma, usando cinco grãos (entre 1.000 e 5.000 operações de mineração). As tabelas 5.11, 5.12, 5.13 e 5.14 mostram os resultados (média e desvio padrão de diversas execuções). Além disso, nas plataformas 3 e 4, o experimento foi estendido com quatro granulosidades (entre 10.000 e 25.000). As figuras 5.26, 5.27, 5.28 e 5.29 apresentam gráficos dos resultados. Os dois últimos são compostos de duas partes. A primeira apresenta os resultados com os grãos entre 1.000 e 5.000 e a segunda entre 5.000 e 25.000. Uma ligação mostra a relação entre as partes. A tabela 5.13 e a figura 5.28 integram ainda os resultados obtidos usando a HoloJava (tabela 5.1 e figura 5.11). O programa *performance.holo* implementa a mesma situação simulada pelo caso A.

Os experimentos permitem as seguintes conclusões em cada plataforma:

- **Plataforma 1:** o caso C é a melhor solução aproximadamente após 2.500 operações. Esta constatação estimula o uso do paralelismo. Além disso, o caso B supera o caso A aproximadamente após 3.750 operações. Este fato ocorre devido ao caso A ter sido executado no nodo de menor poder computacional (nodo 1, veja tabela 5.9). Sendo assim, os custos da rede (tabela 5.10) foram superados pelo uso de nodos mais poderosos na mineração das minas 2 e 3;
- **Plataforma 2:** o considerável aumento do custo de uma operação de mineração (500 vezes), acentua as constatações realizadas durante a análise da plataforma 1. O caso C sempre foi a melhor solução e o caso B sempre supera o caso A;
- **Plataforma 3:** o caso C torna-se a melhor solução próximo a 11.000 operações. Contrastando com as plataformas 1 e 2, o caso B nunca supera o caso A. A igualdade do poder computacional dos nodos (arquitetura homogênea) faz com que os custos de rede tornem o caso B sempre mais lento. O gráfico mostra ainda que o desempenho obtido com a execução usando a HoloJava é praticamente o mesmo obtido na simulação do caso A. Ambos os experimentos usaram a plataforma 3. A única diferença consiste que a simulação executa duas operações de mobilidade física em Voyager, mesmo com a execução ocorrendo em um único nodo. Esta situação ocasiona uma pequena diferença de desempenho em favor da HoloJava;
- **Plataformas 4:** devido ao baixo custo da mobilidade em Horb (veja tabela 5.10), a execução paralela (caso C) sempre foi a mais rápida. Mesmo com este baixo custo de rede, devido à homogeneidade dos nodos, o caso B sempre foi mais lento do que o caso A.

As seguintes constatações gerais merecem ser ressaltadas:

- mesmo com os consideráveis custos de rede envolvidos no experimento (tabela 5.10), nas quatro plataformas houve ganho de desempenho com o uso de paralelismo (caso C). Este fato estimula o uso do Holoparadigma para o processamento paralelo [BAR 2000c, BAR 2001b];
- pensando em funcionalidade, torna-se importante destacar que na maioria dos casos a distribuição das minas está relacionada com localidade. A obtenção de *speed-up* é desejada mas não uma necessidade;
- os experimentos provaram que os seguintes pacotes Java podem ser usados conjuntamente: Voyager e Jada (plataformas 1 e 3), Voyager e JavaSpaces (plataforma 2), Horb e Jada (plataforma 4).

TABELA 5.11 – *Benchmarks* (ms) - Plataforma 1

Número de Operações	Caso A		Caso B		Caso C	
	Média	Desvio	Média	Desvio	Média	Desvio
1.000	746,4	12,4	1.635,3	75,1	1.298,3	66,2
2.000	1.273,6	8,0	1.841,1	36,2	1.413,6	15,4
3.000	1.818,5	17,2	2.079,1	43,6	1.605,0	85,0
4.000	2.409,5	16,8	2.310,7	23,2	1.714,8	56,6
5.000	2.903,9	85,8	2.674,2	142,4	1.874,6	88,4

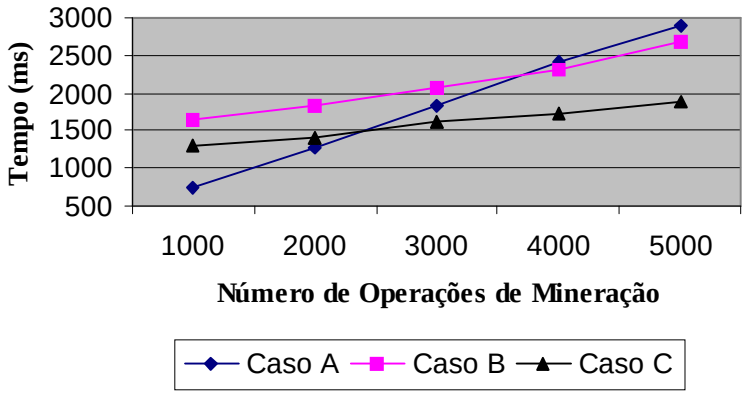


FIGURA 5.26 – *Benchmarks* (ms) - Plataforma 1 - Tabela 5.11

TABELA 5.12 – *Benchmarks* (ms) - Plataforma 2

Número de Operações	Caso A		Caso B		Caso C	
	Média	Desvio	Média	Desvio	Média	Desvio
1.000	107.247,3	4.067,7	78.604,8	3.842,2	45.336,3	775,4
2.000	197.727,0	2.181,8	146.070,5	987,7	86.037,1	2.145,3
3.000	290.483,8	4.862,4	212.098,6	2.251,1	126.861,8	2.937,2
4.000	382.717,5	1.642,9	279.444,4	2.119,5	167.042,6	6.163,3
5.000	470.244,1	3.325,0	343.627,3	1.868,8	208.607,6	1.125,6

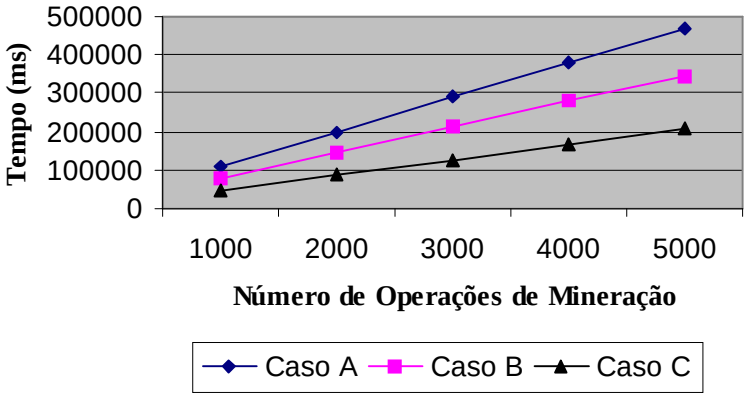


FIGURA 5.27 – *Benchmarks* (ms) - Plataforma 2 - Tabela 5.12

TABELA 5.13 – *Benchmarks* (ms) - Plataforma 3

Número de Operações	Caso A		Caso B		Caso C		HoloJava (*tabela 5.1*)	
	Média	**Desvio**	**Média**	**Desvio**	**Média**	**Desvio**	**Média**	**Desvio**
1.000	414,5	4,1	819,4	791,9	37,6	19,4	378,9	14,9
2.000	484,0	5,3	910,9	840,5	39,9	13,9	446,3	17,9
3.000	555,0	2,5	1.040,6	843,8	25,7	305,4	527,5	15,8
4.000	666,0	4,1	1.087,5	871,1	17,9	53,9	608,8	13,9
5.000	719,9	17,1	1.197,9	927,5	47,2	70,7	680,8	14,2
10.000	1.063,0	19,3	1.594,5	1.122,8	42,5	25,9	1.055,6	17,8
15.000	1.446,4	14,7	2.155,0	1.232,6	53,6	446,5	1.439,0	22,5
20.000	1.795,0	18,8	2.602,6	1.381,3	43,0	502,4	1.800,0	25,0
25.000	2.152,0	45,4	3.040,5	1.495,9	48,3	573,9	2.137,1	35,2

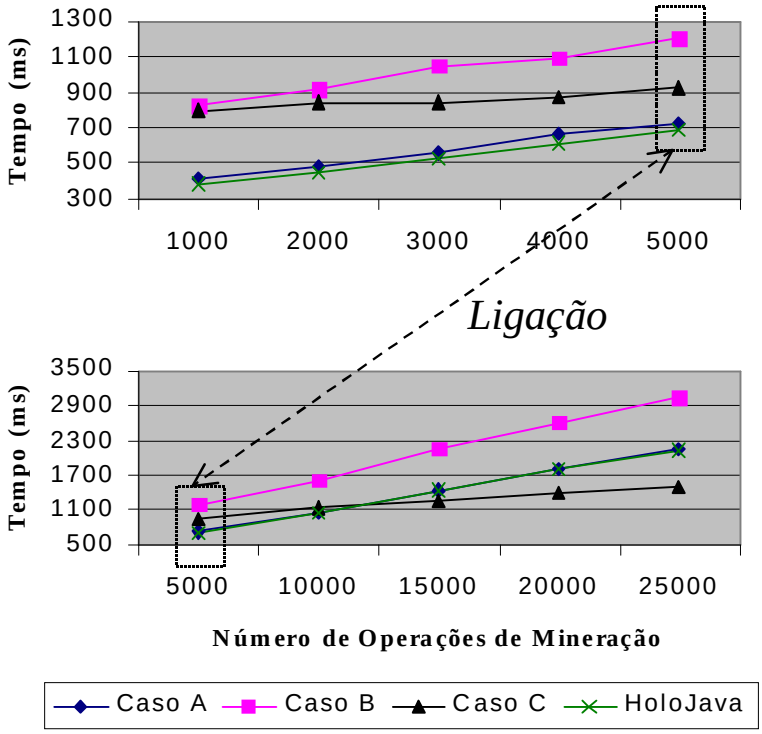


FIGURA 5.28 – *Benchmarks* (ms) - Plataforma 3 - Tabela 5.13

TABELA 5.14 – *Benchmarks* (ms) - Plataforma 4

Número de Operações	Caso A		Caso B		Caso C	
	Média	Desvio	Média	Desvio	Média	Desvio
1.000	600,3	79,0	719,0	105,5	578,5	45,0
2.000	696,1	59,8	813,1	80,3	583,9	41,0
3.000	744,8	114,7	822,3	33,2	678,3	100,4
4.000	839,9	110,3	898,4	52,3	685,1	64,4
5.000	938,5	56,1	1.022,8	71,4	752,3	97,5
10.000	1.292,4	23,0	1.312,5	75,5	864,4	66,3
15.000	1.667,8	55,8	1.884,8	121,9	1.124,5	128,2
20.000	2.015,6	76,1	2.387,5	130,7	1.541,3	85,2
25.000	2.418,0	235,0	2.974,1	179,2	1.976,9	47,5

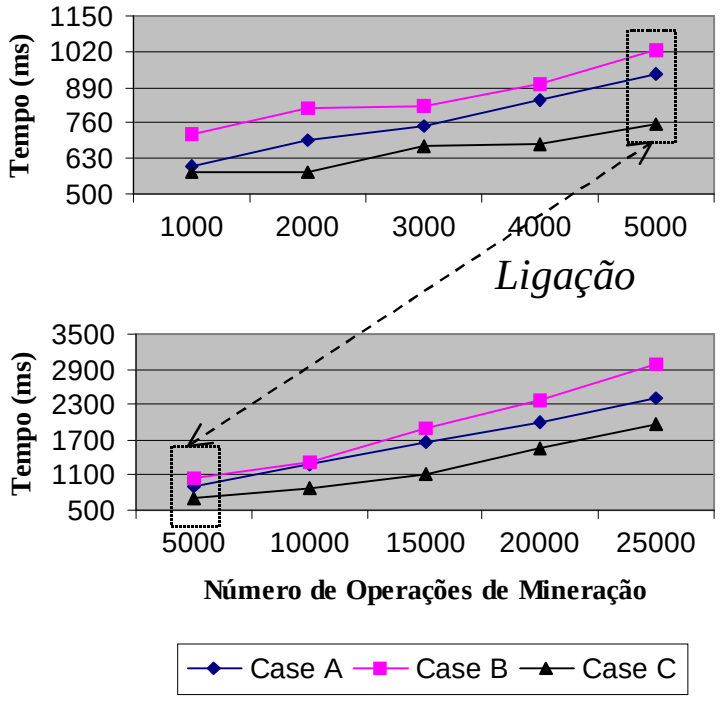


FIGURA 5.29 – *Benchmarks* (ms) - Plataforma 4 - Tabela 5.14

5.4 Considerações finais

Este capítulo discutiu a Holoplataforma, enfatizando seus três componentes: HoloJava [BAR 2001a, BAR 2001c], HoloEnv (*HoloEnvironment*) [BAR 99a, SOA 2000] e DHolo (*Distributed Holo*) [BAR 2001, BAR 2001b]. As principais constatações foram as seguintes:

- a HoloJava permitiu uma rápida prototipação da Hololinguagem. Mesmo com limitações, atualmente diversos holoprogramas podem ser executados, permitindo assim, a obtenção de resultados e a realização de análises. As principais funcionalidades previstas para a Hololinguagem foram implementadas: gerenciamento da mobilidade através da HoloTree (figura 5.4b), mobilidade de grupos, suporte à troca de contexto (história e visibilidade de entes) após a mobilidade, concorrência inter-entes, mecanismo de controle para a concorrência usando invocação implícita (história), todos os tipos de perguntas para a história (tabela 4.1), invocação explícita nos três níveis previstos na visibilidade (interações dos tipos 3, 4 e 5, na figura 3.12). A HoloJava 1.0 é composta de 10 arquivos que totalizam 5.701 linhas em Java;

- o JavaCC [JAV 2001] mostrou-se um ferramenta adequada para desenvolvimento do protótipo. A especificação da HoloJava 1.0 consiste em 1.832 linhas descrevendo a gramática da linguagem e ações semânticas de conversão de Holo para Java. Usando esta especificação, o JavaCC gera 5.609 linhas em Java. A utilização do JavaCC faz com que a especificação da (gramática) mantenha-se sempre atualizada na medida em que a linguagem evolui;

- o Jada e o Prolog Café suportaram a implementação da história e das MLAs. Por outro lado, o Prolog Café mostrou-se lento na conversão de Prolog para Java. Acredita-se que no futuro, essa ferramenta deva ser substituída;

- o uso do HoloEnv simplifica o desenvolvimento de programas. O ambiente integra com sucesso as plataformas usadas para conversão (HoloJava e Prolog Café) e execução (ambiente Java) de holoprogramas. Além disso, possui facilidades de edição, gerenciamento de arquivos e acompanhamento da conversão e execução (janelas *Build* e *Run*). O HoloEnv 1.0 é composto de 27 arquivos totalizando 5.749 linhas em Java;

- as simulações do DHolo permitiram constatações interessantes que servirão de base para uma futura criação do ambiente distribuído. Neste contexto, foram experimentados de forma integrada as seguintes plataformas: Voyager, JavaSpaces, Jada e Horb.

O próximo capítulo apresenta as considerações finais do trabalho.

6 Considerações Finais

Os quatro capítulos anteriores contêm uma seção final tecendo comentários específicos sobre o tema abordado. Este capítulo apresenta considerações finais genéricas organizadas em quatro seções: aplicações previstas, principais contribuições, conclusões e trabalhos futuros.

6.1 Aplicações previstas

O Holoparadigma é um modelo genérico. No entanto, as seguintes características estimulam sua aplicação na solução de problemas específicos:

- **mobilidade e distribuição implícita:** o enfoque na mobilidade lógica estimula a aplicação do modelo em problemas que envolvam deslocamento em nível de modelagem, independentemente de considerações sobre deslocamento entre nodos de arquiteturas distribuídas. A independência entre mobilidades lógica e física motiva a exploração automática do paralelismo e distribuição;
- **multiparadigma:** o enfoque multiparadigma estimula a aplicação do modelo em problemas que possam usufruir da utilização conjunta de comportamento imperativo e lógico;
- **composição dinâmica:** a mobilidade lógica permite a mudança dinâmica da composição de entes (entes compostos). Esta dinamicidade não é suportada diretamente pelos paradigmas básicos. Lea [LEA 2001] discute este aspecto no âmbito da orientação a objetos. O Holoparadigma suporta a solução de problemas que necessitem de composição dinâmica (suporte a grupos);
- **símbolos:** a utilização de símbolos estimula a aplicação do modelo em problemas que possam ser melhor resolvidos com processamento simbólico;
- **modelo de coordenação baseado em invocações implícita e explícita:** o suporte a ambos os tipos de invocação, estimula a aplicação do modelo em problemas que possam ser melhor solucionados com a utilização conjunta de *blackboards* e mensagens.

Destacam-se como principais aplicações previstas para o Holoparadigma:

- ***Datamining*** [BER 97]: a mineração de dados é uma aplicação emergente na área de informática. Atualmente, um dos principais objetivos do *datamining* é a realização de *marketing* de precisão [GOD 2001]. Diversos exemplos no decorrer do texto são dedicados à mineração e demonstram como o encapsulamento de *blackboards* em entes organizados em níveis, permite o uso da mobilidade para mineração em diferentes contextos. Além disso, a distinção entre mobilidades lógica e física, estimula o *datamining* paralelo e distribuído (seção 5.3). Finalmente, o suporte ao comportamento lógico simplifica a implementação de algoritmos que necessitem fazer inferências lógicas para tratamento dos dados minerados (por exemplo, programa *datamining.holo* na figura 5.5 e anexo 3.1);

- **Computação móvel** [AUI 2002]: o enfoque na mobilidade e no controle de contextos usando composição de entes, estimula a aplicação do Holoparadigma na criação de soluções para computação móvel (**ISAM** [AUI 2001, AUI 2002] e **ExEHDA** [ADE 2001]);
- **Aplicações que envolvam grupos** [LIA 90, BIR 93, NUN 98, LEA 2001]: um ente composto assemelha-se a um grupo. Portanto, o Holoparadigma se adapta a aplicações que possam explorar este perfil. Neste contexto, destaca-se a tecnologia *Groupware* [NUN 98, p.14] que pode ser aplicada na criação de vários tipos de sistemas (suporte a reuniões, espaços de trabalho compartilhados, comunicação mediada pelo computador e controle de fluxo de trabalho e de processos). A utilização de grupos em sistemas de Internet [NUN 98, p.16] também é uma possível aplicação para o Holoparadigma;
- **Mundos virtuais** [TAR 93, FEA 99, BRP 2001]: um mundo virtual (ou espaço virtual) é um sistema que suporta um grupo dinâmico de usuários que podem se conectar e desconectar com freqüência. Os usuários interagem entre si e com as entidades virtuais do mundo. Além disso, podem existir vários mundos e os usuários e entidades podem migrar entre eles. Portanto, mundos são sistemas abertos e distribuídos [BRP 2001, p.1]. As entidades virtuais costumam possuir uma vida longa (persistência) e os mundos costumam existir de forma independente de um usuário específico. Entre as principais aplicações previstas para este tipo de sistema estão jogos virtuais que envolvam múltiplos jogadores. O Holoparadigma possui diversas características que estimulam sua aplicação na construção deste tipo de sistema. Os componentes de um mundo virtual (mundos, usuários e entidades) podem ser tratados como entes. Além disso, a mobilidade de usuários e entidades entre mundos assemelha-se a mobilidade lógica proposta pelo Holo. O suporte a vários níveis de entes, permite a modelagem de mundos que encapsulem outros mundos. Por sua vez, a história de um ente suporta a existência (dados no *blackboard*) de um mundo independente dos seus componentes (entidades e usuários). Entre as atividades futuras previstas para o Holoparadigma estão o tratamento de persistência de entes e a tolerância a falhas em sistemas distribuídos. A inserção destas características permitirá a criação de mundos virtuais completos [BRP 2001, p.2]. Finalmente, Tarau et al [TAR 93, p.2] afirmam que entre as principais características de plataformas orientadas à construção de mundos virtuais estão o suporte a um modelo de coordenação de alto nível (proposto na seção 3.7 para Holo) e o suporte adequado à construção de algoritmos que envolvam dedução (comportamento lógico, descrito na seção 3.4 e implementado na Hololinguagem através de MLAs);
- **Agentes** [SHO 93, COM 94, AMA 96, AMA 97, BRAD 97]: o suporte à utilização conjunta de *blackboards* e dos paradigmas em lógica, imperativo e orientado a objetos, estimula a aplicação do Holoparadigma na construção de agentes. A lógica permite a representação declarativa de conhecimento e a busca dinâmica de soluções [AMA 97, p.92], simplificando assim, o gerenciamento de estados mentais [SHO 93, p.60]. Por sua vez, o paradigma imperativo pode ser utilizado para descrição do comportamento deterministico. Os aspectos da orientação a objetos (modularidade, encapsulamento, herança, etc) permitem a estruturação das partes que

compõem um agente [AMA 97, p.20]. Vários trabalhos discutem a aplicação da orientação a objetos na criação de agentes [SHO 93, p. 55; AMA 97, p.59]. Finalmente, o *blackboard* é reconhecido como um modelo adequado para resolução de problemas de inteligência artificial [ALV 97, p.2; PFL 97; p.2];

- **Sistemas multiagentes** [COS 94; ALV 97; AMA 97, p.19; GLA 97; OSS 99; WEI 99]: conforme afirmam Alvares e Sichman [ALV 97, p.3], a área de inteligência artificial distribuída divide-se em duas subáreas: resolução distribuída de problemas (RDP) e sistemas multiagentes (SMA). O Holoparadigma pode ser aplicado em ambas. No caso da RDP, torna-se necessário o suporte à solução distribuída de um problema específico utilizando um controle global implícito (centralizado ou distribuído) [ALV 97, p.5]. Além disso, a interação entre agentes (unidades de processamento) pode ser baseada em troca de mensagens ou compartilhamento de dados. O modelo de coordenação do Holo permite ambos os tipos de interação. Por sua vez, o encapsulamento de entes em vários níveis simplifica a criação de um controle global distribuído para RDPs. Holo estimula ainda a distribuição implícita, simplificando assim, a distribuição do problema abordado pela RDP. No caso de SMAs, assume importância o tratamento (criação, organização e interação) de sociedades abertas de agentes autônomos [ALV 97, p.6]. Neste contexto, destacam-se a mobilidade de agentes entre sociedades [COS 94] e a reorganização das sociedades [GLA 97]. Uma sociedade equivale a um ente composto e a mobilidade entre sociedades assemelha-se à mobilidade lógica proposta pelo Holoparadigma. Conforme destacam Costa et al [COS 94, p.536], o principal problema para criação de SMAs é o mantenimento da integridade funcional de uma sociedade após a movimentação de um agente. Se o agente entra na sociedade, surgem problemas que envolvem reconfiguração da sociedade e estratégias de comunicação com os demais. Se o agente sai, novos agentes devem assumir suas responsabilidades sociais. Uma sociedade implementada como um ente composto, permite a inserção de ações no comportamento que enfoquem estes problemas. Neste caso, a sociedade possuirá um comportamento independentemente de quais entes estiverem nela. A história de uma sociedade pode ser utilizada como repositório de informações a serem consultadas por um ente quando entra na sociedade. Além disso, a possibilidade de níveis de entes compostos permite a existência de sociedades formadas por sociedades.

6.2 Principais contribuições

O capítulo 2 descreve o estado da arte no contexto do Holoparadigma. Os estudos apresentados naquele capítulo aliados com os resultados alcançados, permitem destacar as seguintes contribuições:

- integração de diversos conceitos e linhas de pesquisa da ciência da computação. Neste sentido, o trabalho permite a *integração do conhecimento* defendida por Ortega y Gasset [ORT 92], Bohm [BOH 80] e Einstein [EIN 81];
- desenvolvimento de estudos sobre o estado da arte no contexto do Holoparadigma (capítulo 2) e concretização dos resultados através da criação de uma taxonomia multiparadigma [BAR 2000, BAR 2000d];

- criação dos princípios do Holoparadigma [BAR 99, BAR 2000a, BAR 2001], os quais nortearam o desenvolvimento de todas as atividades do trabalho;
- criação de uma linguagem baseada no paradigma (**Hololinguagem** [BAR 2001a]);
- modelagem e implementação de uma ferramenta para conversão de Holo para Java (ferramenta **HoloJava** [BAR 2001c]);
- especificação e implementação de um ambiente integrado para desenvolvimento de programas em Holo (ambiente **HoloEnv** [BAR 99a, SOA 2000]);
- especificação parcial de um ambiente para execução distribuída de holoprogramas (**DHolo** [BAR 2001, BAR 2001b]). Diversas simulações foram realizadas para avaliação de ferramentas a serem utilizadas na sua futura implementação (seção 5.3);
- participação na aplicação do Holoparadigma em duas áreas emergentes na ciência da computação: *datamining* [BAR 2001b, BAR 2001c] e computação móvel [AUI 2001, AUI 2002, YAM 2001];
- utilização de um *blackboard* lógico que armazena CLEIs. As propostas existentes utilizam *blackboards* que armazenam apenas termos lógicos. Além disso, propõe-se a política FAFI para gerenciamento de CLEIs no *blackboard*;
- encapsulamento de *blackboards* em tipos abstratos de dados (entes). Além disso, organização hierárquica dos *blackboards* encapsulados;
- uso da história como amálgama em entes compostos. A história serve ainda como suporte para a mobilidade e a distribuição;
- completa distinção entre mobilidades lógica e física;
- utilização conjunta das invocações implícita e explícita através de um modelo de coordenação que organiza de forma coerente as duas propostas de comunicação e sincronização;
- baseado no espírito holístico que guia o desenvolvimento do Holoparadigma, propõe-se na seção 3.9 a integração dos domínios estático (programa) e dinâmico (execução). Neste contexto, a clonagem é proposta como única operação para criação de entes em ambos os domínios. Além disso, propõe-se a clonagem múltipla seletiva;
- utilização de cinco tipos de ações multiparadigma (seção 4.2). O uso das ações é baseado em uma política de composição em módulos (ACG, figura 4.12) e em uma política de invocação (AIG, figura 4.13);
- padronização de invocações implícita e explícita (ECI, seção 4.3). O padrão integra ainda os tipos e modos de invocação (seção 3.8) propostos pelo Holoparadigma e suporta múltiplas respostas para uma pergunta;
- estabelecimento de uma política de clonagem (seção 4.5, [BAR 2001a]) de comportamento (BCP) baseada em regras de clonagem (BCR);
- descrição de ações e operadores básicos da linguagem (seções 4.6 e 4.7).

6.3 Conclusões

No final dos capítulos 2, 3, 4 e 5 existem seções de considerações finais, onde podem ser encontradas conclusões relacionadas com os temas abordados no trabalho. Merecem destaque ainda, as seguintes conclusões genéricas:

- o Holoparadigma é uma abordagem multidisciplinar no âmbito da ciência da computação, pois integra conceitos oriundos de várias linhas de pesquisa, tais como: arquitetura de software [IEE 95, SHA 96], paradigmas de programação [BAR 98, GHE 98, SEB 99], multiparadigma [BAR 2000d, BAR 2001], sistemas paralelos e distribuídos [BAL 89, AND 91, JOU 97, IEE 98, SKI 98], mobilidade [ROY 97, IEE 98, LAN 98, FER 99, FER 2001a], *blackboard* [VRA 95, PFL 97], concorrência [AND 83; SEB 99, p.489; HAS 2000] e grupos [BIR 93, LIA 90, NUN 98, LEA 2001];
- o Holoparadigma é um modelo genérico. Espera-se que este modelo sirva de guia para o desenvolvimento de diversas linguagens de programação. Neste sentido, o modelo proposto poderá ser considerado um paradigma quando assumir uma posição de destaque entre desenvolvedores de linguagens;
- a Hololinguagem implementa os principais conceitos do Holoparadigma. Sendo assim, serve de instrumento para corroboração das idéias propostas pelo modelo. Por sua vez, a ferramenta HoloJava permite a execução de programas desenvolvidos em Holo, corroborando assim, a Hololinguagem;
- a aprovação do projeto Holoparadigma em dois editais da FAPERGS (edital 2000/1 para projetos novos e edital PROADE 2001) impulsionou o desenvolvimento do trabalho. Os recursos obtidos foram utilizados para concretização das idéias apresentadas neste texto;
- os dois principais objetivos traçados no princípio do trabalho foram alcançados, ou seja: (1) criação de uma estrutura para desenvolvimento de pesquisas relacionadas com linguagens multiparadigma e sistemas distribuídos (Holoparadigma, Hololinguagem, HoloJava, HoloEnv e DHolo); (2) estímulo a integração de atividades de pesquisa no contexto onde o Holoparadigma está sendo desenvolvido (veja trabalhos futuros na próxima seção).

6.4 Trabalhos futuros

Este texto descreve os primeiros passos no surgimento do Holoparadigma. Durante essa etapa, foram detectadas atividades que transcendem os limites deste trabalho. Neste contexto, merecem destaque as seguintes atividades futuras:

- aperfeiçoamento da HoloJava. A HoloJava 1.0 descrita na seção 5.1 contém diversas limitações e não implementa aspectos da Hololinguagem considerados indispensáveis para a programação em Holo. Sendo assim, serão mantidos esforços para o aprimoramento desta ferramenta;
- aperfeiçoamento do HoloEnv. O HoloEnv 1.0 descrito na seção 5.2 possui as funcionalidades básicas para o desenvolvimento de holoprogramas. Diversas melhorias podem ser realizadas para comodidade do usuário e aumento da eficiência na holoprogramação;

- depuração de holoprogramas. A criação de um depurador [ROS 96] para programas em Holo (**HoloDebugger**) tornará a programação mais simples e segura, estimulando o uso da linguagem. O Holodebugger será integrado no HoloEnv, permitindo assim, facilidades de depuração através do ambiente;
- desenvolvimento da semântica formal da Hololinguagem. Conforme discutido por Cardelli [CAL 99, p. 23; CAL 99b, p.83], a especificação de linguagens para computação móvel é uma das principais aplicações do Cálculo de Ambientes [CAL 2000, CAL 2000a]. A seção 3.6 traça um paralelo entre este formalismo e os conceitos usados pelo Holoparadigma. No futuro a Hololinguagem poderá ser especificada usando este instrumento;
- criação de uma máquina virtual multiparadigma (**HoloVM**, figura 3.16). Neste contexto, serão especificadas a arquitetura da máquina e o código virtual multiparadigma. Além disso, será criado um compilador de holoprogramas para o código virtual. A criação de uma plataforma própria para a Hololinguagem tornará a execução de programas mais eficiente;
- integração do HoloEnv e da HoloVM. Esta integração tornará o HoloEnv independente da plataforma Java. A HoloJava poderá ser mantida no HoloEnv, criando-se assim uma estrutura que permitirá ao usuário optar pela geração e execução orientada a arquitetura Holo ou a arquitetura Java;
- especificação e implementação de um ambiente de execução distribuída para holoprogramas. A seção 5.3 contém a gênese deste ambiente (*Distributed Holo*). Além disso, atualmente existem diversas atividades orientadas para este objetivo [FER 2001a, SIL 2001, SIU 2001, YAM 2001, AUI 2002];
- estudo dos seguintes temas considerados relevantes para a evolução do trabalho: tratamento da persistência [SES 96] de entes, tolerância a falhas em sistemas distribuídos [TAN 95, p.212] baseados no DHolo, tratamento de exceções [SEB 99] na Hololinguagem e a criação de padrões de projeto baseados nos conceitos do Holoparadigma [PRE 2000];
- especificação de uma linguagem visual de modelagem (semelhante à UML [FUR 98, LAR 98]) para o Holoparadigma. Esta linguagem deverá suportar a especificação da história, a Holoclonagem e o gerenciamento de ações multiparadigma. O diagrama de entes (figuras 4.17 e 4.18, [BAR 2001a]) servirá de base para esta atividade;
- implementação de uma ferramenta de modelagem e/ou programação visual para o Holoparadigma (**HoloCASE**, figura 3.16). Um protótipo inicial para a HoloCASE foi criado por Soares [SOA 2000]. Além disso, uma ferramenta de programação visual para Java, recentemente criada na UFRGS (**DOBuilder** [MAL 2001]), poderá ser adaptada para o Holoparadigma;
- incorporação do paradigma funcional na Hololinguagem [DUB 2001]. O anexo 1.7 contém os principais conceitos para realização dessa atividade;
- análise estática de programas multiparadigma. Os estudos sobre análise global de programas no paradigma em lógica [BAR 96, BAR 96b, BAR 97, AZE 98, AZE 98a, AZE 99, BAR 2000b] e nos paradigmas imperativo e orientados a objetos [AZE 2001, AZE 2001a] servirão de suporte para o desenvolvimento de atividades relacionadas com análise estática de holoprogramas.

Anexo 1 Tópicos Complementares

Este anexo apresenta tópicos relacionados com os capítulos 3 (Holoparadigma) e 4 (Hololinguagem). Os temas abordados são considerados complementares. Sendo assim, visando a simplificação do texto foram colocados em anexo.

1.1 Arquiteturas distribuídas

Shatz e Wang [SHS 89, p.1] definem um **sistema computacional distribuído** como um sistema composto de um conjunto de computadores (nodos de processamento) interconectados por uma rede de comunicação. A figura A1.1 mostra sua organização. Afirmam ainda os autores, que o sistema distribuído é considerado fracamente acoplado porque os processadores comunicam-se trocando mensagens através da rede. Além disso, alguns dos nodos de processamento podem ser compostos de conjuntos de computadores, os quais podem se comunicar através de memória compartilhada. Através dessa afirmação, os autores destacam duas importantes características dos sistemas distribuídos, ou seja, a organização hierárquica e a heterogeneidade. Nos próximos parágrafos a definição de Shatz e Wang será aperfeiçoada e o termo **arquitetura distribuída** será utilizado para nomear a nova organização computacional proposta.

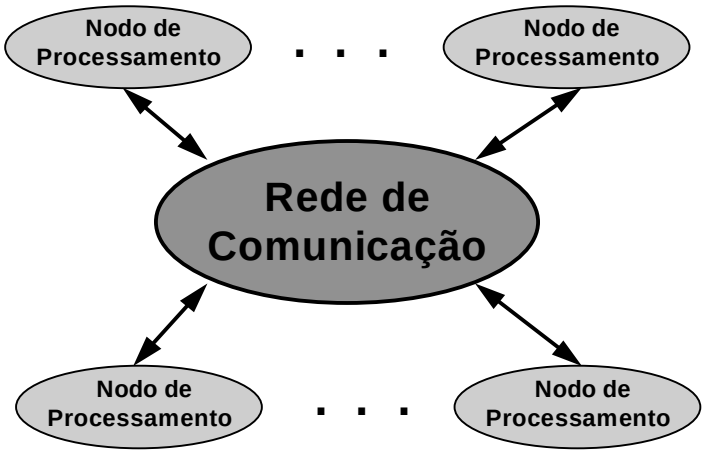


FIGURA A1.1 – Organização de um sistema computacional distribuído

No ponto de vista de Shatz e Wang, o nodo de um sistema distribuído pode ser composto de vários computadores. Sendo assim, o sistema distribuído fica restrito a apenas um nível hierárquico, conforme mostrado na figura A1.1. Por outro lado, em uma **arquitetura distribuída** um nodo pode ser composto de outros nodos. Sendo assim, a arquitetura distribuída é organizada em vários níveis hierárquicos. Cada nível é composto de vários nodos que se comunicam através de mensagens. Nos níveis mais básicos, a definição de Shatz e Wang será aplicada, ou seja, um nodo será composto de um conjunto de computadores, os quais também serão considerados nodos. Surgem assim, as seguintes definições:

- **Definição 1: Arquitetura Distribuída**

Um arquitetura distribuída é uma organização computacional formada de nodos que se comunicam via mensagens trocadas através de um meio de comunicação.

- **Definição 2: Nodo**

Um nodo é o componente organizacional da arquitetura distribuída.

- **Definição 3: Nodo Elementar**

Um nodo elementar é formado por um ou mais processadores acessando uma memória. Sendo assim, um uniprocessador é um nodo elementar. Essa configuração é denominada **arquitetura elementar**. Em complemento, uma arquitetura paralela baseada em compartilhamento é também um nodo elementar.

- **Definição 4: Nodo Composto**

Um nodo composto é um nodo formado por outros nodos, os quais podem ser elementares e/ou compostos.

- **Definição 5: Nodo Homogêneo**

Um nodo homogêneo é um nodo composto de nodos do mesmo tipo, ou seja, somente elementares ou somente compostos.

- **Definição 6: Nodo Heterogêneo**

Um nodo heterogêneo é um nodo composto de nodos de tipos diferentes, ou seja, um ou mais nodos elementares misturados com um ou mais nodos compostos.

- **Definição 7: Nodo Homogêneo Básico ou Nodo Básico**

Um nodo básico é um nodo composto de nodos elementares.

- **Definição 8: Nodo Homogêneo Avançado ou Nodo Avançado**

Um nodo avançado é um nodo composto de nodos compostos.

A figura A1.2 organiza hierarquicamente os possíveis tipos de nodos existentes em uma arquitetura distribuída.

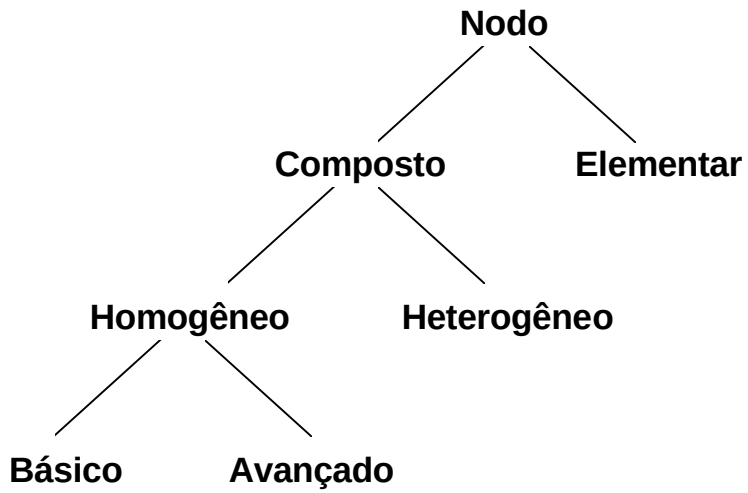


FIGURA A1.2 – Classificação dos nodos das arquiteturas distribuídas

Tendo como base as definições surge o primeiro postulado:

Postulado 1 - Equivalência Arquitetura Distribuída/Nodo Composto: Todo nodo composto equivale a uma arquitetura distribuída e vice-versa.

Através deste postulado deduz-se que os tipos de nodos compostos apresentados na figura A1.2 podem também ser utilizados para classificação das arquiteturas distribuídas. Além disso, a terceira definição permite a criação de um segundo postulado.

Postulado 2 - Relação Arquitetura Elementar/Nodo Elementar: Toda arquitetura elementar é um nodo elementar, no entanto, nem todo nodo elementar é uma arquitetura elementar (caso dos multiprocessadores).

As definições e postulados introduzidos nesta seção podem ser utilizadas para criação de uma taxonomia para arquiteturas de sistemas computacionais. A figura A1.3 apresenta uma proposta de taxonomia.

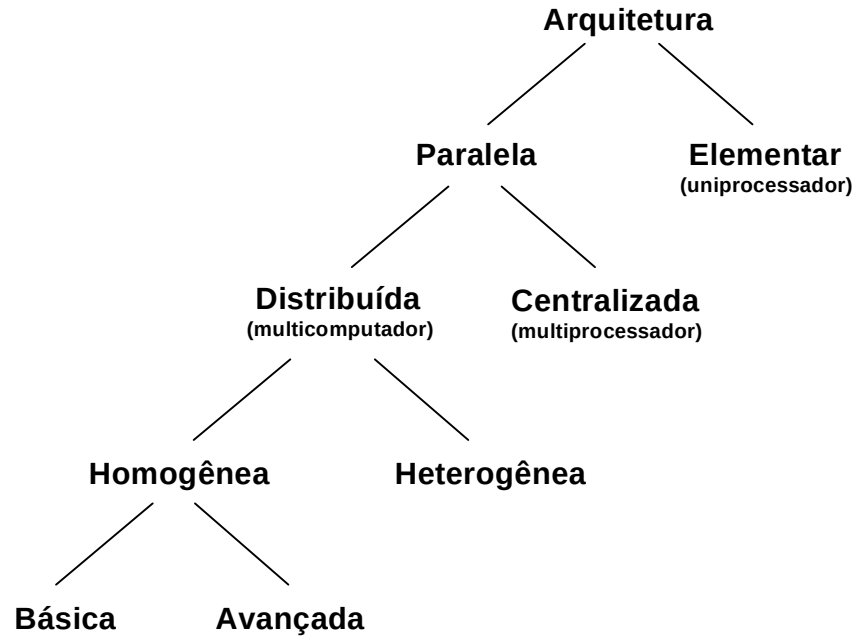


FIGURA A1.3 – Taxonomia para arquiteturas de sistemas computacionais

Nos últimos anos, o número de computadores e redes de comunicação vem aumentando de forma significativa. Em complemento, nota-se um crescente interesse pelo uso das redes de computadores como arquiteturas paralelas [JOU 97, IEE 98]. A presente realidade permite a previsão de que em um futuro próximo as redes, organizadas através de diversos níveis, constituirão a base para organização dos sistemas computacionais. A figura A1.4 apresenta uma rede de computadores fictícia composta de vinte e oito estações de trabalho organizadas em diversos níveis e topologias. Por sua vez, a figura A1.5 mostra a mesma rede como uma arquitetura distribuída destacando os diversos níveis de nodos. A figura A1.6 mostra a arquitetura distribuída do ponto de vista da composição, sem levar em consideração as conexões entre as estações. Nesta figura utiliza-se o círculo como representação de nodos compostos e o quadrado como representação de nodos elementares. A figura mostra a

evolução dos detalhes da arquitetura na medida em que é decomposto cada nodo. Assim, ficam explícitos os vários níveis de abstração da arquitetura distribuída.

Do ponto de vista de distribuição espacial, as redes podem ser classificadas em dois tipos, ou seja, LANs e WANs. Ambos os tipos podem ser organizados em arquiteturas distribuídas e representados através da notação mostrada nas figuras A1.4, A1.5 e A1.6. Tomando-se como exemplo as redes de computadores existentes em um país, pode-se considerar o país como o primeiro nodo da arquitetura distribuída (WAN nacional interligando os estados). Por sua vez, as WANs de cada estado comporiam os nodos do segundo nível. Em um terceiro nível poderiam ser encontradas as WANs dos municípios conectados à rede estadual. Por último, em cada município poderiam haver várias LANs (universidades, empresas, ambientes domésticos, etc).

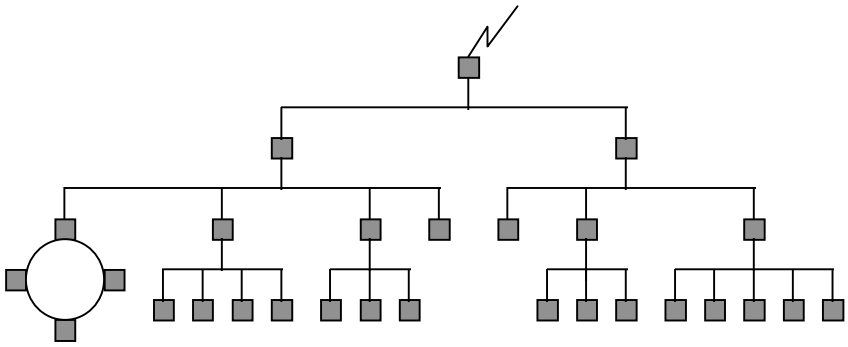


FIGURA A1.4 – Rede de computadores fictícia

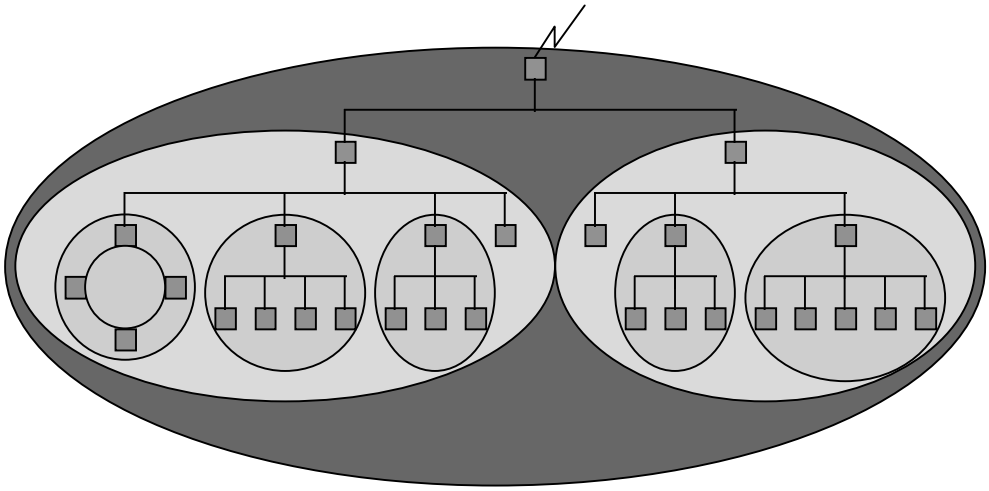


FIGURA A1.5 – Rede como arquitetura distribuída

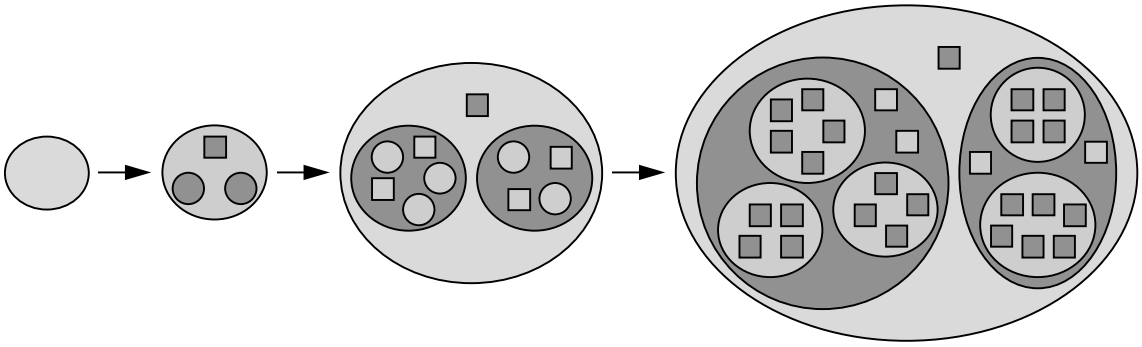


FIGURA A1.6 – Níveis de abstração da arquitetura distribuída

As principais vantagens introduzidas pelas arquiteturas distribuídas são:

- **Escalabilidade**: o poder de processamento (poder das CPUs) de uma arquitetura distribuída está diretamente relacionado com a quantidade e o poder dos seus nodos elementares. Sendo assim, o simples acréscimo de novos nodos ou a simples troca de nodos já existentes por outros mais poderosos aumenta o potencial da arquitetura. Esta flexibilidade é valiosa para organização dos sistemas computacionais. A eficiência de uma arquitetura distribuída depende ainda da velocidade dos canais de comunicação e da qualidade do software que gerencia o sistema. Neste sentido, a gerência da escalabilidade assume uma posição de destaque;

- **Confiabilidade**: a arquitetura distribuída permite a implementação de várias técnicas de tolerância a falhas baseadas na redundância introduzida pela existência de múltiplos nodos elementares (múltiplos recursos, tais como: processadores e discos). As características introduzidas pelas arquiteturas distribuídas trazem novos problemas de confiabilidade que devem ser resolvidos por novas técnicas de tolerância a falhas. Por exemplo, o fluxo de mensagens entre nodos é uma fonte de falhas que deve ser cuidadosamente pesquisada. O simples envio de uma mensagem não significa sua recepção. Vários fenômenos criam riscos durante sua viagem entre emissor e receptor;

- **Localidade**: a natureza distribuída da arquitetura descrita nessa seção é bastante atraente para organização dos sistemas computacionais, pois permite a distribuição espacial dos recursos. Essa característica coincide com a forma da humanidade organizar suas instituições. Atualmente, os computadores estão distribuídos em vários níveis dentro de cada organização (países, estados, municípios, cidades, bairros, universidades, empresas, ambientes domésticos, etc). As arquiteturas distribuídas refletem de forma natural essa realidade.

1.2 Índices de entes

Os entes podem ser organizados em vários níveis. Considerando essa capacidade de composição, podem ser criados índices que facilitem a descrição dos entes. Um **Índice de Ente** (INE) descreve algum aspecto relevante de um ente. A seguir são apresentados dez INEs:

- **Nível de Abstração (NIA):** os entes compostos podem ser visualizados em vários níveis de abstração. O NIA estabelece o nível que deve ser usado para percepção de um ente. Os NIAs são numerados crescentemente, sendo que o nível mais alto de abstração é 0. Um ente elementar possui apenas um NIA, ou seja, o nível 0. Um ente composto possui pelo menos dois NIAs, ou seja, 0 e 1. O ente mostrado na figura 3.5c pode ser percebido em três NIAs. No NIA 0 visualiza-se o ente como uma unidade, sem considerar sua composição. No NIA 1 percebe-se o ente como composto de dois entes compostos e um ente elementar. A figura mostra o NIA dois, ou seja, toda a composição do ente;

- **Total de Níveis de Abstração (TONIA):** o TONIA determina o total de níveis de abstração existentes em um ente. Um ente elementar possui um TONIA igual a 1. Por outro lado, o TONIA de um ente composto deve ser maior do que 1, no entanto, não possui limite máximo. O TONIA estabelece o intervalo de variação do NIA de acordo com a regra 1.

$$\mathbf{0 \leq NIA \leq TONIA - 1} \quad \textit{(1)}$$

O ente mostrado na figura 3.5c possui um TONIA igual a 3. Aplicando a regra 1 descobre-se que seu NIA pode variar entre 0 e 2.

- **Nível de Composição (NIC):** o NIC determina a profundidade de visualização da composição de um ente. O NIC assemelha-se ao NIA, no entanto, enfoca a composição e aplica-se somente a entes compostos. Ambos os índices possuem sempre o mesmo valor, no entanto, não existe NIC 0. Os NICs são numerados a partir de 1 e crescem na medida em que desvenda-se a composição de um ente. O ente mostrado na figura 3.5c possui dois NICs. O NIC 1 mostra três entes componentes, dois compostos e um elementar. A figura visualiza o NIC 2. As seguintes regras podem ser utilizadas para relacionamento do índice NIC com os índices NIA e TONIA:

$$\mathbf{NIC = NIA} \quad \textit{(2)}$$

$$\mathbf{1 \leq NIC \leq TONIA - 1} \quad \textit{(3)}$$

Tendo como base a regra 2, sempre que for indiferente o enfoque do índice (composição ou abstração) será utilizado somente o termo **nível** e a abreviatura NI. Por exemplo, existem dois entes compostos e um ente elementar no nível 1 do ente mostrado na figura 3.5c;

- **Total de Níveis de Composição (TONIC):** este índice determina o total de níveis de composição existentes em um ente. Um ente elementar possui TONIC igual a 0. O TONIC de um ente composto possui valor mínimo igual a 1 e não possui limite máximo. A seguinte regra relaciona os índices NIC e TONIC:

$$\mathbf{1 \leq NIC \leq TONIC} \quad \textit{(4)}$$

Em complemento, as seguintes regras podem ser usadas para o relacionamento do TONIC com os índices NIA e TONIA:

$$\mathbf{0 \leq NIA \leq TONIC} \quad \textit{(5)}$$

$$\mathbf{TONIC = TONIA - 1} \quad \textit{(6)}$$

O ente mostrado na figura 3.5C possui um TONIC igual a 2. Aplicando-se as regras 4 e 5 determina-se que seu NIC pode variar entre 1 e 2 e seu NIA pode variar entre 0 e 2. Em complemento, a regra 6 determina que seu TONIA é 3.

- **Número de Entes Componentes Elementares (NECE):** o NECE determina o número de entes componentes elementares localizados em um determinado nível. O NECE pode ser obtido através da seguinte pergunta: quantos entes elementares surgem se desvendado um determinado nível? Um ente elementar não possui índice NECE. Por sua vez, nos entes compostos o NECE está sempre ligado a um nível. Por exemplo, na figura 3.5c:

  Se NI = 1 então NECE = 1

  Se NI = 2 então NECE = 6

  O NIA zero dos entes compostos não possui índice NECE.

- **Número de Entes Componentes Compostos (NECO):** este índice indica o número de entes componentes compostos *que se localizam* em um determinado nível. Um ente elementar não possui índices NECO. Nos entes compostos o NECO está relacionado com um nível. Na figura 3.5c:

  Se NI = 1 então NECO = 2

  Se NI = 2 então NECO = 0

  O NIA zero dos entes compostos não possui índice NECO.

- **Número de Entes Componentes (NEC):** este índice determina o número de entes componentes *que se localizam* em um determinado nível de um ente. Um ente elementar não possui índices NEC. Por sua vez, nos entes compostos o NEC deve estar relacionado com um nível. Por exemplo, na figura 3.5c:

  Se NI = 1 então NEC = 3

  Se NI = 2 então NEC = 6

  O NIA zero dos entes compostos não possui índice NEC. Além disso, em um determinado nível de um ente, a seguinte regra relaciona os índices NEC, NECO e NECE:

  NEC = NECO + NECE *(7)*

  Por exemplo, na figura 3.5c:

  Se NI = 1 então NEC = NECO + NECE = 2 + 1 = 3

  Se NI = 2 então NEC = NECO + NECE = 0 + 6 = 6

- **Total de Entes Componentes Elementares (TECE):** o índice TECE determina o número total de entes componentes elementares que existem em um ente. Todos os níveis de um ente são considerados. Por exemplo, o ente da figura 3.5c possui TECE igual a 7. Se o índice TECE é vinculado a um nível, determina o total de entes componentes elementares até o nível estabelecido. Por exemplo, no ente da figura 3.5c:

  Se NI = 1 então TECE = 1

  Se NI = 2 então TECE = 7

  O NIA 0 não possui índice TECE. Além disso, o índice TECE do último nível de um ente é igual ao TECE do ente.

- **Total de Entes Componentes Compostos (TECO):** o índice TECO determina o número total de entes componentes compostos existentes em um ente. O índice TECO considera todos os níveis de um ente. Por exemplo, na figura 3.5c o ente possui TECO igual a dois. Se o índice TECO for associado a um nível de um ente, determina o total de entes componentes compostos até o nível previsto. Na figura 3.5c:

  Se NI = 1 então TECO = 2

  Se NI = 2 então TECO = 2

  O NIA 0 não possui índice TECO. Além disso, o TECO do último nível de um ente é igual ao TECO do ente.

- **Total de Entes Componentes (TEC):** este índice determina o número total de entes componentes existentes em um ente. O índice TEC considera todos os níveis e ambos os tipos de entes (elementares e compostos). Por exemplo, o ente mostrado na figura 3.5c possui TEC igual a 9. Por outro lado, se o índice TEC for associado a um determinado nível de um ente, ele determina o número total de entes componentes existentes até o nível estabelecido. Por exemplo, na figura 3.5c:

  Se NI = 1 então TEC = 3

  Se NI = 2 então TEC = 9

  O NIA 0 não possui índice TEC. Além disso, o índice TEC do último nível de um ente é igual ao índice TEC do ente. A seguinte regra relaciona os índices TEC, TECO e TECE:

  TEC = TECO + TECE *(8)*

  Por exemplo, na figura 3.5c:

  TEC = TECO + TECE = 2 + 7 = 9

  Se NI = 1 então TEC = TECO + TECE = 2 + 1 = 3

  Se NI = 2 então TEC = TECO + TECE = 2 + 7 = 9

1.3 Compartilhamento de entes

Entes podem compartilhar entes. Um **ente compartilhado** é componente de dois ou mais **entes compartilhadores**. Um ente compartilhado pode ser elementar ou composto. Um ente elementar compartilhado, gera um **compartilhamento elementar**. Por sua vez, um ente composto compartilhado, gera um **compartilhamento composto**. Além disso, entes podem compartilhar vários entes. O número de entes compartilhados é denominado **grau de compartilhamento**. Quando os entes compartilhados são do mesmo tipo (elementares ou compostos), tem-se um **compartilhamento homogêneo**. Por outro lado, quando os entes compartilhados são de tipos diferentes surge um **compartilhamento heterogêneo**.

A figura A1.7 mostra três entes compartilhando um ente elementar. Este é um caso de *compartilhamento elementar de grau um*. Os compartilhamentos de grau um sempre são homogêneos. Conforme mostrado na figura, o ente compartilhado tem acesso à história dos entes compartilhadores. Sendo assim, torna-se um elo de ligação entre eles. Um ente compartilhado atua como um conector (semelhante aos conectores da arquitetura de software [SHA 95, SHA 96]), o qual regula o fluxo de informações entre entes através de uma lógica própria.

Os índices de entes apresentados na seção anterior podem ser utilizados para descrição dos aspectos relevantes dos compartilhamentos. Neste caso, passam a ser chamados de **índices de compartilhamento**. A aplicação dos índices está baseada na **analogia compartilhamento/ente**. Essa analogia indica que o compartilhamento pode ser compreendido como um ente composto virtual. O compartilhamento equivale ao nível de abstração 0 de um ente. O grau de compartilhamento equivale ao TEC do nível um. Entes compostos podem ser compreendidos como conjuntos. Sendo assim, o compartilhamento pode ser tratado como uma operação de interseção e a **Teoria dos Conjuntos** [LIP 78, FIL 85] pode ser aplicada na análise dos entes e do compartilhamento entre eles.

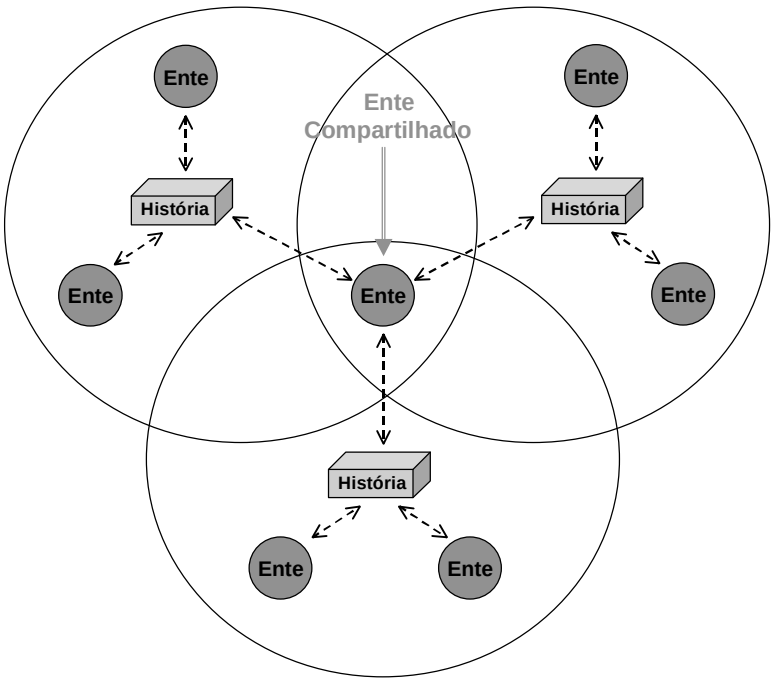


FIGURA A1.7 – Exemplo de ente compartilhado

1.4 Agregação e fragmentação

Além do processo de clonagem descrito na seção 3.9, os entes podem ser manipulados através das seguintes técnicas:

- **Agregação**: ocorre quando um ente composto é criado pela agregação de entes dinâmicos já existentes. Um ente criado por agregação é denominado **ente agregador**. Os entes que foram reunidos são denominados **entes agregados**. Os entes agregados são extintos. As partes do agregador (interface, comportamento, componentes e história) são formadas pela fusão construtiva das partes dos agregados (utilizando as políticas de clonagem discutidas na seção 3.9). Portanto, a agregação é uma clonagem dinâmica e construtiva que extingue os clonados. Os entes agregados sempre estão em um mesmo nível. O ente agregador posiciona-se neste nível. Além disso, os entes agregados podem ser elementares e compostos. A agregação de entes elementares resulta em um ente elementar e a agregação de entes compostos resulta em um ente composto. A agregação de entes elementares e compostos resulta em um ente composto. Denomina-se **agregação homogênea** quando os entes agregados são do mesmo tipo. Neste caso, surge um **ente agregador homogêneo**. Por sua vez, quando a agregação envolve entes de tipos diferentes ocorre uma **agregação heterogênea** e surge um **ente agregador heterogêneo**. A figura A1.8 exemplifica a criação de um ente agregado no nível 1 do ente mostrado na figura 3.5c;

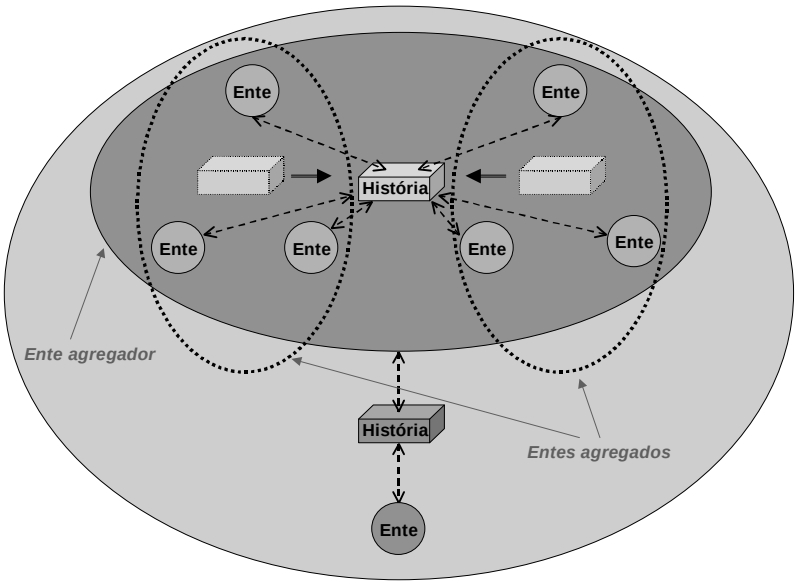


FIGURA A1.8 – Agregação de entes

- **Fragmentação:** Ocorre quando um ente composto é fragmentado em dois ou mais entes. O **ente fragmentado** deixa de existir. Os entes criados por fragmentação são denominados **entes fragmentos** ou somente **fragmentos**. Os fragmentos herdam o comportamento, a história e a interface do fragmentado. Os entes componentes são distribuídos pelos fragmentos. Os fragmentos passam a existir no mesmo nível em que estava o ente que sofreu o processo de fragmentação. A figura A1.9 exemplifica a fragmentação de um dos entes compostos no nível 1 da figura 3.5c. Neste caso, são criados dois novos entes.

- **Desagregação**: ocorre quando um ente composto é desagregado, ou seja, deixa de existir, mas seus entes componentes permanecem. Neste caso, o comportamento, a interface e a história do extinto desaparecem. Os entes componentes passam a existir no nível do desagregado.

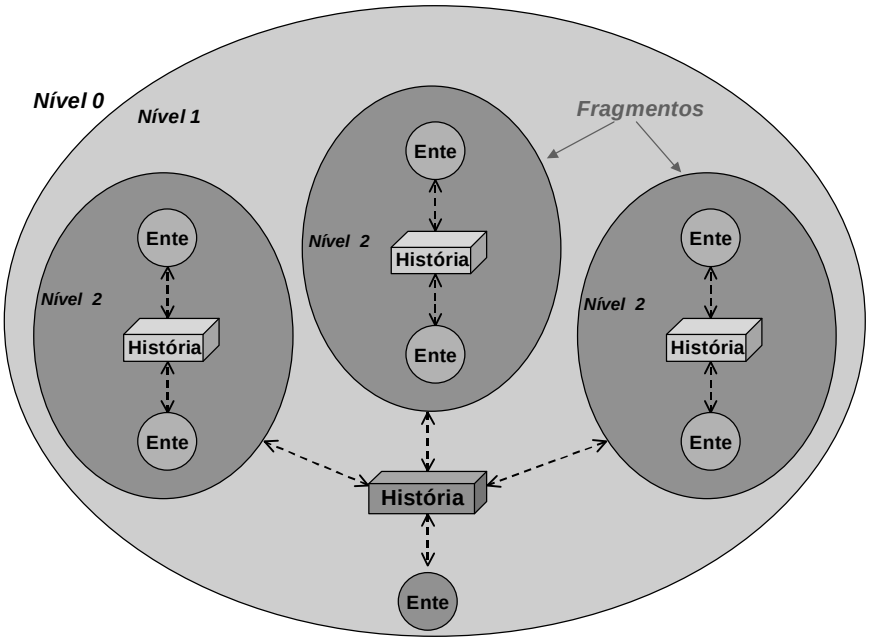


FIGURA A1.9 – Fragmentação de entes

1.5 Holomodelagem

O processo de criação de um modelo computacional para representação de uma realidade é conhecido como modelagem. Toda modelagem tem como base um paradigma de desenvolvimento de software. No paradigma são encontradas as abstrações que suportam a criação do modelo. A modelagem através de Holo é denominada **Holomodelagem**. Além disso, o modelo criado através do Holoparadigma é chamado **Holomodelo**.

No âmbito do Holoparadigma toda existência é um ente. Sendo assim, o holomodelo pode ser percebido como um ente. O nível 0 deste ente representa o nível mais alto de abstração do modelo. Por sua vez, os níveis de abstração mais baixos representam os vários níveis de composição do holomodelo. A figura A1.10 mostra a visão do modelo computacional como um ente resultante da holomodelagem.

A holomodelagem é uma abordagem *top-down*. Após a definição do ente de mais alto nível de abstração (nível 0) torna-se necessária a definição dos seus entes componentes, os quais localizam-se no nível 1. Este processo de refinamento deve prosseguir até a definição de todos os níveis de composição do holomodelo. Sendo assim, a estrutura organizacional do holomodelo representa a organização da realidade modelada. Durante a modelagem torna-se necessária a definição de **limites de composição**, ou seja, a criação de entes elementares que definam o término do refinamento.

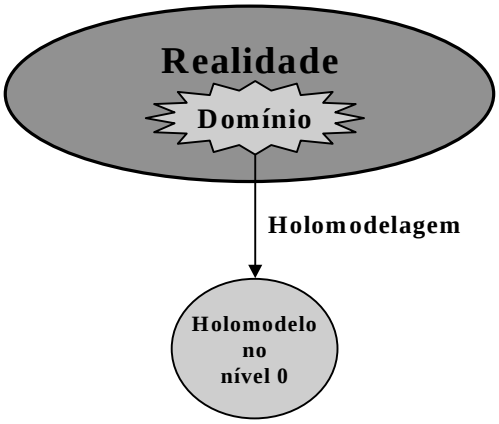


FIGURA A1.10 – Modelo computacional como um ente

O limite de composição depende dos níveis finais de abstração a serem modelados. Por exemplo, o corpo humano pode ser percebido como um ente composto, o qual pode ser modelado em vários níveis. A figura A1.11 mostra um exemplo de holomodelagem do corpo humano. Uma modelagem em três níveis (figura A1.11a) pode estabelecer que o ente corpo possui seis entes componentes, ou seja: dois braços, duas pernas, uma cabeça e um tronco. Por sua vez, os entes componentes são compostos de entes elementares representando as células. Neste caso, o limite de composição localiza-se nas células. No entanto, o limite poderia ser estendido para níveis mais baixos de abstração. Uma célula poderia ser modelada em moléculas e as moléculas modeladas em átomos (figura A1.11b). Neste caso, o limite de composição seria estabelecido nos átomos e o corpo seria modelado em cinco níveis. Outros níveis de abstração poderiam ser criados. Por exemplo, algumas células do tronco poderiam ser organizadas em entes compostos denominados órgãos.

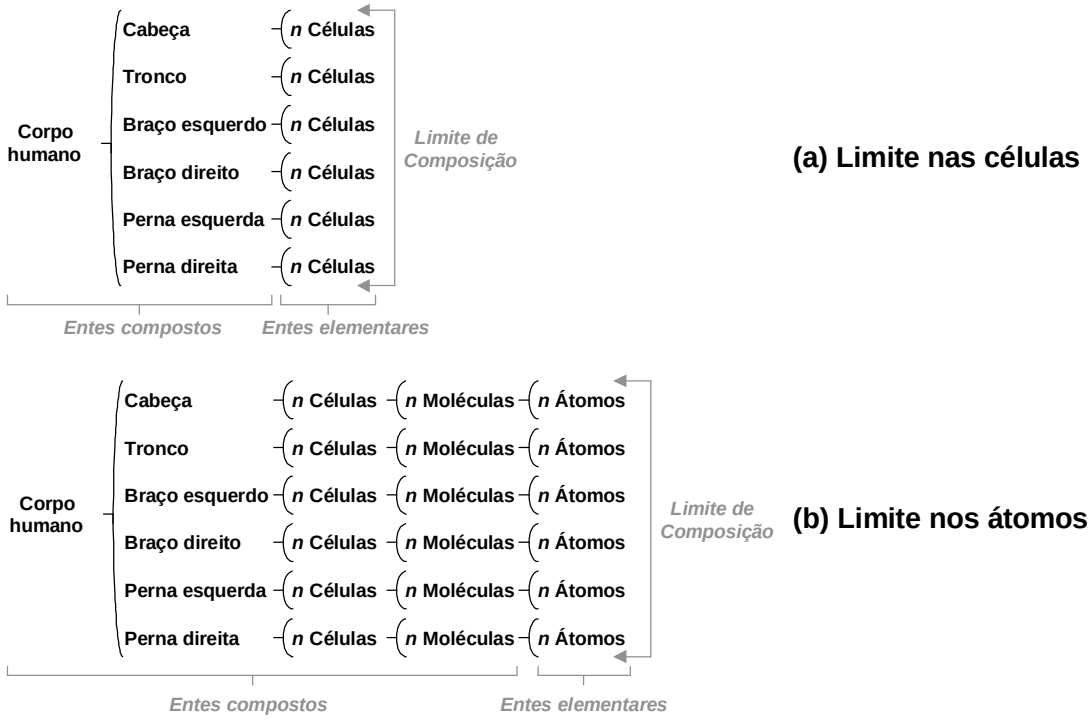


FIGURA A1.11 – Exemplo de limite de composição

1.6 Aplicação do Holo na automação residencial

A criação de uma nova proposta para solução de problemas traz consigo novas características que, normalmente, somente são descobertas durante sua aplicação em situações reais. Sendo assim, o uso de protótipos para validação de novas propostas é uma prática comum. No caso de novos paradigmas, torna-se interessante a avaliação de suas abstrações na solução de um problema real. Esta seção apresenta a aplicação do Holoparadigma na automação de residências.

Os avanços da microeletrônica e, em especial, o surgimento dos microprocessadores vem fazendo com que os aparelhos eletrônicos tornem-se cada vez mais sofisticados. Televisores e videocassetes microprocessados são uma realidade. Além disso, estudos envolvendo vários outros aparelhos eletrodomésticos estão sendo desenvolvidos (por exemplo, geladeiras e fogões). Como complemento, em breve o acesso à Internet não estará mais restrito a apenas computadores. Todos equipamentos que puderem obter vantagem da rede mundial de comunicação o farão. Sendo assim, não encontra-se muito distante a época em que as residências estarão repletas de equipamentos eletrônicos computadorizados, interligados e com acesso ao mundo externo através da Internet. Este tipo de residência pode ser percebido com uma arquitetura distribuída. A holomodelagem deste padrão de residência será descrita nos próximos parágrafos.

A figura A1.12 mostra um exemplo de uma residência composta de três dependências. Nela estão distribuídos seis equipamentos eletrônicos, os quais estão conectados por uma rede. Além disso, a família que vive na casa é composta de um casal e dois filhos. Conforme mostra a figura, a conexão da residência com o mundo externo encontra-se no microcomputador. Os equipamentos são todos microprocessados. Cada um deles atua como um nodo de uma arquitetura distribuída.

Em Holo, todas as existências são modeladas como entes. Portanto, através da holomodelagem pode-se considerar a residência como um ente composto. Por sua vez, os equipamentos eletrônicos e as pessoas da família podem ser considerados como entes elementares. Neste caso, os equipamentos e as pessoas compartilham uma história, ou seja, a história da casa. A figura A1.13 mostra essa situação. A história da casa cresceria na medida em que o tempo avançasse. Toda informação que fosse aprendida por um ente e fosse considerada relevante para todos os demais, seria colocada na história. Por exemplo, o número de pessoas que vivem na casa, suas idades e datas de nascimento.

Quando um novo equipamento eletrônico fosse adquirido e instalado, automaticamente passaria a fazer parte do ente composto e, portanto, passaria a ter acesso à história compartilhada. Sendo assim, a simples instalação de um equipamento na casa faria com que ele se tornasse um equipamento específico daquela família, conhecendo todos os fatos históricos armazenados na história da residência. Do ponto de vista lógico, todos os entes elementares tem acesso à história. No entanto, do ponto de vista físico ela deve estar armazenada em algum local da residência. Neste caso, o microcomputador poderia manter o armazenamento da história. Outra opção seria o uso de equipamentos específicos para gerenciamento da rede da residência e para armazenamento da história.

No exemplo da residência como ente, o hardware seria uma arquitetura distribuída composta de seis nodos dos quais cinco dedicados (eletrodomésticos) e um de uso genérico (microcomputador). O ambiente de execução suportaria o gerenciamento dos entes. Esse gerenciamento poderia envolver mobilidade, protocolos de comunicação, tolerância a falhas, escalonamento de tarefas, interface com o mundo externo, suporte à memória compartilhada distribuída, etc.

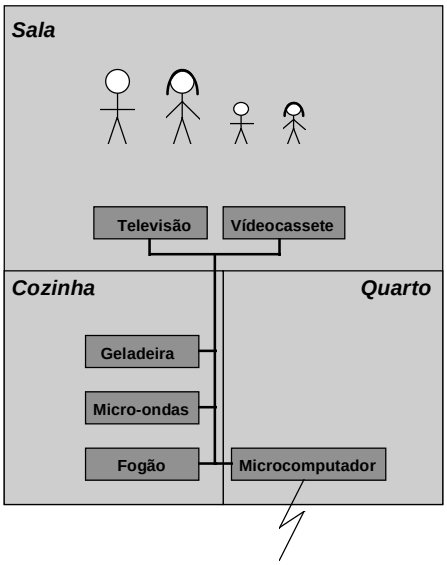


FIGURA A1.12 – Residência utilizada na holomodelagem

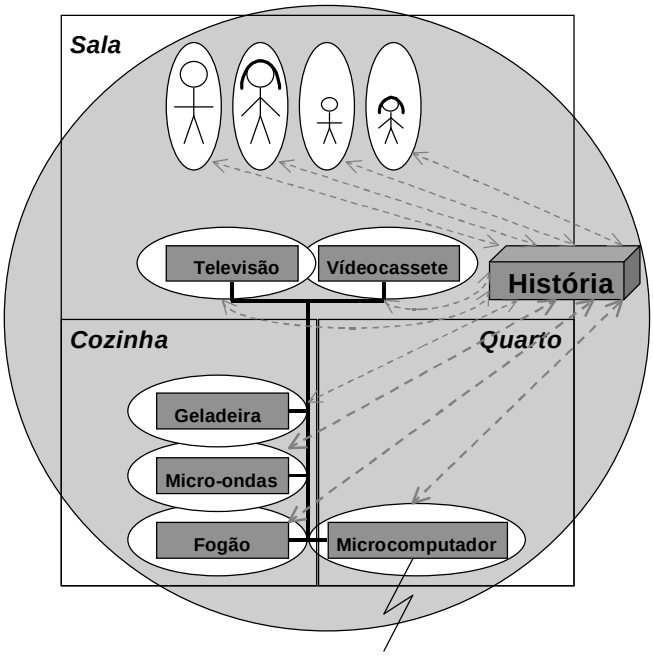


FIGURA A1.13 – Residência como um ente composto

A modelagem mostrada na figura A1.13 poderia ser expandida. A casa poderia ser modelada como um ente componente de um bairro. Por sua vez, o bairro poderia fazer parte de uma cidade. A composição poderia ser estendida para vários outros níveis, tais como: município, estado, região e país. Cada nível teria sua história. Por exemplo, o bairro teria uma história que seria compartilhada por todas as residências localizadas nos seus limites. Quando uma família fosse residir em um bairro passaria a compartilhar sua história. No momento em que o ente residência passasse a existir, teria acesso à história compartilhada. A figura A1.14 mostra os vários níveis de composição possíveis na holomodelagem de um país.

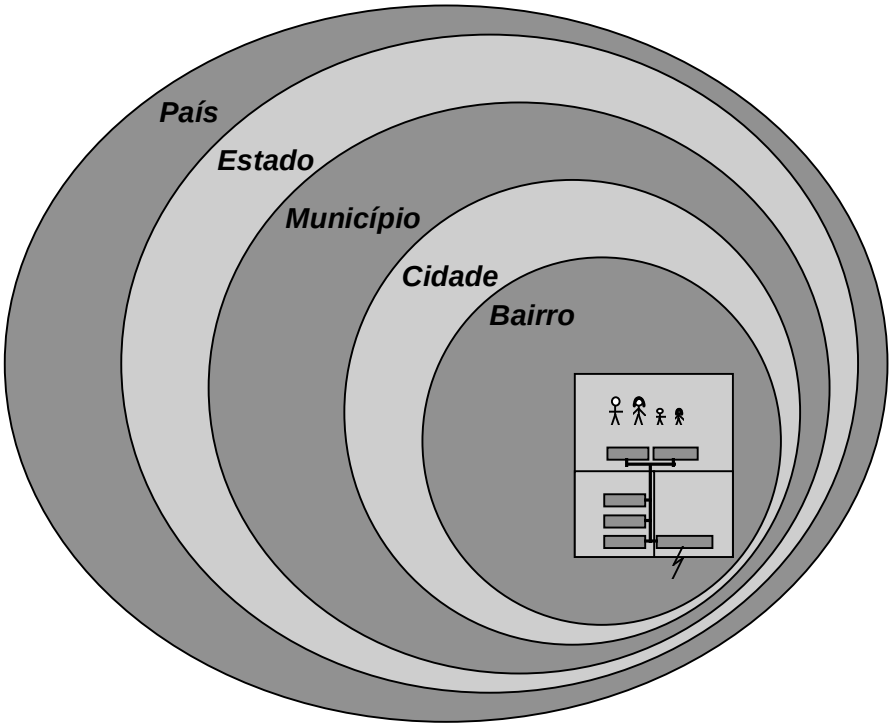


FIGURA A1.14 – Organização de um país através de Holo

1.7 Paradigma funcional na Hololinguagem

Nos últimos anos vem crescendo o uso da plataforma Java para implementação de linguagens declarativas. Na programação funcional existem diversos exemplos dessa abordagem. Por exemplo, vários compiladores para a linguagem Haskell [HUD 2001] usam Java como linguagem intermediária. Wakeling [WAK 97, WAK 98, WAK 98a] implementa um ambiente de execução para linguagens funcionais usando Java. Os programas Haskell são convertidos para classes Java que utilizam o ambiente. Du Bois [DUB 2001a] usa a mesma abordagem, convertendo programas em Haskell para componentes JavaBeans. Por sua vez, o compilador linguagem multiparadigma Curry [HAN 99] gera diretamente *byte code* Java.

A versão original da Hololinguagem (capítulo 4) suporta cinco tipos de ações no comportamento de entes (IAs, MIAs, LAs, MLAs e Mas). Du Bois, Barbosa e Geyer [DUB 2001] propuseram a introdução do paradigma funcional na linguagem, através da criação de dois novos tipos:

- **Ação Funcional (*Functional Actions* - FA):** ação formada por uma função em Haskell;
- **Ação Modular Funcional (*Modular Functional Actions* - MFA):** ação formada por um módulo que encapsula ações funcionais.

Os grafos ACG (figura 4.12) e AIG (figura 4.13) foram alterados para suporte as novas ações. A figura A1.15 mostra o novo ACG. As FAs são usadas na composição de MFAs e MAs. Por sua vez, a figura A1.16 apresenta o novo AIG. As FAs e MFAs foram colocadas na região declarativa e seguem os mesmos princípios de chamadas usados nas LAs e MLAs (veja seção 4.2). A figura A1.17 mostra o exemplo da figura 4.15 trocando a LA por uma FA em Haskell. Os seis passos de execução são os mesmos.

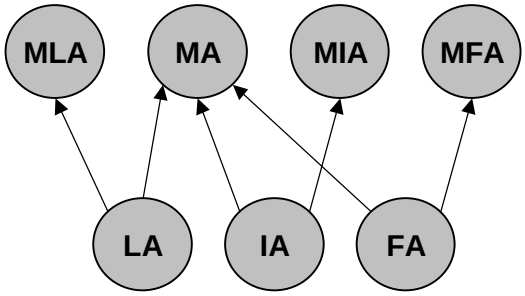


FIGURA A1.15 – ACG com ações funcionais

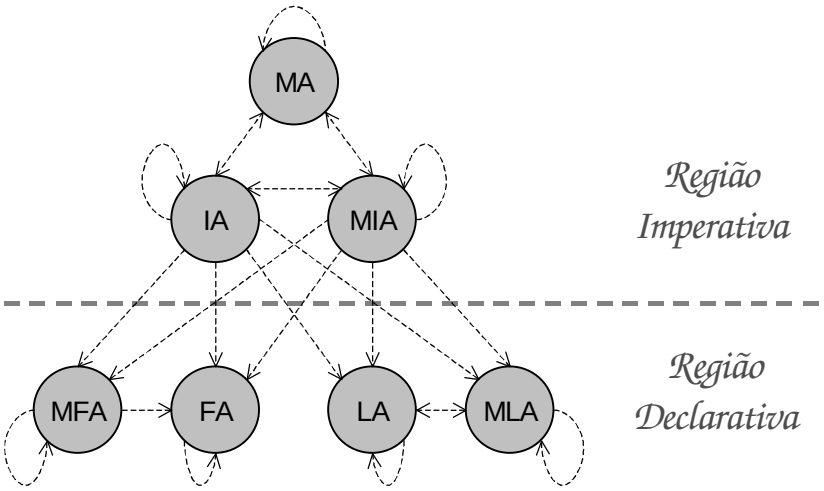


FIGURA A1.16 – AIG com ações funcionais

O suporte ao paradigma funcional na Hololinguagem utiliza a tecnologia desenvolvida no contexto de **Fun** [DUB 2001a]. Fun é uma pequena linguagem funcional compilada para Java. Normalmente, as linguagens funcionais são implementadas usando redução de grafos. O ambiente de execução utiliza uma máquina de redução e o compilador gera código para essa máquina. A execução de Fun é baseada na **G-Machine** [AUG 84]. Cada função é convertida para uma seqüência de instruções da máquina, as quais ao serem executadas criam uma instância de um corpo de função [JON 92]. A G-Machine implementada em Fun é simplesmente uma classe Java (GM class). Esta classe possui métodos estáticos que compõem as instruções.

Cada função de um programa Fun é compilada para uma classe Java. Esta classe contém as instruções (chamadas para métodos estáticos da classe GM) necessárias para instanciação da função na G-Machine. Por exemplo, a função `F x = id x` é convertida na seguinte classe:

```
class f extends Nsuperc{
   f(){
       narg=1;
       name = new String (“f”);
      }
   public void code(){
       GM.push(0);
       GM.pushglobal(new id());
       GM.mkap();
       GM.update(1);
       GM.pop(1);
      }
}
```

A classe abstrata `Nsuperc` suporta as funções definidas no programa compilado. O construtor da classe (no exemplo `f()`) sempre define dois valores:

- *`narg`*: número de argumentos para a função;
- *`name`*: define o nome da função (usado na depuração).

Todas as classes que implementam a classe `Nsuperc` possuem o método `code()`. Este método possui no seu corpo as instruções G-Machine que instanciam o corpo da função. A inclusão de Haskell em holoprogramas é baseado em dois aspectos:

- **Compilação**: a HoloJava (seção 5.1) traduzirá FAs e MFAs para classes contendo conjuntos de instruções G-Machine;
- **Execução**: o ambiente de execução possuirá suporte à G-Machine. A execução de classes oriundas de FAs ou MFAs utilizará esse suporte.

Entre as atividades futuras relacionadas com o suporte ao paradigma funcional na Hololinguagem merecem destaque:

- exploração de paralelismo durante a execução de FAs e MFAs: embora a linguagem Fun suporte a avaliação paralela usando `par` e `seq` [TRI 98], atualmente essa característica ainda não está sendo explorada na Hololinguagem;

- o AIG poderá ser adaptado para suporte a chamadas entre o paradigma em lógica e o paradigma funcional. Neste sentido, serão conduzidos estudos sobre a compatibilidade do código Java gerado nas conversões em Fun [DUB 2001a] e no Prolog Café [BAN 99, PRO 2001a].

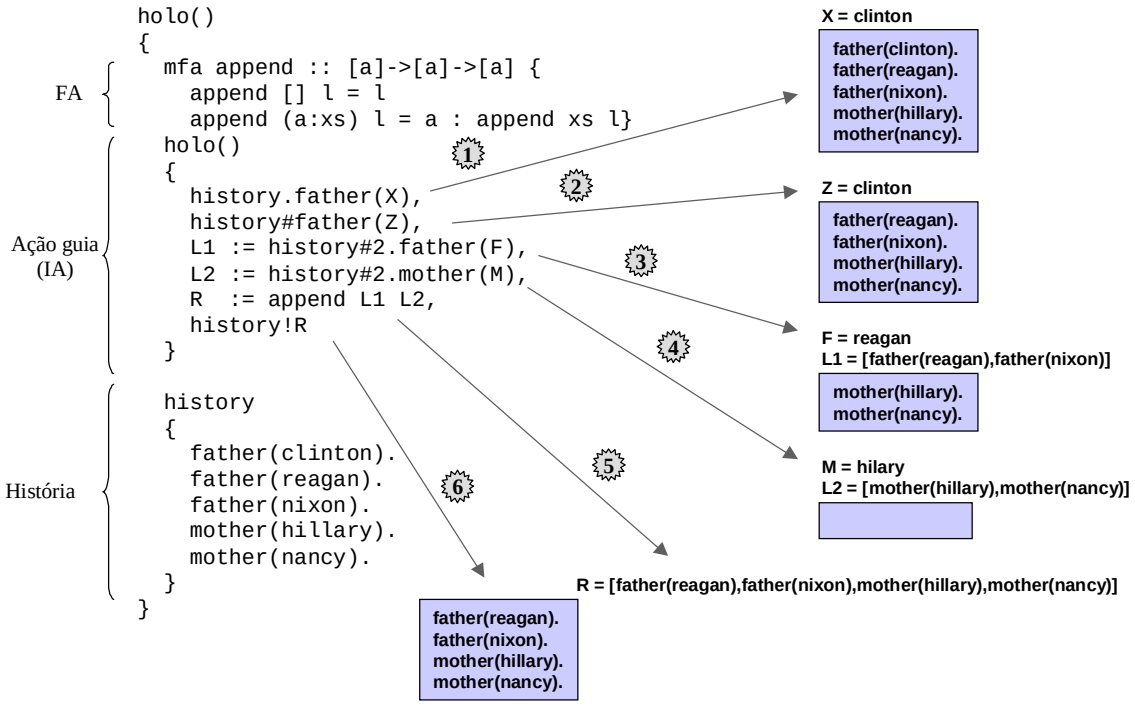


FIGURA A1.17– Exemplo de ação funcional em Haskell

Anexo 2 Definições de Termos usados no Holoparadigma

Fontes consultadas

❶ Novo Dicionário Aurélio da Língua Portuguesa. Editora Nova Fronteira, 1986.

❷ Michaelis - Moderno Dicionário da Língua Portuguesa. Editora Melhoramentos, 1998.

❸ Oxford Advanced Learner' s Dictionary. Oxford University Press, 1992.

❹ Language Master model LM-6000. Franklin Electronic Publishers Inc, USA, 1991. Words from Merriam-Webster.

Agregação

❶ **1.** Ação ou efeito de agregar(-se). **2.** Reunião em grupo; associação, aglomeração.

❷ **1.** Ação ou efeito de agregar. **2.** Aglomeração, associação, conjunto, reunião. **3.** propriedade que têm as moléculas de cada corpo de justapor-se ou reunir-se graças à força de coesão. **4.** Processo de formar grupos demográficos pelo acúmulo de indivíduos através de imigração ou de aumento natural. **5.** Grupo de população assim formado.

❸ Não consta.

❹ **Aggregation: 1.** a group, body, or mass composed of many distinct parts. **2.** the collecting of units or parts into a mass or whole.

Clone

❶ [Do grego *klón*, 'b roto'] Conjunto de indivíduos originários de outros por multiplicação assexual (divisão, enxertia, apomixia, etc). [Todos os membros de um clone têm o mesmo patrimônio genético].

❷ [Do grego *klón*] Conjunto da progênie, produzida assexualmente, de um indivíduo, quer naturalmente (como os produtos de fissão repetida de um protozoário), quer vegetativamente (como na propagação de determinada planta por gemação ou mudas através de muitas gerações).

❸ **Clone: 1.** (any of a) group of plants or organisms produced asexually from one ancestor. **2.** computer designed to copy the functions of another model.

❹ **Clone:** [Greek *klon* "twig"] **1.** the offspring produced asexually from an individual (as a plant increased by grafting). **2.** an individual grown from a single body cell or its parent and genetically identical to the parent. **3.** one that appears to be a copy of an original form.

Composto

❶ [Do latim *compositu*] **1.** Constituído por dois ou mais elementos. **2.** Substância ou corpo composto. **3.** Complexo de várias coisas combinadas.

❷ [Do latim *compositu*] **1.** Que é formado por dois ou mais elementos. **2.** Designativo do corpo resultante da combinação de vários elementos.

❸ **Composed:** made up or formed from.

❹ Não consta.

Distinção

❶ [Do latim *distinctione*] **1.** Ato ou efeito de distinguir(-se), diferença, separação. **2.** Caracteres, características, qualidades, pelos quais uma pessoa ou uma coisa difere de outra.

❷ [Do latim *distinctione*] **1.** Ato ou efeito de distinguir. **2.** Percepção da diferença entre pessoas ou coisas. **3.** Sinal ou qualidade por que uma coisa se diferencia de outra. **4.** Diferença, separação. **5.** Sinal exterior destinado a evitar a confusão entre pessoas ou coisas.

❸ **Distinction: 1.** Difference or contrast between one person / thing and another. **2.** Separation of things or people into different groups according to quality, grade, etc.

❹ **Distinction: 1.** the act of distinguishing a difference. **2.** Difference. **3.** a distinguishing quality or mark.

Distribuição

❶ [Do latim *distributione*] **1.** Ato de distribuir; repartição. **2.** Classificação, disposição.

❷ [Do latim *distributione*] **1.** Ato ou efeito de distribuir; repartição. **2.** Classificação. **3.** Disposição, ordenamento.

❸ **Distribution: 1.** (instance of) giving or being given to each several people. **2.** (instance of the) positioning or allocation of items, features, etc within an area.

❹ **Distribution: 1.** the act or process of distributing. **2.** the position, arrangement, or frequency of occurrence (as of the members of a group) over an area or throughout a space or unit fo time.

Elementar

❶ **1.** Relativo ou pertencente a elemento(s). **2.** De composição ou funcionamento simples; primário, rudimentar. **3.** Simples, fácil, claro. **4.** Que está na base, essência, origem; essencial, fundamental, básico.

❷ **1.** Que é da natureza do elemento ou que serve de elemento. **2.** Principal, fundamental.

❸ **Elementary: 1.** of or in the beginning stages (of a course of study). **2.** dealing with the simplest facts (of a subject); basic. **3.** easy to solver or answer.

❹ **Elementary:** simple, rudimentary.

Ente

❶ [Do latim *ente*.] **1.** Aquilo que existe; coisa, objeto, matéria, substância, ser. **2.** Aquilo que supomos existir. **3.** Tudo que é de maneira concreta, fática ou atual independentemente de, em qualquer nível, tornar-se objeto de reflexão.

❷ [Do latim *ente*] **1.** O que é, existe ou pode existir. **2.** Ser. **3.** Coisa, objeto, substância.

❸ **Being: 1.** Existence. **2.** One' s essence or nature, self**3.** Living creature.

❹ **Being: 1.** existence, life. **2.** the qualities or constitution of an existent thing. **3.** a living thing.

Fragmento

❶ [Do latim *fragmentu*] **1.** Cada um dos pedaços de uma coisa partida ou quebrada. **2.** Parte de um todo; pedaço, fração.

❷ [Do latim *fragmentu*] **1.** Partícula isolada do todo. **2.** Pequena fração. **3.** Cada uma das pequenas partes em que se dividiu um todo.

❸ **Fragment: 1.** small part or piece broken off. **2.** separate or incomplete part.

❹ **Fragment:** a part broken off, detached, or incomplete.

História

❶ [Do grego *historía*, pelo latim *historia*] **1.** Narração metódica dos fatos notáveis ocorridos na vida dos povos, em particular, e na vida da humanidade, em geral. **2.** Conjunto de conhecimentos adquiridos através da tradição e/ou por meio dos documentos, relativos à evolução, ao passado da humanidade. **3.** Ciência e método que permitem adquirir e transmitir aqueles conhecimentos. **4.** Estudo das origens e processos de uma arte, de uma ciência ou de um ramo do conhecimento. **5.** Narração de acontecimentos, de ações, em geral cronologicamente dispostos. **6.** Narração de fatos, acontecimentos ou particularidades relativas a um determinado assunto.

❷ [Do grego *história*] **1.** Narração ordenada, escrita, dos acontecimentos e atividades humanas ocorridas no passado. **2.** Ramo da ciência que se ocupa de registrar cronologicamente, apreciar e explicar os fatos do passado da humanidade em geral, e das diversas nações, países e localidades em particular. **3.** Os fatos do passado da humanidade registrados cronologicamente. **4.** Exposição de fatos, sucessos ou particularidades relativas a determinado objeto digno de atenção pública.

❸ **History: 1.** Study of past events, especially the political, social and economic development of a country, a continent or the world. **2.** Past events, especially when considered as a whole. **3.** Systematic description of past events. **4.** Series of past events or experiences connected with an object, a person or a place.

❹ **History:** [Latin *historia*, from Greek, "inquiry, history", from *histor*, *istor* "knowing, learned"] **1.** an account of events or facts pertinent to a situation. **2.** a chronological record of significant events often with an explanation of their causes. **3.** a branch of knowledge that records and explains past events. **4.** events that form the subject matter of history.

Holismo

❶ [De hol(o)- + -ismo.] Tendência, que se supõe seja própria do Universo, a sintetizar unidades em totalidades organizadas.

❷ [De holo+ismo] **1.** Doutrina que considera o organismo vivo como um todo indecomponível. **2.** Compreensão da realidade em totalidades integradas onde cada elemento de um campo considerado reflete e contém todas as dimensões do campo, conforme a indicação de um holograma, evidenciando que a parte está no todo, assim como o todo está na parte, numa inter-relação constante, dinâmica e paradoxal.

❸ Não consta.

❹ **Holism: 1.** a theory that the universe and especially nature should be viewed as interacting wholes rather than as distinct parts.

Hol(o)

❶ [Do grego *hólos, hóle, hólon*.] Elemento de composição = Inteiro, completo.

❷ [Do grego *holós*] Elemento de composição = Introduz a idéia de totalidade.

❸ Não consta.

❹ Não consta.

Implícito

❶ [Do latim *implicitu*] **1.** Que está envolvido, mas não de modo claro; tácito, subentendido.

❷ [Do latim *implicitu*] **1.** Que está envolvido mas não expresso claramente; tácito. **2.** Não expresso por palavras; subentendido.

❸ **Implicit:** Implied, but not expressed directly; not explicit.

❹ **Implicit:** Understood though not directly stated or expressed.

Mobilidade

❶ [Do latim *mobilitate*] **1.** Qualidade ou propriedade do que é móvel ou obedece às leis do movimento. **2.** Facilidade de mover-se ou ser movido. **3.** Facilidade com que se passa de um estado para outro; inconstância, volubilidade. **4.** Facilidade de modificar-se ou variar.

❷ [Do latim *mobilitate*] **1.** Propriedade do que é móvel ou do que obedece às leis do movimento. **2.** Deslocamento de indivíduos, grupos ou elementos culturais no espaço social. **3.** Falta de estabilidade, de firmeza; inconstância.

❸ **Mobility:** being mobile.

❹ Não consta.

Paradigma

❶ [Do grego *parádeigma*, pelo latim *paradigma*.] Modelo, padrão, estalão.

❷ [Do grego *parádeigma*] **1.** Modelo, padrão, protótipo. **2.** Conjunto de unidades suscetíveis que aparecem num mesmo contexto, sendo, portanto, comutáveis e mutuamente exclusivas. No paradigma, as unidades têm, pelo menos, um traço em comum (a forma, o valor ou ambos) que as relaciona, formando conjuntos abertos ou fechados, segundo a natureza das unidades.

❸ **Paradigm: 1.** Set of all the different forms of a word. **2.** Type of, pattern, model.

❹ **Paradigm: 1.** model, pattern.

Paralelismo

❶ **1.** Posição de linhas ou superfícies paralelas. **2.** Correspondência ou simetria entre duas ou mais coisas, comparável ao paralelismo das retas. **3.** Correspondência de idéias ou opiniões.

❷ **1.** Estado do que é paralelo. **2.** Correspondência ou simetria entre duas coisas.

❸ **Parallelism:** state of being parallel; similarity.

❹ Não consta.

Semântica

❶ [Do grego *semantiké*, isto é, *téchne semantiké*, ' a arte da significação'**1.**] Estudo das mudanças ou translações sofridas, no tempo e no espaço, pela significação das palavras; semasiologia, semantologia, semiótica. **2.** O estudo da relação de significação dos signos e da representação do sentido dos enunciados.

❷ [Do grego *semantiké*, de *sema*] **1.** Estudo da evolução do sentido das palavras através do tempo e do espaço; semiótica, semasiologia, sematologia.

❸ **Semantic: 1.** of the meaning of words.

❹ **Semantic: 1.** of or relating to meaning.

Vínculo

❶ [Do latim *vinculu*] **1.** Tudo o que ata, liga ou aperta. **2.** Ligação moral **3.** Relação, subordinação.

❷ [Do latim *vinculu*] **1.** Tudo o que ata, liga ou aperta. **2.** Ligação moral. **3.** Nexo jurídico que estabelece subordinação de uma coisa a outra. **4.** Relação, subordinação.

❸ Não consta.

❹ Vinculum: 1. a unifying bond. 2. a straight horizontal mark placed over two or more members of a mathematical expression as a symbol of grouping.

Anexo 3 Exemplos de Holoprogramas

Este anexo contém dez exemplos de holoprogramas. Os programas são os seguintes:

- anexo 3.1: *datamining.holo*
- anexo 3.2: *performance.holo*
- anexo 3.3: *semaphores.holo*
- anexo 3.4: *philosofers.holo*
- anexo 3.5: *buffers.holo*
- anexo 3.6: *travel.holo*
- anexo 3.7: *hanoi.holo*
- anexo 3.8: *fibo.holo*
- anexo 3.9: *lists.holo*
- anexo 3.10: *family.holo*

Anexo 3.1 Programa *datamining.holo*

```
/*************************************************************
   NOME: datamining.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Simulacao de datamining usando uma MLA (Fibonacci).
   ENTRADA: Nenhuma.
   SAIDA: Mensagens na tela mostrado os resultados da simulacao.
   OBSERVACOES:
     1) Cria tres minas e um mineiro;
     2) O mineiro entra em cada mina e obtem um valor;
     3) Este valor e utilizado para calcular Fibonacci;
     4) O calculo de Fibonacci utiliza uma MLA.
*************************************************************/

//********************  ENTE PRINCIPAL ********************

holo()     // Ente principal.
{
  holo()   // Acao guia.
  {
    writeln('HOLO: Vou criar tres minas e um mineiro'),
    clone(mine(1),mine_d1),  // Cria a primeira mina. O parametro identifica a mina.
    clone(mine(2),mine_d2),  // Cria a segunda mina.
    clone(mine(3),mine_d3),  // Cria a terceira mina.
    clone(miner,miner_d),    // Cria o mineiro.
    for X := 1 to 3 do       // Aguarda pelos resultados da mineracao.
    {
      history#list(#Ident,#Num,#Fibo),   // Obtem o resultado de uma mineracao.
      writeln('HOLO: Terminou a mineracao da mina:',Ident),
      writeln('HOLO: Fibonacci de ',Num,' e ',Fibo)
    }
    writeln('HOLO: Terminou a mineracao')
  }
}

//********************  ENTE MINA *******************
 mine()
{
  mine(Ident)
  {
    writeln('MINA ',Ident,': Fui criada')
  }

  history         // A historia da mina de cada mina possui
  {               // o mesmo conteudo. O primeiro numero
    list(1,2).    // identifica a mina. O segundo e o numero
    list(2,4).    // usado para o calculo do Fibonacci.
    list(3,6).
  }
}
```

```
//********************  ENTE MINEIRO *******************

miner()
{
  miner()
  {
    writeln('MINEIRO: Inicio da mineracao.'),
    move(self,mine_d1),                       // Passo 1  - Entra na mina 1
    mining(1,Num1,Res1),                      // Passo 2  - Minera a mina 1
    move(self,out),                           // Passo 3  - Sai da mina 1
    out(history)!list(1,Num1,Res1),           // Passo 4  - Salva o resultado 1
    move(self,mine_d2),                       // Passo 5  - Entra na mina 2
    mining(2,Num2,Res2),                      // Passo 6  - Minera a mina 2
    move(self,out),                           // Passo 7  - Sai da mina 2
    out(history)!list(2,Num2,Res2),           // Passo 8  - Salva o resultado 2
    move(self,mine_d3),                       // Passo 9  - Entra na mina 3
    mining(3,Num3,Res3),                      // Passo 10 - Minera a mina 3
    move(self,out),                           // Passo 11 - Sai da mina 3
    out(history)!list(3,Num3,Res3),           // Passo 12 - Salva o resultado 3
    writeln('MINEIRO: Fim da mineracao.')
  }

  mining(Ident,Num,Result)           // IA que realiza a mineracao.
  {
    out(history)#list(Ident,#Num),   // Minera a historia externa.
    fib(Num,#Result)                 // Chama a MLA para determinar Fibonacci.
  }

  fib/2()          // Acao Modular Logica (MLA) que calcula Fibonacci.
  {
    fib(1,1).
    fib(2,1).
    fib(M,N) :-
      M > 2,
      M1 is M-1,
      M2 is M-2,
      fib(M1,N1),
      fib(M2,N2),
      N is N1+N2.
  }
}
```

Anexo 3.2 Programa *performance.holo*

```
/*********************************************************************
   NOME: performace.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Simulacao de datamining com numero de operacoes variaveis.
   ENTRADA: Atraves do parametro:
              Oper = Numero de operacoes de mineracao
   SAIDA: Mensagens na tela mostrado os resultados da simulacao e,
          em especial, a medida de desempenho.
   OBSERVACOES:
     1) Cria tres minas e um mineiro;
     2) O mineiro entra em cada mina e minera;
     3) A mineracao consiste em uma simples leitura na historia.
*********************************************************************/


//********************  ENTE PRINCIPAL ********************

holo()
{
  holo(Oper)
  {
    writeln('HOLO: Vou criar tres minas e um mineiro'),
    clone(mine(1),mine_d1),     // Cria as tres minas.
    clone(mine(2),mine_d2),
    clone(mine(3),mine_d3),
    clone(miner(Oper),miner_d), // Cria o mineiro.
    time(Inicio),
    for X := 1 to 3 do
    {
      history#list(#Data),      // Aguarda a escrita dos resultados da mineracao.
      writeln('HOLO: Terminou a mineracao na mina:',Data)
    }
    time(Fim),
    writeln('HOLO: Terminou a mineracao. Tempo = ',(Fim-Inicio),' milisegundos.')
  }
}


//********************  ENTE MINA ********************

mine()
{
  mine(Ident)
  {
    writeln('MINA ',Ident,': Fui criada')
  }

  history
  {
    list(1,2).  // Dado que sera minerado.
  }
}
```

```
//********************  ENTE MINEIRO *******************

miner()
{
  miner(Oper)
  {
    writeln('MINEIRO: Inicio da mineracao.'),
    move(self,mine_d1),         // Passo 1  - Entra na mina 1
    mining(1,Oper),             // Passo 2  - Minera na mina 1
    move(self,out),             // Passo 3  - Sai da mina 1
    out(history)!list(1),       // Passo 4  - Escreve o resultado 1
    move(self,mine_d2),         // Passo 5  - Entra na mina 2
    mining(2,Oper),             // Passo 6  - Minera na mina 2
    move(self,out),             // Passo 7  - Sai da mina 2
    out(history)!list(2),       // Passo 8  - Escreve o resultado 2
    move(self,mine_d3),         // Passo 9  - Entra na mina 3
    mining(3,Oper),             // Passo 10 - Minera a mina 3
    move(self,out),             // Passo 11 - Sai da mina 3
    out(history)!list(3),       // Passo 12 - Escreve o resultado 3
    writeln('MINEIRO: Termino da mineracao.')
  }

  mining(Ident,Oper)
  {
    writeln('MINEIRO: Estou minerando ',Oper,' vezes na mina ',Ident),
    for Cont := 1 to Oper do
    {
      out(history).list(1,2)      // Realiza a mineracao.
    }
  }
}
```

Anexo 3.3 Programa *semaphores.holo*

```
/*********************************************************************
   NOME: semaphores.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Gerenciamento de semaforos.
   ENTRADA: Atraves de um parametro:
              Quantos = Indica quantos entes vao concorrer pelo recurso
   SAIDA: Mensagens na tela mostrado a evolucao do acesso ao recurso.
   OBSERVACOES:
     1) O entes competem por um recurso
     2) Cada ente acessa o recurso tres vezes;
     3) Este programa mostra como uma acao sendo chamada sem argumentos.
*********************************************************************/


//*******************  ENTE PRINCIPAL *******************

holo()
{
  holo(Quantos)
  {
    writeln('INICIO DO PROGRAMA SEMAFOROS: ',Quantos,' entes competirao.'),
    writeln('CADA ENTE ACESSA O RECURSO TRES VEZES.'),
    for X := 1 to Quantos do     // Cria um grupo de entes que competirao.
    {
      clone(competidor(X),null)  // O nome dos entes criado nao interessa (null).
    }
    for X := 1 to Quantos do     // Aguarda o aviso do termino da competicao.
    {
      history#saiu               // Cada ente indica que saiu.
    }
    writeln('FIM DO PROGRAMA SEMAFOROS')
  }

  p()
  {
    history#token   // Operador "p". Tenta retirar token. Se conseguir, entra na
  }                 // secao critica. Se nao, aguarda ate alguem colocar um token.

  v()
  {
    history!token   // Operador "v". Coloca "token" avisando que saiu da secao critica.
  }

  history
  {
   token.          // Inicializa a historia com um token para controlar secao critica.
  }                // Um token na historia, autoriza a entrada na secao critica.
}


//*******************  ENTE COMPETIDOR *******************

competidor()
{
  competidor(Name)
  {
    for C := 1 to 3 do  // O ente vai entrar e sair da secao critica tres vezes.
    {
      writeln('ENTE ',Name,' - ',C,': Tentando entrar...'),
      out(behavior).p(),      // Chamada de uma acao sem argumentos.
      writeln('ENTE ',Name,' - ',C,': Dentro.'),
      for I := 1 to 10000 do
      {}                      // Dentro da SC
      out(behavior).v(),      // Chamada de uma acao sem argumentos.
      writeln('ENTE ',Name,' - ',C,': Fora.')
    }
    out(history)!saiu         // Avisa que ja entrou e saiu tres vezes.
  }
}
```

Anexo 3.4 Programa *philosofers.holo*

```
/*********************************************************
   NOME: philosophers.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Classico problema do Jantar de Filosofos.
   ENTRADA: Nenhuma.
   SAIDA: Mensagens na tela mostrado a evolucao do jantar.
   OBSERVACOES:
     1) Cria cinco filosofos. Cada filosofo e um ente;
     2) Cada filosofo come cinco vezes.
*********************************************************/


//*******************  ENTE PRINCIPAL *******************

holo()
{
  holo()
  {
    writeln('CINCO FILOSOFOS VAO JANTAR. CADA FILOSOFO COME CINCO VEZES.'),
    writeln('O JANTAR ESTA INICIANDO.....'),
    for X := 1 to 5 do   // Cria os cinco filosofos.
    {
      clone(filosofo(X),null)
    }
    for X := 1 to 5 do   // Aguarda o jantar de cada filosofo.
    {
      history#end
    }
    writeln('O JANTAR ACABOU.')
  }

  history          // A historia guarda os talheres e os assentos.
  {                // Utiliza um "token" para controle do jantar.
    chopstick(1).
    chopstick(2).
    chopstick(3).
    chopstick(4).
    chopstick(5).
    ticket. ticket. ticket. ticket.
    seat(1,2). seat(2,3). seat(3,4). seat(4,5). seat(5,1).
  }
}


//*******************  ENTE FILOSOFO *******************

filosofo()
{
  filosofo(Ident)
  {
    writeln('Fisolofo esta jantando: ',Ident),
    for Cont := 1 to 5 do
    {
      out(history)#ticket,         // Retira um token.
      out(history)#seat(#F1,#F2),  // Pega um assento.
      out(history)#chopstick(#F1), // Pega os respectivos talheres.
      out(history)#chopstick(#F2),
      for Atraso := 1 to 1000 do   // Aguarda algum tempo.
      {}
      out(history)!chopstick(F2),  // Devolve os talheres.
      out(history)!chopstick(F1),
      out(history)!seat(F1,F2),    // Devolve o assento.
      out(history)!ticket,         // Libera para outro jantar.
      writeln('Comendo ',Ident,' - ',Cont)
    }
    out(history)!end  //Avisa que terminou de jantar.
  }
}
```

Anexo 3.5 Programa *buffers.holo*

```
/*******************************************************************************
   NOME: buffes.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Gerenciamento de buffers.
   ENTRADA: Via parametros abaixo:
              Prod = Numero de produtores
              Cons = Numero de consumidores
              Tam = Tamanho do buffer
   SAIDA: Mensagens na tela mostrado a interacao entre produtores e consumidores.
   OBSERVACOES:
     1) Cada produtor produz cinco produtos;
     2) Cada consumidor consome cinco produtos;
     3) A historia do ente "holo" armazena o buffer;
     4) Neste exemplo, o nome do buffer e constante, ou seja, "buffer";
     5) O sincronismo entre produtores e consumidores e realizado pela historia;
     6) O programa pode bloquear se o numero de producoes superar o tamanho do buffer
        ou se um consumidor tentar consumir um dado que nao sera colocado;
     7) Nao existe historia inicial em "holo". Por outro lado, a acao "init" coloca
        fatos na historia. Este e um caso de historia que inicia vazia, mas mesmo assim
        e utilizada no programa.
*******************************************************************************/

//*******************  ENTE PRINCIPAL *******************

holo()                      // Ente principal.
{
  holo(Prod,Cons,Tam)       // IA onde comeca a execucao.
  {
    init(buffer,Tam),       // Inicializa o buffer indicando o nome e o tamanho.
    for X := 1 to Prod do   // Cria produtores.
    {
      clone(produtor(X,buffer),null)   // Cria um produtor.
    }                                  // O nome do clone nao e relevante (null).
    for X := 1 to Cons do              // Cria os consumidores.
    {
      clone(consumidor(X,buffer),null) //Cria um consumidor.
    }
  }                                    // Fim da acao guia principal.

  init(Name,Tam)              // IA que inicializa o buffer.
  {
    for Cont := 1 to Tam do   // Preenche o buffer com o controlador de tamanho.
    {                         // O tamanho e controlado atraves de "tokens" na historia.
      history!Name            // Poderiam haver varios buffers usando diferentes tokens.
    }
  }

  put(Name,Data)              // Coloca no buffer. Recebe o nome do buffer e o dado.
  {
    history#Name,             // Verifica se esta cheio (sem tokens). Se esta, bloquea
    history!data(Name,Data)   // ate um token surgir. Logo apos, coloca o dado no buffer.
  }

  get(Name,Data)              // Retira um dado do buffer.
  {
    history#data(Name,Data), // Tenta retirar um dado do buffer. Se nao existe, bloquea.
    history!Name             // Quando um dado for colocado, desbloquea e coloca token.
  }
}
```

```
//********************  ENTE PRODUTOR ********************

produtor()
{
  produtor(Eu,Buffer)      // Recebe identificacao e o nome do buffer.
  {
    for Cont := 1 to 5 do  // Coloca cinco produtos na buffer.
    {
      out(behavior).put(Buffer,Cont), // Coloca dado no buffer (historia acima).
      writeln('PRODUTOR   ',Eu,' - ',Cont,': Coloquei o dado.'),
      for Atraso := 1 to 1000 do      // Consome tempo antes de produzir o proximo.
      {}
    }
  }
}


//********************  ENTE CONSUMIDOR ********************

consumidor()
{
  consumidor(Eu,Buffer)     // Recebe a identificacao e o buffer.
  {
    for Cont := 1 to 5 do   // Consome cinco produtos.
    {
      out(behavior).get(Buffer,Cont),  // Obtem um dado no buffer (historia acima).
      writeln('CONSUMIDOR ',Eu,' - ',Cont,': Recuperei o dado.'),
      for Atraso := 1 to 1000 do      // Aguarda tempo antes de consumir o proximo.
      {}
    }
  }
}
```

Anexo 3.6 Programa *travel.holo*

```
/*******************************************************************************
   NOME: travel.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Demonstra o uso de mobilidade, acoes predefinidas obtendo informacoes
           sobre entes e o uso da historia para sincronismo durante a mobilidade.
           Sao criados dois estados. O RS possui tres municipios e SC possui dois.
           O viajante visita todos os municipios.
   ENTRADA: Nenhuma
   SAIDA: Mensagens na tela contando a historia da viagem.
   OBSERVACOES:
     1) Sao utilizadas as acoes predefinidas:
        -> whoami(X): Retorna o nome do ente dinamico que executa a acao;
        -> whereami(X): Retorna o ente composto no qual esta localizado o ente que
                        executa a acao.
     2) O ente estado usa um IF para implementar dois comportamentos diferentes para
        RS e SC;
     3) A historia dos entes e utilizada para varios sincronismos;
     4) O viajante somente e criado quando ambos os estados estao prontos;
     5) Todas as mensagens aparecem na ordem correta, contando a viagem de forma
        organizada. Esta organizacao exige o sincronismo da threads.
*******************************************************************************/

//********************  ENTE PRINCIPAL ********************
holo()
{
  holo()
  {
    whoami(Eu),                // Retorna "holo".
    writeln(Eu,': Vou criar dois estados.'),
    clone(estado,rs),
    clone(estado,sc),
    for X := 1 to 2 do
    {
      history#estado_criado    // Aguarda a criacao dos estados.
    }
    writeln(Eu,': Vou criar um viajante e aguardar seu retorno.'),
    clone(viajante_e,viajante),
    history#voltou,            // Aguarda o aviso de retorno do viajante.
    writeln(Eu,': O viajante voltou.')
  }
}

//******************** ENTE ESTADO ******************
estado()
{
  estado()
  {
    whoami(Eu),         // Retorna "rs" ou "sc".
    whereami(Pos),      // Retorna "holo"
    writeln(Eu,': Estado foi criado. Eu faco parte de ',Pos,'.'),
    history!estado(Eu), // Cada estado coloca seu nome na historia.
    if (Eu == rs) then  // Aqui sao implementados dois comportamentos
    {
      clone(municipio,piratini),        // RS cria tres municipios.
      clone(municipio,pelotas),
      clone(municipio,portoalegre)
    }
    else
    {
      clone(municipio,florianopolis),   // SC cria dois municipios.
      clone(municipio,camboriu)
    }
    howmany(X),         // Descobre quantos municipios foram criados
    for Y := 1 to X do  // RS avisa tres vezes e SC apenas duas.
    {
      history#municipio_criado
    }
    out(history)!estado_criado  // Avisa sobre a criacao do estado.
  }
}
```

```
//********************  ENTE CIDADE *******************

municipio()
{
  municipio()
  {
    whoami(Eu),        // Retorna o nome do municipio.
    whereami(Pos),     // Retorna "rs" ou "sc".
    writeln(Eu,': Municipio foi criado. Faco parte de ',Pos,'.'),
    out(history)!municipio_criado, // Avisa sobre a criacao do municipio.
    history#cheguei,  // Aguarda a chegada do viajante.
    history!bemvindo, // Boas vindas.
    writeln(Eu,': O municipio de ',Eu,' agradece pela visita.')
  }
}


//********************  ENTE VIAJANTE *******************

viajante_e()
{
  viajante_e()
  {
    whoami(Eu),
    writeln(Eu,': Fui criado. Quero viajar!!'),
    move(self,rs),
    out(history).estado(Whereami), // Viajante descobre onde esta usando historia.
    writeln(Eu,': Estou no ',Whereami,'.'),  // Esta no Rio Grande do Sul.
    move(self,piratini),
    whereami(W1),
    writeln(Eu,': Estou em ',W1,'. Primeira capital do RS.'), // Esta em Piratini.
    out(history)!cheguei,                    // Avisa Piratini que chegou.
    out(history)#bemvindo,                   // Boas vindas de Piratini.
    move(self,out),
    move(self,pelotas),
    whereami(W2),
    writeln(Eu,': Estou em ',W2,'. Cidade dos doces.'),   // Esta em Pelotas.
    out(history)!cheguei,                    // Avisa Pelotas que chegou.
    out(history)#bemvindo,                   // Boas vindas de Pelotas.
    move(self,out),
    move(self,portoalegre),
    whereami(W3),
    writeln(Eu,': Estou em ',W3,'. Visitando a capital.'), // Esta em Porto Alegre.
    out(history)!cheguei,                    // Avisa Porto Alegre que chegou.
    out(history)#bemvindo,                   // Boas vindas de Porto Alegre.
    move(self,out),
    move(self,out),
    whereami(W4),
    writeln(Eu,': Voltei para ',W4,'.'),     // Esta em Holo.
    move(self,sc),
    out(history).estado(Where),      // Descobre onde esta usando a historia.
    writeln(Eu,': Estou em ',Where,'.'),     // Esta em Santa Catarina.
    move(self,florianopolis),
    whereami(W5),
    writeln(Eu,': Estou em ',W5,'. Praias....'), // Esta em Florianopolis.
    out(history)!cheguei,                        // Avisa Florianopolis que chegou.
    out(history)#bemvindo,                       // Boas vindas de Pelotas.
    move(self,out),
    move(self,camboriu),
    whereami(W6),
    writeln(Eu,': Estou em ',W6,'. Mais praias...'), // Esta em Camboriu.
    out(history)!cheguei,                    // Avisa Camboriu que chegou.
    out(history)#bemvindo,                   // Boas vindas de Camboriu.
    move(self,out),                          // Mais praias........
    move(self,out),
    whereami(W7),
    writeln(Eu,': Voltei para ',W7,'. Acabei a viagem.'),     // Voltou para Holo.
    out(history)!voltou                      // Informa Holo que a viagem acabou.
  }
}
```

Anexo 3.7 Programa *hanoi.holo*

```
/*****************************************************************
   NOME: hanoi.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Implementa o resultado para o jogo Torre de Hanoi.
   ENTRADA: Atraves do parametro:
              Pecas = Numero de pecas a ser considerado.
   SAIDA: Mensagem com resultado e analise de desempenho.
   OBSERVACOES:
     1) Programa simples para Torre de Hanoi usando uma MLA;
     2) Permite a analise de desempenho do codigo gerado pelo
        Prolog Cafe;
     3) Este programa possui apenas um ente.
*****************************************************************/

holo()
{
  holo(Pecas)
  {
    writeln('PROGRAMA TORRE DE HANOI'),
    time(Inicio),
    hanoi(Pecas,p1,p2,p3,Result),
    time(Fim),
    writeln('O resultado :',Result,'.'),
    writeln('Tempo para o calculo: ',(Fim-Inicio),' milisegundos')
  }

  hanoi/5()
  {
      hanoi(1,A,B,C,[mv(A,C)]).
      hanoi(N,A,B,C,M) :-
           N > 1, N1 is N - 1,
           hanoi(N1,A,C,B,M1),
           hanoi(N1,B,A,C,M2),
           append(M1,[mv(A,C)],T),
           append(T,M2,M).


      append([],L,L).
      append([H|L],L1,[H|R]) :-
            append(L,L1,R).
  }
}
```

Anexo 3.8 Programa *fibo.holo*

```
/*****************************************************************
   NOME: fibo.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Calculo do numero de Fibonacci com medida de desempenho
   ENTRADA: Atraves do parametro:
              Num = Numero que sera utilizado no calculo
   SAIDA: Mensagem com resultado e analise de desempenho.
   OBSERVACOES:
     1) Programa simples para calcular Fibonacci usando uma MLA;
     2) Permite a analise de desempenho do codigo gerado pelo
        Prolog Cafe;
     3) Este programa possui apenas um ente.
*****************************************************************/

holo()
{
  holo(Num)
  {
    writeln('PROGRAMA CALCULO DE FIBONACCI'),
    time(Inicio),
    fib(Num,Result),
    time(Fim),
    writeln('O Fibonacci de ',Num,' e ',Result,'.'),
    writeln('Tempo para o calculo: ',(Fim-Inicio),' milisegundos')
  }

  fib/2()
  {
    fib(1,1).
    fib(2,1).
    fib(M,N) :-
      M > 2,
      M1 is M-1,
      M2 is M-2,
      fib(M1,N1),
      fib(M2,N2),
      N is N1+N2.
  }
}
```

Anexo 3.9 Programa *lists.holo*

```
/*****************************************************************
   NOME: lists.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Testa o potencial para manipulacao de listas em uma MLA.
   ENTRADA: Atraves do parametro:
              Tam = Tamanho da lista que sera manipulada
   SAIDA: Mensagens para acompanhamento dos resultados.
   OBSERVACOES:
     1) MLAs suportam tratamento de listas;
     2) HoloJava 1.0 nao possui suporte para listas;
     3) MLAs devolvem listas para as IAs. Estas listas sao tratadas
        como simbolos;
     4) Este programa possui apenas um ente.
*****************************************************************/

holo()
{
  holo(Tam)
  {
    writeln('PROGRAMA PARA TRATAMENTO DE LISTAS'),
    writeln('A lista tera tamanho ',Tam),
    listas(Tam,Lista_inv,Lista_certa,Tamanho),
    writeln('A lista gerada e    : ',Lista_inv),
    writeln('A lista invertida e : ',Lista_certa),
    writeln('Confirmado, o tamanho da lista e: ',Tamanho)
  }

/*
   T = Tamanho da lista (entrada)
   L = Lista gerada invertida (saida)
   Z = Lista na ordem certa
   K = Tamanho da lista calculada por nrev
*/
  listas/4()
  {
      listas(T,L,Z,K) :-
          gera_lista(T,L),
          nrev(L,Z),
          size(Z,K).

      nrev([],[]).
      nrev([H|L],R) :-
          nrev(L,R1),
          append(R1,[H],R).

      append([],L,L).
      append([H|L],L1,[H|R]) :-
          append(L,L1,R).

      gera_lista(0,[]).
      gera_lista(N,[Ca|Co]) :-
          Ca is N - 1,
          gera_lista(Ca,Co).


      size([],0).
      size([X|Y],T) :-
          size(Y,Z),
          T is Z + 1.
  }
}
```

Anexo 3.10 Programa *family.holo*

```
/*****************************************************************
   NOME: family.holo
   DATA: 14/12/2001
   RESPONSAVEL: Jorge Barbosa (barbosa@inf.ufrgs.br)
   FUNCAO: Gerenciador de informacoes familiares :-)
   ENTRADA: Atraves do comando read:
              Avo = Nome do avo
   SAIDA: Mensagens na tela mostrado o nome de um neto.
   OBSERVACOES:
     1) Programa simples para demonstracao de uma base Prolog;
     2) Utiliza a acao predefinida READ para leitura de teclado;
     3) Este programa possui apenas um ente.
*****************************************************************/

import read  // Autoriza o uso da acao READ "neste" ente.
holo()
{
  holo()
  {
    write('Digite o avo:'),
    read(Avo),                // Aqui esta a acao READ.
    familia(Avo,Neto),
    writeln(Avo,' e avo de ',Neto)
  }

  familia/2() // MLA que gerencia a base Prolog.
  {
    familia(X,Y) :- pai(X,Z),pai(Z,Y).
    familia(X,Y) :- pai(X,Z),mae(Z,Y).
    familia(X,Y) :- mae(X,Z),pai(Z,Y).
    familia(X,Y) :- mae(X,Z),mae(Z,Y).

    pai(nadir,jorge).
    pai(nadir,bia).
    pai(nadir,sergio).
    pai(sergio,antonio).
    pai(sergio,helena).
    mae(zita,jorge).
    mae(zita,bia).
    mae(zita,sergio).
    mae(bia,victoria).
    mae(bia,catarina).
  }
}
```

Anexo 4 Arquivos Gerados pela Conversão de *datamining.holo*

Este anexo contém os cinco arquivos gerados pela conversão do programa *datamining.holo* (anexo 3.1 e figuras 5.7 e 5.8) usando a HoloJava e o Prolog Café. Os arquivos são os seguintes:

- anexo 4.1: *holo.java*
- anexo 4.2: *mine.java*
- anexo 4.3: *miner.java*
- anexo 4.4: *fib.pl*
- anexo 4.5: *PRED_fib_2.java* (*fib.pl* convertido pelo Prolog Café)

Anexo 4.1 Arquivo *holo.java*

```
import jada.*;
import holoj.lang.*;

public class  holo extends Being{

        holo(){
        this.father=null;
        this.name="holo";
        this.son=new BVector();
        history = new ObjectSpace();
        }

        public static void main (String args[]){
        Being holo_temp = null;
        holo holo = new holo();
        Tuple tuple = null;
        holo.insert_history();
        System.out.println("HOLO: Vou criar tres minas e um mineiro");
        Being mine_d1 = new mine(holo,"mine_d1","1");
        (new Thread((Runnable)mine_d1)).start();
        holo.son.addElement(mine_d1);
        Being mine_d2 = new mine(holo,"mine_d2","2");
        (new Thread((Runnable)mine_d2)).start();
        holo.son.addElement(mine_d2);
        Being mine_d3 = new mine(holo,"mine_d3","3");
        (new Thread((Runnable)mine_d3)).start();
        holo.son.addElement(mine_d3);
        Being miner_d = new miner(holo,"miner_d");
        (new Thread((Runnable)miner_d)).start();
        holo.son.addElement(miner_d);
        String X = "";
        for (int X_integer = 1;X_integer<=3;X_integer++){
        X =""+X_integer;
        tuple=(Tuple)holo.history.in(new Tuple("list",new String().getClass(),
        new String().getClass(),new String().getClass()));
        //Position: 4 -- counthist:3
        String Ident = (String)tuple.getItem(1);
        String Num = (String)tuple.getItem(2);
        String Fibo = (String)tuple.getItem(3);
        System.out.println("HOLO: Terminou a mineracao da mina:"+Ident);
        System.out.println("HOLO: Fibonacci de "+Num+" e "+Fibo);
        }
        System.out.println("HOLO: Terminou a mineracao");
        }

        public void insert_history() {
        }
}
```

Anexo 4.2 Arquivo *mine.java*

```
import jada.*;
import holoj.lang.*;

public class mine extends Being implements Runnable{


        public String Ident;

        mine(Being father,String name, String Ident){
        this.Ident=Ident;
        this.father=father;
        this.son=new BVector();
        this.name=name;
        history = new ObjectSpace();
        }

        public void run(){
        insert_history();
        Tuple tuple=null;
        Being holo_temp = null;
        System.out.println("MINA "+Ident+": Fui criada");
        }

        public void insert_history() {
        history.out (new Tuple ("list","1","2"));
        history.out (new Tuple ("list","2","4"));
        history.out (new Tuple ("list","3","6"));
        }
}
```

Anexo 4.3 Arquivo *miner.java*

```
import jada.*;
import holoj.lang.*;

public class miner extends Being implements Runnable{

        miner(Being father,String name){
        this.father=father;
        this.son=new BVector();
        this.name=name;
        history = new ObjectSpace();
        }

        public void run(){
        insert_history();
        Tuple tuple=null;
        Being holo_temp = null;
        System.out.println("MINEIRO: Inicio da mineracao.");
        this.father.son.removeElement(this);
        ((Being)this.father.son.getBeing("mine_d1")).son.addElement(this);
        this.father= ((Being)this.father.son.getBeing("mine_d1"));
        holoString int1_variavelholo = new holoString("1", false);
        int1_variavelholo.num=true;  String Num1="";
        holoString Num1_variavelholo = new holoString(Num1, true); String Res1="";
        holoString Res1_variavelholo = new holoString(Res1, true);
        this.mining(int1_variavelholo,Num1_variavelholo,Res1_variavelholo);
        if(Num1_variavelholo.update)
        {
        Num1=Num1_variavelholo.value;
        }
        if(Res1_variavelholo.update)
        {
        Res1=Res1_variavelholo.value;
        }

        this.father.son.removeElement(this);
        this.father.father.son.addElement(this);
        this.father=this.father.father;
        father.history.out (new Tuple ("list","1",Num1,Res1));
        //Position: 4 -- counthist:0
        this.father.son.removeElement(this);
        ((Being)this.father.son.getBeing("mine_d2")).son.addElement(this);
        this.father= ((Being)this.father.son.getBeing("mine_d2"));
        holoString int2_variavelholo = new holoString("2", false);
        int2_variavelholo.num=true;  String Num2="";
        holoString Num2_variavelholo = new holoString(Num2, true); String Res2="";
        holoString Res2_variavelholo = new holoString(Res2, true);
        this.mining(int2_variavelholo,Num2_variavelholo,Res2_variavelholo);
        if(Num2_variavelholo.update)
        {
        Num2=Num2_variavelholo.value;
        }
        if(Res2_variavelholo.update)
        {
        Res2=Res2_variavelholo.value;
        }

        this.father.son.removeElement(this);
        this.father.father.son.addElement(this);
        this.father=this.father.father;
        father.history.out (new Tuple ("list","2",Num2,Res2));
        //Position: 4 -- counthist:0
        this.father.son.removeElement(this);
        ((Being)this.father.son.getBeing("mine_d3")).son.addElement(this);
        this.father= ((Being)this.father.son.getBeing("mine_d3"));
        holoString int3_variavelholo = new holoString("3", false);
        int3_variavelholo.num=true;  String Num3="";
        holoString Num3_variavelholo = new holoString(Num3, true); String Res3="";
        holoString Res3_variavelholo = new holoString(Res3, true);
        this.mining(int3_variavelholo,Num3_variavelholo,Res3_variavelholo);
```

```
if(Num3_variavelholo.update)
{
Num3=Num3_variavelholo.value;
}
if(Res3_variavelholo.update)
{
Res3=Res3_variavelholo.value;
}

this.father.son.removeElement(this);
this.father.father.son.addElement(this);
this.father=this.father.father;
father.history.out (new Tuple ("list","3",Num3,Res3));
//Position: 4 -- counthist:0
System.out.println("MINEIRO: Fim da mineracao.");
}

public void mining(holoString Ident_, holoString Num_, holoString Result_){
Tuple tuple=null;
String Ident = Ident_.value;String Num = Num_.value;String Result =
Result_.value;
tuple = (Tuple)father.history.in(new Tuple("list",Ident,new
String().getClass()));
//Position: 3 -- counthist:1
Num = (String)tuple.getItem(2);
holoString Num_variavelholo = new holoString(Num, false);  holoString
Result_variavelholo = new holoString ("", true);
this.fib(Num_variavelholo,Result_variavelholo);
if(Num_variavelholo.update)
{
Num=Num_variavelholo.value;
}
if(Result_variavelholo.update)
{
Result=Result_variavelholo.value;
}

if(Ident_.update)
{
Ident_.value=Ident;
}
if(Num_.update)
{
Num_.value=Num;
}
if(Result_.update)
{
Result_.value=Result;
}

}
```

```
        public void fib(holoString parametro_0, holoString parametro_1)
        {

        Term parametro_0_prolog,parametro_1_prolog;
        PrologInterface p;
        Predicate predicado;
        Predicate cont;
        Predicate programa;
        if(parametro_0.update)
        {
        parametro_0_prolog = new VariableTerm();
        }else{
        if(Character.isDigit(((String)parametro_0.value).charAt(0)))
        {
        parametro_0_prolog = new IntegerTerm(Integer.parseInt(parametro_0.value));
        }else{
        parametro_0_prolog = SymbolTerm.makeSymbol(parametro_0.value);
        }} if(parametro_1.update)
        {
        parametro_1_prolog = new VariableTerm();
        }else{
        if(Character.isDigit(((String)parametro_1.value).charAt(0)))
        {
        parametro_1_prolog = new IntegerTerm(Integer.parseInt(parametro_1.value));
        }else{
        parametro_1_prolog = SymbolTerm.makeSymbol(parametro_1.value);
        }}
        p = new PrologInterface();
        cont = new ReturnJava(p);
        programa = new PRED_fib_2(parametro_0_prolog,parametro_1_prolog,cont);
        p.setPredicate(programa);
        p.call();

        if(parametro_0.update)
        {
        parametro_0.value = ""+parametro_0_prolog.dereference();
        } if(parametro_1.update)
        {
        parametro_1.value = ""+parametro_1_prolog.dereference();
        }
        }


        public void insert_history() {
        }
}
```

Anexo 4.4 Arquivo *fib.pl*

```
fib(1,1).
fib(2,1).
fib(M,N):-M>2,M1 is M-1,M2 is M-2,fib(M1,N1),fib(M2,N2),N is N1+N2.
```

Anexo 4.5 Arquivo *PRED_fib_2.java*

```
/*
 * *** Please do not edit ! ***
 * @(#) PRED_fib_2.java
 * @procedure fib/2 in fib.pl
 */

/*
 * @version Prolog Cafe 0.44, on November 12 1999
 * @author Mutsunori Banbara (banbara@pascal.seg.kobe-u.ac.jp)
 * @author Naoyuki Tamura    (tamura@kobe-u.ac.jp)
 */
public class PRED_fib_2 extends Predicate {
    static final Predicate fib_2_1 = new PRED_fib_2_1();
    static final Predicate fib_2_2 = new PRED_fib_2_2();
    static final Predicate fib_2_3 = new PRED_fib_2_3();
    static final Predicate fib_2_var = new PRED_fib_2_var();

    public Term arg1, arg2;

    public PRED_fib_2(Term a1, Term a2, Predicate cont) {
        arg1 = a1;
        arg2 = a2;
        this.cont = cont;
    }

    public PRED_fib_2(){}
    public void setArgument(Term[] args, Predicate cont) {
        arg1 = args[0];
        arg2 = args[1];
        this.cont = cont;
    }

    public Predicate exec() {
        engine.aregs[1] = arg1;
        engine.aregs[2] = arg2;
        engine.cont = cont;
        return call();
    }

    public final Predicate call() {
        engine.setB0();
        return engine.switch_on_term(
                                fib_2_var,
                                fib_2_var,
                                fib_2_3,
                                fib_2_3,
                                fib_2_3
                                );
    }

    public int arity() { return 2; }

    public String toString() {
        return "fib(" + arg1 + ", " + arg2 + ")";
    }
}

final class PRED_fib_2_var extends PRED_fib_2 {
    static final Predicate fib_2_var_1 = new PRED_fib_2_var_1();

    public final Predicate exec() {
        return engine.jtry(fib_2_1, fib_2_var_1);
    }
}

final class PRED_fib_2_var_1 extends PRED_fib_2 {
    static final Predicate fib_2_var_2 = new PRED_fib_2_var_2();

    public final Predicate exec() {
        return engine.retry(fib_2_2, fib_2_var_2);
    }
}
```

```
final class PRED_fib_2_var_2 extends PRED_fib_2 {

    public final Predicate exec() {
        return engine.trust(fib_2_3);
    }
}

final class PRED_fib_2_1 extends PRED_fib_2 {
    static final IntegerTerm s1 = new IntegerTerm(1);

    public final Predicate exec() {
        Term a1, a2;
        a1 = engine.aregs[1].dereference();
        a2 = engine.aregs[2].dereference();
        this.cont = engine.cont;

        if ( !s1.unify(a1, engine.trail) ) return engine.fail();
        if ( !s1.unify(a2, engine.trail) ) return engine.fail();
        return cont;
    }
}

final class PRED_fib_2_2 extends PRED_fib_2 {
    static final IntegerTerm s1 = new IntegerTerm(2);
    static final IntegerTerm s2 = new IntegerTerm(1);

    public final Predicate exec() {
        Term a1, a2;
        a1 = engine.aregs[1].dereference();
        a2 = engine.aregs[2].dereference();
        this.cont = engine.cont;

        if ( !s1.unify(a1, engine.trail) ) return engine.fail();
        if ( !s2.unify(a2, engine.trail) ) return engine.fail();
        return cont;
    }
}

final class PRED_fib_2_3 extends PRED_fib_2 {
    static final IntegerTerm s1 = new IntegerTerm(2);
    static final IntegerTerm s2 = new IntegerTerm(1);

    public final Predicate exec() {
        Term a1, a2, a3, a4, a5, a6;
        Predicate p1, p2, p3, p4, p5;
        a1 = engine.aregs[1].dereference();
        a2 = engine.aregs[2].dereference();
        this.cont = engine.cont;

        a3 = engine.makeVariable();
        a4 = engine.makeVariable();
        a5 = engine.makeVariable();
        a6 = engine.makeVariable();
        p1 = new PRED_$plus_3(a5, a6, a2, cont);
        p2 = new PRED_fib_2(a4, a6, p1);
        p3 = new PRED_fib_2(a3, a5, p2);
        p4 = new PRED_$minus_3(a1, s1, a4, p3);
        p5 = new PRED_$minus_3(a1, s2, a3, p4);
        return new PRED_$greater_than_2(a1, s1, p5);
    }
}
```

Anexo 5 Arquivo Contendo a Gramática da HoloJava 1.0

```
/**********************************

 HOLOJAVA 1.0

 Authors:  André Rauber Du Bois and
           Jorge Luis Victoria Barbosa
 Date: 15/6/2001
 Date of this Version:12/12/2001

 Contact us: holo@inf.ufrgs.br


**********************************/

PARSER_BEGIN(HoloJava)

import java.io.*;
import java.util.Vector;

class tipovariavel{
  public String variavel,tipo,pai;

  tipovariavel(String tipo,String variavel, String pai)
  {
   this.variavel=variavel;
   this.tipo=tipo;
   this.pai = pai;
  }
}

public class HoloJava
{
    static PrintWriter out;
    static PrintWriter prologfile;
    static int MAXVAR = 50;
    static int nparametros = 0;
    static int varcount=0;
    static int varname=0;
    static boolean useread=false;
    static boolean useexception=false;
    static boolean chamadaemacaoholo=false;
    static String nameprologfile ="";
    static String[] parametros = new String[10];
    static String[] variables = new String[MAXVAR];
    static String[] varhist = new String[10];
    static boolean[] varhistbool = new boolean[10];
    static int counthist,position = 0;
    static String enteAtual = "";
    static Vector tipvar = new Vector();


    public static void main (String [] args)
    {
        HoloJava holojava;
        String inputfile=null,outputfile=null;
        long initTime = 0;
        long parseTime = 0;
        long startTime = 0;
        long stopTime = 0;
        if (args.length == 1)
        {
           inputfile  = args[0]+".holo";
           outputfile = args[0]+".java";
        } else
        if (args.length == 2)
        {
           inputfile  = args[0]+".holo";
           outputfile = args[1]+".java";
        } else
        {
           System.out.println("=======================================\n");
           System.out.println("     HoloJava 1.0");
           System.out.println("    --------------\n");
```

```
            System.out.println("    Usage is: java HoloJava <inputfile>\n");
            System.out.println("=======================================");
            return;
        }

        try
        {
            out = new PrintWriter(new FileWriter("holo.java"));
        }catch (IOException e)
        {
            System.out.println("(HoloJava)\tError creating holo.java file");
        }

        try
        {
            startTime = System.currentTimeMillis();
            holojava = new HoloJava(new java.io.FileInputStream(inputfile));
            stopTime = System.currentTimeMillis();
            initTime = stopTime - startTime;
        } catch (java.io.FileNotFoundException e)
        {
            System.out.println("=======================================\n");
            System.out.println("     HoloJava 1.0");
            System.out.println("    --------------\n");
            System.out.println("    Error: File " + inputfile + " not found.\n");
            System.out.println("    Usage is: java HoloJava <inputfile>\n");
            System.out.println("=======================================");
            return;
        }

        try
        {
            startTime = System.currentTimeMillis();
            holojava.Holoprogram();
            stopTime = System.currentTimeMillis();
            parseTime = stopTime - startTime;
            System.out.println("=======================================\n");
            System.out.println("     HoloJava 1.0");
            System.out.println("    --------------\n");
            System.out.println("    Holoprogram: "+inputfile);
            System.out.println("    Successfull Conversion!\n");
            System.out.println("    Execution time: " + (initTime + parseTime) + " ms.");
            System.out.println("    Initialization time: " + initTime + " ms.");
            System.out.println("    Conversion time: " + parseTime + " ms.\n");
            System.out.println("=======================================");
            out.close();
        } catch (ParseException e)
        {
            System.out.println("=======================================\n");
            System.out.println("     HoloJava 1.0");
            System.out.println("    --------------\n");
            System.out.println("    Holoprogram: "+inputfile);
            System.out.println("    Errors During Parse!\n");
            System.out.println("---------------------------------------\n");
            System.out.println(e.getMessage());
            System.out.println("=======================================");
        }
    }

//---------------------------------------------------------------------
// Important Methods

    public static void initvarhist()
    {
        varhist = new String[10];
        counthist=0;
        position=0;
    }

    public static void variablesInit()
    {
        variables = new String[MAXVAR];
        varcount = 0;
    }
```

```
// initComandos: Initializes some important variables when using commands
    public static void initcomandos()
    {
        if (useread)
        {
         out.println("\tInputStreamReader reader_ = new InputStreamReader(System.in);");
         out.println("\tBufferedReader myInput = new BufferedReader (reader_);");
        }
    }

// endComandos: Finalizes the global variables used for commands
    public static void endcomandos()
    {
        useread = false;
    }
    public static void variablesEnd()
    {
        variables=null;
        varcount = 0;
    }

//endexception: Ends the exception Block
    public static void endexception()
    {
        if(useexception)
        {
          out.println("\t} catch (Exception e)
          {
            System.out.println (\"Exception: \" + e);} ");
            useexception=false;
          }
    }
    public static void includeVariable(String var)
    {
        variables[varcount]=var;
        varcount++;
    }

 //CheckVariable: checks if a variable exists
    public static boolean CheckVariable(String variable)
    {
        for (int i =0; i< varcount; i++)
        {
           if(variables[i].equals(variable))
           {
             return(true);
           }
        }
        return(false);
    }

//declareVar: checks if a variable already exists, if yes returns a string
//with the name of the variable if not it puts the variable in the variables
//table and returns a string that declares the variable in the program =
// "String variable_name"

    public static String declareVar (String var)
    {
        if(CheckVariable(var))
        {
           return("\t"+var);
        }
        else
        {
           includeVariable(new String (var));
           return("\tString " +var);
        }
    }
```

```
public synchronized static String getType(String var)
{
    int count=0;
    boolean ok= true;
    String tipo = "";
    tipovariavel temp = new tipovariavel("","","");

    if (tipvar.isEmpty())
       return null;
    else{
       int size = tipvar.size();
       while (ok)
       {
         temp = (tipovariavel) tipvar.elementAt(count);
         if (temp.variavel.equals(var))
         {
           ok =false;
         }
         else {
           count ++;
           if (count== size)
           {return null;}
         }
       }
    }
    return (temp.tipo);
}

public synchronized static String getTypeFather(String var)
{
    int count=0;
    boolean ok= true;
    String tipo = "";
    tipovariavel temp = new tipovariavel("","","");

    if (tipvar.isEmpty())
       return null;
    else{
       int size = tipvar.size();
       while (ok)
       {
         temp = (tipovariavel) tipvar.elementAt(count);
         if (temp.tipo.equals(var))
         {
           ok =false;
         }
         else {
           count ++;
           if (count== size)
           { return null;}
         }
       }
    }
    return (temp.pai);
}

public static String declareVarHist (String var)
{
    if(CheckVariable(var))
    {
       return(var);
    }
    else
    {
       includeVariable(new String (var));
       varhist[counthist]=(""+ position)+var;
       varhistbool[counthist]=false;
       counthist++;
       return("new String().getClass()");
    }
}
```

```
    public static String declareVarHistOp (String var)
    {
        if(CheckVariable(var))
        {
           varhist[counthist]=(""+ position)+var;
           varhistbool[counthist]=true;
           counthist++;
           return("new String().getClass()");
        }
        else
        {
           includeVariable(new String (var));
           varhist[counthist]=(""+ position)+var;
           varhistbool[counthist]=false;
           counthist++;
           return("new String().getClass()");
        }
    }

    public static void OpenFile (String outputfile)
    {
        try
        {
           out.close();
           out = new PrintWriter(new FileWriter(outputfile));
        }catch (IOException e)
        {
           System.out.println("(HoloJava)\tError creating output file");
        }
    }

    public static void openPrologFile (String outputfile)
    {
        try
        {
           prologfile = new PrintWriter(new FileWriter(outputfile+".pl"));
        }catch (IOException e)
        {
           System.out.println("(HoloJava)\tError creating Prolog file");
        }
    }
// Declares the variables for the actions (Begin and End)
    public static String iniciodachamada (String variavel)
    {
        String resposta ="";
        if (CheckVariable(variavel))
        {
           if (CheckVariable(variavel+"_variavelholo"))
           {
             resposta = variavel+"_variavelholo = new holoString("+ variavel +
                        ", false);";
           }
           else
           {
             includeVariable(variavel+"_variavelholo");
             resposta = "\tholoString " + variavel + "_variavelholo = new holoString(" +
                        variavel + ", false);";
           }
        }
        else
        {
           if (CheckVariable(variavel+"_variavelholo"))
           {
             resposta = declareVar(variavel) + "=\"\";\n";
             resposta = resposta  + variavel + "_variavelholo = new holoString(" +
                        variavel + ", true);";
           }
           else
           {
             includeVariable(variavel+"_variavelholo");
             resposta = declareVar(variavel) + "=\"\";\n";
             resposta = resposta +"\tholoString "+ variavel +
                        "_variavelholo = new holoString("+ variavel + ", true);";
           }
        }
        return (resposta);
    }
```

```
public static String iniciodachamadaop (String variavel)
{
    String resposta ="";

    if (CheckVariable(variavel))
    {
       if (CheckVariable(variavel+"_variavelholo"))
       {
         resposta =  variavel + "_variavelholo = new holoString(\"\", true);";
       }
       else
       {
         includeVariable(variavel+"_variavelholo");
         resposta =  "\tholoString " + variavel +
                     "_variavelholo = new holoString (\"\", true);" ;
       }
    }
    else
    {
       if (CheckVariable(variavel+"_variavelholo"))
       {
         resposta = declareVar(variavel) + "=\"\";\n";
         resposta = resposta  + variavel +
                    "_variavelholo = new holoString (\"\", true);" ;
       }
       else
       {
         includeVariable(variavel+"_variavelholo");
         resposta = declareVar(variavel) + "=\"\";\n";
         resposta = resposta +"\tholoString "+ variavel +
                    "_variavelholo = new holoString(\"\", true);";
       }
    }
    return (resposta);
}

public static String iniciodachamadaref (String variavel)
{
    String resposta ="";
    if (CheckVariable(variavel+"_variavelholo"))
    {
      resposta =  variavel + "_variavelholo = new holoString("+ variavel +
                  ", true);";
    }
    else
    {
       includeVariable(variavel+"_variavelholo");
       resposta =  "holoString " +variavel + "_variavelholo = new holoString("+
                   variavel + ", true);";
    }
    return (resposta);
}

public static String chamadasymbol (String variavel)
{
    String resposta ="";
    if (CheckVariable(variavel+"_variavelholo"))
    {
       resposta = "";
    }
    else
    {
       includeVariable(variavel+"_variavelholo");
       resposta =  "\tholoString " + variavel + "_variavelholo = new holoString(\""+
                   variavel + "\", false);";
    }
    return (resposta);
}
```

```
    public static String chamadanumero (String variavel)
    {
        String resposta ="";

        if (CheckVariable("int"+variavel+"_variavelholo"))
        {
           resposta =  "\tint" + variavel + "_variavelholo = new holoString(\""+
                       variavel + "\", false);"+ "\n\tint" + variavel +
                       "_variavelholo.num=true;";
        }
        else
        {
           includeVariable("int"+variavel+"_variavelholo");
           resposta =  "\tholoString int" + variavel +
                       "_variavelholo = new holoString(\""+ variavel + "\", false);"+
                       "\n\tint" + variavel +"_variavelholo.num=true;";
        }
        return (resposta);
    }

    public static String fimdachamada (String variavel)
    {
        String resposta ="";

        resposta = "\tif(" + variavel + "_variavelholo.update)\n" +
                   "\t{\n\t" + variavel + "=" + variavel + "_variavelholo.value;"+
                   "\n\t}\n";
        return (resposta);
    }

    public static String varprolog (String variavel)
    {
        String resposta ="";

        includeVariable(variavel);
        resposta = "\tif("+variavel+".update)\n\t{\n\t" +
                   variavel+ "_prolog = new VariableTerm();\n\t}else{\n\t"+
                   "\tif(Character.isDigit(((String)"+variavel+
                   ".value).charAt(0)))\n\t{\n\t" + variavel+
                   "_prolog = new IntegerTerm(Integer.parseInt("+variavel+
                   ".value));\n\t}else{\n\t"+variavel+
                   "_prolog = SymbolTerm.makeSymbol("+variavel+".value);\n\t}}";
        includeVariable(variavel+"_prolog");
        return(resposta);
    }

    public static String fimprolog (String variavel)
    {
        String resposta ="";

        includeVariable(variavel);
        resposta = "\tif("+variavel+".update)\n\t{\n\t" +
                   variavel +".value = " +"\"\"+"+variavel+ "_prolog.dereference();"+
                   "\n\t}";
        return(resposta);
    }

    public static String varaction(String variavel)
    {
        String resposta ="";

        resposta = "String " +variavel + " = " + variavel + "_.value;";
        return (resposta);
    }

    public static String fimaction (String variavel)
    {
        String resposta ="";
        resposta = "\tif(" + variavel + "_.update)\n" + "\t{\n\t" + variavel +
                   "_.value=" + variavel + ";\n\t}\n";
        return (resposta);
    }
}

PARSER_END(HoloJava)
```

```
////////////////////////////////////////////////////
/// Here starts the parser

/* WHITE SPACE */

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

/* COMMENTS */

MORE :
{
  "//" : IN_SINGLE_LINE_COMMENT
|
  "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <MULTI_LINE_COMMENT: "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_MULTI_LINE_COMMENT>
MORE :
{
  < ~[] >
}

/* RESERVED WORDS AND LITERALS */
TOKEN :
{
  < MOVE: "move" >
| < CLONE: "clone" >
| < TOINTEGER: "toInteger" >
| < SPAWN: "spawn" >
| < TIME : "time" >
| < NEWNAME : "newName" >
| < CLONING: "cloning" >
| < INTERFACE: "interface" >
| < HISTORY: "history" >
| < HOLO: "holo" >
| < FOR: "for" >
| < NULL: "null" >
| < TO: "to" >
| < IF: "if" >
| < THEN: "then" >
| < ELSE: "else" >
| < WHILE: "while" >
| < DO: "do" >
| < WRITE: "write">
| < WRITELN: "writeln">
| < LENGTH: "length" >
| < OUT : "out" >
| < SELF : "self" >
| < IS : "is" >
| < RANDOM: "random" >
| < WHOAMI: "whoami" >
| < WHEREAMI: "whereami" >
| < HOWMANY: "howmany" >
| < DELAY: "delay" >
| < IMPORT: "import" >
| < READ: "read" >
| < PROLOG: "prolog" >
}
```

```
/* SYMBOL AND VARIABLE */

TOKEN :
{
  < SYMBOL: ["a"-"z"](["a"-"z","A"-"Z","0"-"9","_"])* >
| < VARIABLE: ["A"-"Z"](["a"-"z","A"-"Z","0"-"9","_"])*>
| < NUMBER : (["0"-"9"])+>
| < ARITH : ["+","-","*"]>
| < STRING : "'" (["a"-"z","A"-"Z","0"-"9","_"," ",":","=","[","]","(",")",",",
"-",".","!"])+ "'">
}

/* SEPARATORS */

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < BLOCK: "." >
| < NONBLOCK: "?" >
| < BLOCK_DEST: ".#" >
| < NONBLOCK_DEST: "?#" >
}

/* OPERATORS */

TOKEN :
{
  < AS : ":=" >
| < UN : "=:=" >
| < ID : "==" >
| < UI : "=" >
| < GT : ">" >
| < LT : "<" >
| < LE : "<=" >
| < GE : ">=" >
}

/**************************************
 * THE HOLO LANGUAGE GRAMMAR STARTS HERE *
 **************************************/

// defines the structure of an Holo program
void Holoprogram() :
{}
{
   HoloDeclaration()
   (Static_being())*
   <EOF>
}

// the imports
void imports() :
{}
{
   <IMPORT> {out.print ("import ");}
   (<READ> {useread=true;
            out.print ("java.io.InputStreamReader;\n
            import java.io.BufferedReader;\n
            import java.io.IOException;\n");
           }
     | <RANDOM>  {out.println ("random... ");}
   )
}
```

```
//---------------------------------
// declarition of the main being (holo)
void HoloDeclaration() :
{}
{
   (imports())*
   HoloHead() "{" HoloBody() "}"
   {out.println("}"); }
}
//---------------------------------
// defines the head of the main being
void HoloHead() :
{}
{
   <HOLO>
   { enteAtual = "holo";
     out.println("import jada.*;\nimport holoj.lang.*;\n\n"+
                 "public class  holo extends Being{\n\n" +
                 "\tholo(){\n\tthis.father=null;\n\tthis.name=\"holo\";\n\t
                 this.son=new BVector();\n\thistory = new ObjectSpace();\n\t}");
   }
   "(" [VariableList()] ")" ["cloning" "(" CDL() ")"] ["interface" AIList() "."]
}
//-------------------------------------------------------------
// defines the body of the main being
void HoloBody() :
{}
{
   {variablesInit();}
   HoloAction()
   {variablesEnd();}
   ({chamadaemacaoholo=true;} Action(){chamadaemacaoholo=false;})*
   {endcomandos();}
   {out.println("\n\n\tpublic void insert_history() {");}
   [History()]
   {out.println("\t}");}
}

//-----------------------------------
// Defines the main Action of the main being (the holo action)
void HoloAction() :
{}
{
   <HOLO>
   {
     out.println("\n\n\tpublic static void main (String args[]){" +
                 "\nBeing holo_temp = null;"+
                 "\n\tholo holo = new holo();\n\tTuple tuple = null;" +
                 "\n\tholo.insert_history();");
   }
   "(" [VariableList_Holo()] ")" "{"
   {
     for (int i=0;i<nparametros;i++)
     {
       out.println ("String "+parametros[i]+ " = args["+i+"];");//out.println("\n\t}");
     }
   }
   {initcomandos();}
   [(for_holo()| if_holo()|do_actions_holo())] "}"
   {endexception();
    out.println("\t}\n");
   }
}
//---------------------------------------
// parses the list of variables
void VariableList_Holo():
{Token token1;}
{
   {nparametros=0;}
   token1=<VARIABLE>
   {
     parametros[nparametros]= token1.image;
     nparametros++;includeVariable(token1.image);
   }
   ("," token1=<VARIABLE> {parametros[nparametros]=token1.image;
     includeVariable(token1.image);nparametros++;})*
}
```

```
//--------------------------------------
// defines what can occur inside of the HoloBeing
void do_actions_holo():
{}
{
   (AccessHistoryOut("holo") | Comandos() | Chamadas() | Atribuicao())
   ["," (for_holo()| if_holo()| do_actions_holo())]
}

//------------------------------------------------
// defines the if clause for the holo being
void if_holo():
{Token token1,token2;String varteste="";}
{

   <IF> <LPAREN>
   ( token1=<VARIABLE> {varteste=token1.image;}
     | token1=<SYMBOL> {varteste= "\""+token1.image+"\"";}
     | token1=<NUMBER> {varteste=token1.image;}
   )
   {out.print("\tif (" + varteste);}
   [ (token1=<ID> | token1=<GT> | token1=<LT> | token1=<LE> | token1=<GE>)
     {out.print(token1.image);}
     ( token1=<VARIABLE> {varteste=token1.image;}
      | token1=<SYMBOL>  {varteste= "\""+token1.image+"\"";}
      | token1=<NUMBER>  {varteste=token1.image;}
     )
     {out.print(varteste);}
   ]
   <RPAREN>{out.println("){");}
   "{"
   ( for_holo() | if_holo() | do_actions_holo() )
   "}"
   {out.println("\t}");}
   [ <ELSE> "{"{out.println("\telse{");}
     ( for_holo() | if_holo() | do_actions_holo() )
     "}"
     {out.println("\t}");}
   ]
   [( for_holo() | if_holo()| do_actions_holo()) ]
}

//----------------------------------
// defines the if clause of the other types of beings
void if_():
{Token token1,token2;String varteste=""; }
{
   <IF> <LPAREN>
   ( token1=<VARIABLE> {varteste=token1.image;}
    | token1=<SYMBOL>  {varteste= "\""+token1.image+"\"";}
    | token1=<NUMBER>  {varteste=token1.image;}
   )
   {out.print("\tif (" + varteste);}
   [( token1=<ID> | token1=<GT> | token1=<LT> | token1=<LE> | token1=<GE> )
    {out.print(token1.image);}
    ( token1=<VARIABLE> {varteste=token1.image;}
      | token1=<SYMBOL> {varteste= "\""+token1.image+"\"";}
      | token1=<NUMBER> {varteste=token1.image;}
    )
    {out.print(varteste);}
   ]
   <RPAREN>{out.println("){");}
   "{"
   (for_()|if_()|do_actions())
    "}"
   {out.println("\t}");}
   [ <ELSE> "{"{out.println("\telse{");}
    ( for_() | if_() | do_actions() )
    "}"
    {out.println("\t}");}
   ]
   [ (for_() | if_() | do_actions()) ]
}
```

```
//---------------------------------
// defines the destructive update of a varibale
void Atribuicao():
{Token token1,token2;}
{
   token1=<VARIABLE> ":="
   ( token2 = <VARIABLE>
     {out.println(declareVar(token1.image)+" = " + token2.image + ";");}
     | token2 = <SYMBOL>
     {out.println(declareVar(token1.image) +" =\"" + token2.image + "\";");}
     | token2 = <NUMBER>
     {out.println(declareVar(token1.image) +" =\"" + token2.image + "\";");}
     | {out.print(declareVar(token1.image));} AccessHistory())
}

// defines the call of a method
void Chamadas():
{Token token1,token2,token3;
 String finaldocomando ="",chamadainicio = "" , chamadafim = "", chamada ="";
}
{
 token1=<SYMBOL>
 //local call
 ////////////////
 (
  (<LPAREN>
   {
     if(enteAtual.equals("holo") && (!chamadaemacaoholo))
     {
       chamada =  "\tholo." + token1.image + "(" ;
     }
     else
     {
       chamada = "\tthis." + token1.image + "(";
     }
   }
   [("&" token1 = <VARIABLE>
     {
       chamada= chamada +token1.image + "_variavelholo";
       chamadainicio = iniciodachamadaref(token1.image);
       chamadafim = fimdachamada(token1.image);
     }
     | "#" token1 = <VARIABLE>
     {
       chamada = chamada +token1.image + "_variavelholo";
       chamadainicio = iniciodachamadaop(token1.image);
       chamadafim = fimdachamada(token1.image);
     }
     | token1 = <VARIABLE>
     {
       chamada= chamada +token1.image + "_variavelholo";
       chamadainicio = iniciodachamada(token1.image);
       chamadafim = fimdachamada(token1.image);
     }
     | token1 = <SYMBOL>
     {
       chamada= chamada +token1.image + "_variavelholo";
       chamadainicio = chamadasymbol(token1.image);
     }
     | token1 = <NUMBER>
     {
       chamada= chamada +"int"+token1.image + "_variavelholo";
       chamadainicio = chamadanumero(token1.image);
     }
    )
    ( ","
      ( "&" token1=<VARIABLE>
        {
          chamada= chamada + "," +token1.image + "_variavelholo";
          chamadainicio = chamadainicio +iniciodachamadaref(token1.image);
          chamadafim = chamadafim + fimdachamada(token1.image);
        }
        | "#" token1 = <VARIABLE>
        {
          chamada = chamada +"," +token1.image + "_variavelholo";
          chamadainicio = chamadainicio+iniciodachamadaop(token1.image);
          chamadafim = chamadafim+fimdachamada(token1.image);
```

```
       }
       | token1 = <VARIABLE>
       {
         chamada= chamada + "," +token1.image + "_variavelholo";
         chamadainicio = chamadainicio +iniciodachamada(token1.image);
         chamadafim = chamadafim + fimdachamada(token1.image);
       }
       | token1 = <SYMBOL>
       {
         chamada= chamada + "," + token1.image + "_variavelholo";
         chamadainicio = chamadainicio + chamadasymbol(token1.image);
       }
       | token1 = <NUMBER>
       {
         chamada= chamada +",int"+token1.image + "_variavelholo";
         chamadainicio = chamadainicio+chamadanumero(token1.image);
       }
     )
   )*
  ]
 <RPAREN> {chamada = chamada + ");";}
)
{out.println(chamadainicio+"\n"+chamada+"\n"+chamadafim);}


//////////////////////
//calling other being
//////////////////////
| ( "." token2 = <SYMBOL>
    {
      if(enteAtual.equals("holo"))
      {
        chamada = ("(("+ getType(token1.image)+
                  ")holo.son.getBeing(\""+token1.image+"\"))." + token2.image+"(");
      }
      else
      {
        chamada =("(("+ getType(token1.image)+
                  ")this.son.getBeing(\""+token1.image+"\"))." + token2.image +"(");
      }
    }
    <LPAREN>
    [ ( "&" token1 = <VARIABLE>
        {
          chamada= chamada +token1.image + "_variavelholo";
          chamadainicio = iniciodachamadaref(token1.image);
          chamadafim = fimdachamada(token1.image);
        }
        | "#" token1 = <VARIABLE>
        {
          chamada = chamada +token1.image + "_variavelholo";
          chamadainicio = chamadainicio+iniciodachamadaop(token1.image);
          chamadafim = chamadafim+fimdachamada(token1.image);
        }
        | token1 = <VARIABLE>
        {
          chamada= chamada +token1.image + "_variavelholo";
          chamadainicio = iniciodachamada(token1.image);
          chamadafim = fimdachamada(token1.image);
        }
        | token1 = <SYMBOL>
        {
          chamada = chamada + token1.image + "_variavelholo";
          chamadainicio = chamadasymbol(token1.image);
        }
        | token1 = <NUMBER>
        {
          chamada= chamada +"int"+token1.image + "_variavelholo";
          chamadainicio = chamadanumero(token1.image);
        }
      )
      ( ","
        ( "&" token1 = <VARIABLE>
          {
             chamada= chamada + "," +token1.image + "_variavelholo";
             chamadainicio = chamadainicio +iniciodachamadaref(token1.image);
             chamadafim = chamadafim + fimdachamada(token1.image);
```

```
            }
            | "#" token1 = <VARIABLE>
            {
               chamada = chamada +"," +token1.image + "_variavelholo";
               chamadainicio = chamadainicio+iniciodachamadaop(token1.image);
               chamadafim = chamadafim+fimdachamada(token1.image);
            }
            | token1 = <VARIABLE>
            {
               chamada= chamada + "," +token1.image + "_variavelholo";
               chamadainicio = chamadainicio +iniciodachamada(token1.image);
               chamadafim = chamadafim + fimdachamada(token1.image);
            }
            | token1 = <SYMBOL>
            {
               chamada = chamada + "," + token1.image + "_variavelholo";
               chamadainicio = chamadainicio + chamadasymbol(token1.image);
            }
            | token1 = <NUMBER>
            {
              chamada= chamada +",int"+token1.image + "_variavelholo";
              chamadainicio = chamadainicio+chamadanumero(token1.image);
            }
          )
        )*
      ]
      <RPAREN> {chamada = chamada + ");";}
      {out.println(chamadainicio+"\n"+chamada+"\n"+chamadafim);}
    )
  )
}

// defines the call of actions using out
void chamadaout():
{
  Token token1,token2,token3;
  String finaldocomando ="",chamadainicio = "" , chamadafim = "", chamada ="",temp="";
}
{
   token1 = <SYMBOL> <RPAREN> "." token2 = <SYMBOL>
   {
     if(token1.image.equals("behavior"))
     {
       chamada="(("+ getTypeFather(enteAtual)+")this.father)." + token2.image +"(";
     }
     else
     {
       chamada =(("(("+ getType(token1.image)+
                ")this.father.son.getBeing(\""+token1.image+"\"))." +token2.image +"(");
     }
   }
   <LPAREN>
   [
     (
       "&" token1=<VARIABLE>
       {
          chamada= chamada +token1.image + "_variavelholo";
          chamadainicio = iniciodachamadaref(token1.image);
          chamadafim = fimdachamada(token1.image);
       }
       | "#" token1 = <VARIABLE>
       {
          chamada = chamada +token1.image + "_variavelholo";
          chamadainicio = chamadainicio+iniciodachamadaop(token1.image);
          chamadafim = chamadafim+fimdachamada(token1.image);
       }
       | token1 = <VARIABLE>
       {
         chamada= chamada +token1.image + "_variavelholo";
         chamadainicio = iniciodachamada(token1.image);
         chamadafim = fimdachamada(token1.image);
       }
       | token1 = <SYMBOL>
       {
         chamada= chamada +token1.image + "_variavelholo";
         chamadainicio = chamadasymbol(token1.image);
       }
```

```
      | token1 = <NUMBER>
      {
        chamada = chamada +"int"+token1.image + "_variavelholo";
        chamadainicio = chamadanumero(token1.image);
      }
    )
    ( ","
      (
        "&" token1 = <VARIABLE>
        {
          chamada= chamada + "," +token1.image + "_variavelholo";
          chamadainicio = chamadainicio +iniciodachamadaref(token1.image);
          chamadafim = chamadafim + fimdachamada(token1.image);
        }
        | "#" token1 = <VARIABLE>
        {
          chamada = chamada +"," +token1.image + "_variavelholo";
          chamadainicio = chamadainicio+iniciodachamadaop(token1.image);
          chamadafim = chamadafim+fimdachamada(token1.image);
        }
        | token1 = <VARIABLE>
        {
          chamada= chamada + "," +token1.image + "_variavelholo";
          chamadainicio = chamadainicio +iniciodachamada(token1.image);
          chamadafim = chamadafim + fimdachamada(token1.image);
        }
        | token1 = <SYMBOL>
        {
          chamada= chamada + "," + token1.image + "_variavelholo";
          chamadainicio = chamadainicio + chamadasymbol(token1.image);
        }
        | token1 = <NUMBER>
        {
          chamada= chamada +",int"+token1.image + "_variavelholo";
          chamadainicio = chamadainicio+chamadanumero(token1.image);
        }
      )
    )*
  ]
  <RPAREN> {chamada = chamada + ");";}
  {out.println(chamadainicio+"\n"+chamada+"\n"+chamadafim);}
}


// defines the access of the history
void AccessHistory():
{
  Token token,token1;
  String tipo;boolean tenho_que_imprimir_ofinal=true;
}
{
  {
    initvarhist();
  }
  "history"
  ( <BLOCK> {tipo="read";}
    | <NONBLOCK> {tipo="read_nb";}
    | "!" {tipo="out";}
    | <BLOCK_DEST> {tipo="in";}
    | "#" {tipo= "in";}
    | <NONBLOCK_DEST> {tipo="in_nb";}
  )
  ( token = <SYMBOL>
    {
      if (tipo == "read" || tipo == "read_nb" || tipo == "in_nb" || tipo=="in")
      {
        out.print("\ttuple = (Tuple) history."+ tipo +
                  "(new Tuple(\"" + token.image+"\"");
      }
      else
      {
        out.print("\thistory.out (new Tuple (\"" + token.image +"\"" );
      }
      position++;
    }
```

```
      [
        <LPAREN>
        {
          tenho_que_imprimir_ofinal=false;
          out.print(",");
        }
        ( token1 = <SYMBOL>      {out.print( "\"" + token1.image+"\"");}
         | token1 = <NUMBER>     {out.print( "\"" + token1.image+"\"");}
         | token1 = <VARIABLE>   {out.print (declareVarHist(token1.image));}
         | "#" token1=<VARIABLE> {out.print (declareVarHistOp(token1.image));}
        )
        {position++;}
        ( ","
         ( token1 = <SYMBOL> {out.print( ",\"" + token1.image+"\"");}
          | token1 = <NUMBER> {out.print( ",\"" + token1.image+"\"");}
          | token1 = <VARIABLE> {out.print(","+declareVarHist(token1.image));}
          | "#" token1 = <VARIABLE> {out.print(","+declareVarHistOp(token1.image));}
         )
         {position++;}
        )*
       <RPAREN>
       {out.println("));");}
       {
         out.println("//Position: " + position + " -- counthist:" + counthist);
         for (int i=0; i<counthist;i++)
         {
           if(varhistbool[i])
           {
             out.println("\t"+varhist[i].substring(1)+" = (String)tuple.getItem("+
                         varhist[i].substring(0,1)+");");
           }
           else
           {
             out.println("\tString "+varhist[i].substring(1)+" = (String)tuple.getItem("+
                         varhist[i].substring(0,1)+");");
           }
         }
       }
      ]
      {
        if (tenho_que_imprimir_ofinal) {out.println("));");}
      }
      | token1 = <VARIABLE>
       {
         if (tipo == "read" || tipo == "read_nb" || tipo == "in_nb" || tipo=="in")
         {
           out.println("tuple = (Tuple)history."+ tipo +
                       "(new Tuple( new String().getClass()));");
           out.println (declareVar(token1.image) + " = (String)tuple.getItem(0);");
         }
         else
         {
           out.print("history.out (new Tuple (" + token1.image + "));");
         }
       }
    )
}

// deines the History access using out
void AccessHistoryOut(String valor):
{
  Token token,token1;
  String tipo;boolean tenho_que_imprimir_ofinal=true;
}
{
  { initvarhist(); }
  "history"
  [ <RPAREN> ]
  (
    <BLOCK> {tipo="read";}
    | <NONBLOCK> {tipo="read_nb";}
    | "!" {tipo="out";}
    | <BLOCK_DEST> {tipo="in";}
    | "#" {tipo= "in";}
    | <NONBLOCK_DEST> {tipo="in_nb";}
  )
```

```
  (
    token = <SYMBOL>
    {
      if (tipo == "read" || tipo == "read_nb" || tipo == "in_nb" || tipo=="in")
      {
        out.print("tuple = (Tuple)"+valor+".history."+ tipo + "(new Tuple(\"" +
                  token.image+"\"");
      }
      else
      {
        out.print(valor+".history.out (new Tuple (\"" + token.image + "\"");
      }
      position++;
    }
    [
      <LPAREN>
      {tenho_que_imprimir_ofinal=false;out.print(",");}
      (
        token1 = <SYMBOL>     {out.print( "\"" + token1.image+"\"");}
        | token1 = <NUMBER>   {out.print( "\"" + token1.image+"\"");}
        | token1 = <VARIABLE> {out.print (declareVarHist(token1.image));}
        | "#" token1 = <VARIABLE> { out.print (declareVarHistOp(token1.image));}
      )
      {position++;}
      ( ","
        (
          token1 = <SYMBOL>     {out.print( ",\"" + token1.image+"\"");}
          | token1 = <NUMBER>   {out.print( ",\"" + token1.image+"\"");}
          | token1 = <VARIABLE> { out.print (","+declareVarHist(token1.image));}
          | "#" token1 = <VARIABLE> { out.print (","+declareVarHistOp(token1.image));}
        )
        {position++;}
      )*
      <RPAREN>
      {out.println("));");}
      {
        out.println("//Position: " + position + " -- counthist:" + counthist);
        for (int i=0; i<counthist;i++)
        {
          if(varhistbool[i])
          {
            out.println("\t"+varhist[i].substring(1)+" = (String)tuple.getItem("+
                        varhist[i].substring(0,1)+");");
          }
          else
          {
            out.println("\tString "+varhist[i].substring(1)+" = (String)tuple.getItem("+
                        varhist[i].substring(0,1)+");");
          }
        }
      }
    ]
    { if(tenho_que_imprimir_ofinal){ out.println("));"); } }
    | token1 = <VARIABLE>
      { if (tipo == "read" || tipo == "read_nb" || tipo == "in_nb" || tipo=="in")
        {
          out.println("tuple = (Tuple)"+valor+".history."+ tipo +
                      "(new Tuple( new String().getClass()));");
          out.println (declareVar(token1.image) + " = (String)tuple.getItem(0);");
        }
        else
        {
          out.print(valor+".history.out (new Tuple (" + token1.image + "));");
        }
      }
  )
}
```

```
// defines the possible commands in Holo
void Comandos():
{
   Token token1,token2,token3;String param ="";
}
{
   // WRITELN DEFINITION (BEGIN)
   (
    ( <WRITELN> <LPAREN>
      {out.print("\tSystem.out.println(");}
      (
        (
          token1=<VARIABLE>{out.print (token.image);}
        )
        | token1 = <SYMBOL> {out.print ("\""+token.image+"\"");}
        | token1 = <STRING>
         {out.print ("\"" +
                    ((String)token.image).substring(1,(token.image.length())-1)+"\"");
         }
      )
      ( "," {out.print("+");}
        (
          (
             token1 = <VARIABLE> {out.print (token.image);}
          )
          | token1 = <SYMBOL> {out.print ("\""+token.image+"\"");}
          | (<LPAREN> token1 = <VARIABLE> token2 = <ARITH> token3 = <VARIABLE>
              {out.print("("+token1.image+ " "+ token2.image+ " " + token3.image);}
              ( token2 = <ARITH> token3 = <VARIABLE>
                {out.print(token2.image+ " " + token3.image);}
              )*
              <RPAREN> {out.println(")");}
            )
          | token1 = <STRING>
             { out.print ("\"" +
                    ((String)token.image).substring(1,(token.image.length())-1)+"\"");
             }
        )
      )*
      <RPAREN> {out.println (");");}
    )
    //(END) WRITELN DEFINITION
    | (<WRITE> <LPAREN> {out.print("\tSystem.out.print(");}
        (
          (
            token1 = <VARIABLE> {out.print (token.image);}
          )
          | token1 = <SYMBOL> {out.print ("\""+token.image+"\"");}
          | token1 = <STRING> {out.print ("\"" +
                    ((String)token.image).substring(1,(token.image.length())-1)+"\"");}
        )
        ( "," {out.print("+");}
          (
             (
                token1=<VARIABLE>{out.print (token.image);}
             )
             | token1 = <SYMBOL> {out.print ("\""+token.image+"\"");}
             | (
                 <LPAREN> token1 = <VARIABLE> token2 = <ARITH> token3 = <VARIABLE>
                 {out.print("("+token1.image+ " "+ token2.image+ " " + token3.image);}
                 (
                    token2=<ARITH> token3=<VARIABLE>{out.print(token2.image+ " " +
                            token3.image);}
                 )*
                 <RPAREN> {out.println(")");}
               )
             | token1 = <STRING>
               {out.print ("\"" +
                    ((String)token.image).substring(1,(token.image.length())-1)+"\"");
               }
          )
        )*
        <RPAREN> {out.println (");");}
      )
```

```
  | ( <CLONE> <LPAREN> token1 = <SYMBOL>
      [
        "("
        (
          token3=<NUMBER> {param = param + ",\""+ token3.image+ "\"";}
          | token3 = <SYMBOL> {param = param + ",\""+ token3.image+ "\"";}
          | token3 = <VARIABLE> {param = param + "," +token3.image ;}
        )
        ( ","
          (
            token3   = <NUMBER>   {param = param + ",\""+ token3.image+ "\"";}
            | token3 = <SYMBOL>   {param = param + ",\""+ token3.image+ "\"";}
            | token3 = <VARIABLE> {param = param + "," + token3.image ;}
          )
        )*
        ")"
      ]
      ","
      (
         token2 = <SYMBOL> <RPAREN>
         {
           tipvar.addElement(new tipovariavel(token1.image,token2.image,enteAtual));
           out.print ("\tBeing "+token2.image+" = new "+token1.image+"(");
           if (enteAtual.equals("holo"))
           {
             out.print("holo");
             out.println(",\""+token2.image+"\""+param+");\n\t(new Thread((Runnable)"+
                         token2.image+")).start();");
             out.println("\tholo.son.addElement("+token2.image+");");
           }
           else
           {
             out.print("this");
             out.println(",\""+token2.image+"\""+param+");\n\t(new Thread((Runnable)"+
                         token2.image+")).start();");
             out.println("\tthis.son.addElement("+token2.image+");");
           }
         }
         | token2 = <NULL> <RPAREN>
           {
             tipvar.addElement(new tipovariavel(token1.image,"varname_"+
                               varname,enteAtual));
             out.print ("\tholo_temp  = new "+token1.image+ "(");
             if (enteAtual.equals("holo"))
             {
               out.print("holo");
               out.println(",\"varname_"+varname+"\""+param+
                           ");\n\t(new Thread((Runnable)"+"holo_temp)).start();");
               out.println("\tholo.son.addElement(holo_temp);\n\tholo_temp=null;");
               varname ++;
             }
             else
             {
               out.print("this");
               out.println(",\"varname_"+varname+"\""+param+
                           ");\n\t(new Thread((Runnable)"+"holo_temp)).start();");
               out.println("\tthis.son.addElement(holo_temp);\n\tholo_temp=null;");
               varname ++;
             }
           }
      )
    )
    | (
        <OUT> <LPAREN>
        (
           AccessHistoryOut("father") [ <RPAREN> ] | chamadaout()
        )
      )
)
```

```
    | <WHOAMI> <LPAREN> token1 = <VARIABLE> <RPAREN>
      {if (enteAtual.equals("holo"))
        {
          out.println(declareVar(token1.image) + " = holo.name;");
        }
        else
        {
          out.println(declareVar(token1.image) + " = this.name;");
        }
      }
    | <WHEREAMI> <LPAREN> token1 = <VARIABLE> <RPAREN>
      {if(enteAtual.equals("holo"))
        {
          out.println(declareVar(token1.image) + " = erro;");
        }
        else
        {
          out.println(declareVar(token1.image) + " = this.father.name;");
        }
      }
    | <HOWMANY> <LPAREN> token1 = <VARIABLE> <RPAREN>
      {if(enteAtual.equals("holo"))
        {out.println(declareVar(token1.image) + " = \"\" + holo.son.size();");}
        else
        {out.println(declareVar(token1.image) + " = \"\" + this.son.size();"); }
      }
    | ( <READ> <LPAREN> token1 = <VARIABLE> <RPAREN>)
      {
        out.println("\ttry{");
        out.println(declareVar(token1.image) + " = myInput.readLine();");
        useexception=true;
      }
    | ( <TOINTEGER> <LPAREN> token1 = <VARIABLE> "," token2 = <VARIABLE> <RPAREN>
       {
        includeVariable(token2.image);
        out.println("int "+token2.image+" = Integer.parseInt("+token1.image+");");
       }
      )
    | ( <TIME> <LPAREN> token1=<VARIABLE> <RPAREN>
       {
         includeVariable(token1.image);
         out.println("long "+ token1.image+ " = System.currentTimeMillis();");
       }
      )
    | ( <MOVE> <LPAREN> (token1 = <SELF> | token1 = <SYMBOL>) ","
        ( token2 = <OUT> | token2 = <SYMBOL> )
        <RPAREN>
        {
           if (token1.image.equals("self"))
           {
             if(token2.image.equals("out"))
             {
               out.println("this.father.son.removeElement(this);");
               out.println("this.father.father.son.addElement(this);");
               out.println("this.father=this.father.father;");
             }
             else
             {
               out.println("this.father.son.removeElement(this);");
               out.println("((Being)this.father.son.getBeing(\""+
                           token2.image+"\")).son.addElement(this); ");
               out.println("this.father= ((Being)this.father.son.getBeing(\""+
                           token2.image+"\"));");
             }
           }
           else
           {
             out.println("((Being)holo.son.getBeing(\""+token2.image+
                         "\")).son.addElement((Being)holo.son.getBeing(\""+
                         token1.image+"\"));");
             out.println("holo.son.removeBeing(\""+token1.image+"\");\n");
           }
        }
      )
}
```

```
//-------------------------------------
// defines the for command
void for_holo():
{
  Token token1,token2,token3;
  boolean var=false;
}
{
   <FOR> token1 = <VARIABLE> ":=" token2 = <NUMBER> "to"
   (
     token3=<VARIABLE>
     { if(!CheckVariable(token3.image+"_integer"))
       {
         includeVariable(token3.image+"_integer");
         out.println("int "+token3.image+"_integer = Integer.parseInt("+
                     token3.image+");");
         var=true;
       }
       else
       {
         out.println(token3.image+"_integer = Integer.parseInt("+ token3.image+");");
         var=true;
       }
     }
     | token3 = <NUMBER>) "do"
       "{"
       {
         if(!CheckVariable(token1.image+"_integer"))
           {includeVariable(token1.image+"_integer");}
         if(!CheckVariable(token1.image))
           {includeVariable(token1.image);
         out.println("String "+token1.image+" = \"\";");}
         out.print("for (int "+ token1.image +"_integer = "+
         token2.image+";"+token1.image+"_integer"+"<=");
         if(var)
           {out.print(token3.image+"_integer");}
         else
           {out.print(token3.image);}
         out.println(";"+token1.image+"_integer"+"++){\n\t"+token1.image+" =\"\"+"+
                     token1.image+"_integer;");
       }
       [
         (
           for_holo() | if_holo()| do_actions_holo()
         )
       ]
       "}"
       {out.println("}");}
       [
         (
           for_holo() |if_holo() | do_actions_holo()
         )
       ]
}


// defines the structure of the static beings
void Static_being() :
{}
{
   Head() "{" Body() "}"
  {
    out.println("}");
  }
}
```

```
//-------------------------------------
// defines the head of the static beings
void Head() :
{
   Token token;
   String imports="";
}
{
   ///BEGIN IMPORTS
   (
     <IMPORT> {imports=imports+"import ";}
     (
       <READ>{imports=imports+"java.io.InputStreamReader;" +
                     "\nimport java.io.BufferedReader;\nimport java.io.IOException;\n";
              useread=true;
             }
       | <RANDOM> {imports=imports+" random... \n";}
     )
   )*
   ///END IMPORTS

   token = <SYMBOL>
   {
     OpenFile((token.image)+".java");
     enteAtual= token.image;
     out.print(imports);
     out.println("import jada.*;\nimport holoj.lang.*;\n\npublic class " +
     token.image + " extends Being implements Runnable{\n\n\t");
   }
   "("
   [ VariableList() ]
   ")"
   [ "cloning" "(" CDL() ")"  ]
   [ "interface" AIList() "." ]
}

//-------------------------------------
//defines the Body of the static beings
void Body() :
{}
{
   main_being()
   (Action())*
   {endcomandos();}
   {out.println("\n\n\tpublic void insert_history() {");}
   [History()]
   {out.println("\t}");}
}

//-------------------------------------
//defines the main action of the being
void main_being() :
{
   Token token,token1;
   String declaration="";
   String param="";String setvalues="";
}
{
  {variablesInit();//existecorpo=true;}
  token = <SYMBOL>
  "("
  [
    token1 = <VARIABLE>
    {
      declaration = declaration+"public String " + token1.image +";\n\t";
      setvalues=setvalues+"this." +token1.image+"="+token1.image+";\n\t";
      param=param+", String " + token1.image; includeVariable(token1.image);
    }
    ("," token1 = <VARIABLE>
      {
        declaration = declaration+"public String " + token1.image +";\n\t";
        setvalues=setvalues+"this." +token1.image+"="+token1.image+";\n\t";
        param = param+", String " + token1.image;
        includeVariable(token1.image);
      }
    )*
  ]
```

```
  ")"
  { out.println(declaration);
    out.print(enteAtual +"(Being father,String name");
    out.print(param);
    out.println("){\n\t" + setvalues + "this.father=father;\n\tthis.son=
                 new BVector();\n\tthis.name=name;\n\t
               history = new ObjectSpace();\n\t}\n");
  }
  { out.print("\tpublic void run(");
    out.println("){"+"\n\tinsert_history();\n\t
               Tuple tuple=null;"+"\nBeing holo_temp = null;");
  }
  {initcomandos();}
  "{"
  [( for_() | if_() | do_actions() ) ]
  {variablesEnd();}
  "}"
  {
    endexception();
    out.println("\t}\n");
  }
}

//---------------------------------------
// defines the actions of all beings
void Action() :
{
   Token token;
   String declaracao="",resposta="";
}
{
   token = <SYMBOL>
   {out.print("\tpublic void "+ token.image + "(");}
   (
     ( "/" {
             openPrologFile(token.image);
             nameprologfile=token.image;
           }
       mla()
     )
     | ( "("
         {variablesInit();}
         // PAREMETROS INICIO
         [
           token=<VARIABLE>
           {
             out.print("holoString " + token.image+"_");
             includeVariable(token.image+"_");
             includeVariable(token.image);
             declaracao = declaracao +varaction(token.image);
             resposta = resposta + fimaction(token.image);
           }
           ( "," token = <VARIABLE>
             {
               out.print(", holoString " + token.image+"_");
               includeVariable(token.image+"_");
               includeVariable(token.image);
               declaracao = declaracao +varaction(token.image);
               resposta = resposta + fimaction(token.image);
             }
           )*
         ]
         ")"
         {out.println("){\n\tTuple tuple=null;");}
         { initcomandos();
           out.println(declaracao);
         }
         "{"
         [(for_() | if_() | do_actions() ) ]
         "}"
         {
           out.println(resposta);
           endexception();variablesEnd();
           out.println("\t}\n");}
       )
   )
}
```

```
//--------------------------------------
// defines the MLA Modular Logic Action
void mla() :
{
   Token token,number;
   String declaracao="",token1="", resposta="",variaveis="",roda="";int contador;
}
{  {
     roda="p = new PrologInterface();\n\tcont = new ReturnJava(p);\n\t
           programa = new PRED_";
   }
   number = <NUMBER> "("  ")"
   { contador = Integer.parseInt(number.image); }
  "{"
  { variablesInit(); }
  token = <SYMBOL> "("
  { prologfile.print(token.image+"("); }
  { token1 = "parametro_0";
    roda = roda + token.image + "_" + number.image+"(";
  }
  { out.print("holoString " + token1);
    includeVariable(token1);
    declaracao = varprolog(token1);
    resposta = fimprolog(token1);
    variaveis = "Term " + token1 + "_prolog";
    roda = roda +token1+ "_prolog";
  }
  { for (int i=1;i<contador;i++)
    {
      token1 = "parametro_"+i;
      out.print(", holoString " + token1);
      includeVariable(token1);
      declaracao = declaracao +varprolog(token1);
      resposta = resposta +fimprolog(token1);
      variaveis = variaveis + "," + token1 + "_prolog";
      roda = roda +"," +token1 + "_prolog";
    }
  }

  //PARSER PARAMETERS
  { roda =roda +",cont);\n\t p.setPredicate(programa);\n\tp.call();" ;
    out.println(")\n\t{\n\t");
    out.println(variaveis+";");
    out.println("\tPrologInterface p;\n\tPredicate predicado;\n\tPredicate
                 cont;\n\tPredicate programa;");
    out.println(declaracao);
    out.println(roda);
    out.println("\n"+resposta);
  }
  // PARAMETERS END

  ( ( token = <SYMBOL> | token = <NUMBER> |token = <VARIABLE> | token = "("
      | token = ")" |token = ","|token = ">"|token = "<"|token = "["|token = "]"
      | token = <ARITH>
      | token ="."
      | token = <IS>
      | token = "|"
      | token = ":-"
    )
    {
      if(token.image.equals("is"))
        {prologfile.print(" "+token.image+" ");}
      else
      {
        prologfile.print(token.image);
        if(token.image.equals("."))
          {prologfile.print("\n");}
      }
    }
  )*
  "}"
  { out.println("\t}");
    prologfile.close();
    variablesEnd();
  }
}
```

```
//--------------------------------------
// The actions that can occur in the being
void do_actions():
{}
{
  (
     AccessHistory() | Comandos() | Chamadas() | Atribuicao()
  )
  [ "," (for_() | if_() | (do_actions()) ) ]
}

//--------------------------------------
// Defines the history
void History() :
{}
{
  "history"
  "{" (Clausulas())* "}"
}

// defines the content of the initial history
void Clausulas () :
{Token token1,token2;}
{
  (token1 = <SYMBOL> | token1 = <NUMBER>)
  { out.print ("\thistory.out (new Tuple (\"" + token1.image +"\""); }
  [
    <LPAREN>
    ( token2 = <SYMBOL> | token2 = <NUMBER> )
    {out.print(",\"" + token2.image + "\"" );}
    ( "," ( token2 = <SYMBOL> | token2 = <NUMBER>)
      {out.print(",\"" + token2.image + "\"");}
    )*
    <RPAREN>
  ]
  "." {out.println ("));");}
}

//---------------------------------------
// defines the list of variables
void VariableList():
{Token token1;}
{
   token1 = <VARIABLE>
   { out.print("String " + token1.image);
     includeVariable(token1.image);
   }
   ( "," token1 = <VARIABLE>
     { out.print(", String " + token1.image);
       includeVariable(token1.image);
     }
   )*
}

//---------------------------------------
void CDL() :
{}
{
  <SYMBOL>
  | "[" <SYMBOL> ("," <SYMBOL>)* "]"
  | <HOLO>
}

//---------------------------------------
void AIList() :
{}
{
  <SYMBOL> "/" <NUMBER> ("," <SYMBOL> "/" <NUMBER>)*
}
```

```
//-----------------------------------------------
// defines for
void for_():
{
   Token token1,token2,token3;
   boolean var=false;
}
{
   <FOR> token1 = <VARIABLE> ":=" token2 = <NUMBER> "to"
   ( token3 = <VARIABLE>
     { if(!CheckVariable(token3.image+"_integer"))
       {
         includeVariable(token3.image+"_integer");
         out.println("int "+token3.image+"_integer = Integer.parseInt("+
                     token3.image+");");
         var=true;
       }
       else
       {
         out.println(token3.image+"_integer = Integer.parseInt("+ token3.image+");");
         var=true;
       }
     }
     | token3 = <NUMBER>
   )
   "do" "{"
   { if(!CheckVariable(token1.image+"_integer"))
       {includeVariable(token1.image+"_integer");}
     if(!CheckVariable(token1.image))
       {
         includeVariable(token1.image);
         out.println("String "+token1.image+" = \"\";");
       }
     out.print("for (int "+ token1.image +"_integer = "+ token2.image+";"+
               token1.image+"_integer"+"<=");
     if(var)
       {out.print(token3.image+"_integer");}
     else
       {out.print(token3.image);}
     out.println(";"+token1.image+"_integer"+"++){\n\t"+token1.image+" =\"\"+"+
                  token1.image+"_integer;");}
     [ ( for_() | if_() | do_actions() ) ]
     "}"
     { out.println("}"); }
     [ ( for_() | if_() | do_actions() ) ]
}
```

Anexo 6 Arquivos de Definição da Linguagem na HoloJava

Este anexo contém as classes Java criadas para definição da Hololinguagem. No pacote de distribuição da HoloJava 1.0, estes arquivos ficam armazenados no diretório *holoj.lang* (veja a figura 5.8 e a tabela 5.5). Os arquivos são os seguintes:

- anexo 6.1: *BVector.java*
- anexo 6.2: *Being.java*
- anexo 6.3: *holoString.java*

Anexo 6.1 Arquivo *Bvector.java*

```
package holoj.lang;

import java.util.Vector;

public class BVector extends Vector {

  public synchronized Being getBeing (String name)
  {
    int count=0; boolean ok= true;
    Being being=null;
    if (this.isEmpty())
       return null;
    else{
      int size = this.size();
      while (ok)
      {
       being = (Being) elementAt(count);
       if (being.name.equals(name))
       {
        ok =false;
       }
       else {
       count ++;
       if (count== size)
       { return null;}
       }
      }
   }
   return (being);
}

  public synchronized Being removeBeing (String name)
  {
    int count=0; boolean ok= true;
    Being being=null;
    if (this.isEmpty())
       return null;
    else{
      int size = this.size();
      while (ok)
      {
       being = (Being) elementAt(count);
       if (being.name.equals(name))
       {
        ok =false;
        removeElementAt(count);
       }
       else {
       count ++;
       if (count== size)
       { return null;}
       }
      }
  }
   return (being);
}
}
```

Anexo 6.2 Arquivo *Being.java*

```
package holoj.lang;

import java.util.Vector;
import jada.*;

abstract public class Being {
  public Being father;
  public String name;
  public BVector son;
  public ObjectSpace history;
}
```

Anexo 6.3 Arquivo *holoString.java*

```
package holoj.lang;

public class  holoString {

        public String value;
        public boolean update,num;
        public holoString(String valor, boolean update)
        {
         this.value = valor;
         this.update = update;
         this.num = false;
        }

    public String toString ()
    {
     return this.value;
    }
}
```

Bibliografia

[ACM 89] ACM COMPUTING SURVEYS. **Parallel Processing and Paradigm of Software**. New York, v.21, n.3, September 1989. 510p.

[AIT 91] AÏT-KACI, Hassan. **Warren's Abstract Machine - A Tutorial Reconstruction**. Cambridge: MIT Press, 1991. 114p.

[ALV 97] ALVARES, Luis Otávio; SICHMAN, Jaime Simão. Introdução aos Sistemas Multiagentes. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA (JAI), 16., 1997, Brasília. **Proceedings...** Brasília: SBC, 1997. 37p.

[AMA 96] AMANDI, Analía; PRICE, Ana. A Linguagem OWB: Combinando Objetos e Lógica. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 1., 1996, Belo Horizonte. **Anais...** Belo Horizonte: SBC, 1996. 404p. p.305-318.

[AMA 97] AMANDI, Analía A. **Programação de Agentes Orientada a Objetos**. 1997. 208p. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[AMB 96] AMBRIOLA, Vincenzo; CIGNONI, Giovanni A.; SEMINI; Laura. A Proposal to Merge Multiple Tuple Spaces, Object Orientation and Logic Programming. **Computer Languages**, Elmsford, v.22, n.2/3, p.79-93, July/October 1996.

[AND 83] ANDREWS, Gregory R. Concepts and Notations for Concurrent Programming. **ACM Computing Surveys**, New York, v.15, n.1, p.3-43, March 1983.

[AND 91] ANDREWS, Gregory R. Paradigms for Process Interaction in Distributed Programs. **ACM Computing Surveys**, New York, v.23, n.1, p.49-90, March 1991.

[AND 92] ANDREWS, Gregory R.; OLSSON, Ronald A. **The SR Programming Language**. New York: The Benjamin/Cummings Publishing Company, 1992. 344p.

[APE 2001] APPELO – Ambiente de Programação Paralela em Lógica. Disponível em: <http://www.inf.ufrgs.br/procpar/opera/APPELO/>. Acesso em: novembro 2001.

[APT 98] APT; R. K. et al. Alma-0: An Imperative Language that Supports Declarative Programming. **ACM Transactions on Programming Languages and Systems**, New York, v.20, September 1998.

[AUG 84] AUGUSTSSON, L. A. Compiler for Lazy ML. In: ACM SYMPOSIUM ON LISP AND FUNCTIONAL PROGRAMMING, Austin, 1984. **Proceedings…** New York: ACM Press, 1984. p.218-227.

[AUI 2001] AUGUSTIN, Iara; YAMIN, Adenauer C.; BARBOSA, Jorge L V.; GEYER, Cláudio F. R. Requisitos para o Projeto de Aplicações Móveis Distribuídas. In: CONGRESO ARGENTINO DE LA CIENCIA DE LA COMPUTACION, CACIC, 8., 2001. **Proceedings…** Santa Cruz: [s.n.], 2001. 1 CD

[AUI 2002] AUGUSTIN, Iara; YAMIN, Adenauer C.; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Towards a Taxonomy for Mobile Applications with Adaptive Behavior. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS, PDCN, Innsbruck, Austria, 2002. **Proceedings…** Aceito, mas ainda não publicado.

[AZE 98] AZEVEDO, Silvana C.; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R.; CASTRO, Luis Fernando P. Integração ParTy-Granlog: Interpretação Abstrata Aplicada a Análise de Granulosidade de Programas em Lógica. In: CONGRESSO ARGENTINO DE CIÊNCIA DA COMPUTAÇÃO, CACIC, 4., 1998, Neuquen, Argentina. **Proceedings...** Neuquen: [s.n.], 1998.

[AZE 98a] AZEVEDO, Silvana C.; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Integração ParTy-Granlog: Interpretação Abstrata Aplicada a Paralelização de Programas em Lógica. In: OFICINA DE INTELIGÊNCIA ARTIFICIAL, 2., 1998, Pelotas. **Proceedings…** Pelotas: UCPel, 1998.

[AZE 99] AZEVEDO, Silvana C.; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Automatização da Análise Global no modelo Granlog. In: CONFRESSO LATINOAMERICANO DE INFORMÁTICA, 25., Asuncion, Paraguai. **Proceedings...** Asuncion: [s.n.], 1999. p.601-612.

[AZE 2001] AZEVEDO, Silvana C.; VARGAS, Patrícia .K.; BARBOSA, Jorge L. V.; YAMIN, Adenauer C.; GEYER, Cláudio F. R. DEPAnalyzer: um analisador estático de dependências para programas Java. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2., 2001. **Anais...** Pirenópolis: SBC, 2001. p.135-141.

[AZE 2001a] AZEVEDO, Silvana C.; VARGAS, Patrícia .K.; BARBOSA, Jorge L. V.; YAMIN, Adenauer C.; GEYER, Cláudio F. R. Análise de Dependências em Programas Java Seqüenciais. In: CONGRESO ARGENTINO DE LA CIENCIAS DE LA COMPUTACION (CACIC), 8., 2001. **Proceedings…** Santa Cruz: [s.n.], 2001. 1 CD

[BAC 78] BACKUS, John. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. **Communications of the ACM**, New York, v.21, n.8, p.613-641, August 1978.

[BAL 89] BAL, Henri E.; STEINER, Jennifer G.; TANENBAUM, Andrew S. Programming Languages for Distributed Computing Systems. **ACM Computing Surveys**, New York, v.21, n.3, p.261-322, September 1989.

[BAN 99] BANBARA, Mutsunori; TAMURA; Naoyuki. Translating a Linear Logic Programming Language into Java. In: WORKSHOP ON PARALLELISM AND IMPLEMENTATION TECHNOLOGY (CONSTRAINT) LOGIC PROGRAMMING LANGUAGES, 1999, Las Cruces. **Proceedings…** Las Cruces: New Mexico State University, 1999. p.19-39.

[BAR 93] BARBOSA, Jorge L. V. **Construção de Compiladores para Máquinas de Processamento Paralelo**. 1993. 73p. Monografia (Especialização em Engenharia de Software) – Escola de Informática, Universidade Católica de Pelotas, Pelotas.

[BAR 94] BARBOSA, Jorge L. V. **Análise da Granulosidade de Tarefas para Processamento Paralelo**. 1994. 93p. Trabalho Individual (CPGCC-UFRGS) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BAR 94a] BARBOSA, J.L.V.; WERNER, O.; GEYER, C. F. R. Automatic Granularity Analysis in Logic Programming. In: LOGIC PROGRAMMING WORKSHOP, WLP, 10., 1994, Zurich. **Proceedings...** Zurich: Institut für Informatik der Universität Zürich, 1994. 185p. p.85-88.

[BAR 95] BARBOSA, J.L.V.; GEYER, C. F. R. GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 10., 1995, Canela. **Anais...** Porto Alegre: SBC, 1995. 639p. p.61-75

[BAR 95a] BARBOSA, J. L. V.; GEYER, C. F. R. Integração OPERA-GRANLOG: Aplicação da Análise de Granulosidade na Execução Paralela de Programas em Lógica. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 22., 1995,Canela. **Anais...** Porto Alegre: SBC, 1995. 1482p. p.189-200

[BAR 96] BARBOSA, Jorge L. V. **GRANLOG**: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica. 167p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BAR 96a] BARBOSA, J. L. V.; GEYER, C. F. R. Análise de Grãos na Programação em Lógica. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 23., 1996, Recife. **Anais...** Recife: SBC, 1996. p.345-356

[BAR 96b] BARBOSA, J. L. V.; GEYER, C. F. R. Análise Global na Programação em Lógica. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 1., 1996, Belo Horizonte. **Anais...** Belo Horizonte: SBC, 1996. p.89-102

[BAR 97] BARBOSA, J. L. V.; GEYER, C. F. R. Análise de Complexidade na Programação em Lógica: Taxonomia, Modelo GRANLOG e Análise OU. In: CONGRESSO LATINO AMERICANO DE INFORMÁTICA, CLEI, 23., Valparaíso, Chile, 1997. **Anais...** Valparaíso: [s.n.], 1997.

[BAR 98] BARBOSA, Jorge L. V. **Paradigmas de Desenvolvimento de Sistemas Computacionais**. 1998. 49p. Trabalho Individual (CPGCC-UFRGS) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BAR 98a] BARBOSA, Jorge L. V.; GEYER, Cláudio F R. Taxonomia Multiparadigma. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 4., 1998, Neuquén. **Proceedings...** Neuquén: Universidad Nacional del Comahue, 1998. p.1162. Poster.

[BAR 99] BARBOSA, Jorge L. V. **Princípios do Holoparadigma**. 1999. 75p. Trabalho Individual (CPGCC-UFRGS) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BAR 99a] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Software Multiparadigma Distribuído. **Revista de Informática Teórica e Aplicada (RITA)**, Porto Alegre, v.6, n.2, p.67-87, dezembro 1999.

[BAR 2000] BARBOSA, Jorge L. V. **Software Multiparadigma Paralelo e Distribuído**. 2000. 104p. Exame de Qualificação (PPGC-UFRGS) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BAR 2000a] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Princípios do Holoparadigma. In: ARGENTINE SYMPOSIUM ON COMPUTING TECHNOLOGY, AST, 1., 2000, Tandil. **Proceedings...** Tandil: SADIO, 2000.

[BAR 2000b] BARBOSA, Jorge L. V.; VARGAS, Patrícia K.; GEYER, Cláudio F. R. DUTRA, Inês C. GRANLOG: An Integrated Granularity Analysis Model for Parallel Logic Programming**.** In: WORKSHOP ON PARALLELISM AND IMPLEMENTATION TECHNOLOGY (CONSTRAINT) LOGIC PROGRAMMING, 1., 2000, London. **Proceedings…** London: [s.n.], 2000.

[BAR 2000c] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Um Modelo Multiparadigma para Desenvolvimento de Software Paralelo e Distribuído. In: WORKSHOP ON HIGH PERFORMANCE COMPUTING SYSTEMS, WSCAD, 1., 2000, São Pedro. **Proceedings…** São Pedro: [s.n.], 2000.

[BAR 2000d] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Uma Taxonomia para Modelos Multiparadigma. In: ARGENTINE SYMPOSIUM ON COMPUTING TECHNOLOGY, AST, 1., 2000, Tandil. **Proceedings...** Tandil: SADIO, 2000.

[BAR 2000e] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Holoparadigma: Desenvolvimento de Software Multiparadigma Distribuído. In: CONGRESSO ARGENTINO DE CIÊNCIA DA COMPUTAÇÃO CACIC, Ushuaia, 2000. **Proceedings...** Ushuaia: [s.n.], 2000.

[BAR 2000f] BARBOSA, Jorge L. V.; VARGAS, Patrícia K.; GEYER, Cláudio F. R. Aplicação da Análise de Complexidade na Exploração de Paralelismo na Programação em Lógica**.** In: CONGRESSO ARGENTINO DE CIÊNCIA DA COMPUTAÇÃO, CACIC, Ushuaia, 2000. **Proceedings...** Ushuaia: [s.n.], 2000. Poster.

[BAR 2001] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Integrating Logic Blackboards and Multiple Paradigms for Distributed Software Development. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, Las Vegas, 2001. **Proceedings…** Las Vegas: CSREA Press, 2001. p.808-814.

[BAR 2001a] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Uma Linguagem Multiparadigma Orientada ao Desenvolvimento de Software Distribuído. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, SBLP, 5., 2001, Curitiba. **Proceedings…** Curitiba: SBC, 2001.

[BAR 2001b] BARBOSA, Jorge Luis Victória Barbosa et al. Using Mobility and Blackboards to Support a Multiparadigm Model Oriented to Distributed Processing. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 13., 2001, Pirenópolis, Brasil. **Proceedings…** Brasília: UNB, 2001. p.187-194.

[BAR 2001c] BARBOSA, Jorge Luis Victória; DU BOIS, André; PAVAN, Altino; GEYER, Cláudio Fernando Resin. HoloJava: Translating a Distributed Multiparadigm Language into Java. In: CONFERÊNCIA LATINOAMERICANA DE INFORMÁTICA, 27., 2001, Mérida, Venezuela. **Proceedings…** Mérida: Universidad de Los Andes, septiembre 2001. 1 CD

[BER 97] BERRY, Michael J. A.; LINOFF, Gordon. **Data Mining Techniques**: for marketing, sales and customer support. New York: John Wiley & Sons, 1997. 454p

[BIR 93] BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing**. Communications of the ACM**, New York, v.36, n.12, p.37-53, December 1993.

[BLU 94] BLUME, William et al. Automatic Detection of Parallelism – A Grand Challenge for High-Performance Computing. **IEEE Parallel and Distributed Technology: Systems and Applications**, New York, v.2, n.3, p.37-47, Fall 1994.

[BOH 80] BOHM, David. **A Totalidade e a Ordem Implicada - Uma Nova Percepção da Realidade**. São Paulo: Cultrix, 1980. 292p.

[BOH 86] BOHM, David. A Ordem Implícita e a Ordem Superimplícita. In: WEBER, Renée (Ed.). **Diálogos com Cientistas e Sábios - A Busca da Unidade**. São Paulo: Cultrix, 1986. p.43-73.

[BOH 86a] BOHM, David. Criatividade: a assinatura da natureza. In: WEBER, Renée (Ed.). **Diálogos com Cientistas e Sábios - A Busca da Unidade**. São Paulo: Cultrix, 1986. p.121-133.

[BOS 93] BOSSCHERE, K.; TARAU, P. Blackboard Communication in Logic Programming. In: INTERNATIONAL CONFERENCE PARCO, 1993, Grenoble. **Proceedings…** Grenoble: [s.n.], 1993. p.257-264

[BOS 96] BOSSCHERE, K.; TARAU, P. Blackboard-based Extensions in Prolog. **Software – Practice and Experience**, New York, v.26, n.1, p.49-69, January 1996.

[BRA 97] BRADSHAW, Jeffrey M. **Software Agents**. London: MIT Press, 1997. 480p.

[BRI 90] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Opera**: a Parallel Prolog System and its Implementation on Supernode. Grenoble: Laboratoire de Genie Informatique de Grenoble/CAP-Gemini-Innovation, 1990. Technical report.

[BRI 90a] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Scheduling of OR-parallel Prolog on Scalable, Reconfigurable, Distributed-Memory Multiprocessor**. Grenoble: Laboratoire de Genie Informatique de Grenoble/CAP-Gemini-innovation, 1990. Technical Report.

[BRI 91] BRIAT, J.; FAVRE, M.; GEYER, C; CHASSIN de Kergommeaux. Scheduling of OR-parallel Prolog on a scalable reconfigurable distributed memory multiprocessor. In: PARALLEL ARCHITECTURES AND LANGUAGES EUROPE, PARLE, 1991. **Proceedings...** Berlin: Springer-Verlag, 1991. p.385-402. (Lecture Notes in Computer Science, v. 506).

[BRJ 96] BRIOT, Jean-Pierre; GUERRAOUI, Rachid. **A Classification of Various Approaches for Object-Based Parallel and Distributed Programming**. Disponível em: <http://lsewww.epfl.ch/~rachid/papers/surv96.ps/>. Acesso em: julho 1996.

[BRJ 98] BRIOT, Jean-Pierre; GUERRAOUI, Rachid; LÖHR, Klaus-Peter. Concurrency and Distribution in Object-Oriented Programming. **ACM Computing Surveys**, New York, v.30, n.3, p.291-329, September 1998.

[BRO 91] BROGI, Antônio; CIANCARINI, Paolo. The Concurrent Language, Shared Prolog. **ACM Transactions on Programming Languages and Systems**, New York, v.13, n.1, p.99-123, January 1991.

[BRP 2001] BRAND, P.; FRANZEN, N.; KLINTSKOG, E.; HARIDI, S. **A Platform for Constructing Virtual Spaces**. Disponível em: <http://www.mozart-oz.org/papers/abstracts/virtual.html/>. Acesso em: novembro 2001.

[BUG 94] BUGLIESI, Michele; LAMMA Evelina; MELLO, Paola. Modularity in Logic Programming. **Journal of Logic Programming**, New York, v.19/20, p.443-502, May/July 1994.

[BYT 95] A Brief History of Programming Languages. **Byte**, Peterborough, v.20, n.9, p.121-122, September 1995.

[CAB 97] CABILLIC, Gilbert; PUAUT, Isabelle. Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations. **Journal of Parallel and Distributed Computing**, New York, v.40, n.1, p.65-80, January 1997.

[CAD 2000] CABEZA, D.; HERMENEGILDO, M. A New Module System For Prolog. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LOGIC, 2000, London. **Proceedings…** Berlin: Springer-Verlag, July 2000. p.131-148.

[CAL 97] CARDELLI, Luca. **Mobile Ambient Synchronization**. Cambridge: Digital Equipment Corporation Systems Research Center, July 1997. (Technical report, SRC Technical Note 1997-013).

[CAL 98] CARDELLI, Luca; GORDON, Andrew D. Mobile Ambients. In: NIVAT, Maurice (Ed.). **Foundations of Software Science and Computational Structures**. Berlin: Springer-Verlag, 1998. p.140-155. (Lecture Notes in Computer Science, v. 1378).

[CAL 99] CARDELLI, Luca. Wide Area Computation. In: INTERNATIONAL COLLOQUIUM IN AUTOMATA, LANGUAGES AND PROGRAMMING, ICALP, 26., 1999. **Proceedings…** Berlin: Springer-Verlag, 1999. p.10-24. (Lecture Notes in Computer Science, v. 1644).

[CAL 99a] CARDELLI, Luca; GORDON, Andrew D. Types for Mobile Ambients. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 26., 1999. **Proceedings…** New York: ACM Press, 1999. p.79-92.

[CAL 99b] CARDELLI, Luca. Abstractions for Mobile Computation. In: VITEK, Jan; JENSEN, Christian (Ed.). **Secure Internet Programming: Security Issues for Mobile and Distributed Objects**. Berlin: Springer-Verlag, 1999. p.51-94. (Lecture Notes in Computer Science, v. 1603).

[CAL 2000] CARDELLI, Luca. Mobility and Security. In: BAUER, Friedrich L.; STEINBRÜGGEN, Ralf (Ed.). **Foundations of Secure Computation**. Marktoberdorf: IOS Press, 2000. p.3-37.

[CAL 2000a] CARDELLI, Luca; GHELLI, Giorgio; GORDON; Andrew D. Ambient Groups and Mobility Types. In: INTERNATIONAL CONFERENCE IFIP TCS, 2000, Sendai, Japan. **Proceedings…** Berlin: Springer-Verlag, 2000. p.333-347. (Lecture Notes in Computer Science, v. 1872).

[CAL 2001] CARDELLI, Luca; GORDON, Andrew D. Logical Properties of Name Restriction. In: INTERNATIONAL CONFERENCE ON TYPED LAMDA CALCULI AND APPLICATION, TCLA, 2001, Krakow, Poland. **Proceedings…** Berlin: Springer-Verlag, 2001. p.46-60. (Lecture Notes in Computer Science, v. 2044).

[CAM 99] CARRO, M.; HERMENEGILDO, M. Concurrency in Prolog Using Threads and a Shared Database. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 1999. **Proceedings…** Cambridge: MIT Press, November 1999. p.320-334.

[CAP 82] CAPRA, Fritjof. **Ponto de Mutação - A Ciência, a Sociedade e a Cultura Emergente**. São Paulo: Cultrix, 1982. 447p.

[CAR 86] CARRIERO, Nicholas; GELERNTER, David. Linda and Friends. **Computer**, New York, v.19, n.8, p.26-34, August 1986.

[CAS 87] CASANOVA, M. A.; GIORNO, F. A. C.; FURTADO, A. L. **Programação em Lógica e a Linguagem Prolog**. São Paulo: E. Blücker, 1987. 461p.

[CHA 97] CHAKRAVARTY, Manuel M. T.; LOCK, Hendrik C. R. Towards the Uniform Implementation of Declarative Languages. **Computer Languages**, Elmsford, v.23, n.2-4, p.121-160, July-December 1997.

[CHI 91] CHIN, Roger S.; CHANSON, Samuel T. Distributed Object-Based Programming Systems. **ACM Computing Surveys**, New York, v.23, n.1, p.91-124, March 1991.

[CIA 96] CIAMPOLINI, A.; LAMMA, E.; STEFANELLI, C; MELLO, P. Distributed Logic Objects. **Computer Languages**, Elmsford, v.22, n.4, p.237-258, December 1996.

[CIP 92] CIANCARINI, Paolo. Parallel Programming with Logic Languages: A Survey. **Computer Languages**, Elmsford, v.17, n.4, p.213-239, October 1992.

[CIP 94] CIANCARINI, Paolo. Distributed Programming with Logic Tuple Spaces. **New Generating Computing**, Berlin, v.12, n.3, p.251-283, 1994.

[CIP 2001] CIANCARINI, Paolo; ROSSI, D. **JADA**: A Coordination Toolkit for Java. Disponível em: <http://www.cs.unibo.it/~rossi/jada/>. Acesso em: novembro 2001.

[COM 86] COMPUTER. **Parallel Processing and Software's Paradigm**. New York: IEEE, v.19, n.8, August 1986.

[COM 91] COMMUNICATIONS OF THE ACM. **LISP – Adapting to the environment**. New York: ACM, v.34, n.9, September 1991.

[COM 94] COMMUNICATIONS OF THE ACM. **Intelligent Agents**. New York: ACM, v.37, n.7, July 1994. 170p.

[COS 94] COSTA, Antônio Carlos da Rocha. On Entering an Open Society. In: SIMPÓSIO BRASILEIRO DE INTELIGÊNCIA ARTIFICIAL, 11., 1994, Fortaleza. **Anais...** Fortaleza: SBC, 1994. p.535-546.

[DAV 93] DAVISON, Andrew. A Survey of Logic Programming-Based Object-Oriented. In: AGHA, G.; WEGNER, P.; YONEZAWA, A. (Ed.). **Research Directions in Concurrent Object-Oriented Programming**. Cambridge: MIT Press, 1993. p.42-106

[DER 96] DERANSART, Pierre. **Prolog**: the standard – reference manual. Berlin: Springer-Verlag, 1996.

[DUB 2001] DU BOIS, André Rauber; BARBOSA, Jorge Luis Victória Barbosa; GEYER, Cláudio Fernando Resin. Adding Functional Programming into the Holo Language. In: FUNCTIONAL AND (CONSTRAINT) LOGIC PROGRAMMING, WFLP, 10., 2001, Kiel, Germany. **Proocedings…** Kiel: Christian-Albrechts-Universität, September 2001. p.45-58.

[DUB 2001a] DU BOIS, André Rauber; COSTA, Antônio Carlos da Rocha. Distributed Execution of Functional Programs using the JVM. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED SYSTEMS THEORY, EUROCAST, 8., 2001, Las Palmas, Spain. **Proocedings…** Las Palmas: Universidad de Las Palmas de Gran Canaria, 2001.

[DUT 99] DUTRA, Inês de C.; COSTA, Vítor S.; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, SBAC-PAD, 11., 1999, Natal. **Proceedings...** Natal: SBC, 1999. p. 248- 254.

[EIN 81] ENSTEIN, Albert. **Como Vejo o Mundo**. Rio de Janeiro: Nova Fronteira, 1981. 213p.

[EIN 95] ENSTEIN, Albert. Pan-Europa. In: MOREIRA, Ildeu C.; VIDEIRA, Antonio A. P. (Ed.). **Einstein e o Brasil**. Rio de Janeiro: UFRJ, 1995. p.71-75

[FEA 99] FERSCHA, Alois; JOHNSON, James. Distributed Interaction in Virtual Spaces. In: INTERNATIONAL WORKSHOP ON DISTRIBUTED INTERACTIVE SIMULATION AND REAL-TIME APPLICATIONS, 3., 1999, Maryland. **Proceedings...** [S.l.:s.n.], 1999.

[FER 99] FERRARI, Débora N. **Um Estudo sobre Mobilidade em Sistemas Distribuídos**. 1999. 50p. Trabalho Individual (PPGC-UFRGS) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[FER 99a] FERRARI, Débora N.; VARGAS, Patrícia K.; GEYER, Cláudio F. R.; BARBOSA, Jorge L. V. Modelo de Integração PloSys-GRANLOG: aplicação da análise de granulosidade na exploração do paralelismo OU. In: CONGRESSO LATINOAMERICANO DE INFORMÁTICA, 25., 1999, Asuncion. **Proceedings...** Asuncion: [s.n.], 1999. p.911-922.

[FER 2001] FERRARI, Débora N. **Um Modelo de Replicação em Ambientes que Suportam Mobilidade**. 2001. 88p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[FER 2001a] FERRARI, Débora N.; MANGAM, Patrícia K. V.; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Um Modelo de Replicação em Ambientes que Suportam Mobilidade de Objetos Distribuídos. In: CONFERÊNCIA LATINOAMERICANA DE INFORMÁTICA, 27., 2001, Mérida, Venezuela. **Proceedings…** Mérida: Universidad de Los Andes, Septiembre 2001. 1 CD

[FIL 85] FILHO, Edgard de Alencar. **Teoria Elementar dos Conjuntos**. São Paulo: Nobel, 1985. 324p.

[FLA 2000] FLANAGAN, David. **Java**: O Guia Essencial. Rio de Janeiro: Campus, 2000. 718p.

[FRE 99] FREEMAN, Eric; HUPFER, Susanne; ARNOLD; Ken. **JavaSpaces Principles, Patterns and Practice**. New York: Addison-Wesley, 1999. 344p.

[FRI 92] FRIEDMAN, Linda W. From Babbage to Babel and Beyond: A Brief History of Programming Languages. **Computer Languages**, Elmsford, v.17, n.1, p. 1-17, 1992.

[FUR 98] FURLAN, José Davi. **Modelagem de Objetos através da UML**. São Paulo, Makron Books, 1998. 329p.

[GAR 95] GARLAN, D. et al. Research Directions in Software Engineering. **ACM Computing Surveys**, New York, v.27, n.2, p.257-276, June 1995.

[GEY 92] GEYER, Cláudio; YAMIN, Adenauer; WERNER, Otilia. Projeto OPERA: Um Modelo E/OU para Prolog. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 1992. **Proceedings…** [S.l.]: SBC, 1992. p.269-281.

[GEY 99] GEYER, Cláudio F. R.; BARBOSA, Jorge L. V. et al. The APPELO Project - Parallel Environment for Logic Programming. In: PROTEM-CC PROJECTS EVALUATION WORKSHOP (CNPq), 1999. **Proceedings…** [S.l.]: CNPq, 1999. p.421-454.

[GEY 2001] GEYER, Cláudio F. R. et al. Sistemas Distribuídos. In: ERAD – ESCOLA REGIONAL DE ALTO DESEMPENHO, 1., 2001. **Proceedings…** Gramado: SBC, 2001. p.195-204.

[GHE 98] GHEZZI, Carlo; JAZAYERI, Mehdi. **Programming Language Concepts**. New York: John Wiley & Sons, 1998. 427p.

[GLA 97] GLASER, Norbert; MORIGNOT, Philippe. The Reorganization of Societies of Autonomous. In: MAAMAW, 8., 1997. **Proceedings…** Berlin: Springer-Verlag, 1997. p.98-111. (Lecture Notes in Computer Science, v. 1237).

[GOD 2001] GODIGITAL Tecnologia e Participações Ltda. Disponível em: http://www.godigital.com.br/. Acesso em: novembro 2001.

[GUP 93] GUPTA, G.; JAYARAMAN, B. AND-OR Parallelism on Shared-Memory Multiprocessors. **Journal of Logic Programming**, New York, v.17, n.1, p.59-89, October 1993.

[HAI 86] HAILPERN, Brent. Multiparadigm Research: A Survey of Nine Projects. **IEEE Software**, New York, v.3, n.1, p. 70-77, January 1986.

[HAJ 96] HARDWICK, Jonathan; SIPELSTEIN, Jay. **Java as an Intermediate Language**. Pittsburgh: School of Computer Science, Carnegie Mellon University, August 1996. (Technical report, CMU-CS-96-161)

[HAN 94] HANUS, Michael. The Integration of Functions into Logic Programming from Theory to Practice. **Journal of Logic Programming**, New York, v.19/20, p.583-628, May/July 1994.

[HAN 97] HANUS, Michael. Lazy Narrowing with Simplification. **Computer Languages**, Elmsford, v.23, n.2-4, p.61-85, July-December 1997.

[HAN 99] HANUS, Michel. SANDRE, R. An Abstract Machine for Curry and its Concurrent Implementation in Java. **Journal of Functional and Logic Programming**, New York, v.6, 1999

[HAR 98] HARIDI, Seif et al. Programming Languages for Distributed Applications. **New Generating Computing**, Berlin, v.16, n.3, p.223-261, 1998.

[HAR 99] HARIDI, Seif et al. Efficient Logic Variables for Distributed Computing. **ACM Transactions on Programming Languages and Systems**, New York, v.21, n.3, p.569-626, May 1999.

[HAR 2001] HARIDI, Seif; FRANZÉN, Nils. **Tutorial of Oz**. Disponível em: http://mozart-oz.org/documentation/tutorial/index.html/. Acesso em: novembro 2001.

[HAS 2000] HASSELBRING, Wilhelm. Programming Languages and Systems for Prototyping Concurrent Applications. **ACM Computing Surveys**, New York, v.32, n.1, p.43-79, March 2000.

[HEN 97] HENZ, Martin. **Objects in Oz**. 1997. 199p. PhD Thesis, Universität des Saarlandes, Saarbrüchen.

[HIR 98] HIRANO, Satoshi; YASU, Yoshiji; IGARASHI, Hirotaka. Performance Evaluation of Popular Distributed Technologies for Java. In: WORKSHOP ON JAVA FOR HIGH-PERFORMANCE NETWORK COMPUTING, New York, 1998. **Proceedings…** New York: ACM Press, 1998.

[HOL 2001] HOLOPARADIGMA: Software Multiparadigma Distribuído. Disponível em: <http://www.inf.ufrgs.br/~holo/>. Acesso em: novembro 2001.

[HOR 2001] HORB Pages. Disponível em: <http://horb.a02.aist.go.jp/horb/>. Acesso em: novembro 2001.

[HUD 2001] HUDAK, P.; PETERSON, J.; FASEL, J. H. **A Gentle Introduction to Haskell – Version 1.4**. Disponível em: <http://www.haskell.org/>. Acesso em: novembro 2001.

[IEE 86] IEEE SOFTWARE. **Multiparadigm Languages and Environments**. New York: IEEE, January 1986.

[IEE 95] IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. **Software Architecture**. New York: IEEE, v.21, n.4, April 1995.

[IEE 98] IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. **Mobility and Network-Aware Computing**. New York: IEEE, v.24, n.5, May 1998.

[JAN 93] JANSON, Sverker; MONTELIUS, Johan; HARIDI, Seif. Ports for Objects in Concurrent Logic Programs. In: AGHA, G.; WEGNER, P.; YONEZAWA, A. (Ed.). **Research Direction in Concurrent Object-Oriented Programming**. Cambridge: MIT Press, 1993. p.211-231

[JAN 94] JANSON, Sverker; HARIDI, Seif. An Introduction to AKL: A Multi-Paradigm Programming Language. In: CONSTRAINT PROGRAMMING, Parnu, Estonia, 1994. **Proceedings…** Berlin: Springer-Verlag, 1994. p.411-443. (NATO Advanced Science Institute Series, v. 131).

[JAV 2001] JAVACC – The Java Parser Generator. Disponível em: <http://www.webgain.com/products/java_cc/>. Acesso em: novembro 2001.

[JEN 88] JENSEN, Kathleen; WIRTH, Niklaus. **Pascal ISO**: Manual do Usuário e Relatório. Rio de Janeiro: Campus, 1988. 262p.

[JON 92] JONES, Peyton S. L.; LESTER, D. **Implementing Functional Languages. A Tutorial**. New Jersey: Prentice-Hall, 1992.

[JOU 97] JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING. **Workstation Clusters and Network-Based Computing**. New York: Academic Press, v.40, n.1, January 1997.

[KEL 2000] KELLERMAN, Gustavo A; BARBOSA, Jorge L. V.; VARGAS, Patrícia K.; GEYER, Cláudio F. R. CALEBE: Uma Máquina Virtual Paralela com Suporte à Linguagens Multiparadigma. In: CONGRESSO ARGENTINO DE CIÊNCIA DA COMPUTAÇÃO, CACIC, Ushuaia, 2000. **Proceedings…**Ushuaia: SADIO, 2000. Poster.

[KER 94] KERGOMMEAUX, Jacques Chassin; CODOGNET, Philippe. Parallel Logic Programming Systems. **ACM Computing Surveys**, New York, v.26, n.3, p.295-336, September 1994.

[KOW 79] KOWALSKI, Robert. **Logic for Problem Solving**. New York: Elsevier, 1979.

[KOW 79a] KOWALSKI, Robert. Algorithms = Logic + Control. **Communications of the ACM**, New York, v.22, n.7, p.424-436 July 1979.

[KUH 70] KUHN, Thomas S. **A Estrutura das Revoluções Científicas**. São Paulo: Perspectiva, 1970. 257p.

[KUH 77] KUHN, Thomas S. Reconsiderações Acerca dos Paradigmas. In: KUHN, T. (Ed.). **A Tensão Essencial**. Lisboa: Ed. 70, 1977. p.353-382

[LAN 98] LANGE, Danny B. Mobile Objects and Mobile Agents: The Future of Distributed Computing? In: JUL, Eric (Ed.). **Object-Oriented Programming.** Berlin: Springer-Verlag, 1998. p.1-12. (Lecture Notes in Computer Science, v. 1445).

[LAR 98] LARMAN, Craig. **Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design**. New Jersey: Prentice-Hall, 1998. 507p.

[LEA 2001] LEA, Doug. **Objects in Groups**. Disponível em: <http://gee.cs.oswego.edu/dl/groups/>. Acesso em: novembro 2001.

[LIA 90] LIANG, Luping; CHANSON, Samuel T.; NEUFELD, Gerald W. Process Group and Group Communications: Classifications an Requirements. **Computer**, New York, v.23, n.2, p.56-66, February 1990.

[LEE 97] LEE, J. H. M.; PUN, P. K. C. Object Logic Integration: A Multiparadigm Design Methodology and a Programming Language. **Computer Languages**, Elmsford, v.23, n.1, p.25-42, April 1997.

[LIK 89] LI, Kai; HUDAK, Paul. Memory Coherence in Shared Virtual Memory Systems. **ACM Transactions on Computer Systems**, New York, v.7, n.4, p.321-359, November 1989.

[LIP 78] LIPSCHUTZ, Seymour. **Teoria dos Conjuntos**. São Paulo: McGraw-Hill, 1978. 337p.

[LPA 2001] LOGIC Programming Associates. Disponível em: <http://www.lpa.co.uk/>. Acesso em: novembro 2001.

[MAL 2001] MALACARNE, Juliano. **Ambiente Visual para Programação Distribuída em Java.** 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[MEY 87] MEYER, Bertrand. Reusability: The Case for Object-Oriented Design. **IEEE Software**, New York, v.4, n.2, p.50-63, March 1987.

[MEY 96] MEYER, Bertrand. The many faces of inheritance: A taxonomy of taxonomy. **Computer**, New York, v.29, n.5, 105-110, May 1996.

[MEY 98] MEYER, Bertrand. The Future of Object Technology. **Computer**, New York, v.31, n.1, 140-141, January 1998.

[MEY 99] MEYER, Bertrand. On To Components. **Computer**, New York, v.32, n.1, p.139-140, January 1999.

[MUL 95] MULLER, Martin; MULLER, Tobias; ROY, Peter V. Multiparadigm Programming in Oz. In: SMITH, Donald; RIDOUX, Olivier; ROY, Peter V. (Ed.). **Visions for the Future of Logic Programming**: Laying the Foundations for a Modern Sucessor of Prolog. Portland, Oregon: [s.n.], 1995.

[NGK 95] NG, K. W.; LUK, C. K. I^+: A Multiparadigm Language for Object-Oriented Declarative Programming. **Computer Languages**, Elmsford, v.21, n.2, p. 81-100, July 1995.

[NIE 93] NIERSTRASZ, Oscar. Composing Active Objects. In: AGHA, G.; WEGNER, P.; YONEZAWA, A. (Ed.). **Research Directions in Concurrent Object-Oriented Programming**. Cambridge: MIT Press, 1993. p.151-171

[NUN 98] NUNES, Raul Ceretta. **Programação Orientada a Grupos: o Ponto de Vista das Aplicações**. 1998. 45p. Trabalho Individual (PPGC-UFRGS) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[OMI 94] OMICINI, Andrea; NATALI, Antonio. Object-Oriented Computations in Logic Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 8., 1994, Bologna, Italy. **Proceedings...** Berlin: Springer-Verlag, 1994. p.194-212. (Lecture Notes in Computer Science, v.821).

[OPE 2001] OPERA – Prolog Paralelo. Disponível em: <http://www.inf.ufrgs.br/procpar/opera/OPERA/index.html/>. Acesso em: novembro 2001.

[ORT 85] ORTEGA y GASSEY, José. **Europa y la Idea de Nacion**. Madri: Alianza Editorial, 1985. 215p.

[ORT 87] ORTEGA y GASSEY, José. **A Rebelião das Massas**. São Paulo: Martins Fontes, 1987. 258p.

[ORT 92] ORTEGA y GASSEY, José. **Mision de la Universidad**. Madri: Alianza Editorial, 1992. 238p.

[OSS 99] OSSOWSKI, Sascha. **Co-ordination in Artificial Agent Societies: Social Structure and Its Implications for Autonomous Problem-Solving Agents**. Berlin: Springer-Verlag, 1999. 221p. (Lecture Notes in Computer Science, v.1535).

[PET 96] PETTOROSSI, Alberto; PROIETTI, Maurizio. Rules and Strategies for Transforming Functional and Logic Programs. **ACM Computing Surveys**, New York, v.28, n.2, p.360-414, March 1992

[PFL 97] PFLEGER, Karl; HAYES-ROTH, Barbara. **An Introduction to Blackboard-Style Systems Organization**. Stanford: Computer Science Department, Stanford University, July 1997. Technical report.

[PIN 99] PINEDA, A.; HERMENEGILDO, M. **O'CIAO**: An Object Oriented Programming Model Using CIAO Prolog. Madrid: Facultad de Informática, UPM, July 1999. Technical report.

[PLA 91] PLACER, John. The Multiparadigm Language G. **Computer Languages**, Elmsford, v.16, n.3/4, p.235-258, 1991.

[PLJ 2001] PROGRAMMING Languages for the Java Virtual Machine. Disponível em: <http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html>. Acesso em: novembro 2001.

[PRE 2000] PRETZ, Eduardo. **Desenvolvendo Padrões de Projeto para o Holoparadigma**. 2001. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Escola de Informática, Universidade Católica de Pelotas, Pelotas.

[PRJ 96] PROTIC, Jelica; TOMASEVIC, Milo; MILUTINOVIC, Veljko. Distributed Shared Memory: Concepts and Systems. **IEEE Parallel and Distributed Technology: Systems and Applications**. New York, v.4, n.2, p.63-79, Summer 1996.

[PRO 99] PROCEEDINGS of the IEEE. **Distributed Shared Memory Systems**.New York: IEEE, v.87, n.3, March 1999.

[PRO 2001] PROLOG++. Disponível em: <http://www.lpa.co.uk/ppp.htm>. Acesso em: novembro 2001.

[PRO 2001a] PROLOG Cafe Pages. Disponível em: <http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/>. Acesso em: novembro 2001.

[ROB 92] ROBINSON, J. A. Logic and Logic Programming. **Communications of the ACM**, New York, v.35, n.3, p.40-65, March 1992.

[ROS 96] ROSENBERG, Jonathan B. **How Debuggers Work**. New York: John Wiley & Sons, 1996. 256p.

[ROY 97] ROY, Peter V. et al. Mobile Objects in Distributed Oz. **ACM Transactions on Programming Languages and Systems**, New York, v.19, n.5, p.804-851, September 1997.

[ROY 2001] ROY, Peter V.; HARIDI, Seif; BRAND, Per. **Distributed Programming in Mozart – A Tutorial Introduction**. Disponível em: <http://mozart-oz.org/documentation/dstutorial/>. Acesso em: novembro 2001.

[SEB 99] SEBESTA, Robert W. **Concepts of Programming Languages**. New York: Addison Wesley, 1999. 670p.

[SES 96] SESSIONS, Roger. **Object Persistence**: beyond object-oriented databases. New Jersey: Prentice-Hall, 1996. 250p.

[SHA 95] SHAW, M. et al. Abstractions for Software Architecture and Tools to Support Them. **IEEE Transactions on Software Engineering**, New York, v.21, n.4, p.314-335, April 1995.

[SHA 96] SHAW, M.; GARLAN, D. **Software Architecture**: Perspectives on an Emerging Discipline. New Jersey: Prentice-Hall, 1996. 242p.

[SHO 93] SHOHAM, Y. Agent-oriented programming. **Artificial Intelligence**, Amsterdam, v.60, n.1, p.51-92, March 1993.

[SHS 89] SHATZ, Sol M.; WANG Jia-Ping. **Tutorial**: Distributed-Software Engineering**.** New York: IEEE Computer Society Press, 1989. 279p.

[SIC 95] SICSTUS Prolog. Advanced Prolog Technology – User's Manual. Kista: Swedish Institute of Computer Science, 1995. 378p.

[SIL 2001] SILVA, Edson J.; AUGUSTIN, Iara; YAMIN, Adenauer C.; BARBOSA, Jorge L V.; GEYER, Cláudio F. R. Hierarquia de Gerenciamento de Redes com Componentes Móveis. In: CONGRESO ARGENTINO DE LA CIENCIAS DE LA COMPUTACION, CACIC, 8., 2001. **Proceedings…** Santa Cruz: [s.n.], 2001. 1 CD

[SIU 2001] SILVA, Luciano; YAMIN, Adenauer C.; AUGUSTIN, Iara; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Mecanismos de Suporte ao Escalonamento em Sistemas com Objetos Distribuídos em Java. In: CONGRESO ARGENTINO DE LA CIENCIAS DE LA COMPUTACION, CACIC, 8., 2001. **Proceedings…** Santa Cruz: [s.n.], 2001. 1 CD

[SKI 98] SKILLICORN, David B.; TALIA, Domenico. Models and Languages for Parallel Computation. **ACM Computing Surveys**, New York, v.30, n.2, p.123-169, June 1998.

[SMO 95] SMOLKA, Gert. The Oz Programming Model. In: COMPUTER SCIENCE TODAY, 1995. **Proceedings...** Berlin: Springer-Verlag, 1995. p. 324-343. (Lecture Notes in Computer Science, v.1000).

[SMO 95a] SMOLKA, Gert; SCHULTE, Christian; VAN ROY, Peter. **PERDIO – Persistent and Distributed Programming in Oz**. 1995. Disponível em: <http://www.sics.se/~seif>. Acesso em: jul. 1999.

[SOA 2000] SOARES, Leonardo Campos. **Princípios de uma ferramenta CASE para o Holoparadigma**. 2000. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Escola de Informática, Universidade Católica de Pelotas, Pelotas.

[STE 86] STERLING, L.; SHAPIRO, E. **The Art of Prolog**. Cambridge: MIT Press, 1986. 437p.

[TAL 91] TALBOT, Michael. **O Universo Holográfico**. São Paulo: Best Seller, 1991. 390p.

[TAN 95] TANENBAUM, Andrew S. **Distributed Operating Systems**. New Jersey: Prentice-Hall, 1995. 614p.

[TAR 93] TARAU, P.; BOSSCHERE, K.; DAHL, V.; ROCHEFORT, S. LogiMOO: an Extensible Multi-User Virtual World with Natural Language Control. **Journal of Logic Programming**, New York, v.12, p.1-199, 1993.

[TAR 99] TARAU, P.; BOSSCHERE, K.; DAHL, V.; ROCHEFORT, S. LogiMOO: an Extensible Multi-User Virtual World with Natural Language Control. **Journal of Logic Programming**, v.38, n.3, p.331-353, March 1999.

[TAR 2001] TARAU, P. **Jinni**: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. Disponível em: <http://www.cs.unt.edu/~tarau/research/99/jpaper.html/>. Acesso em: novembro 2001.

[TRI 98] TRINDER, P. W.; HAMMIND, K. LOIDL. H.W.; JONES, P. S. L. Algorithm+Strategy = Parallelism. **Journal of Functional Programming**, New York, v.8, n.2, p.23-60, January 1998.

[VAR 95] VARGAS, Patrícia K.; GEYER, Cláudio. **Implementação de um Analisador de Granulosidade para Prolog**. 1995. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[VAR 95a] VARGAS, Patrícia K.; BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. Protótipo GRANLOG: Implementação de um Analisador de Granulosidade para Prolog no Projeto OPERA. In: SALÃO DE INICIAÇÃO CIENTÍFICA, 7., 1995. **Livro de resumos**. Porto Alegre: UFRGS, 1995.

[VAR 2000] VARGAS, Patrícia K.; BARBOSA, Jorge L. V.; FERRARI, Débora N. GEYER, Cláudio F. R.; CHASSIN, J. Distributed OR Scheduling with Granularity Information**.** In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, SBAC-PAD, 2000, São Pedro. **Proceedings...** São Pedro: SBC, 2000.

[VOY 2001] VOYAGER Pages. Disponível em: <http://www.objectspace.com/products/voyager/>. Acesso em: novembro 2001.

[VRA 95] VRANES, Sanja; STANOJEVIC, Mladen. Integrating Multiple Paradigms within the Blackboard Framework. **IEEE Transactions on Software Engineering**, New York, v.21, n.3, p.244-262, March 1995.

[WAK 97] WAKELING, D. A Haskell to Java Virtual Machine Code Compiler. In: INTERNATIONAL WORKSHOP ON THE IMPLEMENTATION OF FUNCTIONAL LANGUAGES, IFL, 9., 1997, St Andrews, Scotland. **Proceedings…** Berlin: Springer-Verlag, 1998. p. 39-52 (Lecture Notes in Computer Science, v.1467).

[WAK 98] WAKELING, D. Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine. In: PRINCIPLES OF DECLARATIVE PROGRAMMING, PLIP, 1998, Pisa, Italy. **Proceedings…** Berlin: Springer-Verlag, 1998. p.335-352

[WAK 98a] WAKELING, D. Compiling Lazy Functional Programs for the Java Virtual Machine. **Journal of Functional Programming**, New York, v.1, n.1, January 1998.

[WEB 86] WEBER, Renée. **Diálogos com Cientistas e Sábios - A Busca da Unidade**. São Paulo: Cultrix, 1986. 302p.

[WIA 92] WYATT, Barbara B.; KAVI, Krishna; HUFNAGEL, Steve. Parallelism in Object-Oriented Languages: A Survey. **IEEE Software**, New York, v.9, n.6, p.56-66, November 1992.

[WEG 93] WEGNER, Peter. Tradeoffs between Reasoning and Modeling. In: AGHA, G.; WEGNER, P.; YONEZAWA, A. (Ed.). **Research Direction in Concurrent Object-Oriented Programming**. Cambridge: MIT Press, 1993. p.22-41

[WEG 97] WEGNER, Peter. Why interaction is more powerful than algorithms. **Communications of the ACM**, New York, v.40, n.5, p.80-91, May 1997.

[WEI 99] WEISS, Gerhard. **Multiagent Systems**: A Modern Approach to Distributed Artificial Intelligence**.** London: MIT Press, 1999. 619p.

[WIR 73] WIRTH, Niklaus. **Systematic Programming**: An Introduction. New Jersey: Prentice-Hall, 1973. 169p.

[YAM 2001] YAMIN, Adenauer C.; AUGUSTIN, Iara; BARBOSA, Jorge L. V.; SILVA, Luciano; GEYER, Cláudio F. R. Explorando o Escalonamento no Desempenho de Aplicações Móveis Distribuídas. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2., 2001. **Anais...** Pirenópolis: SBC, 2001. p.1-8.