UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

MATHEUS BASSI BLANK GONÇALVES

IMPLEMENTAÇÃO DE UM INTERPRETADOR PROLOG COM EXTENSÃO PARA LÓGICA MODAL

FLORIANÓPOLIS/SC
2012.1

MATHEUS BASSI BLANK GONÇALVES

IMPLEMENTAÇÃO DE UM INTERPRETADOR PROLOG COM EXTENSÃO PARA LÓGICA MODAL

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção do título de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Ricardo Azambuja Silveira

FLORIANÓPOLIS/SC
2012.1

MATHEUS BASSI BLANK GONÇALVES

IMPLEMENTAÇÃO DE UM INTERPRETADOR PROLOG COM EXTENSÃO PARA LÓGICA MODAL

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciências da Computação do Curso de Ciências da Computação da Universidade Federal de Santa Catarina e aprovado, em sua forma final, em Julho de 2012

_____

Prof. Dr. Vitório Bruno Mazzola
Coordenador do Curso

Banca Examinadora:

_____

Orientador: Prof. Dr. Ricardo Azambuja Silveira

_____

Prof. Dr. Elder Rizzon Santos

_____

Prof. Dra. Jerusa Marchi

AGRADECIMENTOS

Agradeço aos meus pais por tudo que me proporcionaram para que eu chegasse até este ponto.

À toda minha família por todo incentivo que me foi dado durante todo o curso da graduação.

À Fran, por estar sempre ao meu lado, me apoiando nos meus projetos.

Ao meu orientador neste trabalho, Professor Ricardo Azambuja Silveira, pelo apoio, orientações e sugestões.

Aos membros da banca avaliadora, professores Dr. Elder Rizzon Santos e Dra. Jerusa Marchi pela dedicação de seu tempo à este trabalho.

Aos colegas do laboratório IATE pelas dicas e sugestões.

À todos os excelentes professores do INE que tive o prazer de ser aluno durante a graduação.

RESUMO

O presente trabalho propõe a construção de uma ferramenta para programação em lógica com suporte à Lógica Modal de Primeira Ordem. Para tanto, foi realizado um estudo sobre os fundamentos teóricos da lógica modal assim como também foi feita uma profunda investigação sobre as estruturas e mecanismos implementados nos interpretadores da linguagem Prolog.

Sistemas de Lógica Modal são extensões de sistemas de Lógica Clássica obtidos com a introdução do conceito de modalidades, representadas por operadores modais. Tais operadores podem ser aplicados sobre uma Lógica Proposicional Clássica, resultando em uma Lógica Modal Proposicional, assim como também podem ser aplicados sobre uma Lógica de Primeira Ordem, resultando em uma Lógica Modal de Primeira Ordem.

O uso de modalidades enriquece a expressividade de uma linguagem lógica, tornando mais simples e intuitivo a modelagem de problemas de maior complexidade.

Palavras chave: Programação em Lógica, Lógica Modal, Inteligência Artificial.

ABSTRACT

This paper proposes the construction of a Logic Programming Tool which supports First Order Modal Logic. For that purpose, the foundations of Modal Logic Theory was studied as was also conducted a deep investigation on the structures and mechanisms implemented on Prolog language interpreters.

Modal Logic Systems are extensions of Classical Logic Systems obtained by the introduction of the concept of modalities, which are represented by modal operators. These operators may be applied to a Classical Propositional Logic, resulting a Modal Propositional Logic, as well to a First Order Classical Logic, resulting a First Order Modal Logic.

The use of modalities enriches the expressiveness of a logical language, becoming much simple and intuitive the modeling process of a higher complexity problem.

Key words: Logic Programming, Modal Logic, Artificial Intelligence

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Um dos temas muito estudados no campo da Inteligência Artificial Simbólica é "Sistemas Multi-Agentes". O desenvolvimento de um sistema multi-agente se difere das tradicionais técnicas de desenvolvimento de software oferecendo ao programador um nível mais elevado de abstração. No paradigma de programação orientada a agentes (AOP), a modelagem de uma aplicação consiste em identificar e descrever as entidades autônomas, especialmente chamadas de "Agentes Inteligentes" e suas responsabilidades durante a execução do sistema.

Quando um sistema multi-agente está em execução, os agentes interagem constantemente uns com os outros e com o ambiente onde estão instanciados, buscando realizar as tarefas que lhe são atribuídas. Agentes podem ter níveis variados de "inteligência" [RUS-95] com diferentes dosagens de reflexão e deliberação. Agentes Reflexivos, são os tipos mais simples em um sistema multi-agente e suas atuações no ambiente dependem exclusivamente de suas percepções momentâneas. Já Agentes Deliberativos, mantêm uma representação "mental" do ambiente, atualizada de acordo com suas percepções do mesmo. Desta forma, Agentes Deliberativos não agem por reflexo, mas sim, apresentam raciocínio lógico para justificar suas tomadas de decisão.

Uma importante classe de agentes deliberativos, é a classe dos Agentes BDI. Um agente BDI, do inglês, *Crenças – Desejos – Intenções*, faz intenso uso de deliberação em seu ciclo de execução [BOR-07]. As crenças representam conhecimentos, que o agente pode ter a priori ou adquirir durante seu ciclo de vida por meio da percepção do ambiente e através de comunicações com outros agentes. Os *Desejos* representam os objetivos gerais do agente na aplicação. Quando um agente se compromete com a execução e conclusão de um desejo específico, este é então classificado como uma *Intenção*.

A principal motivação deste trabalho, emerge do intenso uso que é feito da Lógica Formal de Primeira Ordem em sistemas para descrição de crenças, objetivos e planos de agentes BDI em sistemas multi-agentes. Como mostrado na sequência deste trabalho, algumas

lógicas modais podem ser de bastante utilidade para representação de conhecimento de agentes deliberativos em um sistema multi-agente ou ainda, conhecimentos com diferentes "graus de validade" em um sistema especialista baseado em regras.

1.2 OBJETIVO GERAL

O objetivo geral deste trabalho é desenvolver um ambiente de Programação em Lógica que ofereça suporte à Lógica Modal de Primeira Ordem (Prolog Modal). Como anteriormente citado, tal ferramenta pode ser utilizada para o desenvolvimento de aplicações do escopo da Inteligência Artificial fundamentadas em Lógica Formal, como sistemas especialistas de automação e controle, sistemas de suporte a tomada de decisões e sistemas multi-agentes, aumentando a expressividade de discurso para descrição de modelos lógicos.

1.3 OBJETIVOS ESPECÍFICOS

Os objetivos específicos considerados neste trabalho são enumerados abaixo:

- Apresentar os mecanismos do cálculo de predicados computadorizado;
- Apresentar os conceitos fundamentais da Lógica Modal;
- Modelar e Desenvolver em Java um interpretador Prolog;
- Expandir a ferramenta desenvolvida incluindo suporte à Lógicas Modais;
- Exemplificar a aplicação da ferramenta para um problema de *diferentes graus de crença*;
- Exemplificar a aplicação da ferramenta para um problema de *crenças distribuídas em um ambiente multi-agente*;

1.4 JUSTIFICATIVA

As principais justificativas apresentadas para a realização deste trabalho são enumeradas abaixo:

- promover o enriquecimento da expressividade de uma linguagem declarativa de programação em lógica (*Prolog*);

- oferecer compatibilidade: o interpretador desenvolvido neste trabalho é implementado em uma plataforma amplamente conhecida e utilizada (*Java*);

- oferecer expansibilidade: o interpretador desenvolvido neste trabalho é de fácil expansão (predicados especiais podem ser mapeados para código Java);

- aplicável à atuais plataformas de sistemas multi-agente: por ser desenvolvida na plataforma Java e de fácil expansão, a ferramenta pode ser utilizada em conjunto com JADE e Jason em um sistema multi-agentes;

1.5 ESTRUTURA DO TRABALHO

Este trabalho está dividido em 9 capítulos, listados abaixo:

1. **Introdução** – este capítulo fornece uma visão geral do trabalho, apresentando a motivação, o objetivo geral, os objetivos específicos e as justificativas;

2. **Trabalhos Correlatos** – este capítulo apresenta trabalhos previamente existentes que buscam objetivos correlatos aos deste;

3. **Programação em Lógica** – este capítulo introduz os principais conceitos do paradigma de programação lógico, apresentando seus principais conceitos;

4. **Prolog** – este capítulo fornece um estudo sobre a linguagem Prolog, principal representante do paradigma de programação em lógica;

5. **Introdução à Lógica Modal** – este capítulo introduz os fundamentos teóricos da Lógica Modal, apresentando conceitos de modalidades, semântica de Kripke, sistemas base e seus principais axiomas;

6. **Lógica Modal de Primeira Ordem** – este capítulo apresenta os problemas mais comuns na construção de um sistema de Lógica Modal de Primeira Ordem e mostra duas abordagens diferentes de como podem ser tratados;

7. **Desenvolvimento** – este capítulo apresenta um relato de todos os principais elementos desenvolvidos para a ferramenta proposta neste trabalho;

8. **Testes e validação** – este capítulo apresenta resultados obtidos na execução de programas utilizados para teste da ferramenta desenvolvida;

9. **Conclusões** – este capítulo apresenta as conclusões obtidas durante o desenvolvimento deste trabalho apresentando também propostas de trabalhos futuros;

2 TRABALHOS CORRELATOS

A proposta de se construir uma ferramenta para Programação em Lógica com suporte a lógicas não clássicas como lógicas modais, temporais, multivaloradas entre outras não é, de forma geral, uma idéia nova. Existe uma grande variedade de trabalhos correlatos a este que também buscam o mesmo objetivo. Entretanto, a implementação de um sistema para Programação em Lógica Modal não é uma tarefa trivial, devido, principalmente, ao não determinismo presente em alguns mecanismos de resolução e unificação e em algumas classes de programas, o que faz o tema ainda ser de interesse de diversos pesquisadores atualmente.

Os trabalhos correlatos estudados são bastante variados em termos de propostas e também de épocas em que foram produzidos. Por exemplo, dois dos principais trabalhos sobre este tema são de 1986 e 2006 e são, respectivamente, o *Molog* [CER-86], de *Fariñas del Cerro* e o *MProlog* [NGU-06] de *Linh Anh Nguyen*. Diferentes abordagens de como se implementar os principais elementos de uma ferramenta de programação em lógica modal são propostas. De elementos simples como a forma de entrada do programa à complexos algoritmos de resolução e unificação, todos os trabalhos apresentam propostas variadas, com diferentes restrições impostas ao domínio de aplicação.

O trabalho *MProlog* é melhor explicado nos capítulos 6 e 7 (Lógica Modal de Primeira Ordem e Desenvolvimento), onde também são expostas algumas questões teóricas já correspondidas com questões práticas de implementação.

2.1 ABORDAGENS: DIRETA X INDIRETA

Um sistema de lógica modal é um sistema lógico não clássico obtido basicamente pelo acréscimo de modalidades à um sistema de lógica clássica. Daí, surge a principal questão quando se propõe uma linguagem para Programação em Lógica Modal: "*como as Modalidades devem ser tratadas?*

Como resposta à esta pergunta, de fundamental relevância para sequência do processo de modelagem de uma ferramenta para programação em Lógica Modal, tem-se duas abordagens possíveis: a *abordagem Direta* e a *abordagem Indireta*.

Na Abordagem Indireta, as modalidades são representadas no programa lógico através de tradução [BIZ-98], isto é, as modalidades são expressas no programa utilizando-se apenas termos de um sistema de lógica clássica. Nesta abordagem, parâmetros adicionais são incluídos nos predicados, representando os caminhos de um modelo de Kripke [BEN-10] (proposta para representação de modalidades detalhadamente estudada no capítulo 5). Já na Abordagem Direta [MAL-02], as modalidades são tratadas diretamente pelo programa através de uso de novas estruturas sintáticas para sua representação e algumas modificações que são realizadas no algoritmo de unificação para suportá-las corretamente.

Dos trabalhos anteriores, é possível identificar representantes de ambas as abordagens explicadas acima. Da abordagem Indireta, se destacam os trabalhos de *Debart et al.* , que utiliza funções de tradução para tratar modalidades, e de *Ohlbach* [OHL-88], que representa a acessibilidade de mundos como valores adicionais nos predicados.

Da abordagem Direta, se destacam os trabalhos de *Fariñas del Cerro,* o *Molog* [CER-86] e o trabalho de *Linh Anh Nguyen*, o *MProlog* [NGU-05]. A seguir, esses dois trabalhos são introduzidos.

O *Molog* [CER-86], foi efetivamente a primeira implementação de uma ferramenta para programação em Lógica Modal. Desenvolvido em 1985 no Instituto de Pesquisa em Informática de Toulouse (IRIT), a ferramenta se baseia no método de resolução para lógica modal proposto por *Luis Fariñas del Cerro*.

O método de resolução de *Fariñas del Cerro*, consiste em uma extensão do método de resolução clássico com o acréscimo de regras para resolução modal, que definem como as modalidades devem ser tratadas no programa, dependendo da classe de lógica modal a qual o programa pertence. O método implica algumas restrições: as modalidades não podem ser declaradas no corpo das cláusulas e as respostas computadas são limitadas a ‘sim’ / ‘não’.

Originalmente foi programado como uma extensão do interpretador *C-Prolog* e posteriormente adaptado também para outros interpretadores. Atualmente está disponível na forma de um meta-interpretador *Prolog,* e pode ser utilizado com os interpretadores *C-Prolog*, *Quintus-Prolog*, *Sicstus-Prolog* e *SWI-Prolog*. [CER-86].

Apesar de implementado como um meta-interpretador Prolog, quando uma determinada instância Molog é criada para execução de um determinado programa modal, esta não reconhece mais programas Prolog diretamente. No Molog, as cláusulas clássicas do Prolog devem ser executadas através de um predicado especial: "*prolog*" e os operadores ":-" e "," são substituídos por, "<--" e "&", respectivamente.

Assim, o *Molog* estende o Prolog Clássico para que lógicas modais como *Lógicas de Crenças*, *Lógicas Temporais* e *Lógicas de Ações* possam ser tratadas por interpretadores usuais. Também é possível que novas regras sejam definidas pelo usuário, aumentando assim a expressividade dos programas.

Outro representante da abordagem Direta de se tratar modalidades é o *MProlog* [NGU-06]. O trabalho foi desenvolvido por *Linh Anh Nguyen* na Universidade de Varsóvia, na Polônia e publicado em 2006, portanto, 20 anos depois do desenvolvimento do *Molog*, de *Fariñas del Cerro*.

O *MProlog* é uma implementação bastante diferente do *Molog*, já que este utiliza listas para representação de modalidades e uma técnica de Rotulação para a representação de operadores modais Existenciais. Assim, o cálculo de resolução permite indicar para uma determinada meta, a navegação entre os mundos possíveis da estrutura que é feita durante a prova, e não somente 'sim' / 'não' são apresentados como resultado final.

Os símbolos que no *Molog* se diferem do Prolog, como os de *implicação (":-")* e de *conjunção (",")*, no *MProlog* são preservados. Assim, um programa Prolog é também, sintaticamente, um programa *MProlog*. Em outras palavras, um programa Prolog pode ser executado diretamente no *MProlog*, sem a necessidade de predicados especiais para declaração de cláusulas clássicas.

Na sequência deste trabalho, os mecanismos de representação de modalidades, rotulação e unificação propostos por *Linh Anh Nguyen* são melhor explicados nos capítulos 6 e 7, que descrevem os principais elementos implementados neste trabalho.

3 PROGRAMAÇÃO EM LÓGICA

Este capítulo fornece uma breve introdução ao paradigma de programação em lógica e seus conceitos fundamentais para o desenvolvimento deste trabalho.

3.1 PROGRAMAÇÃO EM LÓGICA

Programação em Lógica é um paradigma de programação que se baseia na lógica simbólica para se expressar programas de forma declarativa. Diferente dos tradicionais programas escritos em linguagens de programação imperativa, em um programa lógico não se faz necessário o uso de algoritmos complexos para a descrição de procedimentos repletos de comandos do tipo "*get*", "*set*", "*call*", "*read*", "*write*", etc.

Um programa escrito em uma linguagem estritamente lógica, se resume à uma lista de declarações de objetos e de relações entre si. Desta forma, não é preciso escrever um algoritmo para se obter um resultado. Os resultados são produzidos via inferência lógica [RUS-95].

3.2 CÁLCULO DE PREDICADOS

O cálculo de predicados é o alicerce da programação em lógica. Este é o mecanismo que é utilizado para obtenção de resultados em um sistema lógico. Pode se dizer, ainda, que o cálculo de predicados é o processo que é executado sobre um programa lógico escrito na linguagem da lógica simbólica [CAS-87].

A lógica simbólica, ou lógica formal, como também é referenciada, é um sistema formal constituído por um alfabeto de símbolos e um conjunto de regras gramaticais bem definidas. É utilizada para se representar axiomas e teoremas em um programa lógico de forma não ambígua.

No cálculo de predicados, os axiomas e teoremas, isto é, informações previamente conhecidas, são combinados para se obter novas proposições. Em um sistema lógico, chamamos esse processo de inferência e é estudado mais detalhadamente adiante.

A seguir são apresentados os principais conceitos e características do cálculo de predicados.

3.3 PROPOSIÇÕES

Uma proposição é uma declaração lógica utilizada para se representar objetos e suas relações em um sistema lógico, podendo assumir, na Lógica Clássica, os valores "*verdadeiro*" ou "*falso*" [SCO-99]. Como visto anteriormente, um programa lógico é representado por uma lista de proposições que através da lógica formal, podem ter sua validade verificada.

3.3.1 TERMOS

Os elementos básicos das proposições são chamados de objetos. Um objeto, pode ser representado por um termo simples, que pode ser constante ou variável, ou ainda por um termo composto, quando este formar uma estrutura [SEB-03].

Uma constante é um símbolo utilizado em um programa lógico para representação de um único objeto, imutável durante sua execução. Já uma variável, é um símbolo que pode representar diferentes objetos em diferentes contextos de execução do programa.

Termos compostos são utilizados para representação de estruturas complexas. Representados como uma relação de termos simples, sua notação é similar a notação de funções matemáticas [SCO-99]. De forma genérica, os termos compostos são representados da seguinte forma:

$$functor(arg_0, arg_1, \ldots , arg_n)$$

No modelo acima, functor é o símbolo funcional que representa o nome da relação e a lista de argumentos, representa a n-tupla de objetos, representados por termos, inclusive

termos compostos aninhados é permitido. Nos termos compostos, o número *n* de objetos da relação recebe o nome de aridade.

Em um programa lógico, uma proposição pode ser declarada de três formas distintas: *premissas* e *teoremas*, previamente conhecidos e *hipóteses*, cujo valor verdade deve ser verificado no programa via inferência lógica [SCO-99].

Os termos e as proposições de um programa lógico não carregam nenhuma semântica intrinsicamente. Desta forma, a atribuição de significados às proposições de um programa lógico é de total responsabilidade do programador.

Por exemplo, o termo *gosta(joão,lógica)*, de functor '*gosta*' e aridade *2*, expressa uma relação entre os termos '*joão'* e '*lógica*'. Entretanto, a semântica de '*gosta*' não é expressa formalmente. Assim, o programador tem a liberdade para interpretar o termo como '*joão gosta de lógica*' ou 'a *lógica gosta do joão*'.

3.3.2 PROPOSIÇÕES ATÔMICAS E COMPOSTAS

Em sua forma mais simples, uma proposição recebe o nome de *Proposição Atômica*, que é representada por um único termo, simples constante ou compost [SEB-03]. Já uma proposição composta é definida por duas ou mais proposições atômicas ligadas por conectores e operadores lógicos. Nos programas puramente lógicos, são utilizados os operadores da negação, conjunção, disjunção, implicação e equivalência, mostrados na tabela abaixo.

Operador	Símbolo	Exemplo de uso	Semântica
Negação	¬	¬A	"não" A
Conjunção	∧	A ∧ B	A "e" B
Disjunção	v	A v B	A "ou" B
Implicação	→	A → B	"se" A, "então" B
Equivalência	≡	A ≡ B	A "equivale a" B

Quadro 1: Operadores e Conectivos Lógicos

3.3.3 USO DE VARIÁVEIS

Em um programa lógico, as variáveis são introduzias nas proposições por símbolos especiais que recebem o nome de quantificadores, classificados como universais ou existenciais. [SCO-99] Os quantificadores universais são representados pelo símbolo ∀x e expressam a noção de "para todo elemento x", enquanto que os quantificadores existenciais são representados pelo símbolo ∃x e expressam a noção de "existe algum elemento x". Exemplos de quantificadores são mostrados abaixo.

Semântica	Proposição
todo homem gosta de lógica	∀x gosta(x,lógica)
existe um homem que gosta de lógica	∃x gosta(x,lógica)

Quadro 2: Quantificadores Universal e Existencial

3.3.4 FORMA CLAUSAL

Um crítico problema para o cálculo de predicados computadorizado é a redundância existente na forma de se declarar as proposições [SEB-03]. Uma mesma proposição pode ser declarada de diversas formas, o que torna muito ineficiente o cálculo de predicados por vias computacionais.

Para se minimizar os efeitos causados pela redundância, utiliza-se uma forma padrão sem perda de generalidade para declaração de proposições em um programa lógico, a *Forma Clausal*. Na sintaxe clausal, uma proposição composta é declarada como uma lista de termos antecedentes e uma lista de termos consequentes, como mostrada abaixo.

$$A_0 \wedge A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow B_0 \vee B_1 \vee B_2 \vee B_3 \vee \ldots \vee B_m$$

e pode ser lida como:

"se todos os As forem verdadeiros, ao menos um B também será"

Ao se utilizar a Forma Clausal para declaração de proposições, observa-se algumas características interessantes [SCO-99]: Não é mais necessário o uso de quantificadores

existenciais ou universais, que ficam implícitos na presença ou não de variáveis nos termos e apenas os operadores de implicação e conjunção são necessários para se expressar qualquer sistema.

Exemplo:

Proposição	Forma Clausal
∀x gosta(x,lógica)	gosta(x,lógica)
∃x gosta(x,lógica) {x = joão}	gosta(joão,lógica)

Quadro 3: Exemplo de Forma Clausal

3.4 DEMONSTRAÇÃO DE TEOREMAS

No cálculo de predicados, os axiomas iniciais de um sistema lógico são representados por proposições expressas na forma Clausal. Assim, o cálculo permite, através dos mecanismos de prova, verificar a validade lógica de uma hipótese qualquer no sistema [CAS-87]. Esse processo recebe o nome de demonstração de teoremas e é possível através do mecanismo da resolução.

3.4.1 RESOLUÇÃO

O mecanismo de resolução consiste numa regra de inferência da lógica simbólica utilizada para se obter novas proposições a partir de axiomas previamente conhecidos em um programa lógico [SEB-03]. Estudado pela primeira vez em 1965 por *Alan Robinson*, na *Syracuse University*, o mecanismo se aplica a programas expressos na Forma Clausal da seguinte forma: 1. Une-se os lados esquerdos (consequentes) e direitos (antecedentes) de duas ou mais proposições, resultando em uma nova cláusula formada por lados esquerdos unidos (por OU) + '←' + lados direitos unidos (por E). 2. Remove-se termos que se repetem em ambos os lados.

Abaixo é mostrado um exemplo de resolução. Este exemplo ilustra como é possível se inferir as proposição "*se x é inteligente, então x tem boas notas*" à partir das proposições "*se x for bom aluno, então x tem boas notas*" e "*se x for inteligente, então é x bom aluno*".

proposição 1	boas_notas(x) ← bom_aluno(x)
proposição 2	bom_aluno(x) ← inteligente(x)
resolução passo 1	boas_notas(x) v bom_aluno(x) ← bom_aluno(x) ∧ inteligente(x)
resolução passo 2	boas_notas(x) ← inteligente(x)

Quadro 4: Exemplo de Resolução

3.4.2 UNIFICAÇÃO E INSTANCIAÇÃO

No processo de resolução, os termos que se repetem em ambos os lados da implicação devem ser removidos. Para isso, todos os termos devem ser comparados entre si, para que sua equivalência seja verificada. O processo de comparação de termos é conhecido por Unificação. Dizer que dois ou mais termos unificam, significa dizer que os termos são iguais entre si, ou que representam os mesmos objetos ou rolações.

A unificação pode inicialmente parecer um procedimento trivial, mas nem sempre o é. A questão complicada da unificação é quando há presença de variáveis nos termos que devem ser comparados. Para se proceder a comparação, variáveis devem ser resolvidas, ou seja, instanciações devem ser feitas.

O processo de instanciação consiste em atribuir valores temporários à variáveis buscando promover o sucesso de uma unificação de termos [CAS-87]. Porém é bastante comum durante o processo de resolução, que instanciações feitas sejam descartadas por não resultarem em unificações bem sucedidas.

Abaixo é mostrado um exemplo de resolução com unificação e instanciação:

premissa 1 (1)	q(x) ← p(x)
premissa 2 (2)	p(0) ←
hipótese	← q(0)
passo 1: unifica q(0) com q(x), instancia x = 0 em (1)	q(0) ← p(0)
passo 2: une-se os consequentes (v) e antecedentes (∧):	q(0) ← p(0) ∧ q(0)
passo 3: remove-se termos repetidos	← p(0)
Passo 4: unifica p(0) com a premissa (2)	{ }

Quadro 5: Exemplo de Resolução com Instanciação

Partindo-se das premissas “se p(x) é válido, então q(x) é válido” e “p(0) é válido”, pode-se provar a validade da hipótese “q(0)” no programa, instanciando a variável x com o valor 0: { x / 0 }.

3.4.3 CLÁUSULAS DE HORN

Na programação em lógica, a Forma Clausal foi escolhida para a declaração de proposições visando minimizar os problemas decorrentes da redundância (formas distintas de se expressar uma mesma proposição). Porém, para se viabilizar o processo de resolução computacional, mais uma restrição é aplicada à *Forma Clausal* anteriormente explicada: um único termo pode ser declarado como consequente em uma proposição, e não mais uma lista de possíveis consequentes. Essa *Forma Clausal* especial recebe o nome de *Cláusulas de Horn* em homenagem ao matemático *Alfred Horn*, que as estudou.

$$B \leftarrow A_0 \wedge A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

Nas Cláusulas de Horn, o lado esquerdo (consequente) é constituído de apenas um termo e recebe o nome de “cabeça da cláusula”, enquanto o lado direito (lista de antecedentes) é constituído de uma lista de termos, potencialmente vazia [SCO-99].

4 PROLOG

Neste capítulo, é introduzida a linguagem Prolog, a principal representante do paradigma lógico. Apresenta-se desde suas origens à alguns detalhes de suas implementações.

4.1 ORIGENS

No início da década de 70, o desenvolvimento da linguagem Prolog foi um projeto comum entre duas universidades europeias. Na França, na *Universidade de Marselha*, o projeto era desenvolvido por *Alain Colmerauer* e *Phillipe Roussel*, que pesquisavam processamento de linguagens naturais, enquanto na Escócia, na *Universidade de Edimburgo*, a pesquisa era liderada por *Robert Kowalsky*, cujo interesse era desenvolver um provador automático de teoremas.

O Prolog original, desenvolvido em parceria das duas universidades de Marselha e Edimburgo, ficou conhecido como *Projeto Fundamental do Prolog*. Nos meados da década de 70, o projeto tomou rumos diferentes em cada uma das duas universidades, resultando em variações de dialetos com sintaxes distintas.

Durante toda a década de 70, as pesquisas sobre o paradigma de programação em lógica e linguagens para programação em lógica estavam concentradas em Marselha e Edimburgo, até que em 1981 o governo japonês inicia um grande projeto de pesquisa entitulado *Fifth Generation Computing System (FGCS)*, objetivando o desenvolvimento de máquinas inteligentes e utilizando o Prolog como base [RUS-95].

O anúncio do projeto despertou um grande interesse global em Inteligência Artificial e Programação em lógica, promovendo o surgimento de novas pesquisas nessas áreas nos Estados Unidos e em diversos países europeus.

4.2 ELEMENTOS

A sintaxe de Edimburgo é hoje o dialeto Prolog mais utilizado. Sua primeira implementação é do ano de 1979, concebida por *Warren et al*. [WAR-77]. A seguir são mostrados os principais elementos e características do Prolog de Edimburgo.

4.2.1 TERMOS

Como visto no capítulo 3, as proposições que formam um programa lógico são declaradas através de termos, que representam objetos e suas relações. No Prolog, esses termos podem ser de tipos Constantes, Variáveis e Estruturas.

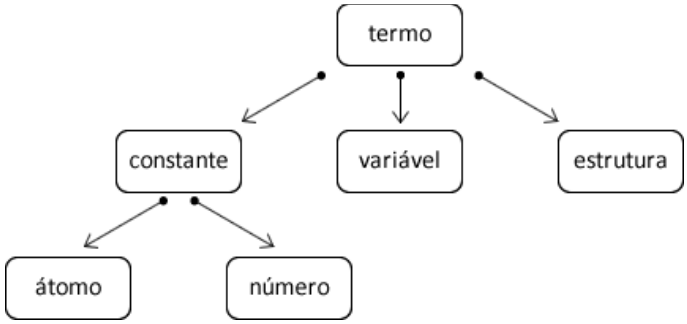


Figura 1: Termos Prolog

Uma constante pode ser um átomo ou um número. Um átomo é um símbolo definido como uma cadeia de caracteres iniciados por letras minúsculas e é utilizado para a representação de objetos.

Define-se uma variável como um símbolo que pode representar diferentes termos em diferentes contextos. Uma variável é dita instanciada no momento em que a mesma está representando algum outro termo, constante ou estruturado. No Prolog, variáveis são representadas por cadeias de caracteres iniciadas por letras maiúsculas.

Por fim, os termos estruturados, ou simplesmente estruturas, são formados por um símbolo funcional, chamado de functor seguido de uma tupla de argumentos. No Prolog, as estruturas representam as proposições atômicas do cálculo de predicados e são representadas da seguinte forma: $functor(arg_0, arg_1, ..., arg_n)$.

Exemplo de uma estrutura no Prolog:

aluno(nome(matheus), matricula(07132052, universidade(ufsc), curso(cco))).

4.2.2 CLÁUSULAS

Num programa escrito em Prolog não se espera encontrar instruções de baixo nível, tipicamente utilizadas para descrição de procedimentos. No Prolog, o conjunto das proposições declaradas pelo programador é que formam o conjunto de instruções do programa [CAS-87]. Estas são expressas como *Cláusulas de Horn*, e são divididas em três classes: *Fatos*, *Regras* e *Consultas*.

Um *fato* é uma proposição que representa uma informação presumida, um axioma no sistema lógico. É representado por uma Cláusula de Horn sem corpo, apresentando apenas a cabeça da cláusula.

Exemplo de fato: “o ano de descobrimento do Brasil é 1500”:

descobrimento(brasil, 1500).

Uma *regra*, é também uma proposição presumida utilizada para se representar teoremas. Representada por uma Cláusula de Horn completa, com cabeça e corpo, uma regra Prolog permite que novos conhecimentos sejam inferidos à partir dos axiomas iniciais [SEB-03]. Se todas as proposições antecedentes (corpo da regra) forem válidas, então pode-se assumir a proposição consequente (cabeça da regra) como também verdadeira.

Exemplo de regra: “toda pessoa que nasce no Brasil é brasileira”

brasileiro(X) :- nasceu_em(X,brasil).

No Prolog, a implicação (←) é introduzida nas regras pelo símbolo ‘ :- ‘.

A terceira e última classe de instruções Prolog são as Consultas. Uma *consulta* expressa uma meta de prova. Representa uma pergunta que é feita ao sistema sobre a validade de uma dada hipótese. O sistema responde Sim, se a hipótese é válida naquele ambiente, ou Não, caso a meta ou alguma submeta não possa ser provada. Quando há variáveis envolvidas em uma consulta, o sistema apresenta ainda as instanciações realizadas durante o processo de resolução.

Exemplo de consulta (1): "joão é brasileiro?"

?- brasileiro(joão).

Exemplo de consulta (2): "quem é o pai de joão?"

?- pai(X,'joão').

No Prolog, consultas sem a presença de variáveis, como no exemplo 1, são resolvidas diretamente e o programa simplesmente responde 'Sim' ou 'Não' quanto a validade da meta. Entretanto, quando há variáveis envolvidas, como no exemplo 2, o sistema tenta ainda encontrar unificações que tornem válida a consulta. Assim, se no sistema existisse, por exemplo, um fato *pai('mário','joão'),* a consulta 2 retornaria 'Sim', mostrando as instanciações feitas { x / 'mário' }.

4.2.3 LISTAS

No Prolog, assim como as proposições atômicas, uma lista também é uma estrutura de dados básica. Uma lista consiste em uma sequência de termos, potencialmente de naturezas distintas, como átomos, números, estruturas e até mesmo outras listas [WAR-77]. No Prolog, uma lista é expressa de forma convencional: elementos entre colchetes e separados por vírgulas.

Exemplo de lista no Prolog: *[zero,1,2,3,quatro,cinco(X)]*

Observe diferentes tipos de termos presentes na lista: *átomos* { zero, quatro }, *números* { 1, 2, 3 } e até mesmo *estruturas* { cinco(X) }.

Para construção e manipulação de uma lista em um programa Prolog, não existem funções específicas como "crie nova lista", "pegue elemento", "insere elemento", etc. A partir de uma lista qualquer, representada como [a_1, a_2, a_3, ... ,a_n], a mesma pode ser manipulada utilizando a notação [Cabeça|Cauda]. *Cabeça* representa o primeiro elemento da lista, enquanto que a *Cauda* representa uma outra lista formada pelos n-1 elementos seguintes.

Os exemplos a seguir mostram algumas operações envolvendo listas.

1. Declaração de uma lista vazia:

```
nova_lista([]).
```

2. Inserir elemento no início da lista:

```
inserir([],Elemento,[Elemento]).
inserir(Lista,Elemento,[Elemento|Lista]).
```

3. obter elemento na posição P da lista:

```
getElemento([Cabeca|Cauda],0,Cabeca).
getElemento([Cabeca|Cauda],P,Elemento) :- Q is P-1,
getElemento(Cauda,Q,Elemento)
```

4.1 INFERÊNCIA

O processo de inferência, é um dos elementos mais importantes do cálculo de predicados. Quando uma consulta é estabelecida, a mesma é colocada em uma pilha de metas e o procedimento da resolução é iniciado para tentar prová-la. Se o sistema consegue encontrar uma sequência de regras, que quando ativadas, ligam a consulta à fatos declarados, a consulta é dita bem sucedida e assim provada verdadeira [CAS-87]. Porém, se o sistema não puder prová-la, a consulta é dita falha.

A resolução de consultas pode se tornar um processo complicado na presença de variáveis, quando o mecanismo de instanciação deve ser constantemente ativado para se buscar unificações possíveis.

4.1.1 FORMAS DE RESOLUÇÃO

O mecanismo da resolução consiste em encontrar um caminho que ligue uma consulta submetida a apenas fatos inicialmente declarados no programa lógico [WAR-77]. Na teoria da lógica formal, a forma como este mecanismo é implementado não é determinante. Porém, num projeto real de um interpretador Prolog, o desenvolvedor deve optar por uma das duas formas de implementação: *Encadeamento Progressivo* ou *Encadeamento Retrógrado*.

No *Encadeamento Progressivo*, também conhecido por resolução *bottom-up*, o processo de resolução é executado de baixo para cima, isto é, a partir da base de dados, busca-se provar uma consulta [SCO-99]. Novas proposições são obtidas gerando-se combinações de fatos e regras até que se encontre, entre as combinações geradas, uma que seja equivalente à meta submetida na consulta.

A figura abaixo ilustra o processo bottom-up de se executar uma consulta. À partir dos fatos p(0), p(1) e da regra q(X) :- p(X), as combinações q(0) :- p(0) e q(1) :- p(1) são geradas, provando também os fatos q(0) e q(1).

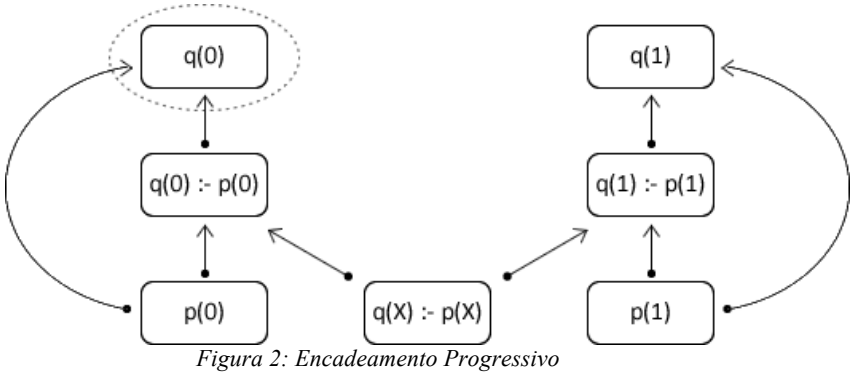


Figura 2: Encadeamento Progressivo

Já no *Encadeamento Retrógrado*, também conhecido por resolução *top-down*, o inverso ocorre. O sistema inicia a busca a partir da consulta, realizando sucessivas

substituições de submetas por corpos definidos em regras cujas cabeças unificam até que todas as submetas na pilha de prova unifiquem somente com fatos inicialmente declarados no programa.

A figura abaixo ilustra a execução de uma consulta de forma top-down. À partir do objetivo de prova q(0), e de uma regra do programa, q(X) :- p(X), o interpretador consegue chegar ao termo p(0), que é declarado como um fato no programa.

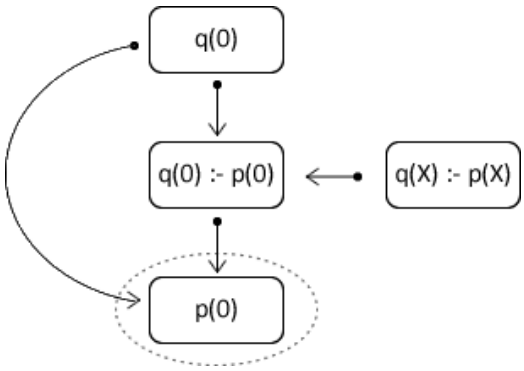


Figura 3: Encadeamento Retrógrado

Outra importante questão no projeto de um interpretador Prolog em relação à implementação do mecanismo de resolução é a forma de se tratar consultas formadas por mais de uma submeta. Deve-se optar entre buscar *primeiramente em Profundidade* ou buscar *primeiramente em Largura*. Na busca em Profundidade, as submetas são provadas uma por vez, enquanto na busca em Largura, todas as submetas são trabalhadas em paralelo, o que demanda maior uso de memória [SEB-03].

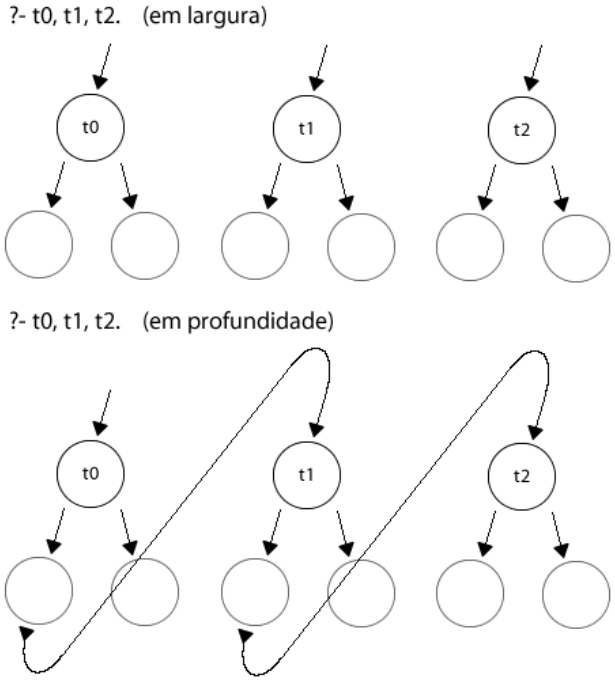


Figura 4: Busca em Profundidade X Busca em Largura

4.1.2 ENCADEAMENTO RETRÓGRADO E BACKTRACKING

As implementações de Prolog mais conhecidas utilizam Encadeamento Retrógrado com Busca primeiramente em Profundidade. Dessa forma, é bastante comum durante o processamento de uma consulta que uma primeira submeta unifique para um determinado conjunto de instanciações enquanto que submetas seguintes não unifiquem para o mesmo conjunto [RUS-95]. Quando isto ocorre, o sistema deve então retroceder à submetas anteriormente provadas, abandonar instanciações préviamente feitas e continuar buscando novas instanciações que resultem em uma unificação. Esse procedimento recebe o nome de Backtracking.

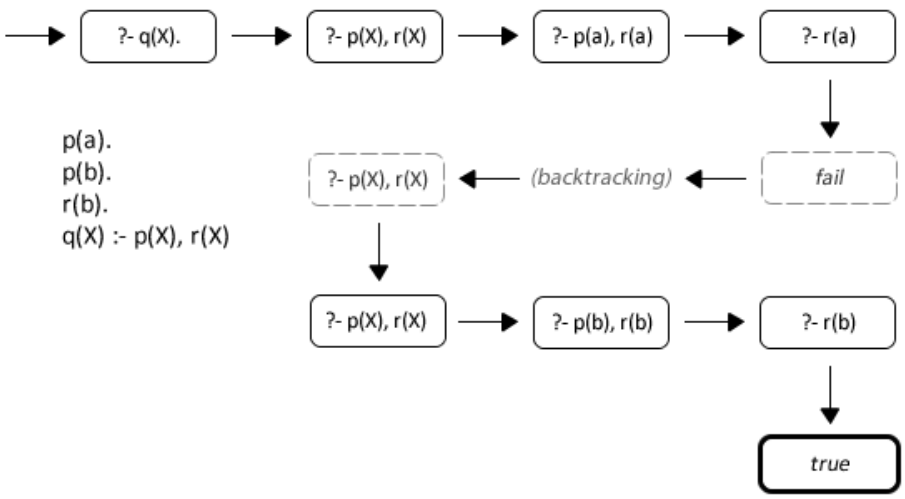


Figura 5: Exemplo de Backtracking

4.1.3 CONTROLE DE FLUXO

Diferente de um ambiente estritamente lógico, é muito importante que o programador tenha ciência de como o mecanismo de resolução é implementado no Prolog. Lembrando que o sistema percorre o banco de fatos e regras sequencialmente, a ordem em que essas instruções são declaradas é de grande relevância [SCO-99]. Uma simples inversão na ordem em que duas proposições são declaradas é o suficiente para tornar a computação mais lenta em alguns casos. Em outros, pode até mesmo resultar em recursões infinitas.

Exemplo: Seja p um programa formado pelas seguintes cláusulas:

p(0).
p(X) :- p(X - 1).

O programa expressa que a propriedade p pode ser observada para o valor 0, e ainda, que p pode ser observada para um valor X, se p puder ser observada para seu antecessor. Para X > 0, p(X) resulta *true*. Porém, se X < 0, o programa entra em uma recursão infinita: *p(-1) :- p(-2) :- ... :- p(-infinito)*.

4.1.4 OPERADOR CUT

Outro importante recurso que o Prolog oferece para o programador possa ter maior controle sobre o fluxo de execução durante o processo de resolução é o mecanismo de *Backtracking*. Esse recurso é automaticamente habilitado quando se executa uma consulta. Entretanto, existe a possibilidade do programador desabilitar o backtracking a partir de um determinado ponto da execução de uma cláusula, o que é de grande utilidade em alguns casos e estratégias de busca específicas. Através de um operador especial, o *Cut*, o backtracking pode ser desabilitado no escopo de uma consulta.

O operador *Cut* é introduzido pelo símbolo exclamação (!) e pode aparecer em uma cláusula como qualquer outra submeta. Sua semântica é a seguinte: quando um operador *Cut* aparecer no topo da pilha de submetas, o mesmo é provado como verdade e o programa continua tentando provar as submetas seguintes [SEB-03]. No entanto, se uma submeta seguinte falhar e o mecanismo de backtracking for ativado, as submetas são então desempilhadas até que se encontre novamente o operador *Cut* no topo da pilha. A partir desse momento, o backtracking é interrompido, e a consulta retorna *falha*.

Exemplo:

d :- a, !, b, !, c.

No exemplo acima, a proposição *d* é válida se as proposições *a, b e c* também forem. O mesmo resultado seria obtido com a regra: *d :- a, b, c.* A notável diferença entre essas duas regras, é a presença do operador *Cut* (!), que neste exemplo, impede a ação do backtracking ao falhar a prova de *c* ou de *b*.

4.2 ARITMÉTICA

No Prolog, valores numéricos podem ser representados como Termos nas proposições, assim como operações aritméticas também são suportadas. Expressões numéricas podem ser expressas utilizando-se constantes numéricas, variáveis e

operadores básicos (soma, subtração, multiplicação e divisão). O resultado de uma expressão numérica pode ser atribuído a uma variável através do operador *is*.

O operador *is*, sempre aparece acompanhado de uma variável à sua esquerda e uma expressão aritmética à sua direita com duas restrições: a variável que aparecer à esquerda, a qual vamos atribuir o valor resultante da expressão, nunca pode estar instanciada no momento em que a cláusula for executada, ou esta falhará, assim como a presença de variáveis não instanciadas no lado direito do operador também provocará falha na execução [SEB-03]. Caso contrário, a cláusula é bem sucedida, e no momento em que resultar *true*, a variável do lado esquerdo estará instanciada com o valor resultante da expressão.

Exemplos:

1. somar C = A + B:

   soma(A,B,C) :- C is A+B.

2. (Contra Exemplo) incrementar o valor da variável X em uma unidade:

   inc(X) :- X is X+1.

O primeiro exemplo será bem sucedido e depois da execução da cláusula, a variável C estará, de fato, instanciada com o valor da soma de A e B. Entretanto, o segundo exemplo falhará, por que a variável X não poderia estar instanciada no momento da atribuição do operador *is*, e nesse caso X já está instanciada.

4.3 MUNDO FECHADO E O PROBLEMA DA NEGAÇÃO

A base de fatos e regras declaradas pelo programador representa tudo que é conhecidamente verdade em um programa lógico. Qualquer outra informação que não esteja inicialmente declarada é então dita desconhecida. Dessa forma, qualquer hipótese que possa ser provada em um programa Prolog é de fato verdade, enquanto que nem toda consulta que falhe é necessariamente falsa, devido à possibilidade de que com alguma informação desconhecida a meta pudesse ser provada [SCO-99]. Esse modelo é

conhecido como *Pressuposição de Mundo Fechado* e por isso o Prolog é enquadrado como um sistema *verdadeiro/falha* e não um sistema *verdadeiro/falso*.

A natureza de mundo fechado em que se baseia a resolução Prolog distorce o verdadeiro significado do operador *not* definido na lógica simbólica. Num sistema de lógica formal, para uma proposição p qualquer, temos que *not(not(p)) é equivalente a p*, o que não é verdade no Prolog. No Prolog, o operador *not* tem outra semântica: *not(p)* resulta verdade, se uma proposição *p* não puder ser provada pelo sistema, ou seja, se *p* falhar. O exemplo abaixo mostra uma situação em que uma consulta Prolog falha, assim como sua negação.

Exemplo: considerar um programa formado pelas seguintes cláusulas:
p(a).
p(b).

Neste caso, uma consulta *p(c)* falha, já que não pode ser provada, assim como *not(p(c))* também falha, pois também não pode ser provada.

4.4 TRACING

O fluxo de execução do processamento de uma consulta pode ser detalhadamente monitorado pelo programador através do mecanismo de rastreamento, ou *Tracing*. Quando ativado, o interpretador Prolog emite mensagens a cada passo que é realizado na resolução. Cada vez que o sistema tenta satisfazer uma submeta uma mensagem do tipo *call* (1) é emitida. Para cada submeta que é satisfeita, o programa emite uma mensagem do tipo *exit* (2). Quando o *backtracking* é ativado para se tentar satisfazer novamente alguma submeta já provada por outro caminho, o sistema sinaliza uma mensagem do tipo *redo* (3). E por fim, uma mensagem do tipo *fail* (4) é emitida quando uma submeta falha [SEB-03].

Exemplo: no processamento da consulta *soma(6,11,R)*, o Prolog produziria o seguinte trace:

```
1       call    soma(6,11, _0)?
2       call    _0 is 6 + 11?
2       exit    17 is 6 + 11
1       exit    soma(6,11,17)
R = 17
```

5 INTRODUÇÃO À LÓGICA MODAL

Neste capítulo, é realizado um estudo sobre os fundamentos da Lógica Modal Proposicional, sistemas de Lógica Modal, Semântica de Kripke, satisfação, caracterização por estruturas e modelo.

5.1 FUNDAMENTOS

Desde os tempos de Aristóteles, quando se buscava formalizar um instrumento para representação e estudo sistemático do raciocínio via silogismos, que originaria a ciência hoje conhecida como Lógica, já existia um questionamento sobre a Lógica Modal. A palavra modal se deve à distinção estabelecida na filosofia entre potência e ato e se refere ao modo de como uma sentença é avaliada [COS-08]. Uma sentença modal não é somente avaliada como verdadeira ou falsa, mas também é verificada quanto a possibilidade (em potência) e necessidade (em ato).

Assim, a Lógica Modal consiste em um sistema lógico não clássico, que permite expressar conceitos de necessidade e possibilidade em suas sentenças [GIR-95]. Quando uma sentença pode ser verificada como verdadeira em todas as situações, esta é dita *necessária*. Uma sentença é dita *possível*, se a mesma é válida em pelo menos uma situação. Quando uma sentença não pode ser verificada em nenhuma situação, esta é dita *impossível*. Por fim, para uma sentença que não seja necessária e nem impossível, pode-se ainda, dizer que a mesma é *contingente* [MAL-02]. Os conceitos de *necessidade*, *possibilidade* e *contingência* são introduzidos nas fórmulas modais por operadores especiais mostrados abaixo.

Operador	Símbolo	Semântica
Necessidade	$\Box p$	p é válido em todas as situações
Possibilidade	$\Diamond p$	p é válido em alguma situação
Contingência	∇p	$\Diamond p \wedge \Diamond \neg p$

Quadro 6: Operadores Modais

Os primeiros estudos referentes à Lógica Modal, concentravam-se nas modalidades de *necessidade* e *possibilidade*, que ficaram conhecidas como *modalidades Aléticas*. Entretanto, existem também outras variações de sistemas lógicos modais para

se trabalhar com outros tipos de modalidades. Por exemplo, *lógicas epistêmicas* trabalham com conceitos de certeza nas sentenças como "*certamente é*" ou "*possivelmente é*". *Lógicas temporais* trabalham com conceitos de tempo, como "*sempre será*", "*será*", "*sempre foi*" e "*foi*". *Lógicas deônticas* trabalham com conceitos de dever e permissão, como "*é obrigatório*", "*é permitido*" e "*é proibido*". Esses são apenas alguns dos exemplos. Ainda é possível se definir inúmeras outras lógicas modais através da composição de operadores primitivos [COS-08].

5.2 CONSTRUÇÃO DE SISTEMAS DE LÓGICA MODAL

Uma Lógica Modal consiste em um sistema formal que estende uma Lógica Clássica com o acréscimo de modalidades. Desta forma, um sistema de *Lógica Modal Proposicional* é obtido através do acréscimo de modalidades sobre a Lógica Proposicional Clássica, assim como um sistema de *Lógica Modal de Primeira Ordem* pode ser obtido através do acréscimo de modalidades sobre a Lógica de Primeira Ordem [BIZ-98].

Um importante exemplo de um sistema de lógica modal é a *Lógica Modal Alética*, obtida através da inclusão das modalidades de "*necessidade*" e "*possibilidade*" sobre lógicas clássicas. A Lógica Modal Alética é a principal representante das lógicas modais, sendo muitas vezes confundida como a única, já que os conceitos de "*necessidade*" e "*possibilidade*" são os mais clássicos exemplos para se ilustrar o conceito de modalidades.

5.2.1 OPERADORES MODAIS

Os *operadores modais* são os símbolos especiais utilizados para representação das modalidades nas fórmulas de um sistema de Lógica Modal [GIR-95]. Por exemplo, a modalidade de necessidade é introduzida pelo símbolo $\Box$, que pode ser lido como "é necessário que", enquanto a modalidade de possibilidade é introduzida pelo símbolo $\Diamond$ e pode ser lido como "é possível que". Existe também uma terceira modalidade chamada de contingência. Uma proposição p é dita contingente quando $\Diamond p \wedge \Diamond \neg p$.

É importante notar, neste contexto, que os 3 operadores modais apresentados podem ser resumidos em apenas um operador primário com uso de negações. Dizer que uma fórmula "é necessária", é o mesmo que dizer que sua negação não é possível: $\Box A \leftrightarrow \neg\Diamond\neg A$, assim como dizer que uma fórmula "é possível" significa que existe uma situação em que tal fórmula pode ser verificada, e assim, sua negação não é necessária: $\Diamond A \leftrightarrow \neg\Box\neg A$.

5.2.2 SISTEMA NORMAL K

Para se definir um *Sistema de Lógica Modal* a partir de uma Lógica Clássica, somente incluir operadores modais não é suficiente. Precisa-se também definir algumas regras que permitam a transformação de teoremas da Lógica Proposicional para o novo sistema lógico. Para a construção de um *Sistema Normal*, deve-se incluir também as duas seguintes regras: *Regra da Necessitação* e o *Axioma K*. A *Regra da Necessitação* afirma que "todo teorema é necessário": $p \rightarrow \Box p$, enquanto o *Axioma K*, representa a "Lei de Aristóteles" e diz que "de uma implicação necessária e uma premissa necessária se obtém uma consequência necessária": $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$.

O *Sistema Normal K* oferece uma base para construção de outros sistemas de Lógica Modal [MAL-02], que podem ser obtidos através da inclusão de novos axiomas e de restrições aplicadas às definições de mundos possíveis.

5.3 SEMÂNTICA DE KRIPKE

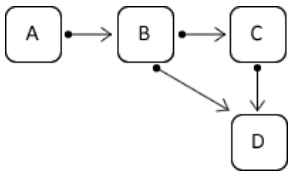
Proposta por *Saul Kripke* como uma ferramenta para representação de um sistema semântico para Lógicas Modais, a *Semântica dos Mundos Possíveis de Kripke* oferece meios para representação de aspectos semânticos dos mundos possíveis de um sistema de Lógica Modal através de *estruturas* e *modelos* [BEN-10].

5.3.1 ESTRUTURA

Uma *estrutura*, ou *frame*, consiste em um par de valores E(W,R) onde:

W é o conjunto não vazio dos *mundos possíveis*;
R é a relação binária de *acessibilidade*;

Na relação de acessibilidade R, seus elementos são pares de valores (x,y), que também podem ser representados por *xRy*. Sua interpretação pode ser x "*acessa*" Y, ou x "*enxerga*" y [BIZ-98]. Uma estrutura pode ainda ser representada por um grafo orientado, no qual os mundos são representados pelos vértices e os elementos da relação de acessibilidade são representados pelas arestas.



W = { A, B, C, D }
R = { (A,B), (B,C), (B,D), (C,D) }

Figura 6: Estrutura de Kripke

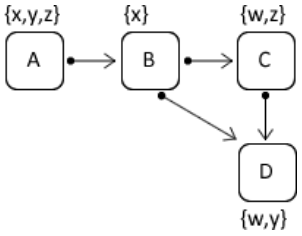
5.3.2 MODELO

Em um sistema de Lógica Modal, seja P o conjunto das suas proposições elementares e S o conjunto potência de W (o conjunto dos mundos possíveis), um *modelo* consiste em uma tripla de valores M(W,R,V) onde:

(W,R) é a *estrutura* que expressa as relações entre os mundos;
V é a *função de avaliação* definida de P em S;

A função V(p) mapeia as proposições elementares de P em S [GIR-95]. Assim, para uma proposição p, V(p) resulta um subconjunto de mundos possíveis onde uma esta é satisfeita.

Exemplo:



S = { {A}, {B}, {C}, {D}, {A,B}, {A,C},
{A,D}, {B,C}, {B,D}, {C,D}, {A,B,C},
{A,B,D}, {A,C,D}, {B,C,D}, {A,B,C,D} }
V(x) = {A,B}
V(y) = {A,D}
V(z) = {A,C}
V(w) = {C,D}

Figura 7: Modelo de Kripke

5.4 SATISFAÇÃO

Uma formula é dita satisfeita em um ambiente lógico quando sua validade pode ser verificada no mesmo. Entretanto, na Lógica Modal, a validade de uma fórmula pode assumir diferentes valores em diferentes mundos. Por isso, quando se deseja verificar a satisfação de uma proposição p em um sistema de Lógica Modal, deve-se especificar um mundo w e um modelo M onde se deseja verificar a validade de p [COS-08]. Quando um mundo w de um modelo M satisfaz uma proposição p em uma Lógica Modal, denota-se $M,w \vDash p$.

De forma geral, seja M(W,R,V) um modelo, $w \in W$ um mundo de M, R a relação de acessibilidade e V uma funcão de avaliação, pode-se dizer:

- [$p = \bot$] , M,w não satisfaz p em M;
- [$p = \alpha \in P$] , w satisfaz p em M, se $w \in V(\alpha)$;
- [$p = \alpha \vee \beta$] , w satisfaz p em M, se w satisfaz α em M OU w satisfaz β em M;
- [$p = \alpha \wedge \beta$] , w satisfaz p em M, se w satisfaz α em M E w satisfaz β em M;
- [$p = \neg\alpha$] , w satisfaz p em M, se w não satisfaz α em M;
- [$p = \Diamond\alpha$] , w satisfaz p em M, se $\exists u \in W \mid wRu \wedge u$ satisfaz p em M;
- [$p = \Box\alpha$] , w satisfaz p em M, se $\forall u \in W \mid wRu \rightarrow u$ satisfaz p em M;

5.4.1 SATISFAÇÃO GLOBAL

Quando uma proposição p é válida em todos os mundos possíveis w ϵ W de um modelo M(W,R,V), pode-se dizer que a fórmula em questão é *globalmente satisfazível no modelo*. A satisfação global pode ser expressa como "M satisfaz p". Isto é:

$$M \vDash p \equiv \forall w \, (M,w \vDash p)$$

5.4.2 VALIDADE EM ESTRUTURAS

Quando uma fórmula φ é globalmente satisfazível em todos os modelos M baseados em uma estrutura E(W,R), pode-se, ainda, dizer que a fórmula em questão é *válida na estrutura*. A validade em estruturas pode ser expressa como "*E satisfaz φ*". Isto é:

$$E(W,R) \vDash \varphi \equiv \forall \, M(W,R,V) \, (M \vDash \varphi)$$

5.5 DEFINIÇÕES GERAIS

A seguir são apresentados alguns conceitos teóricos de lógica modal. Estes conceitos são também apresentados nos trabalhos de *Bruno C. Coscarelli* [COS-08] e *Linh Anh Nguyen* [NGU-06].

5.5.1 CONSISTÊNCIA

Uma lógica L é dita *consistente*, quando a fórmula [p ∧ ¬p] não é satisfazível em L.

5.5.2 CORRETUDE

Uma lógica L é dita *correta* em relação à uma classe de modelos C, quando todo teorema de L é válido em C.

5.5.3 COMPLETUDE

Uma lógica L é dita *completa* em relação à uma classe de modelos C, quando toda fórmula válida em C é um teorema de L.

5.5.4 CONSEQUÊNCIA SEMÂNTICA LOCAL

Uma fórmula φ é dita uma *consequência semântica local* de um conjunto de fórmulas Σ em uma classe de modelos C, quando em todo mundo w em que Σ é satisfeito, φ também for:

$$\forall M \in C \; (\forall w \in W \; (\; w \vDash \Sigma \rightarrow w \vDash \varphi \;))$$

5.5.5 COMPLETUDE FORTE

Uma lógica L é dita *fortemente completa* em relação à uma classe de modelos C, quando toda fórmula φ consequência semântica local de um conjunto de fórmulas Σ pode ser deduzida sintaticamente a partir de Σ.

5.5.6 L-CONSISTÊNCIA

Para uma lógica L, um conjunto de fórmulas Σ é dito *L-consistente*, quando não se pode demonstrar nenhuma inconsistência com L a partir de Σ.

5.5.7 MODELO CANÔNICO

O *Modelo Canônico* de uma Lógica Modal Normal L é definido como um modelo M(W,R,V) onde as seguintes condições são verificadas [COS-08]:

1. W (mundos) é o conjunto de todos os conjuntos L-Consistentes Maximais;
2. R (relação de acessibilidade) é a relação que relaciona mundos de W da seguinte forma: $wRv \in R$, quando $\forall \varphi \in v, \Diamond \varphi \in w$;
3. V (valoração) é uma função definida da seguinte forma: para qualquer proposição elementar p, se $p \in w$, então $w \in V(p)$;
4. Lema: $\forall w,v \in W \; (\; wRv \leftrightarrow \forall \varphi \; (\; (\; \Box \varphi \in w \;) \rightarrow (\varphi \in v) \;)$;

5.5.7.1 LEMA DA EXISTÊNCIA

O *Lema da Existência* afirma que em uma Lógica Modal Normal L, de modelo canônico M(W,R,V), para um mundo w ϵ W, se uma fórmula ◊φ é válida em w, então existe um mundo v ϵ W tal que φ é válida em v e wRv:

$$\forall w \in W\ (\Diamond \varphi \in w) \rightarrow \exists v \in W\ (\varphi \in v \wedge wRv).$$

5.5.7.2 LEMA DA VERACIDADE

O *Lema da Veracidade* afirma que em uma Lógica Modal Normal L, de modelo canônico M(W,R,V), um mundo w ϵ W, satisfaz uma fórmula φ se e somente se φ ϵ w:

$$\forall w \in W\ (M,w \vDash \varphi \leftrightarrow \varphi \in w)$$

5.5.7.3 TEOREMA DO MODELO CANÔNICO

O Teorema do Modelo Canônico diz que toda Lógica Modal Normal é fortemente completa em relação ao seu Modelo Canônico.

5.6 CARACTERIZAÇÃO POR ESTRUTURAS

Uma Lógica Modal L é dita *correta* ou *completa* em relação à uma classe de modelos C, respectivamente, se todos os teoremas de L são fórmulas válidas em C e se todas as fórmulas de C são teoremas de L. Assim, se uma lógica modal L for simultaneamente *correta* e *completa* em relação à uma classe de modelos C, L também é dita *caracterizada pela classe de modelos C* [COS-08].

Quando uma lógica modal L é caracterizada por uma classe de modelos C, definida a partir de uma estrutura E(W,R), as seguintes características podem ser verificadas:

- Teoremas da Lógica Proposicional Clássica são válidos em L;
- Modus Ponens é uma regra de inferência válida em L;
- Substituição Uniforme é uma regra de inferência válida em L;

- Necessitação é uma regra de inferência válida em L;
- O axioma K [$\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$] é uma fórmula válida em L;

Quando uma lógica modal L é caracterizada por uma classe de modelos C que apresente todos os possíveis modelos M(W,R,V) obtidos a partir de uma estrutura E(W,R), L pode também ser dita *caracterizada pela estrutura* E.

Da mesma forma, uma lógica modal L é caracterizada por uma classe de estruturas Θ, quando L é *correta* e *completa* em relação à todas as classes de modelos M(W,R,V) possíveis obtidas à partir de todas as estruturas E(W,R) ϵ Θ. Quando L é caracterizada por Θ, as características de L caracterizada por uma classe de modelos C, listadas acima, continuam válidas.

Assim, a classe de todas as estruturas possíveis Θ caracteriza uma *Lógica Modal Normal (K)*, uma vez que a regra de *Necessitação* e o *Axioma K* são válidos nas lógicas modais caracterizadas por estruturas.

5.7 SISTEMAS DE LÓGICA MODAL

Os fundamentos definidos na semântica dos mundos possíveis de Kripke se resumem ao menor sistema normal da Lógica Modal caracterizado por estruturas, o *Sistema K*. Este sistema é básico, e pode ainda ser estendido com o acréscimo de novos axiomas, originando novos sistemas. Cada axioma basicamente impõe uma condição na estrutura de representação dos mundos possíveis, especificamente, na relação de acessibilidade [GIR-95, BIZ-98, COS-08]. A seguir são mostrados os principais sistemas normais de Lógica Modal.

5.7.1 SISTEMA T

O *Sistema T* é um sistema normal de Lógica Modal obtido pelo acréscimo do Axioma T ao Sistema base K. O Axioma T é definido como: □p → p - "se p é necessário, então p é válido".

Dessa forma, um Sistema T é uma Lógica Modal L caracterizada por uma Estrutura E(W,R) onde R é uma relação reflexiva. Isto é, todos os mundos possíveis "acessam" a si mesmos: ∀w ϵ W (wRw). Assim, um Sistema T é dito completo em relação à classe das estruturas Reflexivas.

Axiomas em T:

(K) □(p → q) → (□p → □q)
(T) □p → p

Diagrama:

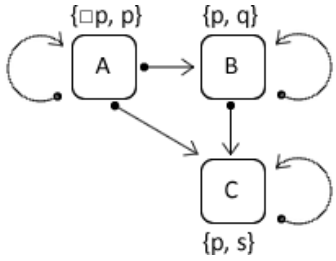


Figura 8: Sistema Modal T

5.7.2 SISTEMA D

O *Sistema D* é um sistema normal de Lógica Modal obtido pelo acréscimo do Axioma D ao Sistema base K. O Axioma D é definido como: □p → ◊p - "se p é necessário, então p é possível".

O Axioma D oferece uma interpretação diferente sobre o operador modal □, quando comparado ao Axioma T. Em algumas lógicas pode não ser adequado considerar qualquer proposição p válida à partir de um □p. Por exemplo, em uma lógica

deôntica o operador □ representa a modalidade de obrigação. Porém, nem tudo que é "obrigatório" necessariamente ocorre, como garante o Axioma T. Assim, em casos como esse, é mais conveniente utilizar lógicas que apresentam o Axioma D.

Dessa forma, um Sistema D pode ser definido como uma Lógica Modal L caracterizada por uma Estrutura E(W,R) onde R é uma relação serial. Isto é, não existe um mundo w terminal em W: $\forall w \in W \; \exists v \in W \; (wRv)$. Assim, um Sistema D é dito completo em relação à classe das estruturas Seriais.

Axiomas em D:

(K) $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$
(D) $\Box p \rightarrow \Diamond p$

Diagrama:

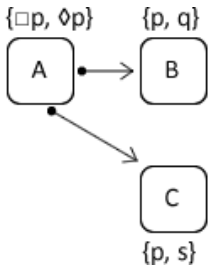


Figura 9: Sistema Modal D

5.7.3 SISTEMA B

O *Sistema B* é um sistema normal de Lógica Modal obtido pelo acréscimo do Axioma B ao Sistema T. O Axioma B é definido como: $p \rightarrow \Box\Diamond p$.

Dessa forma, um Sistema B é uma Lógica Modal L caracterizada por uma Estrutura E(W,R) onde R é uma relação reflexiva e simétrica. Isto é, $\forall w, v \in W \; (wRv \leftrightarrow vRw) \wedge \forall u \in W \; (uRu)$. Assim, um Sistema B é dito completo em relação à classe das estruturas Reflexivas e Simétricas.

Axiomas em B:

(K) □(p → q) → (□p → □q)
(T) □p → p
(B) p → □◊p

Diagrama:

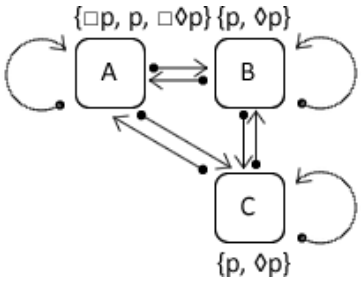


Figura 10: Sistema Modal B

5.7.4 SISTEMA S4

O *Sistema S4* é um sistema normal de Lógica Modal obtido pelo acréscimo do Axioma 4 ao Sistema T. O Axioma 4 é definido como: □p → □□p.

Dessa forma, um Sistema S4 é uma Lógica Modal L caracterizada por uma Estrutura E(W,R) onde R é uma relação reflexiva e transitiva. Isto é, ∀w, v, u ϵ W (wRv ∧ vRu ↔ wRu) ∧ ∀x ϵ W (xRx). Assim, um Sistema S4 é dito completo em relação à classe das estruturas Reflexivas e Transitivas.

Axiomas em S4:

(K) □(p → q) → (□p → □q)
(T) □p → p
(4) □p → □□p

Diagrama:

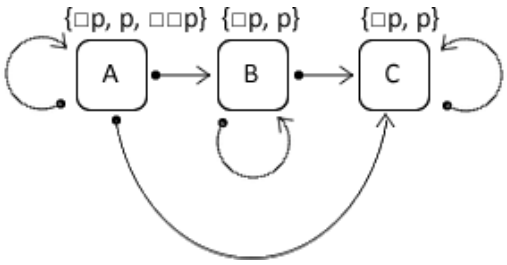


Figura 11: Sistema Modal S4

5.7.5 SISTEMA S5

O *Sistema S5* é um sistema normal de Lógica Modal obtido pelo acréscimo do Axioma 5 ao Sistema T. O Axioma 5 é definido como: ◊p → □◊p.

Dessa forma, um Sistema S5 é uma Lógica Modal L caracterizada por uma Estrutura E(W,R) onde R é uma relação euclidiana. Isto é, ∀w, v, u ϵ W (wRv ∧ wRu ↔ vRu) ∧ ∀x ϵ W (xRx). Portanto, um Sistema S5 é dito completo em relação à classe das estruturas Reflexivas, Transitivas e Simétricas.

Axiomas em S5:

(K) □(p → q) → (□p → □q)
(T) □p → p
(5) ◊p → □◊p

Diagrama:

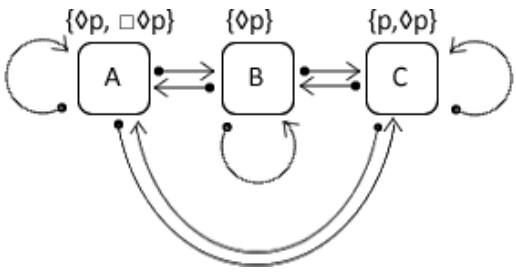


Figura 12: Sistema Modal S5

5.8 LÓGICAS MULTIMODAIS

Como visto até agora, os operadores modais universal e existencial, representados pelos símbolos □ e ◊, denotam a validade de uma determinada proposição como necessária, quando em todos os mundos acessíveis esta é válida, ou como possível, quando existe ao menos um mundo acessível em que esta seja válida. Percebe-se ainda, que uma lógica modal é definida por um modelo, do qual faz parte uma única relação de acessibilidade de mundos.

Em Lógicas Multimodais, o que se tem é a presença de diferentes relações de acessibilidade sobre um mesmo modelo. Para cada relação, representadas por índices diferentes, pode-se assumir axiomas e restrições diferentes. Os operadores □ e ◊, também devem ser indexados de acordo com a relação ao qual fazem parte, da seguinte forma: $\Box_i$ e $\Diamond_I$ [NGU-06]. Relações entre operadores ainda podem ser definidas em um sistema multimodal, como por exemplo, em um sistema de diferentes graus de crença, no qual o índice i represente um grau de crença, é conveniente considerar a regra $\Box_i \rightarrow \Box_{i-1}$. Isto é, tudo que se acredita com grau i também se acredita com grau i-1.

Nos capítulos seguintes, são apresentadas lógicas multimodais para se tratar problemas de lógica de *diferentes graus de crença*, e problemas de lógica de *crenças de diferentes agentes*. O primeiro tipo de problema pode ser modelado como uma lógica multimodal KDI45, onde I representa o axioma $\Box_i \rightarrow \Box_{i-1}$, enquanto o segundo problema pode ser modelado como uma lógica multimodal KD45.

6 LÓGICA MODAL DE PRIMEIRA ORDEM

Enquanto o capítulo anterior fornece uma introdução aos fundamentos da Lógica Modal de uma forma geral, este capítulo é dedicado à Lógica Modal de Primeira Ordem com foco nos mecanismos de resolução e formas de representação de seus programas.

6.1 LÓGICAS MODAIS QUANTIFICADAS

Como visto anteriormente, um sistema de Lógica Modal é obtido pelo acréscimo de operadores modais sobre sistemas de Lógica Clássica. Desta forma, é possível se estender sistemas de *Lógica de Primeira Ordem* com o acréscimo de modalidades, obtendo sistemas de *Lógica Modal de Primeira Ordem*.

Um sistema de *Lógica Modal de Primeira Ordem* é, portanto, um sistema de Lógica Proposicional com a presença de *Quantificadores* e *Operadores Modais*, e assim, deve apresentar características de ambas as lógicas [BIZ-98]. Enquanto na Lógica de Primeira Ordem, os quantificadores são aplicados a indivíduos de um único universo, um sistema de Lógica de Primeira Ordem Modal deve considerar a possibilidade de existência de diversos mundos. Daí, surge uma questão interessante: como deve ser tratada a possibilidade da existência de diferentes indivíduos em diferentes mundos?

6.2 FÓRMULA DE BARCAN

Uma abordagem de como se tratar a possibilidade da existência de diferentes indivíduos em diferentes mundos foi levantada em 1946 por *Ruth. C. Barcan*, e ficou conhecida como *Fórmula de Barcan* [GIR-95]:

$$\forall x\ (\Box P(x)) \rightarrow \Box(\forall x(P(x)))$$

A aceitação desta fórmula impõe um universo *constante* ou *decrescente* nos mundos possíveis. Isto é, não existe nenhum mundo v acessível direta ou indiretamente a partir do mundo atual w em que exista novos indivíduos não existentes em w.

Para se impor um universo *constante* ou *crescente*, pode-se também optar por aceitar a *Fórmula de Barcan reversa*:

$$\Box(\forall x(P(x))) \rightarrow \forall x\,(\Box P(x))$$

A aceitação simultânea das fórmulas de Barcan e sua reversa implicam um universo constante, onde em todo mundo w ϵ W possível em um modelo M(W,R,V), existe o mesmo conjunto invariante de indivíduos.

6.3 ABORDAGEM INDIRETA

Como visto no capítulo 2, a *Abordagem Indireta* é aquela que faz uso de tradução para expressar problemas de lógica modal somente com termos de lógica clássica. Para ilustrar como é feita a tradução e como é realizado o cálculo de resolução, vamos considerar o trabalho de *Ohlbach* [OHL-88].

Ohlbach, propõe em seu trabalho, uma forma de indireta de se tratar problemas de lógica modal como problemas de lógica clássica. Seu método consiste em uma tradução que é aplicada sobre o programa original (modal), de forma que o programa resultante seja formado apenas por fórmulas na *Forma Normal Conjuntiva*. Com isso, se limita a variedade sintática da lógica modal sem perda de expressividade.

Este método é somente aplicável à lógicas modais com os dois operadores $\Box$ e $\Diamond$ e limitado à estruturas seriais. As propriedades reflexividade, simetria e transitividade também podem ser observadas nessa classe de estruturas, resultando assim nos sistemas modais quantificados T, S4, S5, B, D, D4 e DB. Uma última restrição para o uso deste método é assumir um universo constante no modelo.

6.3.1 WORLD-TERMS

Para representação de contextos modais, *Ohlbach* propõe em seu trabalho termos especialmente denominados "world-terms". Um world-term é um termo especial que representa o contexto modal no qual um determinado predicado é válido.

Ohlbach também define uma segunda estrutura especial, que varia sintaticamente dos world-terms, chamada "world-paths". Os world-paths representam o mesmo que os world-terms, porém, como uma estrutura de dados mais conveniente para se trabalhar nos algoritmos de unificação. Mais adiante esta estrutura é detalhadamente apresentada.

6.3.2 P-LÓGICA

Entende-se por P-lógica, uma lógica no estilo de lógica de Predicado que neste caso, é definida pela substituição de operadores modais de uma lógica modal por world-terms. Uma P-lógica pode ser definida como P (V_D , F_D , P , 0 , V_W , F_W), onde:

V_D é o conjunto D-Variáveis das variáveis do domínio

F_D é o conjunto D-Estimados de símbolos funcionais relativos ao domínio

P é o conjunto de símbolos predicativos

V_W é o conjunto W-Variáveis dos mundos possíveis

0 é um símbolo de V_W que representa o mundo inicial

F_W é o conjunto dos símbolos funcionais relativos aos mundos possíveis

Exemplos:

Lógica Modal	P-Lógica	Comentários
□P	∀w P(w(0))	O predicado P é necessário, portanto, é

		verdade PARA TODO mundo w acessível à partir do mundo 0.
◊P	P(g(0))	O predicado P é possível, portanto, é verdade PARA ALGUM mundo g acessível à partir do mundo 0.

Quadro 7: Exemplos de World-Terms

6.3.3 TRADUÇÃO DE PROGRAMAS PARA P-LÓGICA

O processo de tradução de uma lógica modal para uma P-Lógica, é dado por uma função Π(P) que transforma fórmulas da lógica modal P em fórmulas da P-lógica, atualizando os conjuntos V_W com as variáveis que substituem o operador □ e as *funções de skolem* que substituem o quantificador ∃ e o operador ◊. Uma função auxiliar π é também necessária para a atualização dos conjuntos da P-Lógica.

A função π faz o descendente recursivo nas fórmulas e termos modais do programa original, registrando como um segundo argumento o contexto modal na forma de um World-Term w e como um terceiro argumento as variáveis universalmente quantificadas D-vars. Seja p uma variável global a ser atualizada durante o descendente recursivo, então:

D-Vars + x denota a concatenação de uma lista D-vars = (x_1, ... , x_n) com x, resultando D-vars + x = (x_1, ... , x_n, x)

f(w + D-vars) denota o termo f(w, x_1, ..., x_n) onde D-vars = (x_1, ... , x_n)

As regras de Tradução são dadas por:

1. Π(P) → π(P,0,()), onde () é uma lista vazia.
2. π(P ∧ Q, w, D-vars) → π(P, w, D-vars) ∧ π(Q, w, D-vars)
3. π(P v Q, w, D-vars) → π(P, w, D-vars) v π(Q, w, D-vars)
4. π(∀xP, w, D-vars) → ∀x π(P, w, D-vars + x)
5. π(□P, w, D-vars) → ∀u π(P, u(w), D-vars), u é adicionado em V_W como um novo mundo.
6. π(∃xP, w, D-vars) → π(P, w, D-vars) [x ← f(w + D-vars)], a f é adicionado à F_D como um novo símbolo funcional.

7. $\pi(\Diamond P, w, D\text{-vars}) \rightarrow \pi(P, g(w + D\text{-vars}), D\text{-vars})$, g é adicionado à F_W como novo símbolo funcional relativo a mudança de mundos

Agora seja P um símbolo predicativo n-ário e f um símbolo funcional n-ário:

8. $\pi(\neg P(t_1, ..., t_n), w, D\text{-vars}) \rightarrow \neg P(w, \pi(t_1, w, D\text{-vars}), ..., \pi(t_n, w, D\text{-vars}))$
9. $\pi(P(t_1, ..., t_n), w, D\text{-vars}) \rightarrow P(w, \pi(t_1, w, D\text{-vars}), ..., \pi(t_n, w, D\text{-vars}))$
10. $\pi(f(t_1, ..., t_n), w, D\text{-vars}) \rightarrow f(w, \pi(t_1, w, D\text{-vars}), ..., \pi(t_n, w, D\text{-vars}))$
11. $\pi(x, w, D\text{-vars}) \rightarrow x$, onde x é uma D-variável

6.3.4 WORLD-PATHS

Os "world-terms", como visto anteriormente, contém W-Variáveis em uma posição funcional. Para se definir um algoritmo de unificação, utiliza-se a sintaxe de "world-paths", que é mais conveniente para este propósito. A transição de world-terms para world-path pode ser semanticamente explicada, ao se interpretar os símbolos funcionais e W-Variáveis (mundos), como funções.

Posteriormente, uma operação conhecida como *currying* é então aplicada sobre essas funções removendo o "argumento de mundo", deixando uma função que é aplicável a somente elementos de domínio. Dessa forma, uma função aninhada $f(g(w, s_1, ..., s_n), t_1, ..., t_n)$ depois do *curry* ficaria $g(w, s_1, ..., s_n)\ f(t_1, ..., t_n) = w\ (g(s_1, ..., s_n) \circ f(t_1, ... , t_n))$. O símbolo '$\circ$' que representa composição de funções pode ser removido e assim, os world-paths podem ser representados como uma lista: $[w\ g(s_1, ... s_n)\ f(t_1, ... t_n)]$. Um termo $f(t_1, ..., t_n)$ cujo primeiro argumento não é um world-term é chamado CW-Term ou "curried world term".

Alguns exemplos de World-Paths:

Lógica Modal	World-Terms	World-Paths
$\Box P$	$\forall w\ P(w(0))$	$\forall w\ P\ [0w]$
$\Diamond P$	$P(g(0))$	$P[0g]$
$\forall x\ \Diamond P(x)$	$\forall x\ P(g(0), x)$	$\forall x\ P([0g], x)$

Quadro 8: Exemplos de World-Paths

6.3.5 UNIFICAÇÃO COM WORLD-PATHS

O algoritmo de unificação com world-paths não se difere do algoritmo de unificação da lógica de predicados de primeira ordem, exceto por que as substituições produzidas devem ser *R-compatíveis*, isto é, compatíveis com a relação R de acessibilidade dos mundos possíveis. Para que as propriedades das relações sejam devidamente consideradas durante o processo de unificação, o algoritmo geral é mapeado para versões R-Dependentes de acordo com as propriedades da relação R.

Relações Seriais

Para uma relação de acessibilidade serial apenas, que não apresente nenhuma outra propriedade como simetria, reflexividade ou transitividade, uma substituição R-compatível permite que um world-path parcial seja substituído por exatamente um único CW-Term. Assim, o world-path [0 v a] unifica com [0 b w], para v = b e w = a. Porém o mesmo não unifica com [0 v u w], pois uma substituição não compatível (v = [b c] e w = a) seria necessária.

Relações Reflexivas

Para uma relação de acessibilidade que apresente apenas reflexividade como uma propriedade adicional, uma substituição de uma W-Variável w por um World-Path vazio [] passa a ser R-compatível. Pela definição de uma relação reflexiva, um mundo pode ser acessado à partir de si mesmo, ou seja, w pode ser completamente removido da definição de um world-path. Assim, os world-path [0 v a] e [0 b u w] unificam, por exemplo, para as unificações v = b, u = a e w = []. Ainda, outras substituições podem ser possíveis, como v = b, u = [] e w = a. Portanto, o algoritmo de unificação deve considerar todas as possibilidades da ocorrência de w = [].

Relações Simétricas

Para uma relação de acessibilidade que apresente apenas simetria como uma propriedade adicional, uma substituição de um World-Path parcial [a w] por [] é R-Compatível, para uma W-Variável $w = a^{-1}$, representando o acesso de volta ao mundo de onde o mundo a era anteriormente acessado. Assim, é permitido que um world-path parcial seja substituído por exatamente um CW-Tem ou por exatamente um CW-Term inverso, já que em interpretações simétricas, sempre existe um CW-Term inverso. Por exemplo, os world-paths [0 v w] e [0] unificam para $w = v^{-1}$. Neste caso, o algoritmo de unificação deve considerar todas as possibilidades de se substituir uma W-Variável w, em um world-path parcial [t w], por t^{-1} para todos os predecessores t no world-path.

Relações Reflexivas e Simétricas

Para uma relação de acessibilidade que apresente apenas reflexividade e simetria como suas duas únicas propriedades adicionais, une-se as idéias básicas vistas anteriormente. Assim, o algoritmo de unificação deve considerar todas as possibilidades de se remover W-Variáveis w de um world-path [t w], substituindo w por [], ou substituindo [t w] ($w = t^{-1}$) por [].

Relações Transitivas

Para uma relação de acessibilidade que apresente apenas transitividade como uma propriedade adicional, uma substituição de uma W-Variável w por um World-Path parcial é R-Compatível. Por exemplo, dois world-paths [0 v c d] e [0 a b w d] unificam para v = [a b] e w = c, assim como também unificam para v = [a b w'] e w = [w' c]. O algoritmo de unificação para dois world-paths $s = [s_1 \ldots s_n]$ e $t = [t_1 \ldots t_n]$, deve então trabalhar da esquerda para a direita, de acordo com os passos básicos:

1. O termo s1 unifica com t1 e o algoritmo é chamado recursivamente para a unificação de $[s_2 \ldots s_n]$ e $[t_2 \ldots t_n]$.

2.a. Se s_1 é uma W-Variável, então para i = 2, ..., m, cria-se componentes de unificação $s_1 = [t_1 \ ... \ t_n]$ e então o algoritmo de unificação é chamado recursivamente para a unificação de $[s_2 \ ... \ s_n]$ e $[t_{i+1} \ ... \ t_m]$.

2.b. Se t_i é uma W-Variável, então t_i é desmembrada em duas W-Variáveis [u v], definindo assim as componentes de unificação $s_1 = [t_1 \ ... \ t_{i-1} \ u]$ e $t_i = [u \ v]$ e então o algoritmo é recursivamente chamado para a unificação de $[s_2 \ ... \ s_n]$ e $[v \ t_{i+1} \ ... \ t_m]$

Relações Reflexivas e Transitivas

Para uma relação de acessibilidade que apresente apenas reflexividade e transitividade como suas duas únicas propriedades adicionais, une-se as idéias básicas vistas anteriormente. Assim, o algoritmo de unificação definido para tratar relações transitivas deve ser aumentado com um passo adicional que remove W-Variáveis w com a substituição w = [].

Relações de equivalência

No caso particular de um sistema de lógica modal S5 na forma normal de grau modal 1, um world-path consiste de, no máximo, dois CW-Terms. Assim, os world-paths podem ser como [0] ou [0 t]. Para uma W-Variável w, dois world-paths [0] e [0 w] unificam, quando w = [].

6.4 ABORDAGEM DIRETA

Como visto no capítulo 2, a *Abordagem Direta* é aquela que trabalha com as modalidades diretamente expressas no programa através de novas representações sintáticas, dispensando a necessidade de se transformar problemas de lógica modal em problemas de lógica clássica. Na sequência deste trabalho, vamos estudar mais detalhadamente como *Linh Anh Nguyen* trata as modalidades de forma direta em seu trabalho *MProlog* [NGU-06].

Linh Anh Nguyen, propõe em seu trabalho, uma forma direta de se tratar problemas de lógicas modais normais com suporte aos axiomas T, D, B, 4 e 5. Seu método consiste em uma técnica de rotulação que é aplicada às cláusulas e termos dos programas modais e que representam um caminho na estrutura dos mundos possíveis de Kripke. Assim, uma declaração do tipo ◊p(a) pode ser reescrita como <p(a)> p(a). Isto é, se ◊p(a) é válido em um mundo atual t, então existe um mundo w, acessível a partir de t, tal que p(a) seja válido em w. Para se representar esta ligação de t → w na relação R e que em w p(a) é válido, utiliza-se a forma rotulada <p(a)> p(a) em t.

No *MProlog*, o modelo não é explicitamente declarado no programa como um conjunto de mundos e uma (ou mais, no caso de uma lógica multimodal) relação de acessibilidade, mas sim deduzido a partir das declarações que são feitas acerca do mundo atual. *Linh Anh Nguyen* define um conjunto gerador de modelo como um conjunto de proposições atômicas que pode ser obtido fazendo sucessivas anotações das formas rotuladas para cada ocorrência do operador ◊ nas proposições iniciais.

Por exemplo, seja 1. ◊p(a) e 2. □(q(X) :- p(X)) declarações de fatos válidos no mundo atual (t), então existe um mundo w acessível à partir de t, tal que em w, p(a) seja válido. A declaração p(a) em t pode ser reescrita como <p(a)> p(a), identificando o mundo w por <p(a)>. Da proposição 2, temos que a regra q(X) :- p(X) é válida também em w, podendo ser reescrita como <p(a)> (q(X) :- p(X)). Como q(X) :- p(X) e p(a) são proposições válidas em w, então q(a) também é uma proposição válida em w, podendo ser reescrita como <p(a)> q(a). Por fim, um gerador de modelo para este exemplo seria o conjunto { <p(a)> p(a), □(q(X) :- p(X)), <p(a)> (q(X) :- p(X)), <p(a)> q(a) }.

O conjunto gerador de modelo representado acima, define o seguinte modelo de Kripke:

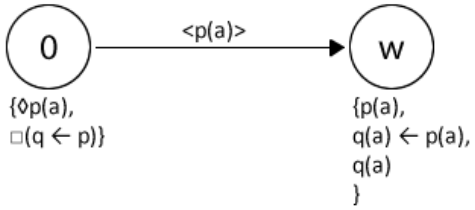


Figura 13: Modelo de Kripke resultante

6.4.1 A LINGUAGEM MPROLOG

MProlog é o nome da linguagem proposta por *Linh Anh Nguyen* para programação em lógica modal. A linguagem é tão expressiva quanto cláusulas de Horn na Lógica Modal, porém por questões práticas, a linguagem sofre algumas restrições para se tratar problemas quando o axioma 5 se faz presente no sistema, como por exemplo para lógicas de crenças, definindo assim uma linguagem *L-MProlog*.

Define-se *modalidade* como uma sequencia, possivelmente nula, de operadores modais, representada pelo símbolo ∇. Uma modalidade é dita *universal*, quando esta é composta apenas por operadores modais universais e nesse caso é representada pelo símbolo $\boxdot$. Da mesma maneira que a programação em lógica clássica, o *MProlog,* faz uso da forma clausal $\boxdot(A \leftarrow B_1, ..., B_N)$ para expressar fórmulas do programa.

Um programa *MProlog*, é então definido como um conjunto finito de cláusulas, que por sua vez são definidas como $\boxdot(A \leftarrow B_1, ..., B_N)$, sendo A, B_1, ..., B_N (com N >= 0) os termos da cláusula. Um termo é denotado por E, $\Box E$ ou $\Diamond E$, sendo E um termo clássico. $\boxdot$ representa o contexto modal da cláusula. Uma consulta *MProlog* é definida como uma conjunção de submetas, representadas por termos *MProlog*, da forma $\leftarrow s_1, ..., s_N$.

A restrição que é imposta à um programa *L-MProlog*, diz respeito ao comprimento do contexto modal, isto é, número de operadores modais aninhados, declarado nas cláusulas seja de no máximo 2 em uma lógica *KD5*, e no máximo 1 em lógicas *KD45* e *S5*.

6.4.2 NOTAÇÕES E OPERADORES MODAIS NA FORMA ROTULADA

Na programação em lógica clássica, o operador da consequência direta opera conjuntos de proposições atômicas utilizando as cláusulas declaradas no programa para

computar as consequências diretas de um conjunto de entrada. Para a lógica clássica, este operador é monotônico e contínuo. Entretanto, para que um resultado similar seja obtido na lógica modal, é necessário definir um domínio para este operador. Ao se obter uma proposição atômica da forma ◊*E*, podemos utilizar a notação rotulada para simplificar a tarefa. Assim, uma proposição ◊*E* passa a ser denotada por <*E*> *E*, onde o rótulo E representa um próximo mundo (eventualmente mais de um) em que a proposição E seja válida.

A rotulação do operador ◊ é importante também em casos de consultas serem estabelecidas com contextos modal. Uma consulta ← ◊(A ∧ B) não poderia ser desmembrada pelas submetas ← ◊A, ◊B, já que A pode ser possível em um próximo mundo u, enquanto B pode ser possível em um próximo mundo v diferente de u e a consulta resultaria verdade. Porém, utilizando a forma rotulada, uma consulta ← <X>(A ∧ B) seria desmembrada em submetas ← <X> A, <X> B, o que seria permitido, pois a identidade do próximo mundo onde A e B são válidos é preservada.

Notações:

- T : o símbolo “verdade” (verum);
- E, F : proposições atômicas clássicas;
- X, Y, Z : variáveis para átomos clássicos ou T, chamada *variável de átomo*
- <E>, <X> : operador ◊ rotulado por E ou X;
- ∇: □, ◊, <E> ou <X>, chamado de *operador modal;*
- Δ: uma sequencia (possivelmente vazia) de operadores modais, chamada *modalidade*;
- ⧈ : modalidade universal (uma modalidade que só contém operadores universais)
- A, B : fórmulas do tipo E ou ∇E, chamadas de *átomos*;
- Subscritos identificam nos operadores modais e nas modalidades o índice do operador modal que se refere (nas lógicas multimodais).
- Sobrescritos nas modalidades representam a quantidade de operadores modais que aparecem na sequencia de operadores.

Substituição:

O mecanismo da substituição também é redefinido para que se considere corretamente as variáveis de átomos e as fórmulas rotuladas. Uma substituição é um conjunto da forma $\theta = \{x_1/t_1, x_2/t_2, \ldots, x_N/t_N, X_1/E_1, X_2/E_2, \ldots, X_N,E_N, Y_1/Z_1, Y_2/Z_2, \ldots, Y_N/Z_N\}$, onde $x_1, x_2, \ldots x_N$ são variáveis distintas, $t_1, t_2, \ldots, t_N$ são termos, $X_1, X_2, \ldots, X_N$, $Y_1, Y_2, \ldots, Y_N$ são variáveis de átomo distintas, e para cada elemento v/s, v é diferente de s. O conjunto $\{x_1, x_2, \ldots, x_N, X_1, X_2, \ldots, X_N, , Y_1, Y_2, \ldots, Y_N\}$ é então chamado domínio da substituição e denotado por $Dom(\theta)$.

O rótulo que é aplicado ao operador ◊ recebe o nome de *EdgeLabel* (rótulo de aresta). A semântica de um rótulo de aresta é especificada da seguinte forma: Seja M = (W,R,V) um modelo de Kripke e P o conjunto das proposições elementares, uma função de ◊-realização é definida como $\sigma : W \times Rótulos \rightarrow W$ tal que se $\sigma(w,<E>) = u$, então existe uma aresta (w,u) pertencente ao conjunto R e neste mundo u, a proposição E é satisfeita. Dados uma função de ◊-realização σ, um mundo w e uma proposição elementar p, a relação de satisfação M, σ, w ⊨ p é definida usualmente, exceto para M, σ, w ⊨ <E> p quando σ(w,<E>) está definido e M, σ, σ ⊨ p. Denota-se M, σ, t ⊨ p por M, σ ⊨ p, para o mundo atual t.

Variáveis de átomos que aparecem nos operadores rotulados como <X> são geralmente interpretadas como substituições. Quando uma proposição p é considerada semanticamente, todos os operadores modais <X> em p são tratados como <T> (sendo T o símbolo que denota verdade, o verum). Dados um modelo M, uma função de ◊-realização σ e uma proposição rotulada p, escreve-se M, σ ⊨ ∀(p) para denotar que para qualquer substituição θ que substitua todas as variáveis por termos elementares e que não substitua variáveis de átomos, M, σ ⊨ p θ δ_T, onde δ_T = { X/T | X é uma variável de átomo }.

6.4.3 GERADORES DE MODELO

Como mencionado anteriormente, o operador de consequência direta $T_{L,P}$ é redefinido para um programa *MProlog* P, de forma que entradas e saídas sejam

constituídas por átomos do tipo ΔE, onde Δ é uma sequencia de operadores modais da forma □ ou <E>. Essas entradas são processadas por geradores de modelos de $T_{L,P}$, já que representam o modelo mínimo para P.

Em uma lógica modal L, um átomo pode ser reduzido à uma forma mais compacta de representação. Por isso, para cada lógica modal L, define-se uma forma L-normal de se representar modalidades. Uma modalidade Δ está rotulada na forma L-normal, se Δ está na forma L-normal e Δ não contém operadores modais do tipo ◊ ou <T>. Um átomo está na forma L-normal se este é do tipo ΔE com Δ na forma L-normal ou, ainda, está na forma quase L-normal se o mesmo é do tipo ΔA com Δ na forma L-normal e sendo A formado por átomos.

Um gerador de modelo L-normal consiste em um conjunto de proposições elementares na forma L-normal que não contenham ◊, <T> ou T e representa um L-modelo (modelo mínimo). Um modelo mínimo pode ser então construído à partir do gerador de modelo L-normal da seguinte forma: fórmulas do tipo □p que apareçam em um mundo w indicam que p é válido para todo mundo possível a partir de w, e fórmulas do tipo <E> p que apareçam em um mundo w indicam que existe um mundo possível identificado por w<E>, um caminho de w para w<E> na relação R e que p é válido no mundo w<E>.

Uma regra "*Forward*" é definida para duas proposições atômicas p e q como p → q, enquanto uma regra "*Backward*" é definida como p ← q. Regras do tipo Forward são utilizadas quando se trabalha com semântica de ponto fixo enquanto regras do tipo Backward são utilizadas para se processar o cálculo de resolução.

Um operador Ext_L é então definido como um conjunto de regras Forward, capaz de produzir uma extensão de um gerador de modelo L-normal I, de forma que a extensão $Ext_L(I)$ apresente todas as proposições atômicas rotuladas na forma L-normal deriváveis à partir de I. Seja W' = EdgeLabels* o conjunto de todas as sequências possíveis de <E>, uma relação R' contendo os mundos W x W' e H : W x W' → P uma

função de validade que mapeia para cada mundo de W x W' às proposições satisfeitas, as seguintes regras são válidas:

- Se <E>p é pertinente em H(w), então o par (w, w<E>) é incluído na relação R e {E, p} são incluídos em H(w<E>);

- Se □p é pertinente em H(w) e o par (w,w<E>) está definido na relação R', então {p} é incluída em H(w<E>).

6.4.4 RESOLUÇÃO

Enquanto o processo de se construir o modelo à partir de um gerador de modelo pode ser visto como um método que ocorre de baixo para cima, no qual um conjunto inicial de proposições atômicas é expandido gerando-se novas combinações de cláusulas e formas rotuladas de modalidades, o cálculo de resolução é um processo que ocorre de cima para baixo, partindo das consultas submetidas ao programa em direção aos fatos declarados.

A principal tarefa para se definir um cálculo de resolução para o MProlog foi especificar uma analogia reversa para o operador de consequência direta $T_{L,P}$ que fosse capaz de tratar corretamente as variáveis, já que estas não apareciam antes no processo de derivação das proposições para o gerador de modelo I mostrado anteriormente.

Uma consulta, ou meta, é constituída de submetas (p_i), da forma $\leftarrow a_1, a_2, ..., a_N$. Cada submeta representa uma proposição atômica que se deseja verificar em um programa *MProlog* P. Seja $G = \leftarrow a_1, a_2, ..., a_N$ uma consulta e $p = \boxdot(A \leftarrow B_1, B_2, ..., B_N)$ uma cláusula do programa P, então G' é derivado de G e p utilizando uma substituição θ se as seguintes condições forem válidas:

- $a_i = \Delta$'A', com Δ' na forma L-normal rotulada é chamado de *átomo selecionado* e A' é chamado de *átomo de cabeça de cláusula selecionado*;

- Δ' é uma L-instancia de uma modalidade universal $\boxdot$' e $\boxdot(A \leftarrow B_1, ..., B_N)$ é uma L-instancia de uma cláusula p.

- θ é uma substituição de A' e da forma rotulada de A.

- G' é uma meta $\leftarrow (a_1, ..., a_{i-1}, \Delta'B_1, ... \Delta'B_N, a_{i+1}, ... , a_k)\ \theta$.

Exemplo de Resolução:

Seja um programa P definido pelas seguintes cláusulas:

c1: $\Box_2 (p(X) \leftarrow \Diamond_2 q(X))$
c2: $\Box_1 (q(X) \leftarrow r(X), s(X))$
c3: $\Box_1 (\Box_1 r(X) \leftarrow s(X))$
c4: $\Diamond_1 s(a)$.

Para executar o cálculo de resolução que tenta provar a meta $\Box_1 p(X)$ no programa de lógica modal *KDI4$_S$5* definido acima, considera-se as seguintes regras:

(a) $\nabla E \leftarrow <X>_i \nabla E$, se ∇ é um operador $\Box_i$ ou $<E>_i$ e X é uma variável de átomo livre;

(b) $\Delta\Diamond_i E \leftarrow \Delta<X>_i E$, se X é uma variável de átomo livre;

(c) $\Delta\nabla_i p \leftarrow \Delta\Box_j p$, se i <= j;

(d) $\Delta\Diamond_i E \leftarrow \Delta\Diamond_j E$, se i > j;

(e) $\Delta\Delta'E \leftarrow \Delta'E$, se Δ' é um operador $\Box_i$ ou $\Diamond_i$

Pilha de prova	Cláusulas / Regras	Instanciações
$\leftarrow \Box_1 p(X)$	c1	
$\leftarrow \Box_1 \Diamond_2 q(X)$	(e)	
$\leftarrow \Diamond_2 q(X)$	(d)	

← $\Diamond_1$ q(X)	(b)	
← $<W>_1$ q(X)	c2	
← $<W>_1$ r(X), $<W>_1$ s(X)	(c)	
← $\Box_1$ r(X), $<W>_1$ s(X)	(a)	
← $<V>_1$ $\Box_1$ r(X), $<W>_1$ s(X)	c3	
← $<V>_1$ s(X), $<W>_1$ s(X)	c4	{ X / a, V / s(a) }
← $<W>_1$ s(a)	c4	{ W / s(a) }

Quadro 9: Exemplos de Resolução MProlog

7 DESENVOLVIMENTO

O desenvolvimento prático deste trabalho se inicia com a definição de uma gramática para programas Prolog estendida com suporte a modalidades. Uma modalidade, assim como no trabalho *MProlog*, é representada por uma lista de operadores modais, que são representados como átomos ou estruturas básicas na frente de termos e cláusulas. Este capítulo explica detalhadamente como foi feita a implementação da ferramenta para programação em lógica, tema deste trabalho.

7.1 VISÃO GERAL

Um programa lógico consiste em um texto no qual fatos conhecidos e regras válidas são declarados. A execução de um programa lógico, no entanto, consiste na tentativa de se provar uma determinada consulta, isto é, verificar sua validade no sistema. No Prolog, provar uma meta significa tentar encontrar um caminho, através de sucessivas instanciações e substituições de suas submetas por regras, de forma que a mesma possa ser expressa apenas em função de fatos declarados verdadeiros no programa.

Neste trabalho, todas as funcionalidades necessárias para a execução de um programa Prolog foram integralmente desenvolvidas em Java, sem a utilização de qualquer outra biblioteca ou Framework. O resultado obtido foi um programa interpretador da linguagem Prolog, com uma extensão para tratar modalidades, capaz de ler um arquivo que contém as declarações de fatos e regras, ler consultas submetidas por usuários, processar as consultas e apresentar o resultado da execução de uma consulta: se é ou não válida e eventuais instanciações realizadas.

Os principais elementos desenvolvidos são programados nas classes *Parser.java,* que contém as lógicas para análises Léxica e Sintática, *SemanticParser.java,* que apresenta as lógicas para construção das estruturas (Termos e Cláusulas) do programa, *Engine.java,* que contém toda a lógica para preparação e execução de um programa, *ExecutionContext.java*, objeto que representa um contexto de execução, *Clause.java*,

objeto que representa uma cláusula e finalmente, *Term.java*, objeto que representa um termo.

7.2 PARSER

Para que um programa qualquer seja executado, independente do paradigma: imperativo, declarativo, procedural ou lógico, o mesmo deve conter algum valor semântico para seu executor, isto é, o programa deve ser interpretável, compreensível. Compreender corretamente um programa implica análises que devem ser realizadas sobre o mesmo. São basicamente três: *Análise Léxica*, *Análise Sintática* e *Análise Semântica*. *Parser* é o nome dado ao dispositivo que realiza a analise de um programa em termos léxicos, sintáticos e semânticos. Neste trabalho, foi construído um parser descendente recursivo preditivo LL(1), isto é, os símbolos funcionais da gramática foram programados como métodos Java que são recursivamente invocados durante a análise. O processo é determinístico, e assim, o método Java correspondente à uma determinada construção sintática é corretamente identificado à partir de um único token.

A *Análise Léxica* consiste no reconhecimento de símbolos expressos em um programa, também chamados de *Tokens*. Os Tokens válidos de uma linguagem são definidos em um conjunto de palavras especiais chamado *Alfabeto*. Uma vez que todos os Tokens de um programa são corretamente identificados como elementos do Alfabeto da linguagem, pode-se então realizar a Análise Sintática. Neste trabalho, os aspéctos léxicos da linguagem não foram alterados.

A *Análise Sintática* consiste no reconhecimento de *sentenças* formadas por sequências de Tokens que aparecem no programa. O universo das sentenças possíveis é definido por uma *gramática*, um instrumento formal que especifica as construções sintáticas que são válidas para uma determinada linguagem. Neste trabalho, as funcionalidades de análise léxica e análise sintática estão programadas na classe *Parser.java*. Novas construções sintáticas foram definidas para permitir a representação de operadores modais como listas de termos Prolog que antecedem termos ou cláusulas.

A Terceira e última etapa que é realizada é a *Análise Semântica*. A Análise Semântica consiste em identificar os *significados* dos tokens e sentenças reconhecidas nas etapas anteriores. Somente após a validação semântica, um programa pode ser classificado como válido ou não para uma determinada linguagem. É a partir desse ponto que um programa pode ser compilado (transcrito para código de máquina) ou interpretado, de acordo com a linguagem e ambiente de execução. Neste trabalho, as funcionalidades de análise semântica estão programadas na classe *SemanticParser.java.*

Para a inclusão de operadores modais na linguagem Prolog, não se fez necessário o uso de novos símbolos léxicos. Com o mesmo alfabeto do Prolog clássico, os operadores modais são introduzidos através de apenas novas construções sintáticas adicionadas à gramática. Tais construções permitem que modalidades sejam declaradas na frente de termos e na frente de blocos de cláusulas na forma de listas. Abaixo alguns exemplos:

Exemplo de termos com modalidades:

```
[pos]: exemplo_de_atomo
[nes,nes]: exemplo_de_estrutura(atomo).
```

Exemplo de blocos de cláusulas com modalidades:

```
[pos] { clausula. }
[nes,nes]
{
        fato1.
        regra1.
        fato2.
}
```

A gramática completa especificada neste trabalho está disponível no Apêndice A.

7.3 TERMOS E CLÁUSULAS

Dizer que um programa é interpretado, não necessariamente significa dizer que seu texto de código fonte é consultado a cada passo de sua execução. Na verdade, quando um programa é submetido à análises no Parser, não somente sua validação em termos léxicos, sintáticos e semânticos é realizada, mas também, uma estrutura de dados para representação do mesmo é construída em memória. Assim, a interpretação de um programa não se dá diretamente no texto de entrada, mas sim à partir dessa nova estrutura que é criada pelo Parser.

O Parser implementado neste trabalho, está fundamentado em duas principais estruturas para representação de um programa: Termos e Cláusulas. Um programa é definido como um conjunto de cláusulas, que por sua vez, são formadas por termos. Assim, o Parser implementado atua fazendo uma leitura completa do texto de entrada, construindo em memória, para cada termo e cláusula reconhecida, um objeto que o representa na estrutura do programa. Neste trabalho, os objetos cláusulas são definidos como instâncias da classe *Clause.java*, e os objetos termos são definidos como instâncias da classe *Term.java*.

Quando um programa é carregado, é criada uma lista de cláusulas, onde posteriormente são armazenados os objetos do tipo Cláusula do programa. Objetos do tipo Cláusula, são formados por Termos e representam os fatos e regras reconhecidos pelo parser no processo de análise. Um Termo é um objeto que pode representar um valor numérico, um átomo, uma estrutura, uma String, uma lista ou ainda uma variável.

Genericamente, um objeto Cláusula é composto por um termo, chamado Cabeça e uma lista de termos chamada Corpo. Uma Regra, é um tipo de cláusula dita completa, por apresentar tanto cabeça quanto corpo, diferente de um fato, que apresenta apenas cabeça. Há ainda um terceiro tipo de cláusula: a consulta. Uma consulta não faz parte das declarações de um programa, mas por sua vez, representam o ponto inicial da execução do mesmo. Formada apena por uma lista de termos (submetas), uma consulta é um tipo de cláusula que só apresenta corpo e que não é incluída na estrutura de dados

de representação do programa. Os termos construídos na análise semântica de uma consulta são empilhados como submetas que então o programa possa ser executado.

7.4 REPRESENTAÇÃO DE MODALIDADES

Este trabalho propõe a representação de modalidades através de listas Prolog. Uma modalidade consiste em uma sequência de operadores modais, que são representados por termos do tipo átomo ou estrutura. Assim como na proposta de *Linh Anh Nguyen*, este trabalho utiliza o conceito de rotulação de operadores modais para representação de mundos do modelo de Kripke através das modalidades. Abaixo, alguns exemplos de representação de modalidades são mostrados:

[nes, pos] – modalidade formada apenas por átomos

[nes(2), pos(1)] – modalidade formada por estruturas, onde o argumento representa o índice do operador modal. (Para lógicas Multimodais)

[pos(X), pos] – modalidade formada por uma estrutura e um átomo. Na estrutura, X caracteriza uma variável em um operador modal na forma rotulada.

[pos(2,X), pos(1,Y)] – modalidade formada por estruturas, onde o primeiro argumento representa o índice do operador modal (para lógicas Multimodais) e o segundo argumento caracteriza um operador rotulado.

O interpretador Prolog definido neste trabalho reconhece sintaticamente modalidades em dois contextos distintos: na frente de um termo, ou na frente de um bloco de cláusulas. Semanticamente. Uma modalidade de termo expressa o contexto modal de um termo específico, enquanto as modalidades de blocos de cláusulas são válidas para cada cláusula do bloco. Ainda, modalidades de cláusula, são distribuídas para seus termos, como visto nos exemplos abaixo:

[nes]: p(X) – expressa que o termo p(X) é necessário no mundo inicial.

[nes(2), pos(1)]: p(a) – expressa que à partir do mundo inicial, 0, para qualquer mundo w, acessível de 0 através da relação de acessibilidade R_2, existe um mundo acessível à partir de w através da relação de acessibilidade R_1, u, tal que em u, o termo p(a) é válido.

[pos(p(a))]: p(a) – expressa que existe um mundo acessível, w, à partir do mundo atual, 0, tal que p(a) é válido em w. O mundo w fica denotado pela rotulação do operador pos.

```
[nes]
{
        p(a).
        [pos]: q(X) :- p(X), [pos]:r(X).
}
```

é equivalente à:

```
[nes]: p(a)
[nes, pos]: q(X) :- [nes]:p(X), [nes, pos]:r(X).
```

7.5 BUILT-IN PREDICATES, DIRETIVAS E OPERADORES

Built-in Predicates

No Prolog, existem alguns predicados especiais, que quando encontrados na pilha de metas durante a execução de uma consulta, não somente são comparados com outros para unificação, mas que também executam alguns procedimentos específicos. Esses predicados são chamados de *Built-in Predicates*, e recebem esse nome, por serem tratados diferentemente dos demais predicados.

Um bom exemplo de um *Built-in Predicate* é o predicado *consult/1*, que representa o procedimento de carregamento de um programa como um termo Prolog. Quando uma estrutura consult, com um único argumento (aridade = 1) aparece como a submeta corrente a ser provada, o interpretador Prolog trata de forma especial tal

predicado, executando um procedimento que, literalmente, consulta um arquivo em disco e carrega suas cláusulas para o ambiente. O procedimento que faz o carregamento do arquivo não está programado em Prolog, mas sim em um nível mais baixo de abstração, como um procedimento do próprio interpretador.

Neste trabalho, os *Built-in Predicates* são predicados especiais que são mapeados para métodos Java em uma estrutura *Hashtable* que associa *Strings* à *Métodos*. Os métodos são definidos em uma classe chamada *BuiltInPredicates.java*, e as Strings que os identificam são descrições da estrutura na forma *functor/aridade*. Existe uma regra fundamental para que essa vinculação seja automatizada: o método definido como um *Built-in Predicate* deve ser declarado com a seguinte assinatura: "*public static void functor_aridade(Term term)*" para estruturas ou "*public static void átomo_0(Term term)*" para átomos.

Na inicialização do interpretador, é executado um procedimento que faz o registro de todos os métodos definidos como um built-in predicate na estrutura de mapeamento. Assim, cada método estático de assinatura "*public static void functor_aridade(Term term)*" é identificado pela String *"functor/aridade"* e cada método estático de assinatura "*public static void átomo_0(Term term)*" é identificado pela String *"átomo"*. Abaixo, é mostrado um exemplo de uma tabela de mapeamento.

Chave (functor/aridade)	Valor (método Java)
"consult/1"	public static void consult_1(Term term)
"write/1"	public static void write_1(Term term)
"sayHello"	public static void sayHello_0(Term term)

Quadro 10: Exemplos de Built-in Predicates

Com a tabela de mapeamento que associa assinatura de termos à métodos Java inicializada, identificar e executar um método correspondente a um *Built-in Predicate* durante a execução uma consulta se torna uma tarefa trivial. Para cada submeta e cabeça de cláusula selecionadas para unificação, o interpretador verifica se nos termos em questão está definido algum *Built-in Predicate,* e se estiver, invoca o método correspondente, passando o próprio termo como único argumento para o método. Assim, eventuais transformações podem ser feitas no termo, como exemplificado abaixo.

Como exemplo, considerar que um predicado soma/2 está definido como um *Built-in Predicate*, e que em um determinado momento da execução de um programa, aparece um termo *soma(5,10)* como uma submeta a ser provada. Neste momento, o programa busca um método para tratar o predicado soma/2, e o invoca, passando o termo *soma(5,10)* como argumento. O método "*public static void soma_2(Term term)*" identifica os 2 argumentos 5 e 10 e realiza a soma. Ao final de sua execução, o método correspondente ao predicado atualiza o termo referenciado, que deixa de ter o conteúdo soma/2 e passa a assumir o valor constante 15.

Todos os *Built-in Predicates* programados neste trabalho estão disponíveis no Apêndice B.

Diretivas

No momento que um programa é carregado no interpretador Prolog, cada cláusula reconhecida do tipo Fato ou Regra é adicionada à uma coleção de cláusulas que representam o programa em memória. Entretanto, existe um tipo especial de cláusula que não é adicionada como um Fato ou uma Regra na lista de cláusulas do programa, mas sim executada imediatamente de forma similar a um *Built-in Predicate*. Essas cláusulas especiais recebem o nome de *diretivas*, e são constituídas apenas por corpo da cláusula, sem cabeça.

Introduzidas pelo símbolo ':-', as *diretivas* são utilizadas geralmente para se alterar configurações e opções do interpretador. No momento que o Prolog reconhece uma cláusula como uma *Diretiva*, um método Java correspondente é imediatamente invocado para processá-la. Por exemplo, neste trabalho foram implementadas algumas *diretivas* que permitem definir configurações de *tracing*, declarar classes de estruturas para lógicas modais, declarar operadores modais, entre outras. As diretivas são programadas na classe *Directives.java.*

Exemplos de Diretivas implementadas:

:- setFlag('PRINT_GOAL_STACK','true').

a diretiva *setFlag* é mapeada para um método Java que altera propriedades definidas no interpretador. Neste exemplo, a função que imprime a pilha de submetas durante a execução de uma consulta é habilitada.

:- modalOP('UNIVERSAL', 'nes').

a diretiva *modalOP* é mapeada para um método Java que declara operadores modais utilizados no programa. Isso é importante para que ao se carregar programas de lógica modal, procedimentos especiais, que são explicados adiante, sejam corretamente executados.

Todas as *Diretivas* programadas neste trabalho estão disponíveis no Apêndice C.

Operadores

Operadores são comumente utilizados em expressões, por exemplo, aritméticas e lógicas, na forma de símbolos especiais introduzidos entre os termos. Em uma expressão, um operador pode aparecer em 3 situações distintas: entre dois termos, antes de um único termo, ou depois de um único termo. Operadores que aparecem entre dois termos são chamados de *'operadores infixos' (xfx)*; para aqueles que aparecem imediatamente antes do termo em que atuam, denota-se *'operadores pré-fixos' (fx)*; e para os operadores que aparecem imediatamente depois do termo em questão, denota-se *'operadores pós-fixos' (xf)*.

Neste trabalho, é possível se declarar operadores através de uma diretiva específica: *op/3*. A diretiva op de aridade 3, é interpretada como um método Java que é executado com 3 respectivos argumentos: *prioridade do operador*, *tipo de operador (infixo, pré-fixo ou pós-fixo)* e *símbolo do operador.* Quando um programa é carregado em memória as expressões de termos, com base nas declarações de operadores, são convertidas em um termos do tipo estrutura que as representam. Para a construção desses termos, as prioridades dos operadores são respeitadas como no seguinte exemplo:

Considerar as seguintes diretivas:

```
:- op(100, 'xfx', '*').
:- op(200, 'xfx', '+').
```

Considerar a seguinte expressão:

```
5 + 10 * 2.
```

Como resultado, a seguinte estrutura é definida:

```
+(5, *(10, 2)).
```

No exemplo acima, é possível perceber que o Prolog utiliza uma representação estruturada para uma expressão aritmética. No entanto, essa expressão não tem qualquer forma de ser avaliada definida ainda. Para o Prolog, o termo +(5, *(10, 2)) somente será verdadeiro se o mesmo puder ser provado no programa.

Neste trabalho, os operadores aritméticos e relacionais básicos são implementados como *Built-in Predicates*. Dessa forma, um termo dos tipos +/2 e */2 são executados como métodos Java que realizam as operações de dois termos resumindo-se ao valor constante obtido. O exemplo abaixo ilustra esta situação:

+(5, *(10, 2)). → +(5, 20). → 25.
(os termos mais internos são primeiramente resolvidos)

Uma observação importante que deve ser feita neste momento, é que não é possível se definir um método Java de nome '+_2' ou '*_2', por exemplo, devido a restrição da linguagem em relação à regras de formação de identificadores válidos. Neste trabalho, a estratégia adotada para se contornar essa situação foi permitir a atribuição de um *'alias'* para *Built-in Predicates* na estrutura de mapeamento. Assim, o *Built-in Predicate "public static void sum_2(Term term)"*, correspondente ao termo

"sum/2" pode também ser associado ao termo *"+/2"*. Da mesma forma, o *Built-in Predicate "public static void mul_2(Term term)"* passa a ser válido para o termo *"*/2"*.

Todos os *Operadores* programados neste trabalho estão disponíveis no Apêndice C.

7.6 RESOLUÇÃO E UNIFICAÇÃO

O elemento principal da execução de uma consulta é o cálculo de predicados, que se fundamenta nos conceitos de resolução e unificação, e é iniciado quando se submete uma consulta ao programa. Neste trabalho, quando uma consulta é estabelecida pelo usuário, o interpretador executa um método que prepara o contexto para execução: estruturas de dados auxiliares são reinicializadas e os termos do corpo da consulta são colocados na pilha de submetas a serem provadas. Os elementos necessários para a execução de uma consulta são programados nas classes *Engine.java e ExecutionContext.java.*

Uma *pilha de prova* é definida à partir de uma consulta estabelecida. O conceito de pilha é utilizado, por que os termos das consultas são processados sequencialmente da esquerda para direita, o que equivale na estrutura pilha, do topo para baixo. Assim, quando uma submeta é substituída por novas submetas através de uma regra, remove-se a submeta antiga do topo da pilha e empilha-se os novos termos definidos no corpo da regra, originando um novo contexto.

Um *objeto de contexto* é uma estrutura utilizada para se guardar informações necessárias para o mecanismo de *backtracking*. Uma vez que à partir de uma consulta é possível se percorrer vários caminhos distintos na busca de tentar prová-la, pode-se visualizar uma consulta original como um nodo raiz de uma árvore que represente todos esses caminhos possíveis. Assim, cada nodo dessa árvore representa um contexto de prova, isto é, representa todas as informações de um determinado momento da execução: quais submetas ainda devem ser provadas, quais instanciações já foram realizadas, qual a próxima cláusula a ser testada e principalmente, para onde retroceder se a consulta falhar.

Neste trabalho, o conceito de *pilha de prova*, onde as submetas de uma consulta são representadas está definido como um atributo de um objeto *Contexto de Execução,* que é definido na classe *ExecutionContext.java* e que representa as informações de um determinado instante do cálculo de predicados. O motor de inferência implementado no interpretador na classe *Engine.java*, por sua vez, utiliza um conceito de pilha de contextos. Assim, quando uma instanciação ou substituição é realizada, um novo contexto é empilhado para se continuar o processo. Analogamente, quando o mecanismo de *backtracking* é acionado, o interpretador simplesmente descarta o último contexto inserido na pilha, voltando assim ao contexto imediatamente anterior para se buscar outro caminho.

Quando se deseja executar uma consulta, após a inicialização das estruturas de controle de fluxo, pilha de contextos e pilha de prova e identificação das variáveis livres, o programa inicia a execução invocando um método que é utilizado para se provar a submeta corrente (ao topo da pilha de prova) do contexto corrente (ao topo da pilha de contextos). De uma forma geral, este método percorre a lista de cláusulas (fatos e regras) do programa, buscando unificações plausíveis de submetas com cabeças de cláusulas. O processo de unificação está definido em um outro método, que retorna um valor booleano indicando se uma comparação é ou não bem sucedida, e se sim, o método ainda atualiza as anotações de eventuais instanciações realizadas.

Ao se retornar com sucesso do procedimento de unificação para o método que prova as submetas da consulta, o programa verifica se foi realizada alguma instanciação para alguma variável da submeta ou da cláusula testada. Se *sim*, o programa primeiramente substitui as variáveis instanciadas pelos novos valores atribuídos e depois inicia um novo contexto substituindo a submeta corrente pelo corpo da cláusula testada. Se a unificação for bem sucedida sem *nenhuma instanciação* realizada, o programa simplesmente substitui a submeta corrente no topo da pilha de prova pelo corpo da cláusula testada. Já se a unificação falhar para todas as cláusulas do programa, o contexto corrente é descartado através do mecanismo de backtracking, e o programa volta a trabalhar com o contexto anterior, buscando um caminho alternativo de prova.

Abaixo são listados os principais métodos envolvidos na lógica da execução de uma consulta neste trabalho.

*Engine.**runQuery()*** – Método chamado quando se inicia uma consulta. É este método que prepara as estruturas de dados para execução de uma nova consulta, inicializa um contexto inicial, identifica as variáveis da consulta e por fim invoca o método runSubgoal() para iniciar a execução.

*Engine.**runSubgoal()*** – Método responsável pelo processamento da consulta em si. É este método que manipula a pilha de contextos, invoca as unificações, substitui variáveis instanciadas, empilha novas submetas criando novos contextos, executa o *backtracking* retrocedendo a um pondo anterior da prova, entre outras ações.

*Engine.**resolveTerm**(Term term)* – Método responsável pela execução de *Built-in Predicates* e avaliação de expressões. É constantemente utilizado pelo método *runSubgoal()* durante a execução de consultas.

*Engine.**unifyTerms**(Term t0, Term t1, Hashtable instantiations)* – Método chamado para realizar as unificações de termos durante o processamento de uma consulta. Constantemente invocado pelo método *runSubgoal()*, este método faz a comparação de termos um a um, recursivamente para listas, termos estruturados, e modalidades, anotando todas as instanciações necessárias no *Hashtable instantiations* passado por parâmetro. As anotações de instanciações são, posteriormente, processadas pelo método *runSubgoal()* para formação de novos contextos.

*ExecutionContext.**replaceCurrentSubgoal**(ArrayList<Term> newSubgoal)* – Método disponível em um objeto *ExecutionContext* capaz de substituir uma submeta corrente por um conjunto de novas submetas, termos definidos no corpo de uma regra que tenha sido ativada. No caso de uma unificação de uma submeta com um fato, este parâmetro será nulo, e dessa forma, o objeto de contexto apenas remove a submeta corrente de sua pilha de prova, considerando, assim, a submeta provada.

*ExecutionContext.**cleanUnusedVariables()*** – Método responsável pela limpeza de variáveis não mais utilizadas em um objeto de contexto. Invocado pelo método

runSubgoal() da classe *Engine* depois que uma substituição proveniente de uma instanciação é realizada, este método remove variáveis temporárias que tenham sido instanciadas.

*ExecutionContext.**setInstantiatedVariable**(String variable, Term term)* – Método de um objeto de contexto que anota as instanciações temporárias resultantes de uma unificação calculada na classe *Engine* durante o processamento de uma consulta em um *Hashtable*.

ExecutionContext. ***replaceInstantiatedVariables****()* – Método invocado para a efetivação das instanciações temporárias anotadas no *Hashtable* correspondente. Este método primeiramente monta uma lista de todas as variáveis encontradas no contexto que o executa e em um segundo momento, substitui todas as variáveis possíveis por seus respectivos valores de instanciações.

7.7 REPRESENTAÇÃO DE UM PROGRAMA MODAL

Este trabalho está apto a tratar alguns problemas de lógicas modais e multimodais normais que apresentem os axiomas K, D, 4, 5 e I. Como apresentado nos itens anteriores, sintaticamente, as modalidades são representadas como listas de operadores modais, representados como termos que antecedem os termos e cláusulas do programa. Quando utilizada para expressar o contexto modal de um termo, uma modalidade é inserida de acordo com a seguinte notação *[modalidade]:termo.* e quando uma modalidade é utilizada para expressar o contexto modal de uma cláusula, ou uma lista de cláusulas, esta aparece na forma [modalidade] { (clausulas)*. }.

Também é necessário a utilização de diretivas especiais para se especificar como o interpretador deve reconhecer os programas representativos de lógicas modais. Estas diretivas são a *modalOP/2,* utilizada para se declarar os operadores modais universais e existenciais e a *modalLOGICS/1*, utilizada para se declarar a classe de estrutura em que o programa deve ser interpretado. Estas diretivas devem aparecer no programa sempre antes da primeira cláusula do programa, por que estas provocam algumas mudanças na forma que uma cláusula é reconhecida e incluída na coleção de cláusulas do programa.

Uma vez declarada a classe de problema que se deseja tratar e os nomes dos operadores modais utilizados, pode-se iniciar o carregamento de cláusulas para a estrutura de dados que as representam em memória. Na verdade, quando uma cláusula é reconhecida no parser, não somente esta é incluída na coleção de cláusulas do programa, mas também algumas variações obtidas por métodos especiais também o são. As combinações geradas refletem os axiomas utilizados na classe de estrutura do problema modal. Abaixo são mostrados exemplos de derivação de cláusulas para ilustrar:

Derivação de cláusulas definidas em um bloco de cláusulas (Distributividade):

[nes] { p(X) :- [nes]: q(X). } → *[nes]: p(X) :- [nes, nes]: q(X).*

Derivação de uma regra (rotulação de operadores):

[nes(1)]: p(X) :- [pos(2)]: q(X) → *[pos(1,W)]: p(X) :- [pos(2,U)]: q(X).*

Derivação de um fato (rotulação de um termo sem variável):

[pos]: p(a). → *[pos(p(a))]: p(a).*

Por fim, regras adicionais, de transformação de contexto modal para submetas, são incluídas na coleção de cláusulas que representam os fatos e regras declarados no programa. Estas regras são também geradas pelo interpretador no momento que um programa do usuário é carregado, de acordo com a classe de lógica modal do programa especificado pela diretiva *modalLOGICS*. Abaixo alguns exemplos de regras geradas:

Regras geradas para o Axioma 4:

[nes]: p(X) → *[nes, nes]: p(X).*
[nes(I)]: p(X) → *[nes(I), nes(I)]: p(X).*

Regras geradas para o Axioma 5:

[pos]: p(X) → [nes, pos]: p(X).
[pos(I)]: p(X) → [nes(I), pos(I)]: p(X).
[pos(I,W)]: p(X) → [nes(I), pos(I,W)]: p(X).

Neste trabalho, os métodos que atuam na derivação das cláusulas originalmente declaradas no programa são listados abaixo:

*SemanticParser.**generateRuleDerivations**(Clause clause)* – Método da classe *SemanticParser* responsável por gerar todas as derivações possíveis à partir de uma cláusula, de acordo com o sistema de lógica modal em questão. Faz as chamadas para os métodos correspondentes a cada Axioma presente na classe do problema.

*SemanticParser.**generateAxiomKRuleDerivations**(Clause clause)* – Método da classe *SemanticParser* responsável por gerar as derivações de regras relativas ao axioma K. Para qualquer lógica modal, este método é sempre chamado, já que o axioma K é base para uma lógica modal normal, tratada neste trabalho.

SemanticParser. ***generateAxiomDRuleDerivations*** *(Clause clause)* – Método da classe *SemanticParser* responsável por gerar as derivações de regras correspondentes ao axioma D.

SemanticParser. ***generateAxiom4RuleDerivations*** *(Clause clause)* – Método da classe *SemanticParser* responsável por gerar as derivações de regras correspondentes ao axioma 4.

SemanticParser. ***generateAxiom5RuleDerivations*** *(Clause clause)* – Método da classe *SemanticParser* responsável por gerar as derivações de regras correspondentes ao axioma 5.

SemanticParser. ***generateAxiomIRuleDerivations*** *(Clause clause)* – Método da classe *SemanticParser* responsável por gerar as derivações de regras correspondentes ao axioma I para lógicas multimodais.

7.8 RESOLUÇÃO APLICADA A UM PROGRAMA MODAL

Como já mostrado, neste trabalho, as modalidades são introduzidas como listas de termos Prolog que representam um contexto modal para os termos declarados nas cláusulas dos programas. Dessa forma, tratar as unificações e instanciações durante a execução de uma consulta de um programa de lógica modal não é tão diferente de se trabalhar com o cálculo de predicados em sua forma original, exceto por uma única alteração que se faz necessária para que o interpretador trate corretamente os operadores modais que são representados nas listas de modalidades.

A estratégia adotada para que o motor de resolução considere corretamente as modalidades vinculadas aos termos declarados no programa durante sua execução é interpretá-las como argumentos de um termo estruturado qualquer. Assim, o mecanismo de instanciação se mantém inalterado quando comparado ao mecanismo de instanciação clássico. A única nova regra, que é implementada no método de unificação, faz com que que ao se comparar dois termos quaisquer, suas modalidades também sejam comparadas da mesma forma argumentos são para listas ou estruturas.

Dessa forma, o principal mecanismo que torna possível a execução de programas de lógica modal neste trabalho é executado durante o carregamento de um programa. Este mecanismo consiste na geração de derivações das cláusulas e na geração de regras de transformação de contexto modal para as submetas. Tanto as cláusulas derivadas geradas, quanto as regras de transformação modal incluídas no programa são corretamente fundamentadas na classe do problema modal especificado no programa pela diretiva *modalLOGICS*.

8 TESTES E VALIDAÇÃO

A validação da ferramenta desenvolvida neste trabalho foi dada através da execução de uma bateria de testes realizados sobre programas clássicos de lógica de primeira ordem e programas de lógica modal apresentados no trabalho *MProlog* de *Linh Anh Nguyen.* Neste capítulo são apresentados alguns exemplos de execução de consultas desses programas.

8.1 PROGRAMAS DE LÓGICA CLÁSSICA

8.1.1 PROGRAMA DA ÁRVORE GENEALÓGICA

O programa da Árvore Genealógica é frequentemente utilizado para se ilustrar as capacidades de inferência e instanciação presentes no Prolog. Abaixo são apresentadas as cláusulas do programa, uma consulta estabelecida e todos os passos que o interpretador realiza durante a resolução de uma consulta. O programa *ex_family.pl* é fornecido no Apêndice D.

Cláusulas do Programa:

1. pai(antonio, bruno).
2. pai(bruno, carlos).
3. pai(carlos, daniel).
4. avo(X,Y) :- pai(X,Z), pai(Z,Y).
5. bisavo(X,Y) :- pai(X,Z), avo(Z,Y).

Consulta Estabelecida:

“quem é o bisavô de Daniel?” → *bisavo(X, daniel).*

Resolução:

Pilha de Prova	Cláusulas / Regras	Instanciações
bisavo(X, daniel).	5	
pai(X,B), avo(B,daniel).	1	{ X/antonio, B/bruno }
avo(bruno,daniel).	4	
pai(bruno,C), pai(C,daniel).	2	{ C = carlos }
pai(carlos, daniel).	3	
{}		

Quadro 11: Execução de uma consulta: Programa da Árvore Genealógica

Resultados:

Após a execução da consulta, a pilha de submetas se encontra vazia e a variável X está instanciada com o átomo *antonio*. Isto quer dizer que *"Antonio é o bisavô de Daniel"* é uma resposta para consulta.

Comentários:

A execução do programa é iniciada quando a consulta "bisavo(X, daniel)." é submetida pelo usuário. Nesse momento, o interpretador reseta as estruturas de controle de fluxo, carrega os termos da nova consulta para a pilha de metas e inicia o processo do cálculo de predicados que tenta provar a consulta. O primeiro termo da consulta estabelecida, neste caso, o único, "bisavo(X, daniel)." unifica com a regra 5, e então é substituído pelo seu corpo: "pai(X,B), avo(B,daniel).". O processo de resolução se repete ativando na sequência as regras: 1, 4, 2 e 3.

Como é possível perceber, um valor possível para a variável X é encontrado durante a unificação da submeta "pai(X,B)" com o fato (1) "pai(antonio, bruno)". A instanciação é válida durante toda a execução do programa, e assim, a consulta pode ser provada, ou seja, a mesma é aceita no programa com a variável X instanciada com o valor antonio.

8.1.2 PROGRAMA DE EXEMPLO DE LISTA

O programa de Exemplo de Lista é apresentado para se ilustrar as formas possíveis de se declarar uma lista em Prolog e como as funções de manipulação de listas, como por exemplo, insere e remove, para o primeiro elemento da lista podem ser programadas utilizando regras Prolog. O programa *ex_list.pl* é fornecido no Apêndice E.

Cláusulas do Programa:

1. nova_lista([]).
2. insere([], X, [X]).
3. insere(L, X, [X|L]).
4. remove([H|T], H, T).

Consulta Estabelecida:

nova_lista(L), insere(L, 1, L1), insere(L1, 2, L2), remove(L2, H, L3).

Resolução:

Pilha de Prova	Cláusulas / Regras	Instanciações
nova_lista(L), insere(L, 1, L1), insere(L1, 2, L2), remove(L2, H, L3).	1	{ L/[] }
insere([], 1, L1), insere(L1, 2, L2), remove(L2, H, L3).	2	{ L1/[1] }
insere([1], 2, L2), remove(L2, H, L3).	3	{ L2/[2, 1] }
remove([2, 1], H, L3).	4	{ H/2, L3/[1] }
{}		

Quadro 12: Execução de uma consulta: Programa de Exemplo de Lista

Resultados:

Ao término da execução da consulta, isto é, após inicializar uma nova lista, adicionar o elemento 1, adicionar o elemento 2 e remover o elemento da primeira posição, o programa exibe os seguintes resultados: L = [], L1 = [1], L2 = [2,1], L3 = [1] e H = 2.

Comentários:

Neste exemplo, novamente, a execução do programa se inicia com o carregamento da consulta estabelecida pelo usuário para as estruturas de controle do interpretador. Feito isso, o cálculo de predicados é iniciado para se tentar provar a consulta. Começando pela primeira submeta, "nova_lista(L)", o interpretador unifica com o fato (1) "nova_lista([])", instanciando a variável L com uma lista vazia [].

Na sequência, as regras 2, 3 e 4 são ativadas, até que ao final, a pilha de provas se encontra vazia, o que indica uma consulta bem sucedida. Neste momento a consulta é dita provada, e assim, aceita no programa.

8.1.3 PROGRAMA DO CÁLCULO DE FATORIAL

O programa do Cálculo de Fatorial é um outro exemplo clássico utilizado no Prolog por ilustrar um problema onde há intenso uso de recursão para sua solução. Na forma declarativa, o programa é simples de ser representado. Sua execução é mostrada abaixo. O programa *ex_fact.pl* é fornecido no Apêndice F.

Cláusulas do Programa:

1. fact(0, 1).
2. fact(N, F) :- fact(N – 1, F1), F is N * F1.

Consulta Estabelecida:

"quanto é o fatorial de 3?" → *fact(3, X).*

Resolução:

Pilha de Prova	Cláusulas / Regras	Instanciações
fact(3, X).	2	
fact(2, F1), X is 3 * F1	2	
fact(1, F2), F1 is 2 * F2, X is 3 * F1	2	
fact(0, F3), F2 is 1 * F3, F1 is 2 * F2, X is 3 * F1	1	
F2 is 1 * 1, F1 is 2 * F2, X is 3 * F1	-	{ F2/1 }
F1 is 2 * 1, X is 3 * F1	-	{ F1/2 }
X is 3 * F1	-	{ X/6 }
{}		

Quadro 13: Execução de uma consulta: Programa do Cálculo de Fatorial

Resultados:

O valor correspondente ao fatorial de 3, calculado recursivamente no programa, é atribuído à variável X ao final da execução.

Comentários:

A execução do programa é iniciada quando a consulta submetida pelo usuário, “fact(3, X).”, é carregada para a pilha de metas e o processo do cálculo de predicados é iniciado para se tentar provar a consulta. Na sequência, a primeira submeta, “fact(3, X).” unifica com a regra 2, e então é substituída pelo seu corpo: “fact(2, F1), X is 3 * F1.”.

Como é possível perceber, a regra 2 é repetidamente ativada para os termos recursivos: 3, 2 e 1, até que para o termo 0, o interpretador unifica com o fato 1: “fact(0, 1).”. Depois disso, os predicados “is” remanescentes na pilha de provas são resolvidos, até que por fim, a variável X é instanciada com o valor 6. Neste momento, a pilha de prova se encontra vazia, o que indica sucesso na execução da consulta.

8.2 PROGRAMAS DE LÓGICA MODAL

8.2.1 PROGRAMA DE INTRODUÇÃO À LÓGICA MODAL

O programa de Introdução à Lógica Modal é um exemplo básico que ilustra como uma lógica modal normal KD pode ser tratada no interpretador desenvolvido neste trabalho. O programa *ex_intro_modal.pl* é fornecido no Apêndice G.

Diretivas do Programa:

1. :- modalOP('UNIVERSAL', 'nes').
2. :- modalOP('EXISTENTIAL', 'pos').
3. :- modalLOGICS('KD').

Cláusulas do Programa:

1. [nes] { p(a). }
2. [nes] { q(a). }
3. [nes] { s(X) :- q(X), [pos]: t(X). }
6. [nes] { [pos]: t(a). }
7. [nes] { [pos]: t(b). }

Consulta Estabelecida:

"s(a) é possível em algum mundo acessível à partir do mundo inicial onde W seja verdade" → *[pos(W)]:s(a).*

Resolução:

Na tabela abaixo, considerar o símbolo * como uma derivação de cláusula gerada pelo interpretador no carregamento do programa.

Pilha de Prova	Cláusulas / Regras	Instanciações
[pos(W)]: s(a).	3*	
[pos(W)]: q(a), [pos(W), pos(U)]: t(a).	2*	{ W/q(a) }
[pos(p(a)), pos(U)]: t(a).	6*	{ U/t(a) }
{}		

Quadro 14: Execução de uma consulta: Programa de Introdução à Lógica Modal

Resultados:

Ao final da execução do programa, tem-se W instanciado com q(a), o que significa que em um mundo w, rotulado como pos(q(a)), acessível à partir do mundo atual (0), onde q(a) é válido e [pos]:t(a) pode ser verificado como verdade, a proposição s*(a)* também é válida.

Comentários:

Diferente dos exemplos mostrados anteriormente de programas de lógica clássica, em um programa de lógica modal, deve-se utilizar as diretivas "modalOP" e "modalLOGICS" para fazer a configuração adequada da ferramenta para o carregamento correto das cláusulas do programa.

A execução do programa é então iniciada quando a consulta submetida pelo usuário, "[pos(W)]: s(a)", é carregada para a pilha de metas e o processo do cálculo de predicados é iniciado para se tentar provar a consulta. A continuação do processo do cálculo de predicados é equivalente aos exemplos mostrados anteriormente: os termos da pilha de prova são trocados por regras que são sucessivamente ativadas até que a pilha se encontre vazia (sucesso), ou até que não seja mais possível retroceder em busca de um outro caminho alternativo de prova (falha).

O que se percebe de novo neste exemplo é o uso das modalidades na frente dos termos e cláusulas do programa. É possível notar também o símbolo * utilizado no quadro de regras ativadas. Este símbolo representa variações das cláusulas do programa que são automaticamente geradas pelo interpretador no momento em que estas são carregadas. As variações geradas são definidas de acordo com os parâmetros declarados nas diretivas especiais.

8.2.2 PROGRAMA DE EXEMPLO MULTIMODAL

O programa de Exemplo de Lógica Multimodal corresponde ao primeiro programa modal apresentado como exemplo no trabalho MProlog de *Linh Anh Nguyen*. Este programa ilustra como uma lógica multimodal pode ser tratada no interpretador desenvolvido neste trabalho. O programa *ex_multi_modal.pl* é fornecido no Apêndice H.

Diretivas do Programa:

1. :- modalOP('UNIVERSAL', 'nes').
2. :- modalOP('EXISTENTIAL', 'pos').
3. :- modalLOGICS('mKD').

Cláusulas do Programa:

1. [pos(1)] { p(a). }
2. [nes(1)] { [nes(2)]: q(X) :- p(X). }
3. [nes(1)] { [pos(2)]: r(X) :- [nes(2)]: q(X). }
4. [nes(1), nes(2)] { s(X) :- q(X), r(X). }
5. [nes(1)] { t(X) :- [pos(2)]: s(X). }

Consulta Estabelecida:

"t(a) é possível em algum mundo acessível à partir do mundo inicial?" → [W]: t(a).

Resolução:

Na tabela abaixo, considerar o símbolo * como uma derivação de cláusula gerada pelo interpretador no carregamento do programa.

Pilha de Prova	Cláusulas / Regras	Instanciações
[W]: t(a).	5*	{ W/pos(1,U) }
[pos(1,U), pos(2,V)]: s(a).	4*	

[pos(1,U), pos(2,V)]: q(a), [pos(1,U), pos(2,V)]: r(a).	2*	
[pos(1,U)]: p(a), [pos(1,U), pos(2,V)]: r(a).	1*	{ U/p(a) }
[pos(1,p(a)), pos(2,V)]: r(a).	3*	{ V/r(a) }
[pos(1,p(a)), pos(2,r(a))]: q(a).	2*	
[pos(1,p(a))]: p(X)	1*	
{}		

Quadro 15: Execução de uma consulta: Programa de Exemplo Multimodal

Resultados:

Ao final da execução do programa, tem-se W instanciado com *pos(1,p(a))*, o que significa que em um mundo possível w, onde p(a) é válido, acessível a partir do mundo atual (0), *t(a)* também é válido.

Comentários:

Os processos de inicialização e execução de consultas para este exemplo, novamente se assemelham com os processos de inicialização e execução de consultas mostrados anteriormente. O que é possível se notar de novo neste exemplo é o uso de uma lógica multimodal.

Para a representação de diferentes operadores modais, neste exemplo, os operadores modais existenciais e universais são respectivamente indexados como pos(I) e nes(I). Ainda, para que o programa interprete corretamente o termo de indexação (I), definido nos operadores, o argumento de tipo de lógica modal definido na diretiva "modalLOGICS" deve iniciar com a letra minúscula "m", de multimodal. Neste caso, a declaração da diretiva modalLOGICS("mKD") indica que o programa é multimodal e definido pelos axiomas K e D.

8.2.3 PROGRAMA DAS HABILIDADES MATEMÁTICAS

O programa das Habilidades Matemáticas trabalha com *diferentes graus de crença* e pode ser representado por uma lógica multimodal KDI45. Neste problema, os fatos e regras são declarados com diferentes graus de verdade. Abaixo são mostradas

suas cláusulas e o processo de resolução para uma dada consulta estabelecida. O programa *ex_maths_modal.pl* é fornecido no Apêndice I.

Diretivas do Programa:

1. :- modalOP('UNIVERSAL', 'n).
2. :- modalOP('EXISTENTIAL', 'p).
3. :- modalLOGICS('mKDI45').

Cláusulas do Programa:

1 [n(4)]: good_in_maths(X) :- maths_teacher(X).
2 [n(5)] { [n(I)]: good_in_maths(X) :- [n(I)]: mathematician(X). }
3 [n(3)] { [p(I)]: good_in_maths(X) :- maths_student(X). }
4 [n(3)] { [p(I)]: good_in_physics(X) :- physics_student(X). }
5 [n(2)] { [p(2)]: good_in_maths(X) :- good_in_physics(X). }
6 maths_teacher('John').
7 [n(2)]: mathematician('Tom').
8 [n(5)]: maths_student('Peter').
9 [n(5)]: physics_student('Mike').

Consulta Estabelecida:

"Tom é bom em matemática com grau de crença 2?" → *[n(2)]: good_in_maths('Tom').*

Resolução:

Na tabela abaixo, considerar o símbolo C* como uma derivação de cláusula (C) gerada pelo interpretador no carregamento do programa e * como uma regra de transformação modal válida para a classe do modelo do problema.

Pilha de Prova	Cláusulas / Regras	Instanciações
[n(2)]: good_in_maths('Tom').	*	
[n(2), n(2)]: good_in_maths('Tom').	2*	
[n(2), n(2)]: mathematician('Tom').	7*	
{}		

Quadro 16: Execução de uma consulta: Programa das Habilidades Matemáticas

Resultados:

O programa prova a consulta, isto é, prova que Tom é bom em matemática com grau de crença 2 para a lógica modal considerada (KDI45). Nenhuma instanciação é apresentada, uma vez que na consulta não há presença de variáveis livres.

Comentários:

Este programa ilustra uma aplicação prática de uma lógica modal para representação de um problema que trata de *diferentes graus de crença*. Neste programa, representado por uma lógica multimodal KDI45, os operadores modais são rotulados de acordo com o grau de válidade que expressam nas cláusulas e termos.

Através do axioma I, definido no problema, é possível se generalizar a aceitação de cláusulas declaradas com grau de verdade maiores para graus de verdade menores. Por exemplo, se uma proposição p é declarada com grau de verdade 5 – *[n(5)]: p.* – p também deve ser aceita com graus de verdade 4, 3, 2 e 1.

8.2.4 PROGRAMA DOS GOSTOS POSSÍVEIS

O programa dos Gostos Possíveis trabalha com *crenças de diferentes agentes* e pode ser representado por uma lógica multimodal KD45. Neste problema, procura-se modelar regras que permitam inferir a possibilidade de que um determinado indivíduo goste de um determinado produto à partir dos conhecimentos pressupostos de 3 agentes. Abaixo são mostradas suas cláusulas e o processo de resolução para uma dada consulta estabelecida. O programa *ex_likes_modal.pl* é fornecido no Apêndice G.

Diretivas do Programa:

1. :- modalOP(‘UNIVERSAL’, ‘nes’).
2. :- modalOP(‘EXISTENTIAL’, ‘pos’).
3. :- modalLOGICS(‘mKD45’).

Cláusulas do Programa (apenas as necessárias):

1. [nes(1)] { likes(jan, cola). }
2. [nes(2)] { likes(jan, pepsi). }
3. [nes(2)] { likes(X, cola) :- likes(X, pepsi). }
4. [nes(3)] { likes(jan, cola). }
5. [nes(3)] { very_much_likes(X, Y) :- likes(X,Y), [nes(1)]: likes(X,Y), [nes(2)]: likes(X,Y). }
6. very_much_likes(X, Y) :- [nes(3)]: very_much_likes(X, Y).

Consulta Estabelecida:

"tem alguém que goste muito de alguma coisa?" → *very_much_likes(X,Y).*

Resolução:

Na tabela abaixo, considerar o símbolo C* como uma derivação de cláusula (C) gerada pelo interpretador no carregamento do programa e * como uma regra de transformação modal válida para a classe do modelo do problema.

Pilha de Prova	Cláusulas / Regras	Instanciações
very_much_likes(X, Y).	6	
[nes(3)]:very_much_likes(X, Y).	5	
[nes(3)]:likes(X,Y), [nes(3), nes(1)]:likes(X,Y), [nes(3), nes(2)]:likes(X,Y).	4	{ X/jan, Y/cola }
[nes(3), nes(1)]:likes(jan, cola), [nes(3), nes(2)]:likes(jan, cola).	*	
[nes(1)]:likes(jan, cola), [nes(3), nes(2)]:likes(jan, cola).	1	
[nes(3), nes(2)]:likes(jan, cola).	*	
[nes(2)]:likes(jan, cola).	3	
[nes(2)]:likes(jan, pepsi).	2	
{}		

Quadro 17: Execução de uma consulta: Programa dos Gostos Possíveis

Resultados:

Ao final da execução da consulta, o sistema apresenta como resposta os valores de instanciação jan para a variável X e cola para a variável Y. Isto quer dizer que a asserção *"Jan gosta muito de cola"* é válida no programa.

Comentários:

Este programa ilustra uma aplicação prática de uma lógica modal para representação de um problema que trata de *crenças de diferentes agentes*. Neste programa, representado por uma lógica multimodal KD45, os operadores modais são rotulados de acordo com a identificação do agente para o qual uma cláusula ou termo é válida.

9 CONCLUSÕES

9.1 CONSIDERAÇÕES FINAIS

Este trabalho teve início com um estudo sobre programação em lógica utilizando a linguagem Prolog. Os principais elementos desse paradigma e dessa linguagem foram detalhadamente explicados nos capítulos 3 e 4, desde suas estruturas de representação, resumidos à termos e cláusulas, aos múltiplos procedimentos executados durante a execução de uma consulta, como resolução com unificação e instanciação. Foi mostrado também como acontece o controle de fluxo no Prolog e como atua o mecanismo do backtracking.

Durante o estudo realizado sobre o Prolog, pôde-se perceber uma das mais significantes diferenças entre um programa Prolog e um problema puramente lógico: "*o not no Prolog não equivale ao not puramente lógico"*. Isso ocorre devido à uma pressuposição básica que é feita acerca de um programa Prolog que diz "*o que é declarado no programa é verdadeiro. Enquanto o que não é declarado, não necessariamente é falso.*", denominada *Pressuposição de Mundo Fechado*. Com isso, o Prolog não é caracterizado por um sistema *verdadeiro / falso*, mas sim, por um sistema *verdadeiro / falha*.

Paralelamente ao estudo feito sobre a linguagem Prolog, foi realizado também um estudo sobre os fundamentos teóricos sobre Lógica Modal. Durante esse estudo, questões como "*O que é a Lógica Modal?"*, "*como surgem as modalidades?"* e *o "que estas podem representar em um problema lógico?"* foram discutidas.

Foi visto que uma Lógica Modal é uma lógica não clássica, obtida basicamente à partir do acréscimo de operadores modais à uma lógica clássica. Assim, uma *Lógica Modal Proposicional* surge do acréscimo de operadores modais à uma *Lógica Proposicional Clássica,* da mesma forma que uma *Lógica Modal de Primeira Ordem* surge do acréscimo de modalidades à uma *Lógica de Primeira Ordem*. Os operadores modais, quando utilizados em uma linguagem lógica, enriquecem a expressividade

desta, ao representar modificadores quanto ao contexto de avaliação de suas sentenças. Diferentes modalidades representam diferentes contextos.

Nos primeiros sistemas de lógica modal pensados nos tempos de *Aristóteles*, os *sistemas de Lógica Modal Alética,* estes operadores são utilizados para se expressar conceitos de *necessidade* e *possibilidade.* Entretanto, diferentes semânticas podem ser dadas a tais operadores, como, por exemplo, em *Lógicas Deônticas, Lógicas Temporais, Lógicas de Ação,* entre outras. À partir dos estudos de Kripke, que formaliza alguns axiomas para a sistematização de Lógicas Modais, é possível construir diversos sistemas de Lógica Modal, inclusive lógicas multimodais, como lógicas aplicáveis a problemas de *diferentes graus de crenças,* e *crenças de diferentes agentes*, por exemplo, de muita utilidade na computação e na área de Inteligência Artificial.

A *Semântica dos Mundos Possíveis de Kripke* fornece o Sistema Normal K, considerado o sistema base de uma Lógica Modal. Kripke propõe os conceitos de *estruturas* e *modelos* para a representação de mundos onde proposições podem ser diferentemente avaliadas, e define os operadores modais como forma de se representar uma "*navegação*" entre os mundos. O Sistema K é obtido através de um Lógica Clássica estendida com os axiomas $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$ e $p \rightarrow \Box p$.

Foi visto também que ao se estender uma lógica de primeira ordem com operadores modais, surgem novos questionamentos, como por exemplo, "*um indivíduo existente em um determinado mundo, ainda existe em um próximo mundo?*", ou ainda, "*é possível surgir um novo indivíduo, não existente no mundo atual, no próximo mundo?*". Essas questões são respondidas nas *Fórmulas de Barcan*, que implicam um universo decrescente, constante ou crescente, dependendo de qual variação da mesma for assumida no problema.

A implementação computacional de um programa de Lógica Modal pode ser realizada de duas maneiras distintas: A *Abordagem Direta*, na qual os operadores modais são diretamente representados e tratados nos programas e a *Abordagem Indireta*, na qual é realizada um tradução das fórmulas expressas no programa inicial, gerando um novo conjunto de fórmulas clássicas. Neste trabalho, foram apresentados os trabalhos de *Ohlbach* [OHL-88], que propõe um conjunto de regras de tradução de um

programa modal, onde os operadores modais são representados como argumentos adicionais (*World-Paths*) nos termos do programa e o trabalho de *Linh Anh Nguyen* [NGU-06], que trata diretamente os operadores modais na sua proposta *MProlog*.

Neste trabalho foi escolhido trabalhar com os conceitos propostos por *Linh Anh Nguyen,* como por exemplo, representar operadores modais como listas de termos Prolog e de se utilizar formas rotuladas de operadores para se representar uma aresta de um grafo para uma estrutura de mundos possíveis. As técnicas da abordagem direta pareciam mais atrativas para se aplicar à linguagem Prolog.

A Implementação de uma ferramenta para Programação em Lógica Modal, neste trabalho proposta, foi realizada na linguagem Java. Todas as principais estruturas de representação de dados como *Termos*, *Cláusulas* e *Contextos de Execução*, e todos os principais elementos lógicos como o *Motor de Resolução, Parser, Tratador de Diretivas* e *Tratador de Built-in Predicates* foram programados em Java sem a utilização de qualquer biblioteca adicional.

Nesta implementação, diretivas específicas para se carregar corretamente programas de lógica modal foram definidas. Essas diretivas alteram *flags* de controle do Parser, para que este, ao carregar uma cláusula, possa gerar corretamente variações sintáticas compatíveis com a classe do problema, representando assim, parte dos axiomas K, D, 4, 5 e I (para lógicas multimodais). Para completar, o elemento *Engine* da ferramenta, ao terminar o carregamento de programas modais, adiciona cláusulas adicionais, também de acordo com a classe de lógica modal em questão, para que variações possíveis das submetas possam ser obtidas durante a execução de consultas.

O processo de extensão das cláusulas originais de um programa resulta em uma lista de cláusulas extra que são adicionadas ao mesmo. Essa lista pode ser maior ou menor, dependendo do tipo de lógica modal tratada, mas que de qualquer forma impacta no desempenho da execução de uma consulta, uma vez que o mecanismo de resolução pode, em alguns casos, percorrer todas as cláusulas declaradas no programa para que se prove cada submeta da consulta. Com um número maior de cláusulas, maior o tempo necessário para se percorrer a lista inteira. Outro ponto que agrava o desempenho é que

os operadores modais devem ser testados um à um durante a unificação de termos, já que estes são representados como listas Prolog.

Pode-se concluir assim, que um problema de lógica modal pode ser representado e processado como um programa Prolog estendido, desde que as cláusulas derivadas e regras adicionais sejam corretamente geradas pelo interpretador no momento do carregamento de um programa. O processamento de uma consulta de um programa modal pouco se altera quando comparado ao processamento de uma consulta de um programa clássico. A única alteração necessária é que as modalidades devem ser comparadas durante a unificação de termos. Se por um lado, o desempenho de um programa modal se torna inferior ao desempenho de um programa clássico, devido ao maior número de cláusulas e maior número de termos a serem comparados durante a unificação, por outro, se ganha na expressividade da linguagem.

9.2 TRABALHOS FUTUROS

A partir do que foi estudado e desenvolvido durante a realização deste trabalho, surgem como propostas para trabalhos futuros alguns temas, enumerados abaixo:

1. Modelagem de um Sistema Multi-Agente;
2. Estender a ferramenta desenvolvida para Lógica Fuzzy;
3. Integração com plataformas multi-agentes atuais: Jason / Jade;
4. Extensão de funcionalidades através de Built-in Predicates;

Um Sistema Multi-Agente pode ser modelado como uma aplicação para execução na ferramenta desenvolvida neste trabalho utilizando-se lógicas modais do tipo KD45 que pode representar *crenças de diferentes agentes*. Nesse caso, o operador modal universal $\Box_i$ indexado pelo identificador do agente, i, poderia significar "*o agente i tem certeza que (...)*", enquanto que o operador modal existencial $\Diamond_i$ poderia representar "*o agente i considera possível que (...)*".

Uma extensão da ferramenta para desenvolvimento de problemas de lógica Fuzzy também pode ser construída à partir do que a ferramenta oferece atualmente. É

possível se modelar problemas de lógica Fuzzy como problemas de lógica modal, como explicado em [ZHA-05].

Uma aplicação multiagente também pode ser desenvolvida utilizando a ferramenta para programação em lógica modal desenvolvida neste trabalho juntamente com uma plataforma de sistemas multiagente atual, como por exemplo, Jason e JADE. Para tanto, pode ser desenvolvida uma extensão na ferramenta para a execução de algumas lógicas específicas de agentes.

A ferramenta desenvolvida neste trabalho pode ser fácilmente estendida através da programação de Built-in Predicates. Por exemplo, é possível criar predicados especiais para a obtenção de dados na web utilizando as classes de comunicação de dados fornecidas no próprio Java. É possível também criar uma camada de serviços HTTP ou sockets, de forma que outros sistemas programados em outras linguagens possam também executar consultas em uma instância do interpretador, entre outras funcionalidades específicas para cada classe de problema.

REFERÊNCIAS

[MAL-02] Aline Vieira Malanovicz - 2002 - Lógicas Modais: Fundamentos e Aplicações

[COS-08] Bruno Costa Coscarelli - 2008 - Introdução à Lógica Modal

[WAR-77] David H D Warren, Luis M. Pereira, Fernando Pereira - 1977 - PROLOG: The Language and it´s implementation compared with LISP

[BIZ-98] Fernanda Oviedo Bizarro - 1998 - Um Estudo sobre Lógica Modal

[OHL-88] Hans Jürgen Ohlbach - 1988 - A Resolution Calculus for Modal Logics

[LLO-07] J.W. Lloyd, K.S. Ng J. Veness - 2007 - Modal Functional Logic Programming

[NGU-05] Linh Anh Nguyen - 2005 - MProlog: An Extension of Prolog for Modal Logic Programming

[NGU-06] Linh Anh Nguyen - 2006 - The Modal Logic Programming System MProlog: Theory, Design and Implementation

[NGU-10] Linh Anh Nguyen - 2010 - Foundations of Modal Logic Programming: The Direct Approach

[GIR-95] Lucia Maria Martins Giraffa, Marcelo Ladeira, Ricardo Silveira - 1995 - Lógica Modal

[CER-86] L. Fariñas del Cerro - 1986 - MOLOG: a system that extends Prolog with modal logic

[GER-01] Manolis Gergatsoulis - 2001 - Temporal and Modal Logic Programming Languages

[CAS-87] Marco A. Casanova, Fernando A. C. Giorno, Antonio L. Furtado - 1987 - Programação em Lógica e a Linguagem Prolog

[BEN-10] Mario R. F. Benevides - 2010 - Lógica Modal

[SCO-99] Michael L. Scott - 1999 - Programming Languages Pragmatics - Third Edition

[SEB-03] Robert W. Sebesta - 2003 - Conceitos de Linguagens de Programação

[BOR-07] R. H. Bordini, J. F. Hübner, M. Wooldridge - 2007 – Programming Multi-Agent Systems in AgentSpeak using Jason

[RUS-95] S. Russell, P. Norvig - 1995 - Artificial Intelligence: a modern approach - Third Edition
[ZHA-05] Zaiyue Zhang, Yuefei Sui, Cungen Cao - 2005 - Description of Fuzzy First-Order Modal Logic Based on Constant Domain Semantics

APÊNDICE A: GRAMÁTICA DESENVOLVIDA

<program> ::= <clause-block-list> | <query>

<clause-block-list> ::= <clause-block> [<clause-block-list>]

<clause-block> ::= <list> '{' <clause-list> '}' | <clause-list>

<clause-list> ::= <clause> [<clause-list>]

<clause> ::= (<term> [':-' <term-list-or>] | ':-' <term-list>) '.'

<term-list-or> ::= <term-list> [';' <term-list-or>]

<term-list> ::= <in-op-exp> [',' <term-list>]

<in-op-exp> = <term> [_in_op_ <in-op-exp>]

<term> ::= '(' <in-op-exp> ')' | [_pre_op_] <term> [_post_op_] | <struct> | _string_ | _variable_ | _number_ | '!' | <list-term>

<struct> ::= _atom_ ['(' <term-list> ')']

<list-term> ::= <list> [':' (<struct> | _variable_)]

<list> ::= '[' <term-list> ['|' <list>] ']'

<query> ::= '?-' <term-list> '.'

APÊNDICE B: BUILT-IN PREDICATES

consult(source) – ao executar, o predicado *consult* lê um arquivo de caminho especificado na string source que é passada como argumento e então invoca o parser para fazer o carregamento do programa. Se tudo ocorrer bem, o predicado resulta true.

write(term) – ao executar, o predicado *write* imprime no console o termo passado como argumento e resulta true. No Java, o mesmo seria obtido com "System.out.print(term.toString());".

read(variable) – ao executar, o predicado *read* lê do usuário um termo na forma de texto e imediatamente invoca o parser para construir o objeto correspondente. Em seguida, o objeto construído é atribuído à variável livre indicada no parâmetro variable. Se tudo ocorrer bem, o predicado resulta true.

nl – ao executar, o predicado *nl* imprime na tela uma quebra de linha e resulta true. O mesmo seria obtido no Java ao se executar "System.out.print('/n');".

nomodality(term) – ao executar, o predicado *nomodality* assume o mesmo valor que o termo passado no parâmetro term sem modalidades.

invert_list(list) – ao executar, o predicado *invert_list* assume o mesmo valor que a lista passada no parâmetro list na ordem inversa.

sleep(milis) – ao executar, o predicado *sleep* provoca uma pausa na execução do programa e em seguida resulta true. O tempo da pausa é especificado no parâmetro milis, em milesegundos. No Java o mesmo é obtido com "Thread.sleep(milis);".

exit – ao executar, o predicado *exit* provoca o encerramento imediato do programa e fecha o interpretador.

APÊNDICE C: DIRETIVAS E OPERADORES

Diretivas programadas neste trabalho:

op(prioridade, tipo, símbolo) – declara um operador um operador no programa;
setFlag(chave, valor) – define um valor para um flag do interpretador;
modalOP(tipo, símbolo) – declara um operador modal como universal ou existencial;
modalLOGICS(sistema) – declara o sistema de lógica modal em questão;

Operadores declarados por padrão neste trabalho:

‘!‘: cut – operador que habilita o backtracking, implementado como um termo comum;
‘+‘: soma – operador aritmético infixo de prioridade 500;
‘-‘: subtração – operador aritmético infixo de prioridade 500;
‘*‘: multiplicação – operador aritmético infixo de prioridade 400;
‘/‘: divisão – operador aritmético infixo de prioridade 400;
‘is‘: atribuição – operador infixo de prioridade 700;
‘=‘: igualdade – operador relacional infixo de prioridade 700;
‘<‘: menor – operador relacional infixo de prioridade 700;
‘<=‘: menor/igual – operador relacional infixo de prioridade 700;
‘>‘: maior – operador relacional infixo de prioridade 700;
‘>=‘: maior/igual – operador relacional infixo de prioridade 700;
‘++‘: incremento – operador aritmético pós-fixo de prioridade 100;

APÊNDICE D: PROGRAMA EX_FAMILY.PL

```
pai(antonio, bruno).
pai(bruno, carlos).
pai(carlos, daniel).
avo(X, Y) :- pai(X, Z), pai(Z, Y).
bisavo(X, Y) :- pai(X, Z), avo(Z, X).
```

APÊNDICE E: PROGRAMA EX_LIST.PL

```
nova_lista([]).
insere([], X, [X]).
insere(L, X, [X|L]).
remove([H|T], H, T).
```

APÊNDICE F: PROGRAMA EX_FACT.PL

```
fact(0, 1).
fact(N, F) :- fact(N – 1, F1), F is N * F1.
```

APÊNDICE G: PROGRAMA EX_INTRO_MODAL.PL

```
:- modalLOGICS('KD').
:- modalOP('UNIVERSAL', 'nes').
:- modalOP('EXISTENTIAL', 'pos').

[nes] { p(a). }
[nes] { q(a). }
[nes] { s(X) :- q(X), [pos]:t(X). }
[nes] { [pos]: t(a). }
[nes] { [pos]: t(b). }
```

APÊNDICE H: PROGRAMA EX_MULTI_MODAL.PL

```
:- modalLOGICS('mKD').
:- modalOP('UNIVERSAL', 'nes').
:- modalOP('EXISTENTIAL', 'pos').

[pos(1)]: p(a).
[nes(1)] { [nes(2)]:q(X) :- p(X). }
[nes(1)] { [pos(2)]:r(X) :- [nes(2)]:q(X). }
[nes(1), nes(2)] { s(X) :- q(X), r(X). }
[nes(1)] { t(X) :- [pos(2)]:s(X). }
```

APÊNDICE I: PROGRAMA EX_MATHS_MODAL.PL

```
:- modalLOGICS('mKDI4').
:- modalOP('UNIVERSAL', 'n').
:- modalOP('EXISTENTIAL', 'p').

[n(4)]: good_in_maths(X) :- maths_teacher(X).
[n(5)] { [n(I)]:good_in_maths(X) :- [n(I)]:mathematician(X). }
[n(3)] { [p(I)]:good_in_maths(X) :- maths_student(X). }
[n(3)] { [p(I)]:good_in_physics(X) :- physics_student(X). }
[n(2)] { [p(2)]:good_in_maths(X) :- good_in_physics(X). }
maths_teacher('John').
[n(2)]:mathematician('Tom').
[n(5)]:maths_student('Peter').
[n(5)]:physics_student('Mike').
```

APÊNDICE J: PROGRAMA EX_LIKES_MODAL.PL

```
:- modalLOGICS('mKD45').
:- modalOP('UNIVERSAL', 'nes').
:- modalOP('EXISTENTIAL', 'pos').

[nes(1)]
{
        likes(jan, cola).
        likes(piotr, pepsi).
        [pos(1)]:likes(X, cola) :- likes(X, pepsi).
        [pos(1)]:likes(X, pepsi) :- likes(X, cola).
}
[nes(2)]
{
        likes(jan, pepsi).
        likes(piotr, cola).
        likes(piotr, beer).
        likes(X, cola) :- likes(X, pepsi).
        likes(X, pepsi) :- likes(X, cola).
}
[nes(3)]
{
        likes(jan, cola).
        very_much_likes(X, Y) :- likes(X,Y), [nes(1)]:likes(X, Y), [nes(2)]:likes(X, Y).
}

[pos(3)]:likes(piotr, pepsi).
[pos(3)]:likes(piotr, beer).

very_much_likes(X, Y) :- [nes(3)]:very_much_likes(X, Y).
likes(X, Y) :- [pos(3)]:very_much_likes(X, Y).
possibly_likes(X, Y) :- [pos(I)]:likes(X, Y).
```