

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

UM AMBIENTE AMIGÁVEL PARA A CONSTRUÇÃO DE
SISTEMAS ESPECIALISTAS BASEADOS EM REGRAS DE
PRODUÇÃO

JEFFERSON CRISTIAN GUBETTI

BLUMENAU
2015

2015/1-16

JEFFERSON CRISTIAN GUBETTI

**UM AMBIENTE AMIGÁVEL PARA A CONSTRUÇÃO DE
SISTEMAS ESPECIALISTAS BASEADOS EM REGRAS DE
PRODUÇÃO**

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Roberto Heinzle, Doutor - Orientador

**BLUMENAU
2015**

2015/1-16

**UM AMBIENTE AMIGÁVEL PARA A CONSTRUÇÃO DE
SISTEMAS ESPECIALISTAS BASEADOS EM REGRAS DE
PRODUÇÃO**

Por

JEFFERSON CRISTIAN GUBETTI

Trabalho de Conclusão de Curso aprovado
para obtenção dos créditos na disciplina de
Trabalho de Conclusão de Curso II pela banca
examinadora formada por:

Presidente: _____
Prof. Roberto Heinzle, Doutor – Orientador, FURB

Membro: _____
Prof. Joyce Martins, Mestre – FURB

Membro: _____
Prof. Matheus Carvalho Viana, Doutor – FURB

Blumenau, 09 de Julho de 2015

Dedico este trabalho a minha irmã Milene Karine Gubetti, pois apesar das eventuais desavenças, conseguimos manter uma relação saudável de apoio mútuo.

AGRADECIMENTOS

Ao meu orientador, Roberto Heinzle, por todo auxílio prestado e por ter acreditado na conclusão deste trabalho.

À minha irmã, Milene Karine Gubetti, pela ajuda com a edição das imagens neste trabalho e por todo apoio durante a jornada desde o meu ingresso no curso.

Aos meus pais, Cenair Antônio Gubetti e Mariza Vetorazzi Gubetti, por terem me ensinado o quão importante é se dedicar aos estudos.

À minha madrinha, Marlize Vetorazzi, pelo apoio durante toda minha vida, até mesmo quando estava distante fisicamente.

A todos os professores e colegas que contribuíram de alguma maneira durante o curso.

À equipe da Google pelas diversas ferramentas úteis para o auxílio na educação, principalmente seu mecanismo de busca.

Às vezes, você comete um erro. Você tem duas opções. Aceita o erro ou o conserta.

Wolfgang Bogdanow

RESUMO

Com os ambientes defasados utilizados para apresentar o assunto de sistemas especialistas para ingressantes da Inteligência Artificial, sentiu-se a necessidade de criar um ambiente com recursos que facilitem a construção e manutenção da base de conhecimentos de um sistema especialista. Essa monografia apresenta a especificação e implementação de um ambiente para construção de sistemas especialistas baseados em regras de produção. O usuário poderá construir uma base de conhecimentos e realizar o processo de raciocínio sobre ela. O processo de raciocínio se inicia com a base de conhecimentos sendo transformada em uma árvore de decisão que contém os caminhos até todos os objetivos e que cada nó representa uma decisão (premissa) ou uma ação (conclusão). Com a árvore de decisão criada, as listas de variáveis e objetivos instanciados são inicializadas, bem como a lista com todos os caminhos possíveis até os objetivos. Essa lista é ordenada de acordo com dois critérios: caminhos mais curtos e ordem das regras na base de conhecimentos. Então, o processo de inferência de dados se inicia com o ambiente consultando o usuário quanto aos valores de cada variável que está sendo analisada, até que um objetivo seja instanciado e o ambiente exiba a tela com os resultados do raciocínio do *reasoner*. Nos testes realizados o ambiente apresentou resultados satisfatórios, mesmo que com algumas limitações, principalmente por sempre exigir do usuário um valor quando uma consulta é exibida, ou seja, não tratando casos em que essa variável não possui valor conhecido. Além disso, é necessário complementar o ambiente com a inclusão de outras características, como modo de raciocínio de encadeamento para trás e mais opções para a resolução de conflitos.

Palavras-chave: Sistemas especialistas. Regras de produção. Motor de inferência.

ABSTRACT

With the outdated environments used to present the subject of expert systems for freshmen of Artificial Intelligence, it's felt the need to create an environment with resources that make easy the construction and maintenance of the knowledge base of an expert system. This monograph presents the specification and implementation of an environment for building expert systems based on production rules. The user can build a knowledge base and carry out the process of reasoning based on them. The reasoning process begins with the knowledge base being transformed into a decision tree that contains the paths to all objectives and each node represents a decision (premise) or action (conclusion). With the decision tree created, lists of variables and objectives instantiated are initialized, besides the list of all possible paths to the objectives. This list is ordered according to two criteria: shortest paths and order of the rules in knowledge base. Then the data inference process starts with the environment by consulting the user about the values of each variable being examined until a goal is instantiated and the environment to display the screen with the results of the reasoner reasoning. In tests the environment presented satisfactory results, albeit with some limitations, mainly for always require user a value when a query is displayed, that is, not treating cases where this variable has no known value. In addition, it is necessary to supplement the environment with the inclusion of other features such as reasoning mode of back chaining and more options for conflict resolution.

Key-words: Expert systems. Production rules. Inference engine.

LISTA DE FIGURAS

Figura 1 – Relação entre domínio do problema e domínio do conhecimento.....	16
Figura 2 – Elementos básicos de um sistema especialista	17
Figura 3 – Exemplo de árvore de decisão para ordenação de três números.....	23
Figura 4 – Uma regra da base de conhecimentos	26
Figura 5 – Diagrama de casos de uso.....	30
Figura 6 – Diagrama de classes do pacote Model.....	36
Figura 7 – Diagrama de classes do pacote Controller	38
Figura 8 – Diagrama de atividades de uso comum do ambiente.....	40
Figura 9 – Tela principal do ambiente	48
Figura 10 – Tela de variáveis cadastradas.....	49
Figura 11 – Tela de cadastro de variável	49
Figura 12 – Tela de cadastro de regra.....	50
Figura 13 – Tela de cadastro de premissa	50
Figura 14 – Tela de cadastro de conclusão	51
Figura 15 – Exemplo da tela de consulta	51
Figura 16 – Exemplo da tela de justificação	52
Figura 17 – Guia Objetivos da tela de resultados	52
Figura 18 – Guia Variáveis da tela de resultados	53
Figura 19 – Guia Árvore de Pesquisa da tela de resultados	53
Figura 20 – Guia Regras da tela de resultados	54
Figura 21 – Diálogo para salvar uma base de conhecimentos	54
Figura 22 – Diálogo para abrir uma base de conhecimentos	55

LISTA DE QUADROS

Quadro 1 – Exemplos de regras de produção	22
Quadro 2 – Trecho de sentenças com operador lógico ‘E’	26
Quadro 3 – Trecho de sentenças com operador lógico ‘OU’	26
Quadro 4 – Exemplo de regra para venda de produto	27
Quadro 5 – Padrão das regras de produção	28
Quadro 6 – Caso de uso UC01 - Criar regra.....	30
Quadro 7 – Caso de uso UC02 - Criar sentença.....	31
Quadro 8 – Caso de uso UC03 - Criar variável.....	32
Quadro 9 – Caso de uso UC04 - Clonar regra.....	33
Quadro 10 – Caso de uso UC05 - Criar nova base.....	33
Quadro 11 – Caso de uso UC06 - Salvar base.....	34
Quadro 12 – Caso de uso UC07 - Abrir base.....	34
Quadro 13 – Caso de uso UC08 - Executar base	35
Quadro 14 – Trecho de código utilizado para gerar o arquivo JSON.....	41
Quadro 15 – Trecho de código utilizado para recuperar um objeto do arquivo JSON gerado	42
Quadro 16 – Trecho de código do método atualizarReferencias ()	42
Quadro 17 – Trecho de código de utilização da classe AutoComplete	43
Quadro 18 – Trecho de código do método compare da classe ArrayNoSentencaComp	44
Quadro 19 – Trecho de código do método comparaItens ()	44
Quadro 20 – Trecho de código do método organizaCaminhosAteObjetivos ()	45
Quadro 21 – Trecho de código do método executar () da classe Motor.....	46
Quadro 22 – Comparação com os trabalhos correlatos.....	56

LISTA DE ABREVIATURAS E SIGLAS

IDE – *Integrated Development Environment*

JSON – *JavaScript Object Notation*

RF – Requisito Funcional

RNF – Requisito Não-Funcional

SECAJU - Sistema de Diagnóstico de Pragas e Doenças do Cajueiro

UML – *Unified Modeling Language*

UFC – Universidade Federal do Ceará

UFPE – Universidade Federal de Pernambuco

SUMÁRIO

1 INTRODUÇÃO	13
1.1 OBJETIVOS	13
1.2 ESTRUTURA	14
2 FUNDAMENTAÇÃO TEÓRICA.....	15
2.1 SISTEMAS ESPECIALISTAS	15
2.1.1 Componentes	16
2.1.1.1 Base de conhecimento	17
2.1.1.2 Motor de inferência	18
2.1.1.3 Quadro negro.....	20
2.1.1.4 Sistema de consulta e justificação	20
2.2 REPRESENTAÇÃO DO CONHECIMENTO.....	21
2.2.1 Regras de produção.....	22
2.3 ÁRVORES DE DECISÃO.....	23
2.4 TRATAMENTO DE INCERTEZA.....	24
2.5 TRABALHOS CORRELATOS	25
2.5.1 Expert SINTA.....	25
2.5.2 JEOPS	27
3 DESENVOLVIMENTO.....	29
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO	29
3.2 ESPECIFICAÇÃO.....	29
3.2.1 Diagrama de casos de uso	30
3.2.2 Diagrama de classes.....	35
3.2.3 Diagrama de atividades	40
3.3 IMPLEMENTAÇÃO	41
3.3.1 Técnicas e ferramentas utilizadas	41
3.3.1.1 Persistência da base de conhecimentos.....	41
3.3.1.2 Função para auto completar	43
3.3.1.3 Ordenação dos caminhos até os objetivos	43
3.3.1.4 Execução do <i>reasoner</i>	45
3.3.2 Operacionalidade da implementação	48
3.3.2.1 Tela principal	48

3.3.2.2 Adicionar variáveis.....	49
3.3.2.3 Adicionar regras.....	50
3.3.2.4 Tela de consulta e justificção.....	51
3.3.2.5 Tela de resultados.....	52
3.3.2.6 Salvar e recuperar base de conhecimentos.....	54
3.4 RESULTADOS E DISCUSSÕES.....	55
3.4.1 Comparativo entre os trabalhos correlatos.....	56
4 CONCLUSÕES.....	57
4.1 EXTENSÕES.....	57
REFERÊNCIAS.....	59

1 INTRODUÇÃO

Foi com base no modelo matemático de computação inventado por Emil Post em 1943, conhecido por Sistemas de Produção ou Sistemas de Post, que Alan Newell propôs um modelo de computação da inteligência humana em 1973. Sua proposta baseou-se no fato de que as regras desses Sistemas de Post, conjuntos de pares condição-ação, também chamadas de regras de produção, serem parecidas com o modo de raciocinar dos humanos. Posteriormente, esse modelo inspirou o desenvolvimento de diversos sistemas especialistas (KOWALSKI, 2011, p. 94).

Mas qual a motivação para criar sistemas especialistas?

Nas diversas áreas da vida observa-se que o ser humano está sempre necessitando do auxílio de algum especialista para que possa, de uma maneira mais acertada, tomar decisões. A indústria, medicina, educação, comércio, entre outras, são exemplos de áreas nas quais os não especialistas buscam o auxílio especializado para tomar decisões com uma maior probabilidade de acerto, e no menor tempo possível (VICTOR, 2005, p. 1).

Destaca-se, entretanto, que de modo algum o objetivo do desenvolvimento dos sistemas especialistas seja o de substituir humanos nas tomadas de decisão. Na verdade, seu propósito é dispor de uma ferramenta que auxilie o usuário a chegar à melhor solução possível conforme as circunstâncias da situação. Por vezes, esse usuário é um especialista que recém se formou ou está em processo de formação e ainda não tem experiência suficiente para tomar decisões com certa rapidez.

Diante do exposto, este trabalho apresenta um ambiente para a construção amigável de sistemas especialistas baseados em regras de produção. O ambiente disponibiliza recursos que facilitam a construção e o gerenciamento de sistemas especialistas, visando auxiliar principalmente os ingressantes nesta área do conhecimento abrangida pela Inteligência Artificial, para que, dessa forma, ele contribua cada vez mais com a popularização dos sistemas especialistas.

1.1 OBJETIVOS

O objetivo deste trabalho é disponibilizar um ambiente amigável para a construção de sistemas especialistas baseados em regras de produção.

Os objetivos específicos do trabalho são:

- a) disponibilizar um módulo com recursos que facilitem a construção e manutenção da base de conhecimentos do sistema especialista;
- b) criar um motor de inferência, ou *reasoner*, capaz de raciocinar com as regras de produção;

- c) disponibilizar uma interface que permita ao usuário acompanhar o processo de desenvolvimento de um raciocínio.

1.2 ESTRUTURA

O trabalho está dividido em quatro capítulos, sendo que o atual é responsável por apresentar a introdução, os objetivos e a estrutura do mesmo.

O segundo capítulo contém a fundamentação teórica necessária para a compreensão dos temas abordados na implementação, além da apresentação dos trabalhos correlatos ao proposto.

O terceiro capítulo, por sua vez, expõe o desenvolvimento do ambiente, elencando os requisitos e descrevendo os diagramas de casos de uso, de classe e de atividade. Ainda nesse capítulo são descritas as principais partes da implementação, a operacionalidade do ambiente e, por fim, a discussão dos resultados obtidos.

Finalmente, o quarto capítulo refere-se às considerações finais e sugestões de extensões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo contém a revisão bibliográfica dos conceitos importantes referentes ao trabalho. A Seção 2.1 apresenta uma ideia geral dos sistemas especialistas, bem como suas características e componentes. A Seção 2.2 expõe as formas de representação do conhecimento da Inteligência Artificial, suas características e detalha especialmente as regras de produção. Na Seção 2.3 é abordado o conceito das árvores de decisão e seu uso nos sistemas especialistas. A Seção 2.4 apresenta a conceituação do tratamento de incerteza e da teoria das probabilidades em sistemas especialistas. Por fim, na Seção 2.5 encontram-se dois trabalhos correlatos ao trabalho desenvolvido.

2.1 SISTEMAS ESPECIALISTAS

Os sistemas especialistas podem ser definidos como sistemas que apresentam o conhecimento que um especialista adquiriu durante seus anos de trabalho (FERNANDES, 2004, p. 20). Ainda segundo Fernandes (2003, p. 13), “são sistemas que resolvem problemas que são solucionáveis apenas por pessoas especialistas (que acumularam conhecimento exigido) na resolução desses problemas”.

Um sistema especialista deve, além de inferir conclusões, ser capaz de aprender novos conhecimentos e, dessa forma, melhorar a qualidade de suas decisões. Fernandes (2004) justifica dois modos em que sistemas especialistas são úteis:

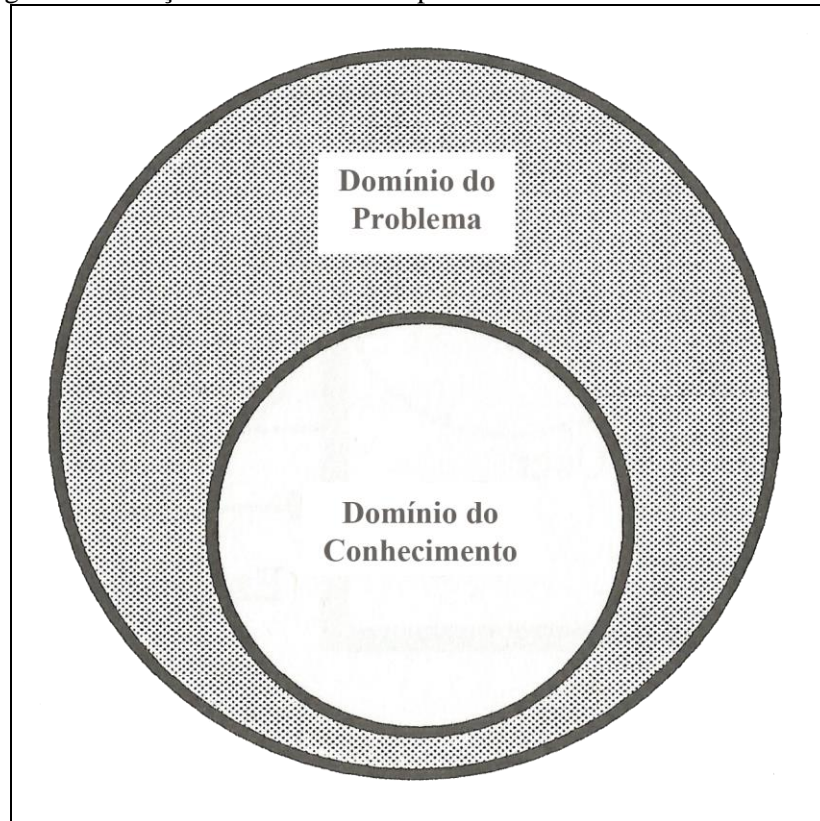
- a) no apoio à decisão: o programa ajuda o profissional experiente, responsável pela tomada de decisões, a lembrar-se de diversos tópicos ou opções que se considera por ele sabidas, mas que possam ter esquecidos ou ignorados;
- b) na tomada de decisão: o programa toma a decisão no lugar de uma pessoa, implicando que o sistema esteja acima do nível de treinamento ou experiência do usuário.

Segundo Giarratano (1994, p. 3), um especialista tem conhecimentos sobre um domínio do problema ao invés de conhecimentos sobre técnicas gerais de resolução de problemas. Esse domínio é de uma área do conhecimento em especial, como medicina, ciência ou engenharia e assim por diante, em que o especialista pode resolver problemas com maestria. Assim como um especialista humano, os sistemas especialistas são projetados para abranger um domínio de problema específico, essa resolução de problemas específicos é chamada de domínio do conhecimento. Por exemplo, um sistema especialista projetado para diagnosticar doenças infecciosas terá uma grande quantidade de informações sobre sintomas

causados por elas, consistindo do conhecimento sobre doenças, sintomas e tratamentos. Porém, apesar desse sistema ter domínio de conhecimento sobre medicina ele não teria conhecimento sobre outros ramos dela, como cirurgia ou pediatria.

A Figura 1 deixa clara a relação entre domínio do problema e domínio do conhecimento. A porção externa ao domínio do conhecimento simboliza os demais ramos da área que o especialista humano e o sistema especialista não tem conhecimento.

Figura 1 – Relação entre domínio do problema e domínio do conhecimento

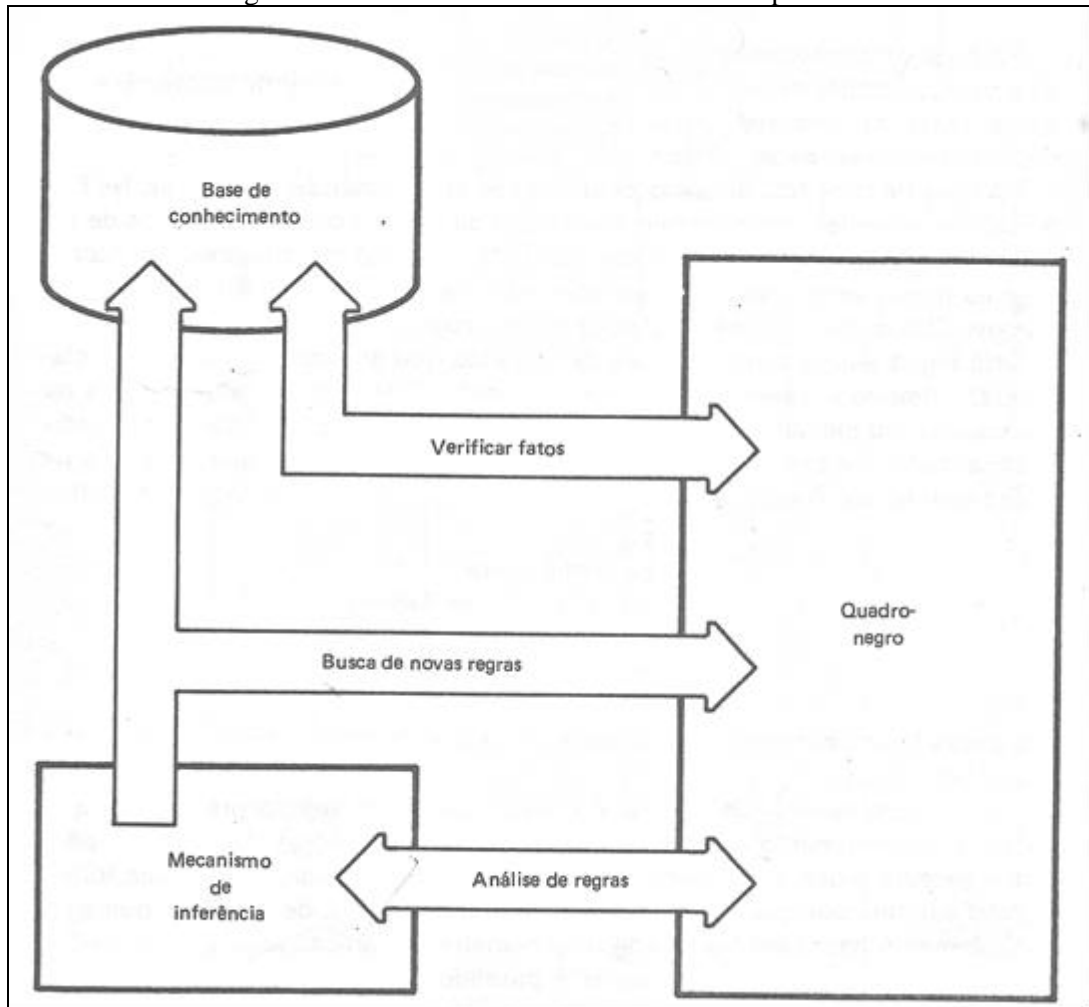


Fonte: adaptado de Giarratano (1994, p. 4).

2.1.1 Componentes

A arquitetura de um sistema especialista é definida, de uma forma mais ampla, em três componentes básicos: quadro-negro (memória de trabalho), base de conhecimento e motor de inferência (RIBEIRO, 1987, p.19). A Figura 2 mostra estes componentes e seus relacionamentos.

Figura 2 – Elementos básicos de um sistema especialista



Fonte: Ribeiro (1987, p. 19).

Além desses elementos básicos, também existem alguns componentes complementares presentes em alguns sistemas especialistas como o sistema de consulta e o sistema de justificação (HEINZLE, 1995, p. 11). Esses componentes são discutidos mais detalhadamente nas subseções a seguir.

2.1.1.1 Base de conhecimento

A base de conhecimento é definida como

[...] o local onde se armazenam fatos, heurísticas, crenças etc, ou seja, é um depósito de conhecimentos acerca de um determinado assunto. Esse conhecimento é passado ao sistema pelo especialista e armazenado de uma forma própria que permite ao sistema fazer posteriormente o processamento ou inferências (HEINZLE, 1995, p. 12).

Ela está relacionada diretamente com a parte mais complexa na construção de um sistema especialista, já que o conhecimento de um especialista não se encontra formalizado, e deve ser construída da maneira mais flexível possível para futuras atualizações. A qualidade dela define a potência dos resultados das consultas ao sistema especialista (HEINZLE, 1995,

p. 12). Isso é facilitado porque a base de conhecimentos é separada do motor de inferência tornando as adições, modificações e remoções de regras viáveis (FERNANDES, 2003, p. 17).

Segundo Ribeiro (1987, p. 22), “[...] é sempre feita para uma área bem determinada de uma ciência do conhecimento”. Essa necessidade de limitar ao máximo o domínio do conhecimento deixando apenas o suficiente para que o sistema especialista funcione como esperado, se dá pelo fato da possível complexidade do problema a ser resolvido refletir eventualmente em uma base de conhecimentos com milhares de regras, perdendo totalmente a vantagem da legibilidade dos sistemas especialistas.

2.1.1.2 Motor de inferência

Ele é considerado o núcleo do sistema especialista. É por meio do ciclo de raciocínio dele que os fatos e regras oriundos da base de conhecimentos são tratadas e aplicadas ao processo de resolução do problema. Conseqüentemente, é ele quem gerencia o comportamento dos demais componentes do sistema especialista para chegar ao objetivo (MENDES, 1997). Para Ribeiro (1987, p. 22), o motor de inferência é o responsável pela busca na base de conhecimento das regras a serem avaliadas, a ordenação dessas regras de maneira lógica e o direcionamento do processo de inferência que leva ao resultado final.

O processo de inferência é rigorosamente associado com a estrutura de armazenamento da base de conhecimentos, mas, de modo geral, existe um encadeamento lógico responsável por tirar conclusões a partir dela. Assim, novos conhecimentos são gerados. Sem o auxílio do motor de inferência, as informações armazenadas na base de conhecimentos são apenas informações estáticas (HEINZLE, 1995, p. 14).

Bittencourt (2006) elenca quatro funcionalidades características do motor de inferência:

- a) modo de raciocínio: existem dois modos de raciocínio aplicáveis a regras de produção, são eles:
 - encadeamento à frente: a parte esquerda da regra (premissa) é comparada com os dados adquiridos na memória de trabalho. As regras que satisfazem essa situação têm sua parte direita (conclusão) executada, assim novos dados são adicionados a memória de trabalho,
 - encadeamento para trás: o comportamento é dirigido por uma lista de objetivos. Um objetivo pode ser diretamente satisfeito por um dado da memória de trabalho, mas geralmente existem regras que permitem inferir algum objetivo dessa lista. Quando um objetivo não pode ser inferido por nenhuma dessas

regras, ele é descartado. O processamento termina quando o objetivo inicial é satisfeito ou quando não há mais objetivos alcançáveis;

- b) estratégia de busca: conforme Bittencurt (2006, p. 221, grifo do autor), “[...] é dita ‘cega’ se ela não leva em conta informações específicas sobre o problema a ser resolvido.” Existem duas estratégias ‘cegas’ para a busca em árvore:
- busca em largura: cada nível da árvore é visitado antes que os próximos níveis da árvore sejam visitados,
 - busca em profundidade: a prioridade de visita é dos sucessores do último nó visitado;
- c) resolução de conflito: com a finalização do processo de busca, o retorno será um conjunto de regras a serem analisadas. Se o conjunto estiver vazio a execução é encerrada, senão, é necessário escolher em qual ordem esse conjunto de regras será executado. Os métodos de resolução de conflitos mais utilizados seguem os seguintes critérios:
- prioridades atribuídas estaticamente (como a ordem das regras na base de conhecimentos),
 - características da estrutura das regras como complexidade, simplicidade e especificidade,
 - características associadas às regras como o tempo decorrido desde sua obtenção, sua confiabilidade ou grau de importância,
 - em última instância, seleção por acaso;
- d) representação de incerteza: geralmente, os domínios representados em sistemas especialistas têm necessidade de representar informações com certa porcentagem de certeza para evitar casos em que as respostas finais sejam incompletas, inexatas ou incertas. Existem diversos métodos propostos para tratar desse problema, em sua maioria, é atribuída uma medida numérica para representar a porcentagem de confiança na afirmação, além de ter um limite mínimo (proposto pelo usuário) para que a afirmação seja considerada.

A resolução de conflitos abordada neste trabalho considera o menor caminho até o objetivo (retirado da árvore de decisão), ou seja, o menor conjunto de premissas e conclusões até algum objetivo. Além disso, para decidir entre caminhos com o mesmo tamanho é utilizada a ordem da regra (na base de conhecimentos) em que o objetivo se encontra.

2.1.1.3 Quadro negro

Apesar de todos os sistemas especialistas fazerem uso do quadro negro, nem todos o destacam como componente do sistema. Ele é descrito por Heinzle (1995, p. 16) como “[...] uma área de trabalho que o sistema utiliza durante o processo de inferência. Nessa área são armazenadas informações de apoio e suporte ao funcionamento do sistema quando este está raciocinando”.

Além disso, para se chegar a uma solução

[...] há necessidade de se avaliarem regras que são recuperadas da Base de Conhecimento para uma área de trabalho na memória, nesse local as regras são ordenadas periodicamente em uma nova ordem para serem avaliadas. Durante essa avaliação deve-se verificar fatos e hipóteses e também há necessidade de uma área onde são guardados os valores das variáveis para se trabalhar tais fatos e hipóteses (RIBEIRO, 1987, p. 19).

Assim, pode-se afirmar que o quadro negro é o componente principal na ligação entre a base de conhecimentos e o motor de inferência no processo de raciocínio. Ele é responsável por manter e gerenciar as informações que são apresentadas pelo usuário durante a consulta ou inferidas pelo *reasoner* para que os demais componentes tenham acesso a essa informação no momento em que for necessário.

2.1.1.4 Sistema de consulta e justificação

A comunicação entre o usuário e o ambiente é intensa principalmente no processo de raciocínio. Nesse momento, em que o ambiente, além de apresentar informações referentes à conclusão do processo de inferência, também se faz necessário que o usuário entre com os dados necessários para a continuidade da execução do *reasoner*. Esse conjunto de atividades é considerado como sistema de consulta e em sua maioria faz uso de técnicas simples para interação com o usuário como perguntas pré-formatadas (ou informadas pelo usuário no momento da construção da base de conhecimentos) e respostas de acordo com as informações das regras na base de conhecimentos (HEINZLE, 1995, p. 15).

O sistema de justificação tem como objetivo permitir ao usuário perguntar como o ambiente chegou a determinada conclusão ou ainda por que certa informação é necessária no processo de inferência (HEINZLE, 1995, p. 16). Para responder ao porquê de uma pergunta ser feita, geralmente é apresentada a conclusão da regra que está sendo avaliada, já para a pergunta de como certa conclusão foi alcançada é exibida uma árvore de pesquisa (ordem em que as regras foram avaliadas durante o processo de inferência) no momento da apresentação dos resultados (RIBEIRO, 1987, p. 33).

Ambos os sistemas são componentes que tornam o ambiente de construção dos sistemas especialistas mais completo. Esses componentes tornam o processo de raciocínio mais natural para o usuário.

2.2 REPRESENTAÇÃO DO CONHECIMENTO

Um sistema especialista precisa de uma quantidade razoável de conhecimentos referentes ao domínio do problema a ser resolvido. É necessário que este conhecimento seja formalizado em uma estrutura de dados que facilite o gerenciamento pelo usuário e que também permita ao ambiente utilizá-lo em sua execução (HEINZLE, 1995, p. 17). Algumas características das formas de representação do conhecimento enumeradas por Fernandes (2003) são:

- a) escopo e granularidade: detalhamento nas partes do domínio de conhecimento considerado;
- b) modularidade, compreensibilidade: possibilidade de dividir o conhecimento em pequenas partes, tornando-o mais legível;
- c) conhecimento explícito e flexibilidade: toda a informação necessária à solução do problema está na base de conhecimento e não embutida em outro componente.

Considerando essas características, Fernandes (2003) ainda elenca alguns paradigmas de representação do conhecimento:

- a) redes semânticas: o conhecimento é representado por um rótulo de grafos direcionados, cujos nós representam conceitos e entidades, enquanto os arcos representam a relação entre entidades e conceitos;
- b) frames: muito parecidos com a rede semântica, exceto que cada nó representa conceitos e/ou situações. Cada nó tem várias propriedades que podem ser especificadas ou herdadas por padrão;
- c) lógica: um modo de declaração que representa o conhecimento;
- d) regras: sistemas de produção para codificar regras de condição/ação;
- e) casos: usa experiência passada, acumulando casos e tentando descobrir, por analogia, soluções para outros problemas.

Na próxima subseção segue o detalhamento da representação do conhecimento utilizada neste trabalho.

2.2.1 Regras de produção

Conforme Heinzle (1995, p. 25, grifo do autor), “O termo ‘sistema de produção’ é atualmente usado para descrever os sistemas que têm em comum o fato de serem construídos de um conjunto de regras para descrever condições e ações”. Essas regras são armazenadas como uma coleção de declarações ‘situação-ação’, na forma SE <premissas> ENTÃO <conclusões>.

A parte SE da regra é chamada de lado esquerdo ou parte antecedente e define o contexto para aplicação da regra. Deve ser avaliada em relação a base de conhecimentos como um todo. Quando o *reasoner* avalia a situação da premissa como verdadeira, a ação correspondente especificada no lado direito, ou parte consequente, é executada. Para que a ação na parte consequente seja considerada, as condições na parte antecedente da regra devem ser satisfeitas. Se qualquer premissa falhar, o lado direito também falha (HEINZLE, 1995, p. 25).

Uma ação refere-se a algum procedimento que leva a uma conclusão ou mudança no estado atual do quadro negro. Segundo Heinzle (1995, p. 25), “A representação por regras de produção é a forma mais utilizada em sistemas especialistas”. A justificativa para isso se baseia na naturalidade que o par ‘condição-ação’ representa para o homem. O Quadro 1 mostra três exemplos de regras de produção que poderiam fazer parte de um sistema especialista para orientação de candidatos ao curso de mestrado.

Quadro 1 – Exemplos de regras de produção

Regra	SE um candidato se inscrever para o curso de mestrado E o candidato preencher todos os requisitos exigidos E existirem vagas disponíveis ENTÃO o candidato passa a ser aluno do curso de mestrado
Regra	SE o aluno de mestrado for aprovado em todas as disciplinas do currículo ENTÃO ele deve escrever uma dissertação
Regra	SE a dissertação está concluída será submetida a uma banca examinadora E a banca examinadora aprovar o trabalho ENTÃO o aluno receberá o grau de mestre

Fonte: Heinzle (1995, p. 26).

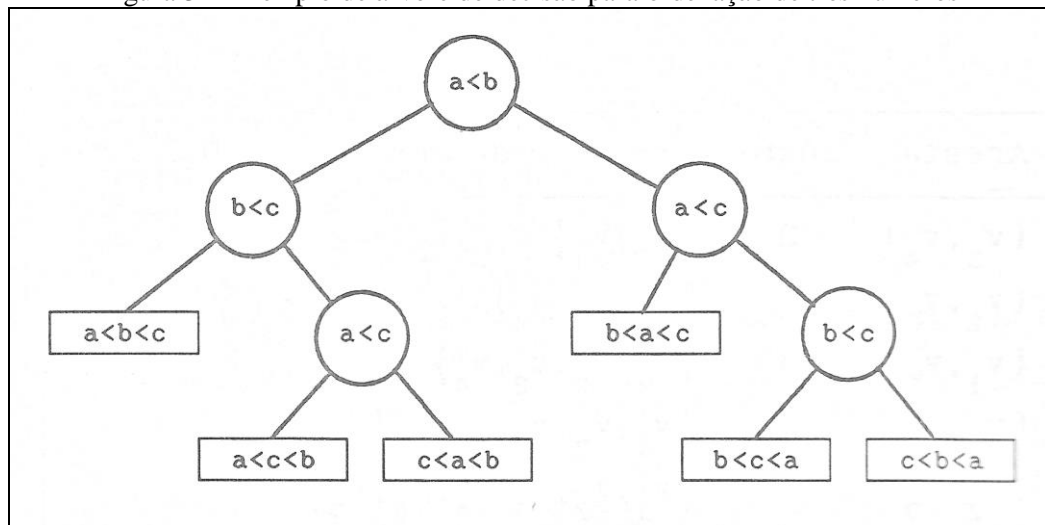
Além do benefício da naturalidade já citado, as regras podem ser manipuladas independentes e novas regras podem ser adicionadas a qualquer momento, tornando o sistema modular. Essa é uma característica importante, pois o conhecimento de um sistema especialista tende a aumentar com o passar do tempo. O padrão único utilizado para todas as regras caracteriza o sistema como uniforme, facilitando a manutenção e permitindo que pessoas não familiarizadas com o mesmo possam usufruir dele (HEINZLE, 1995, p. 27).

2.3 ÁRVORES DE DECISÃO

Uma árvore é uma estrutura hierárquica composta por nós ou vértices que possuem conhecimento ou informações e ramos ou arestas que fazem a conexão entre eles. Árvores são úteis para representar objetos classificáveis por causa de sua natureza. Um exemplo simples é uma árvore genealógica que representa a relação de ancestralidade de pessoas numa família. Além disso, são úteis para representar decisões, sendo chamadas de árvores de decisão nesse caso (GIARRATANO, 1994, p. 107).

Segundo Rabuske (1992, p. 112), “[...] é denominada uma árvore de decisão se cada vértice não-folha representa uma decisão. O teste tem início na raiz e o controle é, então, passado para um de seus filhos, dependendo do resultado do teste”. A Figura 3 mostra uma árvore de decisão com o intuito de ordenar três números a , b , c . As decisões são feitas pelos vértices e o controle é movido para o próximo vértice conforme a resposta for sim ou não, finalizando com os nós folhas que representam uma ação ou conclusão.

Figura 3 – Exemplo de árvore de decisão para ordenação de três números



Fonte: Rabuske (1992, p. 112).

Giarratano (1994) apresenta duas características primárias que comprovam a usabilidade das árvores de decisão em sistemas especialistas:

- a) fornecem a resposta para um problema a partir de um conjunto pré-determinado de possíveis respostas. Alguns exemplos disso são:
 - taxonomia: a identificação de uma joia a partir do conjunto de todas as pedras preciosas conhecidas,
 - diagnóstico: a seleção de uma solução possível a partir de um conjunto de recursos, ou ainda, a identificação de uma falha de um conjunto de possíveis causas;

- b) derivam uma solução através da redução do conjunto de soluções possíveis com uma série de perguntas que diminuem o espaço de busca da árvore de decisão.

Árvores de decisão são árvores dirigidas ou arborescências, classificadas como dígrafos acíclicos, em que cada nó possui apenas um nó pai e zero ou mais filhos, exceto a raiz que não possui nenhum nó pai. Um nó é considerado raiz de uma arborescência se qualquer outro nó dessa arborescência pode ser alcançado a partir dele, além disso é definido como folha qualquer nó que não possuir nenhum filho. Um caminho partindo de um nó n e percorrendo todos os seus filhos constitui uma subárvore na qual n é a raiz dessa subárvore (RABUSKE, 1992, p. 91).

2.4 TRATAMENTO DE INCERTEZA

Nem sempre os valores atribuídos às variáveis são determinísticos ('sim' e 'não'). Em alguns casos, existe a necessidade de representar um valor não exato baseado no contexto da situação. Conforme Morgado et al. (1991, p. 118, grifo do autor), “um experimento é *determinístico* quando repetido em condições semelhantes conduz a resultados essencialmente idênticos. Os experimentos que repetidos sob as mesmas condições produzem resultados geralmente diferentes serão chamados experimentos *aleatórios*”. Eventos aleatórios ocorrem com frequência no cotidiano. Ainda de acordo com Morgado et al. (1991, p. 119), algumas perguntas que podem demonstrar isso são: “vai chover amanhã? Qual será a temperatura máxima no próximo domingo? Qual será o número de ganhadores da Loteria Esportiva?”.

A teoria das probabilidades é um ramo da matemática que desenvolve modelos capazes de serem utilizados para estudar eventos aleatórios. O modelo utilizado para estudar algum evento aleatório em particular varia sua complexidade matemática conforme o evento em questão (MORGADO et al., 1991, p. 119). Sabendo que o ser humano não possui um conhecimento determinístico, raramente os especialistas afirmam uma determinada conclusão com certeza absoluta. Por isso é necessário o auxílio de um tratamento de incerteza para que um sistema especialista se aproxime da abordagem de um especialista humano. Porém existem algumas dificuldades na representação da confiabilidade das informações, Dantas (2010) cita duas delas:

- a) especialistas não se sentem confortáveis em pensar em termos de probabilidade. Suas estimativas não precisam corresponder àquelas definidas matematicamente;
- b) tratamentos matemáticos de probabilidade podem usar informações indisponíveis ou simplificações que não são claramente justificáveis em aplicações práticas.

Levando essas dificuldades em consideração, decidiu-se aplicar a mesma técnica para o tratamento de incerteza adotado pelo trabalho correlato Expert SINTA (LIA, 1999). Trata-se de uma abordagem da teoria das probabilidades sobre graus de confiança mais generalizada e sem uma base matemática forte, ao invés de fórmulas estatísticas rigorosas.

2.5 TRABALHOS CORRELATOS

A seguir são comentados dois trabalhos correlatos ao estudo realizado: Expert SINTA (LIA, 1999), uma ferramenta visual para criação de sistemas especialistas; e o JEOPS (FIGUEIRA, 2000), um motor de inferência para o desenvolvimento de aplicações inteligentes em Java.

2.5.1 Expert SINTA

O Expert SINTA é uma ferramenta para criação de sistemas especialistas desenvolvida pelo Laboratório de Inteligência Artificial (LIA) da Universidade Federal do Ceará (UFC). Tem como objetivo simplificar as etapas da criação dos mesmos, tendo sido desenvolvido com a IDE Delphi. Esse *software* utiliza o modo de raciocínio por encadeamento para trás e justifica essa opção da seguinte maneira:

O encadeamento para trás destaca-se em problemas nos quais há um grande número de conclusões que podem ser atingidas, mas o número de meios pelos quais elas podem ser alcançadas não é grande (um sistema de regras de alto grau de fan out), e em problemas nos quais não se pode reunir um número aceitável de fatos antes de iniciar-se a busca por respostas. (DANTAS, 2010, p.8).

Desse modo, ele tenta instanciar um objetivo por vez. Depois da escolha de um objetivo, é criada uma lista com as regras que o contenham como conclusão, sendo que a resolução do conflito de seleção da próxima regra a ser analisada é dada pelas características estáticas, ou seja, a ordem em que as regras aparecem na base de conhecimento. Além disso, o Expert SINTA disponibiliza o tratamento de incerteza com base na atribuição de fatores de confiança para premissas e conclusões. O cálculo do fator de confiança é baseado numa abordagem básica da teoria das probabilidades utilizando apenas conceitos de conjunção e disjunção para lidar com os fatores de confiança atribuídos aos valores. Originalmente, ele considera 50% como valor mínimo de confiança para que uma afirmação seja considerada verdadeira, mas esse valor pode ser ajustado pelo usuário. Levando em consideração o trecho de sentenças apresentado no Quadro 2, se o grau de confiança da premissa *estados das folhas = esfarelam facilmente* é 80% e o grau de confiança da premissa *presença de manchas irregulares = sim* é 70%, um cálculo de conjunção, representado pela fórmula:

$(\text{valor1} * \text{valor2}) / 100$, entre as duas sentenças retornará um grau de confiança de 56%, já que esse é o produto dos dois valores.

Quadro 2 – Trecho de sentenças com operador lógico ‘E’

SE estados das folhas = esfarelam facilmente
E presença de manchas irregulares = sim

Fonte: Dantas (2010, p. 15).

Conforme o trecho de sentenças apresentado no Quadro 3, considerando o grau de confiança da premissa besouros vermelhos = sim sendo 80% e o grau de confiança da premissa larvas marrons = sim sendo 70%, um cálculo de disjunção, representado pela fórmula: $\text{valor1} + \text{valor2} - ((\text{valor1} * \text{valor2}) / 100)$, entre as duas sentenças retornará um grau de confiança de 94%. O mesmo cálculo se aplica para os casos em que uma variável recebe duas vezes o mesmo valor em pontos diferentes do processo de raciocínio.

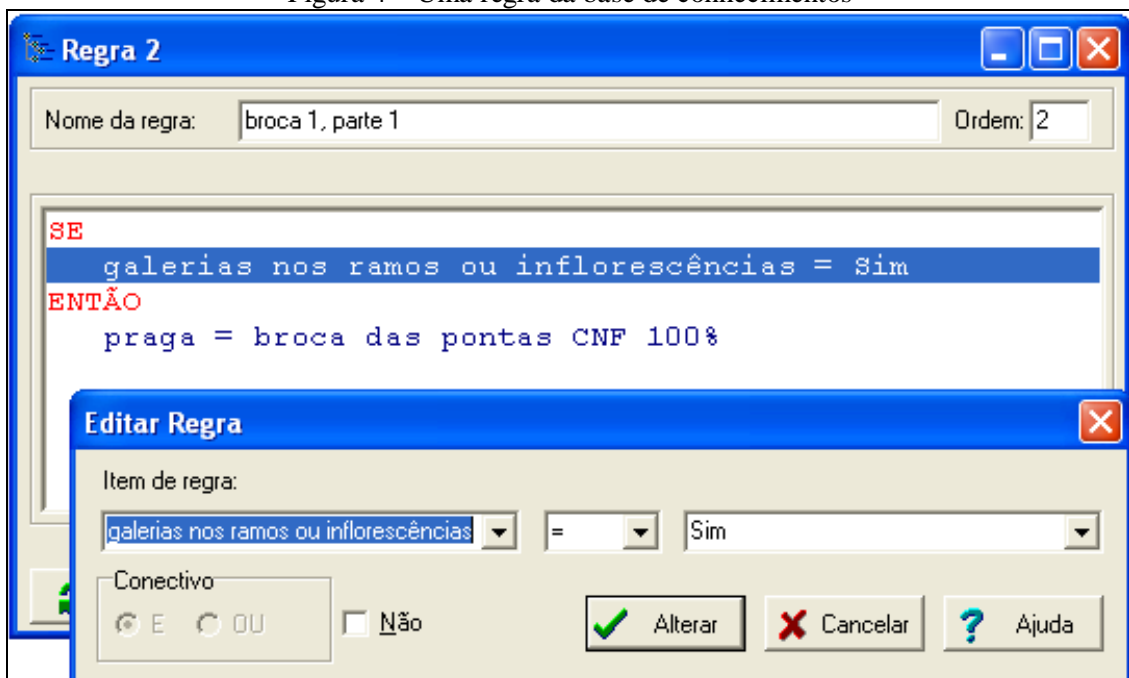
Quadro 3 – Trecho de sentenças com operador lógico ‘OU’

SE besouros vermelhos = sim
OU larvas marrons = sim

Fonte: Dantas (2010, p. 16).

A tela com os resultados do processo de raciocínio aparece toda vez que as regras relativas ao objetivo pesquisado acabarem de ser analisadas, ou seja, quando não existir mais nenhuma regra que instancie aquele objetivo. Esse processo é relativo aos objetivos multivalorados (podem possuir mais de um valor ao mesmo tempo). Já para os objetivos univalorados e numéricos após instanciarem a primeira vez, a tela de resultados aparece e o programa continua para o próximo objetivo, se existir.

Figura 4 – Uma regra da base de conhecimentos



Fonte: Software Expert SINTA.

Conforme pode ser observado na Figura 4, a representação de uma regra no Expert SINTA segue o modelo geral das regras de produção. Seu modo de edição da base de conhecimentos não necessita que o usuário domine conceitos avançados de programação, pois consegue manter a naturalidade na sua representação.

2.5.2 JEOPS

O projeto surgiu como um trabalho para uma disciplina de graduação, Inteligência Artificial Simbólica, em 1997 na Universidade Federal de Pernambuco (UFPE). O objetivo do *Java Embedded Object Production System* (JEOPS) era, segundo Figueira (2000, p. 56, grifo do autor), “[...] desenvolver um motor de inferência para uma linguagem ‘convencional’”. Figueira (2007) ainda lista alguns motivos para a escolha do Java como linguagem do projeto:

- a) linguagem conhecida por toda a equipe envolvida;
- b) estrutura elegante que permitia o desenvolvimento de aplicações segundo os conceitos de orientação a objetos;
- c) apresentava facilidades de definição de interfaces gráficas, acesso a rede, entre outras;
- d) sobretudo, era uma linguagem nova e não havia nenhum mecanismo de inferência integrado à ela.

O JEOPS utiliza o modo de raciocínio de encadeamento progressivo (encadeamento para frente). Possui um mecanismo flexível para a resolução de conflitos, podem-se citar alguns como: escolha entre regra mais recente utilizada ou regra menos recente utilizada, característica estática (ordem das regras no arquivo define a prioridade) e também permite que o usuário crie sua própria resolução de conflitos (FIGUEIRA, 2000, p. 76).

Quadro 4 – Exemplo de regra para venda de produto

```
rule trade {
  /* In this rule, one agent will buy a product from another agent. */
  declarations
    Salesman s;
    Customer c;
    Product p;
  conditions
    c.needs(p);    // he won't buy junk...
    s.owns(p);
    s.priceAskedFor(p) <= c.getMoney();
  actions
    s.sell(p);
    c.buy(p);
}
```

Fonte: Figueira (2007).

O Quadro 4 mostra um exemplo de uma regra capaz de ser interpretada pelo JEOPS. As implementações dos métodos não são apresentadas pois os mesmos possuem nomes descritivos o suficiente para o entendimento do exemplo.

Nessa regra é possível notar o uso de conceitos da orientação a objetos, em que `Salesman`, `Customer` e `Product` são classes Java e cada uma é responsável pelos métodos que afetam diretamente a ela. De qualquer modo, essa regra pode ser expressa no padrão geral das regras de produção, como mostrado no Quadro 5.

Quadro 5 – Padrão das regras de produção

Regra
SE cliente precisa do produto
E vendedor tem o produto
E preço do produto <= dinheiro do cliente
ENTÃO vendedor vende o produto
E cliente compra o produto

Apesar de essa representação ser possível, na sentença onde é verificado se o ‘dinheiro de cliente’ é maior ou igual ao ‘preço do produto’ normalmente existe uma limitação com os sistemas especialistas, pois não permitem representar as regras dessa maneira. Isso é possível com o JEOPS porque ele possui uma estrutura de regras baseadas na abordagem da programação orientada a objetos do Java. Infelizmente, a legibilidade das regras é perdida dificultando o acesso a ferramenta para pessoas que não estejam familiarizadas com programação, tornando necessário um programador para a construção da base de conhecimentos.

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas de desenvolvimento do ambiente proposto. Na Seção 3.1 são apresentados os principais requisitos. A Seção 3.2 apresenta a especificação por meio de diagramas. A Seção 3.3 apresenta os detalhes da implementação do ambiente e, por fim, a Seção 3.4 apresenta os resultados obtidos e algumas discussões sobre eles.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O ambiente deverá:

- a) disponibilizar um módulo para o gerenciamento da base de conhecimentos (Requisito Funcional – RF01);
- b) disponibilizar um módulo *reasoner* capaz de seguir um raciocínio utilizando a base de conhecimentos com o intuito de chegar a uma variável objetivo (RF02);
- c) permitir que o usuário atribua um fator de confiabilidade para cada premissa no momento de raciocínio do *reasoner* (RF03);
- d) permitir que o usuário atribua um fator de confiabilidade para cada conclusão no momento do gerenciamento da base de conhecimentos (RF04);
- e) disponibilizar uma interface gráfica para o usuário visualizar a árvore de pesquisa (sequência em que as regras foram utilizadas pelo *reasoner*) que o ambiente gerou após encontrar um objetivo (RF05);
- f) disponibilizar uma interface gráfica para o usuário visualizar as variáveis que foram instanciadas e os respectivos valores atribuídos após o ambiente encontrar um objetivo (RF06);
- g) disponibilizar uma opção para persistir a base de conhecimentos (RF07);
- h) utilizar como critério na resolução de conflitos as regras que levam ao objetivo com o menor caminho possível (Requisito Não-Funcional – RNF01);
- i) disponibilizar uma opção para habilitar/desabilitar uma dica para regras que não alcançam nenhum objetivo (RNF02);

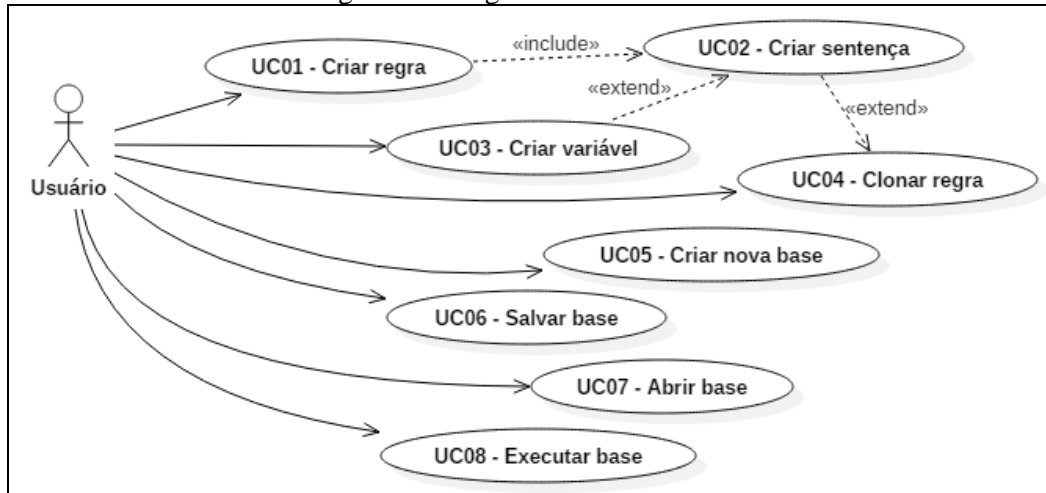
3.2 ESPECIFICAÇÃO

A especificação do ambiente foi realizada seguindo conceitos da orientação a objetos e utilizando alguns diagramas da *Unified Modeling Language* (UML). A ferramenta StarUML foi escolhida para criação dos diagramas de casos de uso, classes e atividades.

3.2.1 Diagrama de casos de uso

Nesta seção são descritos os casos de uso do ambiente. A Figura 5 ilustra o diagrama com os casos de uso identificados. O ambiente possui apenas um ator, denominado *Usuário*, o qual faz uso de todas as funcionalidades disponíveis.

Figura 5 – Diagrama de casos de uso



O caso de uso UC01 - Criar regra descreve a ação que pode ser realizada pelo *Usuário* para que uma regra seja criada. Detalhes deste caso de uso estão descritos no Quadro 6.

Quadro 6 – Caso de uso UC01 - Criar regra

Número	01
Caso de uso	Criar regra
Requisitos atendidos	RF01
Descrição	Este caso de uso permite a criação de uma regra para a base de conhecimentos.
Ator	Usuário
Pré-condições	Estar com o ambiente aberto.
Cenário principal	<ol style="list-style-type: none"> 1. O <i>Usuário</i> clica no botão Inserir Regra. 2. O software abre a tela para cadastrar uma Regra. 3. O <i>Usuário</i> cadastra pelo menos uma Premissa. 4. O <i>Usuário</i> cadastra pelo menos uma Conclusão. 5. O <i>Usuário</i> clica no botão Salvar.
Fluxo alternativo 1	A partir do passo 2 o <i>Usuário</i> pode optar por definir uma Descrição para a regra.
Exceção 1	No passo 5, caso não exista nenhuma Premissa, será mostrada uma caixa de diálogo com o erro.
Exceção 2	No passo 5, caso não exista nenhuma Conclusão, será mostrada uma caixa de diálogo com o erro.
Pós-condição	O software grava a regra na base de conhecimentos com a última ordem disponível.

O caso de uso UC02 - Criar sentença descreve a ação que é realizada pelo *Usuário* quando uma sentença está sendo cadastrada para uma regra. Detalhes deste caso de uso estão descritos no Quadro 7.

Quadro 7 – Caso de uso UC02 - Criar sentença

Número	02
Caso de uso	Criar sentença
Requisitos atendidos	RF01, RF04
Descrição	Este caso de uso permite a criação de uma sentença (premissa/conclusão) para uma regra.
Ator	Usuário
Pré-condições	Estar cadastrando uma regra.
Cenário principal	<ol style="list-style-type: none"> 1. O Usuário clica no botão + na região da sentença desejada. 2. O software abre a tela para cadastrar uma Sentença. 3. O Usuário digita um nome de Variável. 4. O Usuário digita um Valor. 5. O Usuário clica no botão Salvar.
Fluxo alternativo 1	No passo 3, se a Sentença for uma Premissa, o software atualiza a lista de operadores conforme o Tipo da variável digitada.
Fluxo alternativo 2	Após o passo 3, se a Sentença for uma Premissa, o Usuário pode escolher um Operador da lista de operadores possíveis.
Fluxo alternativo 3	No passo 3, se a Variável que o Usuário digitou não existir, ao sair do campo de nome da variável um diálogo será mostrado perguntado o Tipo da variável (Univalorada, Multivalorada, Numérica).
Fluxo alternativo 4	No fluxo alternativo 3, se o Tipo de variável selecionado no diálogo for Numérica, a tela de cadastro de Variável é aberta preenchida com o nome e o tipo da variável.
Fluxo alternativo 5	No passo 3, enquanto o Usuário estiver digitando o nome da variável, o software procurará esse nome nas variáveis já cadastradas e dará uma opção para autopreenchimento.
Fluxo alternativo 6	No passo 4, enquanto o Usuário estiver digitando um Valor, o software procurará os valores cadastrados para essa variável e dará uma opção para autopreenchimento.
Fluxo alternativo 7	Após o passo 4, se a Sentença for uma Conclusão, o Usuário pode digitar um grau de confiança.
Fluxo alternativo 8	No passo 5, se o Valor digitado não estiver cadastrado na Variável escolhida, mas a Variável já existir na base de conhecimentos, o software mostrará um diálogo perguntado se deseja adicionar esse novo Valor à Variável.
Exceção 1	No passo 5, se a Variável estiver em branco, será mostrada uma caixa de diálogo com o erro.
Exceção 2	No passo 5, se o Valor estiver em branco, será mostrada uma caixa de diálogo com o erro.
Pós-condição	O software grava a sentença na regra atual, se a variável não existir, ela é criada automaticamente.

O caso de uso UC03 - Criar variável descreve a ação que é realizada pelo Usuário quando há necessidade de criar uma variável manualmente. Detalhes deste caso de uso estão descritos no Quadro 8.

Quadro 8 – Caso de uso UC03 - Criar variável

Número	03
Caso de uso	Criar variável
Requisitos atendidos	RF01
Descrição	Este caso de uso permite a criação de uma variável.
Ator	Usuário
Pré-condições	Estar com o ambiente aberto.
Cenário principal	<ol style="list-style-type: none"> 1. O Usuário clica no botão Variáveis. 2. O Usuário clica no botão Inserir. 3. O Usuário digita um Nome. 4. O Usuário escolhe um Tipo de variável. 5. O Usuário clica no botão Salvar.
Fluxo alternativo 1	A qualquer momento o Usuário pode digitar uma Pergunta para a variável.
Fluxo alternativo 2	A qualquer momento o Usuário pode selecionar a opção para alternar entre Variável/Variável Objetivo.
Fluxo alternativo 3	Após o passo 4, se a variável for Univalorada ou Multivalorada, o Usuário pode adicionar Valores a ela clicando no botão Inserir Valor e digitando no novo campo que aparece.
Fluxo alternativo 4	Após o passo 4, se a variável for Numérica, o Usuário deve digitar um valor Mínimo e um valor Máximo.
Fluxo alternativo 5	No passo 5, se a variável for Univalorada e não existir nenhum valor na lista de Valores, o software vai inserir os valores SIM e NÃO.
Exceção 1	No passo 5, se o Nome estiver em branco, será mostrada uma caixa de diálogo com o erro.
Exceção 2	No passo 5, se o Tipo de variável for Multivalorada e não existir nenhum valor na lista de Valores, será mostrada uma caixa de diálogo com o erro.
Exceção 3	No passo 5, se o Tipo de variável for Numérica e o valor Mínimo/Máximo estiver em branco, será mostrada uma caixa de diálogo com o erro.
Exceção 4	No passo 5, se o Tipo de variável for Numérica e o valor Mínimo for maior que o valor Máximo, será mostrada uma caixa de diálogo com o erro.
Pós-condição	O software grava a variável na base de conhecimentos.

O caso de uso UC04 - Clonar regra descreve a ação que é realizada pelo Usuário quando há necessidade de clonar toda a estrutura de uma regra existente a fim de modificar pequenos detalhes para criar uma nova regra. Detalhes deste caso de uso estão descritos no Quadro 9.

Quadro 9 – Caso de uso UC04 - Clonar regra

Número	04
Caso de uso	Clonar regra
Requisitos atendidos	RF01
Descrição	Este caso de uso permite que o usuário clone uma regra existente na base de conhecimentos.
Ator	Usuário
Pré-condições	Estar com a regra a ser clonada selecionada.
Cenário principal	<ol style="list-style-type: none"> 1. O Usuário clica no botão Clonar Regra. 2. O software exibe a tela de edição de regra com os dados da regra selecionada. 3. O Usuário altera as sentenças necessárias. 4. O Usuário clica no botão Salvar.
Fluxo alternativo 1	A partir do passo 2 o Usuário pode optar por redefinir a Descrição da regra.
Exceção 1	No passo 4, caso não exista nenhuma Premissa, será mostrada uma caixa de diálogo com o erro.
Exceção 2	No passo 4, caso não exista nenhuma Conclusão, será mostrada uma caixa de diálogo com o erro.
Pós-condição	O software cria uma nova regra com as informações fornecidas e com a última ordem disponível.

O caso de uso UC05 - Criar nova base descreve a ação que é realizada pelo Usuário com o intuito de criar uma nova base. Detalhes deste caso de uso estão descritos no Quadro 10.

Quadro 10 – Caso de uso UC05 - Criar nova base

Número	05
Caso de uso	Criar nova base
Requisitos atendidos	RF07
Descrição	Este caso de uso permite a criação de uma nova base de conhecimentos.
Ator	Usuário
Pré-condições	Estar com o ambiente aberto.
Cenário principal	<ol style="list-style-type: none"> 1. O Usuário clica no botão Nova Base. 2. O software mostra uma mensagem de confirmação.
Fluxo alternativo 1	No passo 2, se o Usuário responder que não quer criar uma nova base de conhecimentos, o caso de uso encerra.
Pós-condição	O software descarta a base atual e cria uma nova base de conhecimentos em branco.

O caso de uso UC06 - Salvar base descreve a ação que é realizada pelo Usuário ao salvar a base de conhecimentos atual. Detalhes deste caso de uso estão descritos no Quadro 11.

Quadro 11 – Caso de uso UC06 – Salvar base

Número	06
Caso de uso	Salvar base
Requisitos atendidos	RF07
Descrição	Este caso de uso permite salvar a base de conhecimentos em arquivo no computador.
Ator	Usuário
Pré-condições	Estar com o ambiente aberto.
Cenário principal	<ol style="list-style-type: none"> 1. O Usuário clica no botão Salvar Base. 2. O software exibe a consulta às pastas do computador. 3. O Usuário seleciona um Local. 4. O Usuário digita um Nome para o arquivo. 5. O Usuário clica no botão Salvar.
Fluxo alternativo 1	No passo 5, se o Usuário escolher a opção Cancelar ao invés da opção Salvar, o caso de uso encerra.
Pós-condição	O software salva a base de conhecimentos atual no local especificado pelo Usuário.

O caso de uso UC07 – Abrir base descreve a ação que é realizada pelo Usuário abrir uma base de conhecimentos salva anteriormente. Detalhes deste caso de uso estão descritos no Quadro 12.

Quadro 12 – Caso de uso UC07 – Abrir base

Número	07
Caso de uso	Abrir base
Requisitos atendidos	RF07
Descrição	Este caso de uso permite abrir uma base de conhecimentos previamente salva.
Ator	Usuário
Pré-condições	Estar com o ambiente aberto e possuir uma base de conhecimentos salva.
Cenário principal	<ol style="list-style-type: none"> 1. O Usuário clica no botão Abrir Base. 2. O software exibe a consulta às pastas do computador. 3. O Usuário procura o local onde o arquivo está salvo. 4. O Usuário seleciona o arquivo. 5. O Usuário clica no botão Abrir.
Fluxo alternativo 1	No passo 5, se o Usuário escolher a opção Cancelar ao invés da opção Abrir, o caso de uso encerra.
Pós-condição	O software carrega a base de conhecimentos do local especificado pelo Usuário.

Por fim, o caso de uso UC08 – Executar base descreve o conjunto de ações realizadas pelo Usuário no momento do processo de raciocínio do *reasoner*. Detalhes deste caso de uso estão descritos no Quadro 13.

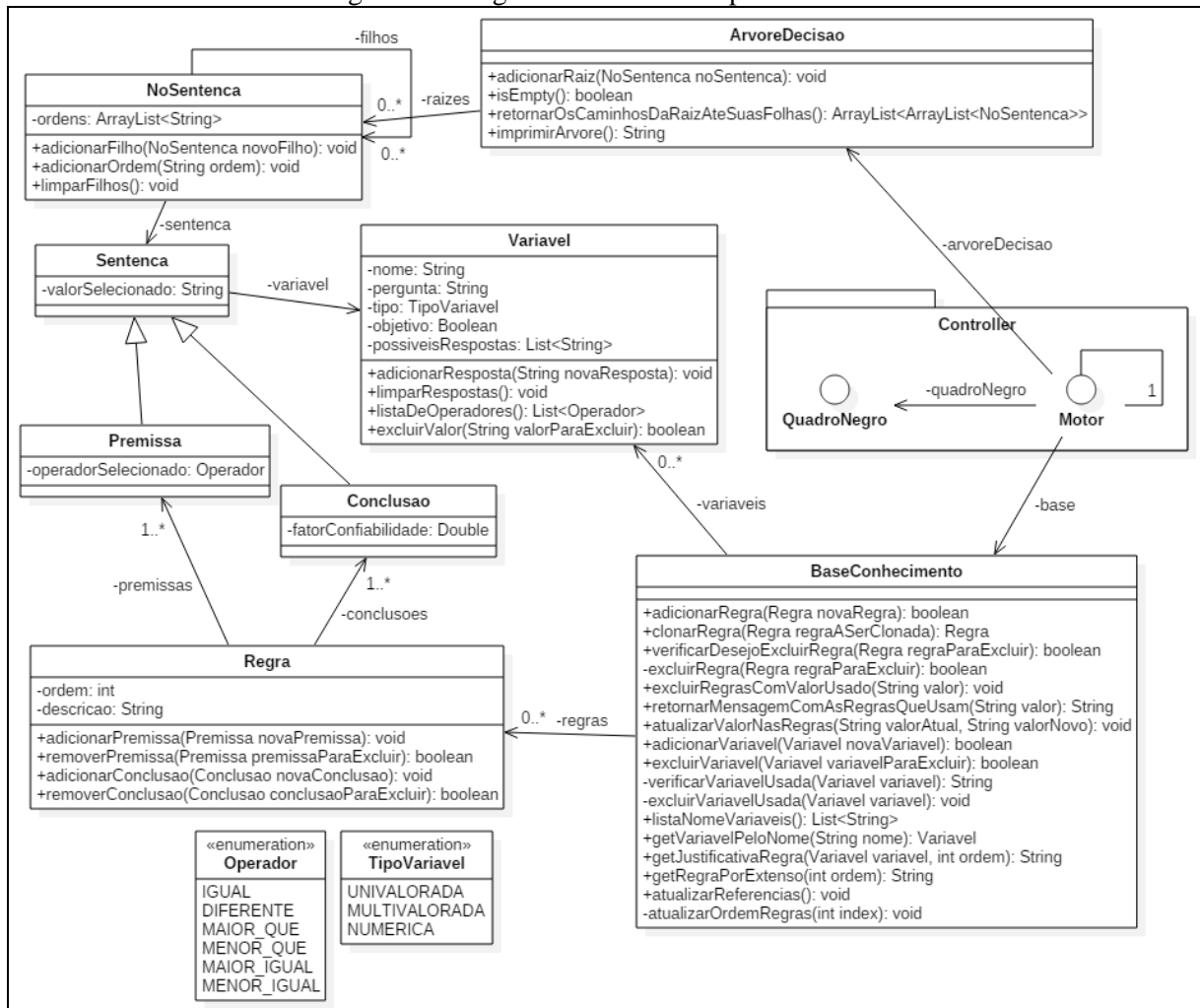
Quadro 13 – Caso de uso UC08 – Executar base

Número	08
Caso de uso	Executar base
Requisitos atendidos	RF02, RF03, RF05, RF06
Descrição	Este caso de uso permite a execução da base de conhecimentos com a finalidade de encontrar um objetivo ao final da consulta.
Ator	Usuário
Pré-condições	Estar com o ambiente aberto e a base de conhecimentos cadastrada.
Cenário principal	<ol style="list-style-type: none"> 1. O Usuário clica no botão Executar. 2. O software apresenta a tela de consulta. 3. O Usuário entra com informações e seus respectivos graus de confiança. 4. O software apresenta a tela de resultados.
Fluxo alternativo 1	Os passos 2 e 3 se repetem até que o <i>reasoner</i> avalie todas as regras de um objetivo Multivalorado ou até que ele instancie um objetivo Univalorado/Numérico.
Fluxo alternativo 2	A qualquer momento o Usuário pode clicar no botão ‘Por que?’ e analisar a justificativa apresentada pelo software.
Fluxo alternativo 3	A qualquer momento o Usuário pode fechar a tela atual e cancelar o processo de raciocínio e com isso encerrar o caso de uso.
Fluxo alternativo 4	No passo 4, caso existam outros objetivos, o Usuário pode clicar no botão Continuar, passando controle para o fluxo alternativo 1.
Fluxo alternativo 5	No passo 4, o Usuário pode visualizar a árvore de pesquisa clicando na guia de mesmo nome.
Fluxo alternativo 6	No passo 4, o Usuário pode visualizar as variáveis e seus respectivos valores instanciados clicando na guia Variáveis.
Pós-condição	O software mostra a tela de resultados uma última vez antes de encerrar o processo de raciocínio.

3.2.2 Diagrama de classes

A Figura 6 apresenta o diagrama de classes do pacote `Model` com as principais classes que compõem o software desenvolvido. Optou-se por exibir meramente os métodos, funções e atributos com maior relevância para o funcionamento do ambiente.

Figura 6 – Diagrama de classes do pacote Model



A classe `Variavel` é responsável por armazenar o nome dela, uma pergunta que o usuário pode definir, o tipo dela (determina o comportamento durante o processo de raciocínio com a base), um atributo que determina se ela é uma variável objetivo e uma lista com as possíveis respostas que podem ser atribuídas a ela. O método `listaDeOperadores()` retorna os operadores possíveis conforme o atributo `tipo` e o método `excluirValor()` verifica se o valor está sendo utilizado em alguma regra (perguntando ao usuário se deseja continuar e excluir as regras que o contenham) antes de efetivamente excluir o valor da lista de `possiveisRespostas`. O atributo `tipo` é definido pelo Enum `TipoVariavel` que contém os seguintes tipos:

- UNIVALORADA:** a variável pode conter apenas um valor quando instanciada, permite somente os operadores `IGUAL` e `DIFERENTE`;
- MULTIVALORADA:** quando instanciada a variável pode contar quantos valores forem necessários, permite somente os operadores `IGUAL` e `DIFERENTE`;

- c) NUMERICA: tem um comportamento igual a univalorada, porém aceita apenas valores numéricos, além de permitir qualquer operador.

Já a classe `Sentenca` armazena um objeto `Variavel` e um valor da lista de valores possíveis da variável em questão. Por sua vez, a classe `Conclusao` e a classe `Premissa` herdam esses atributos em comum da classe `Sentenca`, sendo que a classe `Conclusao` tem como adicional o atributo do fator de confiabilidade que precisa ser armazenado como parte de uma regra e a classe `Premissa` tem adicionalmente o atributo que representa o operador selecionado pelo usuário. Esse operador é definido pelo Enum `Operador` que contém os seguintes operadores (comuns na maioria das linguagens de programação):

- a) IGUAL: representa o operador de igualdade '=' (ou '==' em algumas linguagens de programação);
- b) DIFERENTE: representa o operador de igualdade '<>' (ou '!=' em algumas linguagens de programação);
- c) MAIOR_QUE: representa o operador relacional '>';
- d) MENOR_QUE: representa o operador relacional '<';
- e) MAIOR_IGUAL: representa a conjunção entre o operador relacional MAIOR_QUE e o operador de igualdade IGUAL, sendo representado por '>=';
- f) MENOR_IGUAL: representa a conjunção entre o operador relacional MENOR_QUE e o operador de igualdade IGUAL, sendo representado por '<='.

A classe `Regra` contém a ordem em que aparece na tela principal, sua descrição, as listas de premissas e conclusões e os respectivos métodos para adicionar e remover itens em ambas as listas. Já a classe `BaseConhecimento` contém a lista de regras e variáveis, com os métodos responsáveis para adicionar/excluir objetos de cada uma das listas. É possível destacar o método `atualizarReferencias()` que é chamado quando uma base de conhecimentos é carregada do arquivo, fazendo com que as referências entre variáveis e regras sejam propriamente atribuídas. O método `atualizarValorNasRegras()` é chamado quando um valor de uma variável é alterado. Ele busca e substitui o valor antigo com o novo valor em todas as regras que o utilizam. Já o método `listaNomeVariaveis()` é utilizado para trazer a lista de todas as variáveis já cadastradas para serem utilizadas nos componentes de auto completar. Por fim, já que este trabalho não trata regras com conectivos lógicos 'OU' e é totalmente possível transpor esses tipos de regras utilizando somente conectivos lógicos 'E', criou-se o método `clonarRegra()` que facilita a transposição dessas regras, já que essa transposição requer a repetição de várias regras idênticas com pequenas modificações.

Já a classe `ArvoreDecisao` é utilizada somente no processo de raciocínio do *reasoner*. Suas raízes são compostas por uma lista de objetos da classe `NoSentenca` e seu principal método é o `retornarOsCaminhosDaRaizAteSuasFolhas()` que, conforme a própria descrição, é o método responsável por retornar os caminhos das raízes até suas folhas, ou seja, os caminhos que representam o conjunto de regras (com premissas e conclusões) necessárias para se chegar a um objetivo. Seu método chamado de `imprimirArvore()` mostra no console a árvore de decisão organizada em formato de texto. Finalmente, a classe `NoSentenca` é a representação de cada nó da árvore, ela contém o atributo `sentenca` que pode receber tanto uma instância de `Premissa` quanto uma instância de `Conclusao`, sendo que uma premissa representa uma decisão e uma conclusão representa uma ação que gera novos dados. Além disso, a classe `NoSentenca` também inclui o atributo `ordens` que é uma lista com as ordens das regras que contém certa premissa. Isso ocorre porque a árvore de decisão do trabalho dá preferência de inclusão para as premissas que mais se repetem afim de criar uma árvore com número menor de nós, pois evita repetir a mesma premissa em posições diferentes da árvore (quando possível). A Figura 7 apresenta o diagrama de classes do pacote `Controller`, ilustrando apenas os principais métodos e funções.

Figura 7 – Diagrama de classes do pacote `Controller`



O pacote `Controller` trata da comunicação entre as telas do ambiente e o pacote `Model`. A classe `Motor` é o núcleo do ambiente e segue o padrão de projeto `Singleton` representado pelo atributo estático `_instancia` e pelo método `getInstancia()`, de modo que exista apenas um objeto `Motor` instanciado a cada execução do ambiente. Além disso contém o método `limparMotor()` que limpa o atributo estático `_instancia`, e é chamado quando uma nova base de conhecimentos deve ser criada. Ela possui referências para as classes `BaseConhecimento`, `ArvoreDecisao` e `QuadroNegro`, e é responsável pelo uso dos recursos disponíveis por elas. O método `montarArvoreDecisao()` é utilizado no início da execução do processo de raciocínio e serve para transformar as premissas e conclusões das regras da base de conhecimento em nós da árvore de decisão. Ele retorna `false` caso dê algum erro nessa transformação. Já o método `verificarPremissa()` e seus derivados são utilizados para checar a veracidade de uma premissa, retornando o coeficiente de confiabilidade se verdadeira ou -1 para identificar o contrário.

Por outro lado, a classe `QuadroNegro` é responsável por armazenar os demais recursos necessários no processo de raciocínio como a árvore de pesquisa, a lista das variáveis e seus respectivos valores instanciados, os caminhos até os objetivos (que são removidos conforme necessário) e os objetivos que ainda não foram instanciados (para caso haja mais de um). Além dela possuir os métodos referentes a inclusão e recuperação das variáveis instanciadas (inclusive objetivos), ela também possui métodos para verificar se ainda existem objetivos não instanciados (representado por `acabouObjetivosNaoInstanciados()`) e remover um objetivo dessa lista (representado por `removerObjetivoInstanciado()`) pois ele foi instanciado.

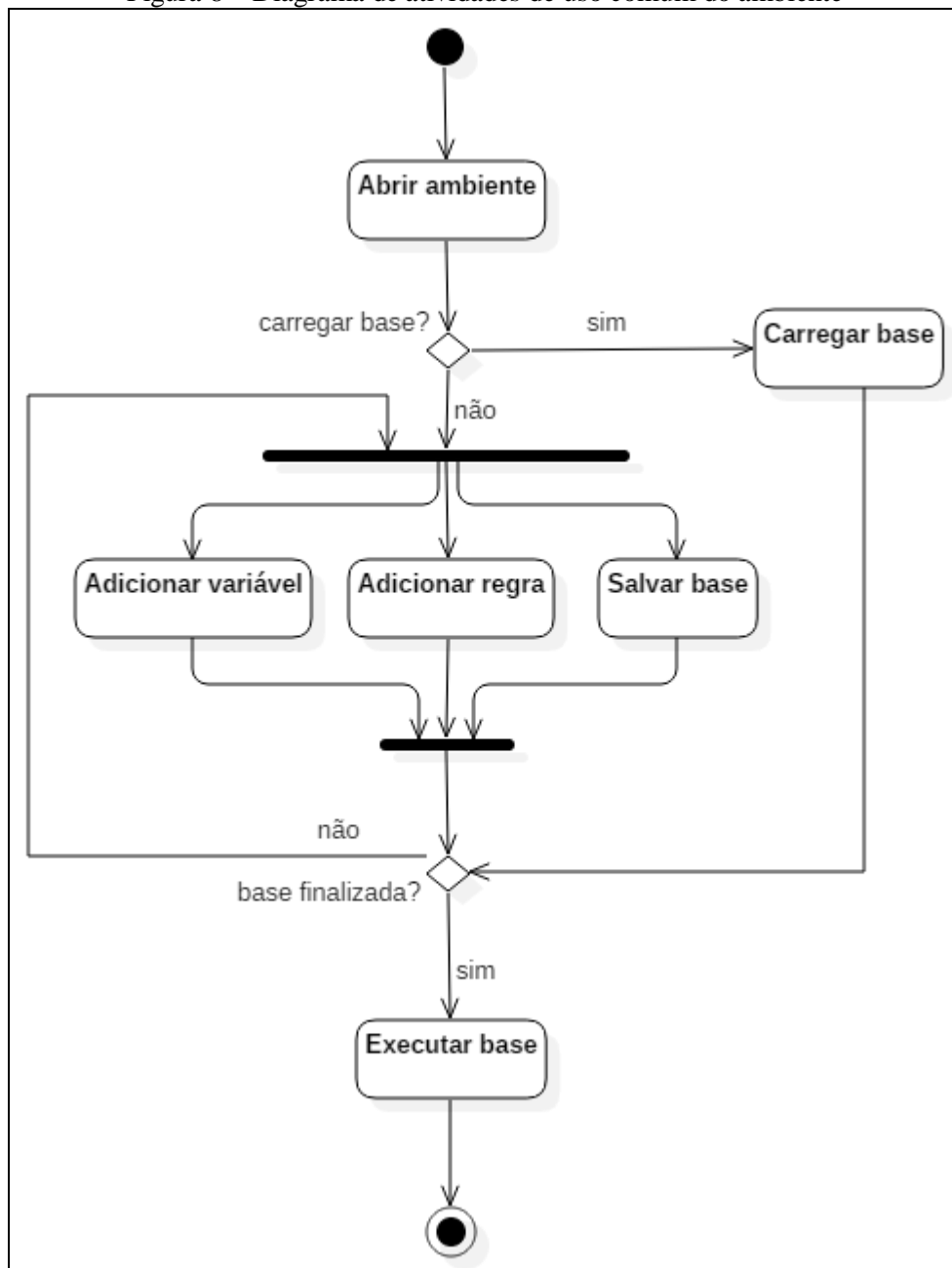
Os demais métodos da classe `QuadroNegro` se referem ao tratamento da lista de caminhos até os objetivos, no qual o método `isUltimaSentencaCaminhoAtual()` é utilizado para confirmar se existe mais algum nó no caminho que está sendo analisado, pois se for o último, é chamado o método `eliminarCaminhoAtual()` para remover o caminho pois ele já foi todo analisado e os objetivos aceitos. Outro momento em que o método `eliminarCaminhoAtual()` é invocado é quando uma premissa do caminho que está sendo analisado for rejeitada, além de remover o caminho atual, esse método também invoca o método `atualizarCaminhosPossiveis()` que remove em cadeia os caminhos caso o último objetivo instanciado seja diferente de `MULTIVALORADO`. O método `proximoCaminho()` é utilizado no laço de repetição do `Motor` que analisa todos os nós da árvore de decisão, já o método `organizaCaminhosAteObjetivos()` é chamado somente quando a classe

QuadroNegro é instanciada, sendo responsável por ordenar os caminhos conforme os requisitos propostos (levando em consideração os caminhos mais curtos e utilizando como resolução de conflitos a ordem atribuída a regra, do menor para o maior).

3.2.3 Diagrama de atividades

O diagrama de atividades, apresentado na Figura 8, mostra o fluxo normal de uso do ambiente.

Figura 8 – Diagrama de atividades de uso comum do ambiente



Ao iniciar o ambiente o usuário pode escolher entre carregar uma base de conhecimentos salva anteriormente ou criar uma nova adicionando variáveis e regras. A

qualquer momento o usuário pode decidir salvar a base de conhecimentos que está desenvolvendo para uso posterior. Ao finalizar a inclusão das variáveis e regras na base de conhecimentos do sistema especialista desejado, o usuário pode então executar ele a fim de realizar o raciocínio com o conjunto de regras criados. Este é o caminho de uso comum do ambiente, porém é possível que o usuário opte pelo caminho alternativo, no qual ele apenas adiciona as regras a base de conhecimentos e deixa que o ambiente crie as variáveis de acordo com o necessário.

3.3 IMPLEMENTAÇÃO

Nesta seção são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

A implementação foi feita utilizando o ambiente de desenvolvimento IDE Eclipse Luna na linguagem de programação Java 7, que foi escolhida por sua portabilidade para diversas plataformas e sua imensa comunidade ativa. A interface gráfica foi feita utilizando a biblioteca gráfica *Swing* (nativa do JDK) com o auxílio do complemento *WindowBuilder*. Durante o desenvolvimento optou-se por utilizar o formato de dados *JavaScript Object Notation* (JSON) para criação e recuperação da serialização da base de conhecimentos, para isso foi utilizada a biblioteca Gson (GOOGLE, 2014).

3.3.1.1 Persistência da base de conhecimentos

Conforme citado anteriormente, foi utilizada a biblioteca Gson para converter o objeto da base de conhecimentos para um arquivo JSON e vice-versa. O Quadro 14 mostra o trecho de código utilizado para gerar o arquivo JSON a partir do objeto da base de conhecimentos.

Quadro 14 – Trecho de código utilizado para gerar o arquivo JSON

```
01. public static boolean salvar(File file) {
02.     try {
03.         FileWriter writerFile = new FileWriter(file);
04.         Gson gson = new Gson();
05.         String baseJSON = gson.toJson(Motor.getInstancia().getBase());
06.         writerFile.write(baseJSON);
07.         writerFile.close();
08.     } catch (IOException e) {
09.         return false;
10.     }
11.     return true;
12. }
```

A biblioteca Gson foi escolhida por sua simplicidade, como é possível notar nas linhas 4 e 5. Para gerar um arquivo no formato JSON é necessário apenas instanciar um objeto Gson

e então passar por parâmetro o objeto que será convertido em JSON para o método `toJson()` do objeto `Gson` instanciado. O retorno do método `toJson()` será uma `String` no formato JSON. O Quadro 15 mostra o trecho de código utilizado para recuperar o objeto da base de conhecimentos a partir do arquivo JSON salvo.

Quadro 15 – Trecho de código utilizado para recuperar um objeto do arquivo JSON gerado

```

01. public static boolean abrir(File file) {
02.     try {
03.         FileReader readerFile = new FileReader(file);
04.         Gson gson = new Gson();
05.         BaseConhecimento baseObj = gson.fromJson(readerFile,
06.             BaseConhecimento.class);
07.         Motor.getInstancia().setBase(baseObj);
08.         Motor.getInstancia().getBase().atualizarReferencias();
09.     } catch (Exception e) {
10.         e.printStackTrace();
11.         return false;
12.     }
13.     return true;
14. }

```

Para recuperar um objeto a partir do arquivo JSON criado, é necessário especificar para o método `fromJson()` o arquivo e a classe a partir da qual esse arquivo foi gerado (para que o método identifique os atributos salvos e os recupere), como visto na linha 5. Infelizmente, ao recuperar a base de conhecimentos arquivada, as referências das variáveis nas sentenças das regras foram perdidas. Para resolver isso, foi necessário implementar um passo adicional na classe `BaseConhecimento`, isto é, o método `atualizarReferencias()` representado na linha 8. O Quadro 16 mostra com detalhe esse método.

Quadro 16 – Trecho de código do método `atualizarReferencias()`

```

01. public void atualizarReferencias() {
02.     // Percorre cada objeto Regra da base de conhecimentos
03.     for (Regra regra : getRegras()) {
04.         // Percorre cada objeto Premissa da Regra
05.         for (Premissa premissa : regra.getPremissas()) {
06.             // Substitui o objeto Variavel criado na conversão do arquivo JSON
07.             // pelo objeto Variavel correspondente da lista de variáveis
08.             premissa.setVariavel(
09.                 getVariavelPeloNome(premissa.getVariavel().getNome()));
10.         }
11.         // Percorre cada objeto Conclusao da Regra
12.         for (Conclusao conclusao : regra.getConclusoes()) {
13.             // Substitui o objeto Variavel criado na conversão do arquivo JSON
14.             // pelo objeto Variavel correspondente da lista de variáveis
15.             conclusao.setVariavel(
16.                 getVariavelPeloNome(conclusao.getVariavel().getNome()));
17.         }
18.     }
19. }

```

O método `atualizarReferencias()` tem como finalidade recuperar as referências perdidas com a recuperação do objeto `BaseConhecimento` a partir do arquivo JSON gerado, pois no processo de conversão do arquivo JSON de volta ao objeto `BaseConhecimento`, não há o reconhecimento no relacionamento da lista de variáveis com as sentenças das regras.

Esse relacionamento é recuperado com o método `getVariavelPeloNome()` nas linhas 8 e 15, que retorna a variável correspondente da lista de variáveis conforme o nome passado por parâmetro.

3.3.1.2 Função para auto completar

A função de auto completar foi atribuída ao campo de variável e valor da tela de cadastro de sentenças (premissas e conclusões). Ela foi adaptada de um exemplo do Stack Abuse (ROBINSON, 2013) que por sua vez foi baseada num exemplo da Oracle (2015) de auto completar palavras em frases dentro de componentes `JTextArea`.

O Quadro 17 mostra a utilização da classe `AutoComplete` que foi adaptada para uso do ambiente. A linha 2 carrega uma lista com todos os nomes das variáveis cadastradas na base de conhecimentos, essa lista é passada como parâmetro para o construtor da classe `AutoComplete` junto com o componente que vai receber o comportamento do auto completar.

Quadro 17 – Trecho de código de utilização da classe `AutoComplete`

```

01. private void carregarTodasVariaveis() {
02.     ArrayList<String> listaNomeVariaveis =
03.         new ArrayList<String>(
04.             Motor.getInstancia().getBase().listaNomeVariaveis());
05.     if (!listaNomeVariaveis.isEmpty()) {
06.         AutoComplete autoComplete = new AutoComplete(
07.             textFieldVariavel, listaNomeVariaveis);
08.         textFieldVariavel.getDocument().addDocumentListener(autoComplete);
09.         textFieldVariavel.getInputMap().put(
10.             KeyStroke.getKeyStroke("TAB"), AutoComplete.COMMIT_ACTION);
11.         textFieldVariavel.getActionMap().put(
12.             AutoComplete.COMMIT_ACTION, autoComplete.new CommitAction());
13.     }
14. }

```

A classe `AutoComplete` é uma implementação da interface `DocumentListener`, e é por isso que ela recebe uma notificação quando há alterações no componente que a registra como tal, como ocorre na linha 8. Por fim, é registrado um atalho que executa a ação de completar uma palavra, conforme linhas 9 e 11. O atalho escolhido foi a tecla TAB, que também alterna entre componentes na tela. Com isso, ao apertar a tecla TAB, a palavra é auto completada e o foco passa para o próximo componente, que neste caso é o campo de valor da tela de sentenças. Ao entrar no campo de valor, a lista de valores cadastrados para a variável selecionada é carregada e então essa lista é atribuída a classe `AutoComplete`, assim como no caso anterior.

3.3.1.3 Ordenação dos caminhos até os objetivos

No momento da execução do *reasoner*, após a criação da árvore de decisão, o quadro negro é instanciado e carrega os caminhos até os objetivos retornados no método

`retornaOsCaminhosDaRaizAteSuasFolhas()` da classe `ArvoreDecisao`, feito isso ocorre o processo de ordenação desses caminhos. Esse processo inicia com a utilização do método `sort()` da classe `Collections` nativa do Java, esse método requer dois parâmetros, a lista que será ordenada e uma implementação da interface `Comparator`. A classe que implementa essa interface foi nomeada `ArrayNoSentencaComp` e seu método sobrescrito é representado no Quadro 18.

Quadro 18 – Trecho de código do método `compare` da classe `ArrayNoSentencaComp`

```

01. @Override
02. public int compare(ArrayList<NoSentenca> o1, ArrayList<NoSentenca> o2) {
03.     if (o1.size() > o2.size()) {
04.         return 1;
05.     } else if (o1.size() == o2.size()){
06.         return comparaItens(o1, o2);
07.     } else {
08.         return -1;
09.     }
10. }

```

Conforme descrito nos requisitos a resolução de conflitos deveria considerar aqueles com menores caminhos até os objetivos, representado pela condição da linha 3. Já no momento da implementação, além desta decidiu-se considerar como segunda condição, quando os caminhos possuem o mesmo tamanho, a ordem em que as regras aparecem na base de conhecimentos (linha 6). O Quadro 19 mostra com detalhes o método `comparaItens()` responsável por esse desempate.

Quadro 19 – Trecho de código do método `comparaItens()`

```

01. private int comparaItens(ArrayList<NoSentenca> o1, ArrayList<NoSentenca> o2) {
02.     // recupera a ordem do último nó (objetivo - só possui uma ordem)
03.     int valor1 = Integer.parseInt(o1.get(o1.size() - 1).getOrdens().get(0));
04.     int valor2 = Integer.parseInt(o2.get(o2.size() - 1).getOrdens().get(0));
05.     // compara a menor ordem e retorna de acordo
06.     if (valor1 > valor2) {
07.         return 1;
08.     } else if (valor1 < valor2){
09.         return -1;
10.     }
11.     return 0;
12. }

```

Primeiro o método recupera a ordem do objetivo de cada caminho (linhas 3 e 4), depois ele compara qual a menor ordem (linhas 6 e 8) e retorna de acordo para que fiquem em ordem crescente na lista (linhas 7, 9 e 11). O Quadro 20 mostra o método `organizaCaminhosAteObjetivos()`, é possível notar que o método `sort()` comentado anteriormente é chamado logo no início (linha 2).

Quadro 20 – Trecho de código do método `organizaCaminhosAteObjetivos()`

```

01. private void organizaCaminhosAteObjetivos() {
02.     Collections.sort(this.caminhosAteObjetivos, new ArrayNoSentencaComp());
03.     for (Variavel variavel : objetivosNaoInstanciados) {
04.         ArrayList<ArrayList<NoSentenca>> caminhosParaMudar =
05.             new ArrayList<ArrayList<NoSentenca>>();
06.         for (ArrayList<NoSentenca> caminho : caminhosAteObjetivos) {
07.             for (NoSentenca noSentenca : caminho) {
08.                 if (noSentenca.getSentenca() instanceof Premissa) {
09.                     if (noSentenca.getSentenca().getVariavel().equals(variavel)) {
10.                         caminhosParaMudar.add(caminho);
11.                         break;
12.                     }
13.                 } else {
14.                     break;
15.                 }
16.             }
17.         }
18.         caminhosAteObjetivos.removeAll(caminhosParaMudar);
19.         ArrayList<NoSentenca> ultimoCaminhoObjetivo = getUltimoCaminho(variavel);
20.         if (ultimoCaminhoObjetivo != null) {
21.             int index = caminhosAteObjetivos.indexOf(ultimoCaminhoObjetivo);
22.             caminhosAteObjetivos.addAll(index + 1, caminhosParaMudar);
23.         }
24.     }
25. }

```

Após a ordenação dos caminhos seguindo os critérios descritos anteriormente (linha 2), o método `organizaCaminhosAteObjetivos()` separa em uma lista (`caminhosParaMudar`) os caminhos que contenham a variável objetivo (que está sendo analisada no laço de repetição da linha 3) nas premissas de suas regras (linhas 4 até 17). Após remover a lista `caminhosParaMudar` da lista principal, ela é inserida novamente após o último caminho com a ocorrência da variável objetivo em alguma conclusão (linhas 18 até 23). Esse procedimento é feito para que todas as regras com variável objetivo em alguma conclusão sejam analisadas pelo *reasoner* antes daquelas que contenham a mesma variável objetivo em alguma premissa do caminho, ou seja, o *reasoner* vai tentar instanciar essa variável objetivo antes de precisar comparar ela em alguma premissa.

3.3.1.4 Execução do *reasoner*

O Quadro 21 mostra o laço de repetição referente ao processo de execução do *reasoner*.

Quadro 21 – Trecho de código do método executar() da classe Motor

```

01. while (!this.quadroNegro.acabouObjetivosNaoInstanciados() && !isCancelado()) {
02.     if (this.quadroNegro.getCaminhosAteObjetivos().isEmpty()) {
03.         if (this.quadroNegro.getObjetivosInstanciadosPorExtenso().isEmpty()) {
04.             JOptionPane.showMessageDialog(null,
05.                 "Nenhum objetivo encontrado!", "Aviso", JOptionPane.WARNING_MESSAGE);
06.         }
07.         break;
08.     } else {
09.         Double cnfCaminho = 0d;
10.         // Verifica o próximo caminho da lista
11.         for (NoSentenca noSentenca : this.quadroNegro.proximoCaminho()) {
12.             Util util = new Util();
13.             /* código omitido */
14.             // Trata uma premissa
15.             if (noSentenca.getSentenca() instanceof Premissa) {
16.                 /* código omitido */
17.                 Premissa premissa = (Premissa) noSentenca.getSentenca();
18.                 /* código omitido */
19.                 ArrayList<String> variavelEValores =
20.                     this.quadroNegro.getVariavelInstanciada(premissa.getVariavel());
21.                 ArrayList<String> ordensRegra =
22.                     this.quadroNegro.getOrdensPossiveis(noSentenca.getOrdens());
23.                 List<String> valoresCNF = null;
24.                 if (variavelEValores == null) {
25.                     if (!premissa.getVariavel().isObjetivo()) {
26.                         /* código omitido */
27.                         TelaPergunta tp =
28.                             new TelaPergunta(premissa.getVariavel(), ordensRegra);
29.                         tp.setVisible(true);
30.                         if (tp.isCancelado()) {
31.                             this.setCancelado(true);
32.                             break;
33.                         }
34.                         /* código omitido */
35.                         variavelEValores = new ArrayList<String>();
36.                         variavelEValores.add(premissa.getVariavel().getNome());
37.                         variavelEValores.addAll(tp.getValores());
38.                         this.quadroNegro.instanciarValores(variavelEValores);
39.                         valoresCNF = variavelEValores.subList(1, variavelEValores.size());
40.                     }
41.                 } else {
42.                     valoresCNF = variavelEValores.subList(1, variavelEValores.size());
43.                     /* código omitido */
44.                 }
45.                 Double cnfAtual = verificarPremissa(premissa, valoresCNF);
46.                 if (cnfAtual == -1d) {
47.                     /* código omitido */
48.                     this.quadroNegro.eliminarCaminhoAtual();
49.                     cnfCaminho = 0d;
50.                     break;
51.                 } else {
52.                     if (cnfCaminho == 0d) {
53.                         cnfCaminho = cnfAtual;
54.                     } else {
55.                         cnfCaminho = util.calcularConjuncao(cnfCaminho, cnfAtual);
56.                     }
57.                 }
58.                 // Trata uma conclusão
59.             } else if (noSentenca.getSentenca() instanceof Conclusao) {
60.                 Conclusao conclusao = (Conclusao) noSentenca.getSentenca();
61.                 ArrayList<String> variavelEValores =
62.                     this.quadroNegro.getVariavelInstanciada(conclusao.getVariavel());
63.                 String valorCNF =
64.                     util.concatenarCNF(
65.                         conclusao.getValorSelecionado(),
66.                         String.valueOf(
67.                             util.calcularConjuncao(

```

```

68.         conclusao.getFatorConfiabilidade(), cnfCaminho));
69.     if (variavelEValores == null) {
70.         variavelEValores = new ArrayList<String>();
71.         variavelEValores.add(conclusao.getVariavel().getNome());
72.         variavelEValores.add(valorCNF);
73.         this.quadroNegro.instanciarValores(variavelEValores);
74.     } else if (conclusao.getVariavel().getTipo()
75.         == TipoVariavel.MULTIVALORADA) {
76.         this.quadroNegro.adicionarValor(conclusao.getVariavel(), valorCNF);
77.     }
78.     if (conclusao.getVariavel().isObjetivo()) {
79.         /* código omitido */
80.         this.quadroNegro.removerObjetivoInstanciado(
81.             conclusao.getVariavel());
82.         if (this.quadroNegro.isUltimaSentencaCaminhoAtual(noSentenca)) {
83.             this.quadroNegro.eliminarCaminhoAtual();
84.             cnfCaminho = 0d;
85.             if (isCancelado()) {
86.                 break;
87.             }
88.         }
89.     } else {
90.         /* código omitido */
91.     }
92. }
93. }
94. }
95. }

```

Os códigos omitidos se referem a construção da árvore de pesquisa. O laço de repetição principal (linha 1) é executado até que não haja mais objetivos a serem instanciados ou até que o usuário cancele o processo de raciocínio. As condições das linhas 2 e 3 verificam se acabaram os caminhos e se nenhum objetivo foi instanciado, caso sejam verdadeiras uma mensagem é mostrada ao usuário avisando que nenhum objetivo foi encontrado e o processo de raciocínio termina, senão o laço de repetição da linha 11 percorre cada nó do próximo caminho e o trata de acordo com a instancia do objeto de seu atributo sentença.

Se a sentença do nó for um objeto `Premissa`, o motor de inferência verifica se sua variável já foi instanciada (linha 19), caso não tenha sido instanciada e não for uma variável objetivo (linhas 24 e 25) o ambiente pergunta ao usuário o valor dessa variável (linhas 27 a 37) por meio da tela de perguntas, após isso ele instancia o novo valor. Com os valores já instanciados o motor de inferência então verifica a veracidade da premissa através do método `verificarPremissa()` (linha 45), que retorna o coeficiente de confiabilidade se a premissa for verdadeira e `-1d`, caso seja falsa. Sendo falsa, o caminho atual é excluído pelo método `eliminarCaminhoAtual()` (linha 48), além de todos os caminhos que possuem alguma premissa com a mesma variável serem verificados e eliminados se necessário. Por fim, o laço de repetição continua para o próximo caminho. Caso a premissa seja verdadeira, o coeficiente de confiabilidade é atualizado (linha 55) e o laço segue para o próximo nó do caminho atual.

Por outro lado, se a sentença do nó for um objeto *Conclusao*, o motor de inferência verifica se sua variável já foi instanciada (linha 61), em caso negativo (linha 69) ele a instancia com o valor atribuído na sentença e a conjunção do coeficiente de confiabilidade dos nós desde o início do caminho até ele (linha 63 e 72). Já em caso positivo, se a variável for do tipo *Multivalorada*, o motor de inferência adiciona o valor a variável (linha 76) calculando a disjunção se esse valor em específico já tenha sido instanciado ou apenas instanciando o valor e atribuindo o coeficiente de confiabilidade. Depois disso, se a variável for um objetivo, o motor de inferência a remove da lista de variáveis objetivos não instanciadas (linha 80), então verifica se esse é o último nó do caminho atual (linha 82) eliminando os caminhos necessários em caso positivo. Se não for o último nó, significa que existe mais de uma conclusão, sendo assim, o motor de inferência apenas continua com o fluxo tratando todas as conclusões até chegar a última.

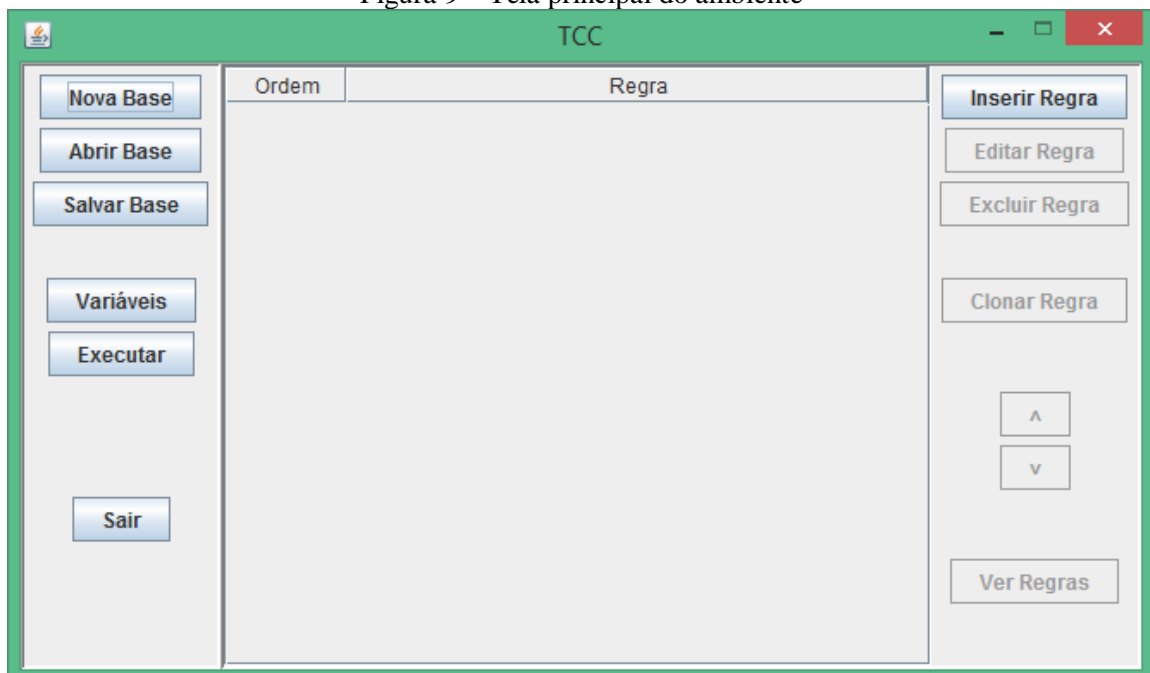
3.3.2 Operacionalidade da implementação

Nesta seção são demonstradas as principais funcionalidades do ambiente e suas diversas telas. O desenvolvimento da interface gráfica foi feito tendo como objetivo oferecer ao usuário uma tela autoexplicativa que permita utilizar seus recursos facilmente.

3.3.2.1 Tela principal

A tela principal do ambiente é constituída por um menu na lateral esquerda acompanhada da lista de regras e seu próprio menu de opções (Figura 9).

Figura 9 – Tela principal do ambiente

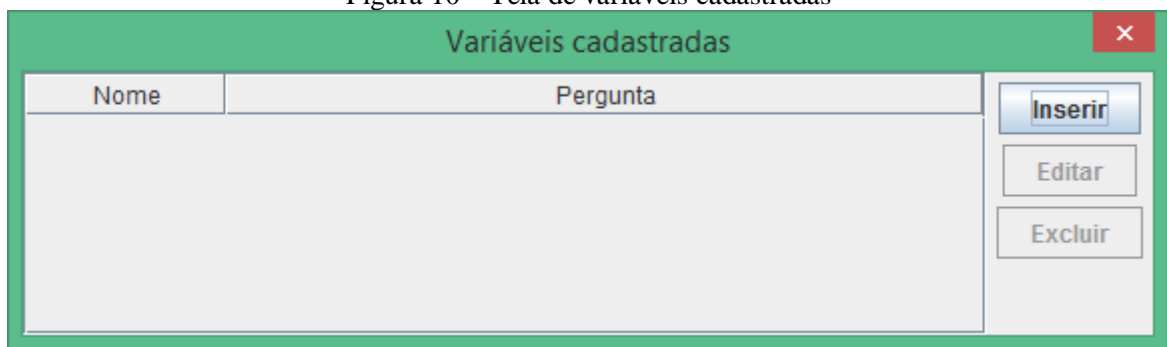


O menu lateral esquerdo da tela principal apresenta um conjunto de botões referentes a persistência da base de conhecimentos (Nova Base, Abrir Base, Salvar Base), um botão para acessar a tela de variáveis (Variáveis), o botão para realizar o processo de raciocínio (Executar) e um botão para sair do ambiente (Sair).

3.3.2.2 Adicionar variáveis

O processo de inclusão de variáveis pode ser realizado de duas maneiras. A primeira é feita através da tela de variáveis (Figura 10) pressionando o botão *Inserir*.

Figura 10 – Tela de variáveis cadastradas



Após pressionar o botão *Inserir*, a tela de cadastro de variável é aberta (Figura 11). Nessa tela é possível informar os dados relevantes à variável como: atribuir um nome, definir a pergunta que aparece no momento da consulta, escolher o tipo (univalorada, multivalorada e numérica), atribuir valores possíveis a ela e definir se é uma variável objetivo ou não, sendo que as informações obrigatórias são o nome, o tipo e os possíveis valores. Ainda, caso a variável seja univalorada e pelo menos o nome seja informado, o ambiente irá gravar as respostas padrões (SIM e NÃO). A segunda maneira de criar uma variável é atribuí-la diretamente na criação de uma sentença como detalhado na próxima subseção.

Figura 11 – Tela de cadastro de variável

3.3.2.3 Adicionar regras

Ao clicar no botão *Inserir Regra* na tela principal, a tela de cadastro de regras é apresentada (Figura 12). Ela é dividida em duas partes, cadastro de premissas e cadastro de conclusões. Os botões de gerenciamento dessas sentenças são apresentados em forma de símbolos para simplificar sua visualização, são eles: + (Inserir), > (Editar) e - (Excluir).

Figura 12 – Tela de cadastro de regra

Uma regra deve obrigatoriamente conter pelo menos uma premissa e uma conclusão. As telas de cadastro de ambas sentenças são ligeiramente semelhantes, ambas possuem campos para inclusão de variável, operador e valor. Porém, diferenças são notadas na tela de cadastro de conclusão, onde pode-se observar o operador fixo (operador de atribuição) e um campo adicional para inclusão do coeficiente de confiabilidade. A tela de cadastro de premissa é apresentada na Figura 13 e a tela de cadastro de conclusão na Figura 14.

Figura 13 – Tela de cadastro de premissa

Figura 14 – Tela de cadastro de conclusão

Os campos de inclusão de variável e valor possuem a função de auto completar em ambas as telas. Além disso, em qualquer um dos dois campos (variável ou valor) é possível digitar uma opção que não foi criada ainda e o ambiente perguntará ao usuário se ele confirma a criação de tal, ou seja, é totalmente possível construir uma base de conhecimentos sem criar nenhuma variável previamente, apenas entrando na tela de variáveis para realizar ajustes como tornar uma variável objetivo. Como falado na subseção anterior em que o ambiente grava as respostas padrões em determinada situação no momento de criação da variável pela tela de cadastro de variável, o ambiente também irá criar essas respostas se o usuário digitar na tela de sentenças qualquer uma das duas (SIM ou NÃO) ao criar um novo valor para a variável.

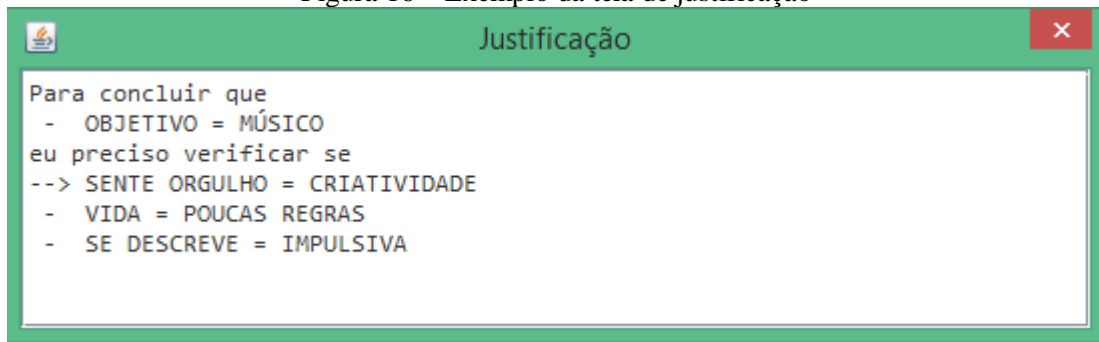
3.3.2.4 Tela de consulta e justificação

Ao clicar no botão *Executar* na tela principal do ambiente, será mostrada uma tela de consulta perguntando ao usuário o valor de alguma variável conforme o exemplo na Figura 15.

Figura 15 – Exemplo da tela de consulta

Na tela de consulta existe um botão chamado ‘Por que?’. Ao clicar nele é exibida uma tela de justificação sensível ao contexto que tenta explicar ao usuário o porquê é necessária aquela variável, conforme pode ser observado na Figura 16.

Figura 16 – Exemplo da tela de justificação

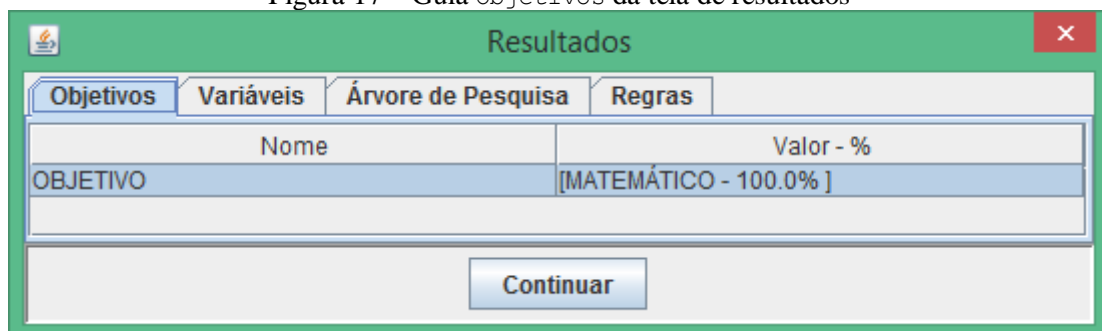


3.3.2.5 Tela de resultados

A tela de resultados é exibida sempre que todas as regras de um objetivo foram analisadas e ele foi instanciado (caso ele seja multivalorado) ou somente quando ele for instanciado (univalorado ou numérico). Ela contém quatro guias, sendo elas:

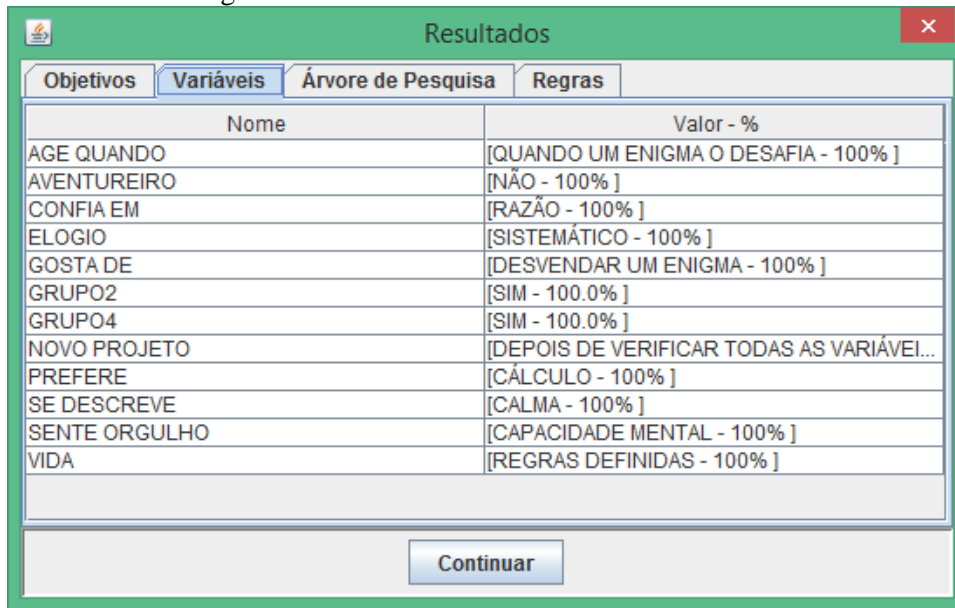
- a) objetivos: mostra a lista de variáveis objetivos e seus respectivos valores acompanhados do coeficiente de confiabilidade (Figura 17);
- b) variáveis: idem ao item anterior, mas somente para variáveis não objetivos (Figura 18);
- c) árvore de pesquisa: apresenta o caminho percorrido pelo ambiente durante o processo de raciocínio (Figura 19);
- d) regras: descrição por extenso e visualmente apresentável de todas as regras da base de conhecimento (Figura 20).

Levando em consideração um sistema especialista de exemplo sobre um teste vocacional que pergunta ao usuário sobre algumas de suas características a fim de chegar a algum objetivo que defina uma profissão para ele, a guia *Objetivos* (Figura 17) apresentaria qual profissão seria a determinada pelas informações fornecidas pelo usuário, conforme Figura 15.

Figura 17 – Guia *Objetivos* da tela de resultados

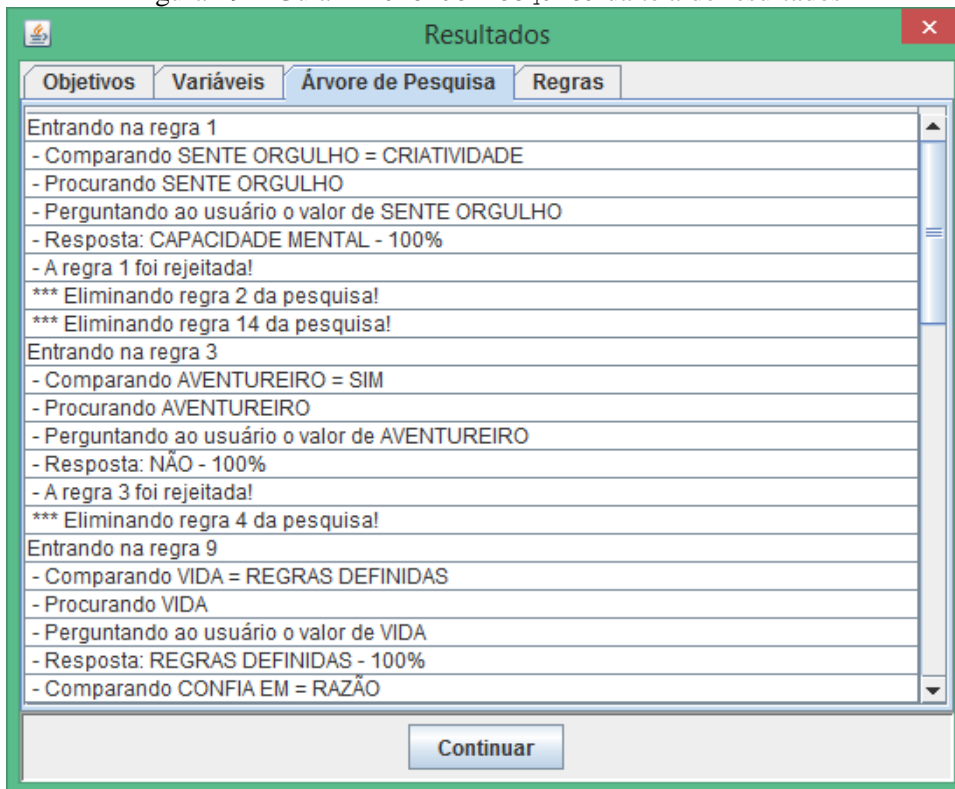
A guia *Variáveis* (Figura 18) apresentaria as informações fornecidas pelo usuário sobre suas características, além de informações inferidas pelo ambiente.

Figura 18 – Guia Variáveis da tela de resultados



Nome	Valor - %
AGE QUANDO	[QUANDO UM ENIGMA O DESAFIA - 100%]
AVENTUREIRO	[NÃO - 100%]
CONFIA EM	[RAZÃO - 100%]
ELOGIO	[SISTEMÁTICO - 100%]
GOSTA DE	[DESVENDAR UM ENIGMA - 100%]
GRUPO2	[SIM - 100.0%]
GRUPO4	[SIM - 100.0%]
NOVO PROJETO	[DEPOIS DE VERIFICAR TODAS AS VARIÁVEI...
PREFERE	[CÁLCULO - 100%]
SE DESCREVE	[CALMA - 100%]
SENTE ORGULHO	[CAPACIDADE MENTAL - 100%]
VIDA	[REGRAS DEFINIDAS - 100%]

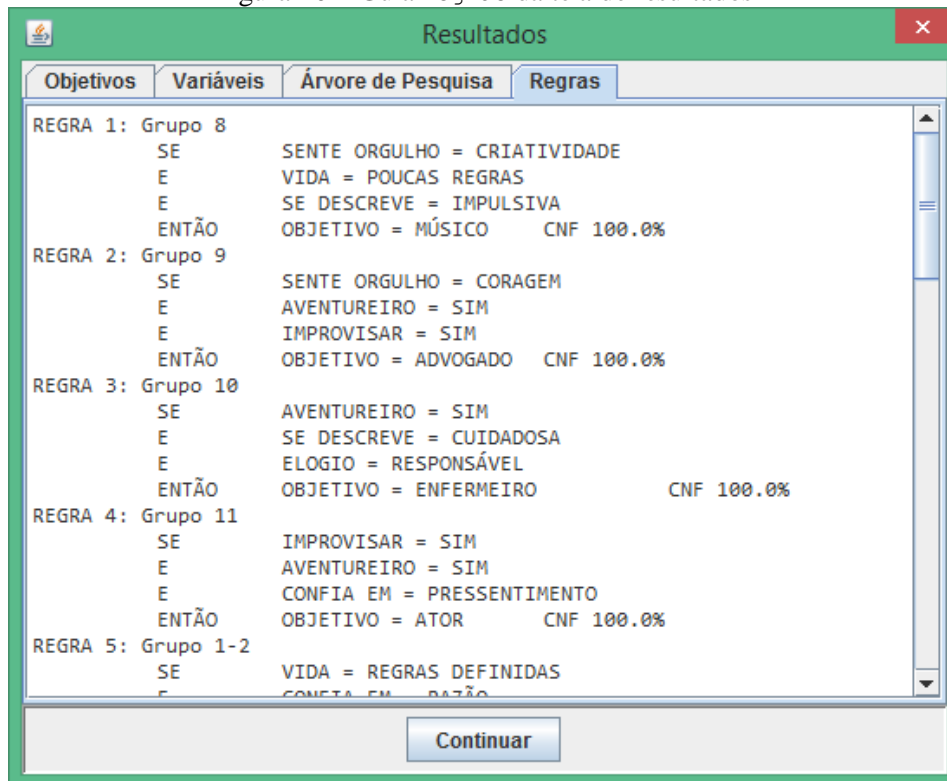
Já a guia *Árvore de Pesquisa* (Figura 19) apresentaria o caminho exato percorrido pelo ambiente durante o processo de consulta ao usuário, incluindo informações de quando uma regra é rejeitada ou aceita, além de informar quais caminhos foram excluídos conforme o ambiente faz uma verificação de todas as regras após terminar de analisar um caminho (seja por aceitação ou rejeição).

Figura 19 – Guia *Árvore de Pesquisa* da tela de resultados


Entrando na regra 1
- Comparando SENTE ORGULHO = CRIATIVIDADE
- Procurando SENTE ORGULHO
- Perguntando ao usuário o valor de SENTE ORGULHO
- Resposta: CAPACIDADE MENTAL - 100%
- A regra 1 foi rejeitada!
*** Eliminando regra 2 da pesquisa!
*** Eliminando regra 14 da pesquisa!
Entrando na regra 3
- Comparando AVENTUREIRO = SIM
- Procurando AVENTUREIRO
- Perguntando ao usuário o valor de AVENTUREIRO
- Resposta: NÃO - 100%
- A regra 3 foi rejeitada!
*** Eliminando regra 4 da pesquisa!
Entrando na regra 9
- Comparando VIDA = REGRAS DEFINIDAS
- Procurando VIDA
- Perguntando ao usuário o valor de VIDA
- Resposta: REGRAS DEFINIDAS - 100%
- Comparando CONFIA EM = RAZÃO

Por fim a guia *Regras* (Figura 20) apresentaria as regras da base de conhecimentos em modo de texto corrido e formatado de maneira que ajude a interpretação do usuário sobre o sistema especialista, principalmente no auxílio da comparação com a árvore de pesquisa gerada.

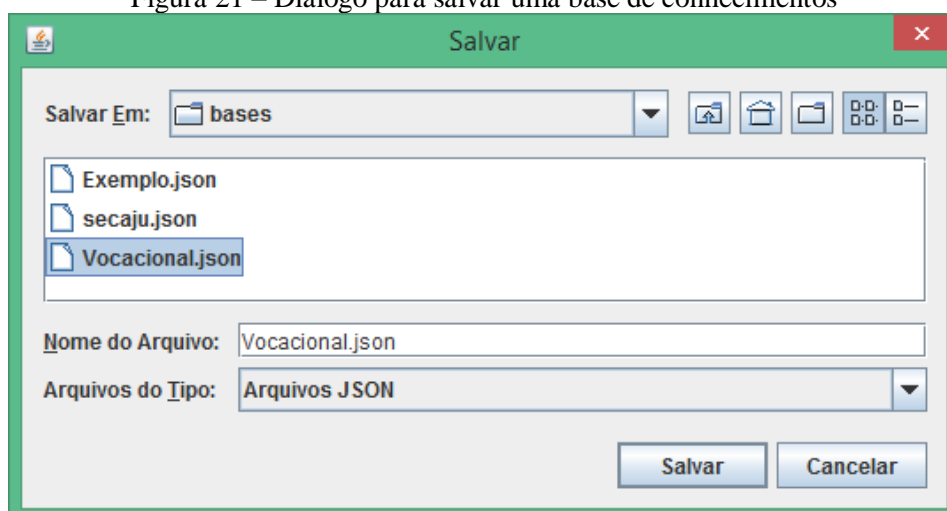
Figura 20 – Guia *Regras* da tela de resultados



3.3.2.6 Salvar e recuperar base de conhecimentos

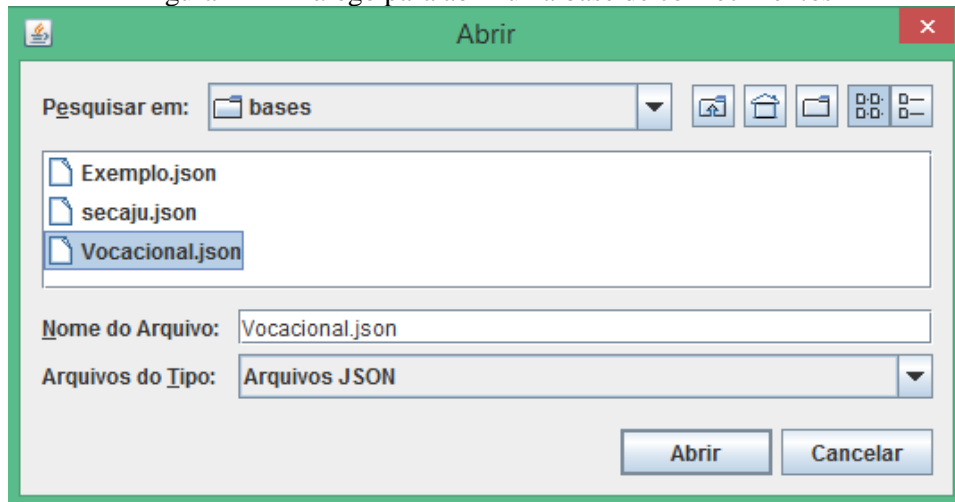
A qualquer momento o usuário pode clicar no botão *Salvar Base* na tela principal que será apresentado o diálogo (Figura 21) com as pastas do computador para que seja escolhido um local e nome para o arquivo.

Figura 21 – Diálogo para salvar uma base de conhecimentos



Do mesmo modo, se existir um arquivo previamente salvo, o usuário pode clicar a qualquer momento no botão *Abrir Base* na tela principal que será apresentado o diálogo (Figura 22) com as pastas do computador para que seja localizado o arquivo da base de conhecimentos salva.

Figura 22 – Diálogo para abrir uma base de conhecimentos



3.4 RESULTADOS E DISCUSSÕES

Com a finalidade de realizar os testes necessários com o motor de inferência do ambiente, foi utilizada uma base de conhecimentos do sistema especialista disponibilizado junto com o download do Expert SINTA (LIA, 1999) chamada de Sistema de Diagnóstico de Pragas e Doenças do Cajueiro (SECAJU). Além disso foi criado um sistema especialista de exemplo para escolha vocacional que serviu para testar outros tipos de casos que o SECAJU não cobriu.

Apesar dos testes demonstrarem êxito, não foi possível realizar testes de desempenho do ambiente com sistemas especialistas maiores. Igualmente não foi viável simular todos os tipos de ocorrências e combinações possíveis visto que a disponibilidade de sistemas especialistas com base de conhecimentos abertas (principalmente os grandes e complexos) é rara, e o desenvolvimento de um novo sistema especialista de qualidade requer muito tempo e esforço, sem levar em conta o domínio do conhecimento necessário.

Algumas limitações foram necessárias, entre as principais estão: a aplicação de apenas um modo de raciocínio; sem opções de resoluções de conflitos adicionais; e descarte do uso do conectivo lógico 'OU' (ainda que seu uso seja pouco comum na construção de sistemas especialistas, seria interessante sua presença para fins didáticos). Essas limitações foram necessárias, visto que esse foi o desenvolvimento da primeira versão do ambiente, de modo

que foram gastos muitos recursos para a criação do módulo de gerenciamento da base de conhecimentos e o módulo do *reasoner*.

3.4.1 Comparativo entre os trabalhos correlatos

O Quadro 22 apresenta as principais características e funcionalidades em comum dos trabalhos correlatos com o ambiente desenvolvido neste trabalho.

Quadro 22 – Comparação com os trabalhos correlatos

características / trabalhos correlatos	ambiente desenvolvido	Expert SINTA	JEOPS
linguagem de programação	Java	Object Pascal (Delphi)	Java
modo de raciocínio	encadeamento para frente	encadeamento para trás	encadeamento para frente
resolução de conflito	menor caminho até um objetivo seguido da característica estática (ordem das regras)	característica estática (ordem das regras)	regra mais recente, regra menos recente, característica estática (ordem das regras)
tratamento de incerteza	fator de confiança	fator de confiança	não possui
representação das regras	modelo geral de representação	modelo geral de representação	sintaxe própria

Ao comparar o modo de raciocínio entre os trabalhos é possível perceber que cada um aplica apenas um tipo. Já ao considerar a resolução de conflito empregada, é consideravelmente notável a flexibilidade do JEOPS, pois além dessas apresentadas no Quadro 22 ele permite que o usuário crie sua própria resolução de conflito ao implementar a interface `jeops.conflict.ConflictSet`. Ainda é possível perceber que apesar dessa flexibilidade na resolução de conflito, o JEOPS não disponibiliza nenhuma estratégia de tratamento de incerteza padrão. Além disso, o ambiente desenvolvido e o trabalho correlato Expert SINTA permitem a representação de regras de uma forma mais amigável ao usuário com menos conhecimento técnico, pois utilizam o modelo geral de representação das regras de produção.

4 CONCLUSÕES

Este trabalho propôs o desenvolvimento de um ambiente para construção de sistemas especialistas baseados em regras de produção. O objetivo principal era criar uma interface amigável com recursos que facilitassem o processo de criação da base de conhecimentos como auto completar variáveis e valores cadastrados, além de permitir a criação de uma regra sem precisar definir nenhuma variável anteriormente.

Esse ambiente foi desenvolvido na linguagem Java, fazendo uso da biblioteca *Swing* por meio dos recursos disponibilizados pelo complemento *WindowBuilder* para criação das interfaces gráficas. Além disso foi utilizada a biblioteca Gson para auxiliar na criação e recuperação da base de conhecimentos, armazenada em um arquivo JSON. Todas essas ferramentas se mostraram eficazes, mesmo considerando a dificuldade que é gerar interfaces gráficas de qualidade com o Java.

Apenas o RNF02 (disponibilizar uma opção para habilitar/desabilitar uma dica para regras que não alcançam nenhum objetivo) não foi atendido, já que houve um atraso no projeto devido a um problema com a unificação do funcionamento com os tipos de variáveis (univalorada, multivalorada e numérica). Observa-se que, futuramente, com a implementação do operador lógico ‘OU’ não seria necessário modificar o modo de raciocínio aplicado, mas somente adicionar esse tratamento na transposição das regras da base de conhecimentos para a árvore de decisão.

Por fim, este trabalho apresentou as funcionalidades bases, com algumas limitações, como abordado anteriormente, de um ambiente para construção de sistemas especialistas que podem servir como ponto de partida para trabalhos futuros, que tem a possibilidade de torná-lo mais completo e flexível. Tendo isso em vista, neste estágio de desenvolvimento, o ambiente mostra-se adequado para ser apresentado como alternativa aos ambientes atuais em aulas que ensinam sobre o assunto de sistemas especialistas e desse modo despertar o interesse de próximos formandos para a continuação do mesmo.

4.1 EXTENSÕES

Como extensão para trabalhos futuros, sugere-se:

- a) incluir o modo de raciocínio de encadeamento para trás;
- b) tornar o mecanismo de resolução de conflitos mais flexível e com mais opções a escolha do usuário;
- c) realizar uma pesquisa e adaptar as telas de acordo com as sugestões;

- d) adicionar um mecanismo de segurança (com senha e criptografia) para a base de conhecimentos;
- e) incluir o conectivo lógico 'OU'.

REFERÊNCIAS

- BITTENCOURT, Guilherme. **Inteligência artificial: ferramentas e teorias**. 3 ed. Florianópolis: Ed. da UFSC, 2006.
- DANTAS, Mario. **Manual do Expert SINTA**. [S.l.], 2010. Disponível em: <<http://mariodantas.wordpress.com/2010/03/10/manual-do-expert-sinta/>>. Acesso em: 12 jun. 2015.
- FERNANDES, Anita M. R. **Inteligência artificial: noções gerais**. Florianópolis: Visual Books, 2003.
- _____. **Inteligência artificial aplicada à saúde**. Florianópolis: Visual Books, 2004.
- FIGUEIRA, Carlos S. **JEOPS: uma ferramenta para o desenvolvimento de aplicações inteligentes em Java**. 2000. 141 f. Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Ciência da Computação, Universidade Federal de Pernambuco, Recife.
- _____. **JEOPS: user's manual**. versão 2.1.2 web, 2007. Disponível em: <<http://www.cin.ufpe.br/~jeops/>>. Acesso em: 18 set. 2014.
- GIARRATANO, Joseph C; RILEY, Gary. **Expert systems: principles and programming**. 2 ed. Boston: PWS Publishing Company, 1994.
- GOOGLE. **Gson**. [S.l.], 2014. Disponível em: <<https://github.com/google/gson/releases/tag/gson-2.3.1>>. Acesso em: 15 abr. 2015.
- HEINZLE, Roberto. **Protótipo de uma ferramenta para criação de sistemas especialistas baseados em regras de produção**. 1995. 145 f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção, Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis.
- KOWALSKI, Robert. **Computational logic and human thinking: how to be artificially intelligent**. New York: Cambridge University Press, 2011.
- LIA, Laboratório de Inteligência Artificial. **Expert SINTA: uma ferramenta visual para criação de sistemas especialistas**. 1999. Disponível em: <<http://www.lia.ufc.br/site/>>. Acesso em: 12 jun. 2015.
- MENDES, Raquel D. Inteligência artificial: sistemas especialistas no gerenciamento da informação. **Ci. Inf.**, Brasília, v. 26, n. 1, 1997. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0100-19651997000100006&lng=en&nrm=iso>. Acesso em: 02 jun. 2015.
- MORGADO, Augusto C. O. et al. **Análise combinatória e probabilidade**. Rio de Janeiro: Sociedade Brasileira de Matemática, 1991.
- ORACLE. **The Java tutorials: how to use text areas**. [S.l.], 2015. Disponível em: <<http://docs.oracle.com/javase/tutorial/uiswing/components/textarea.html>>. Acesso em: 06 abr. 2015.
- RABUSKE, Marcia A. **Introdução à teoria dos grafos**. Florianópolis: Ed. da UFSC, 1992.
- RIBEIRO, Horacio C. S. **Introdução aos sistemas especialistas**. Rio de Janeiro: Livros Técnicos e Científicos, 1987.
- ROBINSON, Scott. **Example: adding autocomplete to JTextField**. [S.l.], 2013. Disponível em: <<http://stackabuse.com/example-adding-autocomplete-to-jtextfield/>>. Acesso em: 06 abr. 2015.

VICTOR, Valcí F. **Sistema especialista para detecção de falhas em comandos elétricos.** 2005. 181 f. Dissertação (Mestrado em Engenharia Elétrica) – Curso de Pós-Graduação em Engenharia Elétrica, Universidade Federal do Rio Grande do Norte, Natal.