



Universidade Federal de Pernambuco
Centro de Informática

Pós-Graduação em Ciência da Computação

JEOPS – Integração entre Objetos e
Regras de Produção em Java

por

Carlos Santos da Figueira Filho

Dissertação de Mestrado

Recife, Outubro de 2000

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

CARLOS SANTOS DA FIGUEIRA FILHO

**JEOPS – Integração entre Objetos e
Regras de Produção em Java**

*Este trabalho foi submetido à Pós-Graduação em Ciência da
Computação do Centro de Informática da Universidade
Federal de Pernambuco como requisito parcial para
obtenção do grau de Mestre em Ciência da Computação*

Orientador: Prof. Dr. Geber Lisboa Ramalho

Recife, 11 de Outubro de 2000

RESUMO

Apesar do conceito de agentes inteligentes ser de grande ajuda na concepção e desenvolvimento de sistemas que requerem um comportamento inteligente, a falta de ferramentas adequadas para ajudar a criação de aplicações baseadas no paradigma de agentes ainda é um obstáculo para a sua maior difusão. Agentes podem ser criados através da utilização de linguagens específicas para este propósito (linguagens orientadas a agentes), ou linguagens de propósito geral. Nós cremos que a segunda alternativa é mais viável para aplicações onde a reutilização de componentes é altamente necessária. Neste caso, é preciso integrar à linguagem de propósito geral mecanismos de inferência que auxiliem a programação de entidades inteligentes. Neste cenário, a integração de objetos e regras de produção é uma abordagem promissora, tendo sido cada vez mais estudada na área denominada EOOPS (de *Embedded Object-Oriented Production Systems*). O nosso trabalho consistiu no projeto, desenvolvimento e validação de JEOPS, um motor de inferência de primeira ordem, com encadeamento progressivo, que visa prover capacidade de raciocínio a Java. O desenvolvimento do sistema foi feito de modo a aumentar o máximo a uniformidade da integração regras-objetos, o que tem sido negligenciado pela maioria dos sistemas atuais. Observamos que esta integração reduz o tempo de aprendizado do sistema, além de ter impacto na reusabilidade e compreensibilidade dos sistemas desenvolvidos. JEOPS já vem sendo utilizado em diferentes projetos por mais de 2 anos, e seus resultados vem sendo promissores.

ABSTRACT

Although the concept of intelligent agents is of great help in the design and development of systems that require intelligent behavior, the lack of standard tools to help the creation of applications based on this paradigm is still an obstacle for its broad use. Agents can be created via the use of either specialized languages (agent oriented languages) or general-purpose ones. We claim that the latter is more suitable for applications where components reuse is highly demanded. In this case, reasoning mechanisms need to be integrated with the language, to aid in the development of intelligent entities. As such, the integration of production rules and objects is a promising approach, and it has been increasingly investigated within the framework of EOOPS (Embedded Object-Oriented Production Systems). Our work consisted in the project, development and validation of JEOPS, a first-order forward-chaining inference engine that aims to provide reasoning capabilities to Java. JEOPS gives a special attention to the problem of the rule-object integration uniformity, which has been neglected by most of the current systems. We have observed that this integration has a strong impact on the user learning curve of the system, as well as on the system reusability and readability. JEOPS has been used for over 2 years in different projects, and the results are encouraging.

ÍNDICE

1	INTRODUÇÃO.....	1
2	ESTADO DA ARTE	8
2.1	SISTEMAS DE PRODUÇÃO	8
2.2	ORIENTAÇÃO A OBJETOS	10
2.3	EOOPS.....	12
2.3.1	<i>De fatos a objetos</i>	<i>12</i>
2.3.2	<i>Metodologia de desenvolvimento</i>	<i>16</i>
2.3.3	<i>Vantagens da integração.....</i>	<i>20</i>
2.3.4	<i>Problemas da integração.....</i>	<i>22</i>
2.3.5	<i>Regras pré-compiladas ou interpretadas</i>	<i>26</i>
2.4	CRITÉRIOS DE ANÁLISE.....	29
2.5	CLIPS.....	31
2.5.1	<i>Apresentação</i>	<i>31</i>
2.5.2	<i>Crítica.....</i>	<i>34</i>
2.6	RAL/C++	35
2.6.1	<i>Apresentação</i>	<i>35</i>
2.6.2	<i>Crítica.....</i>	<i>37</i>
2.7	NÉOPUS.....	38
2.7.1	<i>Apresentação</i>	<i>38</i>
2.7.2	<i>Crítica.....</i>	<i>40</i>
2.8	JESS.....	40
2.8.1	<i>Apresentação</i>	<i>40</i>
2.8.2	<i>Crítica.....</i>	<i>42</i>
2.9	OUTRAS INTEGRAÇÕES DE MECANISMOS DE RACIOCÍNIO COM OBJETOS	43
2.9.1	<i>Integração Java x Prolog</i>	<i>43</i>
2.10	SÍNTESE.....	47
3	JEOPS.....	48
3.1	DA CONCEPÇÃO AO ESTÁGIO ATUAL	48
3.1.1	<i>A proposta inicial</i>	<i>48</i>
3.1.2	<i>A continuidade.....</i>	<i>49</i>
3.1.3	<i>O trabalho de graduação</i>	<i>49</i>

3.1.4	<i>O trabalho de mestrado</i>	50
3.2	PRINCÍPIOS.....	51
3.2.1	<i>Uniformidade da integração</i>	51
3.2.2	<i>Implementação do motor de inferência</i>	52
3.2.3	<i>Algoritmo de unificação</i>	52
3.2.4	<i>Distribuição do sistema</i>	52
3.3	REGRAS JEOPS.....	53
3.4	UTILIZAÇÃO DO SISTEMA.....	55
3.4.1	<i>Definição do problema</i>	55
3.4.2	<i>Definição dos objetos e das regras</i>	56
3.4.3	<i>Pré-compilação da base de regras</i>	59
3.4.4	<i>Utilização da base de conhecimentos</i>	61
3.5	ARQUITETURA INTERNA.....	62
3.5.1	<i>A base de conhecimentos</i>	63
3.5.2	<i>A base de objetos</i>	65
3.5.3	<i>A base interna de regras</i>	65
3.5.4	<i>A rede Rete</i>	66
3.5.5	<i>O conjunto de conflitos</i>	68
3.6	INTERFACE DE DEPURAÇÃO.....	70
3.7	SÍNTESE.....	72
4	IMPLEMENTAÇÃO	73
4.1	A BASE DE OBJETOS.....	73
4.2	O CONJUNTO DE CONFLITOS.....	76
4.3	A REDE RETE.....	78
4.4	A BASE DE CONHECIMENTOS.....	82
4.4.1	<i>Criação da base de conhecimentos</i>	82
4.4.2	<i>A inclusão de objetos</i>	84
4.4.3	<i>A remoção de objetos</i>	84
4.4.4	<i>A modificação de objetos</i>	85
4.4.5	<i>Consulta à memória de trabalho</i>	86
4.4.6	<i>Execução do motor de inferência</i>	86
4.5	A BASE INTERNA DE REGRAS.....	87
4.5.1	<i>Informações genéricas</i>	87
4.5.2	<i>Atribuição e consulta de objetos</i>	88
4.5.3	<i>Consulta a condições</i>	89
4.5.4	<i>Disparo de regras</i>	90
4.5.5	<i>Manipulação da memória de trabalho</i>	90

4.6	A PRÉ-COMPILAÇÃO REGRAS → JAVA	91
4.6.1	<i>O processo de pré-compilação</i>	92
4.6.2	<i>A compilação das classes Java</i>	94
4.7	SÍNTESE	94
5	RESULTADOS OBTIDOS	96
5.1	UTILIZAÇÃO DE JEOPS	96
5.1.1	<i>Mobilet</i>	96
5.1.2	<i>Enigmas</i>	97
5.1.3	<i>PubsFinder</i>	97
5.1.4	<i>Gerador de variações fonéticas</i>	98
5.1.5	<i>NetMaze</i>	98
5.1.6	<i>Utilização como Ferramenta de Ensino</i>	99
5.2	COMPARAÇÃO COM OUTROS SISTEMAS	100
5.2.1	<i>Resumo das características</i>	100
5.2.2	<i>Benchmark</i>	101
5.3	SÍNTESE	102
6	CONCLUSÕES	103
7	REFERÊNCIAS BIBLIOGRÁFICAS	106
	ANEXO A – NOTAÇÃO UTILIZADA (UML)	113
	ANEXO B – BNF DAS REGRAS JEOPS	114
	ANEXO C – ESTUDOS DE CASO	115
C.1	SÉRIE DE FIBONACCI	115
C.2	AS 8 RAINHAS	116
	ANEXO D – A DISTRIBUIÇÃO DO SISTEMA	120
	ANEXO E – MANUAL DO USUÁRIO	122
	ANEXO F – CONVERSÃO PROLOG → JAVA	123
	ANEXO G – CONVERSÃO DE BASE DE REGRAS	124
G.1	INFORMAÇÕES GENÉRICAS SOBRE A BASE DE REGRAS	124

G.2	DECLARAÇÃO DE VARIÁVEIS DE INSTÂNCIA	125
G.3	CONSULTAS SOBRE AS DECLARAÇÕES	125
G.4	ATRIBUIÇÃO DE VALORES ÀS DECLARAÇÕES DAS REGRAS.....	126
G.5	CONSULTA DOS VALORES MAPEADOS ÀS DECLARAÇÕES DAS REGRAS	127
G.6	AVALIAÇÃO DE CONDIÇÕES DAS REGRAS.....	128
G.7	DISPARO DAS REGRAS	129
G.8	FINALIZAÇÃO DA COMPILAÇÃO.....	129

ÍNDICE DE FIGURAS

FIGURA 1 – LIGAÇÃO SIMBÓLICA ENTRE OBJETOS.....	15
FIGURA 2 – LIGAÇÃO NATURAL ENTRE OBJETOS	16
FIGURA 3 – CLASSE PESSOA	18
FIGURA 4 – CLASSE OBJETIVO	18
FIGURA 5 – REFINAMENTO DA CLASSE OBJETIVO	19
FIGURA 6 – O PROBLEMA DA MODIFICAÇÃO DE OBJETOS	25
FIGURA 7 – COMPILAÇÃO DE UM PROGRAMA DA LINGUAGEM C	27
FIGURA 8 – COMPILAÇÃO DE UM PROGRAMA DA LINGUAGEM JAVA	27
FIGURA 9 – PRÉ-COMPILAÇÃO DE REGRAS	28
FIGURA 10 – <i>SCRIPTING</i> VIA JESS	42
FIGURA 11 – INTEGRAÇÃO DE RACIOCÍNIO COM ORIENTAÇÃO A OBJETOS.....	43
FIGURA 12 – CIVILIZATION II.....	56
FIGURA 13 – CLASSES DO DOMÍNIO DE CIVILIZATION II.....	57
FIGURA 14 – REDEFINIÇÃO DA CLASSE CIDADE.....	57
FIGURA 15 – REDEFINIÇÃO DA CLASSE JOGADOR.....	58
FIGURA 16 – PRÉ-COMPILAÇÃO DO ARQUIVO DE REGRAS	60
FIGURA 17 – ETAPAS DA PRÉ-COMPILAÇÃO DO ARQUIVO DE REGRAS	61
FIGURA 18 – ARQUITETURA DE JEOPS	63
FIGURA 19 – IMPLEMENTAÇÃO TRADICIONAL DE RETE.....	67
FIGURA 20 – ESTRUTURA DO CONJUNTO DE CONFLITOS.....	68
FIGURA 21 – CLASSE OBJECTBASE	73
FIGURA 22 – INSERÇÃO DE UM OBJETO DE UMA NOVA CLASSE NA BASE DE OBJETOS.	74
FIGURA 23 – INSERÇÃO DE UM OBJETO DE UMA CLASSE JÁ EXISTENTE NA BASE DE OBJETOS.	75
FIGURA 24 – IMPLEMENTAÇÃO <i>RETE</i> DE JEOPS	78
FIGURA 25 – REDE RETE DA REGRA HOLDOBJECTHOLDSATISFIED	80
FIGURA 26 – MEMÓRIA DO NÓ DE JUNÇÃO	81
FIGURA 27 – CRIAÇÃO DA BASE INTERNA DE REGRAS	83
FIGURA 28 – REMOÇÃO DE OBJETOS DA BASE DE CONHECIMENTOS	85
FIGURA 29 – IMPLEMENTAÇÃO DO MÉTODO MODIFIED.....	86
FIGURA 30 – IMPLEMENTAÇÃO DE RETRACT NA BASE INTERNA DE REGRAS.....	91
FIGURA 31 – ESTADOS DO PRÉ-COMPILADOR DA BASE DE REGRAS	92
FIGURA 32 – NOTAÇÃO (SIMPLIFICADA) UML DE DEFINIÇÃO DE CLASSES	113
FIGURA 33 – SÉRIE DE FIBONACCI.....	115
FIGURA 34 – CLASSE FIBONACCI	115

FIGURA 35 – O PROBLEMA DAS OITO RAINHAS.....	117
FIGURA 36 – CLASSE QUEEN.....	118
FIGURA 37 – PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO JEOPS.....	120

ÍNDICE DE TABELAS

TABELA 1 – REGRA DE PRODUÇÃO	9
TABELA 2 – REGRA OPS5	14
TABELA 3 – CLASSES USADAS NA BUSCA POR ANCESTRAIS	14
TABELA 4 – REGRA ENCONTRAANCESTRAL (OPS5)	14
TABELA 5 – METODOLOGIA DE DESENVOLVIMENTO DE SISTEMAS EOOPS	17
TABELA 6 – PRIMEIRA VERSÃO DA REGRA ENCONTRAANCESTRAIS	18
TABELA 7 – SEGUNDA VERSÃO DA REGRA ENCONTRAANCESTRAIS	19
TABELA 8 – CRIAÇÃO DE OBJETOS DA CLASSE PESSOA	20
TABELA 9 – RESULTADO DA EXECUÇÃO DO SISTEMA DE PRODUÇÃO	20
TABELA 10 – CICLO DE EXECUÇÃO DE UM MOTOR DE INFERÊNCIA	22
TABELA 11 – ALGORITMO DE UNIFICAÇÃO DE UM MOTOR DE INFERÊNCIA	23
TABELA 12 – PROBLEMA DA MODIFICAÇÃO DE OBJETOS	23
TABELA 13 – DEFINIÇÃO DO TIPO PESSOA	32
TABELA 14 – CRIAÇÃO DE FATOS DO TIPO PESSOA	32
TABELA 15 – DEFINIÇÃO DA CLASSE PESSOA	33
TABELA 16 – CRIAÇÃO DE INSTÂNCIAS DA CLASSE PESSOA	33
TABELA 17 – REGRA ENCONTRAANCESTRAIS (CLIPS)	33
TABELA 18 – REGRA ENCONTRAANCESTRAL (RAL/C++)	36
TABELA 19 – DECLARAÇÃO DO TIPO EMPREGADO	36
TABELA 20 – REGRA ENCONTRAANCESTRAIS (NÉOPUS)	39
TABELA 21 – UTILIZAÇÃO DO SISTEMA JASPER	46
TABELA 22 – CLASSES PESSOA E OBJETIVO	53
TABELA 23 – BASE QUE CONTÉM A REGRA ENCONTRAANCESTRAIS (JEOPS)	54
TABELA 24 – REGRA DEFENDECIDADE	58
TABELA 25 – REGRA DEFENDECIDADE (SEGUNDA VERSÃO)	59
TABELA 26 – BASE DE REGRAS DO DOMÍNIO DE CIVILIZATION II	59
TABELA 27 – UTILIZAÇÃO DA BASE DE CONHECIMENTOS DO PROBLEMA DE CIVILIZATION II	62
TABELA 28 – PARES DE CONDIÇÕES SEMANTICAMENTE IDÊNTICAS	67
TABELA 29 – UTILIZAÇÃO DO MECANISMO DE DEPURAÇÃO DE JEOPS	72
TABELA 30 – IMPLEMENTAÇÃO DO MÉTODO OBJECTS (STRING) DA BASE DE OBJETOS	75
TABELA 31 – MÉTODO INSERTELEMENT DA CLASSE DEFAULTCONFLICTSET	77
TABELA 32 – MÉTODO REMOVEELEMENTSWITH DA CLASSE ONSHOTCONFLICTSET	78
TABELA 33 – MÉTODO REMOVEELEMENTSWITH DA CLASSE DEFAULTCONFLICTSET	78
TABELA 34 – REGRA DO PROBLEMA DO MACACO E DAS BANANAS	80

TABELA 35 – INICIALIZAÇÃO DA BASE DE CONHECIMENTOS	82
TABELA 36 – CRIAÇÃO DA REDE RETE	84
TABELA 37 – MÉTODO <code>run()</code> DA BASE DE CONHECIMENTOS.....	87
TABELA 38 – PROCESSAMENTO DAS CONDIÇÕES DE UMA REGRA.....	92
TABELA 39 – IMPLEMENTAÇÃO DO MÉTODO <code>fireRule</code> DO EXEMPLO DE CIVILIZATION II	94
TABELA 40 – COMPARAÇÃO ENTRE OS SISTEMAS APRESENTADOS (* = HIPÓTESE DO MUNDO FECHADO)	100
TABELA 41 – RESULTADOS DOS <i>BENCHMARKS</i>	102
TABELA 42 – REGRAS DO PROBLEMA DA SÉRIE DE FIBONACCI.....	116
TABELA 43 – UTILIZAÇÃO DA BASE DE FIBONACCI	116
TABELA 44 – BASE DE REGRA PARA O PROBLEMA DAS 8 RAINHAS	118
TABELA 45 – UTILIZAÇÃO DA BASE DE CONHECIMENTO DAS 8 RAINHAS	119
TABELA 46 – UTILIZAÇÃO DE JEOPS EM APPLETS	121
TABELA 47 – BASE DE REGRAS <i>FAMILIA</i>	124
TABELA 48 – INFORMAÇÕES GENÉRICAS SOBRE A BASE INTERNA DE REGRAS	125
TABELA 49 – VARIÁVEIS DE INSTÂNCIA DA BASE INTERNA DE REGRAS	125
TABELA 50 – CONSULTAS SOBRE AS CLASSES DAS DECLARAÇÕES DA BASE INTERNA DE REGRAS.....	126
TABELA 51 – ATRIBUIÇÃO DE VALORES ÀS DECLARAÇÕES DAS REGRAS.....	127
TABELA 52 – CONSULTA DOS VALORES MAPEADOS ÀS DECLARAÇÕES DAS REGRAS	127
TABELA 53 – MÉTODOS DE AVALIAÇÃO DAS CONDIÇÕES DAS REGRAS	129
TABELA 54 – MÉTODOS PARA O DISPARO DAS REGRAS	129
TABELA 55 – CONSTRUTOR DA BASE INTERNA DE REGRAS	130
TABELA 56 – BASE DE CONHECIMENTOS GERADA PELA PRÉ-COMPILAÇÃO	130

1 INTRODUÇÃO

O crescimento vertiginoso da Internet tem colocado certos problemas para os quais as abordagens computacionais tradicionais não se mostram viáveis. Estes problemas incluem a recuperação e filtragem de informação na Internet, intermediação de operações de comércio eletrônico, notificação de eventos ao usuário, entre outros. Na recuperação e filtragem de informação da Internet, dado que um ser humano não pode manipular diretamente toda informação contida na rede, é preciso criar programas que possam autonomamente encontrar e disponibilizar informações levando em conta os objetivos e preferências desse ser humano. Operações de comércio eletrônico (compra e venda de produtos, leilões virtuais) podem ser automatizadas com o auxílio de programas negociadores, que devem representar o ser humano (seu usuário) na busca das melhores condições de pagamento, tanto no lado do vendedor como no lado do comprador. Serviços de notificação devem continuamente observar um determinado domínio e notificar o seu usuário sempre que ocorra algum evento de interesse deste (como a de novas páginas de chamadas de trabalhos, oscilações no preço de ações da bolsa de valores ou o resultado de competições esportivas, por exemplo) – podendo aprender com os seus acertos e erros, de modo que cada vez mais informe ao usuário tudo e somente o que este deseje saber. Tipicamente, a resolução destes problemas são tarefas de alta complexidade que requerem a modelagem do conhecimento especializado de um ser inteligente, e que demandam, freqüentemente, a integração de várias entidades de software distribuídas.

Uma outra área onde houve um grande crescimento foi a da indústria da computação pessoal. Nos anos 80 (e aqui no Brasil no início dos anos 90 devido à reserva de mercado), a possibilidade de se substituir a antiga máquina de escrever por um novo aparelho que aumentava a produtividade levou ao primeiro *boom* de vendas dos computadores pessoais. Logo depois, a descoberta de que um computador poderia auxiliar em tarefas de organização de contas, editoração eletrônica, animação, entre outras, trouxe um novo ânimo à indústria. A partir de meados da década de 90, a Internet alavancou as vendas de forma que hoje o computador pessoal já é uma realidade para um número razoável de pessoas. No entanto, há um setor desta indústria que apresenta um crescimento até mesmo maior que a própria computação pessoal; o entretenimento digital,

principalmente o setor de jogos. Este setor utiliza cada vez mais as novas tecnologias para o desenvolvimento de jogos mais elaborados, havendo até mesmo quem afirme que é a indústria dos jogos que leva ao desenvolvimento da computação pessoal [1]. Nos últimos anos, tem crescido a exigência de que o comportamento dos oponentes do jogador controlados pelo computador sejam menos previsíveis, e que eles sejam capazes de aprender com o tempo, de modo a tornar o desafio de derrotá-los cada vez mais difícil. É preciso uma técnica de programação que provenha um certo nível de inteligência para o computador.

Estes dois cenários, Internet e jogos, apresentam problemas para os quais a computação tradicional tem encontrado dificuldades na tentativa de encontrar soluções. Basicamente, são problemas cuja complexidade torna proibitiva a utilização de estruturas algorítmicas clássicas. No entanto, estes tipos de problemas podem ser resolvidos através da utilização do paradigma de agentes inteligentes, que engloba metáforas, técnicas e ferramentas para facilitar a concepção, análise e implementação de programas que requerem um comportamento inteligente. Agentes são entidades de hardware ou (mais freqüentemente) software com as seguintes propriedades principais [2]:¹

- ***Autonomia***
Capacidade de operar sem a intervenção direta de humanos ou de outros, com controle sobre suas ações e estado interno
- ***Habilidade social***
Capacidade de interagir com outros agentes ou seres humanos através de algum tipo de linguagem de comunicação de alto nível
- ***Reatividade***
Capacidade de perceber o ambiente em sua volta e responder às mudanças no ambiente em um tempo razoável)
- ***Pro-atividade***
Capacidade de tomar a iniciativa para satisfazer seus objetivos.

Todas estas características têm na base a capacidade de raciocínio. O fato de uma entidade poder pensar, ou tirar conclusões sobre um conjunto de fatos (possivelmente em

¹ Outras definições incluem características como persistência temporal, mobilidade, etc [3] [4] [5].

um domínio específico) lhe dá as capacidades necessárias para receber a denominação de *agente inteligente* (segundo a definição dada acima). Ela terá autonomia, pois terá a capacidade de *decidir* qual o próximo passo a dar, sem a necessidade de auxílio externo. A capacidade de raciocínio permitirá que o agente se engaje em diálogos com outros agentes, pois poderá *compreender* uma idéia a partir de uma frase formada em uma estrutura sintática conhecida pelos agentes comunicantes (a língua portuguesa, por exemplo). A partir de estímulos do mundo externo, o agente poderá *escolher* qual a melhor ação a ser tomada em resposta. Finalmente, sabendo os objetivos que deve alcançar, um agente poderá *decidir* dentre as alternativas de ações que ele pode realizar aquela que lhe levará para um ponto cada vez mais próximo de seu fim.

Embora o conceito de agentes inteligentes seja de grande auxílio na concepção e desenvolvimento de programas que apresentam um comportamento inteligente, as ferramentas para a criação de aplicações baseadas neste conceito ainda são um pouco incipientes. Nos últimos anos, novas ferramentas têm sido desenvolvidas, sem atingir no entanto um grau de evolução que permitisse a consolidação da área. Com isso, a utilização deste paradigma em todo o ciclo do desenvolvimento das aplicações (ou seja, do nível de conhecimento até o nível de implementação [3]) ainda está um pouco longe de se tornar realidade.

Na área dos agentes inteligentes, bem como em todas as áreas específicas da computação, o problema da criação de aplicações baseadas em um conceito particular pode ser solucionado de duas formas. Uma delas é a utilização de linguagens específicas para a área (ditas “orientadas a agentes”), que provêm uma grande facilidade para o usuário naquilo para que foram projetadas. Agent0 [6], METATEM [7] e PLACA [8] são linguagens concebidas especialmente para implementação de agentes (assim como VRML [9] é uma linguagem dedicada à realidade virtual, por exemplo). Linguagens orientadas a agentes oferecem a vantagem de ter uma fundamentação teórica sólida as suportando, tornando-as ideais para aplicações onde a preocupação principal são as características básicas de um agente.

No entanto, as aplicações que podem ser desenvolvidas com estas linguagens são de certo modo restritas, pois elas geralmente não provêm serviços além dos estritamente necessários para a criação de agentes. Possivelmente isto ocorre para se manter a

homogeneidade da linguagem, ou simplesmente devido à falta de tempo dos criadores destas linguagens para implementar serviços gerais. Em consequência, se o desenvolvedor precisar de uma funcionalidade além das que o sistema provê, ele terá que desenvolvê-la. Outra consequência da falta de serviços é que os sistemas construídos com estas linguagens não conseguem facilmente interagir com outros sistemas externos.

Por exemplo, Agent0 provê características como programação declarativa e modelagem de estados mentais e engajamentos, mas serviços como mobilidade, acesso a bancos de dados, construção de interfaces, acesso sistemas heterogêneos distribuídos não são providos naturalmente. METATEM provê mecanismos de programação concorrente, mas não modela naturalmente engajamentos mentais. Se alguma das linguagens oferece todos os serviços necessários para uma determinada aplicação, então há grandes chances dela ser a ferramenta ideal para a sua implementação.

Para eliminar o problema da falta de serviços, a implementação de agentes em linguagens de propósito geral – especialmente as orientadas a objetos (onde a reutilização de componentes e serviços desenvolvidos por terceiros é encorajada), tais como C++ [10], Smalltalk [11] ou Java™ [12] – aparece como uma segunda opção. De fato, através da utilização de linguagens de programação orientadas a objetos no desenvolvimento de agentes inteligentes, os desenvolvedores podem se aproveitar de todos os serviços já desenvolvidos nestas linguagens por outras pessoas. Estes serviços podem ser divididos em dois grupos: os de propósito geral, como mobilidade, acesso a banco de dados, acesso à rede, ferramentas para construção de interfaces, etc.; e os específicos para a criação de agentes, que são o caso de mecanismos de inferência [13], linguagens de comunicação entre agentes [14], entre outros. Uma outra vantagem da utilização de linguagens orientadas a objetos no desenvolvimento de agentes é a possibilidade de utilização de técnicas de análise e projeto maduras e estáveis, como OMT [15] ou UML [16]. Estas técnicas facilitam a criação e manutenção de grande sistemas, o que pode ser o caso de certas aplicações de agentes.

Por outro lado, as linguagens orientadas a objetos não oferecem de antemão as funcionalidades das linguagens específicas para o desenvolvimento de agentes. Ganha-se então em flexibilidade, mas perde-se em facilidade de uso, pois é preciso que estas funcionalidades sejam programadas pelo desenvolvedor. Além do mais, ainda não há

nenhum *framework* (ou um conjunto de técnicas e ferramentas sobre o qual é feito o desenvolvimento) bem definido para a criação de aplicações baseadas em agentes utilizando linguagens orientadas a objetos: a aplicação se torna uma “colcha de retalhos” de diversos componentes reutilizados, pois todos os componentes são desenvolvidos com uma interface própria, não seguindo uma padronização que tornasse a aplicação como um todo consistente.²

Em suma, apesar da abordagem baseada na programação orientada a objetos não ser tão adequada quanto a anterior, ela é a alternativa mais simples para aplicações que demandam diversos serviços de propósito geral. No caso de se utilizar linguagens orientadas a objetos para implementar agentes inteligentes para estas aplicações, que são muitas [19], é necessária entretanto a utilização (ou o desenvolvimento) de serviços específicos de agentes que possam ser embutidos nestas linguagens. Entre os serviços citados anteriormente estão os mecanismos de raciocínio que, como já dito, estão na base das características de um agente.

Entre as propostas de prover capacidade de raciocínio, a integração de regras de produção [13] com linguagens orientadas a objetos está sendo cada vez mais utilizada [20], no que convencionou chamar-se EOOPS (de *Embedded Object-Oriented Production Systems*). Neste sentido, já foram feitas diversos esforços na criação de componentes de software para prover capacidade de raciocínio para essas linguagens, como CLIPS [21] [22] [23] e RAL/C++ [24] para C++, Opus [25] e NéOpus [26] para Smalltalk, e JESS [27] para Java.

Neste trabalho apresentamos JEOPS [28] [29], um motor de inferência de primeira ordem com encadeamento progressivo integrado à Java. O sistema provê uma transição suave entre a linguagem hospedeira e as regras de produção, por utilizar a mesma filosofia de Java na criação das regras e na utilização de objetos. Este aspecto, negligenciado no todo ou em parte pelos sistemas anteriores (à exceção de Opus e NéOpus, desenvolvidos para Smalltalk), nos possibilitou criar um sistema tão fácil de usar quanto possível para desenvolvedores com experiência em Java, mas não necessariamente com experiência em

² Já existem algumas propostas para metodologias de desenvolvimento de sistemas baseados em agentes, quer adaptando metodologias antigas, quer partindo do zero. Nenhuma delas, entretanto, já é amplamente aceita pela comunidade científica [17] [18].

trabalhos da área de Inteligência Artificial (IA). O sistema também foi desenvolvido de modo a manter as vantagens das linguagens orientadas a objetos, tais como modularidade, compreensibilidade e reusabilidade. Além disso, JEOPS combina as principais vantagens dos sistemas já existentes.

A escolha de Java para a implementação de JEOPS foi feita devido ao número crescente de serviços que esta linguagem provê. Com poucos anos de vida, Java está se tornando um padrão *de facto* para o desenvolvimento de aplicações voltadas à Internet. Sua arquitetura voltada para a rede e suas políticas de segurança tornam a linguagem muito propícia para o desenvolvimento de agentes [30]. Dada a falta de um motor de inferência fácil de usar desenvolvido para a linguagem, decidimos que Java seria a linguagem ideal para o desenvolvimento de JEOPS.

A primeira versão de JEOPS foi desenvolvida dois anos atrás. O sistema tem sido continuamente melhorado desde então, em resposta às necessidades das aplicações que vêm sendo desenvolvidas com ele e de *feedback* dos usuários. As aplicações do sistema incluem, entre outras: jogos interativos, como o “Enigma no Campus” [31], que contêm diversos agentes interagindo entre si; jogos de ação distribuídos, como o NetMaze [32]; agentes móveis para o gerenciamento de sistemas distribuídos heterogêneos; sistemas de recuperação de informação na Internet, que buscam na rede páginas especializadas (chamadas de trabalhos, receitas, *papers*), notificando seus usuários a cada nova ocorrência. JEOPS também vem sendo utilizado com sucesso como uma ferramenta de auxílio ao ensino de Inteligência Artificial em cursos de graduação e pós-graduação da UFPE.

As principais contribuições deste trabalho são a construção de um sistema pioneiro que provê uma integração uniforme da linguagem Java com regras de produção. Esta integração permite que o sistema seja sobretudo fácil de ser utilizado, e de ser implementado segundo parâmetros de qualidade de software. Acreditamos, portanto, que JEOPS representa um avanço no estado da arte de EOOPS, e a partir de sua divulgação cremos que sua evolução ocorrerá naturalmente.

No próximo capítulo, apresentaremos o que já foi feito na área de sistemas híbridos que unem objetos e regras, apresentando uma introdução aos conceitos de sistemas de produção e orientação a objetos e a EOOPS, com suas vantagens e desvantagens. Em seguida, será apresentado JEOPS, a sua história, filosofia e seus principais componentes.

No capítulo seguinte a implementação de JEOPS será apresentada em maiores detalhes, de modo que o leitor possa compreendê-lo em sua totalidade. Serão apresentados os resultados já obtidos, com os projetos que foram implementados utilizando o JEOPS e a experiência de sua utilização como ferramenta de ensino. Finalmente, serão apresentadas as conclusões tiradas com o desenvolvimento deste trabalho, e algumas sugestões de futuros melhoramentos para o sistema. O apêndice deste trabalho ainda apresenta alguns problemas clássicos da área resolvidos como estudos de caso de JEOPS, o que pode ser utilizado como referência da utilização do sistema.

2 ESTADO DA ARTE

Este capítulo apresenta uma série de propostas de projetos de sistemas que integram objetos (ou estruturas próximas) e regras de produção com encadeamento progressivo. Apresentaremos antes uma visão geral da área, com suas vantagens e problemas, bem como um breve contexto histórico dos sistemas desenvolvidos nessa área. Antes disso, apresentaremos uma pequena introdução sobre sistemas de produção e sobre objetos, somente para lembrar os principais conceitos envolvidos neste trabalho.

2.1 SISTEMAS DE PRODUÇÃO

Dentre as diversas maneiras de se representar conhecimento declarativo, ideal para modelar o raciocínio humano [33], os *sistemas de produção* estão entre as formas mais utilizadas [34]. Sistemas de produção são a denominação genérica dada aos sistemas que utilizam para representação de conhecimento as *regras de produção*. Um sistema de produção pode ser definido como um conjunto de produções (ou regras de produção), uma *memória de trabalho* onde são armazenados os fatos, e um algoritmo conhecido por encadeamento progressivo (ou *forward-chaining*), que produz novos fatos a partir de fatos antigos. Uma regra se torna pronta para ser *disparada* quando suas condições são satisfeitas por fatos da memória de trabalho. Uma *estratégia de resolução de conflitos* determina dentre as regras que estão prontas (o *conjunto de conflito*) a próxima a ser disparada [35].

Regras de produção são utilizadas predominantemente para se representar um conhecimento heurístico sobre o mundo (outras formas de conhecimento, como o procedimental também podem ser representados), especificando um conjunto de ações que devem ser realizadas para uma dada situação. Uma regra é composta por uma parte de condições (também conhecida por parte “se” da regra, ou lado *esquerdo* da regra, ou LHS – de *left-hand size*) e uma parte de ações (a parte “então” da regra, ou o lado *direito* da regra, ou RHS – de *right-hand size*). Uma condição apresenta uma lista de símbolos ou variáveis, que devem ser unificadas (ou *casadas*, dependendo do vocabulário utilizado) com os fatos da memória de trabalho. Este processo de unificação (ou *pattern matching*) é feito por um *motor de inferência*. A Tabela 1 ilustra uma regra de produção, que poderia ser usada num sistema de controle de incêndios:

Se	Existe um Sensor s numa sala x indicando uma temperatura ambiente t que é maior que o limiar l
Então	Dispare o alarme indicando possibilidade de incêndio na sala x .

Tabela 1 – Regra de Produção

Em sistemas cuja expressividade é equivalente à da lógica de primeira ordem (com a hipótese do mundo fechado), a regra acima poderia ser lida como “para todo sensor s , se s está na sala x , e s indica que a temperatura t é maior ...”. Esta expressividade não existe, por exemplo, em sistemas onde não se pode definir quantificações para as variáveis que são declaradas.

O ciclo de execução de um sistema de produção com encadeamento progressivo tem três passos principais:

- ***Unificação***

No qual o sistema de produção procura unificar os fatos da memória de trabalho com as declarações das regras de modo a tornar verdade os predicados que compõem as condições das mesmas; a cada unificação bem sucedida, o par <regra, fatos> é inserido no conjunto de conflitos.

- ***Resolução de Conflitos***

Onde é escolhido um par <regra, fatos> do conjunto de conflitos, de acordo com a política de resolução de conflitos utilizada;

- ***Disparo da Regra***

No qual as ações da regra escolhida na etapa anterior são executadas com suas variáveis substituídas pelos fatos que tornaram a regra disparável.

Este ciclo se repete até que não haja mais nenhuma regra disparável. A ação da regra pode alterar a lista de regras disparáveis, através do acréscimo ou da remoção de fatos da memória de trabalho. Estas operações, que convencionou-se chamar de “assert” e “retract” respectivamente, são a base da manipulação de fatos da memória de trabalho.

Esta definição clássica de sistemas de produção (na qual o algoritmo utilizado é o encadeamento progressivo) é um pouco controversa. Outros autores, como Jackson [36], definem sistemas de produção como podendo apresentar tanto encadeamento progressivo quanto regressivo. No encadeamento progressivo, a inferência é feita de fatos que acredita-

se que sejam verdadeiros para novos estados que as condições nos permitem estabelecer. No encadeamento regressivo (também conhecido por *backward-chaining*), o encadeamento se dá de um objetivo para as condições necessárias para que esse possa ser estabelecido. Em sistemas com encadeamento progressivo, o motor de inferência tenta unificar o lado esquerdo da regra com as variáveis da memória de trabalho, executando em seguida as ações definidas no lado direito. Em sistemas com encadeamento regressivo, o motor de inferência tenta unificar o lado direito das regras com os objetivos os quais se tenta provar, gerando como novos sub-objetivos as condições definidas no lado esquerdo, até que se atinjam condições atômicas (fatos já presentes na memória de trabalho).

Uma ressalva deve ser feita em relação ao poder expressivo dos sistemas de produção. Por trabalharem com lógicas de ordem 0, 0+ ou 1 (com restrições), há diversos problemas que não podem ser resolvidos com tais sistemas (que envolvam lógicas de ordens superiores, incerteza ou qualquer outro tipo de lógica não-clássica [3]). No entanto, a quantidade de problemas que podem ser especificados (e resolvidos) na lógica de primeira ordem é grande o suficiente para motivar este trabalho.

2.2 ORIENTAÇÃO A OBJETOS

O aumento da complexidade dos sistemas, principalmente durante a década de 1970, fez com que o custo de desenvolvimento do software tivesse um crescimento tal que estava se tornando proibitiva a evolução dos sistemas. As linguagens desenvolvidas até a época (ditas de primeira, segunda e terceira gerações) foram desenvolvidas para a programação de sistemas de pequeno ou médio porte, de modo que não eram adequadas para os grandes sistemas que precisavam ser construídos. A chamada “crise do software” [37] fez com que diversas pesquisas em linguagens de programação e técnicas de análise e modelagem de software fossem desenvolvidas. Dentre as propostas surgidas na época, e refinadas nos anos subseqüentes, as técnicas de Orientação a Objetos (que compreendem a análise, o projeto e a programação orientada a objetos) aparecem hoje como a melhor alternativa para o desenvolvimento de grandes aplicações.

O modelo utilizado no desenvolvimento orientado a objetos (*o modelo de objetos*) é baseado nos seguintes elementos básicos [38]:

- **Abstração**

É a idéia central do modelo de objetos. A abstração denota as principais características de um objeto, que o distinguem de todos os outros tipos de objetos. Uma abstração é a visão que os demais objetos têm de um determinado objeto, separando o seu comportamento essencial de sua implementação. Uma decorrência da abstração dos objetos é o fato de que os mesmos podem *esconder* informações que não forem relevantes para o mundo exterior, bem como implementar *métodos de acesso* a estas informações, permitindo que os outros objetos acessem partes de seu estado interno. Abstrações são implementadas como *classes* em linguagens orientadas a objetos, que definem um conjunto de elementos do mundo que apresentam uma série de características comuns. Por exemplo, os elementos (ou *instâncias*) da classe das pessoas são caracterizada por terem uma cabeça, um tronco e membros. No domínio da implementação, uma classe é uma estrutura que contém diversos *atributos* e pode responder a *mensagens* ou *métodos*, que são operações que podem ser realizadas pelas instâncias da classe.

- **Encapsulamento**

É o processo de criar uma “porta de entrada” para os serviços providos por um objeto. O encapsulamento serve para separar a interface de um objeto de sua implementação. O encapsulamento é importante para que um objeto possa sempre manter seu estado interno consistente, o que não seria possível se suas informações pudessem ser acessadas ou modificadas livremente pelo mundo externo.

- **Modularidade**

É a propriedade de um sistema que foi decomposto em um conjunto de módulos coesos e fracamente acoplados. Os princípios de abstração, encapsulamento e modularidade são sinérgicos, uma vez que um objeto provê um limite bem definido em torno de uma abstração, e o encapsulamento e a modularidade provêm barreiras em volta desta abstração. É o princípio da modularidade que permite que componentes de software desenvolvidos para um sistema sejam reutilizados em outros projetos, e é esta filosofia de incentivo à reutilização de

componentes que nos motivou a utilizar este paradigma para a implementação de JEOPS.

- ***Hierarquia***

Hierarquia pode ser definida simplesmente como uma ordenação de abstrações. Como exemplos de relações hierárquicas podemos citar a relação de herança (uma pessoa *é um* ser vivo, então toda instância da classe das pessoas também *é um* ser vivo), de agregação (um jardim *é composto por* diversas plantas, que por sua vez são compostas por raiz, caule, folhas e flores), entre outros.

Isto conclui a breve introdução sobre sistemas de produção e orientação a objetos. Com ela, pretendemos apresentar algumas informações que tomaremos por verdadeiras no decorrer deste trabalho, sem perder tempo com maiores explicações.

2.3 *EOOPS*

O termo EOOPS (de *Embedded Object Oriented Production Systems*) é usado para caracterizar os sistemas criados (ou ainda as ferramentas usadas para criá-los) que integram linguagens orientadas a objetos com sistemas de produção. A idéia por trás da integração de objetos com os sistemas de produção é juntar as vantagens de sistemas inteligentes, com toda a estrutura adquirida pelas linguagens orientadas a objetos durante décadas de pesquisa da Engenharia de Software.

Apresentaremos a seguir um pequeno histórico da evolução de sistemas de produção “tradicionais” até os sistemas de produção integrados a objetos, seguido das vantagens e dos problemas decorrentes da integração. Finalmente, apresentaremos uma discussão sobre a interpretação ou compilação de regras em sistemas híbridos.

2.3.1 *De fatos a objetos*

Arquiteturas tradicionais de inferência, tais como Snark [39], propõem uma dicotomia entre a *base de fatos*, que contém as estruturas de dados manipuladas pelo sistema, e a *base de regras*, que contém as regras de produção que permitem manipular os fatos, modificando-os, removendo-os ou criando novos.

Em Snark, os fatos são representados por estruturas de dados relativamente simples, formadas por triplas (objeto, atributo, valor). Estas triplas apresentam uma característica intrinsecamente booleana: sua presença afirma sua veracidade (hipótese do mundo

fechado). Por exemplo, o enunciado “O idioma do Brasil é português” é representada em Snark pela tripla (`brasil idioma portugues`), que pode ser interpretada por “*é verdade que o idioma do Brasil é português*”. As ações de regras são representadas por asserções de fatos.

Este modelo, no entanto, é bastante limitado. A assertiva acima fala que o idioma do Brasil é português. A lógica usada neste sistema não permite, por exemplo, afirmações do tipo “o idioma de **todos** os países é português”, ou do tipo “para todo país **p**, se existe um país **q** que colonizou **p**, e cujo idioma é **i**, então o idioma de **p** também é **i**”. O poder deste tipo de estrutura usada em Snark é apenas um pouco superior ao da lógica proposicional, mas inferior à lógica de predicados, que permitiria a representação da expressão anterior¹.

O sistema OPS5 [40] foi o primeiro a apresentar uma extensão ao modelo da base de fatos proposto por Snark. A idéia é que os fatos sejam substituídos por *instâncias* de classes de fatos. Com isso, os fatos deixam de ser simples relações binárias, passando a ser relações n-árias tipadas, perdendo assim a característica booleana, criando-se a idéia de existência como objeto. Todo objeto da base de fatos é uma instância de uma das classes, com valores particulares para seus atributos. Estes valores, por sua vez, são atômicos (cadeias de caracteres sem significado para a base de conhecimento, apenas para a aplicação), ou vetoriais (uma lista de valores atômicos)². Os tipos dos objetos³ OPS5 são declarados antes das regras, com o comando (`literalize pais nome idioma moeda area`), onde definindo uma classe **país** com atributos nome, idioma, moeda e área.

Uma vez que as classes foram criadas através do comando **literalize**, instâncias particulares destas classes são definidas através do comando **make**, com o circunflexo usado para indexar os atributos:

¹ Nestes casos é comum dizer que se trata de uma lógica de atributo-valor, ou de ordem 0+.

² Em outras palavras, apesar da nomenclatura (classes, objetos) semelhante à de linguagens OO, a linguagem definida por OPS5 não é completamente baseada em objetos, por não haver a possibilidade do relacionamento entre classes (por exemplo, herança, agregação, etc.), associação entre objetos ou a noção de comportamento (métodos). Trata-se de uma linguagem *pré-objeto*, conforme [26].

³ Os termos objeto OPS5 ou instância OPS5 são usados com o mesmo sentido neste documento.

- (make pais ^nome Brasil ^idioma Português ^moeda Real ^area 8511965)
- (make pais ^nome França ^idioma Francês ^moeda Franco)
- (make pais ^nome Paraguai ^idioma Espanhol ^moeda Guarani)

Atributos cujos valores não foram especificados (como a área dos dois últimos países) é inicializada por **nil**.

As regras de produção do sistema OPS5 apresentam a estrutura clássica de condições/ações. As condições são especificadas como filtros para os objetos (ou pré-objetos) presentes na base de fatos. Estes filtros são expressos por restrições simples (comparações ou igualdade) nos valores dos atributos dos objetos. As ações são expressas por uma seqüência de modificações na base de fatos. A sintaxe das regras é similar à apresentada na Tabela 2.

```
(p nome-da-regra
  (condição-1)
  (condição-2) ...
-->
  (ação-1)
  (ação-2) ...
)
```

Tabela 2 – Regra OPS5

Um exemplo, proposto por Brownston [40], ilustra a utilização deste sistema. Trata-se da busca pelos ancestrais de uma pessoa através de sua árvore genealógica. As classes usadas são apresentadas na Tabela 3 (onde alvo é o nome da pessoa pela qual se atingirá os ancestrais):

```
(literalize Pessoa nome pai mae)
(literalize Objetivo alvo)
```

Tabela 3 – Classes usadas na busca por ancestrais

A idéia da resolução do problema é se criar diversas instâncias da classe `Pessoa`, e uma instância da classe `Objetivo`, com o atributo `alvo` contendo o nome da pessoa de que se deseja descobrir os ancestrais. A regra usada na resolução do problema cria novas instâncias de `Objetivo` para a mãe e o pai do objetivo atual, e é apresentada na Tabela 4.

```
(p encontraAncestral
  obj1 <- (Objetivo ^alvo (<nome> <> nil))
  (Pessoa ^nome <nome> ^pai <nome-pai> ^mae <nome-mae>)
-->
  (remove <obj1>)
  (write <nome-pai> e <nome-mae> são ancestrais)
  (make Objetivo ^alvo <nome-pai>)
  (make Objetivo ^alvo <nome-mae>)
)
```

Tabela 4 – Regra encontraAncestral (OPS5)

A regra apresentada na Tabela 4 pode ser lida da seguinte forma: se houver um elemento **obj1** da classe Objetivo, cujo atributo **alvo** não seja nulo, e um elemento **p** da classe Pessoa, cujo valor do atributo **nome** seja o mesmo valor do atributo **alvo** do elemento anterior, então remova o objetivo encontrado da memória de trabalho, imprima uma mensagem dizendo que o valor do atributo pai de **p**, e o valor do atributo mae de **p** são ancestrais, e em seguida crie dois objetivos cujo valor do atributo **alvo** sejam o valor dos atributos **pai** e **mae** de **p**.

Apesar de ter se mostrado como uma grande evolução em relação às arquiteturas existentes na época, OPS5 ainda não podia ser definida como um sistema baseado em objetos, pelas restrições mencionadas anteriormente. Fazendo uma analogia com banco de dados (BD), as ligações entre os objetos OPS5 eram baseadas principalmente no conceito de *chave* de um BD relacional, ou simplesmente uma cadeia de caracteres, conforme apresentado na Figura 1:

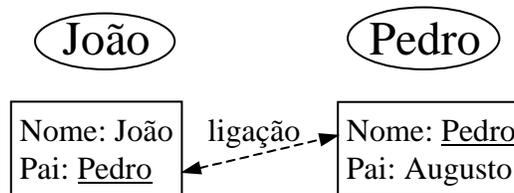


Figura 1 – Ligação simbólica entre objetos

Este modelo, apesar de completo, não é ideal. Se existir mais de uma pessoa com o nome Pedro, alguma outra característica deverá ser usada para distingui-las. Além disso, o valor do atributo “pai” de João não é seu pai, mas o nome dele. Uma extensão do modelo de OPS5, onde o valor dos atributos dos objetos pode ser um objeto foi implementado em diversas ferramentas (algumas das quais iremos apresentar adiante), de modo que a ligação entre os objetos ocorra de forma mais natural. No modelo apresentado na Figura 2, por exemplo, o valor do atributo “pai” de João é realmente o objeto que representa seu pai e não seu nome – como ocorre em sistemas orientados a objetos (OO). A utilização de objetos reais em lugar de tuplas de valores atômicos facilitaria a integração do sistema de produção com uma linguagem de programação orientada a objetos de propósito geral.

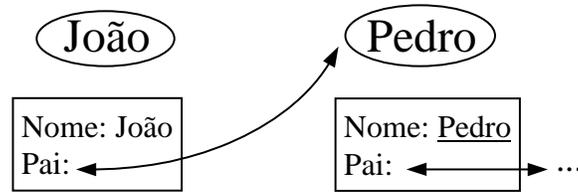


Figura 2 – Ligação natural entre objetos

Uma outra característica importante do OPS5 é a forma como é realizada a unificação entre os objetos da base de fatos e as regras. Ela é realizada através do algoritmo de casamento de padrões Rete [41], que já se tornou um padrão em sistemas de produção, tendo sido implementado (em alguns casos com variações) em praticamente todos os sistemas que encontramos.

Como uma evolução natural de OPS5, o sistema Opus [25] utilizava objetos reais (da linguagem Smalltalk) como elementos da memória de trabalho do sistema de produção. As condições das regras passaram a ser definidas como chamadas a métodos da linguagem; todo e qualquer objeto Smalltalk poderia ser utilizado nas regras. Desta forma, todas as relações ontológicas existentes no mundo Smalltalk puderam ser utilizadas de forma natural pelo sistema, representando um grande avanço na área. Apesar de suas qualidades, Opus foi desenvolvido mais como uma experiência de programação multiparadigma que como um sistema a ser usado em aplicações “reais”, principalmente devido a problemas de eficiência.

2.3.2 Metodologia de desenvolvimento

A utilização de objetos como fatos de um sistema de produção altera a maneira como os sistemas de produção são construídos. Ao invés da existência de um único mundo, onde são definidos tanto as entidades que compõem o problema (os fatos) quanto as regras usadas em sua resolução, a solução do problema passa a envolver dois mundos distintos – o dos objetos e o das regras.

A metodologia utilizada no desenvolvimento de aplicações na área de EOOPS pode ser ilustrada na Tabela 5. Os passos 1 e 2 ilustram uma diferença muito importante em relação a outros sistemas de representação de conhecimento: a separação clara entre a definição das ontologias do problema e das regras de inferência (ao contrário do que ocorre em Prolog [42], onde tanto fatos quanto regras são definidos no mesmo contexto). Após a primeira criação das regras, é comum o desenvolvedor sentir falta de algum serviço dos

objetos. Neste caso, ele retorna ao passo 1, ficando nesta iteração até que as classes definidas sejam suficientemente completas para a definição das regras do problema. Em seguida, as classes criadas no passo 1 são instanciadas, e os objetos criados são inseridos na memória de trabalho do sistema de produção. Finalmente, o sistema de produção é *ativado*, o que faz com que as regras sejam disparadas com os objetos da memória de trabalho que puderam ser unificados com suas variáveis, até que não haja mais nenhuma regra a ser disparada. Uma vez executado, é comum que o desenvolvimento retorne ao início, para ser refinado, ou mesmo estendido com novas funcionalidades.

- | |
|--|
| <ol style="list-style-type: none"> 1. Criar as classes de objetos do domínio do problema 2. Criar as regras que definem o comportamento do problema 3. Se as classes definidas no passo 1 definem tudo o que é necessário para as regras do passo 2, continuar com o passo 4; senão, retornar ao passo 1 4. Criar instâncias das classes definidas no passo 1, inserindo-as na memória de trabalho do sistema de produção 5. Executar o motor de inferência do sistema de produção 6. Retornar ao passo 1, refinando a solução |
|--|

Tabela 5 – Metodologia de desenvolvimento de sistemas EOOPS

Por exemplo, seja o problema da busca por ancestrais, definido na seção anterior. Em primeiro lugar, o usuário deve criar as classes dos objetos do domínio. Para tal, ele deve se perguntar “de que trata o meu problema?”. Basicamente, trata-se de pessoas. A definição de uma classe Pessoa é então o primeiro passo a ser dado na solução do problema. Uma pessoa tem um *nome*, pela qual é identificada, e possui também um *pai* e uma *mãe*, pois estes elementos formam a base da busca por ancestrais. O nome é definido como uma cadeia de caracteres (`String`). O pai e a mãe são instâncias da própria classe `Pessoa`, sendo definidos portanto como auto-relacionamentos da classe. A Figura 3 ilustra a definição da classe `Pessoa`, em notação UML⁴.

⁴ Todos os demais diagramas apresentados neste trabalho também seguirão o padrão de UML. O Anexo A apresenta a notação usada nos diagramas deste trabalho.

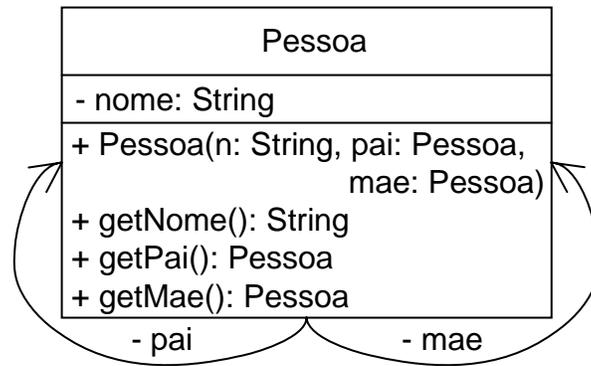


Figura 3 – Classe Pessoa

Definidos os objetos que representam o domínio, o próximo passo é a definição da regra de busca por ancestrais. Neste ponto, surge a necessidade de um novo tipo de objeto. Já é possível representar as pessoas do domínio do problema, mas a busca por ancestrais deve ser feita para uma pessoa em particular, e não para todas as pessoas do mundo (o que poderia tomar um tempo inaceitável). Com isso, o desenvolvedor retorna ao passo 1, para criar uma nova classe de objetos que represente o *objetivo* da busca por ancestrais, conforme apresentado na Figura 4. Cada instância da classe *Objetivo* tem um *alvo*, que é a pessoa cujos ancestrais se deseja encontrar.

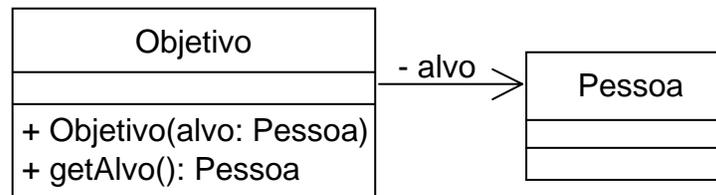


Figura 4 – Classe Objetivo

Criada esta nova classe, volta-se às regras. Neste momento, é possível se criar uma primeira versão da regra de busca por ancestrais, conforme apresentado em pseudocódigo na Tabela 6.

```

Regra EncontraAncestrais
Para todo p da classe Pessoa
Para todo o da classe Objetivo
Se p = o.getAlvo() // Se p é realmente o objetivo da busca
Então
  Escreva que o nome do pai e da mãe de p são ancestrais
  Crie um objetivos o1 cujo alvo é o pai de p
  Crie um objetivos o2 cujo alvo é a mãe de p
  Coloque o1 e o2 na memória de trabalho
  
```

Tabela 6 – Primeira versão da regra EncontraAncestrais

Observando bem a regra da Tabela 6, o usuário nota que após o seu disparo com uma pessoa **p** e um objetivo **o**, as condições da regra continuam verdadeiras: **p** ainda é o alvo de **o**. Para evitar que a regra volte a ser disparada com os mesmos objetos, há duas soluções possíveis: retirar o objetivo da memória de trabalho (como foi feito na regra mostrada na Tabela 4), ou simplesmente *desativar* o objetivo, indicando que o mesmo foi atingido. Para implementar a segunda alternativa, o usuário teria que retornar mais uma vez à definição das classes, para armazenar mais esta informação na classe *Objetivo*, que ficaria conforme a Figura 5.

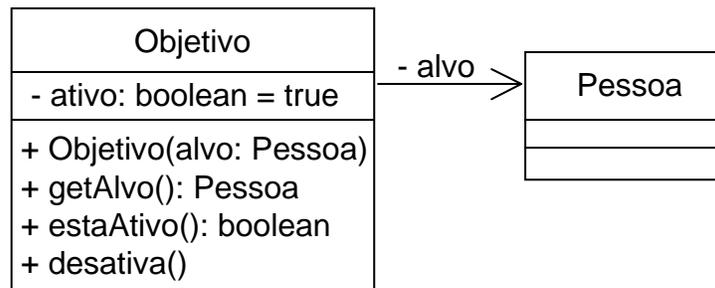


Figura 5 – Refinamento da classe *Objetivo*

Redefinida a classe *Objetivo*, a regra do problema seria alterada, para aproveitar a nova informação que pode ser obtida do objetivo, conforme apresentado na Tabela 7. Deve-se ressaltar que esta iteração continua até que o usuário esteja satisfeito com a definição das classes e das regras do problema.

```

Regra EncontraAncestrais
Para todo p da classe Pessoa
Para todo o da classe Objetivo
Se p = o.getAlvo() // Se p é realmente o objetivo da busca
  e o.estaAtivo() // e o objetivo ainda não obteve sucesso
Então
  Chame o método desativa no objeto o // desativando o objetivo
  Escreva que o nome do pai e da mãe de p são ancestrais
  Crie um objetivos o1 cujo alvo é o pai de p
  Crie um objetivos o2 cujo alvo é a mãe de p
  Coloque o1 e o2 na memória de trabalho
  
```

Tabela 7 – Segunda versão da regra *EncontraAncestrais*

Uma vez que as regras e as classes de objetos estão bem definidas, o usuário deve então criar os objetos do problema (instanciando as classes definidas), e inseri-los na memória de trabalhos, para que as regras possam atuar sobre eles. A Tabela 8 ilustra a criação de alguns objetos da classe *Pessoa*, e um objetivo da busca. Estes objetos devem então ser inseridos na memória de trabalho do sistema de produção.

```

Pessoa avo = new Pessoa("Manoel", null, null);
Pessoa avoh = new Pessoa("Maria", null, null);
Pessoa pai = new Pessoa("Joaquim", avo, avoh);
Pessoa mae = new Pessoa("Rosa", null, null);
Pessoa filho = new Pessoa("Luís", pai, mae);
Objetivo obj = new Objetivo(filho);

```

Tabela 8 – Criação de objetos da classe Pessoa

Finalmente, quando o sistema de produção for *ativado*, seu motor de inferência irá unificar os objetos da memória de trabalho com as declarações da regra, apresentando na saída todos os ancestrais do objetivo da busca. O resultado é apresentado na Tabela 9.

```

Joaquim e Rosa são ancestrais
Manoel e Maria são ancestrais

```

Tabela 9 – Resultado da execução do sistema de produção

2.3.3 Vantagens da integração

A integração de objetos com regras de produção altera a filosofia de trabalho na modelagem e codificação de sistemas de produção. Com efeito, no lugar de simples cadeias de caracteres (com uma certa semântica posicional), os fatos da memória de trabalho são substituídos por entidades que são por definição encapsuladas, cuja estrutura interna não é visível para o mundo exterior, e que pode apresentar diversos relacionamentos com outros objetos. Esta simples substituição de fatos por objetos traz várias vantagens (assim como algumas desvantagens que discutiremos na próxima seção), tanto ligadas a conceitos de qualidade apresentados pela Engenharia de Software (reusabilidade, modularidade, legibilidade, eficiência, manutenibilidade), como relativas à engenharia de conhecimento (representação natural de relações ontológicas).

Por utilizar objetos como fatos de uma base de conhecimentos, o usuário poderá usufruir de todos os serviços embutidos no objeto, que podem ter sido criados por terceiros, ou mesmo pelo próprio usuário em outros projetos. Esta reutilização é, inclusive, incentivada pela filosofia dos sistemas orientados a objetos, conforme apresentado na seção 2.2.

Uma outra vantagem decorrente da utilização de objetos como elementos da memória de trabalho é a legibilidade que se consegue obter nas regras. Em vez de utilizar a posição dos elementos em uma tupla como elemento semântico, as relações entre os objetos podem ser verificadas como métodos da linguagem. Por exemplo, não é claro se o predicado Prolog $\text{pai}(X, Y)$ quer dizer que X é pai de Y ou vice-versa. Este predicado

poderia ser representado em uma linguagem OO por `x.getPai() == y`, ou ainda `x.ehPaiDe(y)`. A facilidade das linguagens orientadas a objetos de se adicionar semântica aos objetos torna as regras escritas em sistemas OO mais compreensíveis⁵. Além disso, como certas operações complexas podem ser transferidas das regras para os próprios objetos, as regras do sistema ficam menores, e em menor número, o que contribui para a legibilidade, a eficiência e para a manutenibilidade do sistema como um todo.

Uma terceira vantagem decorrente da união de dois paradigmas distintos (e não somente de regras e objetos) é a possibilidade do programador escolher para cada módulo do sistema o paradigma que mais lhe convém. Se a integração entre a linguagem e as regras é bem feita, esta escolha pode ser feita de modo que o sistema apresente um alto grau de modularidade. Por exemplo, na implementação de um agente ligado na Internet que executa músicas para o seu usuário, a escolha das músicas a serem carregadas de acordo com as preferências do usuário (e possivelmente outros aspectos, como hora de pico de tráfego da rede, preço das tarifas telefônicas) seria implementada através de regras, enquanto que o protocolo de comunicação utilizado na conexão com os servidores de música seria implementado mais facilmente através de comandos da linguagem orientada a objetos. Se o usuário desejar mudar a maneira que ele escolhe suas músicas, ou se o agente decide utilizar um protocolo de comunicação mais eficiente, estas alterações podem ser feitas sem causar impacto no restante do sistema.

Um outro exemplo pode ilustrar melhor este ponto: supondo que um agente tenha que realizar uma busca complexa em um banco de dados, ou mesmo na Internet, para descobrir se Y é o pai de X – o que pode ser o caso em certos testes de paternidade por DNA – a definição da regra que utiliza o predicado `x.ehPaiDe(y)` não precisaria ser alterada, uma vez que a consulta seria feita pelo próprio objeto. Neste caso, o teste de paternidade poderia ser feito mesmo que para isso o método tivesse de ativar uma nova base de conhecimentos.

Estas vantagens citadas acima podem ser resumidas em apenas uma: qualidade de software. Por permitir que se construa uma aplicação segundo aspectos de reusabilidade,

⁵ Devemos ressaltar que as linguagens OO *facilitam* a legibilidade de código, mas nada impede que o programador escreva o predicado `x.p1(y)` para verificar se y é pai de x, por exemplo.

modularidade, legibilidade, eficiência e manutenibilidade, a possibilidade de sucesso do ciclo de vida do software desenvolvido através da união de regras e objetos aumenta.

As vantagens da integração relativas à engenharia do conhecimento vêm do fato da união entre regras e objetos permitir que o programador separe explicitamente as ontologias do domínio do problema (que podem ser representadas pelas relações entre as classes), e suas regras de inferência. De fato, através da utilização de objetos (estruturas similares aos frames [3], amplamente estudadas pela comunidade de IA), as relações de herança (um tipo de), pertinência (é-um) e agregação (tem-um) podem ser representadas naturalmente. Esta divisão clara entre estes os conceitos de ontologias e inferência vem sendo defendida por certos autores há algum tempo [43], e é importante para que o programador conheça melhor o sistema que está desenvolvendo.

Todas estas vantagens, especialmente às relativas à qualidade do software, nos levam a crer que a integração objetos/regras é interessante, apesar dos problemas que serão apresentados na seção seguinte.

2.3.4 Problemas da integração

As vantagens obtidas através da integração de objetos com regras de produção têm um custo. Apresentaremos sucintamente os problemas criados com esta integração, de modo que, ao apresentar os sistemas existentes, possamos indicar como os problemas são tratados por cada um deles.

2.3.4.1 O problema da modificação dos objetos

Um ciclo de execução de um motor de inferência de um EOOPS pode ser descrito como no pseudocódigo apresentado na Tabela 10.

Para cada regra existente Se existem objetos que tornam suas condições verdadeiras Adicione a regra e os objetos no conjunto de conflitos Selecione uma regra do conjunto de conflitos e execute-a

Tabela 10 – Ciclo de execução de um motor de inferência

O problema desta abordagem é que, se a cada ciclo de inferência todas as combinações de objetos tivessem que ser avaliadas para todas as regras, a degradação da eficiência com o aumento do número de objetos seria tal que inviabilizaria o seu uso. Uma solução para este problema é, a cada ciclo, avaliar apenas aqueles objetos que foram incluídos ou alterados no ciclo anterior, pois as combinações com os demais objetos já

foram avaliadas. A Tabela 11 apresenta o pseudocódigo da etapa de unificação do ciclo de execução de um motor de inferência.

```

Para cada novo objeto o inserido
  Para cada regra r existente
    Se o novo objeto o, possivelmente com outros objetos oi-on,
      torna a regra r disparável
    Adicione r e os objetos (o, oi...on) ao conjunto de conflitos
  
```

Tabela 11 – Algoritmo de unificação de um motor de inferência

Os sistemas orientados a objetos, no entanto, apresentam uma característica que dificulta a detecção de modificação nos objetos. Como objetos são estruturas encapsuladas, não é possível para um agente externo (o motor de inferência) identificar, após alguma invocação de método no objeto, se ele foi alterado ou não. Por exemplo, o sistema não tem como saber se a chamada ao método `tomeRemedioParaPressao()` na regra apresentada na Tabela 12 altera ou não o estado interno do paciente.

```

Regra trataPaciente:
  Declarações
    Paciente p
  Condições
    p.pressao() > 120;
    p.temHistoricoHipertensao();
  Ações
    p.tomeRemedioParaPressao();
  
```

Tabela 12 – Problema da modificação de objetos

Para contornar este problema, conhecido por Problema da Modificação de Objetos ou *Modified Problem*, o sistema de produção tem que decidir entre alguma das seguintes alternativas:

- ***Utilização do algoritmo de unificação da Tabela 10***

A cada ciclo o motor de inferência tenta unificar todos os objetos com todas as regras. Esta opção não é viável para motores de inferência que tenham alguma aspiração de serem usados em sistemas reais de grande porte. No entanto, sua simplicidade a torna ideal para um primeiro estágio de desenvolvimento dos sistemas, sendo posteriormente substituída por uma estratégia mais eficiente.

- ***Quebra do encapsulamento dos objetos***

Desta forma, o sistema é capaz de observar a estrutura interna dos objetos, podendo identificar alterações nos mesmos. A quebra do encapsulamento pode ser feita através da obrigação de que os objetos sejam compatíveis com um

determinado padrão (por exemplo, suas propriedades só podem ser acessadas através de interfaces pré-definidas), ou então pela restrição da chamada de métodos dos objetos. Deve-se observar, no entanto, que a quebra do encapsulamento dos objetos vai totalmente de encontro ao modelo conceitual que define as linguagens OO.

- ***Exigência de que o usuário informe quando um objeto foi modificado***

O desenvolvedor então passa a ter a responsabilidade de notificar o mecanismo de inferência se um objeto foi modificado ou não. Apesar de não alterar a estrutura dos objetos, nem exigir que eles sigam nenhum padrão, esta alternativa cria mais uma preocupação para o usuário: a de decidir quais objetos foram modificados.

- ***Exigência de que o objeto informe ao sistema que foi modificado***

Ao contrário da anterior, esta estratégia exige que os objetos sejam capazes de notificar o motor de inferência de suas próprias modificações. Isto geralmente é feito através da utilização de padrões de projeto já consagrados, como o *Observer* [46] ou *Listener* (dependendo da taxonomia utilizada). No entanto, esta alternativa limita a reutilização de objetos no sistema de produção apenas àqueles capazes de tal notificação.

Os sistemas que decidem não abrir mão da característica de encapsulamento dos objetos, resolvendo o problema da modificação através de um mecanismo de notificação, não ficam totalmente livres do problema da modificação. De fato, uma outra face deste problema é o fato modificações em objetos associados podem não ser facilmente detectadas.

A Figura 6 apresenta um objeto da classe `Pessoa` (que chamaremos de “p”), cujos atributos são o seu nome (um valor `String`), sua idade (um inteiro) e o seu salário (uma instância da classe `Salario`). Após a execução do comando `“p.getSalario().setValor(302.00);”` o objeto da classe `Salario` terá seu valor alterado. Podemos dizer, portanto, que este objeto foi alterado pelo comando, visto que um de seus atributos tinha um valor e após a execução do comando ele passou a ter outro valor. O objeto da classe `Pessoa`, por sua vez, não teve nenhum de seus atributos alterados, pois o seu atributo `salario` continua sendo uma referência para o mesmo objeto. No entanto, é

verdade que houve uma modificação na pessoa: ela recebe um salário maior. Para evitar confusões, definiremos que um objeto foi **modificado diretamente** por um comando se algum de seus atributos tem um valor diferente após a execução deste comando. Por outro lado, dizemos que um objeto foi **modificado indiretamente** por um comando se algum de seus atributos foi direta ou indiretamente modificado pelo comando (em outras palavras, é a definição do fecho transitivo da modificação direta).

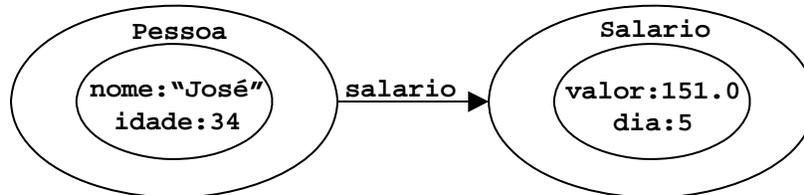


Figura 6 – O problema da modificação de objetos

O problema da identificação de modificações indiretas em objetos, que chamaremos de “Problema da Modificação Transitiva”, ou TMP (de *Transitive Modified Problem*), ainda permanece em aberto. Algumas soluções para contornar este problema serão apresentadas quando introduzirmos os sistemas existentes. Entretanto, deve-se observar que os sistemas que quebram o encapsulamento dos objetos não sofrem deste problema, pois o sistema tem como ver suas estruturas internas, identificando as dependências entre eles.

2.3.4.2 O que colocar nas regras e o que colocar nos objetos?

A vantagem da unificação de dois paradigmas em um único *framework* de desenvolvimento faz com que em algumas ocasiões o desenvolvedor não saiba exatamente o que deve ser feito em um paradigma, o que é melhor ser feito em outro. A união de regras com objetos não é diferente, especialmente se esta união é feita de tal forma a permitir que o desenvolvedor utilize totalmente as potencialidades dos dois paradigmas. Em outras palavras, como diversos problemas podem ser resolvidos tanto via regras de produção (de uma maneira bastante elegante) quanto por métodos embutidos nos objetos, o desenvolvedor não tem critérios claros sobre quando utilizar qual paradigma.

A filosofia de alguns sistemas faz com que este problema seja mais ou menos sentido. Alguns sistemas são criados como ambientes completos de desenvolvimento, de modo que a utilização de serviços fornecidos pela linguagem hospedeira é desencorajada, pela sua dificuldade de implementação. Em outros, onde a idéia é ser uma biblioteca que fornece uma série de funções para a linguagem hospedeira, a utilização dos serviços desta é

facilitada. Neste segundo caso, onde há realmente a possibilidade de utilização tanto de regras quanto de objetos para resolver um determinado problema, é que as dúvidas ocorrem.

Apesar de não existir nenhuma solução amplamente aceita, o que se utiliza atualmente é o bom senso. Regras não são utilizadas em problemas que podem ser resolvidos facilmente através de técnicas convencionais de programação. Problemas de maior complexidade, ou problemas onde se deseja isolar o conhecimento da implementação são candidatos a serem resolvidos através de regras de produção. No entanto, realizar corretamente esta distinção não é tarefa fácil, sendo somente alcançada através da experiência do desenvolvedor.

Este problema também está em aberto, e acreditamos que esta área ainda está carente de propostas de soluções. Uma metodologia consolidada de desenvolvimento orientado a agentes, ou mesmo um catálogo de padrões de projeto (ou *design patterns*) aparecem como um bom caminho para a solução, tal como já foi feito na área de Engenharia de Software [46].

2.3.5 Regras pré-compiladas ou interpretadas

Regras de produção são entidades de representação de conhecimento que não são compreendidas por um computador. Cabe aos sistemas de produção a sua conversão para estruturas que possam ser compreendidas pela linguagem hospedeira. Há basicamente duas maneiras distintas de se implementar esta conversão. Uma delas é a *interpretação das regras* de produção pelo motor de inferência, com este sendo responsável por invocar as operações específicas da linguagem (como operações aritméticas, chamadas a métodos ou criação de novos objetos). Uma segunda abordagem é a *pré-compilação das regras* de produção para estruturas reconhecidas pela linguagem hospedeira.

2.3.5.1 Pré-compilação das regras

Toda compilação nada mais é que uma transformação. Uma compilação da obra de Machado de Assis pode resultar em um resumo de seus melhores livros. Uma compilação dos eventos ocorridos em uma partida de futebol pode resultar nas estatísticas do jogo. Uma compilação das melhores músicas da MPB resultaria em um CD com uma grande procura nas lojas de disco.

Em computação, o termo *compilação* é usado para representar o processo de se transformar um programa escrito em uma determinada linguagem de programação em instruções que possam ser compreendidas pelo computador (código de máquina). A Figura 7 ilustra a compilação do arquivo `Prog.c`, que resulta no programa executável `Prog.exe`.

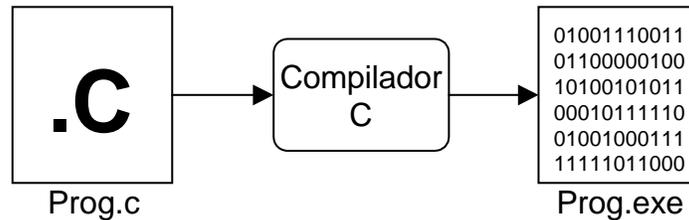


Figura 7 – Compilação de um programa da linguagem C

A compilação de programas de algumas linguagens, como Java, é diferente. Em vez de gerar um arquivo que pode ser executado diretamente pelo computador, o compilador Java transforma um programa da linguagem em uma série de *bytecodes*, instruções de baixo nível que podem ser *interpretadas* por uma *máquina virtual de Java* (ou JVM [44]). Esta característica permite que um programa Java compilado seja executado em diversas plataformas da mesma maneira, garantindo a portabilidade da linguagem. A Figura 8 ilustra o processo de compilação de um programa Java.

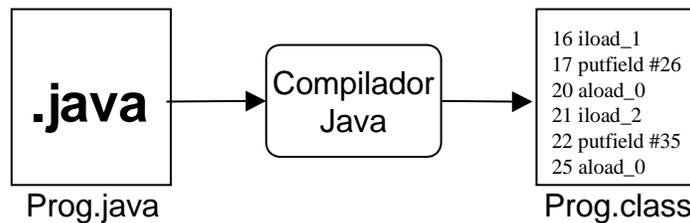


Figura 8 – Compilação de um programa da linguagem Java

A compilação de regras ocorre de maneira similar à compilação de programas. A diferença é que o resultado da compilação das regras é um código fonte da linguagem hospedeira do sistema de produção, que por sua vez será novamente compilado para poder ser executado (seja diretamente pelo computador, seja via uma máquina virtual). Por isso, convencionou-se denominar este processo não de *compilação*, mas de *pré-compilação* das regras. Após esta etapa, as regras podem então ser compiladas pelo compilador da linguagem hospedeira juntamente com o programa que as utiliza, para que seja gerado o código que será executado. A Figura 9 ilustra este processo: as regras do arquivo texto onde foram escritas as regras (`Regras.rules`) são pré-compiladas, gerando o arquivo na

linguagem hospedeira (`Regras.c`). Em seguida, o arquivo gerado e o arquivo que contém a implementação do agente são compilados como programas normais da linguagem C.

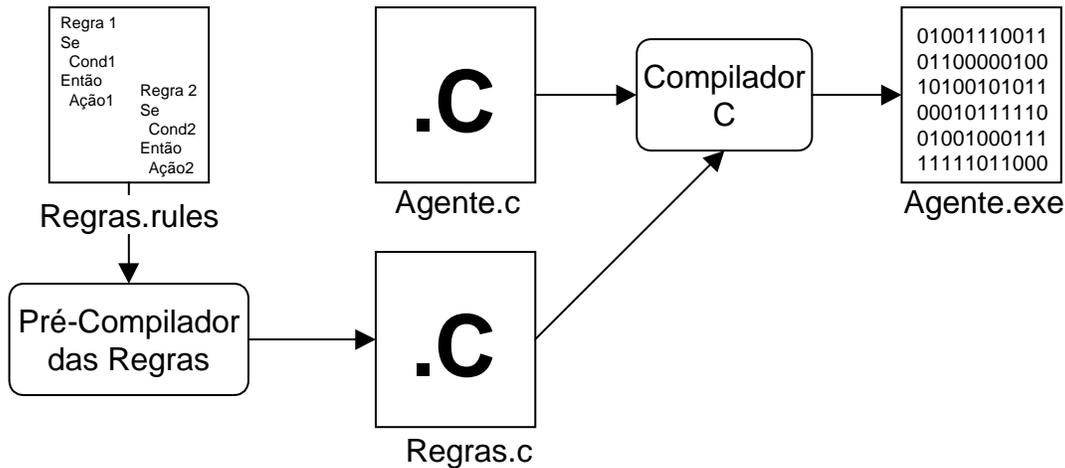


Figura 9 – Pré-compilação de regras

A pré-compilação das regras é uma estratégia muito utilizada nos sistemas de produção, pois ela facilita a implementação do motor de inferência. De fato, se as regras fossem totalmente interpretadas, o próprio motor teria que realizar diversas tarefas que são realizadas pelo compilador, como a avaliação de expressões aritméticas, a definição e utilização de variáveis. Como o código que será executado é gerado pelo compilador, todas as vantagens obtidas pela otimização de código realizada pelo compilador são mantidas. Uma outra vantagem da compilação é o fato de que o código gerado está num nível bem próximo da linguagem de máquina (quando não for o próprio), é executado de forma mais eficiente que se estivesse em um nível mais alto.

A desvantagem da pré-compilação das regras reside no fato de que, uma vez pré-compiladas, a base de regras não pode ser alterada no decorrer da execução da aplicação. Esta limitação, no entanto, é superada em algumas linguagens. Smalltalk, por exemplo, permite que o usuário altere dinamicamente uma classe do sistema, o que torna possível a alteração da base de regras enquanto a aplicação está sendo executada. Algumas máquinas virtuais de Java, como a implementada no ambiente de desenvolvimento Visual Age, da IBM [45], também suportam a alteração dinâmica de classes.

O Anexo G apresenta todos os passos da pré-compilação de uma base de regras JEOPS para uma classe Java.

2.3.5.2 Interpretação das regras

A interpretação de regras em um sistema de produção é o processo pelo qual este lê as regras de alguma fonte (console, arquivo texto, etc.) e armazena-as em uma estrutura interna. Quando da sua execução, o motor de inferência trabalha com esta estrutura, utilizando-a para a unificação das regras, resolução de conflitos, disparo das regras.

A interpretação das regras tem como maior vantagem a possibilidade de se alterar dinamicamente a base de regras do sistema. Como as regras são carregadas em tempo de execução, é possível se realizar uma operação de “*reload*” das regras para que o comportamento do sistema seja alterado. Uma outra vantagem da interpretação das regras é que o sistema de produção fica independente de um compilador da linguagem para funcionar. De fato, sistemas que implementam a interpretação de regras são ideais para a prototipação rápida de aplicações, pois o usuário não tem que ficar compilando um código fonte a cada alteração que é feita.

No entanto, a interpretação das regras pelo motor de inferência impede que sejam realizadas diversas otimizações de performance que poderiam ser feitas pelo compilador da linguagem hospedeira. Além disso, a implementação de um motor de inferência baseado na interpretação das regras é muito mais trabalhoso, pois o sistema passa a ter que realizar a função do próprio compilador em tarefas que não envolvem o motor de inferência, como criação de variáveis, operações aritméticas, entre outras.

2.4 CRITÉRIOS DE ANÁLISE

Antes de apresentarmos os sistemas existentes, nos quais o trabalho comparativo de avaliação de JEOPS foi baseado, apresentaremos alguns critérios de análise destes sistemas. Estes critérios são o desempenho, a expressividade, a possibilidade de encadeamentos, as estratégias de controle e a uniformidade da integração. Com isso, poderemos compará-los com o JEOPS quando este for apresentado.

Para ser utilizado em grandes aplicações, um sistema de produção deve apresentar um bom desempenho. Como o gargalo de processamento de sistemas de produção está na escolha da regra a ser disparada para quais objetos, é este módulo do sistema que determina se ele apresenta um bom desempenho ou não.

A expressividade mede o poder de expressão das regras do sistema. Todos os sistemas que serão apresentados nas próximas seções têm basicamente o mesmo grau de expressividade, equivalente ao da lógica de primeira ordem (com algumas restrições, como a hipótese do mundo fechado, limitações de ordem sintática, etc.). Isto quer dizer que são capazes de definir regras não apenas sobre um único objeto, mas sobre um conjunto de objetos (através da utilização de quantificadores). Os sistemas apresentados, no entanto, dão ao usuário uma maior flexibilidade que sistemas não-monotônicos, pois permitem que um fato seja negado (ou retirado da base de conhecimentos) na conclusão de uma regra (ainda que esta propriedade não esteja bem definida semanticamente).

Em relação aos encadeamentos, a definição clássica de um sistema de produção [13] exige que eles apresentem apenas o progressivo. Alguns sistemas, no entanto, também apresentam o encadeamento regressivo, de modo que o usuário tenha uma maior flexibilidade para o desenvolvimento de aplicações.

As estratégias de controle indicam qual o poder do sistema em definir uma política de resolução de conflitos. Alguns sistemas possuem esquemas simples de definição de estratégias, baseados em prioridade, por exemplo. Outros permitem que o usuário defina, de modo declarativo ou procedimental, estratégias de controle mais complexas.

O quinto critério, e em nossa opinião o mais crítico dado o estado das ferramentas analisadas é a uniformidade da integração. A uniformidade é a medida de quanto o sistema de produção está adequado à filosofia da linguagem hospedeira. Há dois aspectos de uniformidade que devem ser verificados: o sintático e o semântico.

O aspecto sintático da uniformidade é a medida de quão próximas estão as sintaxes da linguagem hospedeira e a utilizada para a definição das regras. Esta característica é importante para que o programador não tenha um impacto grande no aprendizado de uma nova linguagem.

O aspecto semântico mede como as características da linguagem hospedeira (encapsulamento, herança, etc.) são respeitadas pelo sistema de produção. Um sistema de produção não deveria precisar ver a estrutura interna dos objetos. Em decorrência disto, as condições das regras devem ser definidas não a partir da estrutura dos objetos, mas sim de resposta dos objetos a seus métodos. A unificação baseada neste tipo de condição é denominada de unificação comportamental [47], em oposição à unificação estrutural. Um

outro aspecto da uniformidade semântica é a possibilidade do sistema utilizar livremente todos os objetos da linguagem hospedeira, sem impor nenhuma restrição ao programador. Por utilizar um objeto da linguagem, entenda-se que o motor de inferência deve poder unificá-lo com as variáveis das regras, utilizar todos os métodos definidos no objeto ou ainda passá-lo como parâmetro para outras chamadas de métodos.

Como foi discutido anteriormente, um sistema com uma integração uniforme permite a passagem mais suave do mundo da linguagem de programação (com a qual o programador já está acostumado, na maioria das vezes) para o mundo do sistema de produção (reduzindo o seu tempo de aprendizado), além da efetiva implementação de parâmetros de qualidade de software como modularidade, reusabilidade, extensibilidade, e da utilização total de metodologias de desenvolvimento orientadas a objetos.

Apresentaremos a seguir os sistemas mais importantes da área de EOOPS, mostrando que à exceção de NéOpus, implementado para Smalltalk, a questão da uniformidade da integração vem sendo negligenciada.

2.5 CLIPS

Por volta de 1985, a NASA decidiu unificar o desenvolvimento interno de sistemas especialistas. O sistema usado até então, ART [48], não estava definido para todas as plataformas utilizadas pela companhia, especificamente PCs e Macintoshs. Como resultado, o setor de tecnologia de software do Johnson Space Center criou um clone das capacidades de encadeamento progressivo e da sintaxe de ART, introduzindo o CLIPS (*C Language Integrated Production System*) [21] [22] [23] no domínio público.

CLIPS rapidamente conseguiu um grande número de usuários, principalmente no meio acadêmico. Sua sintaxe, parecida com a de OPS5, em nada se assemelhava à da linguagem hospedeira (C). Entretanto, a sua semelhança com a sintaxe de outras linguagens usadas pela comunidade de Inteligência Artificial (notadamente LISP) ajudou na proliferação de seus usuários.

2.5.1 Apresentação

A representação de conhecimento em CLIPS pode ser feita de três formas: regras (de produção), usadas principalmente para a representação de conhecimento heurístico baseado em experiência; funções específicas e funções genéricas, para representação de

conhecimento procedimental; ou programação orientada a objetos, também usada na representação de conhecimento procedimental (com uma certa carga declarativa, dada a possibilidade da definição de relações entre as classes). Apesar de poder ser usado em conjunto com linguagens de programação convencionais (como C, Ada ou C++), CLIPS define uma linguagem de programação própria (COOL, ou *CLIPS Object-Oriented Language*). De fato, uma aplicação pode ser definida inteiramente em CLIPS (ou COOL), que também provê funções de linguagens convencionais, tais como funções de entrada/saída e formatação de texto. Com esta concepção, o mundo de objetos CLIPS pode ser completamente dissociado do mundo dos objetos da linguagem de programação hospedeira.

Em CLIPS há uma diferença entre tipos estruturados de dados e classes, de forma similar à diferença entre os construtores **struct** e **class** em C++. Tipos, ou *templates*, definem um agrupamento de valores, como o construtor **literalize** de OPS5 (com algumas novas funcionalidades, como a possibilidade do valor dos atributos serem fatos, e não apenas valores atômicos). O exemplo da Tabela 13 ilustra a definição do tipo `Pessoa`, tal como definido anteriormente no exemplo do OPS5.

```
(deftemplate Pessoa
  "Uma pessoa em particular"
  (slot nome)
  (slot pai)
  (slot mae))
```

Tabela 13 – Definição do tipo `Pessoa`

Fatos são adicionados à memória de trabalho através da utilização do comando **assert**. Da mesma forma que OPS5, atributos não inicializados recebem o valor nulo, representado **nil**. A Tabela 14 ilustra a criação de dois fatos do tipo `Pessoa`.

```
(assert (Pessoa (nome "Joao")))
(assert (Pessoa (nome "Pedro")))
```

Tabela 14 – Criação de fatos do tipo `Pessoa`

Classes são definidas através do construtor **defclass**, através do qual o usuário pode informar maiores detalhes sobre a estrutura sendo criada, como a hierarquia da classe sendo criada (através do comando **is-a**), o tipo da classe (i.e., se ela é abstrata ou concreta), métodos de acesso para os campos, valores *default*, entre outros. Além disso, ainda é possível se definir novas mensagens a que objetos da classe irão responder (ou métodos de acesso, dependendo da taxonomia utilizada). A Tabela 15 ilustra a criação de uma classe

Pessoa, que responde, além das mensagens de acesso à seus atributos, à mensagem “print-nome”.

```
(defclass Pessoa (is-a USER) (role concrete)
  (slot nome (create-accessor read-write))
  (slot pai (create-accessor read-write))
  (slot mae (create-accessor read-write))
  (defmessage-handler Pessoa print-nome ()
    (printout t "Nome: " (send ?self get-nome) crlf)))
```

Tabela 15 – Definição da classe Pessoa

Objetos CLIPS são criados através do comando **make-instance**. A contrário dos fatos, os objetos criados possuem um *nome*, pelo qual podem ser referenciados posteriormente. A Tabela 16 ilustra a criação de duas instâncias da classe Pessoa.

```
(make-instance [Jose] of Pessoa (nome "Jose"))
(make-instance [Jesus] of Pessoa (nome "Jesus") (pai [Jose]))
```

Tabela 16 – Criação de instâncias da classe Pessoa

Fatos e objetos convivem em uma certa harmonia no mundo CLIPS. Classes e *templates* podem ter o mesmo nome, em um dado instante podem existir na memória de trabalho tanto fatos e objetos, as regras do sistema podem filtrar tanto fatos quanto objetos, enfim, os dois mundos (fatos e objetos) coexistem naturalmente em CLIPS.

Regras CLIPS são definidas de forma similar a OPS5, através do comando **defrule**. O código abaixo ilustra o exemplo da busca por ancestrais, utilizando apenas fatos:

```
(defrule encontraAncestral
  "Cria sub-objetivos para o pai e a mãe"
  ?m <- (Pessoa (nome ?nomeMae))
  ?p <- (Pessoa (nome ?nomePai))
  ?f <- (Pessoa (nome ?nomeFilho) (pai ?p) (mae ?m))
  ?o <- (Objetivo (alvo ?f&:(<> f nil)))
=>
  (retract ?o)
  (printout t ?nomePai " e " ?nomeMae
            " sao ancestrais" crlf)
  (assert (Objetivo (alvo ?p)))
  (assert (Objetivo (alvo ?m)))
)
```

Tabela 17 – Regra encontraAncestrais (CLIPS)

O comando **printout** de CLIPS é usado para o envio de dados para a saída, no dispositivo especificado em seu primeiro argumento. A saída padrão (console) é representada por **t**.

O controle de disparo das regras CLIPS é baseado em uma das 7 estratégias de resolução de conflitos suportadas pelo sistema. Para cada regra, pode-se especificar uma

prioridade (ou *saliência*, no vocabulário de CLIPS), e dentre as regras de mesma saliência, o sistema utiliza a estratégia escolhida para escolher a próxima regra a ser disparada. As estratégias existentes incluem a de profundidade (em que regras recém ativadas têm maior prioridade que as mais antigas), a de largura (o contrário da anterior, apesar infeliz escolha de terminologia), a de simplicidade (em que regras em que precisam ser feitas menos comparações adquirem maior prioridade), a de complexidade (o contrário da anterior), a aleatória, a MEA (onde regras com objetos mais recentes adquirem maior prioridades) e a LEX (o contrário da anterior).

2.5.2 Crítica

CLIPS é uma referência importante em EOOOS por diversos motivos. Foi o primeiro sistema amplamente aceito (especialmente no meio acadêmico) que superasse as limitações de OPS5. Através da utilização do algoritmo de casamento de padrões Rete, o sistema apresentava um ótimo desempenho para aplicações de porte razoável. Além disso, seu número de usuários cresceu bastante, talvez por ser patrocinado por um órgão forte no desenvolvimento de sistemas especialistas, a NASA. No entanto, com o corte de fundos para o desenvolvimento da linguagem há alguns anos, CLIPS raramente é usado para fins comerciais, principalmente devido à falta de suporte [48]. O sistema ainda é, todavia, bastante usado em aplicações acadêmicas, especialmente em cursos de graduação.

A integração de CLIPS com as linguagens hospedeiras, apesar de existir, está longe de ser completa. A sintaxe das regras está muito mais próxima à de LISP que a de C, dificultando o seu aprendizado por programadores desta linguagem (problema conhecido por *Impedance Mismatch* [20]).

Um outro problema de integração é o fato dos objetos do sistema existirem em um mundo à parte do mundo dos objetos da linguagem, apesar de poder haver ligação entre eles. Isso é decorrente da filosofia de CLIPS de que tudo pode ser feito em COOL. O grande problema desta abordagem (que é o mesmo da criação de uma nova linguagem) é que todos os serviços necessários para o desenvolvimento de aplicações tem que ser reconstruídos nesta linguagem. Mesmo utilizando os serviços da linguagem hospedeira, o usuário tem que realizar alguma conversão entre os dois mundos, dificultando a reutilização. E como o suporte a linguagens *mainstream* é muito mais forte que o suporte a

CLIPS, por exemplo, os serviços desenvolvidos para esta última terminam por ficar defasados em relação aos das linguagens convencionais.

2.6 RAL/C++

Desenvolvido por Charles Forgy, o criador do algoritmo Rete, RAL/C++ [24] é uma evolução do sistema RAL/C. O sistema apresenta diversas vantagens em relação aos antecessores que visavam integrar regras a C (especialmente CLIPS). Com a filosofia de permitir que o usuário utilize a própria sintaxe de C++ para especificar as ações das regras, o sistema representa um avanço na direção de uma integração total entre regras de produção e a linguagem C++.

2.6.1 Apresentação

RAL/C++ é descrito como “um dialeto estendido de C++”. Sua idéia é adicionar à linguagem um suporte a regras de produção de modo natural ao programador. Por ser implementado como um pré-processador que converte regras em código C++, as aplicações que utilizam RAL/C++ mantêm a portabilidade das aplicações desenvolvidas nessa linguagem.

Em RAL, o lado direito de uma regra é sintaticamente e semanticamente idêntico a um corpo de função qualquer de C++. Como é definido pelo autor, tudo o que pode ocorrer em uma função de C++ pode ocorrer do lado direito da regra, e vice-versa (com apenas alguns novos modificadores para se criar e trabalhar com elementos da memória de trabalho). Por exemplo, se em CLIPS o usuário utiliza a expressão “(bind ?x (+ ?x 1))” para incrementar o valor de uma variável, em RAL/C++ ele poderia utilizar “x = x + 1;” ou simplesmente “x++;”. O lado esquerdo das regras, no entanto, assemelham-se às condições de OPS5, com a diferença de que expressões C++ também podem ser usadas. A Tabela 18 ilustra uma variação da regra dos ancestrais na sintaxe de RAL/C++.

```

EncontraAncestral {
    // Cria sub-objetivos para o pai e a mãe
    Mae (Pessoa nome::NomeMae;)
    Pai (Pessoa nome::NomePai;)
    Filho (Pessoa nome::NomeFilho; mae==Mae; pai==Pai;)
    Obj (Objetivo alvo==Filho; status==TRUE)
-->
    // imprime na saída padrão os ancestrais.
    printf ("%s e %s são ancestrais.\n", NomeMae, NomePai);
    Objetivo Ob1(Mae);
    Objetivo Ob2(Pai);
    assert Ob1;
    assert Ob2;
    modify Obj {
        Obj->Desativa();
    }
}

```

Tabela 18 – Regra EncontraAncestral (RAL/C++)

O problema da modificação dos objetos é tratado em RAL/C++ através da transferência para o usuário da responsabilidade de notificar o motor de inferência que um objeto será modificado em um determinado bloco. No trecho de código acima, o usuário teve de informar ao sistema que a chamada ao método `Desativa` altera o estado do objeto `Obj`.

Os elementos da memória de trabalho de RAL/C++ são variáveis de tipo **struct** ou **class** de C++. No entanto, deve ser utilizada uma palavra-chave na declaração do tipo para se indicar ao tradutor RAL → C++ quais estruturas estarão na memória de trabalho. Por exemplo, a Tabela 19 ilustra uma declaração do tipo `Empregado` como subclasse de `Pessoa`, com o atributo adicional `salario`.

```

wmedef Empregado:Pessoa {
    double salario;
}

```

Tabela 19 – Declaração do tipo `Empregado`

RAL/C++ diz suportar todos os tipos de C++ (incluindo os tipos definidos pelo usuário). Isto quer dizer que todo tipo de C++ pode ser usado como campos de objetos da memória de trabalho, ou ainda passado como parâmetro para algum método. No entanto, este suporte não é total, pois somente os objetos de tipos definidos com o modificador `wmedef` podem ser adicionados à memória de trabalho.

RAL/C++ utiliza o algoritmo de casamento de padrões `Rete II™` [49], uma versão otimizada de `Rete` que lida melhor com sistemas onde os dados mudam muito rapidamente.

Isto faz com que o sistema apresente um ótimo desempenho, como pode ser ilustrado em diversos *benchmarks* [50].

2.6.2 Crítica

RAL/C++ representa um grande avanço em direção a uma integração uniforme entre regras e objetos. A característica de utilização da própria sintaxe da linguagem hospedeira na escrita da parte direita das regras facilita a compreensão das mesmas pelos programadores com experiência em C++ – em número muito maior que aqueles com experiência no desenvolvimento de sistemas especialistas. Sua eficiência supera a dos demais sistemas (ao menos nos *benchmarks* apresentados pela empresa que o desenvolveu).

No entanto, os autores de RAL/C++ preferiram manter a mesma sintaxe de OPS5 na definição das condições das regras. O resultado desta escolha é que as regras do sistema aparentam estar divididas em duas partes que não se encaixem perfeitamente, tal como uma “colcha de retalhos”.

Um outro problema do sistema é que RAL/C++ utiliza o casamento de padrões baseado no valor dos atributos dos objetos da memória de trabalho. Em outras palavras, o sistema não respeita o encapsulamento dos objetos, impedindo que atributos privados de objetos sejam utilizados na unificação (ainda que seus valores pudessem ser obtidos via métodos de acesso).

Por exemplo, se a classe `Pessoa` (superclasse da classe `Empregado` apresentada na Tabela 19), for definida conforme a Figura 3 (Cf. seção 2.3.2), os elementos da classe `Empregado` possuirão como atributo o seu nome (lembrando, um `Empregado` é-uma `Pessoa`). No entanto, este atributo, por ser escondido (privado), não pode ser utilizados no casamento de padrões, apesar de existir um métodos de acesso (`getNome()`) através do qual o usuário poderia obter seu valor.

Um terceiro problema do sistema é a obrigação de que os objetos que precisem fazer parte da memória de trabalho sejam declarados com um modificador especial. Esta característica limita o uso de classes definidas por terceiros (como componentes reutilizáveis de código). Esta limitação pode, no entanto, ser contornada, através da definição de subclasses das classes desejadas, mas este paliativo não é natural para o desenvolvedor.

2.7 NÉOPUS

Uma outra proposta para a integração entre regras e objetos foi feita por François Pachet em 1992 com o sistema NéOpus [26], uma extensão do sistema Opus [25]. NéOpus é um sistema de produção definido para a linguagem Smalltalk, com características que influenciaram sobremaneira o desenvolvimento de JEOPS.

2.7.1 Apresentação

A idéia por trás de NéOpus é a integração completa entre o sistema de produção e a linguagem hospedeira. O sistema, que utiliza a pré-compilação de regras, é baseado nos seguintes princípios fundamentais:

- Toda expressão booleana Smalltalk pode ser usada como condição das regras, e toda expressão Smalltalk pode ser usada em suas ações;
- Todo objeto Smalltalk pode ser utilizado pelo motor de inferência para ser unificado com as declarações das regras.

O acesso aos atributos dos objetos utilizados nas regras não é mais feito através simplesmente do nome dos atributos; como estruturas encapsuladas, os objetos devem prover mecanismos de acesso a seu estado interno (i.e., métodos). Com isso, as condições das regras NéOpus passam a ser representadas por expressões booleanas da linguagem hospedeira. As restrições entre os objetos filtrados, antes definidas apenas por comparações entre os valores dos atributos, passam a ser definidas por mensagens Smalltalk, ou relações entre estas mensagens (ou seja, NéOpus utiliza a unificação comportamental, conforme apresentada na seção 2.4).

A regra da busca por ancestrais, mostrada na Tabela 20, ilustra bem a integração entre o sistema e a linguagem hospedeira. As declarações do sistema são definidas como declarações Smalltalk. Como os nomes do pai e da mãe da pessoa procurada podem ser obtidos a partir desta, não há a necessidade de declará-los na regra. As ações são também expressões da linguagem, com alguns modificadores para se inserir, remover ou notificar da modificação de elementos da memória de trabalho.

```

EncontraAncestrais
| Pessoa p. Objetivo o |
o alvo = p.
o estaAtivo.
Actions
| o1 o2 |
o1 := Objetivo new alvo: p pai.
o2 := Objetivo new alvo: p mae.
Transcript show: p pai nome; ' e ';
p mae nome; ' sao ancestrais'.
o1 go. o2 go.
o desativa.
o modified.

```

Tabela 20 – Regra `EncontraAncestrais` (NéOpus)

O campo das declarações de variáveis da regra (logo após o nome da regra, entre barras verticais, de acordo com a sintaxe de Smalltalk) declara que as variáveis **p** e **o** são quantificadas universalmente, e o sentido da condição poderia ser lido como “para todo objeto **p** da classe Pessoa, e para todo objeto **o** da classe Objetivo, se a resposta da mensagem ‘alvo’ enviada a **o** for igual a **p**, e o status de **o** for ativo, então...”.

NéOpus, como RAL/C++, funciona como um pré-compilador integrado ao ambiente de trabalho Smalltalk. Quando o usuário carrega uma base de regras, ela é convertida em diversos métodos da linguagem de maneira transparente. Como em RAL/C++, o usuário tem a função de notificar o sistema quando da modificação de um objeto. O problema da modificação transitiva de objetos é contornado através da obrigação do usuário em declarar em uma regra todas as variáveis que, se modificadas, deveriam fazer com que as condições da regra sejam verificadas novamente – mesmo que estes valores possam ser obtidos a partir de outras declarações.

Quanto às estratégias de resolução de conflitos, o usuário de NéOpus pode definir, de forma declarativa, as prioridades no disparo das regras. Através de uma *meta* base de regras, o sistema é informado sobre como implementar a política de resolução de conflitos. Várias estratégias já vem codificadas no sistema, como a de prioridades, a MEA (que prioriza elementos do conjunto de conflitos que utilizam objetos mais recentes), entre outras.

2.7.2 Crítica

NéOpus apresenta uma grande evolução na área de EOOps, conseguindo uma integração quase total – e bastante transparente – entre um sistema de produção e uma linguagem orientada a objetos.

No entanto, a linguagem para a qual o sistema foi desenvolvido, Smalltalk, apresenta uma tendência de afastamento do *mainstream* da computação⁶, principalmente por sua falta de suporte a aplicações para a Internet. Com isso, a possibilidade de reutilização de componentes do sistema fica bastante diminuída, apesar de toda a elegância da linguagem.

Algumas características do sistema tornam inviável sua conversão direta para uma outra linguagem. Utilizando-se de características de Smalltalk, NéOpus redefine algumas operações da raiz da hierarquia de classes da linguagem (Object), adicionando operações de manipulação da memória de trabalho. Esta característica, no entanto, inviabiliza ainda mais a utilização do sistema em aplicações para a Internet, pois seria uma fonte muito grande de problemas de segurança.

2.8 JESS

No início de 1997, o pesquisador Ernest J. Friedman-Hill, do laboratório Sandia da Califórnia, criou um clone de CLIPS para a linguagem Java. Foi a primeira proposta divulgada de integração entre sistemas de produção e a linguagem, que na época já aparecia como uma idéia viável, principalmente pelo apelo da Internet.

2.8.1 Apresentação

A idéia de JESS (*The Java Expert System Shell*) [27] é a conversão de um subconjunto (inicialmente) de CLIPS para Java, mantendo a mesma filosofia do sistema desenvolvido pela NASA. Inicialmente, o sistema apresentava diversas limitações: não suportava naturalmente a herança de objetos, apresentava diversos *bugs* documentados, entre outras. Entretanto, a versão atual (6.0a2) já resolveu grande parte destes problemas, possibilitando, inclusive, a utilização de raciocínio com encadeamento regressivo, serviço não suportado por nenhum dos outros sistemas estudados.

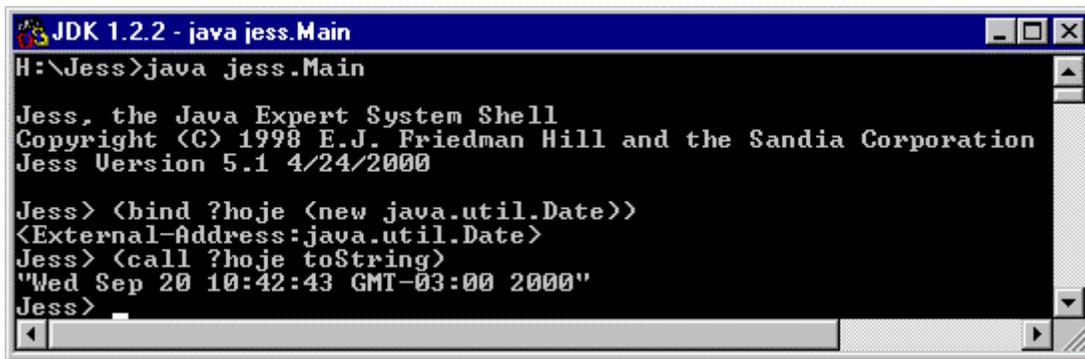
⁶ Uma comparação entre hits em páginas de procura de empregos na Web ilustra este ponto.

A popularidade de JESS cresceu grande parte devido à base de usuários de CLIPS que desejavam migrar de soluções baseadas em C/C++ para Java (e utilizar os serviços oferecidos por esta linguagem). Existem milhares de usuários do sistema, que conta com uma lista de discussão [51] relativamente movimentada, fazendo com que o suporte ao uso deste sistema seja bastante efetivo.

Assim como CLIPS, JESS define toda uma linguagem para ser usada na criação de aplicações, e não apenas de regras. Comandos de criação de interfaces gráficas, comunicação, ou entrada/saída são disponibilizados nesta linguagem, enfim, toda uma aplicação pode ser desenvolvida dentro do *shell* de JESS.

JESS pode utilizar como elementos de sua memória de trabalho objetos Java. No entanto, existe uma restrição para os objetos Java que podem ser usados. Como não existe nenhuma forma explícita de se avisar ao sistema quando um objeto é modificado (como o **modified** de NéOpus ou o **modify** de RAL/C++), os objetos devem ser capazes de avisar quando uma de suas propriedades é alterada. Em outras palavras, para uma propriedade poder ser utilizada na verificação da condição de alguma regra, ela deve ser uma *bound property* [52], de acordo com a nomenclatura do padrão de componentes de Java, JavaBeans [53].

O sistema JESS utiliza a estratégia de interpretação de regras, fazendo com que ele seja também um excelente ambiente de *scripting* para Java. A partir da sua linha de comando, usuário é capaz de criar objetos Java ou invocar métodos sem ter que compilar nenhuma linha de comando. Por exemplo, na seção de JESS apresentada na Figura 10, um objeto da classe `java.util.Date` de Java é criado, e nele é invocado o método `toString()`, sem a necessidade de se compilar o código Java.

A screenshot of a Windows command prompt window titled "JDK 1.2.2 - java jess.Main". The window shows the execution of the Java command "java jess.Main". The output displays the JESS shell interface, including the version information "Jess Version 5.1 4/24/2000" and the execution of the commands "(bind ?hoje <new java.util.Date>)" and "(call ?hoje toString)", resulting in the date string "Wed Sep 20 10:42:43 GMT-03:00 2000".

```
JDK 1.2.2 - java jess.Main
H:\Jess>java jess.Main

Jess, the Java Expert System Shell
Copyright (C) 1998 E.J. Friedman Hill and the Sandia Corporation
Jess Version 5.1 4/24/2000

Jess> (bind ?hoje <new java.util.Date>)
<External-Address:java.util.Date>
Jess> (call ?hoje toString)
"Wed Sep 20 10:42:43 GMT-03:00 2000"
Jess>
```

Figura 10 – Scripting via JESS

2.8.2 Crítica

Apesar de ser o primeiro sistema especialista definido para Java, JESS sofre dos mesmos problemas de CLIPS, basicamente a baixa integração do sistema com a linguagem hospedeira. Ainda existe a separação dos mundos de objetos Java e o de objetos (ou fatos) JESS, e a sintaxe utilizada na definição das regras em nada se assemelha à da linguagem hospedeira, fazendo com que o usuário, tipicamente um programador Java, tenha que passar um tempo maior no entendimento do sistema.

A necessidade que os objetos sejam capazes de avisar sobre modificações ocorridas em suas propriedades também limita a utilização do sistema, que fica impossibilitado, por exemplo, de utilizar algumas classes definidas por terceiros (aquelas que não notificam a mudança de suas propriedades), uma prática bastante comum no desenvolvimento orientado a objetos.

A eficiência de JESS também é um pouco prejudicada por ser um sistema interpretado: ele não toma proveito dos compiladores JIT [54] que já se tornaram padrão para Java. A utilização de um mundo de objetos a parte dos objetos de Java também causa problemas para JESS. Um String em JESS consiste em um objeto da classe `jess.Value` que contém uma String de Java. Com isso, uma chamada ao método `substring` em Java cria um objeto como resultado, enquanto que em JESS dois objetos são criados. O número 5 em Java é um tipo primitivo – pode ser armazenado em um registro – enquanto que em JESS é um objeto que contém o número 5 nele. Com isso, se em Java a operação “2 + 3” não cria nenhum objeto, (+ 2 3) em JESS cria um. Como a criação de objetos é uma operação bastante cara em Java, a performance do sistema fica prejudicada.

2.9 OUTRAS INTEGRAÇÕES DE MECANISMOS DE RACIOCÍNIO COM OBJETOS

Este trabalho foi desenvolvido com foco na integração entre regras e objetos no contexto de EOOPS (ver Figura 11a). Nesta abordagem, um mecanismo de inferência é embutido na linguagem, como um serviço prestado aos demais objetos desta linguagem. No entanto, existem outras abordagens para a integração de raciocínio a objetos, conforme ilustra a Figura 11b e a Figura 11c.



Figura 11 – Integração de raciocínio com Orientação a Objetos

ObjLog [55] é uma extensão de Prolog, linguagem conhecida pela maioria da comunidade de Inteligência Artificial, com os conceitos de orientação a objetos. A linguagem apresentava uma boa implementação dos diversos conceitos de orientação a objetos, como herança, envio de mensagens, agregação de objetos, entre outros. LIFE [56], por sua vez, estende Prolog com alguns conceitos dos paradigmas orientado a objetos e funcional, além de programação por restrições qualitativas. LIFE apresenta uma boa integração entre os paradigmas, com um mecanismo de unificação baseado em interseção de tipos bastante poderoso. Apesar de serem muito interessantes do ponto de vista conceitual, estas linguagens não obtiveram sucesso comercial, provavelmente pela falta de serviços (como discutimos na seção 2.3.3). Atualmente, seu uso praticamente se restringe ao desenvolvimento de aplicações acadêmicas.

2.9.1 Integração Java x Prolog

Para resolver o problema da falta de serviços das linguagens que integravam conceitos de orientação a objetos à lógica, uma arquitetura diferente das apresentadas na Figura 11 foi proposta. A partir do sucesso de Java, diversas equipes desenvolveram

produtos que tentassem integrar Prolog com Java. A motivação desta integração é permitir que o mecanismo de busca e inferência de Prolog seja usado em Java, e/ou permitir que os serviços oferecidos por Java sejam utilizados em Prolog.

Apresentaremos a seguir as arquiteturas em que são desenvolvidos os sistemas que integram Java e Prolog, seguida de uma breve descrição de alguns deles.

2.9.1.1 Arquiteturas

Diversas abordagens foram utilizadas na integração entre Java e Prolog. A escolha da arquitetura define diversas das características do sistema que a implementa, motivo pelo qual acreditamos que seja interessante a apresentação das arquiteturas existentes antes dos sistemas em si. Estas são as arquiteturas que encontramos nos sistemas estudados:

- ***Tradução Prolog → Java***

É a abordagem mais simples. Trata-se de um pré-compilador que traduz os predicados de Prolog em variáveis e métodos de Java. O Anexo F traz um exemplo de como poderia ser feita uma conversão entre um programa Prolog e uma classe Java.

- ***Emulação de Prolog em Java***

Trata-se da implementação de diversas classes que simulem o comportamento de um ambiente Prolog em Java. Estas classes implementam as funcionalidades do mecanismo de busca e de inferência de Prolog para que possam ser utilizadas por outras classes em Java.

- ***Comunicação da máquina virtual de Java (JVM) com o ambiente de execução de Prolog (PRE)***

É uma implementação onde algumas classes de Java são construídas de modo a servirem de ponte para um motor de Prolog já existente. A comunicação entre a JVM e o PRE pode ser feita através de *sockets*, ou via algum sistema que integre uma outra linguagem (como C) com Prolog, através da interface definida por Java para comunicação com outras linguagens (JNI [57]). Esta alternativa mostrou-se bastante comum, provavelmente devido à facilidade de implementação, já que não é preciso implementar todo um motor de inferência, reutilizando-se o que já existe.

2.9.1.2 Prolog Cafe

Prolog Cafe [58] é um sistema tradutor de código fonte Prolog para código fonte Java. O sistema é baseado num método de tradução que cobre um superconjunto de Prolog baseado em lógica intuicionista. O sistema apresenta uma boa integração entre as linguagens, com a possibilidade do código Prolog chamar métodos, construtores, acessar termos de objetos Java, entre outras construções da linguagem. Prolog Cafe apresenta um desempenho razoável, sendo melhor que o sistema jProlog, no qual foi baseado.

2.9.1.3 JavaLog

JavaLog [59] é um interpretador de Prolog escrito em Java, com a intenção de prover um ambiente no qual os paradigmas de programação orientado a objetos e lógicos pudessem coexistir. Inteiramente desenvolvido em Java, JavaLog é um emulador de Prolog, de modo que uma aplicação Java possa resolver naturalmente problemas que requerem algum tipo de inferência lógica. Uma segunda parte do sistema permite que programas Prolog utilizem objetos Java. Estes objetos podem ser criados dentro do ambiente Prolog (pela utilização do predicado **new**), ou passados para o ambiente Prolog por uma chamada de Java.

O sistema peca em alguns aspectos por não suportar todos os construtores da linguagem Prolog em sua implementação em Java (como o *cut*), e também por não suportar todos os construtores de Java em Prolog (como o tratamento de exceções no envio de mensagens aos objetos). Além disso, as consultas à implementação de Prolog são feitas a partir de comandos embutidos no código fonte através de símbolos especiais, sendo preciso que se realize um pré-processamento antes da compilação do código Java.

2.9.1.4 Jasper

Jasper [60] é uma interface bidirecional entre Java e o SICStus Prolog [61]. A parte Java consiste em um pacote de classes que representam o emulador Prolog. A parte Prolog consiste de um módulo de biblioteca e uma extensão de linguagem externa. Jasper na verdade é uma camada desenvolvida sobre uma integração entre C e o SICStus Prolog. As chamadas Java são feitas via métodos nativos (i.e., compilados para uma plataforma específica, e não para a JVM) desenvolvidos em C, e em seguida repassados para o PRE.

A integração a partir de Java é bem feita. No exemplo da Tabela 21, o programa `familia.pl`, apresentado no Anexo F, é carregado, e é realizada uma consulta ao predicado `avo`. Apesar de não ser uma solução muito concisa, a solução Jasper não envolve nenhum passo adicional para se utilizar o motor de inferência de Prolog, e é relativamente fácil de se compreender.

```
import se.sics.jasper.*;
public class Teste {
    public static void main(String[] args) {
        try {
            SICStus sp = new SICStus(args, null);
            sp.load("familia.pl");
            SPPredicate pred = new SPPredicate(sp, "avo", 2, "");
            SPTerm neto = new SPTerm(sp, "joao");
            SPTerm avo = new SPTerm(sp).putVariable();
            SPQuery query = sp.openQuery(sp, new SPTerm[] {neto, avo});
            while(query.nextSolution()) {
                System.out.println(avo.toString() + " e' avo de joao.");
            }
        } catch (Exception e) {}
    }
}
```

Tabela 21 – Utilização do sistema Jasper

2.9.1.5 Crítica

A integração entre Java e Prolog é uma alternativa bastante viável para que todo o legado de aplicações e sistemas especialistas desenvolvidos em Prolog não seja perdido. Diversas implementações foram feitas com este propósito [62], e o campo ainda está aberto para novas propostas. A comunidade de programadores Prolog que têm conhecimento de Java podem tirar grande proveito destes sistemas, integrando todos os serviços oferecidos por Java às aplicações que desenvolvem.

No entanto, as aplicações criadas utilizando estas ferramentas requerem que quem as desenvolva conheça duas linguagens de programação de propósitos e – principalmente – filosofias completamente distintos. Profissionais com este perfil não são facilmente encontrados no mercado, fazendo com que esta solução seja muito mais adequada para programadores Prolog que para programadores Java. Como o nosso trabalho tem por objetivo fornecer uma ferramenta justamente para estes últimos, estes sistemas não podem ser considerados como “concorrentes” de JEOPS.

2.10 SÍNTESE

Diversas tentativas já foram feitas para a integração de capacidade de inferência e linguagens orientadas a objetos. Procuramos com este capítulo apresentar as principais delas, de modo a fornecer ao leitor o contexto em que JEOPS foi desenvolvido, bem como os problemas existentes da área e algumas das soluções apresentadas para eles. Podemos notar que, apesar de tais esforços, ainda não existe nenhum sistema que apresente uma integração uniforme entre uma linguagem orientada a objetos verdadeiramente comercial como Java e um mecanismo de inferência. Tentamos, com JEOPS, unir as principais vantagens destes sistemas, aproveitando as boas soluções que foram propostas para os problemas da área.

3 JEOPS

Este capítulo apresentará a arquitetura do sistema JEOPS, mostrando todas as partes que compõem o sistema, sem no entanto entrar em detalhes de sua implementação que serão apresentados no capítulo seguinte. Inicialmente, será introduzido um breve histórico do desenvolvimento de JEOPS, bem como os princípios que nortearam todo este desenvolvimento.

3.1 DA CONCEPÇÃO AO ESTÁGIO ATUAL

Esta seção ilustrará a trajetória histórica do desenvolvimento de JEOPS, salientando principalmente a evolução dos conceitos apresentados neste sistema.

3.1.1 A proposta inicial

A idéia do sistema surgiu durante a escolha de projetos para uma disciplina de graduação, Inteligência Artificial Simbólica [63], no segundo semestre de 1997. Vários projetos foram apresentados, para que se criasse um agente inteligente com uma certa complexidade. A equipe não tinha nenhuma experiência na utilização de linguagens específicas para a criação de aplicações inteligentes, como Prolog, e além do mais, nenhuma das linguagens conhecidas pela equipe apresentava um mecanismo de inferência, fundamental no desenvolvimento de aplicações inteligentes mais elaboradas. Então, entre a escolha de aprender uma linguagem com capacidades de inferência, desenvolver uma aplicação em uma linguagem de programação através de construções *if-then-else* para simular o raciocínio, escolhemos uma terceira via: desenvolver um motor de inferência para uma linguagem “convencional”.

Java foi escolhida por diversos motivos: a linguagem era conhecida por toda a equipe, sua estrutura elegante permitia o desenvolvimento de aplicações segundo os conceitos de orientação a objetos e apresentava facilidades de definição de interfaces gráficas, acesso a rede, entre outras. Sobretudo, Java era uma linguagem nova (na época tinha apenas 2 anos, a segunda versão do pacote de desenvolvimento da Sun – o JDK 1.1 – tinha sido lançado há poucos meses), e não havia nenhum mecanismo de inferência integrado à linguagem, o que motivou a equipe para aceitar o projeto.

Durante a disciplina foi desenvolvido então aquilo que viria a ser o primeiro protótipo de JEOPS. O motor original apresentava sérios problemas de performance (o algoritmo de unificação era bastante ineficiente), e uma grande limitação do subconjunto da linguagem Java suportado pela ferramenta (não aceitava chamadas encadeadas a métodos, tinha problemas com constantes String, entre outras) a impedia de ser usada em aplicações mais complexas.

3.1.2 A continuidade

Terminada a disciplina, o autor deste trabalho se interessou pelo projeto, e continuou o seu desenvolvimento por conta própria, sob a supervisão informal do professor Geber Ramalho, que havia ensinado a disciplina. Iniciou-se daí, a partir do segundo semestre do ano de 1998, um projeto de iniciação científica com a intenção de continuar o desenvolvimento do sistema, na época chamado de JEPS (acrônimo para *Java Embedded Production System*). Nesta época, alguns alunos de uma disciplina de mestrado começaram a utilizar o protótipo no desenvolvimento de agentes inteligentes – a primeira vez que o sistema era usado em uma aplicação real, além de “*toy examples*”. A utilização do JEPS pelos alunos foi de extrema importância na avaliação de suas limitações e descoberta de falhas (de concepção, projeto ou implementação).

O desenvolvimento que se deu nesta etapa foi concentrado na correção de erros, principalmente de programação, e na otimização do algoritmo de unificação. Atingimos uma melhoria de performance em mais de duas ordens de magnitude com a utilização do novo algoritmo em alguns estudos de caso. Uma outra grande mudança ocorrida nesta etapa foi o nome do sistema, que passou a ser conhecido pelo seu nome definitivo: JEOPS (*Java Embedded Object Production System*), para evitar uma confusão de nomes com outra ferramenta da área, o JESS.

3.1.3 O trabalho de graduação

No início do ano de 1999, o trabalho em andamento foi apresentado como uma proposta para um Trabalho de Graduação em Inteligência Artificial. A idéia era aproveitar o que já fora feito, e usar o período da disciplina para dar ao sistema novas funcionalidades que o tornassem definitivamente uma alternativa viável para a utilização no desenvolvimento de aplicações inteligentes. O protótipo da época, apesar de funcional,

ainda não estava pronto para ser usado em aplicações “reais”, por não incluir características para um controle maior do mecanismo de inferência, tipicamente a definição de estratégias de resolução de conflito. Neste sentido, uma série de mecanismos foram definidos de forma que o desenvolvedor pudesse criar a sua própria estratégia de resolução de conflitos, através da criação de uma classe que implementasse a política desejada, sem precisar alterar nenhum elemento de JEOPS.

Outra alteração ocorrida foi na interface do sistema, onde o seu idioma “oficial” passou a ser o inglês (até a versão anterior, nomes de classes, variáveis, métodos, enfim, todos os elementos básicos de programação eram definidos como palavras ou expressões em português). Comentários de todas as classes, referências cruzadas, e métodos foram traduzidos para o inglês, de modo que a documentação pudesse ser consultada por um número maior de pessoas.

Uma parte do tempo dedicada ao trabalho foi dispensada na definição de um manual do usuário (Cf. Anexo E). Somente após a criação deste documento é que pudemos pensar em divulgar o sistema para a comunidade.

3.1.4 O trabalho de mestrado

Aproveitando o fato de que muito já havia sido desenvolvido em JEOPS, mas que o sistema ainda não estava pronto para ser usado em aplicações de grande porte, decidimos continuar o trabalho mesmo após a conclusão do curso de graduação. A tendência natural foi então utilizar um curso de mestrado como forma de aprofundamento de diversos conceitos que fundamentam este trabalho, melhorando cada vez mais o sistema.

Neste período, JEOPS passou por transformações radicais. O sistema foi totalmente reescrito, com mudanças em sua filosofia (era interpretado, passou a ser pré-compilado), no *parser* das regras (com o sistema suportando não apenas um subconjunto de Java, mas toda a linguagem), no conjunto de conflitos (facilitando a definição de estratégias personalizadas para as necessidades do usuário), na unificação entre as regras e os objetos (utilizando uma versão de Rete), além de outras mudanças menores. Passamos também um tempo estudando as questões mais conceituais da integração objeto/regra, que resultou na submissão e aceitação de um artigo em uma conferência internacional [28].

3.2 PRINCÍPIOS

Conforme dito anteriormente, a concepção de JEOPS foi fortemente influenciada pelo sistema NéOpus. Com isso, os princípios básicos que nortearam a criação de JEOPS são semelhantes aos desse sistema. A idéia era criar um sistema que primasse sobretudo pela facilidade de uso e pelo respeito às características do paradigma orientado a objetos, o que seria conseguido com a maior integração possível entre as regras e a linguagem hospedeira. Além disso, precisávamos de um sistema que pudesse ser utilizado em aplicações reais, razão da necessidade de uma boa performance. Os princípios abaixo descritos foram definidos de modo a atender estas necessidades.

3.2.1 Uniformidade da integração

O primeiro princípio que decidimos adotar desde o início do projeto de JEOPS foi o da uniformidade de integração. Desta forma, definimos que sintaxe das regras seria semelhante ao máximo à de Java. A primeira versão de JEOPS utilizava um *parser* criado automaticamente pela ferramenta JavaCC [64]. Ela permitia que, dado um arquivo com a estrutura sintática da base de regras, fosse gerado uma classe que fazia toda a análise sintática das regras. Para aquela versão, que não suportava toda a linguagem Java, a ferramenta era adequada.

Com a sua evolução, JEOPS passou a suportar cada vez mais elementos sintáticos da linguagem (chamadas aninhadas de métodos, acesso direto a atributos públicos dos objetos, etc.). Com isso, foi atingido um ponto no qual o arquivo com a estrutura sintática das regras estava com tal complexidade que o *parser* tornou-se inviável de se manter, principalmente devido à nossa intenção de suportar *toda* a linguagem Java. Um outro problema do código gerado pelo JavaCC era a sua pouca legibilidade.

Devido a estes problemas, decidimos que iríamos implementar “do zero” um *parser* para analisar as regras do JEOPS. Esta abordagem não se mostrou muito trabalhosa, pois utilizamos apenas algumas regras de rescrita para transformar um arquivo contendo uma base de regras em um arquivo fonte Java.

Quanto aos elementos da memória de trabalho, decidimos desde o início que não poderíamos impor nenhuma restrição em relação aos objetos da linguagem que poderiam ser utilizados como elementos da memória de trabalho. Java, entretanto, apresenta outros

valores que não são objetos – tipos primitivos, que coexistem com os objetos na linguagem – que também eram candidatos a potenciais elementos da memória de trabalho. Como para cada tipo primitivo da linguagem existe uma classe correspondente que pode encapsulá-lo em um objeto, decidimos que estes valores não poderiam ser inseridos diretamente na base de conhecimentos, sem perda de expressividade do sistema.

3.2.2 Implementação do motor de inferência

No momento da criação da primeira versão de JEOPS, as implementações das máquinas virtuais de Java apresentavam diversos problemas de performance. Como a JVM interpreta os *bytecodes* gerados pelo compilador de Java, as vantagens da compilação das regras eram perdidas durante a execução do sistema. Decidimos portanto que essa versão utilizaria a interpretação das regras pelo motor de inferência.

No início do trabalho do mestrado, entretanto, as implementações das JVM já haviam atingido um nível de evolução no qual a solução pré-compilada era mais adequada, razão que nos levou a rescrever o sistema inteiramente. Isto também facilitou a implementação do suporte total à linguagem Java, e de outras características descritas na seção 3.1.4.

3.2.3 Algoritmo de unificação

Uma outra decisão tomada que norteou a criação do sistema diz respeito ao algoritmo de casamento de padrões, que faria a unificação entre as declarações das regras e os objetos da base de fatos. Já era fato conhecido que o algoritmo Rete, já mencionado anteriormente, apresentava a melhor performance em relação aos demais. No entanto, como a maioria das aplicações não apresentava um grande número de regras e objetos, foi decidido que iríamos deixar para implementar Rete mais tarde. Com o início do trabalho de mestrado, demos seqüência ao planejamento, e a versão 2.1 de JEOPS já utiliza o algoritmo Rete de casamento de padrões. As seções 3.5.4 e 4.3 apresentam em mais detalhes a implementação deste algoritmo em JEOPS.

3.2.4 Distribuição do sistema

Desde o início do desenvolvimento, tínhamos a intenção de que JEOPS fosse utilizado pelo maior número de pessoas possível (apesar de que somente começamos a

divulgar amplamente o sistema após a implementação de Rete). Um outro aspecto que observamos foi o fato de que como a área de EOops ainda tem diversos problemas em aberto, acreditamos que abrindo o seu código fonte, JEOPS poderia receber um maior número de contribuições de modo a torná-lo cada vez melhor.

Baseado nestas idéias, decidimos que JEOPS seria distribuído gratuitamente pela rede, com o código fonte aberto, segundo a licença LGPL (*Lesser General Public License*) [65] proposta pela FSF (*Free Software Foundation*) [66]. De acordo com esta licença, o sistema pode ser livremente utilizado e modificado pela comunidade. Com isso, acreditamos que JEOPS possa evoluir mais rapidamente que se fosse um sistema fechado.

3.3 REGRAS JEOPS

Baseado nos princípios citados acima, definimos a estrutura das regras de produção de JEOPS. Na criação destas regras, primamos pela separação explícita de todos os campos de uma regra: declarações de variáveis, condições e ações. Apesar desta separação exigir maior digitação, acreditamos que a legibilidade final das regras compensa o esforço. Regras JEOPS são organizadas dentro de *base de regras*, tal como métodos em Java são organizados dentro de classes. Uma base de regra contém regras, mas também pode conter métodos e declarações de variáveis, como qualquer classe de Java. A sintaxe da bases de regras, em uma estrutura BNF simplificada é apresentada no Anexo B.

Devemos ressaltar neste ponto que as regras JEOPS utilizam objetos. Estes objetos devem, portanto, já estar definidos quando elas forem ser pré-compiladas. A Tabela 22 ilustra um trecho da definição das classes `Pessoa` e `Objetivo`, usadas na regra apresentada na Tabela 23.

<pre>public class Pessoa { private String nome; private Pessoa pai, mae; public String getNome() { ... } }</pre>	<pre>public class Objetivo { private boolean ativo; private Pessoa alvo; public Objetivo(Pessoa p) { ... } }</pre>
--	--

Tabela 22 – Classes `Pessoa` e `Objetivo`

```

ruleBase Familia {
  rule encontraAncestrais {
    declarations
      Pessoa p;
      Objetivo o;
    localdecl
      Pessoa pai = p.getPai();
      Pessoa mae = p.getMae();
    conditions
      p == o.getAlvo(); o.estaAtivo();
    actions
      o.desativa();
      System.out.println(pai.getNome() + " e " +
        mae.getNome() + " são ancestrais");
      assert(new Objetivo(pai));
      assert(new Objetivo(mae));
  }
}

```

Tabela 23 – Base que contém a regra encontraAncestrais (JEOPS)

Esta base de regras contém apenas uma regra, encontraAncestrais, que já foi apresentada nos demais sistemas (Cf. capítulo 2). O campo das declarações contém as variáveis que serão unificadas com os objetos da memória de trabalho. No caso, a regra precisa de um objeto da classe Pessoa (p) e um da classe Objetivo (o). O campo das declarações locais foi criado apenas como “açucaramento sintático”, para a abreviação de expressões que sejam muito utilizadas na regra. No exemplo, toda ocorrência livre dos identificadores pai e mae serão substituídas pelas expressões p.getPai() e p.getMae(), respectivamente.

As condições da regra podem ser qualquer expressão booleana de Java. Em geral, tratam-se de comparações ou chamadas a métodos que retornam valores booleanos. No exemplo da Tabela 23, a regra estará disparável para todo par de objetos o e p das classes Objetivo e Pessoa respectivamente, que tornem as duas expressões (o.getAlvo() == p e o.estaAtivo()) verdadeiras.

Devemos ressaltar a importância de que os métodos chamados nas condições de uma regra **não devem alterar o estado** de seu objeto (ou seja, devem realizar apenas consultas ao estado do mesmo); isso se deve ao fato de que diversas avaliações dos métodos podem ser feitos sem que a regra seja disparada uma única vez com o objeto (porque uma outra condição pode ser falsa, por exemplo).

Finalmente, as ações da regra podem ser qualquer expressões de Java, além da manipulação dos objetos da base de fatos. Como já foi dito, tudo o que pode aparecer como

um corpo de método Java pode aparecer neste campo. Na regra da Tabela 23, por exemplo, são feitas chamadas a métodos, criação de objetos, impressão de mensagens na saída padrão. Enfim, tudo o que pode estar em um método de Java pode estar neste campo da regra. Além disso, através dos comandos **assert** e **retract**, objetos podem ser inseridos e removidos da memória de trabalho da base, respectivamente. Modificações nos objetos, da mesma forma que em NéOpus, devem ser informadas ao sistema pelo usuário, através da chamada ao método **modified**. O problema da modificação transitiva de objetos é contornado, da mesma forma que NéOpus, fazendo com que o usuário declare todas as variáveis que, uma vez modificadas, possam fazer a regra alterar de estado (de disparável para não disparável).

3.4 UTILIZAÇÃO DO SISTEMA

Esta seção apresenta os passos necessários para a utilização de JEOPS. Utilizaremos um exemplo mais interessante que o da busca por ancestrais mostrado anteriormente: a criação de agentes para um jogo de estratégia.

3.4.1 Definição do problema

Civilization II [67] é um dos jogos de estratégia mais conhecidos do mundo. Nele, o jogador é o líder de uma civilização, e tem que conduzi-la durante a história até a conquista do espaço. Para isso, o jogador tem que construir cidades nas localidades do mapa, desenvolver estas cidades (construindo igrejas, mercados, bibliotecas, sistema de esgoto, etc.), pesquisar novas tecnologias, manter seu exército bem equipado (para se defender das civilizações rivais, bem como para atacá-las), etc. À medida que novas tecnologias vão sendo descobertas, o jogador dispõe de novas unidades militares, de acordo com a evolução de seu conhecimento. O jogo ainda dispõe de diversas outras características, como tratados de paz e alianças militares entre as civilizações, corrupção, diversas formas de governo, comércio, etc. A Figura 12 ilustra uma das telas do jogo, na qual pode se ver o mapa do mundo (que vai sendo revelado à medida que o jogador o explora, com seu exército).



Figura 12 – Civilization II

3.4.2 Definição dos objetos e das regras

Observando a descrição do problema, podemos identificar algumas classes de objetos candidatas. Quando falamos no jogo, falamos de *jogadores*, de *unidades militares*, de *avanços tecnológicos*, de *idades*, *localização* no mapa, *benfeitorias* de uma cidade, *rotas de comércio*, etc. Cada um destes termos deve se transformar em uma classe na implementação da solução do problema. Por simplicidade, iremos apresentar nesta seção apenas as regras necessárias para as táticas de combate, e portanto, usaremos a princípio apenas as classes apresentadas na Figura 13. Como há diversos tipos de unidades militares no jogo, decidimos criar uma hierarquia com uma raiz indicando uma unidade militar genérica, e as subclasses representando os tipos específicos de unidades (tais como os pioneiros, guerreiros, falanges, cavaleiros, mísseis nucleares, etc.).

Uma outra decisão deve ser tomada quando da definição dos objetos: quais serão os agentes do problema? A resposta desta pergunta é a mesma da pergunta “Quais objetos devem ter comportamento autônomo?”. Os jogadores com certeza o são. As unidades militares também seriam candidatas a agentes, mas na prática não o são, pois no caso de Civilization II elas são totalmente controladas pelo usuário. Esta decisão depende de cada problema, e de como o programador deseja resolvê-lo.

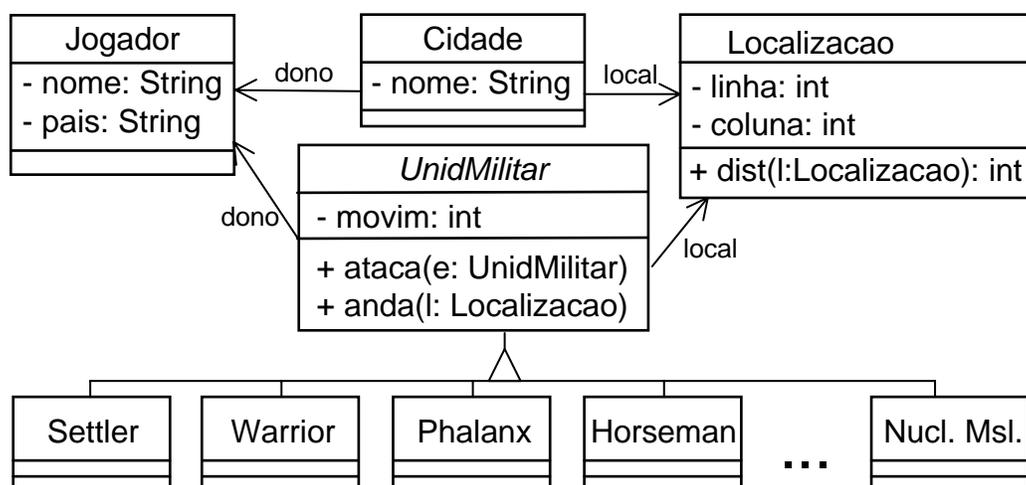


Figura 13 – Classes do domínio de Civilization II

Uma instância da classe `Jogador` representa um agente do mundo. Uma cidade é representada por um nome, e tem associada um dono (um jogador) e uma localização. Cada unidade militar tem uma capacidade de movimentação (por exemplo, unidades montadas se locomovem mais rapidamente que unidades terrestres), além de, como as cidades, terem associadas um jogador (o líder de sua civilização) e uma localização.

Definidos os objetos, o próximo passo é a definição das regras que serão usadas na resolução do problema. Por exemplo, uma das condições para a evolução é a manutenção das cidades, que devem ser defendidas dos ataques dos inimigos. É preciso criar a seguinte regra: “se houver um soldado inimigo próximo o suficiente da cidade, então atacá-lo com a melhor unidade presente na cidade”. Esta informação sobre a melhor unidade, no entanto, não está disponível nos objetos, motivo pelo qual devemos retornar à análise das classes, para que esta informação possa ser obtida de alguma forma. Neste caso, a cidade é o melhor objeto que pode responder qual a melhor unidade presente, e a classe seria redefinida conforme a Figura 14.

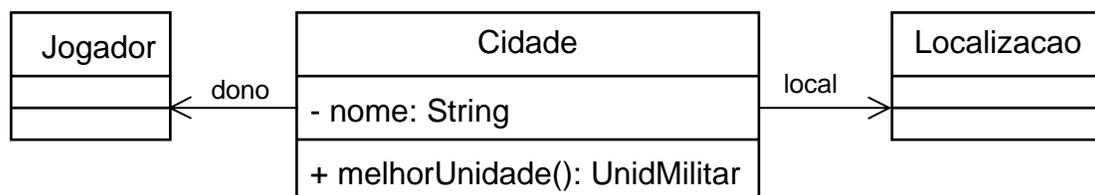


Figura 14 – Redefinição da classe `Cidade`

A primeira regra para se defender uma cidade pode então ser escrita, conforme apresentado na Tabela 24.

```

rule defendeCidade { // se há um inimigo por perto, é melhor atacá-lo
  declarations
    Jogador agente; // o dono da cidade
    Cidade cid; // a cidade sendo atacada
    UnidMilitar exAdv; // o exército adversário
  conditions
    cid.melhorUnidade() != null; // se não existir uma melhor unidade,
    cid.getDono() == agente; // então esta regra não se aplica
    exAdv.getDono() != agente; // verificando se é inimigo
    cid.getLocal().dist(exAdv.getLocal()) <= exAdv.getMovim();
  actions
    cid.melhorUnidade().ataca(exAdv());
}

```

Tabela 24 – Regra defendeCidade

Há, no entanto, uma penalidade para quem quebra tratados de paz no jogo. A regra defendeCidade, apresentada acima, não leva em conta se o jogador está em paz com o dono do inimigo – que pode estar com o exército nas proximidades apenas de passagem. Um novo retorno à definição das classes incorpora a informação sobre os tratados entre as civilizações nos objetos, para que a regra possa levar isso em conta. A Figura 15 ilustra a nova definição da classe Jogador, que armazenará as informações sobre os tratados diplomáticos do jogo.

Jogador
- nome: String - pais: String
+ emPazCom(j: Jogador): boolean

Figura 15 – Redefinição da classe Jogador

Com este novo método, é possível se redefinir a regra defendeCidade, para que a civilização não entre em nenhum conflito diplomático, como apresentado na Tabela 25.

```

rule defendeCidade { // se há um inimigo por perto, é melhor atacá-lo
  declarations
    Jogador agente; // o dono da cidade
    Cidade cid; // a cidade sendo atacada
    UnidMilitar exAdv; // o exército adversário
  conditions
    cid.melhorUnidade() != null; // se não existir uma melhor unidade,
    cid.getDono() == agente; // então esta regra não se aplica
    exAdv.getDono() != agente; // verificando se é inimigo
    cid.getLocal().dist(exAdv.getLocal()) <= exAdv.getMovim();
    !agente.emPazCom(exAdv.getDono()); // só se não estiver em paz
  actions
    cid.melhorUnidade().ataca(exAdv());
}

```

Tabela 25 – Regra defendeCidade (segunda versão)

Uma vez que o agente já sabe como defender uma cidade, o processo de criação das regras continua para os demais processos do jogo (ataque, negociações, pesquisa, construções, etc.). O que deve ficar claro, conforme foi apresentado na seção 2.3.2, é que o processo de definição dos objetos e das regras é iterativo: a cada nova necessidade das regras, os objetos podem ser modificados.

Quando todas as regras estiverem definidas, o usuário deve então colocá-las dentro de uma *base de regras*. Uma base de regras em JEOPS nada mais é que uma coleção de regras, que atuam sobre os mesmos objetos. Desta forma, a base de regras poderia ser definida conforme apresentado na Tabela 26.

```

ruleBase BaseCiv2 {
  rule defendeCidade { ... }
  rule protegeCidade { ... }
  rule atacaCidade { ... }
  rule exploraMundo { ... }
  ...
}

```

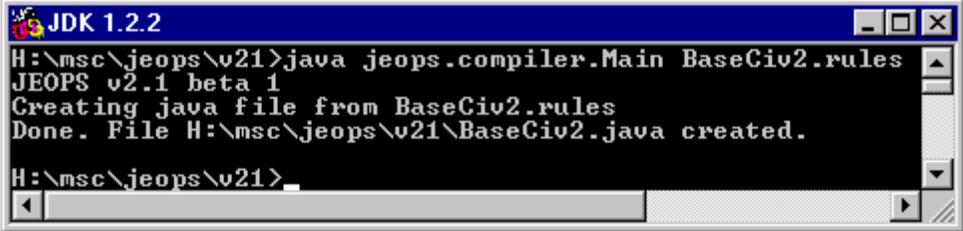
Tabela 26 – Base de regras do domínio de Civilization II

Da mesma forma que Java obriga que uma classe pública **C** seja definida dentro de um arquivo chamado **C.java**, em JEOPS uma base de regras deve ser gravada num arquivo texto, cujo nome é o mesmo da base, e com a extensão “.rules”. Assim, a base de regras da Tabela 26 seria gravada como o arquivo `BaseCiv2.rules`.

3.4.3 Pré-compilação da base de regras

Uma vez definida a base de regras que será utilizada na resolução de um problema (`BaseCiv2.rules`), e as classes que ela utiliza (`Jogador.java`, `Cidade.java`, etc.), o

usuário deve realizar a pré-compilação do arquivo da base de regras. Esta pré-compilação é feita trivialmente através da execução da classe `jeops.compiler.Main`, conforme apresentado na Figura 16. É passado na linha de comando o arquivo que contém as regras, e é gerado um arquivo Java que contém uma classe com o mesmo nome da base de regras (`BaseCiv2.java`) que representa a base de conhecimentos, e uma classe que representa a base interna de regras (Cf. seção 3.5.3), esta última não sendo vista pelo usuário. A decisão de se criar duas classes em vez de uma foi feita para facilitar a criação da base de conhecimentos pelo usuário, conforme será apresentado na seção 4.4.1.



```
JDK 1.2.2
H:\msc\jeops\v21>java jeops.compiler.Main BaseCiv2.rules
JEOPS v2.1 beta 1
Creating java file from BaseCiv2.rules
Done. File H:\msc\jeops\v21\BaseCiv2.java created.
H:\msc\jeops\v21>
```

Figura 16 – Pré-compilação do arquivo de regras

A Figura 17 ilustra os passos que ocorrem durante a execução do comando acima. Em primeiro lugar, o pré-compilador de JEOPS verifica se há alguma classe usada pelas regras que ainda não foi compilada, compilando-a se for o caso. No exemplo, a classe `UnidMilitar` ainda não havia sido compilada (①). Em seguida, o pré-compilador das regras irá converter o arquivo que foi passado como parâmetro (`BaseCiv2.rules`), e gerar um arquivo Java (`BaseCiv2.java`) que define as duas classes que implementam as regras (②). Finalmente, o pré-compilador de JEOPS irá compilar o arquivo Java gerado, gerando os arquivos `BaseCiv2.class`, referente à base de conhecimentos, e `Jeops_RuleBase_BaseCiv2.class`, referente à base interna de regras (③).

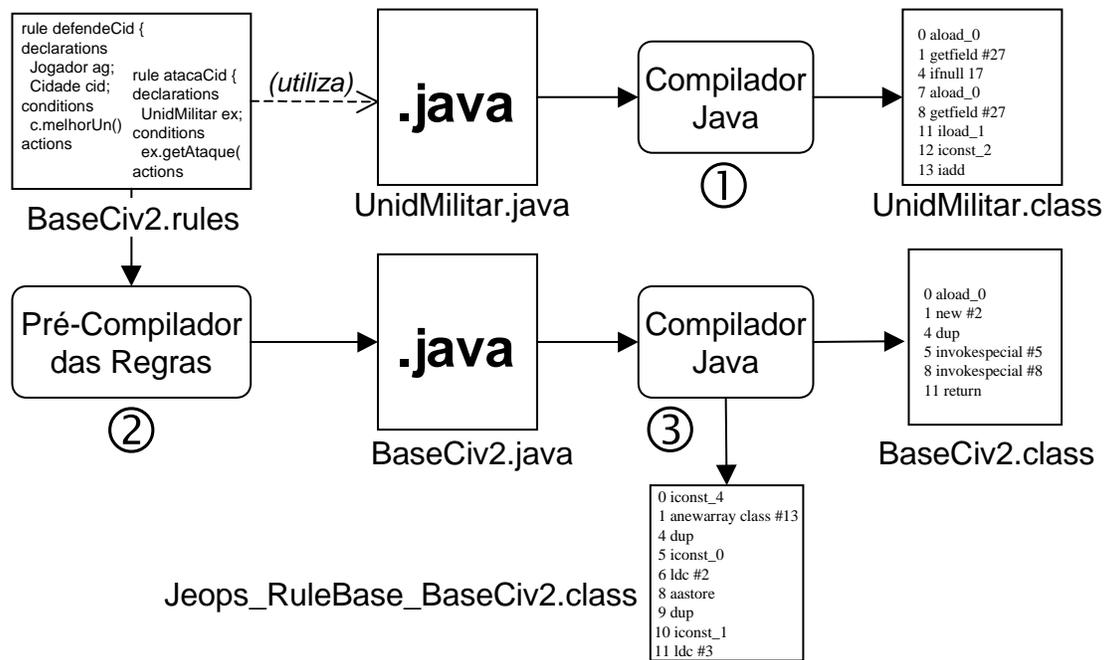


Figura 17 – Etapas da pré-compilação do arquivo de regras

Devemos deixar bem claro neste ponto a diferença entre a base de regras definida pelo usuário no arquivo de regras e a base interna de regras, gerada pelo pré-compilador. A base de regras definida pelo usuário é a maneira que ele vê as regras. Ela é armazenada em um arquivo texto, e pode ser alterada por qualquer editor. A base interna de regras, por sua vez, não precisa ser conhecida pelo usuário. Ela é gerada pelo pré-compilador de JEOPS, e só pode ser alterada a partir de uma nova pré-compilação do arquivo de regras. Deste ponto em diante, utilizaremos os termos *arquivo de regras* para identificar o local onde o usuário define suas regras (textualmente) e *base interna de regras* para representar a classe gerada pelo pré-compilador.

A seção 4.6 apresentará os detalhes de como é feita a pré-compilação das regras. Para exemplificar esta pré-compilação, o Anexo G apresenta a conversão completa de uma base de regras no arquivo Java.

3.4.4 Utilização da base de conhecimentos

Compilada a base de regras, a utilização da base de conhecimentos gerada pelo processo também é trivial. A Tabela 27 apresenta uma classe que utiliza a base de

conhecimentos gerada¹. Na linha 1, é criada uma instância da base de conhecimentos, gerada pelo processo de pré-compilação do arquivo de regras. Como há regras mais importantes que outras no problema, decidimos que usaríamos uma estratégia de resolução de conflitos baseada em prioridades, o que é conseguido passando-se uma instância desta implementação de conjunto de conflitos (Cf. seção 3.5.5) para a base de conhecimentos. Na linha 2 é criado o agente jogador, que será usado na criação dos demais objetos. Na linha 3, é criada a primeira unidade militar que o jogador dispõe (um conjunto de pioneiros ou *settlers*, com os quais poderá ser criada a primeira cidade). Os demais objetos (localizações, exércitos inimigos, etc.) são criados na linha 4. As linhas 5, 6 e 7 tratam de inserir os objetos criados na base de conhecimentos. Finalmente, na linha 8 é enviado o método `run()` à base, que faz com que ela execute as regras disparáveis, até que o conjunto de conflitos esteja vazio – o que só deve ocorrer quando o jogo acabar.

```
public class AgenteCiv2 {
    public static void main(String[] args) {
1. BaseCiv2 base = new BaseCiv2(new PriorityConflictSet());
2. Jogador agente = new Jogador("Winston Churchill", "Inglaterra");
3. UnidMilitar inicial = new Settler(agente, Localizacao.random());
4. ... // criando os demais objetos
5. base.assert(agente); // Inserindo o agente na base
6. base.assert(inicial); // Inserindo os settlers na base
7. ... // inserindo os demais objetos
8. base.run(); // Ativando a base de conhecimentos
    }
}
```

Tabela 27 – Utilização da base de conhecimentos do problema de Civilization II

3.5 ARQUITETURA INTERNA

A arquitetura de JEOPS segue a linha dos sistemas de produção tradicionais. Existe a separação clara entre a memória de trabalho (ou base de fatos ou, na nossa nomenclatura, base de objetos) e a base (interna) de regras (ao contrário de NéOpus, por exemplo, no qual a base de regras não existe). Quando são inseridos na memória de trabalho, os objetos que fazem uma regra se tornar disparável são adicionados junto com a referida regra ao conjunto de conflito. Quando a base de conhecimentos é ativada, ela solicita continuamente

¹ Um método declarado como “`public static void main(String[] args)`” é o ponto de entrada de uma classe Java, assim como a função “`main(int argc, char *argv[])`” é o ponto de entrada de programas escritos em C.

um elemento (par <regra, objetos>) do conjunto de conflitos, e executa a regra com os objetos correspondentes. Este processo continua até que o conjunto de conflitos esteja vazio. O usuário, no entanto, não precisa ter ciência desta arquitetura, podendo trabalhar apenas com a base de conhecimentos (a seção 3.5.1 apresenta uma descrição dos métodos principais desta classe). A Figura 18 ilustra, num alto nível, a arquitetura de JEOPS. Os blocos que a compõem serão apresentados em seguida. Devemos ressaltar que nesta seção serão apresentados apenas os métodos das classes com as quais o usuário interage (ou seja, apresentaremos *o que* pode fazer uma base de conhecimentos JEOPS). Os demais métodos, e os detalhes de implementação das classes serão mostrados no capítulo 4 (em outras palavras, apresentaremos *como* JEOPS é implementado).

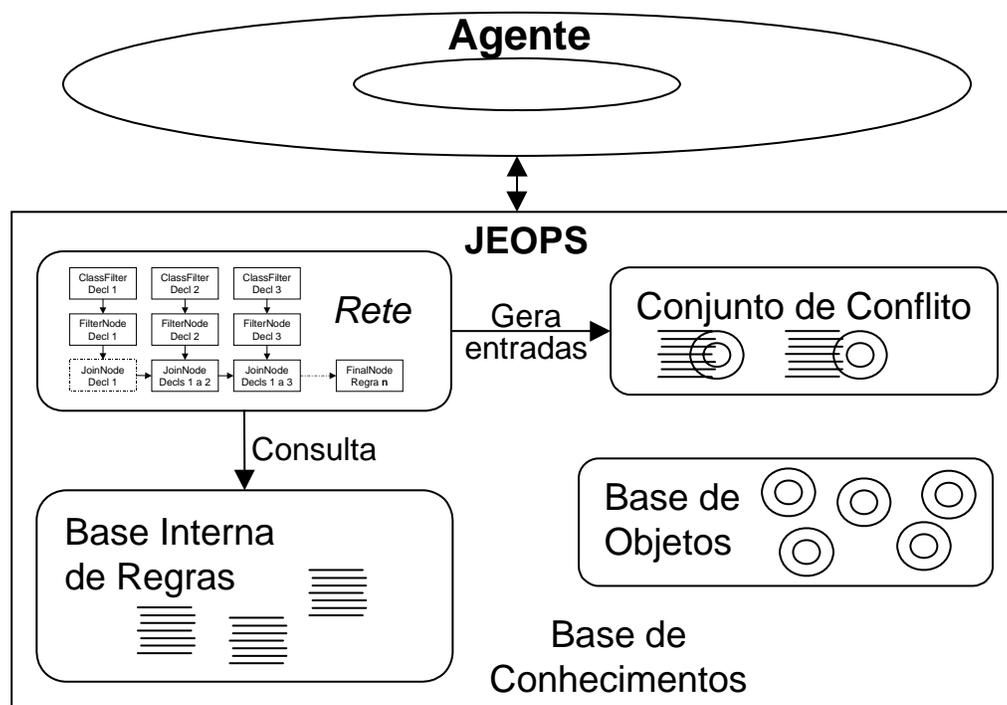


Figura 18 – Arquitetura de JEOPS

3.5.1 A base de conhecimentos

A base de conhecimentos é a grande fachada de JEOPS que é vista pelo programador. Dos seus componentes vistos anteriormente, dificilmente alguém irá precisar compreender ou alterar algo além do conjunto de conflitos (que pode ser definido de maneira completamente independente da base de conhecimentos). É a base de

conhecimentos quem encapsula os demais componentes, de modo a fornecer ao usuário uma porta de entrada única (e simples) para o motor de inferência.

Bases de conhecimento JEOPS são subclasses da classe abstrata `jeops.AbstractKnowledgeBase`. O usuário precisa acessar apenas os métodos principais definidos nesta classe para poder utilizá-la. São métodos de inserção e remoção de objetos na memória de trabalho, consulta dos objetos da mesma, *reset* e ativação da base de conhecimentos. Estes métodos são os seguintes:

- ***assert(Object)***
Insere um objeto na memória de trabalho da base de conhecimentos. Se este objeto puder ser unificado com uma declaração de uma das regras da base, o par <regra, objeto> será inserido imediatamente no conjunto de conflitos.
- ***retract(Object)***
Remove um objeto da memória de trabalho da base de conhecimentos. Se existir algum elemento do conjunto de conflitos que utilize este objeto, ele será imediatamente removido.
- ***objects(String) : Vector***
Esta é a maneira pela qual o agente pode solicitar informações à base de conhecimentos. Este método retorna todos os objetos da classe passada como parâmetro. Devemos ressaltar que este método retorna tanto os objetos que são instâncias diretas da classe dada, como também as instâncias de suas subclasses (como deveria se esperar em um sistema orientado a objetos).
- ***flush()***
Remove todos os fatos (objetos) da base de conhecimentos, limpa a história do conjunto de conflitos (se houver). É o *reset* da base.
- ***run()***
Ativa a base de conhecimentos, colocando o motor de inferência para funcionar. Quando da chamada deste método, a base de conhecimentos irá disparar as regras do conjunto de conflitos até que este esteja vazio.

Maiores detalhes sobre a implementação destes métodos serão apresentados na seção 4.4.

3.5.2 A base de objetos

A base de objetos é o local onde são armazenadas as referências para os objetos que fazem parte da memória de trabalho da base de conhecimentos. Sua principal função até a versão 2.0 de JEOPS era a de prover os objetos necessários para a etapa de unificação do algoritmo de casamento de padrões. Com a implementação do algoritmo Rete, a partir da versão 2.1, esta utilidade da base de objetos tornou-se desnecessária, mas decidimos que esse componente ainda oferecia outros serviços os quais compensavam sua manutenção no sistema.

Em primeiro lugar, a cada novo objeto que é inserido na base de conhecimentos, antes dele ser enviado à rede Rete, é feita uma consulta à base de objetos para saber se o objeto sendo inserido já está presente na memória de trabalho. Se for o caso, a inserção é desconsiderada. Esta precaução é importante para que não haja a possibilidade da existência de objetos sendo armazenados redundantemente nos nós da rede (Cf. seção 3.5.4). É importante a ressalva de que esta informação poderia ser armazenada diretamente na rede, mas ela estaria dispersa, e acreditamos que esta verificação seja feita com uma tal frequência que justifique a existência de um componente especializado em realizá-la de forma mais eficiente.

O segundo serviço provido pela base de objetos é utilizado quando o usuário invoca o método `objects(String)` da base de conhecimentos. Como a base de objetos armazena a referência de todos os objetos da memória de trabalho, ela tornou-se o local ideal para a implementação deste método. É a base de objetos também quem trata a relação de herança entre as classes, de modo que se a base de conhecimentos solicita todos os objetos de uma determinada classe, a base de objetos retornará não somente suas instâncias diretas, mas também as indiretas (ou seja, instâncias diretas de suas subclasses).

Estas são as duas funções da base de objetos na implementação atual de JEOPS. Os detalhes de sua implementação serão apresentados na seção 4.1.

3.5.3 A base interna de regras

A base interna de regras é o componente responsável por responder à base de conhecimentos ou à rede Rete se as condições de uma regra são verdadeiras, quantas e quais regras foram definidas no arquivo pelo usuário, quais as declarações de cada regra. É

também a base interna de regras que é responsável por disparar uma regra, quando solicitada pela base de conhecimentos.

Como foi apresentado anteriormente, a base interna de regras é gerada a partir da pré-compilação do arquivo de regras. A classe gerada, chamada de “Jeops_RuleBase_<nome da base de regras>” é uma subclasse de uma classe mais geral de JEOPS (`jeops.AbstractRuleBase`), que define uma série de assinaturas de métodos que são implementados pela classe gerada. Estes métodos, que são redefinidos na base interna de regras, podem responder à base de conhecimentos por exemplo quantas regras existem em uma base de regras, se uma condição de uma regra é verdadeira, entre outros. Estes métodos dão à base de conhecimentos toda a informação de que ela precisa para realizar a unificação dos objetos da memória de trabalho com as declarações das regras. Com este nível de abstração, conseguimos diminuir bastante a complexidade do código gerado na compilação da base de regras, facilitando a compreensão do código, o que permite a depuração do mesmo em ambientes padrões de desenvolvimento de Java².

Os métodos definidos na classe abstrata `jeops.AbstractRuleBase`, bem como maiores detalhes sobre a implementação da base interna de regras serão apresentados na seção 4.5.

3.5.4 A rede Rete

Em sua implementação clássica, Rete é um algoritmo que apresenta duas vantagens em relação à maneira “tradicional” de casamento de padrões, na qual cada novo objeto inserido na base é testado com todas as demais combinações de objetos já presentes para se encontrar unificações com as condições das regras. A primeira deve-se ao fato de que, criando um grafo de propagação de novos fatos, as avaliações de uma condição que ocorram em mais de uma regra podem ser feita apenas uma vez, conforme pode ser observado na Figura 19. Uma outra vantagem do algoritmo é que ele permite que as avaliações que já foram realizadas sejam armazenadas, de modo que não precisem ser refeitas sem necessidade. Por exemplo, se um novo fato “mãe(Maria, José)” for inserido na memória de trabalho do mesmo exemplo, o sistema só irá testá-lo com os fatos que já

² Quem já tentou por exemplo examinar o *parser* gerado pelo JavaCC [64] ou pelo par LEX/YACC [68] sabe a dificuldade de se compreender os códigos gerados automaticamente.

passaram pelo primeiro nó da rede (“pai(X,Y)”) e cuja variável Y está unificada com o termo “Maria”, evitando desta forma um grande número de testes que não levariam à ativação da regra.

Regra 1:
 Se pai(X,Y) e pai(Y,Z)
 Então avôPaterno(X,Z)

Regra 2:
 Se pai(X,Y) e mãe(Y,Z)
 Então avôMaterno(X,Z)

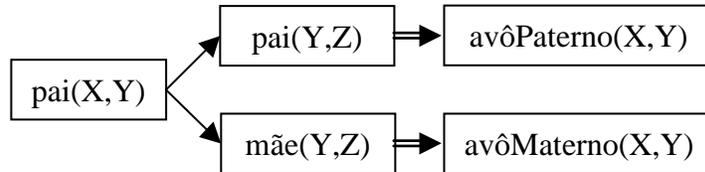


Figura 19 – Implementação tradicional de Rete

A implementação de Rete presente no JEOPS é uma variação da versão original, de modo a acomodar as diferenças que apareceram quando da utilização de objetos para representar fatos. Utilizamos nesta implementação as mesmas idéias empregadas na implementação de Rete no sistema NéOpus.

Basicamente, o sistema utiliza apenas uma das facetas de Rete, a da memorização de avaliações feitas anteriormente, deixando de lado a utilização da rede para eliminar avaliações redundantes. Decidimos não utilizar esta vantagem do algoritmo porque, devido ao fato de permitirmos a utilização de qualquer expressão Java nas condições das regras, uma mesma condição pode ser expressa de diversas maneiras. Por exemplo, todos os pares apresentados na Tabela 28 mostram condições semanticamente idênticas, apesar de serem diferentes na sua forma:

<code>p1.getIdade() > 50;</code>	<code>!p1.getIdade() <= 50</code>
<code>p1.getIdade() == p2.getIdade();</code>	<code>p2.getIdade() == p1.getIdade();</code>
<code>p1.getIdade() == p2.getIdade();</code>	<code>p1.getIdade() == p2.getIdade();</code>
<code>p1.getIdade() == 40;</code>	<code>p2.getIdade() == 40;</code>
<code>p1.getSalario().getValor() > 10000;</code>	<code>p1.ganhaMaisQue(10000);</code>
<code>p1.getSalario().getValor() > p2.getSalario().getValor();</code>	<code>p1.getSalario().maior(p2.getSalario());</code>

Tabela 28 – Pares de condições semanticamente idênticas

Com esta possibilidade de se implementar a mesma condição de formas distintas, o ganho obtido com a implementação do compartilhamento de nós em Rete é potencialmente reduzido. Com a simplificação da implementação de Rete, o esforço necessário para a sua codificação tornou-se menor, pois cada regra pôde ser tratada individualmente. Desta forma, o problema da criação de uma rede que representasse um conjunto de regras reduziu-se à criação de uma “mini-rede” para cada regra.

A seção 4.3 apresentará os detalhes da implementação de Rete utilizada no JEOPS.

3.5.5 O conjunto de conflitos

O conjunto de conflitos é o local onde são armazenados todos os pares (regra, conjunto de objetos que a tornam ativa) da base de conhecimentos. Em JEOPS, procuramos dissociar ao máximo a implementação dos conjuntos de conflitos do motor de inferência propriamente dito, de modo que o usuário pudesse definir suas próprias políticas de resolução de conflitos sem precisar compreender como funciona internamente o motor.

Partindo deste princípio, criamos uma interface em Java³ que define os métodos necessários para o funcionamento de um conjunto de conflitos, sem especificar como deveria ser feita sua implementação (`jeops.conflict.ConflictSet`). Da mesma forma, decidimos que os elementos (regras disparáveis + objetos) do conjunto não seriam utilizar a mesma estrutura usada dentro do motor de inferência, para eliminar esta dependência. Definimos então a classe `jeops.conflict.ConflictSetElement`, que contem toda a informação necessária para que o usuário implemente a política desejada de resolução de conflitos. As classes `ConflictSet` e `ConflictSetElement` são apresentadas na Figura 20.

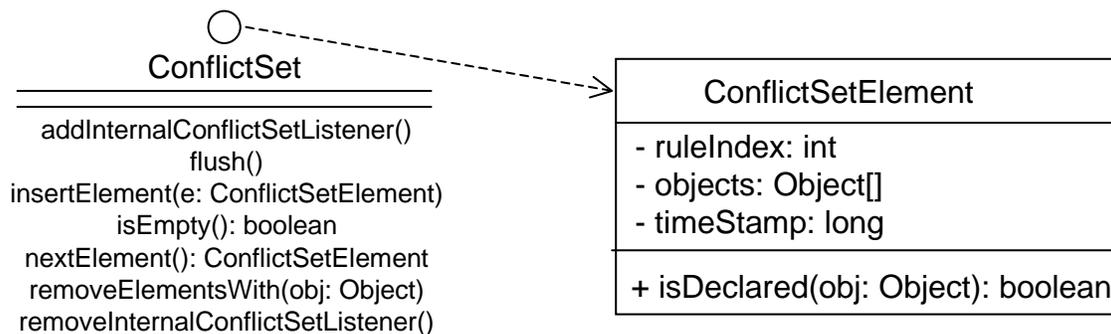


Figura 20 – Estrutura do conjunto de conflitos

O elemento do conjunto de conflitos contem o número da regra que pode ser ativada, os objetos que a tornaram ativa, e um *timestamp*, que armazena o instante em que o mesmo foi criado – útil na definição de políticas que privilegiam objetos criados mais ou menos recentemente. Além dos métodos normais de acesso, a classe apresenta um método auxiliar que verifica se um dado objeto foi declarado na regra que ele representa.

³ Para quem não está familiarizado com o conceito, uma *interface* em Java é uma coleção de definições de métodos (sem implementação). Uma classe que implementa esta interface “concorda” em implementar todos os métodos definidos na interface. Uma descrição mais detalhada pode ser vista em [69].

Devemos ressaltar que esta abordagem difere significativamente da utilizada em NéOpus. Este sistema utiliza instâncias da classe da própria base interna de regras como elementos do conjunto de conflitos, facilitando a implementação do motor de inferência. Como em NéOpus as estratégias de resolução de conflitos são definidas declarativamente, esta mistura de conceitos não é observada pelo usuário. Como este não é o caso de JEOPS, decidimos manter o modelo mais enxuto, apesar do esforço maior de implementação.

A interface `ConflictSet` define alguns métodos essenciais e outros acessórios. Os métodos principais são os seguintes:

- ***flush()***
Limpa todas as regras presentes no conjunto, bem como toda a história que o mesmo possa ter armazenado.
- ***insertElement(ConflictSetElement)***
Insere um novo elemento no conjunto.
- ***isEmpty(): boolean***
Verifica se o conjunto de conflitos está vazio.
- ***nextElement(): ConflictSetElement***
Retorna o elemento que representa a próxima regra a ser disparada, de acordo com a política definida pelo conjunto de conflitos.
- ***removeElementsWith(Object)***
Remove do conjunto de conflitos todos os elementos que contenham o objeto dado.

Os demais métodos definidos pela interface (`addInternalConflictSetListener` e `removeInternalConflictSetListener`) indicam que um conjunto de conflitos deve ser uma *fonte de eventos*, de acordo com a nomenclatura definida pela Sun [70]. Isso é necessário para que interfaces de depuração possam ser acopladas ao JEOPS. O que os conjuntos de conflitos precisam fazer é manter uma lista dos objetos cadastrados para receber seus eventos, e a cada elemento que é adicionado ou removido do conjunto, estes ouvintes devem ser notificados.

Maiores detalhes sobre a implementação destes métodos serão apresentados na seção 4.2, bem como os conjuntos de conflitos já implementados.

3.6 INTERFACE DE DEPURAÇÃO

A evolução dos ambientes de programação, principalmente os visuais, fez com que os programadores se tornassem mais exigentes em relação aos produtos utilizados. Uma ferramenta que já é parte indispensável de todo ambiente é um depurador, no qual o desenvolvedor pode acompanhar passo a passo o que está ocorrendo dentro do sistema, interromper a execução em um determinado ponto, observar o valor de variáveis, entre outras características.

A partir de sua versão 2.0, a base de conhecimentos de JEOPS foi preparada para que uma interface de depuração possa ser “plugada” a ela. Isto foi feito seguindo o modelo de eventos definido a partir da versão 1.1 de Java [70]. A partir deste modelo, uma interface de depuração pode se registrar na base de conhecimentos, tornando-se um *ouvinte* dos eventos gerados pela base. Esta, por sua vez, informa aos ouvintes registrados os eventos que nela ocorrem. Há basicamente dois tipos de eventos gerados pela base de conhecimentos: os que dizem respeito ao disparo das regras, e os que dizem respeito aos elementos do conjunto de conflitos. Uma interface de depuração tanto pode registrar-se como ouvinte de ambos os tipos de eventos como de apenas um deles.

Estes são os eventos que podem ser notificados pela base de conhecimentos. Os dois primeiros ocorrem quando do disparo de alguma regra, e os dois últimos quando da alteração do conteúdo do conjunto de conflitos:

- ***ruleFiring(RuleEvent)***

Enviado aos ouvintes registrados sempre que uma regra está prestes a ser disparada. Com isso, a interface de depuração pode suspender a execução do motor de inferência, realizando a execução passo a passo. É passado para o ouvinte um evento que indica qual regra foi disparada, e que objetos estão mapeados com as declarações da regra, de modo que a interface de depuração possa apresentar seus valores ao usuário.

- ***ruleFired(RuleEvent)***

Enviado aos ouvintes registrados imediatamente após o disparo da regra. Com isso, a interface de depuração pode apresentar ao usuário as alterações que ocorreram nos objetos, bem como suspender mais uma vez a execução para o

usuário continuar com a execução passo a passo. É passado para o ouvinte um evento do mesmo tipo que o do método anterior.

- ***elementAdded(ConflictSetEvent)***

Enviado aos ouvintes registrados quando um novo elemento é inserido no conjunto de conflitos. Com isto, é possível que a interface de depuração apresente não somente o disparo das regras da base de conhecimentos, mas também todas as alterações que ocorrem no conjunto de conflitos. É passado para o ouvinte um evento que contém o elemento que foi adicionado ao conjunto de conflitos (que representa o par <regra, objetos>).

- ***elementRemoved(ConflictSetEvent)***

Enviado aos ouvintes registrados quando um elemento é removido do conjunto de conflitos. Este método, em conjunto com o anterior possibilita uma visão total do comportamento do conjunto de conflitos no decorrer do ciclo de vida da base de conhecimentos. É passado para o ouvinte um evento do mesmo tipo que o do método anterior.

Apesar de JEOPS ainda não ter uma interface de depuração adequada, estes métodos podem ser utilizados em programas desenvolvidos com JEOPS. O trecho de código apresentado na Tabela 29, executa uma base de conhecimentos que resolve o problema de Fibonacci (Cf. Anexo C). Durante a execução, são apresentados na saída padrão todas as regras disparadas, bem como o mapeamento entre suas declarações e os objetos reais.

```

public static void main(String[] args) {
    FibonacciBase base = new FibonacciBase();
    Fibonacci f = new Fibonacci(10);
    base.assert(f);
    base.addRuleFireListener(new RuleFireListener() {
        public void ruleFiring(RuleEvent evt) {
            int i = e.getRuleIndex();
            String name = e.getKnowledgeBase().getRuleBase().getRuleNames()[i];
            System.out.println("Firing rule " + name);
            String[] decls = e.getKnowledgeBase().getRuleBase().
                getDeclaredIdentifiers(i);
            Object[] objs = e.getKnowledgeBase().getRuleBase().getObjects(i);
            System.out.println("Before firing:");
            for (int j = 0; j < decls.length; j++) {
                System.out.println("  " + decls[j] + " = " + objs[j]);
            }
        }
        public void ruleFired(RuleEvent e) { }
    });
    base.run();
    System.out.println("Fib(" + f.getN() + " = " + f.getValue());
}

```

Tabela 29 – Utilização do mecanismo de depuração de JEOPS

A implementação de uma interface gráfica para acompanhamento da execução do motor de inferência de JEOPS é um dos trabalhos futuros do sistema, apresentados no capítulo 6.

3.7 SÍNTESE

Esperamos com este capítulo ter dado uma visão geral de JEOPS, apresentando sua história e seus componentes. Enunciamos os princípios que direcionaram a concepção e o desenvolvimento de JEOPS, mostrando em seguida a sintaxe utilizada na definição das suas regras. Apresentamos também um exemplo passo a passo completo de utilização do sistema, para que o mesmo possa ser compreendido mesmo por uma audiência que não está muito familiarizada com os conceitos da linguagem Java. Finalmente, apresentamos os módulos que compõem a arquitetura de JEOPS, bem como o mecanismo que pode ser utilizado para a construção de ferramentas de depuração. Desta forma, esperamos ter mostrado que JEOPS é de fato o sistema que apresenta uma melhor integração entre regras e objetos para a linguagem Java.

O capítulo seguinte é um complemento deste, onde os elementos que formam o sistema são apresentados em maiores detalhes. O Anexo C também pode ser usado como leitura complementar, bem como a página de documentação da API do sistema [71].

4 IMPLEMENTAÇÃO

Este capítulo apresentará os detalhes internos da implementação de JEOPS. Ele pode ser visto como uma continuação do capítulo anterior, onde os componentes da arquitetura serão apresentados em maiores detalhes. Apresentaremos ainda o processo de pré-compilação, usado para transformar o arquivo das regras nas classes da base de conhecimentos e da base interna de regras.

4.1 A BASE DE OBJETOS

A seção 3.5.2 apresentou os serviços oferecidos pela base de objetos. Esta seção apresentará os detalhes de suas implementações.

A figura Figura 21 ilustra a classe `ObjectBase`, que implementa a base de objetos em JEOPS. A base de objetos é basicamente composta por dois atributos. O primeiro (`objects`) é uma tabela *hash* que mapeia os nomes das classes dos elementos que já foram inseridos na base de objetos em suas instâncias diretas presentes na memória de trabalhos. Com este mapeamento, a verificação da pertinência de um objeto torna-se mais eficiente, uma vez que somente são verificados os objetos da mesma classe. O segundo atributo (`subClasses`) armazena todas as relações de hierarquia (herança e implementação de interfaces) existentes entre as classes dos objetos que já foram inseridos na base, para que a consulta aos objetos das subclasses de uma determinada classe seja feita eficientemente.

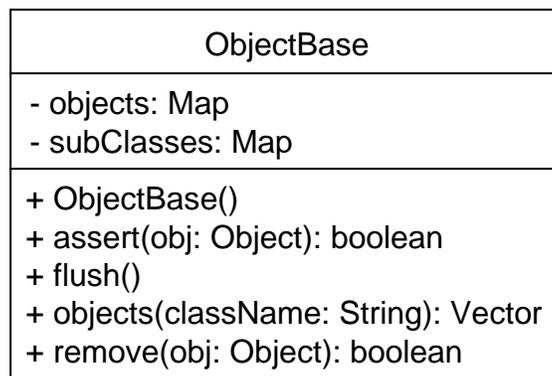


Figura 21 – Classe `ObjectBase`

O método `assert(Object)` é chamado pela base de conhecimentos para inserir um novo objeto na base de objetos. Ele retorna um valor booleano indicando se a inserção teve

sucesso (ou seja, se o objeto ainda não pertencia à base). Os métodos `flush()` e `remove(Object)` são utilizados para se remover elementos da base de objetos: o primeiro remove *todos* os elementos, enquanto que o último remove um objeto em particular. Assim como o método `assert`, `remove` retorna um valor booleano indicando se a remoção ocorreu com sucesso ou não. Finalmente, o método `objects(String)` é usado para se consultar os objetos de uma determinada classe presentes na base de objetos.

A inserção de um novo objeto na base de objetos pode ocorrer de duas formas distintas. Da primeira vez que um objeto de uma determinada classe é inserido na base, toda a informação de herança daquela classe será armazenada na base. É criado um conjunto (implementado pela classe `Vector` de Java) onde serão armazenadas as instâncias da classe, e este conjunto, com o novo objeto, é inserido no mapeamento `objects`. A Figura 22 apresenta o diagrama de seqüência de mensagens utilizadas na implementação deste método.

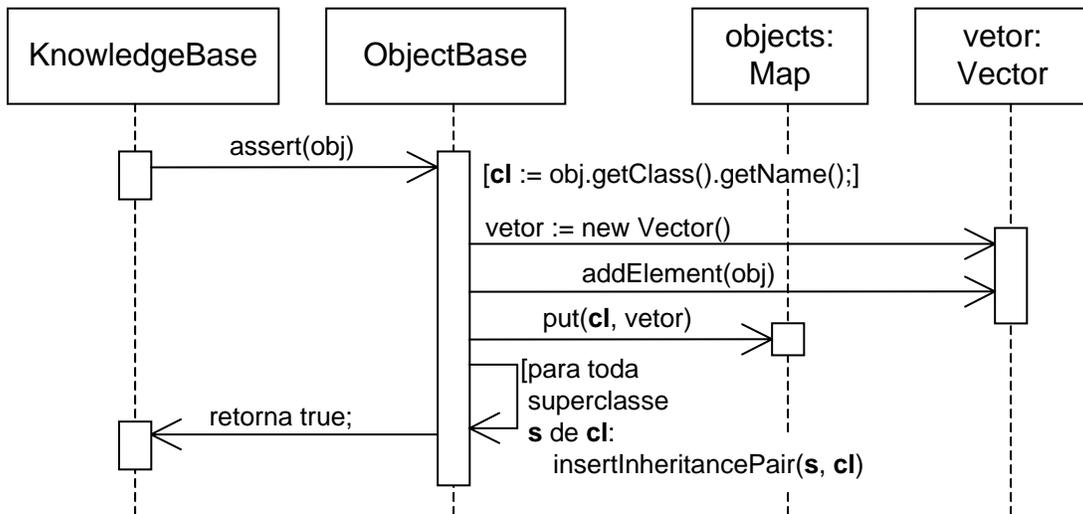


Figura 22 – Inserção de um objeto de uma nova classe na base de objetos.

Caso algum outro objeto da classe já tenha sido inserido na base, o método `assert` irá verificar se o objeto sendo inserido pertence à base: se for o caso, nenhuma alteração é feita na base de objetos e o método retorna o valor falso; caso contrário, o objeto é inserido no conjunto das instâncias de sua classe, e o valor verdadeiro é retornado à base de conhecimentos, indicando que a inserção foi realizada com sucesso. A Figura 23 ilustra este processo.

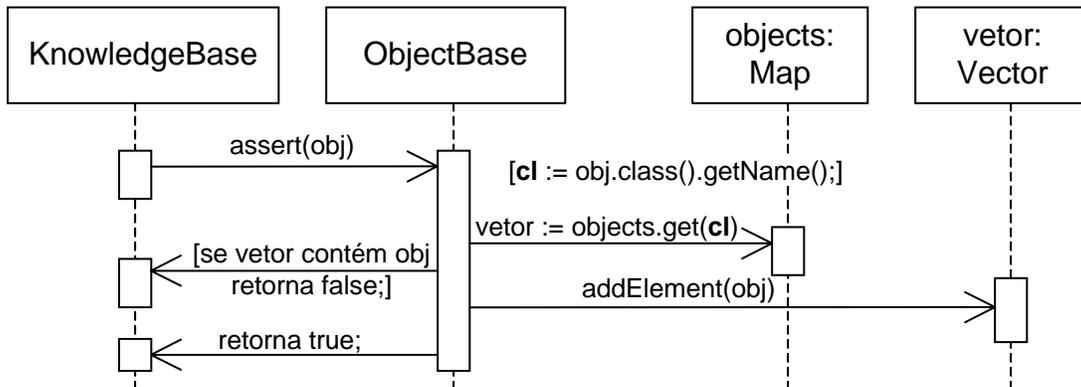


Figura 23 – Inserção de um objeto de uma classe já existente na base de objetos.

O armazenamento de informações sobre a hierarquia de herança, feito através da chamada ao método privado `insertInheritancePair`, apresentado na Figura 22. Este método atualiza o atributo `subClasses`, que mapeia o nome de uma classe no nome de todas as suas subclasses que já tiveram alguma instância inserida na base.

Como os objetos estão organizados de acordo com a sua classe, a implementação do método `remove(Object)` é simplificada: ela se resume à remover o objeto do conjunto de instâncias de sua classe. Da mesma forma, a implementação do método `flush()` simplesmente limpa os mapeamentos `objects` e `subClasses`.

A implementação da consulta dos objetos de uma determinada classe (método `objects(String)`) é apresentada em pseudocódigo na Tabela 30. Primeiramente é armazenado no resultado todos as instâncias diretas da classe desejada. Em seguida, o atributo `subClasses` é consultado para saber quais as subclasses da classe desejada que têm instâncias na base. Para cada uma delas, suas instâncias são adicionadas ao resultado, e por fim, o resultado é retornado para quem chamou este método.

```

Vector objects(nomeClasse) {
    ObjetosDaClasse := objects.get(nomeClasse);
    Result := ObjetosDaClasse
    Subclasses := subClasses.get(nomeClasse);
    Para todo c em Subclasses
        Result := Result ∪ objects.get(c);
    Retorne Result.
}
  
```

Tabela 30 – Implementação do método `objects(String)` da base de objetos.

4.2 O CONJUNTO DE CONFLITOS

Esta seção apresentará algumas implementações de conjuntos de conflitos disponibilizadas com o JEOPS. Será apresentada também uma classe abstrata que facilita a implementação de políticas próprias pelo usuário.

O usuário de JEOPS pode definir sua própria política de resolução de conflitos para ser usada em sua base de conhecimentos. No entanto, já definimos de antemão uma série de implementações da interface `jeops.conflict.ConflictSet`, tanto para testarmos a arquitetura utilizada no sistema, como para evitar que o usuário precise implementar as políticas mais comuns. As implementações já existentes de políticas de resolução de conflitos são as seguintes:

- ***jeops.conflict.DefaultConflictSet***
É o conjunto de conflitos utilizado quando nenhum outro é especificado. Sua política de resolução de conflitos não é especificada. Em outras palavras, o usuário não pode contar com uma determinada ordem de escolha das regras, pois a sua implementação poderá mudar nas futuras versões de JEOPS, de modo que seja sempre a mais eficiente possível.
- ***jeops.conflict.LRUConflictSet***
Política que escolherá sempre a regra menos recentemente utilizada. Se mais de uma regra existe no conjunto de conflitos, aquela que foi disparada há mais tempo passa a ter maior prioridade sobre as demais.
- ***jeops.conflict.MRUConflictSet***
Conjunto de conflitos que escolherá a regra *mais* recentemente utilizada. É a política complementar à anterior.
- ***jeops.conflict.NaturalConflictSet***
Conjunto de conflitos que não permite que uma regra seja disparada mais de uma vez com os mesmos objetos. Esta estratégia requer uma grande quantidade de memória para armazenar a história das regras, devendo portanto ser usada com cautela.
- ***jeops.conflict.OneShotConflictSet***
Conjunto de conflitos que não permite que uma regra seja disparada mais de uma vez, mesmo que com objetos distintos.

- *jeops.conflict.PriorityConflictSet*

Conjunto de conflitos que atribui prioridades às regras, segundo a ordem na qual elas aparecem no arquivo de regras: regras definidas no início do arquivo terão prioridade maior que as regras definidas no final. Em outras palavras, a ordem da definição das regras no arquivo define as suas prioridades.

A implementação de todos os conjuntos de conflitos mencionados acima apresenta diversos pontos em comum. No entanto, como o conjunto de conflitos foi definido como uma interface em Java (que não possui nenhuma implementação dos métodos que ela define), uma grande quantidade de código teria de ser replicado nas diferentes implementações. Para contornar este problema, criamos uma classe abstrata (*jeops.conflict.AbstractConflictSet*) que facilita bastante a implementação de conjuntos de conflito. Aspectos como o registro e notificação de ouvintes de eventos e remoção de elementos do conjunto foram implementados nesta classe. Desta forma, todas as implementações de conjunto de conflitos foram feitas como subclasses dessa classe abstrata.

Por exemplo, a Tabela 31 apresenta a implementação do método `insertElement` da classe `DefaultConflictSet`. Como um novo elemento foi adicionado ao conjunto de conflitos, então todos os ouvintes registrados devem ser notificados. Através de uma simples chamada ao método de *callback*, `elementAdded`, definido na superclasse, esta notificação é realizada. Os métodos que tratam do registro dos ouvintes também são implementados na classe abstrata, não precisando ser redefinidos nas subclasses.

```
public void insertElement(ConflictSetElement element) {
    if (!fireableRules.contains(element)) {
        fireableRules.addElement(element);
        elementAdded(element); // the callback method.
    }
}
```

Tabela 31 – Método `insertElement` da classe `DefaultConflictSet`

Um outro serviço disponibilizado pela classe `AbstractConflictSet` é a remoção de elementos do conjunto de conflitos. Mesmo em implementações mais complexas, onde os elementos não são armazenados em uma estrutura linear, mas indexados pelas regras (o que é o caso das estratégias de prioridade, MRU, LRU e OneShot), esta remoção já está implementada na classe abstrata. A Tabela 32 e a Tabela 33 apresentam a implementação

do método `removeElementsWith` nas classes `OneShotConflictSet` e `DefaultConflictSet`, respectivamente.

```
public void removeElementsWith(Object obj) {
    size -= removeElementsWith_2D(fireableRules, obj);
}
```

Tabela 32 – Método `removeElementsWith` da classe `OneShotConflictSet`

```
public void removeElementsWith(Object obj) {
    removeElementsWith_1D(fireableRules, obj);
}
```

Tabela 33 – Método `removeElementsWith` da classe `DefaultConflictSet`

4.3 A REDE RETE

Como foi dito na seção 3.5.4, a implementação de Rete do JEOPS não busca unificar condições repetidas entre as regras, tratando cada uma delas individualmente. Para cada regra, é então criada uma “mini-rede” por onde passarão os objetos que são inseridos na base de conhecimentos. A Figura 24 apresenta a rede criada para uma regra genérica de JEOPS.

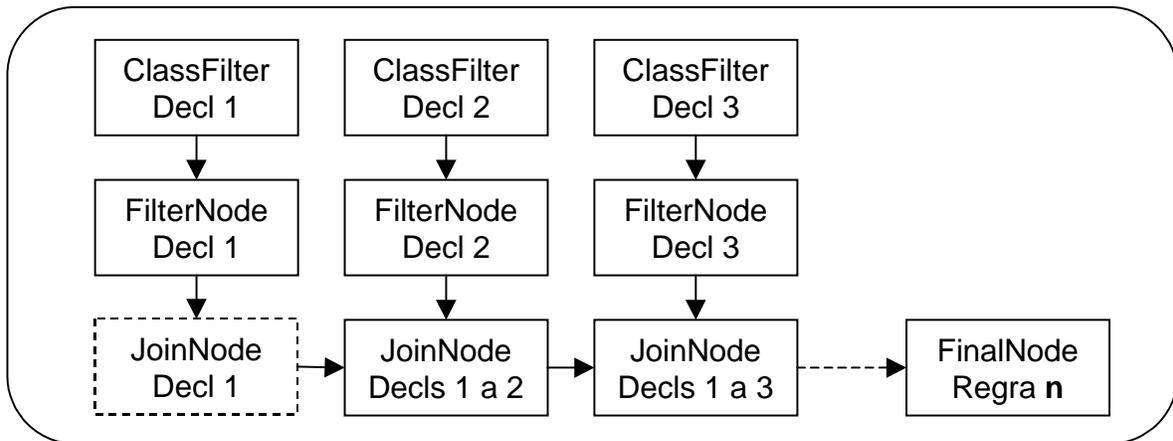


Figura 24 – Implementação *Rete* de JEOPS

Objetos entram na rede pelos chamados nós de *filtro de classe*. Para cada declaração da regra é criado um nó de filtro de classe, representando a classe da variável declarada. Desta forma, sempre que um novo objeto é inserido na base de conhecimentos, ele é enviado para todos os nós de filtro de classe da rede. Aqueles que representarem classes do qual o objeto é uma instância propagarão este objeto para seus sucessores. Caso contrário, o objeto não será propagado para o restante da rede.

Em seguida, as condições das regras são agrupadas de acordo com suas dependências em relação às declarações. Para cada declaração, todas as condições que dependem apenas dela são agrupadas, em um *nó de filtro* da rede. Se alguma das condições não for satisfeita para um dado objeto, ele não será propagado para o restante da rede. Todo nó de filtro sucede o nó de filtro de classe que corresponde à mesma declaração.

Após a filtragem inicial, os objetos que são propagados pelos nós de filtro se encontram nos *nós de junção*. Cada declaração tem um nó de junção associado¹ que verifica todas as condições que dependem da declaração e das declarações anteriores. Se um grupo de objetos passar por este nó quer dizer que todas as condições de que eles dependem já foram satisfeitas.

Finalmente, após o nó de junção associada à última declaração, é criado um *nó final*. Sempre que um conjunto de objetos atinge este nó, um novo elemento é adicionado ao conjunto de conflitos (pois o nó de junção anterior garante que os objetos fizeram com que todas as condições fossem satisfeitas).

A divisão das condições em grupos pode ser melhor explicada com um exemplo. Seja a regra `holdObjectHoldsSatisfied` apresentada na Tabela 34, usada na resolução do problema dos macacos e das bananas². As condições **c1** (que testa se o objetivo está ativo) e **c2** (que testa se o objetivo é segurar um objeto) dependem apenas da declaração **d1**. A condição **c4** (que testa se o objeto é leve) depende apenas da declaração **d3**. Não há nenhuma condição que dependa somente da segunda declaração, ou mesmo da primeira e da segunda juntas. As condições **c3** (que verifica se o objeto filtrado é o mesmo do objetivo), **c5** (que verifica se o macaco e o objeto estão na mesma posição) e **c6** (que verifica se o macaco está segurando o objeto) ficariam no nó de junção correspondente à terceira declaração, pois dependem dela e de declarações anteriores.

¹ A primeira declaração não tem um nó de junção; no entanto, podemos considerar que ele existe, e que propaga todos os objetos que chegam, para simplificar o modelo.

² A solução completa para este problema, definido em [42], se encontra junto com o código fonte de JEOPS [29].

```

rule holdObjectHoldsSatisfied {
  /*
   * Se o objetivo é segurar um objeto, e o macaco já
   * o está segurando, então o objetivo já está satisfeito.
   */
  declarations
    Goal g;           // d1
    Monkey m;         // d2
    PhysicalObject o; // d3
  conditions
    g.isActive();    // c1
    g.getType() == HOLD; // c2
    o == g.getObject(); // c3
    o.getWeight() == LIGHT; // c4
    m.isAt(o.getAt()); // c5
    m.isHolding(o); // c6
  actions
    System.out.println(m.getDescription() + " already holds " + o);
    g.setSatisfied();
    modified(g);
}

```

Tabela 34 – Regra do problema do macaco e das bananas

A Figura 25 ilustra a rede gerada para a regra apresentada acima. Deve-se observar que como não há nenhuma condição que restrinja somente a segunda declaração, todo objeto da classe `Monkey` será propagado diretamente para o nó de junção. E como não há nenhuma condição que dependa somente de $(d1 \wedge d2)$, este objeto será propagado para o terceiro nó de junção juntamente com todos os objetos da classe `Goal` que estiverem memorizados pelo segundo nó de junção (pois passaram pelo primeiro nó de filtro).

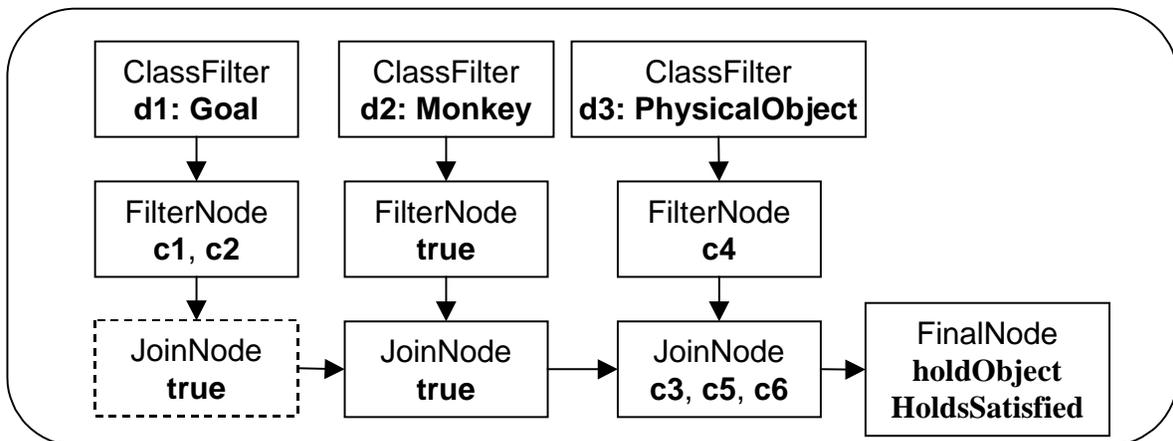


Figura 25 – Rede Rete da regra holdObjectHoldsSatisfied

Os nós de junção são os principais elementos da rede. Como os nós de filtro de classe, de filtro e finais não possuem nenhuma memória, são nos nós de junção que ficam armazenadas as unificações parciais da rede. Conforme pode ser visto na Figura 26, cada nó

de junção armazena duas listas de objetos: uma lista com todos os grupos de objetos que passaram pelo nó de junção anterior, e uma lista com os objetos que passaram pelo nó de filtro correspondente. O terceiro nó de junção, por exemplo, apresenta uma lista de pares de objetos provenientes do segundo nó de junção, e uma lista de objetos vindos do terceiro nó de filtro.

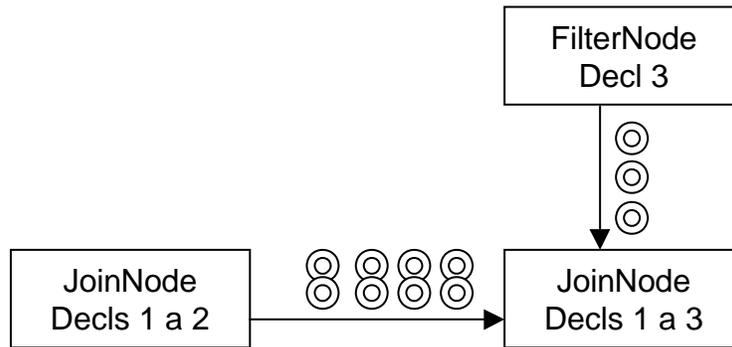


Figura 26 – Memória do nó de junção

Por exemplo, se um objeto da classe `PhysicalObject` for adicionado à rede da Figura 25, e passar pelo nó de filtro correspondente, ele só precisará ser verificado com os objetos que passaram pelo segundo nó de junção. Da mesma maneira, se um novo objeto da classe `Goal` for adicionado à rede e passar pelo nó de filtro da primeira declaração, ele será enviado para o segundo nó de junção: como este nó não verifica nenhuma condição, o objeto será enviado para o terceiro nó juntamente com todos os objetos que estavam na lista de espera do segundo nó de junção.

A implementação da passagem de objetos pelos nós de junção é feita um por um. Decidimos não criar uma estrutura para armazenar os objetos que iam sendo propagados pelos nós, pois é sabido que um dos maiores problemas de performance de Java é a criação de objetos. Como a propagação de objetos pela rede é uma operação que ocorre com uma frequência alta, decidimos que o maior esforço de programação para controlar a passagem individual dos objetos seria recompensada. Com isso, os nós de junção apresentam um número de entradas variável. O segundo nó de junção apresenta duas entradas (uma proveniente do nó de junção anterior e uma do nó de filtro correspondente) e duas saídas. O terceiro nó de junção, por sua vez, apresenta três entradas (duas provenientes das saídas do nó de junção anterior, e uma do nó de filtro). Com isso, a verificação se uma série de

objetos ativa um nó de junção só ocorre ou quando um objeto chega do nó de filtro, ou quando chega da *última* das entradas provenientes do nó de junção anterior.

Da mesma forma, os nós finais apresentam tantas entradas quantas forem as declarações da regra. Quando os objetos começam a chegar, eles são armazenados em um *buffer* interno até que chegue o último objeto. Só então é criado um novo elemento a ser inserido no conjunto de conflitos.

A remoção de objetos da rede é outra operação importante. Como é ela quem armazena, de forma distribuída, as unificações parciais que tiveram sucesso nas regras, se um objeto é removido da memória de trabalho ele também deve ser removido da memória da rede.

4.4 A BASE DE CONHECIMENTOS

A base de conhecimentos é a fachada de JEOPS. Uma vez realizada a pré-compilação do arquivo de regras, o usuário passará a interagir apenas com ela, abstraindo-se dos detalhes dos demais componentes da arquitetura. Esta seção, no entanto, apresentará os detalhes internos da base de conhecimento, mostrando como os métodos vistos pelo usuário são implementados.

4.4.1 Criação da base de conhecimentos

Quando um objeto da classe que representa a base de conhecimentos é criado, uma série de operações é realizada para sua inicialização. As operações principais desta inicialização, que ocorre no único construtor da classe, são apresentadas em pseudocódigo na Tabela 35.

```

Construtor AbstractKnowledgeBase(ConflictSet conflictSet) {
  1. Instancia a base de objetos
  2. Instancia a base interna de regras
  3. Cria a rede Rete
}

```

Tabela 35 – Inicialização da base de conhecimentos

O passo 1 do construtor cria uma base de objetos vazia, que será usada pela base de conhecimentos para o armazenamento das referências para os objetos da memória de trabalho.

A criação da base de regras constituiu um problema na implementação. Como tanto a base de conhecimentos precisa conhecer a base interna de regras (para verificar as

condições de uma regra, por exemplo), quanto a base interna de regras precisa conhecer a base de conhecimentos (para que as regras disparadas efetuem as operações de manipulação dos fatos da memória de trabalho), tivemos que garantir que na criação dos dois objetos eles estivessem consistentes (ou seja, cada um com uma referência para o outro). Não poderíamos, portanto, deixar para o usuário a tarefa de manter esta ligação.

Para resolver este problema, utilizamos o padrão de projeto conhecido por *factory method* [46], que pode ser aplicado em situações onde uma classe precisa deixar para a sua subclasse a instanciação de um determinado objeto. De acordo com a solução apresentada no padrão, a superclasse `jeops.AbstractKnowledgeBase` define um método abstrato usado para criar o objeto da base interna de regras. A subclasse, por sua vez, redefine este método, criando o objeto desejado. Quando o construtor da classe abstrata é chamado, ele invocará então o método correto. Este processo é ilustrado na Figura 27.

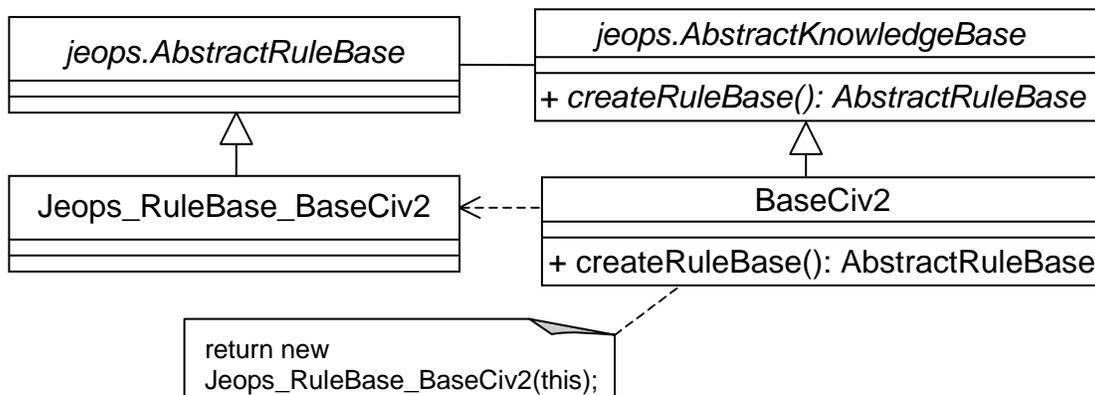


Figura 27 – Criação da base interna de regras

Esta é a razão pela qual são criadas duas classes durante a pré-compilação do arquivo de regras (a base de conhecimentos e a base interna de regras), e não apenas uma. Havia ainda outra possibilidade, a de criar a base de conhecimentos passando-se o nome do arquivo de regras: este arquivo seria então compilado e carregado dinamicamente. No entanto, esta solução limitaria o uso de JEOPS em *applets*, ou em qualquer ambiente no qual o compilador de Java não estivesse disponível, razão pela qual foi descartada.

A criação da rede Rete é o passo mais complexo desta inicialização. O processo de sua criação é apresentado em pseudocódigo na Tabela 36.

```

CreateReteNetwork() {
  Para cada regra r definida no arquivo de regras:
    Para cada declaração d de r:
      Crie um nó de filtro de classe correspondente à classe de d
      Crie um nó de filtro f que avalia as condições de d
      Conecte o nó de filtro de classe a f
      Crie um nó de junção j referente à declaração d
      Se d não é a primeira declaração:
        Conecte as saídas do nó de junção anterior a j
        Conecte a saída de f à última entrada de j.
}

```

Tabela 36 – Criação da rede Rete

Uma otimização foi feita em relação à criação dos nós de filtro de classe. Como várias regras podem ter declarações da mesma classe, então decidimos que estes nós seriam compartilhados entre as regras. Como o objeto chegaria de qualquer maneira a todos os nós, este compartilhamento não trouxe problemas.

4.4.2 A inclusão de objetos

A inclusão de um novo objeto na base de conhecimentos ocorre em duas etapas. Em primeiro lugar, o objeto é enviado à base de objetos, de modo que passe a fazer parte da memória de trabalho. Passando por esta etapa, ele será enviado à rede Rete para que esta tente unificá-lo com alguma declaração das regras, gerando novos elementos do conjunto de conflitos.

Quando o objeto é enviado à base de objetos, esta verifica se o mesmo já pertence à memória de trabalho. Em caso afirmativo, o objeto não será mais enviado à rede Rete, pois isto poderá acarretar a existência elementos idênticos no conjunto de conflitos. Neste caso, a inclusão do objeto é desconsiderada.

Se o objeto não pertencia à memória de trabalho, ele é enviado à todos os nós de filtro de classe da rede Rete. A partir deste ponto, Rete se encarrega de tentar unificá-lo com as declarações das regras (Cf. seção 4.3), encerrando-se a participação da base de conhecimentos na inclusão de objetos.

4.4.3 A remoção de objetos

Da mesma forma que a inclusão, o papel da remoção de um objeto é o de fazer com que ele seja removido tanto da memória de trabalho, quanto da memória de unificações parciais da rede Rete. Além disso, a base de conhecimentos também deve retirar do

conjunto de conflitos todos os elementos que contenham o objeto sendo removido. A Figura 28 apresenta a seqüência de mensagens envolvidas nesta remoção.

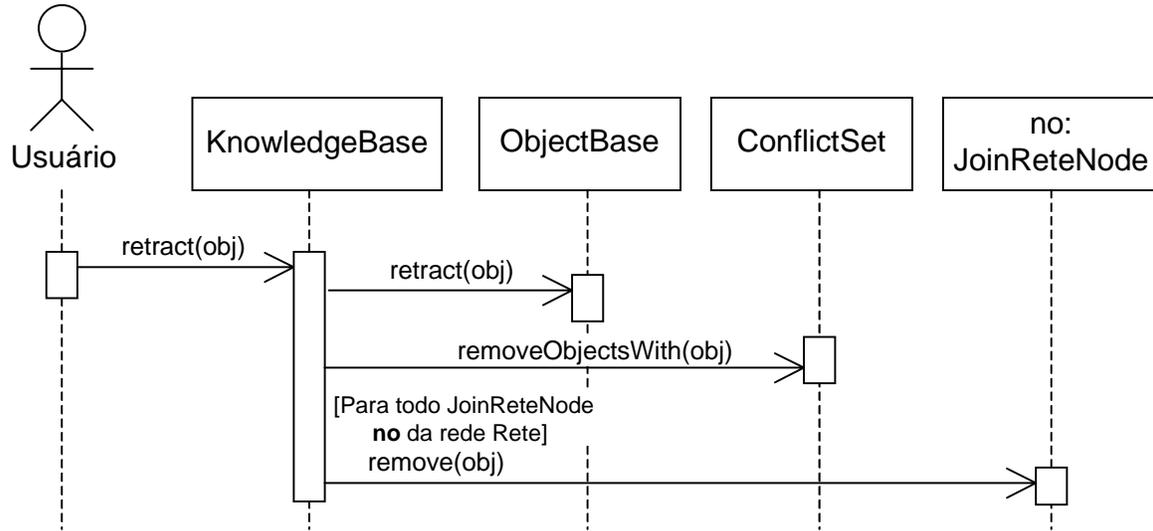


Figura 28 – Remoção de objetos da base de conhecimentos

A remoção do objeto da base de objetos e do conjunto de conflitos é feita simplesmente chamando o método apropriado. No caso da rede Rete, como apenas os nós de junção armazenam informações sobre unificações parciais (os demais nós não têm memória), a base de conhecimentos mantém internamente uma lista com todos os nós de junção criados. Quando o método `retract` é invocado, esta lista é percorrida, e é enviada a mensagem `remove` a todos os nós de junção para que removam de sua memória o objeto dado.

A operação de `reset` da base de conhecimentos, `flush()`, é implementada de forma similar: ao ser invocado, o método chama as operações de `flush` na base de objetos, no conjunto de conflitos e em todos os nós de junção da rede.

4.4.4 A modificação de objetos

A implementação de JEOPS do método usado pelo usuário para avisar que um objeto foi modificado (`modified`) segue a mesma filosofia apresentada na definição de Rete, e seguida por CLIPS, JESS e NéOpus. Nestes sistemas, a modificação de um objeto é implementada através da sua remoção, seguida pela sua inclusão na base de conhecimentos. A Figura 29 ilustra a seqüência de mensagens decorrente da chamada de `modified`.

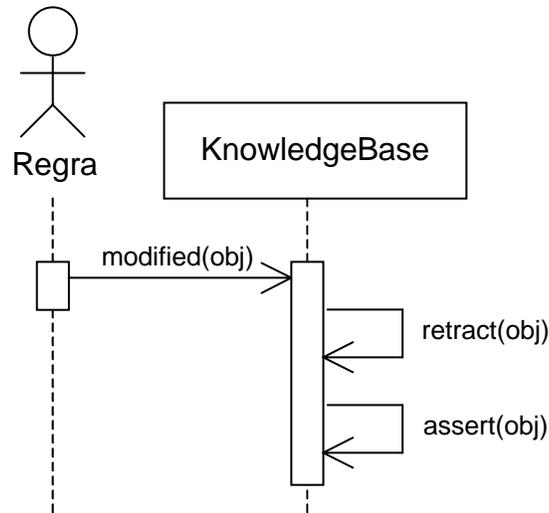


Figura 29 – Implementação do método `modified`

Deve-se ressaltar que esta implementação, apesar de simples, não é a mais eficiente. RAL/C++ utiliza o algoritmo Rete II™ de casamento de padrões, que a torna mais eficiente em aplicações onde a modificação de objetos é uma operação freqüente. Infelizmente, Rete II não está disponível, por se tratar de um produto comercial, razão pela qual não pudemos implementá-lo.

4.4.5 Consulta à memória de trabalho

A consulta aos objetos da memória de trabalho da base de conhecimentos é feita através do método `objects(String)`. Este método é implementado simplesmente através da chamada ao método do mesmo nome da base de objetos, conforme foi apresentado na seção 4.1.

4.4.6 Execução do motor de inferência

A maior complexidade do motor de inferência reside em dois passos principais: a unificação de objetos e declarações das regras; e a resolução de conflitos. Como o primeiro foi implementado na rede Rete, e o segundo é feito pela implementação do conjunto de conflitos, a implementação do método `run()`, que executa o laço principal do motor de inferência, ficou bastante simples, conforme pode ser observado no pseudocódigo da

```

run() {
    Enquanto o conjunto de conflitos não estiver vazio
        e := próximo elemento do conjunto de conflitos.
        i := índice da regra de e
        o := objetos usados na regra de e
        baseInternaRegras.setObjects(i, o);
        baseInternaRegras.disparaRegra(i);
}

```

Tabela 37 – Método run() da base de conhecimentos

Basicamente, o ciclo de execução da base de conhecimentos solicita continuamente ao conjunto de conflitos o próximo elemento (que será retornado de acordo com a política de resolução de conflitos implementada). Em seguida, é feito o mapeamento na base interna de regras das declarações da regra escolhida com os objetos correspondentes. Finalmente, a regra é efetivamente disparada. Este processo continua até que não haja mais nenhum elemento no conjunto de conflitos, quando então o método retorna para a aplicação.

4.5 A BASE INTERNA DE REGRAS

A base interna de regras é o componente responsável por responder à base de conhecimentos ou à rede Rete se as condições de uma regra são verdadeiras, quantas e quais regras foram definidas no arquivo pelo usuário, quais as declarações de cada regra. É também a base interna de regras que é responsável por disparar uma regra, quando solicitada pela base de conhecimentos.

Como foi mencionado na seção 3.5.3, a base de regras é implementada como uma classe abstrata de Java (`jeops.AbstractRuleBase`), que define a interface de seus métodos, sem no entanto implementá-los (o que será feito pela sua subclasse, gerada pela pré-compilação do arquivo de regras). Os principais métodos definidos nesta classe podem ser agrupados em diversos grupos, que serão apresentados nas subseções seguintes.

4.5.1 Informações genéricas

São métodos que retornam o número e o nome das regras, quantas condições e declarações tem cada regra, quais as classes das variáveis declaradas, entre outros. Estes métodos são apresentados a seguir:

- *int getNumberOfRules()*

Retorna o número de regras definidas no arquivo de regras pelo usuário.

- ***String[] getRuleNames()***
Retorna um *array* contendo o nome de todas as regras definidas. Este método é usado principalmente por interfaces de depuração, para apresentar que regra está sendo disparada em um determinado instante, como foi apresentado na Tabela 29 da seção 3.6.
- ***int[] getNumberOfDeclarations()***
Retorna um *array* contendo, para cada regra, o número de declarações que ela possui. É usado na construção da rede Rete pela base de conhecimentos.
- ***String[] getDeclaredIdentifiers(int ruleIndex)***
Retorna um *array* contendo o identificador das declarações de uma dada regra. É usado principalmente por interfaces de depuração, para apresentar o nome das variáveis da regra (e em quais objetos elas estão mapeadas).
- ***String getDeclaredClassName(int ruleIndex, int declarationIndex)***
Retorna o nome da classe de uma declaração de uma regra. É usado para a criação dos nós de filtro de classe da rede Rete.
- ***Class getDeclaredClass(int ruleIndex, int declarationIndex)***
Retorna um objeto que representa a classe de uma declaração de uma regra. Assim como o anterior, é usado para a criação dos nós de filtro de classe da rede Rete.
- ***int[] getNumberOfConditions()***
Retorna um *array* contendo, para cada regra, o número de condições que ela possui. Ainda não tem uso definido, mas foi criado porque achamos que o mesmo pode vir a ser utilizado no futuro.

4.5.2 *Atribuição e consulta de objetos*

São métodos através dos quais a base de conhecimentos e os nós da rede Rete podem mapear um objeto Java para uma das declarações de uma regra. Estes componentes podem também consultar os objetos que estejam mapeados às declarações das regras. O mapeamento dos objetos nas declarações deve ser feito antes da consulta às condições de uma regra, para que a verificação seja feita com os objetos corretos. O mesmo se aplica ao disparo das regras.

Os métodos usados na atribuição e consulta de objetos da base interna de regras são os seguintes:

- ***void setObject(int ruleIndex, int declarationIndex, Object value)***
Mapea uma declaração específica de uma regra em um dado objeto. É usado principalmente pelos nós de filtro da rede: quando um novo objeto chega no nó, ele é mapeado à declaração correspondente.
- ***void setObjects(int ruleIndex, Object[] objects)***
Mapea todas as declarações de uma regra de uma só vez nos objetos passados como parâmetro. É usado principalmente pela base de conhecimentos no momento do disparo de uma regra.
- ***Object getObject(int ruleIndex, int declarationIndex)***
Retorna o valor do objeto mapeado a uma das declarações de uma regra. É usado principalmente por interfaces de depuração, para obter o valor do objeto mapeado a uma dada declaração.
- ***Object[] getObjects(int ruleIndex)***
Retorna um *array* contendo todos os objetos mapeados às declarações de uma regra. Como o anterior, é mais usado por interfaces de depuração que desejem apresentar o valor de um objeto mapeado às declarações de uma regra.

4.5.3 Consulta a condições

São métodos para se verificar uma condição específica, todas as condições que dependem de uma determinada variável declarada, ou ainda todas as condições que dependem de um conjunto de declarações. Os métodos usados com este propósito são os seguintes:

- ***boolean checkCondition(int ruleIndex, int condIndex)***
Verifica se uma condição específica de uma dada regra é satisfeita pelos objetos mapeados às suas declarações. Ainda não tem uso definido no sistema, mas foi criado para o caso de alguma interface de depuração desejar verificar a veracidade de uma condição de uma regra.

- ***boolean checkConditionsOnlyOf(int ruleIndex, int declIndex)***
Verifica se todas as condições que dependam somente de uma declaração da regra dada são satisfeitas. Usado pelos nós de filtro da rede Rete.
- ***boolean checkCondForDeclaration(int ruleIndex, int declIndex)***
Verifica se todas as condições da regra que dependam da declaração dada, e de alguma declaração anterior são satisfeitas. Usado pelos nós de junção da rede Rete.

4.5.4 *Disparo de regras*

Há apenas um método neste grupo, chamado pelo motor de inferência sempre que uma regra tiver de ser disparada. Antes da chamada deste método, os objetos usados pela regra devem ser mapeados às suas declarações.

- ***void fireRule(int ruleIndex)***
Executa o campo de ações da regra dada. É chamado pelo método `run()` da base de conhecimentos para cada elemento retornado do conjunto de conflitos.

4.5.5 *Manipulação da memória de trabalho*

Além dos métodos utilizados pela base de conhecimentos e pela rede Rete, a classe abstrata `jeops.AbstractRuleBase` também implementa diversos métodos utilizados para a manipulação da memória de trabalho. As operações de inserção, remoção e notificação de modificação são implementadas da mesma forma em todas as bases de regras, razão pela qual decidimos colocar a sua implementação diretamente na classe abstrata. Estes métodos, que podem ser usados pelo usuário no campo das ações das regras, são os seguintes:

- ***void assert(Object obj)***
Insere o objeto dado na memória de trabalho. Além disso, ele será enviado à rede Rete para tentar ser unificado com as declarações das regras, gerando novas entradas no conjunto de conflitos.
- ***void retract(Object obj)***
Remove o objeto dado da memória de trabalho. Além disso, todos os elementos do conjunto de conflitos que contiverem este objeto serão removidos.
- ***void modified(Object obj)***
Notifica o motor de inferência que o objeto dado foi modificado.

- *void flush()*

Remove todos os elementos da memória de trabalho. Em consequência, o conjunto de conflitos ficará vazio, e o ciclo de execução do motor de inferência será terminado.

Todos estes métodos são definidos também na base de conhecimentos. De fato, suas implementações simplesmente redirecionam a mensagem recebida para a própria base de conhecimentos. A Figura 30 ilustra a seqüência de mensagens envolvidas na implementação do método de `retract`, por exemplo.

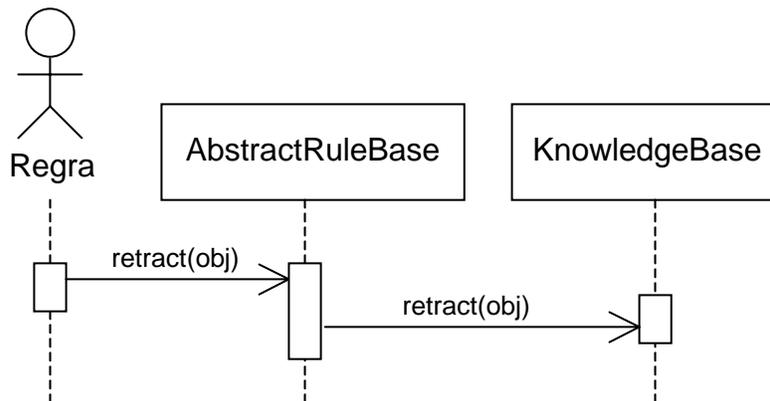


Figura 30 – Implementação de `retract` na base interna de regras

4.6 A PRÉ-COMPILAÇÃO REGRAS → JAVA

A pré-compilação do arquivo de regras consiste na geração de duas classes: uma base interna de regras (subclasse de `jeops.AbstractRuleBase`) e uma base de conhecimentos (subclasse de `jeops.AbstractKnowledgeBase`). A função da base interna de regras é a de implementar todos os métodos definidos pela superclasse (Cf. seção 4.5) de acordo com o especificado nas regras. A base de conhecimentos é gerada principalmente para a implementação do padrão de projeto *factory method*, usado na criação da base interna de regras (Cf. seção 4.4.1).

Conforme foi dito na seção 3.2.1, decidimos que iríamos implementar “do zero” um *parser* para analisar as regras do JEOPS. O trabalho do *parser* consiste em aplicar uma série de regras de rescrita, gerando os métodos da base interna de regras à medida que lê as regras.

4.6.1 O processo de pré-compilação

A pré-compilação é efetuada pelo reconhecimento de uma série de padrões que existem no arquivo de regras. Como pode ser visto no diagrama de estados simplificado da Figura 31, o pré-compilador é iniciado no estado “Fora de base de regras”. Enquanto se encontra neste estado, tudo o que é lido é escrito no arquivo de saída: comentários, declarações de pacotes, importação e definições de classes, etc. No momento que o identificador “ruleBase” aparece na entrada, o pré-compilador escreve na saída a palavra reservada “class”, e a geração da classe da base interna de regras começa a partir deste ponto.

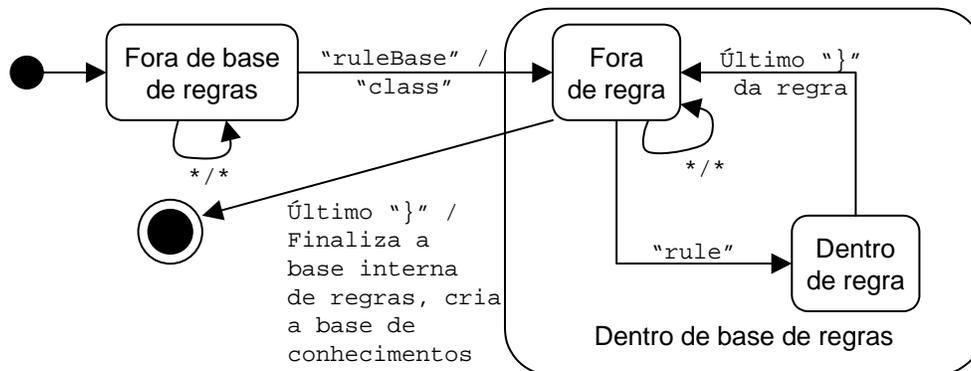


Figura 31 – Estados do pré-compilador da base de regras

Uma vez no estado “Dentro de base de regras”, o pré-compilador continua colocando na saída todos os elementos da entrada – métodos, declarações de variáveis, *inner classes*³ – até que seja encontrado o identificador “rule”.

Uma vez dentro de uma definição de regra, o pré-compilador se encontra no campo das condições de uma regra. Durante o processamento deste campo, é feito o mapeamento dos identificadores das variáveis declaradas em novos nomes que serão declarados como variáveis da classe gerada, conforme apresentado na Tabela 38.

Para cada condição c da forma “ Classe id; ” Escolha um nome n para a variável que representará id Insira no mapeamento das substituições o par (id, n)
--

Tabela 38 – Processamento das condições de uma regra.

O nome escolhido é baseado na classe da declaração. Assim, uma declaração do tipo “`java.util.Vector v;`” poderia ser mapeada em uma variável chamada

³ Definição de uma classe dentro de outra classe de Java [72].

“java_util_Vector_1”, por exemplo. Esta nomenclatura foi escolhida para facilitar o compartilhamento destas variáveis pelas declarações das demais regras da base.

Se houver o campo opcional com declarações locais, estas são lidas e colocadas em um mapeamento a parte; desta forma, no lugar de cada ocorrência livre de um de seus identificadores nas condições ou nas ações, a expressão associada é colocada. Por exemplo, se houver uma declaração local “Pessoa pai = p.getSalario();”, cada ocorrência livre do identificador `pai` será substituída pela expressão “(p.getSalario())”.⁴

Após o campo das declarações de regras, segue-se o campo das condições. Cada condição de uma regra dá origem a um método da classe gerada, que faz a sua verificação. Este método é gerado a partir da substituição de toda ocorrência livre dos identificadores das declarações pela sua variável correspondente. Por exemplo, a condição “v.size()==0”, onde “v” é a variável definida no parágrafo anterior geraria um método com o corpo “return (java_util_Vector_1.size()==0);”.

Finalmente, o campo de ações segue as declarações de uma regra. Para cada regra, é gerado um método com o mesmo nome da regra, que contém todas as instruções definidas no campo de ações (feitas as substituições das declarações da regra). Como foi mostrado na Figura 31, o campo de ações se estende até o último “fecha chave” da regra. Esta verificação é feita através da contagem das chaves que aparecem na regra. Quando a diferença entre as chaves abertas e fechadas for igual a zero, então o *parser* entende que a regra terminou. Da mesma maneira, o *parser* entenderá que a base de regras foi encerrada quando a diferença entre as chaves abertas e fechadas também for igual a zero.

Quando o *parser* atinge o final da base de regras, todos os métodos definidos pela classe abstrata são gerados. Estes métodos são implementados através da chamada aos métodos criados quando da passagem pelas regras. Por exemplo, a implementação do método `fireRule()`, da base de regras definida para o problema do jogo Civilization 2 (Cf. seção 3.4) é apresentada na Tabela 39.

⁴ Por ocorrência livre de um identificador, entenda-se que ele não é um campo de um objeto ou nome de método. Por exemplo, o identificador “pai” é livre na expressão “pai.getNome()” é livre, mas nas expressões “p.pai.getNome()” ou “p.pai()” ele não o é.

```
public void fireRule(int ruleIndex) {
    switch (ruleIndex) {
        case 0: defendeCidade(); break;
        case 1: protegeCidade(); break;
        case 2: atacaCidade(); break;
        case 3: exploraMundo(); break;
        ...
    }
}
```

Tabela 39 – Implementação do método `fireRule` do exemplo de Civilization II

Finalmente, após a conclusão da classe que representa a base interna de regras, o pré-compilador cria a classe da base de conhecimentos. Esta classe define apenas três métodos: dois construtores (um que recebe uma instância de um conjunto de conflitos, e um que utiliza o conjunto padrão), e o *factory method*, usado para se criar a base interna de regras.

4.6.2 A compilação das classes Java

A versão 2.1 de JEOPS incorporou um serviço que facilita a utilização do sistema. Em vez de gerar arquivos Java que em seguida precisariam ser compilados pelo usuário, o próprio pré-compilador de JEOPS se encarrega de realizar esta tarefa. Além disso, se a base de regras utilizar classes que ainda não estejam compilada, o pré-compilador de JEOPS também irá compilar estas classes.

A implementação deste serviço foi feita através de uma chamada ao próprio compilador Java: como a linguagem permite que sejam executados programas externos, utilizamos esta funcionalidade para que a implementação fosse feita da maneira mais simples possível. Com isso, o trabalho do usuário é facilitado.

4.7 SÍNTESE

Apresentamos neste capítulo uma visão mais aprofundada da nossa implementação de JEOPS, feita a partir da arquitetura apresentada no capítulo anterior. Mostramos os principais métodos dos blocos que compõem o sistema, bem como o processo de pré-compilação usado para transformar o arquivo de regras em classes Java que implementam estas regras.

Esperamos ter atingido o principal objetivo deste capítulo: fornecer informações suficientes para que um desenvolvedor seja capaz de, além de compreender o sistema,

contribuir, com novos módulos ou com melhorias nos módulos existentes, para o crescimento de JEOPS.

5 RESULTADOS OBTIDOS

Apresentaremos a seguir alguns exemplos reais de utilização de JEOPS. Para manter o texto conciso, decidimos colocar os “*toy examples*” que utilizamos como estudos de caso do JEOPS no apêndice deste trabalho (Cf. Anexo C), concentrando este capítulo apenas nos problemas “reais”. Apresentaremos também um *benchmark* realizado para validarmos o nosso sistema, comparando-o com os similares.

5.1 UTILIZAÇÃO DE JEOPS

Diversos projetos já foram (e vem sendo) desenvolvidos utilizando o JEOPS como ferramenta de inferência. Suas áreas variam da administração de redes a lingüística computacional, passando por jogos interativos, comércio eletrônico e recuperação de informação na Internet. Esta seção apresentará alguns deles, de modo a dar uma idéia do potencial de JEOPS.

5.1.1 Mobilet

Em um sistema de arquivos de rede, a informação é armazenada nos discos dos diversos nós do sistema. Em diversas ocasiões, o espaço livre de algum destes discos torna-se escasso, comprometendo certas funções do sistema operacional, como paginação de memória ou armazenamento de arquivos temporários pelas aplicações. Neste caso, o operador da rede tem que proceder à limpeza destes discos, movendo arquivo para nós com mais espaço, compactando arquivos pouco utilizados ou removendo arquivos que não são mais necessários.

A proposta do sistema Mobilet é de automatizar esta tarefa do operador, através de um agente móvel que percorre continuamente os nós da rede, realizando a verificação de problemas de espaço em disco. Uma vez detectado algum problema no disco (que é o caso de uma partição estar com o espaço livre abaixo de um limiar mínimo preestabelecido), a agente dispara uma base de conhecimentos que contém diversas regras usadas para a limpeza do disco. Estas regras representam as mesmas ações que seriam tomadas por um administrador “real” da rede. O administrador pode também acompanhar o processo, a partir de um monitor criado pelo sistema.

JEOPS foi usado no desenvolvimento de Mobilet como o seu motor de inferência. Com a facilidade provida por JEOPS, a implementação das regras utilizadas pelo administrador foi feita rapidamente. Mesmo tendo sido utilizada a versão 1.0 de JEOPS (com regras interpretadas, sem Rete), a performance do sistema não apresentava problemas – o maior gargalo era a movimentação do agente pela rede.

5.1.2 *Enigmas*

Enigmas, originalmente conhecido por “Enigma no Campus” [31], é um jogo de aventura, onde o jogador imerge num mundo habitado por diversas criaturas, com as quais pode se engajar em diálogos para desvendar os diversos *enigmas* do Campus. À medida que descobre as soluções para os problemas propostos, o jogador vai acumulando experiência, até se tornar um novo guardião do conhecimento, a partir de que poderá ele mesmo propor novos enigmas.

As criaturas que habitam o mundo no qual se passa o jogo são implementados como *atores sintéticos* (ou *believable agents* [73]), agentes inteligentes que têm como objetivo maior passar uma ilusão de vida, de um personagem. Esta ilusão é conseguida através da apresentação de uma coerência de emoções, personalidades e relações sociais.

JEOPS foi utilizado na implementação do sistema como ferramenta de raciocínio dos agentes/criaturas, bem como do controle do ambiente do mundo. Mais uma vez foi utilizada a versão 1.0 do sistema, sem que apresentasse problemas de performance.

5.1.3 *PubsFinder*

A explosão da Internet ocorrida na última década fez com que a recuperação de informações da rede se tornasse uma tarefa não trivial. Como não há um padrão de estruturação para os documentos presentes na Internet (em princípio, todo arquivo de computador é um documento Internet em potencial), a classificação destes documentos é um processo pouco recomendável (senão impossível).

No entanto, a informação disponível na Web não se encontra tão desestruturada como se imagina. Em certos domínios específicos, é possível se encontrar classes bem definidas de páginas, tais como: *call for papers* (CFP), *frequently asked questions* (FAQ), páginas pessoais, etc. Com características próprias, é possível a criação de agentes de busca que façam o reconhecimento de páginas de uma classe em particular.

O PubsFinder (nome provisório) é um projeto de mestrado que propõe o desenvolvimento de um sistema para a recuperação de documentos de um domínio específico. Neste caso, são páginas eletrônicas na Web que possuam citações a artigos, como por exemplo, páginas pessoais de pesquisadores, páginas de congressos, entre outras.

JEOPS vem sendo usado na implementação das regras de recuperação de informação. O PubsFinder foi o primeiro sistema a utilizar a versão 2.0 de JEOPS, que passou sem problemas pelo seu primeiro teste beta.

5.1.4 Gerador de variações fonéticas

Um serviço disponibilizado pela maioria das operadoras de telefonia fixa é o de auxílio a lista. Através dele, o usuário pode solicitar a um operador o telefone de uma pessoa ou empresa. Além da comodidade para o usuário, este serviço é uma fonte extra de lucros para as operadoras implementado sem maiores dificuldades (por uma consulta simples ao banco de dados dos clientes).

Entretanto, os nomes de algumas pessoas ou estabelecimentos comerciais não seguem à risca a língua portuguesa. O restaurante “*Kasa Nostra*” no Paraná, a cantora “*Selma du coco*” em Pernambuco e diversos outros exemplos mostram que a simples pesquisa pela maneira como o operador acha que se escreve o que o usuário deseja pode deixar de retornar resultados que seriam relevantes.

O projeto do gerador de variações fonéticas [74] surgiu a partir do interesse de uma companhia telefônica em resolver este problema. Basicamente, o sistema foi desenvolvido para ser acoplado em alguma ferramenta de busca (seja da Internet, seja de algum sistema interno, como é o caso do auxílio a lista), gerando todas as variações fonéticas de uma determinada palavra.

JEOPS foi usado para dar uma maior modularidade ao gerador. Por separar as regras de rescrita usadas na geração das variações fonéticas do restante do sistema, a sua manutenção (seja pela adição de novas regras, seja pela alteração de regras existentes) tornou-se muito mais simples.

5.1.5 NetMaze

NetMaze [32] é um jogo de ação multiusuário distribuído que se passa em um labirinto, no qual existe uma caça e vários caçadores. Dependendo do momento no qual o

jogador se conecta ao jogo, ele pode assumir o papel de um caçador (que tem por objetivo procurar a caça no labirinto) ou da caça (que tem por objetivo permanecer viva o maior tempo possível). Há ainda jogadores controlados pelo computador (NPCs), implementados como agentes inteligentes.

JEOPS foi utilizado na implementação da inteligência dos NPCs. As regras são usadas para implementar a estratégia de busca da caça, ou de fuga dos caçadores, dependendo do personagem.

5.1.6 Utilização como Ferramenta de Ensino

Além de implementações práticas, JEOPS vem sendo utilizado como ferramenta de ensino de inteligência artificial no CIn-UFPE. Desde o semestre seguinte à conclusão da sua primeira versão (há dois anos), JEOPS é uma das ferramentas utilizadas por alunos de graduação e mestrado na implementação de vários projetos de disciplinas. Estes projetos incluem equipes de robôs para a RoboCup [76], reconhecimento de páginas com cifras musicais, classificação de páginas da rede, entre outros das mais diversas áreas.

Um resultado importante decorrente desta utilização é o fato de que a maioria dos alunos que usou o JEOPS na implementação de seus projetos não tem tido problemas com o sistema. Além do mais, dentre os alunos que já utilizaram mais de um mecanismo de inferência no desenvolvimento de seus projetos (como JEOPS, JESS e Prolog), quase todos nos disseram JEOPS foi muito mais fácil de ser usado que os demais sistemas. Como a maior parte dos alunos das disciplinas do CIn-UFPE tem um *background* em Java razoavelmente sólido (pois há uma disciplina que ensina a linguagem em um semestre anterior ao que são ministradas as disciplinas da área de Inteligência Artificial), acreditamos que o nosso objetivo de prover um sistema de fácil utilização principalmente para programadores Java foi atingido. E mesmo os alunos que vem com um *background* mais forte da área de IA não tem tido muitos problemas na utilização de JEOPS, passada a fase inicial de adaptação do modelo híbrido regras/objetos.

O retorno que obtivemos desta utilização do JEOPS também foi muito importante na melhoria do sistema. Como os alunos das disciplinas não tinham os conhecimentos de programação ou de Inteligência Artificial nivelados, o manual de utilização do sistema passou por diversas modificações para responder de maneira cada vez mais satisfatória as

dúvidas dos usuários. Além disso, com a utilização maciça do sistema, pudemos encontrar e corrigir diversos *bugs* existentes, principalmente da versão 1.0.

5.2 COMPARAÇÃO COM OUTROS SISTEMAS

Esta seção apresentará uma comparação de JEOPS com alguns dos sistemas apresentados neste trabalho. Será apresentada uma análise das características dos sistemas, bem como da sua performance.

5.2.1 Resumo das características

A Tabela 40 apresenta de forma resumida uma comparação entre as características dos sistemas de produção apresentados neste trabalho.

Critério	Uniformidade de Integração	Serviços	Desempenho	Expressividade	Encadeamentos	Estratégias de controle
Sistema						
CLIPS	-	-	+/-	L1 (*)	Prog.	-
RAL/C++	+/-	+	+	L1 (*)	Prog.	-
NéOpus	+	-	+/-	L1 (*)	Prog.	+
JESS	-	+/-	+/-	L1 (*)	Prog./regr.	-
JEOPS	+	+	+/-	L1 (*)	Prog.	+/-

Tabela 40 – Comparação entre os sistemas apresentados (* = Hipótese do mundo fechado)

Em relação ao primeiro critério, NéOpus e JEOPS apresentam a maior uniformidade da integração entre as regras e a linguagem hospedeira, por utilizarem a mesma filosofia das linguagem nas regras, não impondo nenhuma espécie de restrição aos objetos que podem ser usados no sistema. RAL/C++ representa um passo em direção à integração, mas a utilização de condições no estilo OPS5, quebrando o encapsulamento dos objetos faz com que o sistema perca pontos neste critério. CLIPS e JESS, que utilizam a filosofia de criar um mundo à parte do mundo dos objetos da linguagem, apresentam uma integração muito pouco uniforme.

Em relação aos serviços disponíveis, RAL/C++ pode utilizar toda a base existente de componentes C++. O mesmo ocorre com o JEOPS, em relação à Java. Como JESS limita os objetos que podem ser usados nas regras (Cf. seção 2.8), é possível que existam serviços de Java que não podem ser utilizados. CLIPS, por não facilitar o uso de objetos externos ao mundo COOL, perde pontos neste critério, enquanto que os serviços que NéOpus podem utilizar se restringem aos de Smalltalk, uma linguagem cujo suporte não acompanha o ritmo das demais linguagens.

Apenas RAL/C++ se sobressai em relação ao desempenho. Por implementar uma variação mais eficiente de Rete, o sistema apresenta uma melhor performance na maioria das aplicações. Com respeito à expressividade, todos os sistemas podem representar condições equivalentes à lógica de primeira ordem (com a hipótese do mundo fechado).

Na questão dos encadeamentos, todos os sistemas implementam encadeamento progressivo. JESS dá uma facilidade a mais a seus usuários, por permitir a definição de regras baseadas no encadeamento regressivo.

Finalmente, NéOpus é o sistema que dá mais facilidade ao usuário, por permitir que este defina declarativamente a estratégia de controle do disparo das regras. JEOPS também permite que o usuário defina sua própria política, mas este tem de implementar uma classe específica para isso. Os demais sistemas permitem apenas a escolha de uma das estratégias pré-definidas, não dando maior liberdade ao usuário.

O que a Tabela 40 mostra é que não existe um sistema melhor que outro em todos os aspectos. Em aplicações onde existe a necessidade da utilização de encadeamento regressivo, JESS é o mais indicado. Naquelas onde a definição da estratégia de resolução de conflitos é crucial, NéOpus surge como a melhor opção. Para aplicações que necessitam de uma grande quantidade de serviços, JEOPS se sobressai.

5.2.2 Benchmark

Apesar de não entendermos que a eficiência seja a característica mais importante dos sistemas de produção, achamos interessante apresentar algumas indicações relativas à eficiência das inferências em JEOPS. Como não conseguimos ter acesso a uma implementação de RAL/C++, este sistema foi deixado de fora da comparação. Da mesma forma, como só dispomos de um interpretador Smalltalk para o sistema Unix, e não dispomos do ambiente de execução de Java (versão 1.2, usada pelo JEOPS) para este sistema operacional, também deixamos NéOpus de fora. No entanto, como o RAL/C++ e o NéOpus não são muito utilizados atualmente, acreditamos que a comparação de JEOPS apenas com CLIPS e JESS seja válida.

Nesta seção, apresentaremos um exemplo simples, que pode ser implementado com poucos problemas em todo sistema de primeira ordem: o cálculo dos elementos da série de Fibonacci. A descrição deste problema, bem como sua implementação em JEOPS pode ser vista no Anexo C. Utilizamos este exemplo pois como a solução recursiva para se calcular

o valor de **Fib(n)** (o n-ésimo elemento da série) cresce exponencialmente com o aumento do valor de **n** [73], ele é freqüentemente utilizado como *benchmark* para a avaliação de motores de inferência. Uma solução iterativa para o problema (muito mais eficiente) pode ser encontrada na maioria dos livros de introdução à programação.

A Tabela 41 apresenta os resultados dos testes. Eles foram executados em PCs com processador K6-2 de 300MHz de relógio e memória RAM de 64MB.

Sistema	CLIPS	JESS	JEOPS
Problema	Versão 6.10 Win	Versão 5.1	Versão 2.1b1
Fib(12)	2''	1''	1''
Fib(15)	3'16''	24''	14''
Fib(18)	- - - ¹	8'30''	4'03''

Tabela 41 – Resultados dos *benchmarks*

Como pode ser observado na tabela acima, JEOPS apresenta um desempenho melhor que o de JESS no problema de Fibonacci, mesmo utilizando a unificação comportamental. O mau desempenho de CLIPS nos impressionou, o que nos leva a crer que quando o suporte a um sistema é encerrado, as suas chances de parar de evoluir são grandes.

5.3 SÍNTESE

Este capítulo mostrou que JEOPS já vem sendo utilizado com sucesso em projetos bastante diversos. O *feedback* que recebemos dos seus autores foi essencial na melhoria do sistema (principalmente no seu início), de modo que os projetos que utilizam o JEOPS a partir de sua versão 2.0 já tiveram muito menos problemas que os pioneiros.

Foi apresentado também um quadro comparativo dos sistemas apresentados neste trabalho. Apesar de não existir um sistema ideal para todo tipo de aplicação, mostramos que JEOPS é um bom candidato em uma grande gama de aplicações. Finalmente, foi mostrado que o desempenho de JEOPS está na mesma ordem de grandeza que o de JESS, mesmo com o aumento da facilidade de uso.

¹ O programa ficou em execução por mais de duas horas, quando foi interrompido.

6 CONCLUSÕES

Apesar da existência de problemas próprios, como o da modificação dos objetos, a integração entre regras de produção e linguagens orientadas a objetos têm se mostrado bastante promissora. As vantagens da união, tais como diversidade de serviços, modularidade, legibilidade, estruturação, reusabilidade, possibilidade da utilização de metodologias de desenvolvimento, definição natural de conceitos ontológicos, entre outras, são extremamente importantes para a criação de aplicações baseadas no conceito de agentes inteligentes.

No entanto, as soluções existentes ainda apresentam diversos problemas que impedem a ampla difusão da área. NéOpus [26], o melhor dos sistemas no ponto de vista conceitual, foi desenvolvido para Smalltalk, uma linguagem que está fora do *mainstream* da computação. Os demais sistemas pecam pela dificuldade que impõem ao usuário na criação das regras, por apresentarem uma integração com a linguagem hospedeira pouco uniforme.

Com o trabalho de JEOPS, tentamos construir uma ferramenta que reunisse as principais vantagens dos sistemas estudados, prezando sobretudo para que fosse fácil de ser usada, e que o usuário tivesse à sua disposição todos os serviços oferecidos pela linguagem Java. Para tanto, definimos que a sintaxe utilizada na definição das regras seria bem próxima à de Java, diminuindo o tempo necessário para o aprendizado do sistema. Além disso, permitimos que todo e qualquer objeto da linguagem fosse utilizado como fato na memória de trabalho, garantindo assim ao desenvolvedor a possibilidade da reutilização dos serviços que nos fizeram escolher Java como linguagem de implementação do JEOPS.

Com uma integração uniforme, conseguimos fazer um sistema que mantivesse todas as vantagens da orientação a objetos. Com a clara divisão da definição do problema em elementos da ontologia (os objetos) e do comportamento, o sistema é desenvolvido de forma mais modular, facilitando a sua manutenção. Com o encapsulamento de informações nos objetos, as regras do sistema podem ficar bem mais concisas, facilitando a sua compreensão. Finalmente, por facilitar a reutilização de componentes, a filosofia de trabalho de JEOPS permite que o programador reaproveite módulos de outros sistemas, diminuindo assim o esforço necessário para o desenvolvimento.

JEOPS já foi usado em diversos projetos, conforme apresentado na seção 5.1. Com o seu sucesso em áreas tão diversas quanto fonética, administração de rede, jogos e recuperação de informação, o sistema já apresenta uma maturidade suficiente para ser usado em aplicações mais robustas. Com a implementação de Rete, acreditamos que JEOPS poderá se difundir muito além das fronteiras do Recife.

JEOPS é o resultado de um trabalho de quase três anos de desenvolvimento. No entanto, ainda há muito a ser feito para melhorar cada vez mais o sistema. Além disso, a área de EOOPS ainda têm alguns problemas em aberto, o que abre caminho para novos trabalhos a serem realizados.

Em relação à eficiência do sistema, esta sempre será uma questão importante em um sistema de produção. O preço pela maior abstração dada pela programação declarativa é a complexidade de um controle externo (o motor de inferência) que deve escolher dentre as diversas regras do sistema, e dentre os diversos fatos (ou objetos) da memória de trabalho aqueles que serão ativados em cada instante. Com isso, qualquer ganho que se tenha em termos de performance do sistema são bem vindos, e JEOPS suporta certas melhorias que podem ser feitas no futuro. Em primeiro lugar, a forma como a compilação das regras trata as variáveis locais pode ser melhorado. Como cada ocorrência de uma variável local é expandida pela sua definição (Cf. seção 4.6.1), a avaliação de sua expressão é feita redundantemente, se a mesma é utilizada em mais de um local na regra. Se as mesmas fossem propagadas através da própria rede Rete, este problema estaria resolvido. Uma outra melhoria poderia ser feita em relação à eficiência nos nós de junção da rede. Para armazenar a lista com as unificações parciais (Cf. seção 4.3), utilizamos as estruturas genéricas definidas em Java. Como este é um ponto crítico do sistema, a implementação de estruturas específicas (baseadas em tabelas *hash*, por exemplo) pode melhorar a performance da unificação.

Em relação à facilidade de uso do sistema, acreditamos que uma interface de edição de regras, com ajuda on-line sobre a sintaxe das regras, auxiliaria bastante o usuário, principalmente o iniciante. Além disso, acreditamos que uma interface de execução passo a passo do motor de inferência é essencial para uma aceitação ampla de JEOPS.

Além do trabalho de implementação, ainda há diversos temas que merecem atenção na área de EOOPS. O problema da modificação de objetos, principalmente o transitivo,

deve ser estudado mais a fundo, pois acreditamos que os sistemas atuais (inclusive o JEOPS) utilizam apenas paliativos para resolver este problema. A questão da expressividade de um sistema de produção unido a um modelo de objetos também não está bem definida semanticamente, principalmente em relação à sua não-monotonicidade, o que também merece um estudo mais aprofundado.

Este trabalho também pode servir de base para outros trabalhos importantes para a área, como a criação de um motor de inferência que suporte outras lógicas, bem como o estudo do impacto que esta escolha teria na eficiência do mesmo. Uma outra idéia interessante seria a criação de uma versão reduzida de JEOPS, para que possa ser utilizada em dispositivos que suportem a versão “leve” de Java (J2ME [77]).

A área de EOOPS, apesar de promissora, ainda não atingiu a maturidade de outras áreas da computação. A falta de uma metodologia específica para o desenvolvimento de projetos que integram regras e objetos aparece como um entrave para a sua expansão, e os grandes projetos desta área ainda são fetos de maneira “artesanal”. Um bom começo na direção da criação de uma metodologia para a área é a criação de um catálogo de *padrões de projeto* (ou *design patterns* [46]) relativos à área, tal como foi feito no campo da Engenharia de Software. Com uma metodologia específica, ou mesmo um conjunto de padrões de problemas catalogados, o planejamento do desenvolvimento de um sistema EOOPS poderá ser mais bem feito, tendo portanto maiores chances de sucesso.

7 REFERÊNCIAS BIBLIOGRÁFICAS

- [1] Epstein, J. (1998) *Games: The Window into Technology's Future*. Palestra apresentada no congresso Infonordeste'98. Recife.
- [2] Jennings, N., Sycara, K., Woolridge, M. (1998) A Roadmap of Agent Research and Development. *In Autonomous Agents and Multi-Agent Systems*, 1, pp. 275-306. Boston: Kluiver Academic Publishers.
- [3] Russel, S. & Norvig, P. (1997) *Artificial Intelligence: a Modern Approach*. Englewood Cliffs, NJ: Prentice Hall.
- [4] Franklin, S. & Graesser, A. (1996) Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. *In Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages (ATAL'96)*, Springer-Verlag.
- [5] Coen, M. (2000) *SodaBot: A Brief Introduction*. [online] Disponível via WWW em <http://www.ai.mit.edu/people/sodabot/slideshow/total/P001.html>. Consultado em 2 de agosto de 2000.
- [6] Shoham, Y. (1993) Agent-Oriented Programming, *In Artificial Intelligence*, 60(1):51-92.
- [7] Fisher, M. (1994) A survey of Concurrent METATEM – the language and its applications. *In Temporal Logic – Proceedings of the First International Conference (LNCS 827)*, pp. 480-505. Berlim, Alemanha: Springer-Verlag.
- [8] Thomas, S. (1995) The PLACA Agent Programming Language. *In Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures and Languages: Intelligent Agents*, pp. 355-370. Berlim, Alemanha: Springer-Verlag.
- [9] Schneider, D. & Martin-Michellot, S. (2000) *VRML Primer and Tutorial*. [online] Disponível via WWW em <http://tecfa.unige.ch/guides/vrml/vrmlman-test/>. Consultado em 3 de agosto de 2000.
- [10] Murry, W. & Pappas, C. (1991) *Turbo C++ Completo e Total*. Makron Books.
- [11] Zoku.net (2000) *WWW.SMALLTALK.ORG™ Main Page*. [online] Disponível via WWW em <http://www.smalltalk.org>. Consultado em 9 de agosto de 2000.
- [12] Sun Microsystems (2000) *The Source for Java™ Technology*. [online] Disponível via WWW em <http://java.sun.com>. Consultado em 16 de agosto de 2000.

- [13] Watterman, D. & Hayes-Roth, F. (1978) *Pattern-Directed Inference Systems*. Orlando: Academic Press,.
- [14] Finin, T. (2000) *UMBC KQML Web*. [online] Disponível via WWW em <http://www.cs.umbc.edu/kqml>. Consultado em 15 de setembro de 2000.
- [15] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. (1991) *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice Hall.
- [16] Booch, G., Rumbaugh, J., Jacobson, I. (1998) *The Unified Modeling Language User Guide*. Reading: Addison-Wesley.
- [17] Bryson, J. & McGonigle, B. (1998) Agent Architecture as Object Oriented Design. In *Proceedings of the 4th International Workshop on Agent Theories, Applications and Languages (ATAL'97)*. Berlim, Alemanha: Springer-Verlag.
- [18] Woolridge, M., Jennings, N., Kinny, D. (1999) A Methodology for Agent-Oriented Analysis and Design. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pp 69-76, Seattle: ACM Press.
- [19] *Communications of the ACM*. Special issue on Multiagent Systems on the Net and Agents in E-commerce. Vol 42, n.3. 1999.
- [20] Pachet, F. (1995) On the Embeddability of Production Rules in Object-Oriented Languages. In *Journal of Object-Oriented Programming*, vol. 8, n. 4.
- [21] Giarratano, J. & Riley, G. (1994) *Expert Systems: Principles and Programming*, 2nd Edition. Boston: PWS Publishing Company.
- [22] Giarratano, J. (1998) *CLIPS User's Guide*, Version 6.10.
- [23] Lopez F. (2000) NASA CLIPS RULE-BASED LANGUAGE: [online] Disponível via WWW em <http://www.siliconvalleyone.com/clips.htm>, Consultado em 10 de agosto de 2000.
- [24] Forgy, C (1994) RAL/C and RAL/C++: Rule-Based Extensions to C and C++. In *Proceedings of the OOPSLA'94 workshop on Embedded Object-Oriented Production Systems (EOOPS)*. Paris: Laforia.
- [25] Atkinson, R. & Laursen, J. (1987) Opus: A Smalltalk Production System. In *Proceesings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 377-387.

- [26] Pachet, F. (1992) *Représentation de connaissances par objets et règles: le système NéOpus*. Tese de Doutorado, Université Paris VI. Paris: Laforia.
- [27] Friedman-Hill, E (2000) *JESS, the Java Expert System Shell*. [online] Disponível via WWW em <http://herzberg.ca.sandia.gov/jess>. Consultado em 18 de agosto de 2000.
- [28] Figueira Filho, C. & Ramalho, G. (2000) JEOPS – The Embedded Object Production System. [to be published] *In Advances in Artificial Intelligence*. Monard, M.C. & Sichman, J. Eds. Springer-Verlag, Lecture Notes in Artificial Intelligence, v. 1952, pp. 52-61.
- [29] Figueira Filho, C. (2000) *JEOPS – The Java Embedded Object Production System*. [online] Disponível via WWW em <http://www.cin.ufpe.br/~csff/jeops>. Consultado em 20 de setembro de 2000.
- [30] Sommers, B. (2000) *Agents: Not just for Bond anymore*. [online] Disponível via WWW em <http://www.javaworld.com/jw-04-1997/jw-04-agents.html>. Consultado em 11 de abril de 2000.
- [31] Silva, D., Siebra, C., Valadares, J., Almeida, A., Frery, A., Da Rocha Falcão, J. & Ramalho, G. (2001). Synthetic Actor Model for Long-Term Computer Games. Virtual Reality. Springer-Verlag. (forthcoming)
- [32] Macedo, H., Araújo, A., Cavalcanti, D., Andrade, R., Madeira, C. & Ferraz, C. (2000) Evaluating Multi User Distributed Action Games Architectures on a CORBA Platform. *In Proceedings of The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, USA.
- [33] Schor, M. (1986) Declarative Knowledge Programming: Better Than Procedural?. *In IEEE Expert, Spring*, pp. 36-43.
- [34] Riley, G. (2000) *What Are Expert Systems?*. [online] Disponível via WWW em <http://www.ghgcorp.com/clips/ExpertSystems.html>. Consultado em 16 de agosto de 2000.
- [35] Howe, D. (2000) *Production system from FOLDOC*. [online] Disponível via WWW em <http://www.nue.org/foldoc/foldoc.cgi?query=production+system>. Consultado em 17 de agosto de 2000.
- [36] Jackson, P. (1999) *Introduction to Expert Systems*. 3nd Edition. London: Addison-Wesley Pub.

- [37] Sommerville, I. (1996) *Software Engineering*. 5th Edition. Reading, MA: Addison-Wesley.
- [38] Booch, G. (1994) *Object-Oriented Analysis and Design*. 2nd Edition. Redwood City, CA: Benjamin Cummings.
- [39] Vialatte, M. (1985) *Description et implémentation du moteur d'inférence Snark*. Tese de Doutorado, Université Paris VI. Paris: Laforia.
- [40] Brownston, L. (1985) *Programming Expert Systems in OPS5*. Reading: Addison-Wesley.
- [41] Forgy, C. (1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *In Artificial Intelligence*, v. 19, pp. 17-37.
- [42] Bratko, I. (1990) *PROLOG Programming for Artificial Intelligence*. 2nd Edition, Addison-Wesley Pub Co.
- [43] Wielinga, B., Schreiber, A., Breuker, J. (1992) KADS: A Modelling Approach to Knowledge Engineering. *In Knowledge Acquisition*, v. 4, pp. 5-53.
- [44] Lindholm, T. & Yellin, F. (1999) *The Java™ Virtual Machine Specification*. 2nd Edition. Reading, MA: Addison-Wesley.
- [45] IBM (2000): *VisualAge for Java*. [online] Disponível via WWW em <http://www.software.ibm.com/vajava>. Consultado em 20 de setembro de 2000.
- [46] Gamma, E. et al (1996) *Design Patterns: Elements of Reusable Object-oriented Software*. Reading: Addison Wesley.
- [47] Bouaud, J. & Voyer, R. (1994) Behavioral Match: Embedding Production Systems and Objects. *In Proceedings of the OOPSLA'94 workshop on Embedded Object-Oriented Production Systems (EOOPS)*. Paris: Laforia.
- [48] The Haley Enterprise (2000) *Expert System Languages*. [online] Disponível via WWW em <http://www.haley.com/framed/RuleLanguages.html>. Consultado em 21 de agosto de 2000.
- [49] Production Systems Technologies (2000) *Rete and Rete II*. [online] Disponível via WWW em <http://www.pst.com/rete.htm>. Consultado em 18 de agosto de 2000.
- [50] Forgy, C. (2000) *Benchmarking CLIPS/R2*. [online] Disponível via WWW em <http://www.pst.com/bencr2.htm>. Consultado em 20 de setembro de 2000.

- [51] Friedman-Hill, E. (2000) *The JESS Mailing List*. [online] Disponível via WWW em http://herzberg.ca.sandia.gov/jess/mailling_list.html. Consultado em 20 de setembro de 2000.
- [52] Sun Microsystems (2000) *Bound Properties*. [online] Disponível via WWW em <http://java.sun.com/docs/books/tutorial/javabeans/properties/bound.html>. Consultado em 18 de agosto de 2000.
- [53] Sun Microsystems (2000) *JavaBeans™*. [online] Disponível via WWW em <http://java.sun.com/products/javabeans>. Consultado em 18 de agosto de 2000.
- [54] Sun Microsystems (2000) *Java Compile and Runtime Environments*. [online] Disponível via WWW em <http://java.sun.com/docs/white/platform/javaplatform.doc3.html>. Consultado em 18 de agosto de 2000.
- [55] Masini, G., Napoli, A., Colnet, D. (1991) *Object-Oriented Languages*. Londres: Academic Press.
- [56] Ait-Kaci, H., Dumant, B., Meyer, R., Podelski, A., Van Roy, P. (1994) *The Wild LIFE Handbook*. Technical Report. Paris: Digital Research Laboratory.
- [57] Sun Microsystems (2000): *Java Native Interface*. [online] Disponível via WWW em <http://java.sun.com/docs/books/tutorial/native1.1/index.html>. Consultado em 18 de agosto de 2000.
- [58] Banbara, M. & Tamura, N. (1999) Translating a linear logic programming language into Java. In *Proceedings of the ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pp. 19-39.
- [59] Amandi, A., Iturregui, R., Zunino, A. (1998) Object-Agent Oriented Programming. In *Proceedings of the Argentinian Symposium on Object Orientation*. Buenos Aires, Argentina: SADIO.
- [60] Almgren, J., Bowen, D., Byrd, L., Pereira, F., Pereira, L., Warren, D. (2000) *Mixing Java and Prolog*. [online] Disponível via WWW em http://cswww.essex.ac.uk/TechnicalGroup/sicstus/sicstus_12.html. Consultado em 18 de agosto de 2000.
- [61] Almgren, J., Bowen, D., Byrd, L., Pereira, F., Pereira, L., Warren, D. (2000) *SICStus Prolog User's Manual*. [online] Disponível via WWW em

- <http://www.sics.se/isl/sicstus/docs/latest/pdf/sicstus.pdf>. Consultado em 21 de agosto de 2000.
- [62] Calejo, M. (2000) *Java plus Prolog systems*. [online] Disponível via WWW em <http://www.declarativa.com/InterProlog/systems.htm>. Consultado em 21 de agosto de 2000.
- [63] Ramalho, G. & Robin, J. (2000) *Inteligência Artificial Simbólica*. [online] Disponível via WWW em <http://www.di.ufpe.br/~compint/ias.html>. Consultado em 21 de agosto de 2000.
- [64] Metamata (2000) *JavaCC – the Java Parser Generator*. [online] Disponível via WWW em <http://www.metamata.com/javacc>. Consultado em 13 de setembro de 2000.
- [65] Mhatta (2000): *GNU Lesser General Public License*. [online] Disponível via WWW em <http://www.gnu.org/copyleft/lesser.html>. Consultado em 18 de setembro de 2000.
- [66] Neel (2000): *GNU's Not Unix – the GNU Project and the Free Software Foundation (FSF)*. [online] Disponível via WWW em <http://www.fsf.org/>. Consultado em 18 de setembro de 2000.
- [67] MicroProse (2000): *Sid Meier's Civilization II*. [online] Disponível via WWW em <http://www.microprose.com/gamesdesign/civ2/civ2.html>. Consultado em 25 de setembro de 2000.
- [68] Levine, J., Mason, T., Brown, D. (1992) *Lex and Yacc*. 2nd Edition. Sebastopol, California: O'Reilly and Associates.
- [69] Sun Microsystems (2000) *What Is an Interface?*. [online] Disponível via WWW em <http://java.sun.com/docs/books/tutorial/java/interpack/interfaceDef.html>. Consultado em 21 de setembro de 2000.
- [70] Sun Microsystems (2000) *Event Handling*. [online] Disponível via WWW em <http://java.sun.com/docs/books/tutorial/uiswing/overview/event.html>. Consultado em 11 de setembro de 2000.
- [71] Figueira Filho, C. (2000) *JEOPS API Documentation*. [online] Disponível via WWW em <http://www.cin.ufpe.br/~csff/jeops/api>. Consultado em 18 de setembro de 2000.
- [72] Sun Microsystems (2000): *Inner Classes*. [online] Disponível via WWW em <http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html>. Consultado em 22 de setembro de 2000.

- [73] Bates, J. (1991) Virtual Reality, Art and Entertainment. *In The Journal of Teleoperators and Virtual Environment*. Boston: MIT Press.
- [74] Barros, F. & Figueira Filho, C. (2000) *Um Gerador de Palavras Homófonas*. [online] Disponível via WWW em <http://www.cin.ufpe.br/~csff/fonetica>. Consultado em 23 de setembro de 2000.
- [75] Manber, U. (1989) *Introduction to Algorithms: A Creative Approach*. Addison-Wesley.
- [76] The Robocup Federation (2000) *RoboCup Oficial Site*. [online] Disponível via WWW em <http://www.robocup.org>. Consultado em 24 de setembro de 2000.
- [77] Sun Microsystems (2000): *Java 2 Platform, Micro Edition*. [online] Disponível via WWW em <http://java.sun.com/j2me/index.html>. Consultado em 10 de outubro de 2000.
- [78] Sun Microsystems (2000): *jar – The Java Archive Tool*. [online] Disponível via WWW em <http://java.sun.com/j2se/1.3/docs/tooldocs/win32/jar.html>. Consultado em 18 de setembro de 2000.

Anexo A – NOTAÇÃO UTILIZADA (UML)

A notação utilizada na modelagem das classes do sistema, bem como na definição das classes no corpo deste trabalho, é a definida pelo modelo UML (*Unified Modeling Language* [16]). Exibiremos uma pequena legenda dos diagramas de classes, para facilitar a compreensão dos modelos apresentados neste trabalho.

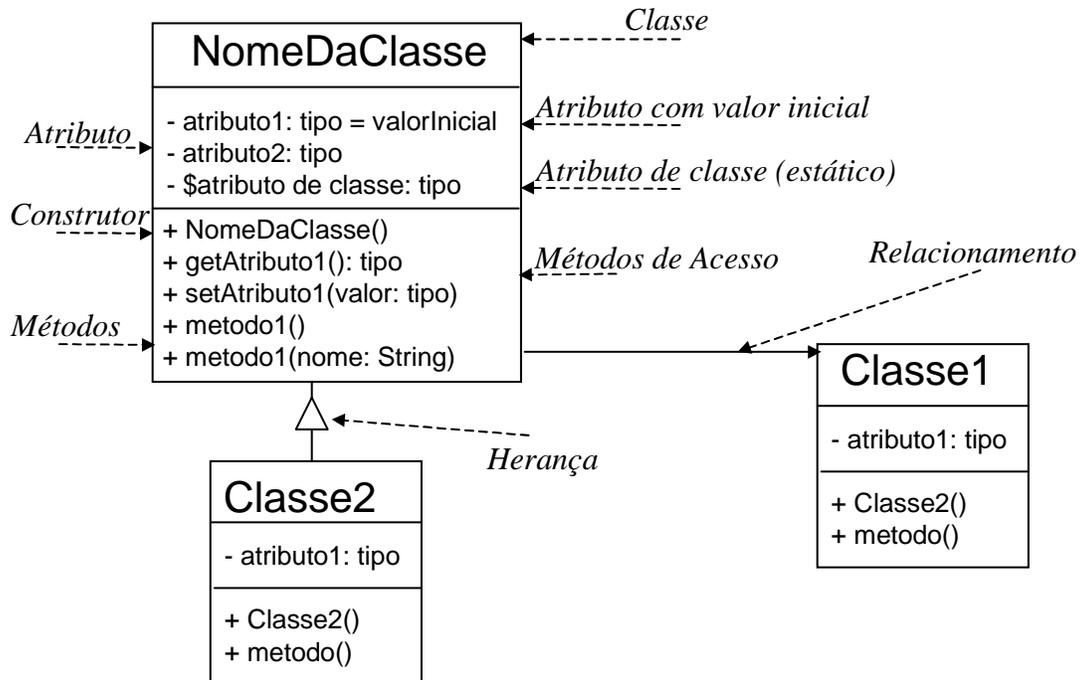


Figura 32 – Notação (simplificada) UML de definição de classes

A notação completa usada na definição de diagramas de classe de UML, bem como a utilizada na definição dos outros tipos de diagramas podem ser encontrados em [16].

Anexo B – BNF DAS REGRAS JEOPS

A sintaxe da bases de regras, em uma estrutura BNF simplificada é apresentada a seguir. Assumimos que o leitor está familiarizado com a estrutura de Java, de modo que não estendemos alguns termos como <expressão booleana> ou <comando>, para manter esta descrição mais concisa.

```

Base de Regra ::= ("package" <ident> ( "." <ident>)* ";" )?
                ("import" <ident> ( "." <ident>)* ( "."* )? ";" )*
                "ruleBase" <ident> "{" <Corpo da Base> "}"
Corpo da Base ::= (
                    <Regra>
                    | <declaração de método>
                    | <declaração de atributo>
                    | <declaração de inner class>
                )*
Regra ::= "rule" <ident> "(" <Corpo da Regra> ")"
Corpo da Regra ::= <Declarações>
                  (<Declarações Locais>)?
                  <Condições>
                  <Ações>
Declarações ::= "declarations"
                (<Nome de classe> <ident> ( "," <ident>)* ";" )*
Declarações Locais ::= "localdecl"
                    (<Nome de classe> <ident> "=" <expressão> ";" )*
Condições ::= "conditions" (<expressão booleana>)*
Ações ::= "actions" (<Ação>)+
Ação ::= "assert" "(" <expressão> ")"
        | "retract" "(" <expressão> ")"
        | "modify" "(" <expressão> ")"
        | <comando>
<Nome de classe> ::= <ident> ( "." <ident>)*
<ident> ::= <Identificador de Java>
<declaração de método> ::= <Declaração de método de Java>
<declaração de atributo> ::= <Declaração de atributo de Java>
<expressão booleana> ::= <Expressão de Java cujo valor é true ou false>
<expressão> ::= <Expressão de Java>
<comando> ::= <Comando de Java>

```

Anexo C – ESTUDOS DE CASO

Apresentaremos nesta seção uma série de estudos de caso que realizamos para validar o JEOPS. Decidimos colocá-los nos apêndices para podermos apresentá-los na íntegra, de modo a fazer com que esta seção seja também uma fonte de referência para os futuros usuários do sistema.

C.1 SÉRIE DE FIBONACCI

Trata-se do problema de encontrar um elemento da série de Fibonacci, onde o valor de um elemento é igual à soma de seus dois antecessores. A Figura 33 ilustra os 10 primeiros elementos da série de Fibonacci.

Fib(0) = 0	Fib(1) = 1
Fib(2) = Fib(0) + Fib(1) = 1	Fib(3) = Fib(1) + Fib(2) = 2
Fib(4) = Fib(2) + Fib(3) = 3	Fib(5) = Fib(3) + Fib(4) = 5
Fib(6) = Fib(4) + Fib(5) = 8	Fib(7) = Fib(5) + Fib(6) = 13
Fib(8) = Fib(6) + Fib(7) = 21	Fib(9) = Fib(7) + Fib(8) = 34

Figura 33 – Série de Fibonacci

A idéia da resolução deste problema através de regras é a de representar nos próprios objetos o mecanismo de recursão. Um objeto da classe Fibonacci possui, além dos atributos que representam seu valor, dois subproblemas relacionados, definido conforme a Figura 34.

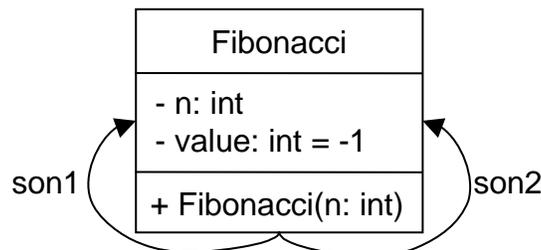


Figura 34 – Classe `Fibonacci`

Para cada cálculo de Fibonacci, são criados dois filhos, que representam os efetuados normalmente pelas chamadas recursivas. Três regras permitem então que se represente este cálculo: uma regra de parada (`BaseCase`), uma regra de geração de filhos

(GoDown) e uma regra de retorno, que calcula o valor quando os seus dois filhos estiverem resolvidos (GoUp). Estas regras são apresentadas na Tabela 42.

<pre>rule BaseCase { declarations Fibonacci f; conditions f.getN() <= 1; f.getValue() == -1; actions f.setValue(f.getN()); modified(f); }</pre>	<pre>rule GoDown { declarations Fibonacci f; conditions f.getN() > 1; f.getValue() == -1; f.getSon1() == null; actions int n = f.getN(); Fibonacci s1 = new Fibonacci(n - 1); Fibonacci s2 = new Fibonacci(n - 2); f.setSon1(s1); f.setSon2(s2); assert(s1); assert(s2); modified(f); }</pre>	<pre>rule GoUp { declarations Fibonacci f, f1, f2; conditions f.getN() > 1; f.getSon1() == f1; f.getSon2() == f2; f1.getValue() != -1; f2.getValue() != -1; actions int v = f1.getValue() + f2.getValue(); f.setValue(v); modified(f); }</pre>
--	--	---

Tabela 42 – Regras do problema da série de Fibonacci

A utilização do sistema se dá criando uma base de regras e inserindo nela uma instância da classe `Fibonacci`. A Tabela 43 ilustra a utilização da base de regras criada (assumindo que o nome do arquivo de regras é `FibonacciBase.rules`).

```
public static void main(String[] args) {
  FibonacciBase base = new FibonacciBase();
  Fibonacci f = new Fibonacci(15);
  base.assert(f);
  base.run();
  System.out.println("Fib(" + f.getN(); + ") = " + f.getValue());
}
```

Tabela 43 – Utilização da base de Fibonacci

A solução recursiva para se calcular o valor de **Fib(n)** cresce exponencialmente com o aumento do valor de **n** [73]. Por este motivo, este problema é frequentemente utilizado como *benchmark* para a avaliação de motores de inferência (Cf. seção 5.2.2). Uma solução iterativa para o problema (muito mais eficiente) pode ser encontrada na maioria dos livros de introdução à programação.

C.2 AS 8 RAINHAS

O problema das 8 rainhas é mais um problema clássico que pode ser resolvido de forma elegante através de JEOPS¹. O problema consiste em se colocar em um tabuleiro de

¹ Este problema é resolvido pelo JEOPS de forma elegante, sem dúvida. No entanto, uma percorrer todo o espaço de busca para encontrar a solução para este problema não é a forma mais eficiente; uma solução por

xadrez 8 rainhas, de modo que não haja um par delas que possa se atacar, de acordo com as regras do xadrez. A Figura 35 ilustra uma solução possível para o problema.

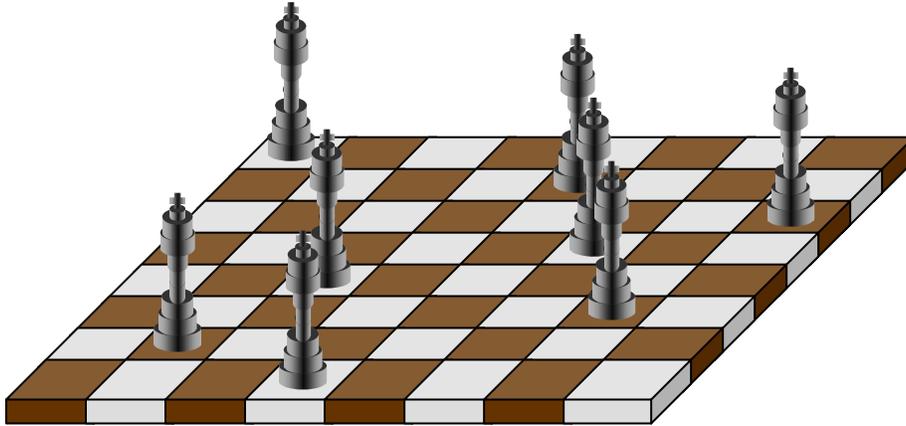


Figura 35 – O problema das oito rainhas

Este é um típico problema de busca por um espaço limitado. Precisamos encontrar, entre todas as possíveis combinações de rainhas no tabuleiro de xadrez, uma com 8 elementos que satisfaça as propriedades descritas acima.

Solucionar este problema com um motor de inferência como o JEOPS pode ser muito mais natural que, digamos, através de uma função recursiva em uma linguagem imperativa. É necessário informar ao motor apenas as condições que devem ser respeitadas (i.e., duas rainhas não podem estar em posição de ataque), e o mesmo se encarregará de encontrar

A classe `Queen`, definida na Figura 36, contém conhecimento apenas para determinar se duas rainhas se atacam. A busca pela solução será feita pela base de conhecimentos.

meio de uma estratégia de CSP (*constraint satisfaction problem*) é mais indicada. O exemplo é válido, contudo, em ilustrar a resolução de um problema tipicamente de busca através de JEOPS, e a eficiência da solução deixa muito pouco a desejar em relação à solução ótima.

Queen
- row: int - column: int
+ Queen(row: int, column: int) + attacks(q: Queen)

Figura 36 – Classe Queen

É necessária apenas uma regra para a resolução deste problema: uma que encontre 8 rainhas, que não se ataquem duas a duas. Com isso, definimos a base de regras apresentada na Tabela 44.

```
ruleBase EightQueens {
  private int solutionNumber = 0;
  rule findSolution {
    declarations
      Queen q1, q2, q3, q4, q5, q6, q7, q8;
    conditions
      // Para evitar permutações da mesma solução
      q1.getRow() == 1; q2.getRow() == 2; q3.getRow() == 3;
      q4.getRow() == 4; q5.getRow() == 5; q6.getRow() == 6;
      q7.getRow() == 7; q8.getRow() == 8;
      // Restrição do problema
      !q1.attacks(q2); !q1.attacks(q3); !q1.attacks(q4);
      !q1.attacks(q5); !q1.attacks(q6); !q1.attacks(q7);
      !q1.attacks(q8);
      !q2.attacks(q3); !q2.attacks(q4); !q2.attacks(q5);
      !q2.attacks(q6); !q2.attacks(q7); !q2.attacks(q8);
      !q3.attacks(q4); !q3.attacks(q5); !q3.attacks(q6);
      !q3.attacks(q7); !q3.attacks(q8);
      !q4.attacks(q5); !q4.attacks(q6); !q4.attacks(q7);
      !q4.attacks(q8);
      !q5.attacks(q6); !q5.attacks(q7); !q5.attacks(q8);
      !q6.attacks(q7); !q6.attacks(q8);
      !q7.attacks(q8);
    actions
      Queen[] sol = new Queen[] {q1, q2, q3, q4, q5, q6, q7, q8};
      System.out.println("Solution " + (++solutionNumber) + ":");
      for (int i = 0; i < 8; i++) {
        System.out.println("Queen " + (i+1) + ": (" +
          sol[i].getRow() + "," + sol[i].getColumn() + ")");
      }
  }
}
```

Tabela 44 – Base de regra para o problema das 8 rainhas

Devemos notar que esta solução enumera todas as soluções do problema, utilizando um artifício de definir uma variável na própria base de regras para “contar” o número de soluções já apresentadas. Merece menção também o fato de que a solução de um problema

não precisa ser “armazenada” em um agente: neste exemplo, ela é simplesmente impressa na saída padrão.

Para executar esta base de conhecimentos, é necessário se inserir as 64 rainhas na base de conhecimentos, e mandar executá-la. Como existe apenas uma regra na base de regras, qualquer estratégia de resolução de conflitos pode ser utilizada. A Tabela 45 apresenta uma classe que ativa a base de conhecimentos gerada a partir da pré-compilação da base de regras apresentada na Tabela 44.

```
public class TestQueens {
    public static void main(String[] args) {
        EightQueens kb = new EightQueens();
        for (int i = 1; i <= 8; i++) {
            for (int j = 1; j <= 8; j++) {
                kb.assert(new Queen(i, j));
            }
        }
        kb.run();
    }
}
```

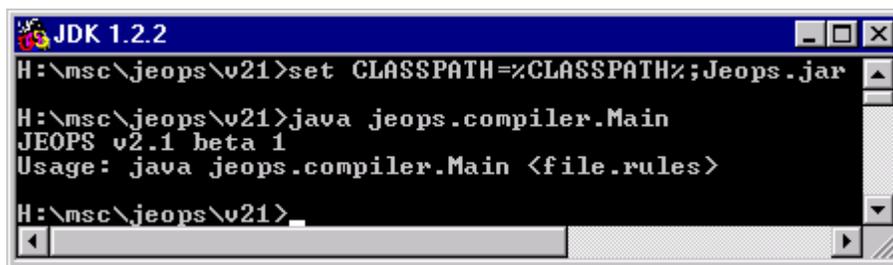
Tabela 45 – Utilização da base de conhecimento das 8 rainhas

Anexo D – A DISTRIBUIÇÃO DO SISTEMA

JEOPS pode ser baixado de sua página principal em duas versões: o código fonte e as classes compiladas. No pacote do código fonte encontram-se todas as classes utilizadas na implementação do sistema, bem como bases de regras dos exemplos e os arquivos com os termos da licença.

O pacote com as classes compiladas foi estruturado de tal forma a facilitar a sua utilização por parte do usuário. Da mesma maneira que o kit de desenvolvimento de Java distribuído pela Sun, JEOPS vem em duas versões: a utilizada no desenvolvimento do sistema, que contém as classes que implementam o compilador de bases de regras em classes Java, além das classes necessárias para a execução do sistema; e uma versão mais leve, que não contém o compilador, utilizada para a distribuição das aplicações que utilizam o JEOPS. A disponibilização de uma versão “light” é especialmente interessante em aplicações distribuídas pela Internet (e.g., applets), pois o tempo necessário para o carregamento das classes antes do início da aplicação é diversas vezes muito importante.

As duas versões citadas no parágrafo anterior se encontram agrupadas cada uma em um único arquivo JAR (acrônimo de Java ARchive, padrão da linguagem para a distribuição de componentes [78]). A versão completa encontra-se agrupada no arquivo `Jeops.jar`, e basta que o arquivo seja adicionado à variável de ambiente `CLASSPATH` para que o compilador JEOPS possa ser utilizado, como mostra a Figura 37.



```
JDK 1.2.2
H:\msc\jeops\v21>set CLASSPATH=%CLASSPATH%;Jeops.jar
H:\msc\jeops\v21>java jeops.compiler.Main
JEOPS v2.1 beta 1
Usage: java jeops.compiler.Main <file.rules>
H:\msc\jeops\v21>_
```

Figura 37 – Preparação do ambiente de desenvolvimento JEOPS

Uma vez finalizada, a aplicação que utiliza o JEOPS não precisa mais do compilador para funcionar (salvo em aplicações que implementem algum mecanismo de compilação e carregamento dinâmico de classes, o que não é o caso da maioria das aplicações). Para minimizar o tamanho total da aplicação, esta pode ser distribuída apenas com a versão *light* de JEOPS, agrupada no arquivo `JeopsRE.jar` (de *JEOPS Runtime*

Environment). Por exemplo, a chamada a um applet que utiliza o JEOPS pode ser feita com o tag `<applet>`, conforme apresentado na Tabela 46.

```
<applet code="homofonos.gui.MainFrame"  
        width=400  
        height=300  
        archive="JeopsRE.jar"  
>  
Infelizmente o seu browser não suporta applets.  
</applet>
```

Tabela 46 – Utilização de JEOPS em applets

Instruções mais detalhadas sobre a instalação do sistema podem ser encontradas no arquivo de distribuição de JEOPS, disponível a partir de sua home page [29].

Anexo E – MANUAL DO USUÁRIO

Este documento está disponível na Internet através da seguinte URL:

<http://www.di.ufpe.br/~csff/jeops/manual/v21/>

Anexo F – CONVERSÃO PROLOG → JAVA

Este é um exemplo de conversão de uma série de predicados Prolog para uma classe Java. Trata-se de uma série de regras para determinar se uma pessoa é avô (ou avó) de outra. O seguinte programa Prolog

Programa familia.pl
avo(X,Z) :- pai(X,Y), pai(Y,Z).
avo(X,Z) :- pai(X,Y), mae(Y,Z).
avo(X,Z) :- mae(X,Y), pai(Y,Z).
avo(X,Z) :- mae(X,Y), mae(Y,Z).
mae(marta, joao).
mae(maria, marta).
pai(jose, joao).
pai(joao, marcos).
mae(sandra, jose).
pai(pedro, jose).

Poderia ser traduzido para Java como a seguinte classe:

Classe Familia.java
<pre> public class Familia { private static Set fatos = new HashSet(); static { fatos.add(new Fato("mae", "marta", "joao")); fatos.add(new Fato("mae", "maria", "marta")); fatos.add(new Fato("pai", "jose", "joao")); . . . } public static boolean avo(String x, String y) { return (Avo_1(x,y) Avo_2(x,y) Avo_3(x,y) Avo_4(x,y)); } private static boolean Avo_1(String x, String y) { for (Iterator i = fatos.iterator(); i.hasNext();) { Fato f = (Fato) i.next(); if (f.getPredicateName().equals("Pai") && f.getParam(1).equals(x) { Fato aux = new Fato("Pai", f.getParam(2), y); If (fatos.contains(aux)) return true; } } return false; } . . . } </pre>

Anexo G – CONVERSÃO DE BASE DE REGRAS

Este é um exemplo da pré-compilação de uma base de regras JEOPS em uma série de métodos que serão utilizados pelo motor de inferência tanto para verificar se suas declarações podem ser unificadas com objetos da memória de trabalho, como para ativá-la. Trata-se de uma das regras utilizadas na resolução do problema da busca por ancestrais, apresentada na Tabela 47.

```

package jeops.examples.familia;
ruleBase Familia {
  rule encontraAncestrais {
    declarations
      Pessoa p;
      Objetivo o;
    localdecl
      Pessoa pai = p.getPai();
      Pessoa mae = p.getMae();
    preconditions
      p == o.getAlvo();
    actions
      retract (o);
      System.out.println(pai.getNome() + " e " + mae.getNome() +
        " são ancestrais.");
      assert(new Objetivo(pai));
      assert(new Objetivo(mae));
  }
}

```

Tabela 47 – Base de regras Família

Após a pré-compilação, são geradas as classes `Familia` e `Jeops_RuleBase_Familia`, cujos métodos podem ser divididos em diversas categorias. Estas categorias são apresentados nas subseções seguintes.

G.1 INFORMAÇÕES GENÉRICAS SOBRE A BASE DE REGRAS

A Tabela 48 ilustra estes métodos, a partir dos quais pode-se saber quantas regras tem a base, seus nomes, quantas declarações e condições tem cada regra. Deve-se observar que os *arrays* retornados pelos métodos são criados apenas uma vez, para evitar os problemas de performance da linguagem Java quando da criação de objetos.

```

public int getNumberOfRules() {
  return 1;
}

private static final String[] File_ruleNames = {
  "encontraAncestrais"
}

```

```

};
public String[] getRuleNames() {
    return File_ruleNames;
}

private static final int[] File_numberOfDeclarations = {
    2
};
public int[] getNumberOfDeclarations() {
    return File_numberOfDeclarations;
}

private static final int[] File_numberOfConditions = {
    1
};
public int[] getNumberOfConditions() {
    return File_numberOfConditions;
}

```

Tabela 48 – Informações genéricas sobre a base interna de regras

G.2 DECLARAÇÃO DE VARIÁVEIS DE INSTÂNCIA

A Tabela 49 ilustra a seção onde estão declaradas as variáveis que correspondem às declarações das regras.

```

private jeops.examples.familia.Pessoa jeops_examples_familia_Pessoa_1;
private jeops.examples.familia.Objetivo
                                     jeops_examples_familia_Objetivo_1;

```

Tabela 49 – Variáveis de instância da base interna de regras

G.3 CONSULTAS SOBRE AS DECLARAÇÕES

A Tabela 50 mostra os métodos a partir dos quais a base de conhecimentos pode saber quais os nomes e as classes das declarações das regras, para criar os nós de filtro de classe da rede Rete.

```

private String[] identifiers_encontraAncestrais = {
    "p",
    "o"
};
private String[] getDeclaredIdentifiers_encontraAncestrais() {
    return identifiers_encontraAncestrais;
}

public String[] getDeclaredIdentifiers(int ruleIndex) {
    switch (ruleIndex) {
        case 0: return getDeclaredIdentifiers_encontraAncestrais();
        default: return new String[0];
    }
}

```

```

private String getDeclaredClassName_encontraAncestrais(int index) {
    switch (index) {
        case 0: return "jeops.examples.familia.Pessoa";
        case 1: return "jeops.examples.familia.Objetivo";
        default: return null;
    }
}

public String getDeclaredClassName(int ruleIndex, int declIndex) {
    switch (ruleIndex) {
        case 0: return getDeclaredClassName_encontraAncestrais(declIndex);
        default: return null;
    }
}

private Class getDeclaredClass_encontraAncestrais(int index) {
    switch (index) {
        case 0: return jeops.examples.familia.Pessoa.class;
        case 1: return jeops.examples.familia.Objetivo.class;
        default: return null;
    }
}

public Class getDeclaredClass(int ruleIndex, int declIndex) {
    switch (ruleIndex) {
        case 0: return getDeclaredClass_encontraAncestrais(declIndex);
        default: return null;
    }
}

```

Tabela 50 – Consultas sobre as classes das declarações da base interna de regras

G.4 ATRIBUIÇÃO DE VALORES ÀS DECLARAÇÕES DAS REGRAS

A Tabela 51 mostra os métodos a partir dos quais pode-se mapear as declarações de uma regra em valores de objetos presentes na memória de trabalho. Isto deve ser feito para se avaliar as condições de uma regra ou para dispará-la. O mapeamento pode ser feito tanto objeto por objeto, como com todos os objetos de uma só vez.

```

private void setObject_encontraAncestrais(int index, Object value) {
    switch (index) {
        case 0: this.jeops_examples_familia_Pessoa_1 =
                (jeops.examples.familia.Pessoa) value; break;
        case 1: this.jeops_examples_familia_Objetivo_1 =
                (jeops.examples.familia.Objetivo) value; break;
    }
}

public void setObject(int ruleIndex, int declIndex, Object value) {
    switch (ruleIndex) {
        case 0: setObject_encontraAncestrais(declIndex, value); break;
    }
}

```

```

private void setObjects_encontraAncestrais(Object[] objects) {
    jeops_examples_familia_Pessoa_1 =
        (jeops.examples.familia.Pessoa) objects[0];
    jeops_examples_familia_Objetivo_1 =
        (jeops.examples.familia.Objetivo) objects[1];
}

public void setObjects(int ruleIndex, Object[] objects) {
    switch (ruleIndex) {
        case 0: setObjects_encontraAncestrais(objects); break;
    }
}

```

Tabela 51 – Atribuição de valores às declarações das regras

G.5 CONSULTA DOS VALORES MAPEADOS ÀS DECLARAÇÕES DAS REGRAS

A Tabela 52 mostra os métodos a partir dos quais pode-se consultar os valores dos objetos que estão mapeados nas declarações de uma regra. Pode-se consultar tanto um objeto particular, como todos os objetos mapeados às declarações de uma regra.

```

private Object getObject_encontraAncestrais(int index) {
    switch (index) {
        case 0: return jeops_examples_familia_Pessoa_1;
        case 1: return jeops_examples_familia_Objetivo_1;
        default: return null;
    }
}

public Object getObject(int ruleIndex, int declIndex) {
    switch (ruleIndex) {
        case 0: return getObject_encontraAncestrais(declIndex);
        default: return null;
    }
}

private Object[] getObjects_encontraAncestrais() {
    return new Object[] {
        jeops_examples_familia_Pessoa_1,
        jeops_examples_familia_Objetivo_1
    };
}

public Object[] getObjects(int ruleIndex) {
    switch (ruleIndex) {
        case 0: return getObjects_encontraAncestrais();
        default: return null;
    }
}

```

Tabela 52 – Consulta dos valores mapeados às declarações das regras

G.6 AVALIAÇÃO DE CONDIÇÕES DAS REGRAS

A Tabela 53 mostra os métodos a partir dos quais pode-se saber se uma determinada condição de uma regra é válida ou não. Há métodos para a avaliação de uma condição específica, de todas as condições que dependem unicamente de uma declaração, e de todas as condições que dependem um grupo de declarações. Antes da chamada destes métodos, algum dos métodos `setObject` deve ter sido chamado para as que as declarações tenham mapeadas a si objetos da memória de trabalho.

```
private boolean encontraAncestrais_cond_0() {
    return (jeops_examples_familia_Pessoa_1 ==
            jeops_examples_familia_Objetoivo_1.getAlvo());
}

private boolean encontraAncestrais_cond(int index) {
    switch (index) {
        case 0: return encontraAncestrais_cond_0();
        default: return false;
    }
}

public boolean checkCondition(int ruleIndex, int condIndex) {
    switch (ruleIndex) {
        case 0: return encontraAncestrais_cond(condIndex);
        default: return false;
    }
}

private boolean checkConditionsOnlyOf_encontraAncestrais
                                (int declIndex) {
    switch (declIndex) {
        case 0:
            return true;
        case 1:
            return true;
        default: return false;
    }
}

public boolean checkConditionsOnlyOf(int ruleIndex,
                                    int declIndex) {
    switch (ruleIndex) {
        case 0: return checkConditionsOnlyOf_encontraAncestrais(declIndex);
        default: return false;
    }
}

private boolean checkCondForDeclaration_encontraAncestrais
                                (int declIndex) {
    switch (declIndex) {
        case 0:
            return true;
    }
}
```

```

    case 1:
        if (!encontraAncestrais_cond_0()) return false;
        return true;
    default: return false;
}
}

public boolean checkCondForDeclaration(int ruleIndex, int declIndex) {
    switch (ruleIndex) {
        case 0:
            return checkCondForDeclaration_encontraAncestrais(declIndex);
        default: return false;
    }
}
}

```

Tabela 53 – Métodos de avaliação das condições das regras

G.7 DISPARO DAS REGRAS

A Tabela 54 mostra os métodos a partir dos quais a base de conhecimento dispara uma regra com um conjunto determinado de objetos. Antes da chamada deste método, algum dos métodos `setObject` deve ter sido chamado para as que as declarações tenham mapeadas a si objetos da memória de trabalho.

```

private void encontraAncestrais() {
    retract (jeops_examples_familia_Objetivo_1);
    System.out.println(
        (jeops_examples_familia_Pessoa_1.getPai()).getNome() + " e " +
        (jeops_examples_familia_Pessoa_1.getMae()).getNome() +
        " são ancestrais via " +
        jeops_examples_familia_Pessoa_1.getNome());
    assert(new Objetivo((jeops_examples_familia_Pessoa_1.getPai())));
    assert(new Objetivo((jeops_examples_familia_Pessoa_1.getMae())));
}

protected void internalFireRule(int ruleIndex) {
    switch (ruleIndex) {
        case 0: encontraAncestrais(); break;
    }
}
}

```

Tabela 54 – Métodos para o disparo das regras

G.8 FINALIZAÇÃO DA COMPILAÇÃO

A Tabela 55 apresenta o construtor da base interna de regras, gerado pela pré-compilação da base de regras. Este construtor será invocado pela base de conhecimentos gerada.

```
public Jeops_RuleBase_Familia(jeops.AbstractKnowledgeBase knowledgeBase)
{
    super(knowledgeBase);
}
```

Tabela 55 – Construtor da base interna de regras

Finalizada a criação da base de regras, é gerada a classe correspondente à base de conhecimentos, com os dois construtores padrão (que utiliza a política padrão de resolução de conflitos, e o que utiliza uma política própria), e o método utilizado para se criar a base de regras. Esta classe é apresentada na Tabela 56.

```
public class Familia extends jeops.AbstractKnowledgeBase {

    public Familia(jeops.conflict.ConflictSet conflictSet) {
        super(conflictSet);
    }

    public Familia() {
        this(new jeops.conflict.DefaultConflictSet());
    }

    protected jeops.AbstractRuleBase createRuleBase() {
        return new Jeops_RuleBase_Familia(this);
    }
}
```

Tabela 56 – Base de conhecimentos gerada pela pré-compilação