Objetos Inteligentes Baseados em CLP para Aplicações Científicas

Christian Cleber Masdeval Braz

Dissertação apresentada ao Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul como parte dos requisitos para obtenção do título de **Mestre em Ciência da Computação.**

ORIENTADOR: Prof. Dr. Paulo Aristarco Pagliosa

Campo Grande - MS
2005

A minha querida mãe.

Agradecimentos

Primeiramente agradeço a minha mãe, Aliete, quem sempre me incentivou e apoiou nas decisões importantes e que nunca mediu esforços em investir na minha educação, possibilitando assim que eu concluísse o mestrado numa instituição pública de ensino superior, passando a constituir um grupo reduzido de pessoas em nosso país.

Ao meu orientador, Paulo Aristarco Pagliosa, que não só conduziu o trabalho com precisão e dedicação, muitas vezes extrapolando suas obrigações como professor, mas também tornou as horas difíceis e exaustivas em momentos mais amenos e agradáveis, deixo aqui o meu mais sincero apreço e admiração.

Aos professores e funcionários do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul, agradeço pela atenção com que sempre me receberam e presteza com que me auxiliaram ao longo dessa jornada.

Resumo

Braz, C.C.M. *Objetos Inteligentes Baseados em CLP para Aplicações Científicas.* Campo Grande, 2004. Dissertação (Mestrado) – Universidade Federal de Mato Grosso do Sul.

A programação lógica com restrições (*constraint logic programming*, ou CLP) é uma classe de linguagens de programação baseada na programação lógica e programação com restrições. Descende de estudos em áreas diversas como inteligência artificial, linguagens de programação e otimização combinatorial e cada vez mais vem sendo utilizada para modelar e resolver muitos problemas reais complexos. O objetivo deste trabalho é propor, validar e implementar um conceito de objeto inteligente baseado em CLP. Um objeto inteligente engloba dados, métodos e também conhecimento declarativo e mecanismos de inferência, os quais permitem a este utilizar seu conhecimento a fim de produzir comportamentos inteligentes. Este conhecimento é formulado através de regras CLP que estão associadas a um objeto na forma de pseudo-métodos declarados na classe do objeto. Esta integração respeita as características principais da orientação a objetos — encapsulamento, herança e polimorfismo —, resultando em aplicações nas quais a computação se dá em um universo constituído por objetos que trocam mensagens entre si. Dessa forma, o controle das ações não é exercido por um mecanismo de inferência central, mas sim pelo fluxo comum de mensagens em aplicações orientadas a objetos. Com isso, a modelagem e a implementação de sistemas complexos que necessitem fazer uso das técnicas introduzidas pela CLP podem continuar sendo baseadas no paradigma da orientação a objetos. Os benefícios sâo reusabilidade, manutenabilidade, legibilidade e eficiênica que podem ser alcançados em programas que utilizam este paradigma, além de tornar, para os engenheiros de sistemas, mais intuitiva e natural a absorção desta técnica de inteligência artificial.

Palavras-chave: *inteligência artificial*, *orientação a objetos*, *programação lógica com restrição.*

Abstract

Braz, C.C.M. *Objetos Inteligentes Baseados em CLP para Aplicações Científicas.* Campo Grande, 2004. Dissertation (M.Sc.) – Universidade Federal de Mato Grosso do Sul.

Constraint logic programming (CLP) is a programming language class based on logic programming together with constraint programming. It is derived from general research areas such as artificial intelligence, programming languages, and combinatorial optimization, and has been employed to model and solve many complex real problems. The objective of this work is to propose, validate, and implement a concept of *intelligent object* based on CLP. An intelligent object encapsulates, besides data and methods as usual in object orientation, declarative knowledge and inference engines, which enable the object to employ its knowledge in order to yield intelligent behavior. In the proposed approach the knowledge is formulated from CLP rules which are associated to an object. Because this integration satisfies the main characteristics of the object oriented programming (OOP) — encapsulation, inheritance, and polymorphism —, the processing in resulting applications is based on an universe made up of objects the change messages each other. The actions are not controlled by a central inference engine, but arise from the message flow among objects. As consequence, modeling and implementation of complex systems that need to make use of CLP techniques could remain based on object oriented paradigm, improving code reusability, maintainability, readability, and efficiency, and providing to software engineering's a more intuitive and natural absorption of this intelligence artificial technique.

Keywords: *artificial intelligence*, *object orientation*, *constraint logic programming.*

Conteúdo

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

1.1 Motivação e Justificativa

Aplicações computacionais numéricas e gráficas para ciências e engenharia são complexas em geral, fazendo uso de consideráveis recursos computacionais das máquinas onde são executadas. Por conta da complexidade, torna-se interessante o emprego de métodos de engenharia de software baseados no paradigma da orientação a objetos para modelar e implementar tais aplicações. Estes métodos provêm ferramentas que, se corretamente aplicadas às fases de análise, projeto e implementação, podem conduzir ao desenvolvimento de aplicações mais eficientes e mais facilmente legíveis, extensíveis e manuteníveis. É determinate, pois, que as linguagens de programação utilizadas na implementação de aplicações dessa natureza, além de proverem mecanismos eficazes que possibilitem a utilização dos recursos computacionais necessários para que estas sejam executadas em máquinas reais, ofereçam também suporte ao paradigma da programação orientada a objetos.

As técnicas introduzidas pela inteligência artificial (IA) têm mostrado, ao longo do tempo, serem bastante úteis para a solução de variados tipos de problemas. Um exemplo de sucesso de sua utilização foi o advento dos sistemas especialistas (ou sistemas baseados em conhecimento), os quais têm como objetivo reproduzir o comportamento de especialistas humanos na resolução de problemas do mundo real. Grande parte dos sistemas especialistas desenvolvidos foram sistemas de produção, isto é, que utilizavam regras de produção como forma de representação do conhecimento [12], as quais consistem de pares de expressões condição/ação. Esta forma de representação do conhecimento baseia-se em linguagens como a lógica proposicional, que, apesar de simples, foi utilizada nos sistemas especialistas para resolver problemas em muitos domínios diferentes.

Sistemas especialistas foram inicialmente implementados utilizando-se linguagens de programação ideais para o processamento simbólico (como LISP, por exemplo), porém pouco aptas a suprirem as necessidades numéricas e gráficas de um software científico. A partir dos anos 90, muitos estudos foram direcionados visando integrar regras de produção com a orientação a objetos [12]. Isso iria facilitar seu uso por parte dos engenheiros de sistemas, já familiarizados com a abordagem orientada a objetos, mas que não necessariamente conheciam as características de sistemas baseados em conhecimento.

A conferência OOSPLA (*Object-Oriented Programming Systems and Languages*) de 94 contou com um workshop dedicado ao estudo dos EOOPS (*Embedded Object-Oriented Production Systems*), sendo um dos principais trabalhos o de Pachet [46], o qual abordou em

detalhes a questão da integração de regras e objetos. Algumas propostas de integração resultantes foram as dos sistemas CLIPS, Néopus [45] (baseado no Smalltalk), ENVY [40], $RAL/C++$ [21], ILOG rules [5], dentre outros.

Uma das propostas mais interessantes neste sentido foi o conceito de *objeto inteligente* proposto por Bomme [13]. Um objeto inteligente integra: dados e métodos de um objeto usual, conhecimento declarativo (formulado através de regras) e capacidade de raciocínio para produzir comportamentos inteligentes. O processamento do conhecimento é mantido sobre o controle do próprio objeto, o que ajuda garantir o encapsulamento dos dados. Essa implementação provê uma arquitetura onde o conhecimento é distribuído entre os diversos objetos da aplicação e seu processamento é descentralizado. Neste contexto, é possível utilizar as técnicas de IA no próprio ambiente em que as aplicações científicas são comumente desenvolvidas. As características da orientação a objetos — encapsulamento, herança e polimorfismo — são completamente respeitadas e é possível obter a eficiência computacional necessária, já que a integração pode ocorrer com linguagens como C++. Tudo isso possibilita que tais aplicações continuem sendo modeladas e implementadas pelos mesmos engenheiros, pois a integração ocorre naturalmente na linguagem hospedeira da aplicação.

Outro exemplo da aplicação das técnicas de inteligência artificial na resolução de problemas complexos são os provadores automáticos de teoremas, fruto de uma das linhas de pesquisa mais antigas na IA, a lógica. Dentre os primeiros provadores de teoremas está o Logic Theorist [54], que foi utilizado com grande sucesso para resolver problemas matemáticos, causando grande euforia na comunidade de IA da época. O estudo da lógica e sua adequação como método computacional para solução de problemas levou ao surgimento do paradigma da programação lógica, sendo um dos mais conhecidos representantes desta classe de linguagem de programação o Prolog.

O Prolog é um provador automático de teoremas baseado na lógica de predicados de primeira-ordem [12], que é uma linguagem de representação de conhecimento mais expressiva e poderosa que as regras de produção geralmente empregadas nos sistemas especialistas. É amplamente reconhecido na literatura [50, 8, 53, 35] que, devido ao fato das linguagens lógicas possuírem uma semântica declarativa e informal, estas, se adequadamente utilizadas, permitem o desenvolvimento de programas menores e mais fáceis de serem manutenidos e estendidos. Isto geralmente significa um menor tempo para escrever e modificar os programas e facilita ao programador experimentar várias formas de solução do problema até que a melhor possa ser encontrada.

Contudo, esta semântica tem suas limitações. Primeiro, os objetos manipulados por um programa em lógica são estruturas sem significado implícito, necessitando que cada objeto semântico seja explicitamente codificado, o que leva à necessidade de um raciocínio sobre o programa num nível muito primitivo [34]. Um segundo problema é que a execução destes programas está baseada num método de busca em profundidade ineficiente, que resulta em problemas de performance em aplicações que requerem busca em grandes bases de dados [24]. A relativa ineficiência de linguagens lógicas como o Prolog, quando comparadas com linguagens como C, resultaram na limitação de sua aplicabilidade e conseqüente declínio de sua aceitação e utilização prática [48].

A programação lógica com restrições ou *constraint logic programming* (CLP) possui características que resolvem em grande parte os problemas da programação lógica convencional. Inicialmente, foi proposta por Jaffar e Lassez como uma classe de linguagens de programação baseada no paradigma da programação lógica e resolução de restrições [34]. O objetivo era introduzir estruturas semânticas mais ricas às linguagens lógicas, possibilitando assim um poder de expressividade superior. Este esquema, chamado CLP(X), torna os programas capazes de tratar naturalmente, por exemplo, expressões algébricas, boolenas, dentre outras. Objetos que têm significado num determinado domínio, como os predicados $=$, $<$, $>$ no domínio dos

inteiros ou reais, passam a estar disponíveis, suplementando assim a natureza puramente abstrata da lógica.

Após sua criação, a CLP foi fortemente influenciada pelos trabalhos relacionados às técnicas de consistência. Estes pregavam o uso de restrições para podar a árvore de busca, limitando assim o espaço de possibilidades e superando os problemas relacionados à abordagem tradicional [24]. A CLP, diferentemente dos provadores de teoremas, destina-se principalmente a aplicações numéricas e aquelas que lidam com problemas combinatórios difíceis, tendo sido usada em diferentes áreas como engenharia, planejamento, banco de dados, interfaces, etc.

1.2 Objetivos e Contribuições

O objetivo geral deste trabalho é definir, validar e implementar um conceito de objeto inteligente nos moldes daquele proposto por Bomme. Sua abordagem possibilita uma integração natural de técnicas de IA no ambiente de desenvolvimento de aplicações científicas, principalmente por valer-se da eficiência computacional destes ambientes e por respeitar completamente as características da orientação a objetos. Porém, ao invés de representarmos o conhecimento dos objetos através de regras de produção, como fez Bomme, iremos fazê-lo através de regras CLP. Estas regras estarão associadas a objetos que poderão, quando necessário, se utilizar do conhecimento descrito por elas para auxiliar na execução de determinadas atividades.

Os objetivos específicos do trabalho são:

- Revisão dos fundamentos da lógica proposicional e de primeira ordem, necessários à compreensão dos princípios da programação lógica com restrições.
- Estudo dos conceitos e técnicas da CLP e suas aplicações.
- Definição de objeto inteligente baseado em CLP.
- Descrição da implementação e exemplos de utilização dos objetos inteligentes propostos.

Esta talvez seja a primeira proposta de integração de objetos com regras que não sejam regras de produção. Esperamos com esta inciativa contribuirmos para o desenvolvimento de um ambiente favorável à utilização da IA e para a disseminação da tecnologia CLP. Algumas das vantagens esperadas são:

- Possibilidade de utilizar a programação com restrições de forma integrada à definição da estrutura e funcionalidade de um objeto.
- Extensibilidade e reusabilidade herdada da abordagem orientada a objetos aplicada ao projeto dos objetos inteligentes.
- Maior facilidade de utilização da CLP por aqueles que não estão familiarizados com esta tecnologia.

1.3 Resumo do Trabalho

Um objeto inteligente é uma entidade que agrega dados e métodos, tal qual um objeto usual, e conhecimento declarativo formulado através de regras CLP, as quais formam a base de conhecimento do objeto. Estas regras estão acopladas aos objetos como pseudo-métodos da

linguagem. O conhecimento declarativo pode ser utilizado seletivamente pelo objeto para ajudá-lo a desempenhar algumas de suas operações, caso essas envolvam raciocínio. Este é realizado por um mecanismo solucionador de restrições acoplado à aplicação que se utiliza de objetos inteligentes. Após realizar as inferências necessárias, este mecanismo retorna uma solução (casa haja alguma), produzindo assim o comportamento inteligente do objeto.

Originalmente, gostaríamos de ter implementado objetos inteligentes em C++; porém, utilizamos Java como linguagem de programação. Além de possibilitar o desenvolvimento mais rápido de um protótipo com o qual pudéssemos validar nossa proposta, os ambientes de execução padrão de Java disponibilizam recursos de reflexão computacional e serialização necessários mas cuja implementação, complicada, deveria por nós ser feita em C++. Esta, porém, não é a solução mais adequada, pois Java ainda não é eficiente como C++ para programação de aplicações científicas. A necessidade de um software (máquina virtual) para traduzir os bytecodes de Java para código nativo em tempo de execução e a dependência de um ambiente nem sempre eficaz para alocação e desalocação de memória, em geral tornam os programas mais lentos e com menos chances de otimização.

Os componentes utilizados neste trabalho a fim de prover a infra-estrutura necessária à criação de objetos inteligentes estão resumidos a seguir.

1.3.1 Criação da Base de Conhecimento

A criação de uma base de conhecimento envolve as etapas sumarizadas na Figura 1.1.

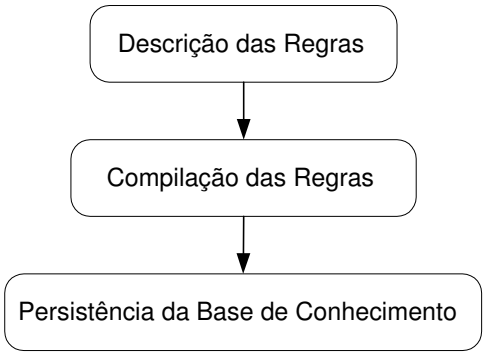

Figura 1.1: Passos para geração de uma base de conhecimento.

- Descrição das Regras: o conhecimento de um objeto é descrito através de regras CLP especificadas de acordo com uma linguagem por nós definida. Esta descrição é embutida em comentários especiais da linguagem Java na declaração da classe de um objeto inteligente. Uma regra é um pseudo-método que possui um nome único e um modificador de acesso que especifica sua visibilidade. Regras podem acessar atributos e métodos do objeto e enviar mensagens a outros objetos. No modelo proposto, todos os objetos de uma classe inteligente terão o mesmo conhecimento e capacidade de raciocínio.

- Compilação das Regras: uma vez definidas todas as regras nas classes dos objetos inteligentes de uma aplicação, os arquivos fontes devem ser compilados. Isso é feito por um compilador Java estendido que, além de traduzir o programa Java original, compila também as regras CLP. Cada uma é verificada quanto à sintaxe e semântica, inclusive o código Java para acesso a atributos e métodos.

- Persistência da Base de Conhecimento: a medida que as regras CLP de uma classe são extraídas pelo compilador, estas são adicionadas em uma estrutura em memória que

representa a base de conhecimento daquela classe de objetos. Ao término da extração das regras esta base é armazenada em um espaço persistente, ou seja, não volátil, de objetos. Este espaço é um arquivo mantido no sistema de arquivos local cujo nome é o mesmo da classe inteligente.

Quando um objeto inteligente é instanciado pela primeira vez na aplicação, sua base de conhecimento é restaurada do meio persistente da seguiten maneira:

1. Todas as regras codificadas na classe do objeto são lidas do arquivo de base de conhecimento correspondente (gerado quando da compilação da classe) e carregadas em memória.

2. As regras declaradas em todas as classes bases inteligentes (caso haja alguma) da classe do objeto são acrescentadas na base de conhecimento, ou seja, um objeto inteligente *herda* as regras das classes bases. Serão acrescentadas apenas as regras públicas que não tenham o mesmo nome de uma regra já existente na base.

1.3.2 Processamento do Conhecimento

O processamento de um programa CLP é feito por mecanismos de inferência conhecidos como *solucionadores de restrições*, os quais possuem uma linguagem CLP própria de acordo com a qual as regras devem ser especificadas. Um solucionador de restrições espera receber um conjunto de regras (o programa CLP), escritas na sua linguagem, e perguntas a serem provadas. Para cada pergunta submetida um processo de inferência é disparado e, caso haja respostas, estas são devolvidas uma a uma ao usuário.

Não implementamos neste trabalho um mecanismo solucionador de restrições, mas utilizamos um já existente, uma plataforma para programação lógica com restrição chamada ECLiPSe [32, 66]. Além de bastante completa no que tange aos domínios que possibilita tratar, possui interfaces para integração com outras linguagens de programação, como Java e C++. O esquema de integração desenvolvido para prover a comunicação entre os objetos inteligentes e o ECLiPSe está resumido na Figura 1.2.

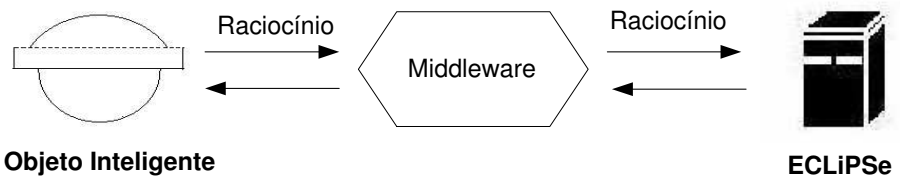

Figura 1.2: Integração entre um objeto inteligente e seu mecanismo de inferência.

Toda comunicação entre um objeto inteligente e o mecanismo capaz de processar seu conhecimento é feita através de um software intermediário cujas funções são:

- Converter a base de conhecimento do objeto — cujas regras foram escritas de acordo com a linguagem por nós definida — em regras especificadas de acordo com a linguagem CLP do ECLiPSe.

- Submeter as regras convertidas para o ECLiPSe.

- Tratar as solicitações do objeto para processamento do seu conhecimento bem como as respostas resultantes, de forma a tornar viável esta comunicação.

1.3.3 Utilização do Conhecimento

Uma vez definida a base de conhecimento de um objeto inteligente e o *middleware* necessário para fazer a ligação com o mecanismo de inferência, perguntas cuja(s) resposta(s) envolve(m) raciocício podem ser feitas ao objeto. Isso é feito enviando-se uma mensagem chamada `ask` ao objeto inteligente, cujo esquema de processamento é ilustrado na Figura 1.3.

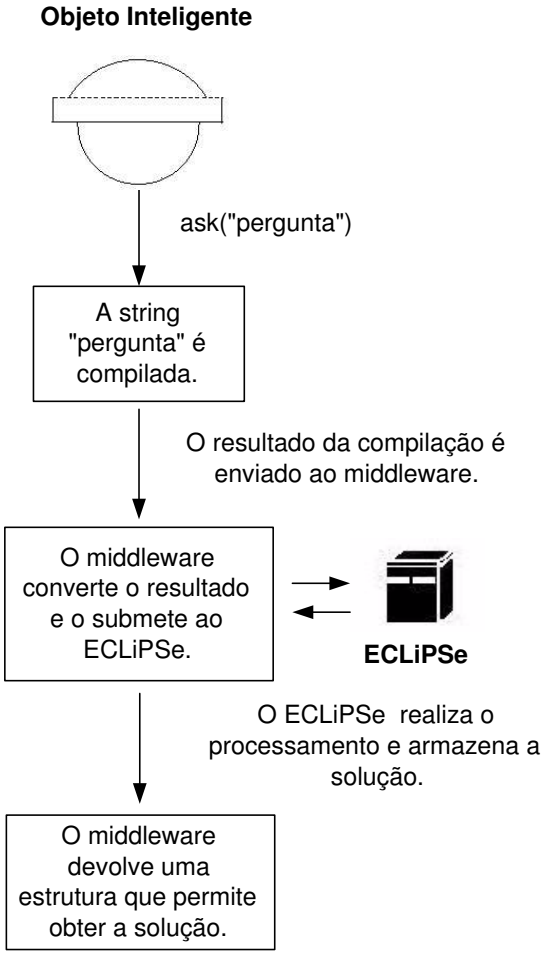

Figura 1.3: Funcionamento do método `ask`.

`ask` é parametrizada com uma string contendo o texto de uma pergunta, escrita de acordo com a linguagem de regras, a ser feita ao objeto. O compilador de regras CLP processa a pergunta, gerando uma representação em memória tal como feito com as regras da base de conhecimento do objeto. Esta representação é então convertida pelo *middleware* e submetida ao ECLiPSe para processamento. O resultado é encapsulado pelo *middleware* em um objeto chamado *conjunto de resultados*, cuja funcionalidade permite:

- Verificar se foi gerada alguma solução.
- Obter uma a uma as soluções existentes.
- Obter o valor de cada uma das variáveis lógicas envolvidas na pergunta, para cada uma das soluções existentes.

1.4 Organização do Texto

Capítulo 2
Programação Lógica
No Capítulo 2 apresentamos uma revisão sobre lógica e seus fundamentos. Discutimos os principais aspectos que compõe um sistema lógico, as características de duas das principais linguagens lógicas, o funcionamento dos mecanismos de inferência e os passos envolvidos num processo de dedução. Analisamos também as principais vantagens e desvantagens associadas a abordagem tradicional da lógica.

Capítulo 3
Programação Lógica com Restrição
No Capítulo 3 discutimos sobre programação lógica com restrição e as características que a tornam mais adequada do que a programação lógica convencional para ser utilizada na solução de determinados tipos de problemas do mundo real. Começamos fazendo uma introdução do seu histórico e das principais idéias que influenciaram sua criação. Depois apresentamos formalmente sua sintaxe e semântica, dando ênfase principalmente à semântica operacional do esquema. Em seguida, exemplificamos o esquema de execução em função de dois dos principais domínios de aplicação: o domínio das expressões aritméticas lineares e não lineares. Abordamos então técnicas de consistência, os problemas que se encaixam nesta categoria e os principais algoritmos utilizados para resolvê-los. Por fim, apresentamos um resumo dos principais sistemas CLP existentes e das aplicações a que eles se destinam.

Capítulo 4
Objetos Inteligentes e CLP
O Capítulo 4 apresenta a proposta de integração da CLP com objetos. Primeiramente, conceituamos o que são objetos para então explicarmos o que vem a ser objetos inteligentes. Em seguida, descrevemos o projeto do sistema e os principais aspectos da arquitetura adotada.

Capítulo 5
Implementação dos Objetos Inteligentes
No Capítulo 5 apresentamos os principais aspectos relacionadas à implementação do sistema de objetos inteligentes. Descrevemos a estrutura, funcionalidade e uso das principais classes envolvidas na solução. Discutimos também o funcionamento dos mecanismos necessários à criação do conhecimento de um objeto inteligente — codificação, extração e validação das regras pelo compilador de regras, persistência de uma base de conhecimento, e criação da base para utilização pelos objetos —, além do processo de utilização do conhecimento através da interação entre objeto inteligente, *middleware* e mecanismo de inferência. Por fim, mostramos alguns exemplos da utilização da solução.

Capítulo 6
Conclusão
No Capítulo 6 fazemos uma avaliação geral do trabalho. Apontamos inicialmente todos os objetivos específicos e o que foi feito para alcançá-los. Depois, discutimos sobre os resultados obtidos, as dificuldades encontradas durante sua execução e as limitações que julgamos existir. Propomos também possíveis melhorias e extensões para trabalhos futuros.

Apêndice A
Gramática da Linguagem CLP

No Apêncice A listamos a gramática da linguagem CLP por nós definida para especificação das regras de uma base de conhecimentos e de perguntas submetidas a um objeto inteligente.

CAPÍTULO 2

Programação Lógica

2.1 Introdução

A lógica possui uma história que remonta a mais de vinte e três séculos atrás, desde o tempo dos filósofos gregos. Dentre eles, o mais importante foi Aristóteles, o qual estabeleceu os fundamentos da lógica de forma sistemática. Suas aplicações vão desde a fundamentação teórica de diversas áreas da matemática, até a representação de conhecimento na Inteligência Artificial.

A lógica é o ponto de partida para muitos dos trabalhos em IA e neste capítulo apresentamos uma revisão de seus principais conceitos. Inicialmente, na Seção 2.2, discutimos sobre a representação de conhecimento através de linguagens lógicas e como ocorre a conexão entre o que representam e fatos do mundo real. A Seção 2.3 define o trabalho a ser realizado por um mecanismo de inferência e os aspectos mais relevantes de seu funcionamento. Nas Seções 2.4 e 2.6 descrevemos a sintaxe, semântica e mecanismos de inferência de duas das principais linguagens lógicas existentes, a lógica proposicional e a lógica de primeira ordem. A Seção 2.5 apresenta o conceito de busca em espaço de estados e dois de seus principais algoritmos.

2.2 Representação de Conhecimento

O principal objetivo da representação de conhecimento é expressar conhecimento de forma tratável sob o ponto de vista computacional [54]. A adequação da lógica como método de representação de conhecimento vem sendo discutida desde os primórdios da IA, sendo tida como o sistema formal mais simples que representa uma teoria semântica interessante para representação de conhecimento [12].

De maneira geral, um sistema lógico consiste em um conjunto de sentenças representando fatos sobre o mundo real (representação de conhecimento) e um conjunto de procedimentos mecânicos que operam nas sentenças e possibilitam produzir raciocínio (mecanismo de inferência).

As sentenças são formadas a partir de uma *linguagem formal* definida por dois aspectos:

- A sintaxe da linguagem descreve as possíveis configurações que podem constituir sentenças dado um conjunto pré-definido de símbolos. Por exemplo, a sintaxe da linguagem das expressões aritméticas permite construções do tipo $x > y$ e $x * 2 = 4y$.

- A semântica determina qual interpretação dar a uma sentença. É ela quem diz que $x > y$ é falso quando y é um número maior ou igual a x e verdadeiro caso contrário.

Uma boa linguagem para representação de conhecimento deve combinar as vantagens das linguagens naturais (como seu poder de representação) com as vantagens das linguagens formais (objetividade, precisão). Deve ser expressiva e concisa, de tal sorte que possibilite expressar de forma sucinta qualquer tipo de conhecimento. Deve ser livre de contexto e sem ambigüidades, de forma que o que dissermos hoje ainda poderá ser interpretado amanhã. Por fim, deve ser efetiva no sentido de possibilitar um mecanismo de inferência que seja capaz de realizar deduções a partir das sentenças da linguagem [54].

Na lógica, o significado de uma sentença é o que ela representa sobre o mundo, a definição de que o mundo é desta maneira e não daquela. Diferentemente das linguagens naturais, onde a interpretação sobre a maioria das coisas já foi fixada a muito tempo, o significado de uma sentença em lógica é essencialmente atribuído pela pessoa que a escreveu. Quem cria a sentença é quem provê uma interpretação para ela, estabelecendo assim qual fato do mundo esta representa.

Como a lógica é altamente declarativa e possibilita construções próximas à linguagem natural, é possível criar sentenças que induzem intuitivamente à interpretação de seu significado, permitindo que a sentença criada por uma pessoa possa ser compreendida por várias outras. A Figura 2.1 mostra que a conexão entre sentenças e fatos é provida pela semântica da linguagem. A propriedade de um fato decorrer de algum outro é espelhada pela propriedade de que uma nova sentença é gerada a partir de outras já existentes (inferência).

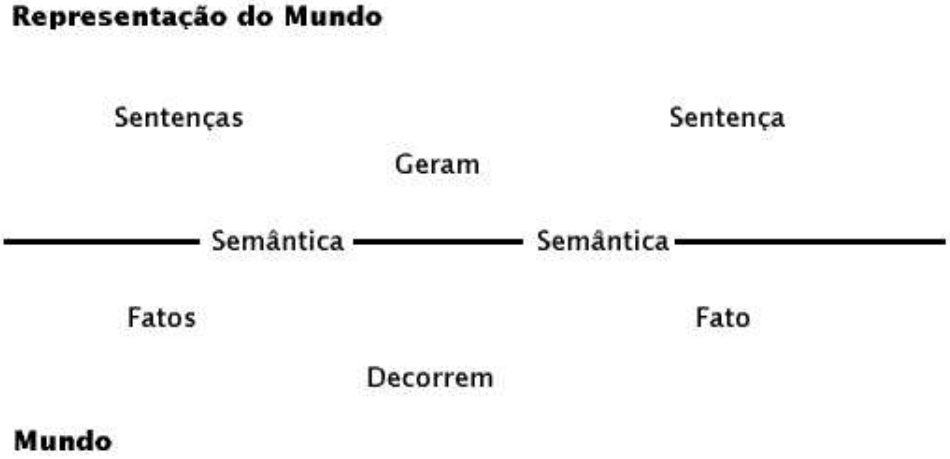

Figura 2.1: Conexão entre as sentenças lógicas e fatos do mundo real.

2.3 Mecanismo de Inferência

Um conjunto de sentenças representando algum conhecimento sobre um determinado domínio[1] forma uma *base de conhecimento*. Um mecanismo de inferência pode, a partir de uma base de conhecimento B, fazer duas coisas: gerar novas sentenças decorrentes de B ou, dada uma sentença S, decidir se S pode ou não ser gerada a partir de B. Queremos que novas sentenças geradas a partir da base de conhecimento sejam sempre verdadeiras dado que as sentenças antigas o são. A chave para isto é ter os passos de inferência respeitando a semântica

[1] Em representação de conhecimento, um domínio é uma porção do mundo sobre a qual desejamos expressar algum conhecimento.

das sentenças em que eles operam. Sendo assim, devem apenas derivar novas sentenças que representam fatos que decorrem de outros contidos na base de conhecimento (Figura 2.1).

Uma *regra de inferência* é uma regra sintática, um padrão de inferência capturado e que, quando aplicado repetidamente a uma ou mais sentenças verdadeiras de uma linguagem, gera apenas novas sentenças verdadeiras. As regras de inferência formam a *teoria de provas* de uma linguagem e fornecem uma estrutura dedutiva às linguagens lógicas [12].

Uma *prova* consiste no conjunto de sentenças geradas através da aplicação de uma seqüência de regras de inferência sobre um conjunto de sentenças para as quais deseja-se provar algo. O trabalho de um mecanismo de inferência é construir provas encontrando a seqüência apropriada de aplicação de regras de inferência.

Dois resultados importantes estão relacionados aos mecanismos de inferência das linguagens lógicas. O primeiro é positivo e diz que: para toda sentença verdadeira é sempre possível encontrar uma prova. O segundo resultado é negativo e conclui que: dada uma sentença falsa, não existe um método que garantidamente seja capaz de decidir, num número finito de passos, sobre sua não validade. Isto significa que a lógica é *semi-decidível*, isto é, dada uma sentença verdadeira é sempre possível encontrar uma prova, mas existem sentenças não verdadeiras para as quais não existe um método finito capaz de decidir sobre sua não validade.

Resumindo, podemos dizer que a lógica é formada pelos seguintes componentes:

1. Uma linguagem formal para descrever o conhecimento, consistindo de:
   - sintaxe, que descreve como construir sentenças
   - semântica, que determina como as sentenças relacionam-se com o conhecimento em questão
2. A teoria de provas — um conjunto de regras para produzir deduções coerentes a partir de um conjunto de sentenças.

A principal característica da lógica é sua alta declaratividade, que constitui seu poder e também sua fraqueza em determinadas situações. Sua generalidade e abstração para representar qualquer coisa que desejamos, força uma representação de conhecimento num nível muito primitivo, levando a um processo "artesenal" e pouco prático. Em aplicações que lidam com domínios que envolvem os números naturais ou reais, valores booleanos, domínios finitos, dentre outros, seria muito útil se o conhecimento básico sobre eles já estivesse disponível, possibilitando assim a utilização de estruturas semânticas mais expressivas. Este é um dos objetivos que lida a introdução de restrições em linguagens lógicas e logo veremos todos os benefícios resultantes de sua utilização.

Nas seções seguintes, iremos apresentar duas das mais conhecidas linguagens lógicas: a lógica proposicional e a lógica de predicados de primeira ordem. A lógica proposicional, apesar de sua simplicidade e limitada expressividade, serve para ilustrar muitos dos conceitos da lógica. Já a lógica de primeira ordem serve como base para a maioria dos trabalhos em IA [54].

2.4 Lógica Proposicional

Na lógica proposicional, símbolos representam toda uma proposição (fato) [54]. Por exemplo, "C" pode ter a interpretação "está chovendo", o que pode ou não ser uma proposição verdadeira. Os símbolos podem ser combinados, através de conectivos booleanos, para gerar sentenças com significados mais complexos. Os símbolos disponíveis são:

- As constantes lógicas *true* e *false*, as quais sozinhas constituem sentenças.

- Símbolos proposicionais tais como C ou P, os quais também constituem sentenças.
- Parênteses "()", que servem para envolver as sentenças e atribuir uma ordem de precedência.

Uma sentença pode ser formada combinando-se outras sentenças entre si através dos conectivos lógicos:

- $\wedge$: Representa o "AND" lógico. Usado para criar conjunções como P $\wedge$ Q.
- $\vee$: Representa o "OR" lógico. Usado para criar disjunções como P $\vee$ (Q $\wedge$ R).
- $\Leftrightarrow$: Indica a equivalência entre duas sentenças.
- $\Rightarrow$: Uma sentença como (P $\wedge$ Q) $\Rightarrow$ R é uma implicação e indica que a premissa (P $\wedge$ Q) implica na conseqüência R.
- $\neg$: Indica a negação de uma sentença.

A gramática da lógica proposicional pode ser expressa em notação BNF como:

Sentença $\rightarrow$ SentençaAtômica | SentençaComplexa

SentençaAtômica $\rightarrow$ *true* | *false* | C | P | Q

SentençaComplexa $\rightarrow$ (Sentença)| Sentença Conectivo Sentença | $\neg$Sentença

Conectivo $\rightarrow \wedge, \vee, \Rightarrow, \Leftrightarrow$

A gramática introduz sentenças atômicas, as quais consistem de um único símbolo, e sentenças complexas, as quais contem conectivos, parênteses ou a negação de uma sentença. É ambígua pois a sentença $P \wedge Q \vee R$ pode ser interpretada tanto como $(P \wedge Q) \vee R$ ou como $P \wedge (Q \vee R)$. Para resolver a ambigüidade, atribui-se uma ordem de precedência para os operadores. A ordem de precedência, da mais alta para a mais baixa, é: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$. Sendo assim, a sentença $\neg P \vee Q \wedge R \Rightarrow S$ é equivalente a $((\neg P) \vee (Q \wedge R)) \Rightarrow S$.

A semântica da linguagem consiste em atribuir um significado para os símbolos proposicionais, para as constantes e para os conectivos lógicos:

- Constantes: *true* sempre significa a veracidade de algo e *false* a falsidade.
- Símbolos proposicionais: podem significar qualquer fato arbitrário.
- Conectivos lógicos: podem ser vistos como uma função que recebe dois valores verdade e retorna um outro.

A tabela verdade 2.1 define a semântica para os conectivos. Sentenças complexas tais como $(P \vee Q) \wedge \neg S$ são definidas através de um processo de decomposição: primeiro determina-se o significado de $(P \vee Q)$ e de $\neg S$ e depois combina-se os resultados utilizando a definição da função $\wedge$. Assim sendo, tabelas verdade não servem apenas para definir o significado dos conectivos, mas podem também ser usadas para determinar se uma dada sentença é ou não verdadeira.

A possibilidade de provar sentenças desta forma é um resultado importante. Ele indica que, se uma máquina possui algumas premissas e uma possível conclusão, ela pode determinar se a conclusão é verdadeira construindo a tabela verdade para a sentença. Se todas as linhas da tabela forem verdadeiras, então a conclusão realmente representa o estado em questão definido pelas premissas. Mesmo pensando que a máquina não tem idéia alguma do que a conclusão significa, o usuário poderá ler as conclusões e usar a sua interpretação para determinar o seu significado.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
FALSE	**FALSE**	TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	**TRUE**	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	**FALSE**	FALSE	FALSE	TRUE	FALSE	FALSE
TRUE	**TRUE**	FALSE	TRUE	TRUE	TRUE	TRUE

Tabela 2.1: Tabela verdade para os cinco conectivos lógicos.

Ao invés de utilizarmos tabelas verdade para provar uma sentença lógica podemos fazer isso através de regras de inferência. Uma regra de inferência pode ser descrita da seguinte forma

$$\frac{A}{B}$$

onde A e B podem ser qualquer sentença. Havendo mais de uma sentença, estas são separadas por vírgula. O funcionamento de uma regra é o seguinte: sempre que alguma sentença na base de conhecimento casa com o padrão que aparece acima da linha, a regra de inferência conclui a sentença abaixo da linha.

Algumas regras de inferência para a lógica proposicional são:

- Modus Ponens ou Eliminação-Implicação: a partir de uma implicação, podemos inferir sua conclusão.
$$\frac{A \Rightarrow B, A}{B}$$
- Eliminação-E: a partir de uma conjunção, podemos inferir qualquer um dos seus itens.
$$\frac{A1 \wedge A2 \wedge A3 \wedge ... \wedge An}{Ai}$$
- Introdução-E: de uma lista de sentenças, podemos inferir sua conjunção.
$$\frac{A1, A2, ..., An}{A1 \wedge A2 \wedge A3 \wedge ... \wedge An}$$
- Introdução-Ou: dada uma sentença, podemos inferir sua disjunção com qualquer outra.
$$\frac{Ai}{A1 \vee A2 \vee A3 \vee ... \vee An}$$
- Eliminação Dupla-Negação: a partir de uma sentença negada duas vezes, podemos inferir uma sentença positiva.
$$\frac{\neg\neg A}{A}$$
- Resolução de Unidade: em uma disjunção, se um dos itens é falso, então podemos inferir que o outro é verdadeiro.
$$\frac{A \vee B, \neg B}{A}$$
- Resolução: é a regra mais complexa e indica a transitividade da implicação.
$$\frac{A \vee B, \neg B \vee Y}{A \vee Y} \text{ ou } \frac{\neg A \Rightarrow B, B \Rightarrow Y}{\neg A \Rightarrow Y}$$

Como já havíamos mencionado, uma prova consiste da aplicação de uma seqüência de regras de inferência, começando inicialmente com sentenças presentes na base de conhecimento e culminando na geração da sentença cuja prova é desejada. O trabalho de um mecanismo de inferência, então, é construir provas encontrando a seqüência apropriada para aplicação de regras de inferência.

Um mecanismo de inferência é dito *completo* quando, dada uma sentença verdadeira, ele *sempre* consegue provar sua validade. O método de construção de tabelas verdade é completo, porém seu algoritmo é exponencial em n (2^n linhas). A construção de procedimentos de provas para sentenças da lógica proposicional recai no problema de verificar sua satisfatibilidade, o qual foi demonstrado por Cook, em 1971, ser NP-completo. Porém, na maioria dos casos, a prova de uma sentença envolve apenas uma pequena porção da base de conhecimento, podendo ser encontrada rapidamente. Sendo assim, um mecanismo de prova utilizando regras de inferência é geralmente mais eficaz que a construção de tabelas verdade, já que precisa verificar apenas poucas sentenças na base enquanto o outro necessariamente gastará um tempo exponencial.

2.5 Busca em Espaço de Estados

Muitos tipos de problemas requerem algum tipo de busca no espaço de possíveis soluções para determinar qual delas é a correta. Antes de apresentarmos a lógica de primeira ordem, mostraremos alguns dos principais mecanismos de busca utilizados para solucionar tais tipos de problemas. Eles são a base para implementação de algoritmos de inferência para sistemas lógicos e outros.

Podemos definir uma busca como sendo a procura por uma solução entre vários estados possíveis que formam a entrada de um problema. No início, estamos no chamado estado inicial e a partir dele desejamos chegar no estado objetivo. O primeiro passo é testar se esse é um estado objetivo. Se não for, teremos que expandir o estado atual e considerar outros estados. Isto pode levar ao surgimento de um ou mais estados, sendo que, sempre quando for mais do que um, teremos que escolher qual considerar primeiro. Esta é a essência da busca: escolher uma opção e pôr as outras de lado para depois, no caso da primeira escolha não levar a uma solução, voltarmos e considerarmos outra opção. Continuamos escolhendo, testando e expandindo até que uma solução seja encontrada ou até que não haja mais estados para serem expandidos. A escolha de qual estado expandir primeiro é determinada pela *estratégia de busca*.

Podemos representar o processo de busca como uma *árvore de busca*, onde o nó raiz corresponde ao estado inicial e cada nó folha corresponde a estados que não possuem sucessores na árvore, seja porque ele não foi expandido ainda ou porque foi expandido mas gerou o estado vazio. Em cada passo, o algoritmo de busca escolhe um nó folha para expandir.

Veremos agora dois métodos que implementam estratégias de busca diferentes, a *busca em largura* e a *busca em profundidade*. Neles, a diferença na estratégia de busca está baseada apenas na ordem em que os nós serão expandidos. Algoritmos mais sofisticados podem fazer uso de heurísticas do problema para ajudar nesta escolha, produzindo assim resultados mais eficientes [54].

2.5.1 Busca em Largura

Na busca em largura o nó raiz é expandido primeiro, depois são expandidos todos os nós gerados a partir do nó raiz e assim sucessivamente. Em geral, todos os nós na profundidade d são expandidos antes dos nós da profundidade $d + 1$.

Se existe uma solução, a busca em largura garantidamente irá encontrá-la, o que faz dela uma estratégia completa. Além disso, ela sempre irá encontrar a solução menos profunda na árvore e isso geralmente significa a solução ótima (dado que o custo de um caminho seja uma função não decrescente da profundidade da árvore).

Num espaço de estados onde cada nó pode ser expandido para b novos estados (fator de ramificação b), teremos b^h nós em cada nível h da árvore, o que resulta num número máximo de b^d nós expandidos antes de se achar uma solução cujo caminho tem tamanho d. Nos deparamos então com uma complexidade de tempo e memória (pois todos os nós folhas devem ser mantidos em memória) de $O(b^d)$, o que torna essa estratégia impraticável mesmo para entradas modestas.

2.5.2 Busca em Profundidade

A busca em profundidade sempre expande um dos nós do nível mais profundo da árvore. Apenas quando a busca chega no nível mais profundo de um ramo é que ela volta e expande os nós dos níveis mais superficiais. Requer uma quantia modesta de memória, pois precisa armazenar somente um único caminho da raiz até a folha. Para um espaço de estados com fator de ramificação b e profundidade máxima m, requer o armazenamento de apenas bm nós. Sua complexidade ainda é exponencial, $O(b^m)$ no pior caso.

Para problemas que possuem muitas soluções, a busca em profundidade geralmente é mais rápida do que a busca em largura, pois existe uma grande chance dela encontrar uma dessas soluções após explorar apenas uma pequena porção da árvore. O maior problema deste método é que a procura pode embrenhar por um caminho errado e nunca mais retornar dele, já que muitos problemas possuem árvores de busca muito profundas ou mesmo infinitas. Dadas suas características, podemos concluir que a busca em profundidade não é nem completa nem ótima e deve ser evitada para árvores de busca muito profundas ou infinitas.

Uma técnica simples para contornar alguns desses problemas é definir um limite máximo para a profundidade da árvore. Neste caso, a busca é completa porém não e ótima. Para a maioria dos problemas no entanto, não é possível definir um limite para a profundidade até que o problema tenha sido resolvido. A estratégia conhecida como *busca em profundidade iterativa* contorna este problema escolhendo o melhor limite de profundidade, pois testa todos os limites possíveis. Esta técnica combina as vantagens da busca em largura e busca em profundidade, ela é ótima e completa como a busca em largura e requer pouca memória como a busca em profundidade. Sua complexidade de espaço é de $O(bd)$ e de tempo $O(b^d)$, onde d é a profundidade e b o fator de ramificação da árvore.

2.6 Lógica de Primeira Ordem

A lógica proposicional é uma das linguagens lógicas mais simples, sendo bastante limitada em sua capacidade de expressar aspectos relevantes do mundo, fazendo apenas a alusão de que o mundo consiste de fatos.

O cálculo de predicados de primeira ordem ou lógica de primeira ordem representa o mundo em termos de *objetos* e das *propriedades* destes objetos. É possível representar também as *relações* existentes entre os objetos, sendo que algumas delas são *funções*, isto é, relações nas quais existe apenas um valor para uma dada entrada. Alguns exemplos de objetos, propriedades, relações e funções são:

- Objetos: pessoas, casas, números, teorias, guerras, cores, etc.
- Propriedades: vermelho, primo, rugoso, etc.

- Relações: irmão de, maior que, dentro de, parte de, tem cor, etc.
- Funções: pai de, melhor amigo, adição, multiplicação, etc.

Dividir o mundo em objetos e suas relações nos ajuda a raciocinar sobre ele e nos permite expressar fatos sobre qualquer aspecto do mundo real.

A lógica de primeira ordem dá ao usuário a liberdade para descrever as coisas da forma mais apropriada para o domínio em questão. Existem muitos esquemas de representação em IA, alguns equivalentes à lógica de primeira ordem em sua teoria e outros não, porém, a lógica de primeira ordem é universal no sentido de que pode expressar qualquer coisa que pode ser programada e é o esquema mais estudado e melhor compreendido já projetado [54].

2.6.1 Sintaxe e Semântica

Na lógica proposicional toda expressão é uma sentença representando um fato. A lógica de primeira ordem possui sentenças, mas também possui *termos*, os quais representam objetos. Constantes, variáveis e funções são usados para construir termos, enquanto quantificadores e símbolos predicados são utilizados para construir sentenças. A gramática completa em notação BNF é,

Sentença → SentençaAtômica
| Sentença Conectivo Sentença
| Quantificador Variável, ... Sentença
| ¬Sentença
| (Sentença)

SentençaAtômica → Predicado(Termo,...) | Termo = Termo

Termo → Função(Termo,...) | Constante | Variável

Conectivo → $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$

Quantificador → $\exists$ | $\forall$

Constante → A | X1 | John | ...

Variável → a | x | ...

Predicado → Antes | TemCor | Chovendo | ...

Função → MãeDe | PernaEsquerdaDe | ...

onde:

- Constantes: uma interpretação deve especificar qual objeto do mundo um símbolo de constante representa. Cada constante refere-se a exatamente um objeto, mas nem todo objeto precisa ter um nome e alguns podem ter mais de um.
- Predicados: uma interpretação especifica que um símbolo predicado refere-se a uma relação em particular no modelo. Por exemplo, o símbolo IrmãoDe pode se referir a relação de irmandade. Neste caso, IrmãoDe é um predicado binário, uma relação entre pares de objetos.
- Funções: algumas relações são funcionais, isto é, um objeto relaciona-se a exatamente um outro objeto através da relação. Um ângulo tem apenas um número que é o seu coseno, uma pessoa tem apenas uma outra pessoa que é o seu pai e assim por diante. Diferentemente dos predicados, que são usados para relatar relações entre certos objetos, os símbolos funcionais são usados para se referir a determinados objetos sem a necessidade de dar um nome a eles.

A escolha dos nomes para os símbolos que serão constantes, predicados e funções é inteiramente do usuário. Formalmente os nomes não têm importância, mas melhoram muito a compreensão se a interpretação pretendida para os símbolos for clara.

- Termos: um *termo* é uma expressão lógica que se refere a um objeto. Sendo assim, variáveis e constantes são termos. Às vezes é mais conveniente usar uma expressão para se referir a um objeto do que ter de dar um nome a ele. É justamente para estes casos que existe as funções. Uma função representa um termo complexo que por sua vez é apenas um tipo complicado de nome.

- Sentenças atômicas: vimos que existem os termos que se referem a objetos e predicados que definem relações entre objetos. Agora podemos combinar os dois para criar *sentenças atômicas*, as quais representam fatos. Uma sentença atômica é formada por um predicado seguido de uma lista de termos como seus argumentos.

- Sentenças complexas: podemos usar conectivos lógicos para criar sentenças mais complexas. A semântica das sentenças formadas utilizando-se os conectivos é como na lógica proposicional.

- Quantificadores: às vezes precisamos expressar propriedades de uma coleção inteira de objetos e os *quantificadores* nos permitem fazer isso. A lógica de primeira ordem contém dois quantificadores, chamados universal e existencial.

  - Quantificador universal ($\forall$): permite expressar regras genéricas, evolvendo uma classe de objetos como em: $\forall$ x, se x=y $\Rightarrow$ x>z.
  - Quantificador existencial ($\exists$): enquanto o quantificador universal permite criar sentenças sobre todos os objetos, o existencial permite criar sentenças sobre algum objeto no universo sem termos de nomeá-lo: $\exists$ x x>z.

- Igualdade: uma outra maneira de criarmos sentenças atômicas é usando o símbolo de igualdade "=". Ele indica que dois termos referem-se ao mesmo objeto. A igualdade pode ser vista como um símbolo predicado com um *significado pré-definido*, isto é, ela representa sempre a *relação de identidade*. A relação de identidade é pré-definida a ser formada por todos os pares de objetos existentes na base de conhecimento onde ambos os elementos de cada par são o mesmo objeto. Podemos usar o símbolo de igualdade juntamente com a negação quando queremos indicar que dois termos não são o mesmo objeto.

2.6.2 Mecanismos de Inferência

As regras de inferência aplicadas à lógica proposicional: Modus Ponens, Eliminação-E, Introdução-E, Introdução-Ou e Resolução, também se aplicam à lógica de primeira ordem. Além destas, outras três regras são necessárias para tratar dos quantificadores. Podemos considerá-las mais complexas do que as anteriores pois elas envolvem a substituição de valores no lugar de variáveis. Usaremos a notação $\text{Subst}(q, a)$ para denotar o resultado da substituição de q na sentença a.

As três novas regras de inferência são as seguintes:

- Eliminação-Universal: para qualquer sentença α, variável v e termo g que não contém nenhuma variável:

$$\frac{\forall v \alpha}{\text{Subst}(v/g, \alpha)}$$

- Eliminação-Existencial: para qualquer sentença α, variável v e constante k que não aparece em nenhum outro lugar na base de conhecimento:

$$\frac{\exists v \alpha}{\text{Subst}(v/k, \alpha)}$$

- Introdução-Existencial: para qualquer sentença α, variável v que não ocorre em α e uma constante g (que não possui variáveis) que ocorre em α:

$$\frac{\alpha}{\exists v \, \text{Subst}(g/v, \alpha)}$$

A aplicação de regras de inferência é basicamente uma questão de casar os padrões que formam suas premissas com as sentenças na base de conhecimento e então adicionar suas conclusões. Exemplos de sua utilização podem ser encontrados em [54].

2.6.3 Modus Ponens Generalizado

Veremos nesta seção uma generalização da regra de inferência Modus Ponens a qual realiza de uma só vez o que geralmente requer uma Introdução-E, uma Eliminação-Universal e um Modus Ponens.

Genericamente, se existe alguma substituição envolvendo x que torna a premissa da implicação idêntica a alguma sentença presente na base de conhecimento, então podemos deduzir a conclusão da implicação. A versão generalizada do Modus Ponens faz isso para nós. Formalmente, a nova regra é definida da seguinte forma:

- Modus Ponens Generalizado: para sentenças atômicas pi, pi', e q, onde existe uma substituição θ tal que $\text{Subst}(\theta, pi') = \text{Subst}(\theta, pi)$, para todo i:

$$\frac{p1', p2', ..., pn', (p1 \wedge p2 \wedge ... \wedge pn \Rightarrow q)}{\text{Subst}(\theta, q)}$$

Existem $n+1$ premissas para esta regra: as n sentenças atômicas pi' e a implicação. Há uma conclusão: o resultado da aplicação da substituição para a conseqüência q.

O Modus Ponens Generalizado é uma regra eficiente por três razões:

1. Combina inferências menores numa única realizada de uma só vez.
2. Realiza substituições que garantidamente irão ajudar na resposta ao invés de randomicamente ficar tentando as eliminações universais. O algoritmo de *unificação* recebe duas sentenças e retorna uma substituição que as faz ficar idênticas caso esta substituição exista.
3. Converte as sentenças na base de conhecimento para sua *forma canônica*. Fazendo isso uma única vez no início, significa que não será necessário gastar tempo com conversões durante o curso da prova.
4. Executa em tempo polinomial (porém, como iremos ver, é incompleto).

Unificação

O trabalho da rotina de unificação, a qual chamaremos de Unify, é obter duas sentenças atômicas p e q e retornar uma substituição que faz com que p e q parecerem iguais (identidade sintática). Se não houver tal substituição a rotina falha. Formalmente,

$$\text{Unify}(p, q) = q$$

onde $\text{Subst}(q, p) = \text{Subst}(q, q)$ e q é chamado de unificador das duas sentenças.

Forma Canônica

Estamos tentando construir um mecanismo de inferência com apenas uma regra de inferência — a versão generalizada do Modus Ponens. Isto significa que todas as sentenças na base de conhecimento devem estar num formato que casa com uma das premissas do Modus Ponens, caso contrário, elas nunca poderão ser usadas.

A forma canônica indica que cada sentença seja uma sentença atômica ou uma implicação com uma conjunção de sentenças atômicas do lado esquerdo e um único átomo no lado direito. Sentenças neste formato pertencem a uma classe denominada sentenças de Horn e possuem a seguinte forma:

$$P1 \wedge P2 \wedge ... \wedge Pn \Rightarrow Q$$

onde Pi e Q são átomos não negados. Quando Q é a constante *false*, temos uma sentença equivalente a $\neg P1 \vee ... \vee \neg Pn$. Quando $n = 1$ e $P1 = true$, temos $true \Rightarrow Q$, que é equivalente a sentença atômica Q. Quando uma base de conhecimento contém apenas sentenças de Horn ela é dita estar na Forma Normal de Horn. Nem toda base de conhecimento pode ser escrita como uma coleção de sentenças de Horn, mas para aquelas que podem, é suficiente usar o Modus Ponens para fazer qualquer inferência. Veremos mais adiante, quando estivermos falando sobre o Prolog, que este utiliza apenas bases que estejam na Forma Normal de Horn.

Na próxima seção, apresentamos algoritmos para realização sistemática de deduções utilizando o Modus Ponens. Estes algoritmos formam a base para muitas aplicações em IA [54].

2.6.4 Encadeamento Progressivo e Regressivo

O Modus Ponens generalizado pode ser usado de duas maneiras. Podemos começar com as sentenças na base de conhecimento e então gerar novas conclusões que poderão possibilitar a realização de mais inferências. Isto é chamado de *encadeamento progressivo* (*forward chaining*) e é geralmente utilizado quando um novo fato é adicionado à base e desejamos gerar suas conseqüências. Alternativamente, podemos iniciar com uma sentença que desejamos provar e então encontrar as implicações que poderão nos ajudar a concluir a prova. Isto é chamado de *encadeamento regressivo* (*backward chaining*) e geralmente é utilizado quando existe um objetivo a ser provado.

Encadeamento Progressivo

Geralmente é disparado assim que um novo fato p é adicionado à base de conhecimento. A idéia é encontrar todas as implicações que possuem p como premissa e então, se as outras premissas que compõe a implicação também são conhecidas, adicionar a conseqüência da implicação na base de conhecimento, o que possivelmente irá disparar novas inferências.

O encadeamento progressivo faz uso de dois conceitos:

- Renomeação: Uma sentença é uma renomeação de outra se ela for idêntica a outra exceto pelo nome de suas variáveis.
- Composição: Uma composição Compose(θ_1, θ_2) é a substituição cujo efeito é o mesmo de aplicar cada substituição separadamente. Isto é,

$$\text{Subst}(\text{Compose}(\theta_1, \theta_2), p) = \text{Subst}(\theta_2, \text{Subst}(\theta_1, p))$$

.

Abaixo segue o algoritmo em pseudo-código do processo de encadeamento progressivo. A expressão `Primeiro(sentenças)` denota a primeira sentença do conjunto `sentenças` e a expressão `Resto(sentenças)` denota o conjunto `sentenças`$-\{$`Primeiro(sentenças)`$\}$.

```
procedure Encadeamento-Progressivo(B,p)
se existe uma sentença em B que é uma renomeação de p então
   retorne
Adicione p a B
para cada (p1 ∧ ... ∧ pn ⇒ q) em B tal que para algum i Unify(p,pi)=θ faça
   Encontre-E-Infira(B, [p1, ..., p_{i-1}, p_{i+1}, ..., pn], q, θ)
fim

procedure Encontre-E-Infira(B,premissas,conclusão,θ)
se premissas = [] então
   Encadeamento-Progressivo(B,Subst(θ,conclusão))
se não para cada p' em B tal que Unify(p',Subst(θ,Primeiro(premissas)))=θ faça
   Encontre-E-Infira(B, Resto(premissas),conclusão,Compose(θ,θ2))
fim
```

Seu funcionamento é o seguinte: se p já está na base de conhecimento B, nada é feito. Se p é novo, cada implicação contida em B que possui sua premissa (ou parte dela) correspondendo com p é analisada. Para cada implicação desta, se todas as sentenças que compõe sua premissa também estão em B, então sua conclusão deve ser adicionada a B. Para cada nova conclusão adicionada, todo o processo se repete procurando por novas inferências.

Encadeamento Regressivo

O objetivo do encadeamento regressivo é encontrar todas as respostas para uma questão feita à base de conhecimento. O algoritmo tenta encontrar todas as implicações onde a conclusão unifica com a questão e então verifica se as premissas destas implicações são conhecidas. O procedimento recebe como argumento uma lista `qlist` com uma conjunção de sentenças que formam a questão e a substituição corrente (inicialmente vazia). O retorno é o conjunto de todas as substituições que possivelmente tornam a questão verdadeira.

O algoritmo em pseudo-código deste método é o seguinte:

```
procedure Encadeamento-Regressivo(B,qlist,θ)
se qlist vazia então
   retorne θ
q ← Primeiro(qlist)
para cada q'_i em B tal que θ_i ← Unify(q,q'_i) faça
   Adicione Compose(θ,θ_i) a resposta
para cada sentença (p1 ∧ ... ∧ p_n ⇒ q'_i) em B tal que θ_i ⇐ Unify(q,q'_i) faça
   resposta ← Encadeamento-Regressivo(B, Subst(θ_i,[p1 ... p_n]),
      Compose(θ,θ_i)) ∪ resposta
retorne a união do Encadeamento-Regressivo(B, Resto(qlist), θ) para cada
   θ pertencente a resposta
fim
```

Perceba o papel importante que a rotina de unificação desempenha em ambos os algoritmos.

2.6.5 Resolução

Vamos supor a seguinte base de conhecimento:

$\forall x P(x) \Rightarrow Q(x)$
$\forall x \neg P(x) \Rightarrow R(x)$

$\forall x Q(x) \Rightarrow S(x)$
$\forall x R(x) \Rightarrow S(x)$

É fácil observar que podemos concluir $S(A)$. $S(A)$ é verdadeiro se $Q(A)$ ou $R(A)$ é verdadeiro e um dos dois certamente é pois $P(A)$ ou $\neg P(A)$ é verdadeiro.

Os algoritmos de encadeamento utilizando Modus Ponens não conseguem derivar $S(A)$. O problema é que $\forall x \neg P(x) \Rightarrow R(x)$ não pode ser convertido para forma normal de Horn e conseqüentemente não pode ser usado pelo Modus Ponens. Isso significa que um procedimento de prova utilizando Modus Ponens é incompleto, isto é, existem sentenças que são geradas pela base de conhecimento mas que o procedimento não é capaz de inferir.

O matemático alemão Kurt Gödel provou, em 1930, a existência de um método completo para a lógica de primeira ordem. Esta não é uma conclusão trivial pois sentenças universalmente quantificadas e símbolos funcionais arbitrariamente aninhados podem levar a bases de conhecimento infinitas [54]. Gödel mostrou que um procedimento de prova existia mas não demonstrou nenhum. Somente em 1965 foi que Robinson publicou seu algoritmo de resolução, o qual iremos discutir em seguida. Um mecanismo de prova completo é de grande valor, pois possibilita uma máquina resolver qualquer problema descrito com a linguagem.

Como havíamos dito no início deste capítulo, a lógica é semi-decidível, isto é, se uma sentença pode ser derivada da base de conhecimento, então isso pode ser provado, porém, quando ela não pode ser gerada, nem sempre podemos mostrar isso. Na prática, isto significa que os métodos para a verificação da validade de sentenças podem nunca terminar.

A regra de inferência resolução, que foi mostrada quando tratamos da lógica proposicional, possui a seguinte forma:

$$\frac{A \vee B, \neg B \vee Y}{A \vee Y} \text{ ou } \frac{\neg A \Rightarrow B, B \Rightarrow Y}{\neg A \Rightarrow Y}$$

A regra pode ser interpretada de duas maneiras. Uma forma seria vê-la como um raciocínio por casos. Se B é falso então, a partir da primeira disjunção, A deve ser verdadeiro; mas se B é verdadeiro então, a partir da segunda disjunção, Y deve ser verdadeiro. Sendo assim, A ou Y devem ser verdadeiros. Uma outra maneira de interpretá-la é como a transitividade da implicação: a partir de duas implicações podemos derivar uma terceira que associa a premissa da primeira com a conclusão da segunda. Note que o Modus Ponens não permite derivar novas implicações, ele apenas deriva conclusões atômicas. Sendo assim, a regra da resolução é mais poderosa do que o Modus Ponens. Podemos generalizar a regra acima e torná-la uma regra de inferência completa para a lógica de primeira ordem. Ao invés das premissas terem apenas duas disjunções, podemos estender para uma regra que diz: para duas disjunções de qualquer tamanho, se uma das disjunções numa cláusula (p_j) unifica com a negação de uma disjunção na outra cláusula (q_k), então infira a disjunção de todos os componentes exceto para aqueles dois.

- Resolução Generalizada (disjunções): Para literais[2] p_i e q_i onde $\text{Unify}(p_j, \neg q_k) = q$

$$\frac{p_1 \vee ...p_j... \vee p_m, q_1 \vee ...q_k... \vee q_n}{\text{Subst}(q, (p_1 \vee ...p_{j-1} \vee p_{j+1}...p_m \vee q_1...q_{k-1} \vee q_{k+1}...q_n))}$$

Uma prova com esta regra de resolução é mais poderosa do que utilizando Modus Ponens, porém ainda não é completa. Por exemplo, se quiséssemos provar $P \vee \neg P$ de uma base vazia, por mais que a sentença seja válida, como não há nada na base de conhecimento, não há nada para a resolução ser aplicada e nada poderá ser provado. Um procedimento de inferência completo usando resolução é a *refutação*, também conhecido como prova por

[2] O termo literal significa uma sentença atômica negada ou não.

contradição. A idéia é que, para provar P, assumimos que P é falso, isto é, que $\neg P$ existe na base de conhecimento, e provamos a contradição. Em outras palavras:

$$(B \wedge \neg P \Rightarrow False) \Leftrightarrow (B \Rightarrow P)$$

Prova por contradição é uma técnica poderosa utilizada pelos matemáticos e a resolução nos dá um modo simples e completo para a utilizarmos. A resolução pode ser utilizada desta maneira num procedimento de encadeamento progressivo ou regressivo tal qual o Modus Ponens.

Um fator decisivo para a aplicabilidade do método de resolução é o tamanho do problema em questão, tendo em vista a complexidade exponencial inerente aos algoritmos de busca. Algumas técnicas foram desenvolvidas para guiar a busca por uma prova e melhorar a eficiência deste processo. Estas baseiam-se na constatação de que, dado um conjunto de cláusulas, nem todas resoluções possíveis em cada nível precisam ser realizadas para se chegar a uma prova por refutação. A maioria das estratégias de controle são completas e cada uma define um critério diferente para a escolha das cláusulas candidatas à resolução. Algumas das principais estratégias são: conjunto de suporte, entrada, linear e preferência de unidade.

O desenvolvimento de métodos sistemáticos, como a teoria da resolução, e de estratégias de controle para estes métodos, contribuíram fortemente para o surgimento da programação lógica, sendo um de seus mais importantes representantes o Prolog [12].

2.6.6 Prolog

O Prolog é um dos mais importantes representantes da classe de linguagens lógicas de programação. Uma de suas principais particularidades é que apenas cláusulas de Horn são permitidas para formar sua base de conhecimento. Na sua forma mais geral, uma cláusula pode ser escrita como:

$$A_1(x_1, ..., x_k) \wedge ... \wedge A_m(x_1, ..., x_k) \Rightarrow B_1(x_1, ..., x_k) \vee ... \vee B_n(x_1, ..., x_k).$$

onde $A_i(x_1, ..., x_k)$ e $B_i(x_1, ..., x_k)$ são sentenças atômicas da forma $P(t_1, ..., t_z)$ onde as variáveis $x_1, ..., x_k$ ocorrem nos termos t_j. Para denotar sentenças atômicas fechadas, isto é, sentenças cujos termos t_j não possuem variáveis, usaremos apenas os símbolos A_i e B_i.

Uma cláusula pode ser lida então, sob a forma de uma *regra*: se $A_1(x_1, ..., x_k)$ e ... e $A_m(x_1, ..., xk)$ então conclua $B_1(x_1, ..., x_k)$ ou ... ou $B_n(x_1, ..., xk)$. Variando os parâmetros k, n e m, obtemos os seguintes tipos de cláusulas com a lógica de primeira ordem:

1. Geral: $k > 0, m \geqslant 1, n > 1$
   Se $A_1(x_1, ..., x_k) e ... e A_m(x_1, ..., xk)$
   então $B_1(x_1, ..., x_k) ou ... ou B_n(x_1, ..., xk)$.

2. Positiva: $k > 0, m = 0, n > 1$
   $B_1(x_1, ..., x_k) ou ... ou B_n(x_1, ..., xk)$.

3. Fechada: $k = 0, m \geqslant 1, n > 1$ Se $A_1 e ... e A_m$ então $B_1 ou ... ou B_n$.

4. Positiva Fechada: $k = 0, m = 0, n > 1$
   $B_1 ou ... ou B_n$.

5. Cláusula de Horn Geral: $k > 0, m \geqslant 1, n = 1$
   Se $A_1(x_1, ..., x_k) e ... e A_m(x_1, ..., xk)$
   então $B_1(x_1, ..., x_k)$.

6. Cláusula de Horn Positiva: $k > 0, m = 0, n = 1$
   $B_1(x_1, ..., x_k)$.

7. Cláusula de Horn Fechada: $k = 0, m \geqslant 1, n = 1$
   Se $A_1 e ... e A_m$ então B_1.

8. Cláusula de Horn Positiva e Fechada: $k = 0, m = 0, n = 1$ B_1.

O Prolog utiliza apenas cláusulas dos tipos 5 a 8, que correspondem às cláusulas de Horn. Perceba como elas se encaixam na definição que demos anteriormente, quando tratamos de Modus Ponens Generalizado, de que cada sentença na forma normal de Horn deve ser uma sentença atômica ou uma implicação. Caso seja uma implicação, a premissa será formada por uma conjunção de átomos não negados e a conseqüência formada por apenas uma sentença atômica.

O núcleo da execução de um programa em Prolog é a rotina de unificação vista anteriormente. Este, utiliza a busca em profundidade para varrer uma base de fatos e encontrar todas as soluções para a consulta de um usuário. Um ponto importante para compreendermos a ineficiência da execução de programas baseados em lógica é que, no Prolog, e na maioria dos sistemas que utilizam a lógica tradicional, quando um novo estado é escolhido na busca pela solução, nenhum teste é feito com antecedência para verificar se aquela escolha tem realmente chance de levar à solução. Isto porque, nestes sistemas, nada é feito para tirar proveito de informações sobre um problema (codificadas na base de conhecimento) para gerar um procedimento de busca mais inteligente. Dessa forma, quando o retrocesso ocorre, ele já montou uma solução completa, isto é, chegou até um nó folha terminal na árvore de busca. Isso faz com que o procedimento de retrocesso (*backtracking*) [9] utilizado assemelhe-se ao método de *generate-and-test* [9] o qual "adivinha" a solução e então testa para verificar se é correta. Para problemas complexos, este processo de busca pode se tornar inviável e caracteriza a raiz da ineficiência do Prolog.

2.7 Comentários Finais

A principal vantagem da construção de sistemas baseados em conhecimento sobre a programação tradicional é que estes, quando devidamente utilizados, podem requerer menos esforço para serem criados [54]. É necessário apenas decidir quais são os objetos e as relações entre estes objetos que precisam ser representadas, enquanto na programação com linguagens procedimentais, além de ser necessário fazer isso, também é preciso decidir como computar as relações. Num sistema baseado em conhecimento especificamos o que é verdadeiro e o procedimento de inferência é que se preocupa em como processar os fatos numa solução para o problema. Além disso, como um fato é verdadeiro não importando qual tarefa estejamos tentando resolver, as bases de conhecimento podem, em princípio, serem reutilizadas para uma variedade de tarefas diferentes sem modificações. Por fim, a depuração de uma base de conhecimento é mais fácil, tendo em vista que qualquer sentença é verdadeira ou falsa por si só, enquanto a corretude de um programa depende fortemente de seu contexto.

A abordagem clássica da lógica que vimos neste capítulo não é adequada para resolver muitos dos problemas reais, principalmente os científicos e de engenharia, os quais requerem uma performance elevada e estão geralmente relacionados com processamento matemático e não somente simbólico. Apesar das inúmeras estratégias de controle propostas para melhorar a performance do algoritmo de resolução e das melhorias para tornar o interpretador Prolog mais eficiente, para problemas complexos, a busca em profundidade nestas bases possui uma performance aquém do necessário. Outro problema com a abordagem clássica é a abstração dos seus elementos semânticos. A falta de estruturas semânticas mais objetivas e expressivas

para trabalhar, por exemplo, com expressões matemáticas no domínio dos inteiros ou reais, torna necessária a definição de objetos e suas relações num nível muito primitivo, resultando em perda de produtividade.

No capítulo seguinte, apresentaremos os fundamentos de uma classe de linguagens de programação baseada no paradigma da programação lógica e resolução de restrições — a Programação Lógica com Restrição — que possui características que resolvem em grande parte os problemas da programação lógica tradicional. Ela possibilita a introdução de objetos semânticos mais expressivos, através da codificação de restrições como predicados, enriquecendo a linguagem e tornando-a mais apta, por exemplo, a tratar de expressões aritméticas. Além disso, a utilização de restrições possibilita um processo de busca muitas vezes mais eficiente, pois estas codificam fatos que guiam o processo de inferência, permitindo "podar" a árvore de busca para eliminar caminhos que certamente não levarão à solução.

CAPÍTULO 3

Programação Lógica com Restrição

3.1 Introdução

A Programação Lógica com Restrição ou, do inglês, *Constraint Logic Programming* (CLP) é uma classe de linguagens de programação baseada na programação lógica e programação com restrições. A CLP descende de estudos em áreas diversas como inteligência artificial, pesquisa de operações e linguagens de programação e em pouco tempo evoluiu, de um simples tópico de pesquisa, para um poderoso paradigma de programação, que cada vez mais vem sendo utilizado para modelar e resolver muitos problemas reais complexos.

Por possuírem a declaratividade e flexibilidade da programação lógica, linguagens baseadas na CLP podem gerar programas mais fáceis de manter, modificar ou estender e, por fazerem uso de restrições e das técnicas desenvolvidas para resolvê-las, alcançam uma eficiência similar às soluções implementadas com linguagens procedimentais [29]. Em 1996 a ACM — *Association for Computing Machinery* — reconheceu o poder da CLP, listando-a como uma das poucas áreas da Ciência da Computação que são classificadas como sendo de "direções estratégicas" [28].

Apesar da dispersão do material sobre este tópico, o qual ainda não foi devidamente consolidado, e da diversidade de terminologias adotadas em diferentes partes da literatura, tentaremos neste capítulo abordar de forma concisa, porém completa, as idéias fundamentais desta importante tecnologia. O capítulo está organizado da seguinte maneira. Na Seção 3.2 apresentamos um breve histórico da programação com restrições, desde suas linhas iniciais até as idéias que levaram à criação da CLP e de um novo paradigma para desenvolvimento e modelagem de sistemas. Na Seção 3.3 descrevemos as bases do Esquema de Programação Lógica com Restrições, chamado de CLP(X), assim como foi formalmente definido em [34]. Será mostrada a sintaxe e semântica declarativa e operacional do esquema, além de um dos mais importantes domínios de aplicação — o domínio aritmético. Desde o princípio a CLP foi fortemente influenciada pelas Técnicas de Consistência, que é o assunto abordado na Seção 3.4. Essas técnicas têm provado serem muito eficientes numa variedade de problemas que envolvem busca, de tal modo que foram introduzidas como um dos principais aspectos da CLP. Depois de sua criação, um dos requisitos aos sistemas CLP passou a ser que "os mecanismos de resolução de restrições sejam passíveis de modificação pelos usuários" [3]. É

disso que tratamos na Seção 3.5, onde apresentamos uma das propostas mais interessantes neste sentido. Por fim, a Seção 3.6 traz um resumo dos principais sistemas e aplicações CLP já implementados.

3.2 Histórico e Principais Conceitos

As restrições ocorrem naturalmente em várias atividades humanas e também como meio de expressão para formalizar muitas das regularidades que caracterizam o mundo físico, natural ou projetado, e suas abstrações matemáticas. Por exemplo, na descrição de um circuito elétrico, uma restrição típica é que "a soma das correntes que chegam a um nó deve ser zero". Ou ainda, em sistemas de interface gráfica, pode existir a restrição de que "dois objetos não devem se sobrepor" [53].

Uma restrição pode ser intuitivamente pensada como uma limitação num espaço de possibilidades [60]. Restrições matemáticas especificam relações lógicas ou aritméticas entre variáveis, as quais possuem seus valores associados a algum domínio. Por exemplo, uma restrição c1, definida por $x + y >= z$, representa, supondo que o domínio seja o dos números naturais e que x, y e z sejam variáveis, que a soma de x mais y deve ser maior ou igual a z. As restrições limitam os valores que as variáveis podem assumir e representam alguma informação (parcial) sobre elas. Algumas de suas características são [60]:

- As restrições podem especificar informação parcial, ou seja, elas não precisam definir exatamente o valor de suas variáveis.
- São aditivas, isto é, supondo que a restrição c1 acima esteja sendo processada, podemos adicionar uma outra restrição c2, definida por $x + y <= z$, sem que isso prejudique o processamento realizado até o momento. A ordem em que as restrições são processadas não importa, somente o que importa é que no final a conjunção destas seja satisfeita.
- As restrições raramente são independentes, podendo a combinação delas produzir novas informações. Por exemplo, uma vez que c1 e c2 tenham sido impostas, é o caso disto gerar a dedução de que $x + y = z$.
- São multi-direcionais: uma restrição em, digamos, duas variáveis x e y, pode ser usada para inferir uma restrição em x, dada a restrição em y, ou vice versa. Por exemplo, a restrição $x = y + 1$ pode ser usada desta forma ou para inferir $y = x - 1$.
- Por último, são declarativas, isto é, alguém pode usá-las para especificar quais relacionamentos devem ser satisfeitos mas sem a necessidade de quem o fez também especificar um procedimento computacional para garantir estes relacionamentos.

Programação com Restrição (do inglês *Constraint Programming*) é o estudo de sistemas computacionais baseados em restrições [8]. A idéia é definir as restrições (requisitos) que fazem parte do problema em questão e então resolvê-lo encontrando uma solução que satisfaça todas as restrições simultaneamente. Uma das primeiras aplicações que fez uso de restrições foi o sistema Sketchpad [59], desenvolvido por Ivan Sutherland no início da década de sessenta. Este era um sistema interativo para desenho, que permitia ao usuário criar e manipular figuras geométricas na tela do computador a partir de primitivas da linguagem e certas restrições. O sistema baseava-se no conceito de restrições como relações declarativas, garantidas pelo computador. Um sistema subseqüente a este, denominado ThingLab [15], foi desenvolvido já na década de oitenta e incluía algumas facilidades adicionais. Uma delas era a compilação de planos de satisfação de restrições, para poder rapidamente resubmetê-las e assim contemplar modificações dos usuários à medida que este interagia com o sistema.

Porém, as idéias iniciais que mais influenciaram a programação com restrição advém da Inteligência Artificial (IA). Os estudos em IA direcionados a solução dos problemas conhecidos como Problemas de Satisfação de Restrições, produziram algoritmos poderosos para tratar de problemas combinatórios difíceis e deram início a técnicas amplamente utilizadas hoje em dia, conhecidas como Técnicas de Consistência. Essas técnicas foram inicialmente utilizadas, ainda na década de setenta, para melhorar a eficiência de programas de processamento de imagens pelos pesquisadores em IA [42, 67]. Nestes trabalhos, o objetivo era reconhecer os objetos numa cena 3D através da interpretação das linhas (arestas) em desenhos 2D. Para isso, era necessário rotular essas arestas em alguns tipos (convexa, côncava e ocultas), sendo que havia muitas maneiras de fazê-lo (3^n onde n é o número de arestas), porém com apenas uma pequena porção delas com algum significado 3D. A idéia para resolver este problema combinatório era encontrar rótulos válidos para junções de arestas, isto é, rótulos que satisfizessem a restrição de que ambas as arestas envolvidas tivessem a mesma classificação. Isto reduz muito o problema, pois há apenas um número limitado de rótulos válidos satisfazendo este critério.

Os principais algoritmos desenvolvidos nesta época visavam alcançar algum tipo de *arco* ou *caminho-consistência* [42, 38]. O primeiro, encontra e elimina valores dos domínios das variáveis que são incompatíveis com as restrições relacionadas a elas, enquanto o outro elimina pares de valores que são válidos para uma determinada restrição c, porém não são válidos quando analisados num caminho de restrições, que passa por outras restrições e retorna a c (mais de uma restrição é analisada para tentar identificar dependências entre elas e então filtrar mais valores inconsistentes de uma só vez). No entanto, nestes sistemas ainda não havia a noção de "programação" com restrições. O problema era modelado diretamente através de conjuntos de restrições que eram solucionadas por algum algoritmo.

Depois disto, estas técnicas foram aprimoradas por pesquisadores em IA em trabalhos como os efetuados por Steele [56] em 1980 e por Mackworth [39] em 1986. O trabalho realizados por Steele talvez tenha sido o primeiro esforço explícito no sentido de projetar uma linguagem de programação baseada em restrições [60]. Porém, o principal passo em direção a um conceito mais robusto e moderno de programação com restrições foi dado quando notou-se que a programação lógica era apenas um tipo particular de programação com restrição e que os esforços para explorar estas idéias deveriam ser unificados sob uma proposta comum, visando a criação de uma abordagem mais poderosa para programar, modelar e resolver problemas [60, 8]. As principais semelhanças observadas entre as duas tecnologias foram:

- A natureza declarativa da lógica a faz próxima a idéia de restrições e também na lógica definimos "o que" deve ser satisfeito mas não "como".
- Tanto os sistemas baseados em lógica como os que fazem uso de restrições utilizam algum mecanismo de retrocesso(*backtracking*) para resolver problemas.
- Porém, o mais importante, foi a observação de que equações envolvendo termos eram apenas tipos especiais de restrições e que o algoritmo de unificação era apenas um tipo particular de algoritmo de resolução de restrições.

Isto levou à definição de uma classe genérica de linguagens de programação, denominada *Programação Lógica com Restrição* (*Constraint Logic Programming* (CLP)), que possui todas as características da programação lógica mas é parametrizada quanto aos tipos de restrições utilizadas na linguagem. O surgimento deste novo paradigma influenciou principalmente áreas que utilizavam extensivamente a lógica tradicional, pois os pesquisadores nestas áreas perceberam que poderiam alternar para um paradigma mais poderoso e expressivo, movendo da utilização da igualdade entre termos para o uso de restrições [37].

De fato, a CLP combina a declaratividade da lógica com a eficiência da resolução de restrições e assim consegue aperfeiçoar alguns aspectos da programação lógica tradicional [24]. Como vimos, os objetos manipulados por um programa em lógica são estruturas sem significado — o conjunto de todos os possíveis termos que podem ser formados a partir dos símbolos funcionais e constantes. Enquanto cada objeto precisa ser explicitamente codificado em um termo num programa em lógica, as restrições descrevem implicitamente o relacionamento destes objetos, os quais podem variar em domínios computacionais como os inteiros, racionais, reais, dentre outros. Outro problema da lógica está relacionado com a forma como esta realiza a busca em grandes bases de conhecimento, baseada geralmente num procedimento de busca em profundidade com retrocesso que, como vimos, não faz uso de nenhuma informação adicional sobre o problema para evitar caminhos errados.

Cada uma das limitações acima mencionadas são tratadas por uma noção diferente da utilização de restrições, noções estas que foram combinadas para criar a atual concepção de linguagens CLP. Primeiramente, a CLP descende de trabalhos que visavam a introdução de estruturas semânticas mais ricas para os sistemas lógicos, a fim de torná-los aptos a manipular, por exemplo, expressões aritméticas lineares. A idéia neste caso é substituir a principal operação da linguagem, a unificação, por um procedimento de resolução de restrições. Este esquema, chamado CLP(X), foi proposto por Jaffar e Lassez [34]. A CLP também foi fortemente influenciada pelos trabalhos relacionados com as Técnicas de Consistência desenvolvidos inicialmente na IA. Com o objetivo de melhorar a performance da busca em sistemas lógicos, estas técnicas foram utilizadas na programação lógica empregando-se, de maneira prioritária, o uso ativo de restrições para podar o espaço de busca, em contraste com a forma passiva que até então vinham sendo utilizadas. Isso levou a uma estreita integração entre um processo determinístico, resolução de restrições, com um processo não determinístico, a busca, pois estes passaram a ser utilizados em conjunto na resolução de problemas.

Resumindo, a CLP atualmente engloba duas tecnologias distintas:

- Técnicas de Consistência: envolve a solução de problemas definidos sobre os domínios finitos, denominados de Problemas de Satisfação de Restrições, cujas técnicas de solução estão fortemente baseadas nas chamadas Técnicas de Consistência. Os problemas que se encaixam nesta categoria podem ser problemas combinatórios muito difíceis (NP-difícil) e as técnicas para resolvê-los estão principalmente destinadas a melhoria da performance de algoritmos de busca, sendo que muitas delas foram incorporadas às implementações de linguagens CLP.

- Resolução de Restrição: inspira-se nas bases da Programação com Restrição, isto é, descrever um problema como um conjunto de restrições e então criar métodos para resolvê-las. As restrições aqui podem estar definidas sobre domínios infinitos, como os números inteiros ou naturais, ou mesmo domínios mais complexos. Ao invés de utilizar métodos combinatórios, os algoritmos para resolução de restrições são baseados em técnicas matemáticas como diferenciação automática, séries de Taylor, método de Newton, dentre outras. O esquema CLP(X) define as bases para esta abordagem.

O que emergiu com a CLP foi um paradigma de programação genérico e declarativo potencialmente mais promissor do que a lógica de primeira ordem completa, que é bastante expressiva mas ineficiente na prática, ou do que versões restringidas desta lógica, como subconjuntos de cláusulas de Horn, que geralmente são mais eficientes na prática porém não são suficientemente expressivas para muitas aplicações. É importante neste ponto destacarmos que, apesar da possibilidade de serem criadas linguagens CLP baseadas em diferentes lógicas, os trabalhos realizados estão quase que exclusivamente dedicados a introdução de métodos de resolução de restrições em linguagens lógicas baseadas no Prolog.

A programação lógica com restrições contribuiu para pesquisas em novas direções em várias áreas distintas, tais como: inteligência artificial (compreensão de linguagem natural, escalonamento, planejamento, configuração, etc), computação concorrente, sistemas de banco de dados, interfaces gráficas, verificação de hardware, estudo de operações e otimização combinatória, linguagens de programação, sistemas reativos, sistemas e algoritmos para computação simbólica, dentre outras.

3.3 O Esquema CLP

Nesta seção, descreveremos as bases do Esquema de Programação Lógica com Restrições, chamado CLP(X) [34]. O esquema CLP(X) representa uma classe de linguagens de programação baseadas na programação lógica e resolução de restrições. Cada vez que X é instanciado com o domínio de discurso pretendido, como o domínio dos números reais ou booleano, obtém-se uma linguagem de programação apta a tratar de muitas operações naturais ao domínio em questão.

Os domínios podem variar de acordo com as características e necessidades das aplicações que fazem uso das restrições. Os mais utilizados e por sua vez mais pesquisados quanto às técnicas de solução a serem empregadas para resolvê-los são: domínio das restrições booleanas, domínio das restrições aritméticas lineares e não lineares, domínios finitos e restrições globais. Quando uma aplicação necessita de uma restrição específica ainda não disponível, propostas foram feitas para permitir ao próprio usuário adicioná-la ao sistema (Seção 3.5).

O principal objetivo do esquema CLP(X) é prover ao usuário mais expressividade e flexibilidade no que tange aos objetos primitivos que a linguagem pode manipular. O usuário quer projetar sua aplicação utilizando conceitos que são o mais próximo possível do seu domínio de discurso. Por exemplo, ele quer usar conjuntos, expressões booleanas, inteiros, racionais ou reais, ao invés de ter de codificar tudo como estruturas sem significado como é feito na programação lógica tradicional. A programação lógica com restrições admite computação diretamente sobre o domínio de discurso pretendido, pois introduz símbolos especiais (restrições) cuja interpretação é fixa e que possibilitam modelar as relações existentes no domínio em questão.

Quando restrições são introduzidas na programação lógica, um mecanismo para solucioná-las também deve ser introduzido. Na programação lógica tradicional a única restrição disponível é a igualdade (sintática) entre termos e o algoritmo de unificação é utilizado para resolvê-la. Uma vez reconhecido que o algoritmo de unificação é, senão, apenas um tipo específico de procedimento de resolução de restrição (*constraint solver*), foi natural sua substituição por mecanismos mais genéricos para possibilitar a utilização de outros tipos de restrições na programação lógica.

3.3.1 Sintaxe e Semântica

Apresentaremos nesta seção a sintaxe e semântica da CLP(X), assim como definido principalmente em [34] e [35]. Começaremos revisando a definição de termo e sentença atômica para lógica de primeira ordem,

Termo ⇒ Função(Termo,...) | Variável | Constante
Sentença Atômica ⇒ Predicado(Termo,...) | Termo = Termo

e seguiremos para definição de alguns conceitos e terminologias.

- Uma *assinatura* define um conjunto de símbolos representando funções e predicados e associa uma aridade[1] para cada um.
- Se Σ é uma assinatura, uma Σ-*estrutura* D consiste de um conjunto D e uma designação de funções e relações sobre o conteúdo de D para os símbolos de Σ respeitando-se as aridades dos símbolos. Em outras palavras, D representa a interpretação a ser dada aos símbolos de Σ.
- Uma Σ-*fórmula* de primeira-ordem é criada a partir de constantes, variáveis, funções e predicados de Σ, os conectivos lógicos $\wedge, \vee, \longrightarrow, \longleftarrow, \longleftrightarrow, \neg$ e os quantificadores $\exists$ e $\forall$ sobre variáveis.
- Uma fórmula *fechada* é aquela onde a ocorrência de todas as suas variáveis está sobre o escopo de um quantificador.
- Uma Σ-*teoria* é uma coleção de Σ-fórmulas fechadas.

Uma *restrição primitiva* tem a forma $p(t_1, ..., t_n)$, onde $t_1, ..., t_n$ são termos e p $\in \Sigma$ é um símbolo predicado. Toda *restrição* é uma fórmula de primeira-ordem construída a partir de restrições primitivas.

Um programa em CLP é sintaticamente uma coleção de cláusulas que podem ser regras ou fatos [24]. Fatos são átomos (sentenças atômicas) que expressam relacionamentos conhecidos e são formados apenas por símbolos predicados diferentes daqueles encontrados em Σ. Regras são geralmente como no Prolog, com a diferença de que podem conter restrições em suas premissas. A premissa de uma regra (corpo) é formada pela conjunção de restrições e átomos e a conclusão (cabeça) apenas por um único átomo. Utilizaremos a notação empregada no Prolog de escrever Q:- P ao invés de $P \Rightarrow Q$, sendo assim, Q corresponde à conclusão e P à premissa. Cada regra em CLP possui a seguinte forma:

$$p_0(t_0) : -c_1(u_1), ..., c_n(u_n), p_1(t_1), ..., p_m(t_m).$$

onde p_i, $0 \leq i \leq m$ são símbolos predicados sem interpretação pré-definida; c_j $1 \leq j \leq n$, são as restrições da regra e t_k e u_l, $0 \leq k \leq m, 1 \leq l \leq n$, são termos. Um *objetivo* (ou consulta) tem a mesma forma que o corpo de uma regra, isto é, é uma conjunção de restrições e átomos.

Dadas as definições acima, podemos dizer que o parâmetro X pode ser compreendido como uma 4-tupla (Σ, D, L, T). Σ é uma assinatura que define os predicados e funções pré-definidas e suas aridades, D é uma Σ-estrutura sobre a qual a computação será realizada, L é uma classe de Σ-fórmulas que representa a classe de restrições que podem ser expressas e T é uma Σ-teoria de primeira-ordem. Chamaremos o par (D, L) de um *domínio de restrição*. Para ficar mais clara a definição de um domínio X, vamos apresentar alguns exemplos de domínios de restrição:

- Domínio aritmético: seja Σ composto pelas constantes 0 e 1, as funções binárias $+$, $-$ e $*$, e os predicados binários $=, <$ e $\leqslant$. Seja D o conjunto dos números reais e D a interpretação usual dos símbolos de Σ (isto é, $+$ é interpretado como adição, etc.). Seja L as restrições geradas a partir das restrições primitivas. Então, $\Re = (D, L)$ é o domínio de restrição aritmético sobre os números reais. Se omitirmos de Σ o símbolo $*$, então o domínio de restrição correspondente $\Re_{Lin} = (D', L')$, representa o domínio de restrição aritmético linear sobre os números reais. Se o domínio for novamente restringido para os números racionais, teremos o domínio Q_{Lin}. Nas restrições em $\Re_{Lin}$ e Q_{Lin} podemos escrever termos como 3 ou 5x. A restrição $\exists\ y\ 5x + y \leqslant 3 \wedge z \leqslant y - 1$

[1] A aridade define o número de variáveis que o símbolo possui.

é válida em $\Re$, $\Re_{Lin}$ e Q_{Lin}, enquanto a restrição $x * x \leqslant y$ é válida apenas em $\Re$. Se estendermos L' para permitir equações negadas (isto é, equações com o símbolo $\neq$) então os domínios de restrição resultantes $\Re^{\neq}_{Lin}$ e $Q^{\neq}_{Lin}$ permitem restrições como $2x + y \leqslant 0 \wedge x \neq y$. Finalmente, se restringirmos Σ para 0, 1, +, =, obtemos o domínio de restrição $\Re_{LinEqn}$, onde as únicas restrições são equações lineares.

- Domínio de árvores finitas: o Prolog pode ser visto como uma linguagem de programação lógica com restrição onde as restrições são equações sobre a álgebra de termos (também chamada de álgebra de árvores finitas, ou domínio de Herbrand). Seja Σ uma coleção de constantes e símbolos funcionais e o predicado binário =. Seja D o conjunto de árvores finitas onde: cada nó da árvore é identificado por uma constante ou função, o número de filhos de cada nó é a aridade dos símbolos de função e os nós filhos são ordenados. A interpretação D das função de Σ as define como construtores de árvores, onde cada $f \in \Sigma$ de aridade n mapeia n árvores para uma árvore cujo nó raiz é identificado por f e cujas sub-árvores são os argumentos do mapeamento. As restrições primitivas são equações entre termos, e L as restrições geradas por estas restrições primitivas. Então, $FT = (D, L)$ é o domínio de restrição de Herbrand como usado no Prolog. Restrição típicas são $x = g(y)$ e $\exists\ zx = f(z, z) \wedge y = g(z)$.

A semântica declarativa e informal é preservada, permanecendo essencialmente a mesma da dos programas lógicos tradicionais. Na verdade, a interpretação operacional das restrições, e não a declarativa, é que distingue a CLP da programação lógica. A semântica operacional é uma generalização da semântica da programação lógica e a apresentaremos como sendo um *sistema de transição de estados*. Cada estado é representando por uma tupla $\langle A, C, S\rangle$ onde A é um conjunto de átomos e restrições e C e S são conjuntos de restrições. As restrições C e S formam o denominado *repositório de restrições*, o qual é utilizado durante um processamento pelo procedimento de resolução. O conjunto A representa uma coleção de átomos e restrições ainda não considerados num determinado processamento, C representa a coleção de restrições ativas e S o conjunto de restrições passivas do repositório. De modo geral, as restrições são consideradas ativas se estão aptas a serem utilizadas de forma útil e consideradas passivas caso contrário. Um estado pode ser ainda um estado de falha.

Vamos assumir a existência de uma *regra de computação*, a qual tem a função de selecionar um tipo de transição e um elemento apropriado de A (se necessário) para cada estado. O sistema de transição também conta com um método para detectar a satisfatibilidade ou consistência de um conjunto de restrições e de outro para realizar inferências. O primeiro, o qual iremos denominar Consistente(C), expressa um teste de consistência em C e geralmente é definido por: Consistente(C) $\Leftrightarrow D \vDash C$. Isto quer dizer que o método retorna que o conjunto C é consistente se e somente se todas as restrições contidas em C são verdadeiras seguindo a interpretação definida por D. Dizemos neste caso que o método é completo, pois consegue decidir sobre a satisfatibilidade de qualquer conjunto de restrições geradas a partir de D. Entretanto, os sistemas podem empregar em determinadas situações um teste parcial ou incompleto de consistência, isto é: se $D \vDash C$ então Consistente(C) retorna verdadeiro, mas algumas vezes Consistente(C) retorna verdadeiro apesar de $D \vDash \neg C$. O método para realizar inferências, denominado Infere(C, S), tem por objetivo computar, a partir dos conjuntos de restrições ativas e passivas existentes, novos conjuntos C' e S'.

As transições no sistema de transições são as seguintes:

$$\langle A \cup a, C, S\rangle \rightarrow_r \langle A \cup B, C, S \cup (a = h)\rangle \tag{3.1}$$

se a é selecionado pela regra de computação, a é um átomo, $h \leftarrow B$ é uma regra do programa, com suas variáveis renomeadas para evitar conflitos, e a e h representam o mesmo símbolo predicado. A expressão $a = h$ é uma abreviação para a conjunção de equações que serão

criadas para relacionar os argumentos correspondentes entre a e h. Dizemos que a é *reescrito* nesta transição.

$$\langle A \cup a, C, S\rangle \rightarrow_c \text{falha} \tag{3.2}$$

se a é selecionado pela regra de computação, a é um átomo e, para toda regra $h \leftarrow B$ do programa, h e a tem diferentes símbolos predicados.

$$\langle A \cup c, C, S\rangle \rightarrow_c \langle A, C, S \cup c\rangle \tag{3.3}$$

se c é selecionado pela regra de computação e c é uma restrição.

$$\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle \tag{3.4}$$

se (C',S')=Infere(C,S).

$$\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle \tag{3.5}$$

se Consistente(C).

$$\langle A, C, S\rangle \rightarrow_s \text{falha} \tag{3.6}$$

se $\neg$ Consistente(C).

A transição $\rightarrow_r$ recebe este nome em referência ao procedimento de resolução no qual a transição se baseia, transições $\rightarrow_c$ introduzem restrições no procedimento de resolução de restrições, transições $\rightarrow_s$ testam se as restrições ativas são consistentes e transições $\rightarrow_i$ inferem mais restrições ativas (e talvez modificam as restrições passivas) a partir da coleção corrente de restrições.

A implementação do método Infere(C, S) varia muito de sistema para sistema, porém, sempre é requerido que $D \vDash (C \wedge S) \leftrightarrow (C', S')$ de tal forma que a informação nunca seja perdida ou "adivinhada". No Prolog não há restrições passivas e podemos definir Infere(C, S) = ($C \cup S, \emptyset$). Em sistemas que implementam o domínio das restrições aritméticas lineares, as restrições não lineares são consideradas passivas e uma possível implementação para o método pode ser simplesmente passar uma restrição de S para C' quando esta se torna linear no contexto de C e então removê-la do conjunto S. Por exemplo, se S é $x * y = z \wedge z * y = 2$ e C é $x = 4 \wedge z \leq 0$ então Infere(C, S) = (C', S') onde C' é $x = 4 \wedge z \leq 0 \wedge 4y = z$ e S' é $z * y = 2$. Um outro exemplo, agora sobre domínios finitos, e que realiza uma inferência mais complexa que a anterior, é dado por: se S é $x = y + 1$ e C é $2 \leqslant x \leqslant 5 \wedge 0 \leqslant y \leqslant 3$ então Infere(C,S) = (C', S') onde C' é $2 \leqslant x \leqslant 4 \wedge 1 \leqslant y \leqslant 3$ e $S' = S$.

Geralmente as restrições ativas são determinadas sintaticamente. No Prolog todas as equações são ativas, em sistemas que empregam o domínio aritmético linear todas as restrições lineares são ativas e em sistemas com domínios finitos todas as restrições unárias, isto é, restrições em apenas uma variável como $x < 9$ ou $x \neq 0$, são consideradas restrições ativas.

Quanto mais informação a coleção de restrições ativas representar, mais cedo uma falha poderá ser detectada e, conseqüentemente, menos busca será necessário. Sendo assim, poderíamos desejar que Infere(C, S) realizasse inferências as mais poderosas possível, de tal sorte que: para cada restrição ativa c, se Infere(C, S) = (C', S') e $D \vDash (C \wedge S) \rightarrow c$, então $D \vDash C' \rightarrow c$. Em outras palavras, a rotina de inferência deveria sempre gerar o maior número de restrições ativas possível a partir de C e S. Entretanto, isto nem sempre é possível e, mesmo quando possível, geralmente não é preferido, tendo em vista o custo computacional poder ser maior que a melhoria obtida com a limitação da busca.

Um sistema CLP é determinado pelo domínio de restrição e por uma semântica operacional detalhada, a qual envolve uma regra de computação e definições para os procedimentos Consistente(C) e Infere(C,S). Iremos agora definir algumas propriedades para sistemas CLP, tomando como princípio (como em [34]) que a classe de sistemas considerados são aqueles onde não há restrições passivas e o teste de consistência é completo.

Seja $\rightarrow_{ris}=\rightarrow_r\rightarrow_i\rightarrow_s$ e $\rightarrow_{cis}=\rightarrow_c\rightarrow_i\rightarrow_s$. Dizemos que um sistema CLP é *rapidamente-verificável* se sua semântica operacional pode ser descrita por $\rightarrow_{ris}$ e $\rightarrow_{cis}$. Um sistema CLP é dito *progressivo* se, para todo estado com uma coleção não vazia de átomos, toda derivação a partir deste estado ou resulta em falha ou contém uma transição $\rightarrow_r$ ou contém uma transição $\rightarrow_c$. Um sistema CLP é *ideal* se for rapidamente-verificável, progressivo, Infere(C,S) = ($C \cup S$,$\emptyset$) (isto é, não há restrições passivas) e Consistente(C) é completo.

Num sistema rapidamente-verificável a inferência de um novo conjunto de restrições ativas é feita e um teste de consistência executado cada vez que a coleção de restrições é modificada no mecanismo de resolução. Assim, dadas as definições acima para Consistente e Infere, o sistema irá detectar inconsistências o mais breve possível. Um sistema progressivo nunca irá ignorar infinitamente a coleção de átomos e restrições em A de um estado durante uma execução, isto é, o sistema garantidamente chegará ao término de um processamento. A grande maioria dos sistemas CLP implementados são rapidamente-verificáveis e progressivos, mas muitos não são ideais.

Uma *derivação* é uma seqüência de transições $\langle A_1, C_1, S_1\rangle \rightarrow ... \rightarrow \langle A_i, C_i, S_i\rangle \rightarrow ... \rightarrow \langle A_n, C_n, S_n\rangle$. Um estado a partir do qual não se pode realizar mais nenhuma derivação é dito ser um estado final. Uma derivação conclui com *sucesso* se ela é finita e o estado final tem a forma $\langle\emptyset, C, S\rangle$. Seja G um objetivo com um conjunto de variáveis X e cuja derivação a partir dele produz um estado final $\langle\emptyset, C, S\rangle$. Então, o conjunto de variáveis X mais $C \wedge S$ formam a resposta das derivações.

Uma derivação falha se é finita e o estado final é um estado de falha. Uma derivação é *satisfatória* se falha ou se para todo i e todo $a \in A_i$, a é reescrito posteriormente numa transição. Uma regra de computação é satisfatória se produz apenas derivações satisfatórias. Um objetivo G é *finitamente falho* se, para qualquer regra de computação satisfatória, toda derivação a partir de G num sistema CLP ideal é falha.

Uma *árvore de derivação* de um objetivo G para um programa P é uma árvore com os nós representando os estados e as arestas representando as transições $\rightarrow_r, \rightarrow_c, \rightarrow_i$ *ou* $\rightarrow_s$ tal que: a raiz é identificada por $\langle G, \emptyset, \emptyset\rangle$; todo nó que possuir mais de uma aresta de saída, todas elas devem representar a mesma transição; se um nó identificado por um estado S tem uma aresta de saída representando $\rightarrow_c, \rightarrow_i$ *ou* $\rightarrow_s$, então o nó tem exatamente um filho e o estado representando o nó filho será obtido a partir de S utilizando a transição correspondente; se um nó identificado por um estado S tem uma aresta de saída representando a transição $\rightarrow_r$, então o nó tem um filho para cada regra em P, e o estado representando cada um dos filhos será obtido a partir de S através da transição $\rightarrow_r$ para a regra em questão; para cada aresta $\rightarrow_r$ ou $\rightarrow_c$, a transição correspondente utiliza o átomo ou restrição selecionado pela regra de computação.

Cada ramo de uma árvore de derivação corresponde a uma derivação e, dada uma regra de computação, toda derivação seguindo esta regra é um ramo da árvore de derivação correspondente. Regras de computação diferentes podem gerar árvores de derivação de tamanhos diferentes e um dos objetivos das transições $\rightarrow_i$ é extrair o máximo de informação possível das restrições passivas de forma que a ramificação das transições $\rightarrow_r$ seja reduzida (pois inconsistências são detectadas mais cedo). A maioria das linguagens CLP existentes utilizam regras de computação baseadas no Prolog, sendo assim, a ordem de escolha dos átomos ou restrições através das conjunções no corpo de uma cláusula é estritamente da *esquerda para direita*. Isto facilita a programação da regra, pois prove um fluxo de controle previsível. Porém, quando um fluxo de controle é dependente dos dados ou muito complexo (como em problemas combinatórios), maior flexibilidade é requerida. A maioria das propostas para melhorar este mecanismo foram originadas na programação lógica tradicional, mas as idéias podem ser movidas para programação lógica com restrições, sendo que existem hoje algumas propostas baseadas nestas idéias [4, 62, 63].

O problema de encontrar respostas para uma consulta pode ser visto como um problema de busca numa árvore de derivação (ou, neste caso, árvore de busca). Muitas das linguagens CLP empregam a *busca em profundidade* com o método de retrocesso cronológico (*chronological backtracking* [9]), apesar de existirem sugestões para se utilizar uma outra forma do método de retrocesso conhecida como retrocesso dirigido por dependência (*dependency-directed backtracking*) [6]. Perceba como as transições aqui definidas levam a um processo de busca mais eficiente do que o empregado na programação lógica tradicional, pois a transição $\rightarrow_s$ é sempre executada (em sistemas CLP rapidamente-verificáveis) após alguma inferência ser executada. Na Seção 3.4.3 veremos ainda outras maneiras de como aperfeiçoar o processo de busca, através de mecanismos incorporados às linguagens CLP que permitem ao usuário configurar a estratégia de busca que melhor se adeqüe às características do seu problema.

Na seção seguinte, apresentamos a semântica operacional do esquema, exemplificando seu processo de funcionamento em termos de um dos mais importantes domínios existentes, o domínio das expressões aritméticas.

3.3.2 Restrições Aritméticas Lineares

Prover aritmética foi uma das motivações para a pesquisa em combinar programação lógica com restrições [24]. Apesar do Prolog possuir facilidades para avaliação de expressões aritméticas, seu comportamento não é idealmente o que alguém espera. Ele não pode manipular, por exemplo, equações como $x - 3 = y + 5$ pois ele não associa significado algum aos sinais $+$ e $-$ numa equação como esta.

Expressões aritméticas lineares são termos compostos de números, variáveis e dos operadores aritméticos usuais: negação ($-$), adição ($+$), subtração ($-$), multiplicação ($*$) e divisão ($/$). Para a condição de linearidade ser satisfeita é preciso que em uma multiplicação no máximo um dos componentes seja uma variável e que em uma divisão o denominador seja um número. Uma restrição aritmética é uma expressão da forma t_1 R t_2 onde R é um dos seguintes predicados $>, >=, =, <=, <, <>$.

Existem vários procedimentos para resolução de um sistema de restrições aritméticas lineares. Geralmente, uma combinação de eliminação Gauseana e um algoritmo *Simplex* modificado são empregados. O algoritmo *Simplex* é um dos mais utilizados pois possui um bom tempo médio de execução, é bem compreendido e pode ser utilizado de forma incremental [24].

Utilizaremos o exemplo abaixo para exemplificarmos o mecanismo de execução das linguagens CLP como foi formalmente definido na seção anterior. A idéia é que, dadas duas partículas com movimentos retilíneos que obedecem às equações $x + y - 2 = 0$ e $x - y = 0$, seja possível, dado um valor para x, calcular o valor de y de ambas as equações e verificar se a posição resultante não implica em colisão das partículas. O programa em CLP para resolvê-lo é apresentado abaixo:

```
Trajeto(X,Y1,Y2):-
   Posição(X),Y1 = 2 - X,Y2 = X,X + Y1 - 2 ≠ X - Y2.
Posição(2).
Posição(1).
Posição(3).
```

Consideremos a seguinte consulta: `Trajeto(X, Y1, Y2)`, que pergunta sobre todas as posições válidas existentes na base de conhecimento. Na representação de um estado de computação, utilizaremos o símbolo "$\diamond$" para separar os objetivos restantes (conjunto A) do repositório (conjuntos C e S) e não faremos distinção entre restrições ativas e passivas. O estado de computação inicial corresponde então a:

◇ Trajeto(X, Y1, Y2).

No primeiro passo de computação, um átomo na consulta que unifique com a consequência de alguma regra na base será escolhido. Neste caso só há uma possibilidade, isto é, Trajeto(X, Y1, Y2) unificar com a regra Trajeto(X, Y1, Y2). O próximo passo consiste em adicionar na lista de objetivos restantes os átomos constantes na premissa da regra e adicionar no repositório o conjunto de restrições contidas na premissa da regra juntamente com as equações criadas para relacionar os argumentos dos predicados que unificaram fazendo a renomeação dos mesmos. Sendo assim, a primeira derivação produz o seguinte estado de computação:

Y11=2 − X1, Y21=X1, X1 + Y11 − 2 ≠ X − Y2, X=X1, Y1=Y11, Y2=Y21 ◇ Posição(X1).[2]

Um sistema CLP procura por todas as soluções tentando sistematicamente todas as possíveis regras e fatos para a redução dos átomos restantes. O processo descrito acima continua até que a derivação chegue num estado de computação terminal. O estado seguinte é:

Y11=2 - 2, Y21=2, 2 + Y11 − 2 ≠ 2 − Y21, X=X1, Y1=Y11, Y2=Y21, X1=2 ◇.

Como estamos usando uma regra de computação como a do Prolog, o próximo átomo escolhido foi o primeiro da esquerda para direita, isto é, Posição(X1), que unifica com o fato Posição(2) causando a atribuição X1=2. Como as informações produzidas são suficientes para resolverem todas as equações este é um estado terminal e como não há inconsistências no repositório, já que 2 + Y11 − 2 ≠ 2 − Y21 é verdadeiro para os valores obtidos de Y11 e Y21, a derivação ocorreu com sucesso.

A busca por todas as posições válidas para as partículas continua fazendo um retrocesso ao estado onde restam outros caminhos a serem avaliados, ou seja, para a configuração onde existe a possibilidade de unificação para o átomo Posição(X1):

Y11=2 − X1, Y21= X1, X1 + Y11 − 2 ≠ X1 − Y21, X=X1, Y1=Y11, Y2=Y21 ◇ Posição(X).

A próxima escolha é a unificação com Posição(1), que leva ao surgimento da equação X1=1 e ao seguinte repositório:

Y11=2 − 1, Y21 = 1, 1 + Y11 − 2 ≠ 1 − Y21, X=X1, Y1=Y11, Y2=Y21, X1=1 ◇.

Segundo as transições 3.2 e 3.6 podemos dizer que uma derivação termina sem sucesso quando: nenhuma cláusula pôde ser escolhida para unificar aos átomos restantes e produzir um novo estado de computação ou o repositório, após algum passo de computação, tornou-se inconsistente. Este é um estado terminal que representa uma derivação sem sucesso, pois a conjunção de restrições Y11=2 − 1, Y21=1, 1 + Y11 − 2 ≠ 1 − Y21, X=X1, Y1=Y11, Y2=Y21, X1=1 não é satisfazível já que 0 ≠ 0 não é verdadeiro.

As restrições aritméticas lineares são suficientes para uma vasta gama de aplicações. Porém, com o surgimento de novas áreas, surgiu também a necessidade de mecanismos de resolução de restrições não lineares. As primeiras aplicações a utilizarem tais restrições foram aplicações em computação geométrica [49] e economia. Na seção seguinte, fazemos uma breve descrição deste domínio e de seus métodos de solução.

[2]Equações triviais como as que mostram a renomeação de variáveis poderiam ser omitidas, mas não o faremos em prol da clareza do exemplo.

3.3.3 Restrições Aritméticas Não Lineares

Expressões aritméticas não lineares são termos compostos de números, variáveis e dos operadores aritméticos usuais: negação ($-$), adição ($+$), subtração ($-$), multiplicação ($*$) e divisão ($/$). Uma expressão é não linear quando possui pelo menos uma multiplicação entre duas variáveis ou uma divisão onde o denominador seja uma variável. Uma restrição aritmética é uma expressão da forma t_1 R t_2 onde R é um dos seguintes predicados $>, >=, =, <=, <, <>$. Para exemplificarmos restrições aritméticas não lineares, vejamos o programa abaixo, o qual descreve algumas propriedades elementares da análise de circuitos:

```
complexmult(c(R1,I1), c(R2,I2), c(R3,I3)):-
   R3 = R1 * R2 - I1 * I2,
   I3 = R1 * I2 + R2 * I1.
ohmlaw(V, I, R):-
   complexmult(I, R, V).
inductorlaw(I, V, L, W):-
   complexmult(c(0,W*L), I, V).
capacitorlaw(I, V, C, W):-
   complexmult(c(0,W*C), V, I).
```

Se o objetivo `complexmult(c(1,2),c(3,4),c(R3,I3))` é submetido, então as equações não lineares tornam-se lineares em tempo de execução e a resposta produzida é:

$$R3 = -5$$
$$I3 = 10$$

Se o objetivo dado for `complexmult(c(1,2), c(R2,I2), c(R3,I3))`, a solução para `R2` e `I2` é uma conjunção de duas igualdades lineares:

$$I2 = 0.2 * I3 - 0.4 * R3$$
$$R2 = 0.4 * I3 + 0.2 * R3$$

Esta resposta é um exemplo de *solução não definida*. A solução é um conjunto infinito de pontos que é representado por um conjunto minimal de restrições relatando relações entre as variáveis da consulta. Para obter valores precisos para `I2` e `R2` (isto é, `I2` igual a uma constante e `R2` igual a uma constante), o usuário terá que instanciar `I3` e `R3`.

Para as duas consultas acima não houve necessidade de um mecanismo de resolução não linear, pois, durante a execução, as variáveis foram sendo instanciadas e tornaram as restrições não lineares em restrições lineares. Isto é o que muitos sistemas CLP adotam para tratar de restrições não lineares, quando uma restrição desse tipo é encontrada durante a computação, o sistema atrasa sua execução na esperança de que variáveis sejam instanciadas e ela se torne linear. Todavia, para a consulta `complexmult(c(R1,2),c(R2,4),c(-5,10))` $\wedge$ `R2 < 3` isto não ocorre e um possível estado de computação final para sistemas que não solucionam restrições não lineares seria:

$$R1 = -0.5 * R2 + 2.5$$
$$3 = R1 * R2$$
$$R2 < 3$$

Duas equações não lineares são encontradas durante a computação. Suas execuções são postergadas mas as variáveis envolvidas não são instanciadas a ponto de torná-las lineares.

Vários algoritmos podem ser usados para resolver restrições não lineares e suas capacidades e complexidades variam muito. Uma boa comparação entre eles pode ser encontrada

em [41]. Em um sistema com mecanismos para solucionar restrições não lineares, a consulta `complexmult(c(R1,2),c(R2,4),c(-5,10))` $\wedge$ `R2` < 3 pode ser completamente resolvida, retornando a resposta:

$$R1 = 1.5$$
$$R2 = 2$$

O domínio aritmético é somente um dos inúmeros domínios para o qual mecanismos de solução foram desenvolvidos. Uma descrição de outros domínios interessantes pode ser encontrada na Seção 3.6.

3.4 Técnicas de Consistência

Técnicas de Consistência foram primeiramente introduzidas para melhorar a eficiência de programas de reconhecimento de imagens pelos pesquisadores em inteligência artificial [67] e desde então têm produzido uma longa linha de algoritmos [61].

A manipulação de restrições utilizando-se técnicas de consistência é diferente do mecanismo de resolução de restrição visto no esquema CLP(X). Estas técnicas não solucionam as restrições através de métodos matemáticos, ao invés disso, elas provêm um meio eficiente de extrair novas informações sobre as variáveis do problema e têm provado serem muito eficientes numa variedade de problemas que envolvem busca, em especial, para solucionar uma vasta classe de problemas denominados Problemas de Satisfação de Restrições ou *Constraint Satisfaction Problems* (CSP).

As técnicas de consistência operam através da *propagação* de informação sobre as variáveis através das restrições existentes entre elas. Por exemplo, consideremos a restrição $x = y + 1$, onde x e y são variáveis. Qualquer atribuição para a variável y, por exemplo $y = 5$, causará uma atribuição para x ($x = 6$). Além disso, a mesma restrição também produz resultados na outra direção, isto é, qualquer atribuição para x, por exemplo $x = 3$, causará uma atribuição para y, neste caso $y = 2$. Numa aplicação gráfica, a propagação de restrições pode ser usada para manter restrições entre objetos gráficos quando estes se movem. Por exemplo, se um objeto é restringido a aparecer no topo de outro objeto e o usuário move um dos objetos, o outro irá mover-se com ele como resultado da propagação das restrições.

3.4.1 Problemas de Satisfação de Restrições

Muitos problemas podem ser formulados como Problemas de Satisfação de Restrições apesar que, geralmente, quem não está familiarizado com eles falha em reconhecê-los e assim deixa de usar as técnicas desenvolvidas especialmente para resolvê-los. Problemas de Satisfação de Restrições têm inspirado uma variedade de pesquisas pois, a despeito de sua forma simples, estes podem ser usados para expressar problemas reais muito difíceis.

O formato padrão de um problema CSP consiste de:

- Um conjunto de variáveis V_i.
- Um domínio D_i para cada variável. O domínio define o conjunto de valores possíveis que cada variável pode assumir e pode ser discreto (finito) ou contínuo (infinito).
- Um conjunto de restrições sobre os valores que as variáveis podem simultaneamente assumirem.

Num CSP, os estados da busca são definidos pelos valores do conjunto de variáveis. Por exemplo, o problema das 8-rainhas[3] pode ser visto como um problema CSP no qual as variáveis representam as localizações de cada uma das rainhas, os valores possíveis são os quadrados do tabuleiro e as restrições definem que duas rainhas não podem estar na mesma linha, coluna ou diagonal. Uma solução deve especificar valores para todas as variáveis de tal forma que todas as restrições sejam satisfeitas simultâneamente. Problemas CSP podem ser resolvidos por algoritmos de busca de propósito geral, porém, devido sua estrutura especial, algoritmos projetados especificamente para eles geralmente possuem uma performance muito superior.

Vamos inicialmente considerar como resolver um problema CSP através de algoritmos de busca genéricos. O estado inicial será o estado onde todas as variáveis estão sem valor. O teste para verificar se o estado atual é o estado objetivo deverá verificar se todas as variáveis estão instanciadas e todas as restrições satisfeitas. A profundidade máxima da árvore de busca é fixa em n (o número de variáveis) e todas as soluções estão na profundidade n, o que nos permite usar com segurança a busca em profundidade (pois não há risco da busca descer muito fundo e também não há necessidade da definição de um limite arbitrário).

Na implementação mais simples, qualquer variável sem valor num determinado estado recebe a atribuição de algum valor do seu domínio. Esta abordagem é conhecida como *generate-and-test* [9] e consiste em atribuir sistematicamente cada possível combinação de valor para as variáveis, testando, sempre que todas as variáveis estão instanciadas, se todas as restrições sobre elas estão sendo satisfeitas. A primeira combinação que satisfaz todas as restrições é a solução. O número de combinações consideradas por este método corresponde ao produto cartesiano do número de elementos no domínio de cada uma das variáveis, neste caso 8^8.

Uma abordagem mais eficiente é tirar vantagem do fato de que a ordem de atribuição para as variáveis não faz diferença para o resultado final. Dessa forma, a maioria dos algoritmos para CSP geram estados sucessores escolhendo o valor de apenas uma única variável em cada nó, numa abordagem conhecida como retrocesso ou *backtracking* [9]. Dessa forma, no problema das 8-rainhas por exemplo, no estado inicial um quadrado é atribuído apenas para a primeira rainha, no estado seguinte atribuí-se outro para a segunda e assim por diante. Uma melhoria imediata que esta abordagem possibilita é a introdução de um teste antes de gerar estados sucessores para checar se alguma restrição foi violada até aquele ponto. Se detectado que sim, o algoritmo pode retroceder e continuar a busca em outro ramo, eliminando rapidamente toda uma porção da árvore que não precisa ser explorada. Todavia, para alguns problemas, é possível ainda tirar vantagem de sua estrutura para construir heurísticas que podem resultar em ganhos de performance significativos. As linguagens CLP fazem uso das restrições definidas sobre as variáveis do problema para criar um processo de busca eficiente como o apresentado acima. Além disso, como veremos na Seção 3.4.3, a CLP também permite em suas linguagens a especificação de heurísticas para uma otimização ainda maior.

Apesar de mais eficiente, o algoritmo de retrocesso como apresentado ainda pode ser melhorado. Suponhamos que as posições escolhidas para as seis primeiras rainhas torna impossível colocar a oitava, tendo em vista que elas atacam todos os oito quadrados na última coluna. O retrocesso irá testar todas as possibilidades para posicionar a sétima rainha, mesmo já estando o tabuleiro numa configuração incorreta. Isso ocorre porque o algoritmo básico de retrocesso não é capaz de detectar conflitos antes que estes realmente ocorram, isto é, antes que todas as variáveis envolvendo uma restrição tenham sido instanciadas. As linguagens CLP podem resolver este e outros tipos de problemas aplicando as técnicas de consistência antes e durante a busca, como será apresentado na seção seguinte.

[3] O problema das n-rainhas consiste em posicionar n rainhas em um tabuleiro de xadrez n x n, de tal forma que qualquer rainha não ataque nenhuma outra.

3.4.2 Domínios Finitos

Como vimos, os problemas que se enquadram na classe CSP caracterizam-se por terem um domínio finito ou infinito associado às suas variáveis. Vamos nos ater apenas a problemas que podem ser expressos através de domínios finitos. As técnicas de consistência, através da propagação, reduzem os domínios das variáveis, filtrando os valores que não fazem parte da solução. A propagação pode ser usada para resolver completamente o problema, entretanto isto raramente é feito devido a questões de performance. Ao invés disso, técnicas de consistência incompletas, porém mais eficientes, são combinadas com a busca para tentar chegar a uma solução.

Um CSP pode ser representado como um grafo de restrições onde os nós correspondem às variáveis e as arestas às restrições. Isto requer que o problema seja um CSP binário, isto é, contenha apenas restrições unárias e binárias. É amplamente reconhecido que um CSP arbitrário pode ser transformado no equivalente binário [9]. Entretanto, como isso nem sempre é viável na prática devido ao custo computacional que a transformação pode requerer, técnicas foram desenvolvidas para tratar também de restrições não binárias. Os ítens abaixo contemplam as principais técnicas de consistência encontradas na literatura. Apenas uma descrição sucinta sobre elas é apresentada, tendo em vista que não é nosso objetivo aprofundar nas minúcias dos inúmeros algoritmos. Maiores detalhes podem ser encontrados em [8, 65, 7, 9].

- Nó-Consistência (NC): é a técnica de consistência mais simples. Remove valores dos domínios das variáveis que são inconsistentes com as restrições unárias nas respectivas variáveis.

- Arco-Consistência (AC): é a técnica mais amplamente utilizada e esta relacionada a restrições binárias. Um arco (V_i, Vj) é arco-consistente se, para todo valor x no domínio de V_i que satisfaça as restrições unárias em V_i, existe algum valor y no domínio de V_j tal que $V_i = x$ e $V_j = y$ é permitido pela restrição binária existente entre V_i e V_j. Em outras palavras, para qualquer valor possível para uma das variáveis deve existir um valor para a outra que satisfaça a restrição existente entre elas. Existem inúmeros algoritmos conhecidos como AC-X (de *arc consistency*) começando em AC-1 e concluindo em torno de AC-7. Eles operam através de repetidas revisões dos arcos (arestas) até que um estado consistente seja alcançado ou algum domínio se torne vazio. Os mais populares dentre eles são o AC-3 e AC-4. O AC-3 realiza revisões somente para aqueles arcos que foram possivelmente afetados pela revisão prévia e o AC-4 trabalha com pares individuais de valores para remover uma potencial ineficiência de ficar checando pares de valores repetidamente.

- Caminho-Consistência (CC): também atua sobre restrições binárias porém remove mais inconsistências do que as duas técnicas anteriores. Dizemos que um caminho $(V_1, ..., V_n)$ é consistente se, para todo par x,y de valores consistentes entre V_1 e V_n, existam valores nos domínios das variáveis existentes no caminho que liga V_1 a V_n que satisfaçam todas as restrições binárias existentes entre estas variáveis. Um CSP é caminho consistente se todos os caminhos de tamanho dois (isto é, caminhos envolvendo três variáveis) são caminhos consistentes [42]. Portanto, os algoritmos para consistência de caminho lidam apenas com caminhos de tamanho dois e, como os algoritmos AC, eles tornam estes caminhos consistentes através de repetidas revisões. Mesmo sendo mais efetivo, o CC raramente é usado na prática na sua forma original devido alguns problemas:

  - Sua performance é muito pior do que a dos anteriores.
  - Requer muito mais memória mesmo para problemas pequenos.

  - Ainda assim não remove todas as inconsistências.

  Uma versão posterior do algoritmo, chamado caminho-consistência restrito, procura unir o que há de melhor no AC e CC e possui uma performance mais aceitável. Este, remove mais inconsistências do que qualquer AC, porém, é mais fraco que a versão original do caminho consistência.

- K-Consistência: todas as técnicas anteriores são instâncias de uma noção mais geral de consistência, denominada K-consistência. NC é equivalente a 1-consistência, AC a 2-consistência e CC a 3-consistência. Existem algoritmos para alcançar consistência maior do que 3, mas eles são ainda mais caros do que o CC e assim geralmente não são empregados na prática.

- Verificação antecipada (*Forward checking*): esta técnica é um caso especial de consistência de arco. Cada vez que uma variável recebe um valor, a verificação antecipada remove (temporariamente) dos domínios das variáveis ainda não instanciadas todos os valores conflitantes com as variáveis já instanciadas. Considera-se esta técnica bastante simples de implementar e às vezes ela é suficiente para resolver completamente o problema, eliminando a necessidade da busca. Quando utilizada juntamente com o retrocesso, melhora significativamente sua performance, tendo em vista que elimina previamente valores inconsistentes e evita que o retrocesso teste alternativas desnecessárias.

- Arco-Consistência generalizado: as técnicas apresentadas até aqui servem para tratar de restrições binárias somente. O arco-consistência generalizado estende a noção de arco-consistência para tratar de restrições não binárias. Uma restrição é arco-consistente generalizado se, para qualquer valor de uma variável na restrição, existam valores para as outras variáveis tal que estes juntos satisfaçam a restrição. O algoritmo AC-3 pode ser estendido para tornar a rede de restrições arco-consistente generalizado, sendo esta a mais difundida técnica utilizada para tratar de restrições não binárias [7].

- Restrições globais: muito do trabalho em relação a domínios finitos na CLP está direcionado à introdução de restrições globais [53]. A noção básica de restrições globais é a seguinte: há casos em que ter apenas uma grande restrição (grande no sentido do número de variáveis que possui) pode ser melhor do que ter várias pequenas. Um exemplo típico é o caso onde existe um conjunto de restrições de desigualdade, onde será necessário realizar uma busca completa no domínio de todas as variáveis para encontrar uma solução. Esta busca pode ser evitada utilizando-se uma restrição chamada *alldifferent*, que conecta todas as variáveis e possibilita visualizar o problema como um grafo bi-partido, no qual algoritmos clássicos de grafos podem ser utilizados para derivar informação suficiente para reduzir a busca. Em geral, a introdução de restrições globais atua da seguinte maneira: ao invés de expressar um problema através de muitas restrições binárias primitivas, podemos expressá-lo com poucas restrições sobre um conjunto de variáveis. Isso permite modelar mais facilmente os problemas, pois teremos um menor número de restrições, e possibilita que algoritmos mais sofisticados e eficientes possam ser empregados. As restrições globais devem ser genéricas o suficientes para serem empregadas em diversas aplicações. Vários grupos de restrições globais já foram propostos [52, 10] motivados por problemas reais. Exemplos típicos são os das restrições *alldifferent* (citada acima), *atmost* e *cumulative*.

Para ilustrar como as técnicas de consistência funcionam, vejamos um problema simples de escalonamento, no qual desejamos organizar a execução de seis tarefas durante cinco horas, cada uma levando uma hora para terminar. A ordem em que as tarefas devem ser executadas

é a seguinte: T1 deve executar antes que T2 e T3, T2 e T5 antes que T6 e T3 e T4 antes que T5. Além disso, existe a restrição adicional que determina que T2 e T3 não devem executar ao mesmo tempo.

Para expressar este problema como um CSP, associaremos uma variável Ti como sendo o tempo inicial de cada tarefa, cujo domínio de possíveis valores é {1,2,3,4,5}. Por fim, as restrições para definirmos o problema são:

```
before(T1,T2)
before(T1,T3)
before(T2,T6)
before(T3,T5)
before(T4,T5)
before(T5,T6)
notequal(T2,T3)
```

A restrição before implementa o arco consistência que, como vimos, tem o objetivo de remover do domínio valores inconsistentes de uma restrição com duas variáveis. Dado que T1 ∈ {1,2,3,4,5} e que T2 ∈ {1,2,3,4,5}, então, baseado na restrição before(T1,T2), conclui-se que T1 ∈ {1,2,3,4} e T2 ∈ {2,3,4,5}. O valor 5 é removido do domínio de T1 pois não existe um valor no domínio de T2 que seja consistente com essa escolha, isto é, T1 não poderá valer 5 pois não seria possível escolher um valor no domínio de T2 que satisfizesse a restrição before(T1,T2). O valor 1 é removido do domínio de T2 pela mesma razão.

A propagação continua até que nenhuma nova redução de domínio possa ser extraída a partir das restrições. O efeito de aplicar o arco consistência neste exemplo é reduzir os domínios associados com o tempo inicial de cada tarefa até chegar na seguinte configuração:

T1 ∈ {1,2}, T2 ∈ {2,3,4}, T3 ∈ {2,3}, T4 ∈ {1,2,3}, T5 ∈ {3,4}, T6 ∈ {4,5}

A aplicação apenas de técnicas de consistência raramente é suficiente para resolver um problema, tendo em vista que em geral ainda restarão combinações de valores nos domínios resultantes que serão inconsistentes. Por exemplo, os valores finais dos domínios de T1 e T2 ainda possibilitam combinações inconsistentes, dado a existência do valor 2 em ambos os domínios. Dessa forma, faz-se necessária a utilização da busca inserida neste processo. Para encontrar a solução do problema de escalonamento acima, o sistema terá que atribuir para uma variável algum valor do seu domínio, o que permitirá a geração de propagações adicionais (as estratégias de busca a serem empregadas as quais determinam, por exemplo, qual variável será primeiramente utilizada, serão vistas na próxima seção). Caso uma atribuição gere um estado inconsistente, o sistema retorna para uma situação anterior e tenta uma atribuição diferente. Por exemplo, suponhamos que a T1 seja atribuído o valor 2. Isto possibilita a dedução de que T2 ∈ {3,4} e que T3 ∈ {3}. Neste ponto, a restrição notequal(T2,T3) é usada pela primeira vez para produzir a informação de que T2 ∈ {4}. A propagação continua até que a seguinte configuração seja obtida:

T1 ∈ {2}, T2 ∈ {4}, T3 ∈ {3}, T4 ∈ {1,2,3}, T5 ∈ {4}, T6 ∈ {5}

3.4.3 Estratégias de Busca

As técnicas de consistência podem ser utilizadas em uma fase de pré-processamento, para tentar logo de início eliminar valores inconsistentes com as informações providas pelas restrições, e também durante o processo de busca. Como vimos no exemplo anterior, durante a busca há casos em que certas suposições sobre o problema precisam ser feitas (como a escolha de um elemento no domínio de uma variável), suposições estas que irão produzir mais informações sobre a solução. Uma melhoria introduzida em algumas linguagens CLP é a possibilidade

do usuário controlar o processo de busca, de tal forma que possa fazer uso de heurísticas específicas ao problema quando tiver que realizar tais suposições.

Uma das estratégias mais eficientes quando há a necessidade de se atribuir um valor para alguma variável é permitir ao usuário escolher qual variável utilizar primeiro e qual valor atribuir a ela [27]. Em muitos casos, a melhor alternativa é usar a variável com o menor domínio no momento da escolha. Este princípio é conhecido como princípio da primeira falha (*first fail*) pois, espera-se que, escolhendo a variável com menor número de alternativas, será detectado mais cedo se esta é ou não uma escolha correta. Alternativamente, a variável que ocorre num maior número de restrições pode ser escolhida, produzindo um efeito similar.

Qual valor atribuir para uma variável é em geral uma decisão mais difícil [24]. Para alguns problemas é possível definir uma métrica, como por exemplo, escolher primeiramente os menores valores do domínio. Para o problema da coloração de mapas por exemplo [65, 24], bons resultados têm sido obtidos rotacionando-se as cores atribuídas para as variáveis. Isto é, para o país A atribuí-se a primeira cor, para o país B a segunda e assim por diante. Esta abordagem tem o efeito de simular a utilização de diferentes cores sempre que possível, já que países conectados devem ter cores diferentes. Em alguns casos, especificar um único valor para a variável pode não ser a melhor opção, sendo mais viável determinar um conjunto de valores. A abordagem clássica é particionar o domínio da variável ao meio e então assumir que o valor está em uma das metades.

Todas as técnicas mencionadas procuram estender uma solução parcial consistente até uma solução completa satisfazendo todas as restrições. Outra técnica que pode ser utilizada para resolver problemas CSP baseia-se em algoritmos gulosos de busca local. Estes algoritmos alteram de forma incremental atribuições inconsistentes de valores para todas as variáveis simultaneamente. Eles utilizam uma metáfora de "reparação" para moverem-se cada vez mais em direção a uma solução completa. Este tipo de busca é conhecida como estocástica [50], sendo um dos seus mais famosos representantes o algoritmo conhecido como *hill-climbing* [8] e sua principal desvantagem é a perda da garantia de completude do método.

Um outro tipo de problema freqüentemente encontrado nas aplicações é a necessidade de obter não qualquer solução para um problema CSP, mas sim uma boa solução. A qualidade da solução é usualmente mensurada através de uma função e o objetivo passa a ser então encontrar uma solução que satisfaça todas as restrições e que minimize ou maximize o valor desta função. Tais problemas são denominados Problemas de Satisfação e Otimização de Restrições e consistem de um problema CSP padrão com a adição de uma função de otimização que mapeia toda solução a um valor numérico [61]. O algoritmo mais utilizado para encontrar soluções ótimas é o *Branch and Bound* [8] que pode ser inserido em um sistema CLP para solucionar também problemas com restrições que requerem otimização.

Para concluir esta seção, vamos analisar novamente o problema das n-rainhas para ilustrar como as diversas técnicas podem ser combinadas para resolvê-lo. Para n=8, a busca em profundidade comum — tentando todas as possibilidades e testando se o estado é um estado consistente apenas após todas as rainhas terem sido posicionadas (*generate-and-test*) — é suficiente, encontrando uma solução em poucos segundos. Entretanto, quando n=16, é necessário fazer uso de restrições juntamente com a busca para encontrar uma solução rapidamente. Quando n=32 é necessário adicionar mais inteligência ao processo. Pode-se usar o arco-consistência para reduzir os domínios das variáveis em conjunto com o princípio da primeira falha para escolher primeiramente aquelas variáveis que representam rainhas com menos "quadrados" disponíveis em seus domínios.

As técnicas acima funcionam bem para até n em torno de 70. Acima disto, deve-se usar uma heurística específica deste problema que diz que deve-se iniciar com as rainhas posicionadas no centro do tabuleiro e então movê-las para os lados. Com a combinação da busca em profundidade com restrições, utilização do arco-consistência com o princípio da

primeira falha e posicionando as rainhas no centro do tabuleiro no início, o problema pode ser rapidamente resolvido para mais de 200 rainhas [54].

3.5 Restrições Definidas pelo Usuário

No início da CLP, os mecanismos de resolução de restrições eram codificados intrinsicamente nos sistemas, com linguagens como C ou Assembly, numa abordagem conhecida como "caixa-preta" [22]. Apesar de eficiente, esta abordagem dificultava sobremaneira a modificação ou extensão dos sistemas, o que mostrou-se ser um grande problema tendo em vista a natureza heterogênea e específica da aplicação de restrições. Com isso, um dos requisitos aos sistemas CLP passou a ser que "os mecanismos de resolução sejam completamente passíveis de modificação pelos usuários" [3] (por usuários entende-se os programadores das aplicações).

Desde então, inúmeras propostas têm sido feitas para permitir maior flexibilidade na customização dos sistemas, em abordagens definidas como "caixa-de-vidro" ou mesmo "sem-caixa" [22]:

- Demons, regras progressivas e condicionais do sistema CHIP [19] permitem definir a propagação de restrições de forma limitada.
- O sistema cc(FD) [29] possibilita a criação de restrições mais complexas a partir de outras mais simples.
- Restrições conectadas a uma variável booleana no BNR-Prolog [11] permitem expressar qualquer fórmula lógica sobre restrições primitivas.
- O clp(FD) [17] possui construções que permitem implementar restrições sobre o domínio finito com um certo nível de abstração.

Uma das propostas mais recentes e promissoras baseia-se na definição de uma linguagem de alto nível para estender linguagens CLP existentes com mecanismos que permitem escrever procedimentos para resolução de restrições. Este mecanismo é conhecido como Regras para Manipulação de Restrições, ou, do inglês, *Constraint Handling Rules* (CHR). As regras para manipulação de restrições podem ser vistas como uma generalização das várias construções existentes no sistema CHIP para a definição de restrições pelo usuário [22]. Com a CHR é possível introduzir restrições primitivas (predicados) definidas pelo usuário a uma linguagem CLP pré-existente pois ela permite especificar como uma restrição deve ser processada.

3.5.1 Sintaxe e Semântica

Faremos agora uma breve descrição da sintaxe e semântica da linguagem CHR. Uma apresentação mais formal e detalhada sobre o assunto pode ser encontrada em [22] e [23].

Como vimos, a linguagem CHR foi projetada para estender uma linguagem hospedeira com capacidades adicionais de resolução de restrições. Nas definições abaixo nos reportaremos às restrições pré-definidas como aquelas manipuladas pelo mecanismo solucionador já existente na linguagem e às restrições definidas pelo usuário como aquelas criadas através de programas CHR.

Um programa CHR é um conjunto finito de cláusulas CHR. Uma cláusula CHR pode ser de três tipos:

- Simplificação: $H_1, ..., H_i \Leftrightarrow G_1, ..., G_j | B_1, ..., B_k$
- Propagação: $H_1, ..., H_i \Rightarrow G_1, ..., G_j | B_1, ..., B_k$

- SimPropagação[4]: $H_1, ..., H_l \backslash H_{l+1}, ..., H_i \Leftrightarrow G_1, ..., G_j | B_1, ..., B_k$

com $i > 0$, $j \geq 0$, $k \geq 0$, $l > 0$ e onde $H_1, ..., H_i$ é uma seqüência não vazia de restrições CHR, $G_1, ..., G_j$ é uma seqüência de restrições pré-definidas e o corpo $B_1, ..., B_k$ é uma seqüência de restrições pré-definidas e restrições CHR. Quando repetidamente aplicadas pelo mecanismo responsável por processá-las, estas regras resolvem a restrição para a qual foram definidas.

Uma importante característica destas regras, e também o que as diferem das outras abordagens, é que elas empregam o conceito de "sentinela". Uma sentinela é uma condição lógica opcional que, quando satisfeita, a cláusula é efetivamente utilizada. Nas definições acima, as restrições encontradas em $G_1, ..., G_j$ compõem a sentinela das regras CHR. Outro conceito importante incorporado pela CHR é a possibilidade de definir regras com "cabeças múltiplas", isto é, conjunção de restrições na cabeça de uma regra. Essa característica, não encontrada em muitas das propostas anteriores, é essencial para possibilitar a solução de conjunções de restrições. Apenas com "cabeças simples" a insatisfazibilidade de restrições nem sempre pode ser detectada e a satisfação de restrições globais não pode ser alcançada [22]. As restrições definidas em $H_1, ..., H_i$ formam a cabeça múltipla da regra.

A simplificação substitui restrições por outras mais simples enquanto preserva a equivalência lógica. A propagação adiciona novas restrições que são logicamente redundantes mas que podem causar simplificações adicionais. Uma vez tendo sido executada, ela não irá executar novamente com a mesma combinação de restrições definidas pelo usuário. Uma regra do tipo simpropagação é um tipo híbrido de regra. Todas as restrições que casam com a cabeça da regra antes do operador \ são removidas, enquanto que aquelas que casam com as outras restrições da cabeça são mantidas. Em qualquer caso, o corpo é executado, isto é, suas restrições são adicionadas ao repositório apropriado.

Para concluir este tópico, vejamos o programa CHR abaixo, o qual define como deve ser processado o predicado <=:

```
X <= Y ⇔ X=Y | true          — reflexividade
X <= Y, Y <= X ⇔ X=Y         — antisimetria
X <= Y, Y <= Z ⇒ X <= Z      — transitividade
```

Estas cláusulas especificam como o operador <= deve ser simplificado e propagado como uma restrição e supõem a existência de um operador de igualdade sintática. A reflexividade define que X <= Y é logicamente verdadeiro se X=Y for verdadeiro (sentinela). Sempre quando ocorrer a restrição X <= Y e a condição for satisfeita ela pode ser substituída por *true*. A antisimetria significa que, se as restrições $X <= Y$ e $Y <= X$ forem encontradas elas podem ser substituídas pela equivalência lógica X=Y. Note a diferente utilização de X=Y nas duas regras. Na reflexividade a igualdade é uma pré-condição enquanto na antisimetria fará parte do conjunto de restrições a ser avaliado. Estas duas regras são simplificações, já a terceira é uma propagação de restrição. A transitividade define que a conjunção X <= Y, Y <= Z implica em X <= Z, que é adicionado como uma restrição redundante. Dada a consulta A <= B, C <= A, B <= C, os passos de aplicação das regras serão:

```
C <= A, A <= B propaga C =< B        — transitividade
C <= B, B <= C simplifica para B=C   — antisimetria
A <= B, C <= A simplifica para A=B   — antisimetria pois B=C
```

Depois da aplicação destas regas, a consulta não possui mais nenhum operador <= para processar e a simplificação termina produzindo a resposta $A = B, B = C$.

[4]Cunhamos este termo com base no que foi encontrado na literatura para definir este tipo de cláusula. O nome original, Simpagation, é uma mistura de Simplification com Propagation.

Solucionadores de restrições construídos com CHR são garantidamente confluentes [23], isto é, não importa a ordem em que as regras são aplicadas, a resposta no final será sempre a mesma.

3.6 Sistemas e Aplicações CLP

Nesta seção faremos um resumo dos principais sistemas CLP já implementados e das principais aplicações que fazem uso desta tecnologia. A intenção é que a lista seja o mais completa possível, entretanto, devido às inúmeras áreas que a CLP se aplica podemos ter deixado de selecionar algo.

A programação com restrições tem sido integrada em diferentes linguagens de programação, variando desde subconjuntos da lógica de primeira ordem, linguagens procedimentais como C++, até linguagens especialmente desenvolvidas [60]. Como vimos, uma das abordagens mais populares é utilizar cláusulas de Horn como base e então estendê-las com um ou mais domínios de restrições. Esta é a abordagem da Programação Lógica com Restrições, a qual possui inúmeros representantes importantes como:

- CHIP [19]: a linguagem de programação lógica com restrições CHIP foi desenvolvida pelo *European Computer-Industry Research Centre* (ECRC). Sua mais importante característica foi a introdução de restrições aritméticas sobre domínios finitos solucionadas por técnicas de consistência. Além de restrições sobre domínios finitos, o sistema provê também restrições booleanas e restrições lineares no domínio dos números racionais. Ele possibilita ao usuário definir suas próprias restrições e controlar a propagação através de demons e construções `if-them-else`. Muitas melhorias têm sido introduzidas desde o seu surgimento e muitos sistemas comerciais atualmente disponíveis baseiam-se em sua tecnologia.

- CLP(R) [36]: linguagem tem sido desenvolvida para demonstrar o esquema CLP(X). Lida com restrições aritméticas lineares utilizando para isso um algoritmo simplex modificado.

- Prolog-III [18]: incluí os domínios aritmético linear, booleano e listas finitas. O domínio aritmético é tratado pelo algoritmo simplex e o booleano é baseado num método de saturação. As facilidades para lidar com listas finitas pode ser encontrada em detalhes em [18].

- Trilogy [64]: é uma linguagem desenvolvida pela *Complete Logic Systems* em Vancouver, Canadá. Seu domínio é sobre as restrições aritméticas com números inteiros, isto é, permite equações lineares, inequações e desigualdades sobre os inteiros. Diferentemente de outros sistemas CLP, o Trilogy não está integrado num ambiente Prolog, mas sim, está baseado em sua própria "teoria de pares".

- CAL [3] e GDCC [2]: CAL *(Constraint Avec Logic)*, foi desenvolvido no ICOT, Tóquio, como a primeira linguagem CLP a prover restrições aritméticas não-lineares. Uma versão paralela do sistema, chamada GDCC, provê restrições sobre os seguintes domínios: equações não-lineares sobre os números reais e solucionadas pelo algoritmo gröbner base, restrições booleanas solucionadas por uma versão modificado do algoritmo gröbner base, além de restrições aritméticas lineares. O sistema também integra o método de busca *branch and bound* para resolver problemas de otimização.

- BNR-Prolog [11]: foi desenvolvido pela *Bell-Northern Research* em Ottawa e projetado especialmente para máquinas Macintosh. A característica interessante deste sistema

sob o ponto de vista da CLP é a introdução da chamada aritmética relacional. Este domínio é baseado numa representação de intervalos, com as variáveis representando um número real entre os valores mínimo e máximo do intervalo.

- ECLiPSe [32, 66]: ECLiPSe é um sistema baseado no Prolog cujo objetivo é servir como plataforma para integrar várias extensões da programação lógica, em particular a programação lógica com restrição. Possui uma vasta biblioteca que integra domínios como: restrições lineares, intervalos, domínios finitos, CHR, dentre outros. Também provê facilidades para integração com outras linguagens como Java e C, além de uma interface e protocolo para comunicação remota com o sistema. Foi desenvolvido pelo IC-PARC *(Centre for Planning and Resource Control)* no Colégio Imperial de Londres e possui diversos artigos publicados pelos seus mentores.

Outra abordagem popular é embutir técnicas de CLP em diferentes linguagens, criando ferramentas como:

- CHARME: linguagem com sintaxe semelhante ao C e domínios finitos baseados no sistema CHIP.
- 2LP: linguagem baseada em C com restrições lineares.
- ILOG Solver [51, 33]: biblioteca C++ com domínios booleanos, finitos, intervalos reais (análogo ao encontrado no BNR-Prolog) e restrições lineares. A biblioteca combina programação orientada a objetos com CLP e oferece conceitos como variáveis lógicas, resolução incremental de restrições e retrocesso.
- JACK [1]: biblioteca baseada em Java para programação com restrições. É composta por três partes: uma linguagem de alto nível denominada JCHR *(Java Constraint Handling Rules)* que permite a criação de restrições pelo usuário, uma ferramenta interativa que permite visualizar a execução dos programas (VisualCHR) e uma máquina genérica de busca denominada JASE *(Java Abstract Search Engine)*.

Uma generalização da CLP surgiu da observação que restrições também poderiam ser usadas para modelar comunicações entre "agentes" e sincronização em sistemas concorrentes [53]. Esta idéia serviu como base para a definição de um esquema para programação concorrente com restrição, conhecida como *cc framework* [55]. A generalização estendeu o escopo das linguagens CLP, possibilitando que características como concorrência, controle e extensibilidade sejam abordadas diretamente pela linguagem. Por fim, alguns sistemas que oferecem linguagens concorrentes com restrições são:

- cc(FD) [29]: o cc(FD) é uma instância do *cc framework* sobre domínios finitos, generalizando várias características e restrições do sistema CHIP. Ele incorpora combinadores de propósito geral que permitem a definição de novas restrições primitivas.
- AKL [26]: linguagem concorrente que engloba domínios finitos e suporte a execução paralela.
- Oz [30, 43]: linguagem concorrente que mescla múltiplos paradigmas (orientação a objetos, processamento simbólico, busca) e prove suporte para processamento distribuído.
- CIAO [31]: linguagem concorrente com restrições lineares. Suporta execução paralela e distribuída bem como inúmeras regras de controle e funções.

Linguagens CLP têm sido utilizadas com sucesso numa ampla variedade de aplicações, porém, são particularmente apropriadas para resolverem problemas de busca com restrições, tais como escalonamento, alocação, layout e projeto de hardware [24]. Algumas aplicações típicas são [53, 60]:

- Problemas de alocação: foi um dos primeiros tipos de problemas solucionados com tecnologia CLP. Estes problemas geralmente consistem em manipular dois tipos de recursos e restrições entre eles. O objetivo é tentar atribuir um recurso do primeiro tipo a um do segundo tipo de tal forma que todas as restrições sejam satisfeitas. Um exemplo simples poderia ser o de atribuir salas para funcionários, no qual alguém impôs a restrição de que "todos os funcionários que possuem o nome iniciando pela mesma letra, devem compartilhar a mesma sala, num máximo de cinco pessoas por sala".

- Gerenciamento de rede: este tipo de aplicação pode ser modelada com domínios finitos e resolvida de forma eficaz pelas técnicas empregadas pela CLP. Alguns sistemas nesta área são o LOCARIM, desenvolvido pela COSYTEC para a Telecom francesa, o PLANETS, desenvolvido pela Universidade da Catalonia em Barcelona para a companhia de eletricidade espanhola e o sistema POPULAR, que executa o planejamento de redes digitais sem fio para comunicação móvel e foi escrito inicialmente em ECLiPSe e depois com as bibliotecas CHR do ECLiPSe.

- Problemas de escalonamento: esta talvez seja a aplicação de maior sucesso para domínios finitos usando CLP. De forma geral consiste em: dado um conjunto de recursos e suas respectivas capacidades, um conjunto de atividades com suas respectivas durações e requisitos de recursos e um conjunto de restrições temporais entre as atividades, uma solução para o problema deve decidir quando executar cada atividade de tal forma que as restrições temporais e de recursos sejam satisfeitas. Inúmeras aplicações comerciais foram desenvolvidas com linguagens CLP para tratar deste tipo de problema.

- Problemas de transporte: estes problemas geralmente são complexos, dado o número e variedade de restrições e a possibilidade de existirem restrições bastante complexas que modelam as relações entre possíveis rotas. Uma variedade de problemas de transporte têm sido tratados utilizando-se restrições.

- Controle de sistemas eletro-mecânicos: a tecnologia CLP também está sendo explorada para construir softwares de controle para sistemas eletro-mecânicos com um número finito de entradas, saídas e estados internos. Cada componente de um sistema complexo está conectado a uma parte menor do sistema completo e seu funcionamento pode ser capturado sem muita dificuldade. Porém, quando o sistema é considerado como um todo, o número de estados globais pode se tornar muito grande, necessitando assim de uma tecnologia apta a tratar de tal explosão combinatorial.

- Interfaces gráficas: esta é uma das mais antigas áreas de aplicação da CLP. A principal função das restrições aqui é manter todos os objetos gráficos corretamente posicionados uns em relação aos outros após a submissão de alguma modificação. A propagação de restrições nestas aplicações deve ser rápida e deve ser capaz de lidar com inconsistências. Por exemplo, se um objeto é movido para uma posição onde irá sobrepor qualquer outro objeto isto deve ser tratado.

3.7 Comentários Finais

Tentamos neste capítulo consolidar os diferentes conceitos relacionados à programação lógica com restrições. Enquanto mantém a principal característica da programação lógica, isto é,

a declaratividade e flexibilidade, a CLP traz nas suas linguagens a eficiência dos algoritmos de propósito específico e a expressividade dos diferentes domínios que engloba. Apesar da facilidade trazida pelas linguagens CLP para descrever problemas, numa recente pesquisa [53] ficou constatado que uma das principais necessidades para tecnologia CLP é torná-la mais fácil de se aprender e utilizar. Trabalhos têm sido desenvolvidos na tentativa de introduzir mecanismos que tornem a modelagem dos problemas mais fácil e natural.

No capítulo seguinte, apresentaremos uma proposta de integração da programação lógica com restrição com a programação orientada a objetos que, esperamos, irá facilitar a sua utilização em aplicações que fazem uso deste paradigma. Definiremos um conceito de objeto inteligente que engloba os métodos e atributos usuais, uma base de conhecimento formada por regras escritas em alguma linguagem CLP e uma máquina de inferência para processar as regras e produzir comportamentos inteligentes. Quando necessário, um objeto inteligente pode disparar um processo de inferência para realizar deduções baseadas no seu estado interno e no conhecimento especificado por suas regras. A integração irá respeitar características de orientação a objetos dos objetos aos quais o conhecimento está associado, isto é, encapsulamento, herança e polimorfismo.

CAPÍTULO 4

Objetos Inteligentes e CLP

4.1 Introdução

Neste capítulo propomos um esquema de integração de objetos com a programação lógica com restrições nos moldes da abordagem adotada por Bomme [13]. Fizemos esta escolha por a considerarmos a mais adequada para o objetivo do trabalho, isto é, integrar técnicas de IA para ajudar na construção de aplicações científicas no próprio ambiente em que estas aplicações são comumente desenvolvidas.

Na Seção 4.2, fazemos uma breve introdução sobre o conceito de objetos e das principais características do paradigma da orientação a objetos, cujo conteúdo é um resumo do capítulo 8 de Pagliosa [47]. Na Seção 4.3, mostramos como um objeto comum se transforma num objeto inteligente, evidenciando como a integração proposta respeita totalmente as características da orientação a objetos dos objetos aos quais o conhecimento está associado. A Seção 4.4 traz detalhes do esquema de integração, apresentando os principais aspectos da arquitetura que adotamos. Primeiramente, apresentamos como as regras CLP são descritas e como estão associadas aos objetos. Mostramos então como o encapsulamento, herança e polimorfismo são obtidos para as regras e como a base de conhecimento de um objeto é criada a partir destas definições. Por fim, falamos sobre o mecanismo de raciocínio e como este se encaixa na arquitetura da solução. Abordamos a questão de como as regras são processadas pela máquina de inferência e o esquema de reflexão computacional que tivemos que adotar para que a solução funcionasse como pretendíamos.

4.2 O que são Objetos?

A orientação a objetos é a técnica de desenvolvimento e modelagem de sistemas que facilita a construção de programas complexos a partir de conceitos que nos permitem modelar o mundo real tão próximo quanto possível da visão que temos desse mundo. Enquanto o desenvolvimento estruturado baseia suas abstrações para compreensão de um problema nos procedimentos necessários para sua solução, a orientação a objetos nos permite modelá-lo num nível maior de abstração, através da identificação natural que fazemos das entidades que o compõe, caracterizando-as a partir de certas propriedades que definem seus principais aspectos estruturais e comportamentais.

Objetos são um conjunto de dados mais um conjunto de procedimentos que representam a estrutura e o comportamento inerentes às entidades, concretas ou abstratas, do mundo real. Entidades concretas são aquelas que representam objetos concretos tais como um carro ou um avião. Possuem uma coleção de atributos (material, características mecânicas, custos, métodos construtivos) e uma coleção de métodos próprios que manipulam e fornecem respostas a partir dos valores desses atributos (reações, deslocamentos, custo total). Além destas, entidades não tão concretas, como a representação de elementos geométricos num plano, por exemplo, são tratadas como objetos. Nesse caso, a estrutura é composta por atributos que constituem variáveis como o número de lados e valores dos ângulos internos e o comportamento é ditado por métodos que informam a área ou algum outro valor que servirá de contribuição para a solução de um problema matemático. Na programação orientada a objetos (POO), nada impede que a essência de um objeto seja o processamento ao invés da estruturação ou vice-versa. Na verdade, objetos podem ser, em princípio, o modelo de qualquer coisa, desde um processamento puro até um dado puro, sendo todos, sob a ótica da POO, igualmente considerados objetos.

Uma vez tendo abstraído os objetos que serão utilizados para compor a solução do problema, utilizaremos o conceito de *tipo abstrato de dados* para os definirmos formalmente. Tipo abstrato de dados é o mecanismo pelo qual uma linguagem de programação orientada a objetos fornece suporte à especificação dos dados de um objeto e dos procedimentos executados por um objeto. A *classe* é a construção de linguagem mais comumente utilizada para implementar um tipo abstrato de dados em linguagens de programação orientada a objetos. Objetos com a mesma estrutura e o mesmo comportamento pertencem a mesma classe de objetos. Ao conjunto de dados de um objeto damos o nome de *atributos* e ao conjunto de procedimentos de *métodos* do objeto. Uma classe, portanto, é uma descrição dos atributos e dos métodos de determinado tipo de objeto. Em termos de programação, definir classes significa formalizar um novo tipo de dado e todas as operações associadas a esse tipo, enquanto declarar objetos significa criar variáveis do tipo definido.

Uma *instância* é um objeto gerado por uma classe. A classe é apenas a representação de um objeto enquanto a instância de uma classe é o objeto propriamente dito, com tempo e espaço de existência. O conteúdo de um objeto, ou seu *estado interno*, é definido por suas *variáveis de instância* (atributos), sendo que dois objetos diferentes da mesma classe podem possuir valores diferentes para seus atributos e conseqüentemente apresentarem estados internos diferentes. Podem existir também variáveis que não pertencem a qualquer instância de uma classe, mas são compartilhadas por todas as instâncias da classe. Estas variáveis são chamadas de *variáveis de classe*. Um objeto pode ter ainda uma *identidade* que é uma propriedade que o distingue dos demais objetos da aplicação.

Numa aplicação estruturada os componentes básicos são os procedimentos e a comunicação entre os mesmos se dá pela passagem de dados, que são processados em blocos estruturados, e migram de um para outro. O fluxo de execução é caracterizado pelo acionamento de procedimentos cuja tarefa é a manipulação dos dados. Num programa orientado a objetos, dados e procedimentos fazem parte de uma só unidade (objeto) e estas unidades, ao estabelecerem comunicação entre si, caracterizam a execução do programa. Um objeto estabelece comunicação com outro através do envio de *mensagens*, que é uma solicitação para o objeto receptor da mensagem executar alguma operação. Quando um objeto recebe uma mensagem, o método correspondente à mensagem é executado, isto é, o método do objeto receptor que possui o mesmo nome e o mesmo tipo e número de parâmetros da mensagem. Chamamos essa operação de *acoplamento mensagem/método*.

4.2.1 Encapsulamento

Como vimos, diferentemente da programação estruturada, na POO os dados e os procedimentos que manipulam estes dados são definidos numa unidade única, a classe, permitindo uma melhor modularidade do código. Além disso, a idéia é poder utilizar os objetos sem ter que se conhecer sua implementação interna, que deve ficar escondida do usuário do objeto que irá interagir com este apenas através de sua *interface*. À propriedade de se implementar dados e procedimentos correlacionados em uma mesma entidade e de se proteger sua estrutura interna escondendo-a de observadores externos dá-se o nome de encapsulamento.

Via de regra, as variáveis de instância declaradas em uma definição de classe, bem como os métodos que executam operações internas sobre estas variáveis, se houverem, devem ser escondidos na definição da classe. Isso é feito geralmente através de construções nas linguagens de programação conhecidas como modificadores de acesso, como por exemplo o **`public`**, **`protected`** e **`private`** do C++ e do Java. O **`public`** define a porção da classe que será visível e poderá ser diretamente acessada por outros objetos na aplicação, enquanto o **`protected`** e **`private`** definem a porção protegida da classe. Quando definimos uma classe, é recomendado que declaremos como públicos apenas os métodos da sua interface. É na interface (ou protocolo) da classe que definimos quais mensagens podemos enviar às instâncias de uma classe, ou seja, quais são as operações que podemos solicitar que os objetos realizem.

O objetivo do encapsulamento é separar o usuário do objeto do programador do objeto. O benefício óbvio é que podemos alterar a implementação de um método ou a estrutura de dados escondidos de um objeto sem afetar as aplicações que dele se utilizam.

4.2.2 Herança

A herança é a principal característica de distinção entre um sistema de programação orientado a objeto e outros sistemas de programação. As classes são inseridas em uma hierarquia de especializações de tal forma que uma classe mais especializada herda todas as propriedades da classe mais geral a qual é subordinada na hierarquia. A classe mais geral é denominada superclasse (classe base, em C++) e a classe mais especializada subclasse (classe derivada, em C++).

O principal benefício da herança é a reutilização de código. A herança permite ao programador criar uma nova classe programando somente as diferenças existentes na subclasse em relação a superclasse. Isto se adequa bem a forma como compreendemos o mundo real, no qual conseguimos identificar naturalmente estas relações. Por exemplo, se quiséssemos modelar elementos geométricos triangulares, poderíamos ter uma classe denominada `Triângulo` que englobasse as informações e comportamento comuns a qualquer triângulo e depois criarmos especializações desta classe para representarmos propriedades e ações específicas de determinados tipos de triângulos, como os equiláteros, isósceles e escalenos.

Consideremos o programa abaixo codificado em C++:

```
class X
{
  public::
     int w,z;
     double k;
     X();
     void M1();
     void M1(int a);
     virtual void M2();
}; // X
```

```
class Y: public X
{
   public::
      Y();
      void M1();
      void M2();
      void M3();
}; // Y
```

Primeiro, a classe X com seus métodos e atributos é declarada e logo em seguida a classe Y que deriva da classe X. A classe Y, além de atributos e métodos próprios, contém todos os atributos e métodos de X, sem que, com isso, tenhamos que redefini-los em Y. Como a classe Y apresenta comportamentos diferentes para alguns métodos que herdou de X, esta pode ainda sobrecarregá-los (consideraremos sobrecarga na próxima seção) para definir sua própria implementação, como fez com M1() e M2(). Como apenas os métodos herdados de X não são suficientes para definir todo o comportamento de Y, a definição da classe conta com um método adicional, M3().

4.2.3 Polimorfismo

O termo polimorfismo origina-se do grego e quer dizer "que possui várias formas". Em programação está relacionado à possibilidade de se usar o mesmo nome para métodos diferentes e à capacidade que o programa tem em discernir, dentre os métodos (ou operadores) homônimos, aquele que deve ser executado. De maneira geral o polimorfismo permite a criação de programas mais claros, pois elimina a necessidade de darmos nomes diferentes para métodos que conceitualmente fazem a mesma coisa, e também programas mais flexíveis, pois facilita em muito a extensão dos mesmos.

O polimorfismo pode ser obtido de duas maneiras:

- através do acoplamento estático ou junção anterior, onde os tipos de objetos envolvidos são conhecidos em tempo de compilação, ou
- através do acoplamento dinâmico ou junção posterior, que é o caso onde o tipo do objeto só é conhecido durante a execução do programa e, portanto, o compilador não pode decidir, em tempo de compilação, qual método de qual classe acoplar à mensagem.

No acoplamento estático o compilador decide qual método executar apenas comparando o tipo do objeto e tipo e número de parâmetros do método em questão. No exemplo acima, se um objeto do tipo definido pela classe X receber uma mensagem para executar o método M1(10), o compilador saberá, em tempo de compilação, que o acoplamento deverá ser feito com o método M1(int a) e não com M1(). Dizemos que o método M1 está *sobrecarregado* pois existem duas versões do método na mesma definição de classe. A sobrecarga pode ocorrer também na herança. Quando Y recebe uma mensagem para executar M1(), este vai executar sua versão do M1() e não a que ele herdou de X. Já se receber uma mensagem solicitando a execução de M1(10), o compilador saberá que deve invocar o método herdado de X.

Contudo, a característica mais notável do polimorfismo em sistemas de programação orientada a objetos é o acoplamento dinâmico, o qual está ligado com o conceito de herança. Consideremos o trecho de programa abaixo, também escrito em C++:

```
...
X *a; //declara um ponteiro para um objeto do tipo X
Y *b = new Y(); //declara e cria um objeto do tipo Y
a = b;
a->M1();
```

```
a->M2();
...
```

Para entendermos a real utilidade do polimorfismo com junção posterior, temos primeiramente que perceber como é útil uma atribuição como a que fizemos em `a = b`. Em programas orientados a objetos é comum e necessária a possibilidade de atribuir para um ponteiro de um tipo base o endereço de um objeto de um tipo derivado direta ou indiretamente deste tipo base. Dito isto, é razoável que queiramos, quando uma atribuição como esta ocorrer, que os métodos executados utilizando-se a variável `a` sejam sempre os apropriados, isto é, que sejam os métodos pertencentes ao objeto que a variável do tipo base aponta naquele momento.

Em C++ a especificação de quais métodos de uma classe poderão ser acoplados dinamicamente é feita seletivamente, com o uso do especificador **`virtual`**. Quando fazemos `a->M1()`, como `a` é um ponteiro (esta é uma das condições para conseguirmos o acoplamento dinâmico), o compilador irá verificar se `M1()` é um método virtual. Como `M1()` não é declarado como **`virtual`** na definição da classe `X`, o compilador irá acoplar, estaticamente, o método `M1()` da própria classe `X`, baseando-se no tipo com o qual a variável `a` foi declarada. Mesmo estando `a` apontando para um objeto do tipo `Y` e existindo em `Y` outra definição do método `M1()`, este não será executado.

Já na definição do método `M2()`, como o método foi declarado como **`virtual`** na classe base, o compilador não fará nenhum acoplamento estático quando encontrar a sentença `a->M2()`, mas sim, irá gerar código para que, em tempo de execução, seja decidido qual método `M2()` será chamado, se o da classe `X` ou o da classe `Y`, dependendo para onde `a` aponta naquele momento. No código acima, como `a` aponta para um objeto do tipo `Y`, o método chamado será `M2()` de `Y`. Isso é o que chamamos de acoplamento dinâmico.

Resumindo, um método declarado como **`virtual`** em uma classe base pode ser sobrecarregado nas classes derivadas, possibilitando que uma especialização forneça sua própria versão para o método e, mais importante, que essa versão possa ser acoplada em tempo de execução. Em Java não é preciso especificar quais métodos serão acoplados dinamicamente pois, por padrão, o compilador da linguagem gera código para possibilitar o polimorfismo dinâmico para todos os métodos públicos de uma classe.

4.3 O que são Objetos Inteligentes?

Um objeto inteligente possui uma definição híbrida [13] consistindo de:

- dados e métodos que representam um objeto usual; e
- conhecimento formulado através de regras (sejam de produção ou CLP) e processado pelo objeto para produzir comportamentos inteligentes.

Os comportamentos inteligentes podem ser invocados através dos métodos usuais de um objeto numa aplicação orientada a objetos. Esta definição resulta na junção de um paradigma procedimental e imperativo com um declarativo e não seqüencial, a fim de gerar objetos híbridos como ilustrado na Figura 4.1. Estes objetos provêm às aplicações uma arquitetura na qual a representação do conhecimento é distribuída, já que cada objeto detém seu próprio conhecimento, e o processamento é descentralizado.

4.3.1 Regras como Pseudo-Métodos

Segundo Graham [25], "permitir o encapsulamento e políticas de herança associados às regras, insere naturalmente a idéia de regra à metáfora da orientação a objetos, levando a um estilo de

Figura 4.1: Objeto inteligente.

expressão uniforme e a um claro entendimento do status das regras no modelo". Então, assim como fez Bomme com o objetivo de tornar sua integração mais clara e natural, consideraremos as regras como *pseudo-métodos* atuando sobre o estado interno dos objetos.

Numa regra é possível acessar diretamente os dados internos do objeto ao qual esta pertence ou enviar mensagens para outros objetos, a fim, por exemplo, de acessar seus dados encapsulados. As regras podem participar de hierarquias de herança e é possível também atribuir modificadores de acesso a elas, restringindo ou não sua visibilidade em classes derivadas. Uma regra pode ainda receber parâmetros, que são utilizados no corpo da regra quando esta estiver sendo processada.

Portanto, apesar de uma regra não ser um método no sentido estrito da palavra, possui muitos atributos que a tornam muito semelhante a um, o que nos permite considerá-la um pseudo-método da linguagem e a definir as seguintes categorias de regras:

- regras públicas, são aquelas visíveis na própria classe e em classes derivadas da classe que contém a regra; e
- regras privadas, são visíveis apenas dentro da própria classe onde estão definidas.

Como os pseudo-métodos respeitam estritamente a definição dos métodos usuais, eles são organizados numa hierarquia induzida diretamente da hierarquia de classes da aplicação orientada a objetos. Se a hierarquia de classes da aplicação é modificada, as mudanças são automaticamente propagadas para a representação do conhecimento; sendo assim, ambas as hierarquias mantém-se sempre consistentes.

4.3.2 Processamento do Conhecimento

O processamento do conhecimento é mantido sobre o controle dos próprios objetos que podem, quando necessário, disparar um mecanismo de inferência para produzir comportamentos inteligentes. Um objeto inteligente poderia disparar este mecanismo em duas situações:

- Quando precisa utilizar seu conhecimento para resolver um problema. Neste caso, um processo de encadeamento regressivo é disparado para provar a consulta em questão.
- Quando seu estado interno é alterado e o objeto precisa verificar a consistência de seus dados. Neste caso, um processo de encadeamento progressivo é disparado para verificar se todas as restrições impostas sobre seus atributos estão sendo satisfeitas.

4.3.3 Visão de Aplicações com Objetos Inteligentes

A Figura 4.2 sumariza a visão de uma aplicação que faz uso de objetos inteligentes. Esta é definida por uma hierarquia de classes onde algumas delas integra conhecimento e o fluxo de execução se dá através de objetos que trocam mensagens entre si. Esta arquitetura difere em muito da arquiteturas dos sistemas de produção tradicionais. Nestes sistemas, existe apenas

uma única base de conhecimento que engloba todas as regras de produção, ou seja, estas não estão acopladas a nenhum objeto específico. As premissas e conclusão de uma regra são construídas em termos dos objetos existentes na aplicação, os quais são agrupados em uma estrutura denominada base de objetos. De acordo com o estado interno destes objetos, regras ficam ativas ou desativas, sendo que esta distinção é feita agrupando-se as regras ativas numa estrutura denominada *conjunto de conflito*. Quando as regras são processadas elas podem inserir ou excluir objetos da base de objetos ou mesmo modificar seu estado interno. Qualquer uma destas operações requer uma reavaliação do conjunto de conflito. Este modelo esta resumido na Figura 4.3 [20].

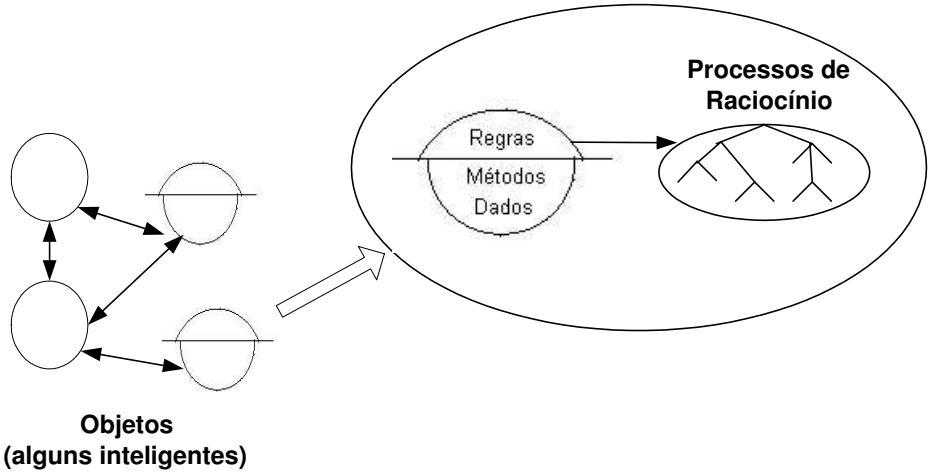

Figura 4.2: Visão de uma aplicação com objetos inteligentes.

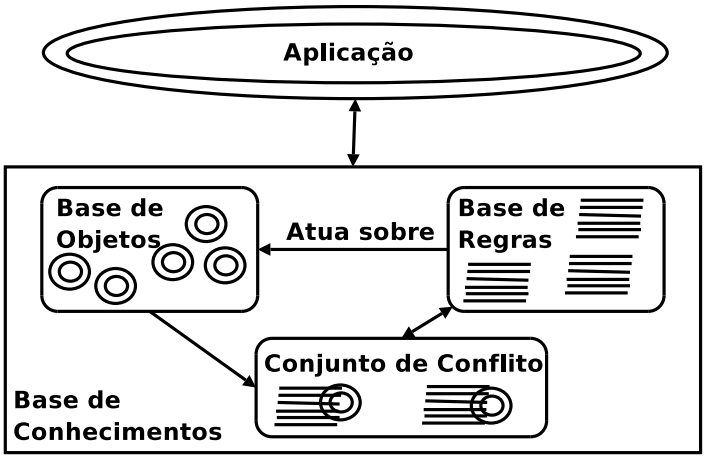

Figura 4.3: Arquitetura usual dos *Embedded Object-Oriented Production Systems*.

Neste esquema, as regras atuam nos objetos e o processo de raciocínio permanece um aspecto global e central do sistema. Não há nenhuma ligação entre a forma como os objetos estão organizados na aplicação com a hierarquia formada pelas regras (Figura 4.4). Já em uma aplicação com objetos inteligentes, como o conhecimento está acoplado na forma de pseudo-métodos associados aos objetos, este encontra-se organizado numa hierarquia idêntica a hierarquia de classes da aplicação [13] (Figura 4.5).

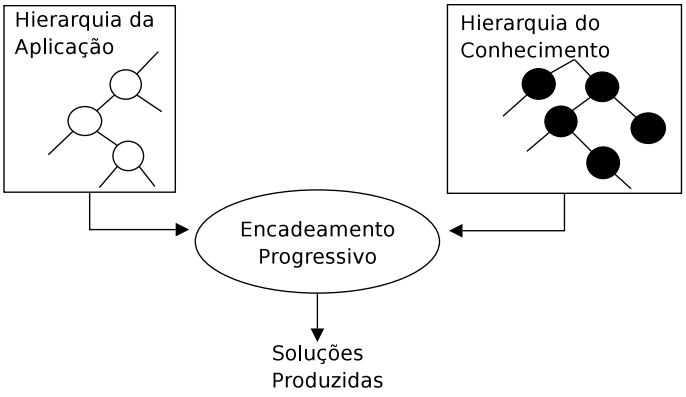

Figura 4.4: Hierarquia do conhecimento na abordagem dos EOOPS.

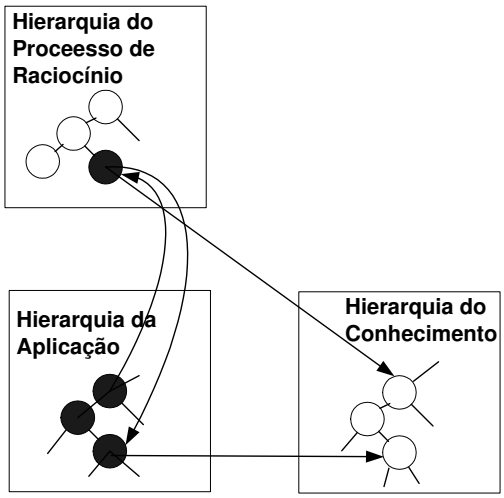

Figura 4.5: Hierarquia do conhecimento em uma aplicação com objetos inteligentes.

4.4 Projeto dos Objetos Inteligentes

Na seção anterior definimos o que é um objeto inteligente e como suas características diferem da abordagem usual que os sistemas baseados em conhecimento orientados a objetos utilizam para integrar regras e objetos. Nessa seção apresentamos o projeto para construção dos objetos inteligentes baseados em CLP, cuja arquitetura está resumida na Figura 4.6. Vale lembrar que, pelos motivos discutidos no Capítulo 1, o sistema de objetos inteligentes implementado neste trabalho utiliza Java como linguagem nativa.

Um objeto inteligente é uma instância de uma classe derivada de `IntelligentObject`. Esta classe define a estrutura e comportamento comum a todo objeto inteligente. Seu principal atributo é a base de conhecimento, um objeto da classe `KnowledgeBase`, e sua principal funcionalidade é definida pelo método `ask()`.

Os passos para criação e utilização do conhecimento por um objeto inteligente são:

Passo 1 Especificação do conhecimento.

Passo 2 Criação da base de conhecimento.

Passo 3 Acesso a mecanismos de raciocínio.

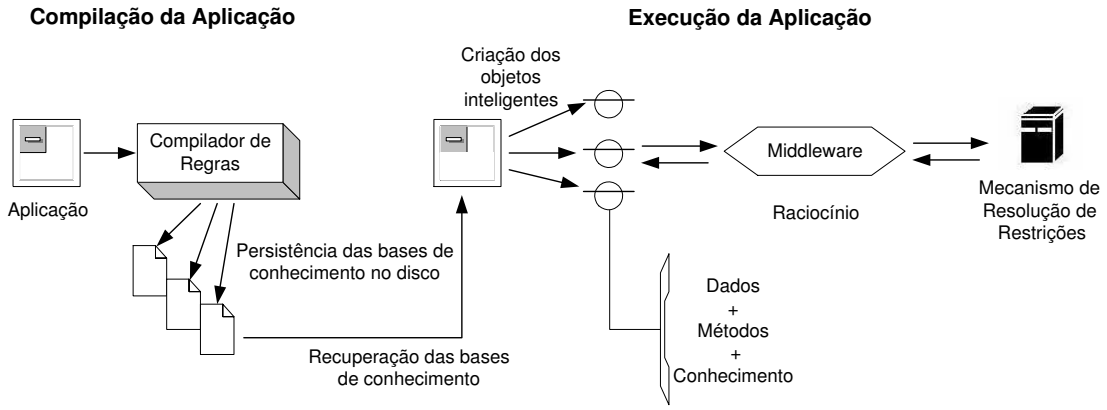

Figura 4.6: Arquitetura do projeto.

4.4.1 Especificação do Conhecimento

Toda classe derivada de `IntelligentObject` pode conter especificação de regras. Diferentemente dos EOOPS convencionais que codificam o conhecimento através de regras de produção, o conhecimento de nossos objetos inteligentes é definido através de regras CLP. Como não queremos nos ater a nenhuma implementação específica de CLP, definimos nossa própria linguagem para representação do conhecimento. Abaixo segue um trecho da gramática que define esta linguagem. A descrição completa encontra-se no Apêndice A.

```
program ::=
   '/*?'  clause_list '*/'

clause_list ::=
   clause
   | clause_list clause

clause ::=
   modifier?  RULE IDENTIFIER rule';'

modifier ::=
   PUBLIC
   | PRIVATE

rule ::=
   fact_list
   | head IMP_OP statement_list

fact_list ::=
   predicate
   | fact_list ',' predicate

head ::=
   predicate

statement_list ::=
   statement
   | statement_list ',' statement

predicate ::=
   IDENTIFIER'(' term_list ')'

statement ::=
   conditional
   | term
```

```
conditional ::=
   IF '(' expression ')' term_list
   | IF '(' expression ')' term_list ELSE term_list

term_list ::=
   term
   | term_list ',' term

term :==
   expression

expression ::=
   primary_expression
   | unary_expression
   | binary_expression

binary_expression ::=
   expression BIN_OP expression

unary_expression ::=
   UN_OP expression

primary_expression ::=
   predicate
   | ATOM
   | list
   | reference
   | '(' term_list ')'
   | variable
   | constant

reference ::=
   THIS
   | reference '.'  IDENTIFIER
   | reference '(' expression_list ')'
   | reference '(' ')'

list ::=
   '[' term_list ']'

expression_list ::=
   expression
   | expression_list ',' expression

constant ::=
   STRING
   | NUMBER
```

Um programa CLP é uma lista de cláusulas. Cada uma deve ter um identificador (um nome) e pode, opcionalmente, conter um modificador de acesso indicando se é pública ou privada (caso nenhum esteja definido, a cláusula é considerada privada). Uma cláusula contém também uma regra, que pode ser uma lista de fatos ou uma implicação. Fatos definem informações fatuais que são utilizadas durante o processamento das regra. Uma implicação é uma regra composta por um predicado na cabeça da regra e de uma conjunção de assertivas no corpo. A interpretação é a seguinte: caso a avaliação das assertivas presentes no corpo resulte em verdade, então pode-se concluir que o predicado que forma a cabeça também é verdadeiro. Se a assertiva no corpo for uma condicional, então só esta poder estar presente. Uma restrição condicional tem a mesma semântica do comando de controle de fluxo **`if`** –

else das linguagens não lógicas de programação, isto é, caso a avaliação da expressão presente no **if** resulte em verdade, então concluí-se o que vem abaixo dele, caso contrário, conclui-se o que estiver no **else** (caso existente). Se a assertiva no corpo não é uma condicional, então será uma conjunção de expressões formadas pela combinação de predicados, átomos, listas, variáveis, constantes, atributos ou métodos, conectados entre si através dos operadores binários.

As cláusulas são escritas junto à classe do objeto na forma de comentários da linguagem Java, iniciando por `/*?` e finalizando com `*/`. As cláusulas com as regras codificam todo o conhecimento que um objeto inteligente terá e irão compor sua base de conhecimento. Para otimizar o processo de extração e carga das regras nas bases de conhecimento dos objetos, cada classe de objetos inteligentes deve ser previamente processada pelo compilador de regras que construímos. Este irá extrair as regras da classe, verificar sua corretude quanto à sintaxe e semântica e adicioná-las numa estrutura de dados temporária e transiente, isto é, uma instância da classe `KnowlegeBase`. Após concluir a extração das regras de uma classe, o compilador copia a base de conhecimento da memória para o disco, tornando-a persistente, num arquivo objeto que possui o mesmo nome da classe.

Quando um objeto inteligente é instanciado pela primeira vez, a classe do objeto verifica se sua base de conhecimento já foi criada em memória. Caso ainda não tenha sido, esta busca por suas regras no arquivo objeto gerado pelo compilador e as carrega em sua base de conhecimento. A busca deve levar em consideração herança e polimorfismo e será abordada em maiores detalhes na próxima seção. Uma vez criada a base de conhecimento de uma classe de objetos, esta fica disponível para ser utilizada por outros objetos da mesma classe.

Para ilustrar o mecanismo de integração, consideremos a classe abaixo, a qual modela um triângulo isósceles[1] e a relação entre seus três lados:

```
public class isosceles extends IntelligentObject
{
   protected double lado1;
   protected double lado2;
   protected double lado3;
   protected double area;

   public double getBase()
   {
      if(lado1 == lado2)
         return lado3;
      else if (lado1 == lado3)
         return lado2;
      else return lado1;
   }

   /*?
      private mantemLado3 mantemLado3(lado1,lado2,lado3) :-
         if (lado1 == lado2) then (lado3 != lado1);
      private mantemLado2 mantemLado2(lado1,lado2,lado3) :-
         if (lado1 == lado3) then (lado2 != lado1);
      private mantemLado1 mantemLado1(lado1,lado2,lado3) :-
         if (lado2 == lado3) then (lado1 != lado2);
      public regra1 calculaArea(X) :- min = this.getBase(),
         altura = min/2,
         X = (min*altura)/2;
   */
}
```

[1] Um triângulo isósceles possui dois lados iguais.

As três primeiras regras definidas podem ser usadas em conjunto para garantir a principal propriedade de um triângulo isósceles, ou seja, que dois dos seus lados possuam tamanhos iguais e o outro tamanho diferente. Isto pode ser feito solicitando que o objeto verifique se as três condições estão sendo satisfeitas sempre que algum valor de um dos seus lados for alterado. A última regra calcula a área do triângulo utilizando um método do próprio objeto para saber qual dos lados é a base do triângulo. O valor calculado é retornado na variável lógica passada como argumento na cabeça da regra.

4.4.2 Criação da Base de Conhecimento

Regras são consideradas pseudo-métodos da linguagem e respeitam as principais características da orientação a objetos dos objetos aos quais o conhecimento é aplicado, isto é, encapsulamento, herança e polimorfismo. A maneira como cada uma destas propriedades é mantida está descrita a seguir.

Encapsulamento

Para eliminar qualquer violação de encapsulamento, o acesso a atributos e métodos numa regra deve obedecer os mesmos princípios que regem o acesso dos objetos a essas estruturas: apenas variáveis e métodos do objeto ao qual a regra esta associada ou variáveis públicas e métodos públicos de outros objetos podem ser usados.

Herança e Polimorfismo

Regras são métodos particulares e como tais devem participar de mecanismos de herança tal qual os métodos usuais fazem. Definir uma regra com um acesso privado é semelhante a definir um método privado em Java. Um método privado é visível na classe em que está declarado e, apesar de não acessível diretamente, é herdado pelas classes derivadas desta. Já uma regra privada está presente apenas na classe em que foi declarada, sendo inexistente para instâncias de qualquer subclasse desta. Regras públicas por sua vez equivalem exatamente a métodos declarados como públicos em Java, sendo herdadas pelas subclasses como qualquer método público. Apesar das regras poderem receber parâmetros, estas são consideradas como métodos sem argumentos e o polimorfismo neste caso será determinado apenas pelo nome da regra.

Para obtermos tal mecanismo de herança, a hierarquia de uma classe deve poder ser acessada em tempo de execução. Em outras palavras, um objeto deve ser capaz de responder mensagens como “qual é sua superclasse” e “qual é sua classe”. Isto é necessário para que seja possível determinar quais são as super-classes de uma classe inteligente e assim poder concatenar seu conhecimento com o herdado das super-classes, caso estas também sejam inteligentes e possuam regras públicas. Este tipo de informação está presente em ambientes de programação como Smalltalk e Java, mas não ainda no C++.

Formação do Conhecimento Orientado a Objetos

Tendo acesso a estas informações, a base de conhecimento de uma determinada classe de objetos será obtida da seguinte forma:

- Com posse do nome da classe, busca-se no disco o arquivo objeto com as regras desta classe e as insere na base de conhecimento.

- Depois, faz-se a união da base de conhecimento resultante com as bases de conhecimento das sucessivas superclasses, levando-se em consideração regras de visibilidade e polimorfismo.

O processo de junção com as bases de conhecimento das superclasses exclui as regras privadas, devido ao seu escopo limitado, neste esquema de herança de regras. As regras públicas por sua vez são adicionadas e são polimórficas. Isto quer dizer que podem ser sobrescritas numa classe derivada e portanto serão adicionadas no processo de junção somente se outra regra com o mesmo nome já não tenha sido.

4.4.3 Mecanismos de Raciocínio

Um objeto pode, quando precisar, disparar um processo de inferência que irá usar o conhecimento disponível para produzir informações. Tais informações poderão ser utilizadas pelo objeto para ajudá-lo a desempenhar suas funções, possivelmente criando comportamentos inteligentes. Para tanto, uma máquina de inferência deve ser usada pelo objeto inteligente. Nossa proposta não implica na implementação de um mecanismo de resolução de restrição, mas sim na integração com outros já existentes. Algumas das vantagens que esperamos obter com isso são:

- o usuário fica livre para escolher o mecanismo de resolução que mais se adeqüe às suas necessidades;
- apesar das regras codificadas para um mecanismo de resolução não funcionarem necessariamente em outro (as restrições podem ter nomes diferentes), esta conversão seria bastante simplificada já que a sintaxe padrão das regras é a mesma;
- a sintaxe padronizada da linguagem CLP facilita o aprendizado e utilização, liberando os usuários de terem que aprender as características de cada novo produto que forem utilizar.

Esta integração será feita através da implementação de um software intermediário, um *middleware* que faz a ligação entre o objeto inteligente e seu conhecimento e o mecanismo de resolução em questão. Sua principal função é traduzir o conhecimento do objeto, que está numa linguagem e formato incompreensíveis ao solucionador, para uma versão que este possa utilizar. Este *middleware* também é responsável por viabilizar a comunicação do objeto com o mecanismo de resolução, quando este precisar consultar seu conhecimento e obter as respostas resultantes. Para cada novo mecanismo de resolução de restrição que se deseja utilizar, um novo *middleware* deve ser construído, para então poder ser acoplado e utilizado pelo objeto inteligente.

Vamos utilizar como máquina de inferência o ECLiPSe [66] pois este provê uma das mais completas implementações de linguagem CLP e dispõe de facilidades de integração com linguagens como C++ e Java, o que tornará possível a implementação do *middleware* correspondente. O ECLiPSe é um sistema baseado no Prolog, cujo objetivo é servir como plataforma para integrar várias extensões da programação lógica, em particular a programação lógica com restrições.

Execução das Regras

As regras estão acopladas aos objetos como se fossem métodos da linguagem e de alguma forma precisam ser acessadas e executadas. Encontramos na literatura duas formas para tornar as regras compreensíveis pela linguagem hospedeira:

1. Uma delas é a pré-compilação destas para estruturas reconhecidas pela linguagem, como por exemplo, a tradução das regras em métodos, como é feito no Néopus [45] e RAL/C++ [21]. A utilização desta forma é feita em dois passos: primeiro a tradução das regras para elementos sintáticos da linguagem e então a compilação destes elementos em conjunto com a aplicação.

2. Outra forma é a tradução das regras para uma estrutura que possa ser compreendida pelo mecanismo de inferência e que este seja capaz de invocar as operações da linguagem descritas nestas estruturas (tal como acesso a atributos e chamadas a métodos). Esta é a implementação encontrada em sistemas como Jeops [20] e também a escolhida por Bomme.

Optamos pela segunda opção por ser a mais conveniente para implementarmos nossa solução como uma biblioteca Java que pode ser utilizada de forma transparente pelas aplicações. Assim, a estrutura a ser compreendida pelo motor de inferência é a base de conhecimento que, como vimos, será povoada com as regras CLP de cada objeto e então traduzida, com o auxilio do *middleware*, para um formato final compreensível ao mecanismo de inferência sendo utilizado. A base de conhecimento trará todas as informações necessárias para possibilitar esta conversão.

Uma outra necessidade é que o mecanismo de resolução seja capaz de invocar as operações da linguagem descritas na base de conhecimento, tais como acesso a atributos e chamadas a métodos. Contudo, todos os elementos que compõem uma regra estão armazenados de forma simbólica na base de conhecimento pois essa abordagem significa que não iremos fazer uma pré-compilação das regras para elementos sintáticos reconhecidos pela linguagem. Para que o mecanismo de inferência possa funcionar corretamente e para atingirmos nosso objetivo de que atributos e métodos do objeto possam ser manipulados dentro das regras é necessário que o mecanismo de inferência tenha acesso real a estas estruturas.

Este tipo de situação pode ser resolvida provendo algum grau de reflexão computacional à aplicação. A reflexão permite que criemos objetos de uma classe ou acessemos dados e métodos de um objeto referenciando apenas o nome destas estruturas. Os ambientes de execução de Java disponibilizam por padrão estes recursos de reflexão computacional.

Ativação do Conhecimento

Um objeto inteligente pode usar seu conhecimento submetendo uma consulta à máquina de inferência, a qual dispara um processo de encadeamento regressivo que irá tentar provar o que foi submetido. Para isto o objeto dispõe, como comentado anteriormente, de um método `ask()`, que recebe como parâmetro o predicado que forma a cabeça de alguma regra definida na base de conhecimento e uma lista de expressões (tal qual as expressões no corpo de uma regra). O predicado da cabeça unifica com algum predicado equivalente, ocorre a unificação dos termos de ambos e o processo de derivação começa, tentando provar a consulta em questão. A decisão de onde e quando acionar o método fica a cargo do programador.

4.5 Comentários Finais

Este trabalho é uma primeira iniciativa em integrar regras e objetos com regras que não sejam regras de produção. Os pontos fundamentais que consideramos para sua elaboração foram:

1. Organização do conhecimento orientado a objetos.

2. Facilidade de utilização da CLP no ambiente em que as aplicações científicas são desenvolvidas.

3. Independência de sistema CLP.

Seguindo a abordagem de considerar regras como pseudo-métodos, conseqüentemente equiparando-as aos métodos usuais, conseguimos uma organização e forma de utilização do conhecimento o mais próximo possível das características da orientação a objetos. Quando aplicada ao contexto de regras CLP, esta integração propicia um ambiente mais familiar ao programador, facilitando o aprendizado e a utilização da tecnologia. Esperamos que o fato das regras obedecerem a uma sintaxe padrão resulte em uma considerável flexibilidade para migrar todo o conhecimento codificado para outro mecanismo de resolução de restrição, no caso deste precisar ser alterado.

O resultado final é uma proposta de integração da tecnologia CLP com o paradigma da orientação a objetos apta a atender, de forma única, tanto aqueles que já fazem uso da CLP, através de linguagens e ambientes destinados unicamente a este propósito, como aos engenheiros de sistemas, que desenvolvem aplicações complexas dos mais variados tipos, nas quais todos os benefícios da CLP poderão ser utilizados.

CAPÍTULO 5

Implementação dos Objetos Inteligentes

5.1 Introdução

Neste capítulo apresentamos os resultados da implementação do projeto exposto no Capítulo 4 e sua aplicação prática na solução de três problemas. Conforme discutido no Capítulo 1, apesar da proposta inicial ter sido uma implementação em C++, optamos por criar um protótipo em Java, pois alguns dos requisitos essenciais para o sucesso do projeto, tais como a reflexão computacional e serialização, já estão disponíveis nativamente nas API's (interface de programação da aplicação, ou *application programming interface*) padronizadas da linguagem. Como queremos inicialmente validar nossa idéia e a utilização de outra linguagem de programação que também provê suporte a orientação a objetos não altera em nada as especificações do projeto, esta decisão não desabona os resultados aqui obtidos.

A Figura 5.1 ilustra a arquitetura geral da implementação do projeto dos objetos inteligentes.

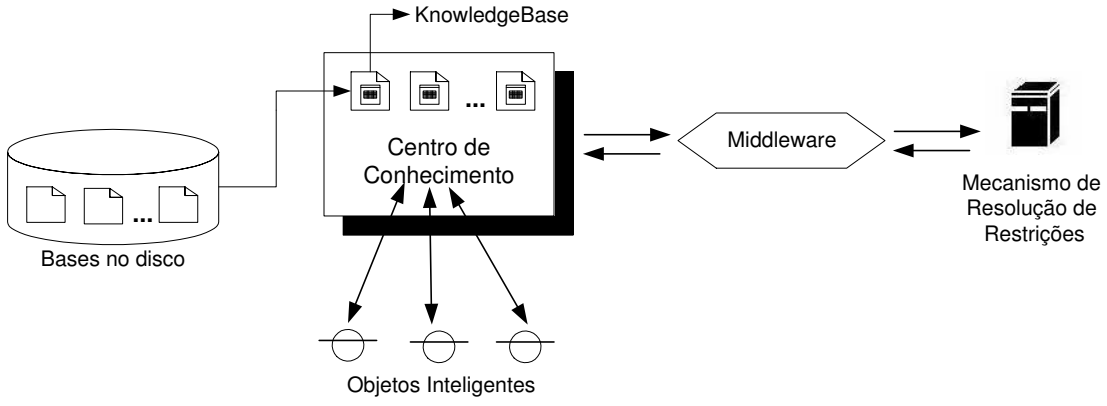

Figura 5.1: Arquitetura da implementação.

O centro de conhecimento atua como um repositório de todas as bases de conhecimento da aplicação e é responsável pelas operações de persistência e recuperação dessas bases.

Estas operações são utilizadas pelo compilador de regras e pelos objetos inteligentes quando da criação de sua base de conhecimento. Quando a base de conhecimento de um objeto inteligente é criada a partir de sua representação em disco, esta é armazenada no centro de conhecimento. Quando uma determinada base de conhecimento é requerida para ser usada na aplicação, o centro de conhecimento é responsável por fornecê-la. O centro de conhecimento mantém também uma referência para o *middleware* utilizado pela aplicação, responsável pelo mapeamento de regras e consultas para um mecanismo de inferência, bem como disponibilização dos resultados para a aplicação.

A Figura 5.2 mostra um diagrama de classes UML [44] com a hierarquia das principais classes da implementação.

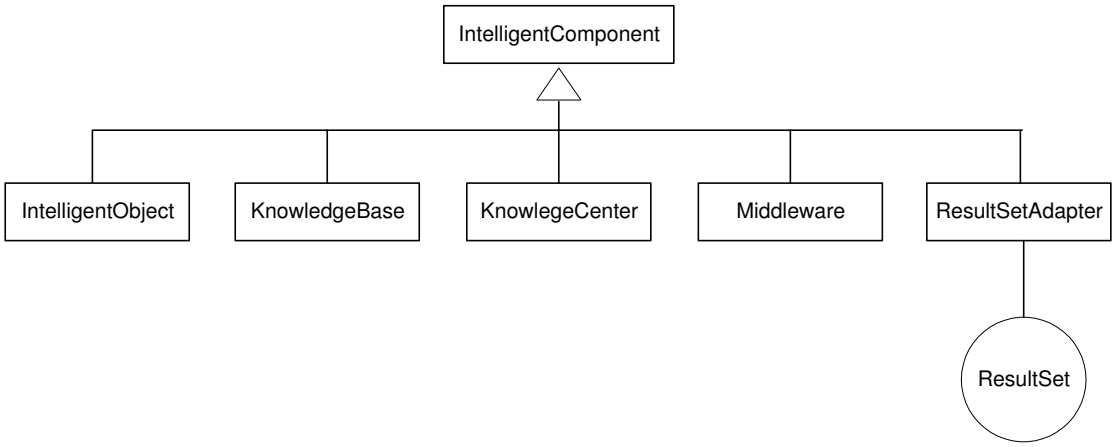

Figura 5.2: Principais classes da implementação.

Conforme introduzido no Capítulo 4, a classe `IntelligentObject` é a classe base de toda classe de objeto inteligente e a classe `KnowledgeBase` representa uma base de conhecimento. `KnowledgeCenter` representa o centro de conhecimento da aplicação. `Middleware` é a classe abstrata que define a funcionalidade padrão de um *middleware*. A classe `ResultSetAdapter` é uma implementação padrão da interface `ResultSet` e representa o conjunto de soluções fornecidas após um processamento. Todos estes componentes são derivados de `Intelligent-Component`, cuja principal utilidade é servir de classe base para as demais, provendo uma raiz comum à hierarquia de classes da aplicação.

A Seção 5.2 detalha a implementação dos objetos inteligentes e todos os aspectos relacionados à base de conhecimento: codificação das regras na classe de objetos e sua extração e validação pelo compilador de regras, persistência e recuperação da base no disco, criação da base respeitando-se os conceitos de herança e polimorfismo apresentados. Na Seção 5.3 mostramos as principais características do *middleware* para interfaceamento com o ECLiPSe. Inicialmente enfocamos a questão da tradução da base de conhecimento para o ECLiPSe, uma das principais funções atribuídas ao *middleware*. Depois explicamos como funciona o processo de raciocínio, isto é, a interação entre objeto inteligente, *middleware* e mecanismo de inferência. A seção 5.4 aborda a implementação das operações disponibilizadas pelo centro de conhecimento utilizadas pelos diversos componentes da aplicação. Primeiramente explicamos os métodos relacionados à persistência e recuperação de uma base de conhecimento e depois aqueles destinados ao processo de raciocínio. Por último, na Seção 5.5 exemplificamos a utilização da solução na resolução de três problemas práticos.

5.2 Objeto Inteligente e a Base de Conhecimento

Um objeto inteligente é uma instância de uma classe derivada direta ou indiretamente da classe abstrata `IntelligentObject`, listada a seguir.

```
1    abstract public class IntelligentObject
2       extends IntelligentComponent
3    {
4       private KnowledgeBase kb;
5
6       public IntelligentObject() throws IntelligenceException
7       {
8          kb = KnowledgeCenter.getKnowledgeBase(getClass().getName());
9       }
10      final public KnowledgeBase getKnowledgeBase()
11      {
12         return kb;
13      }
14      synchronized public ResultSet ask(String query)
15         throws IntelligenceException
16      {
17         return KnowledgeCenter.ask(this, query);
18      }
19      synchronized public ResultSet ask(Term[] terms)
20         throws IntelligenceException
21      {
22         return KnowledgeCenter.ask(this, terms);
23      }
24   }
```

O único atributo de instância declarado na classe é `kb`, linha 4, uma referência para um objeto da classe `KnowlegeBase`, a base de conhecimento do objeto inteligente. Os métodos da classe são:

- `IntelligentObject()`, linha 6: construtor da classe. Realiza a busca pelas regras e as insere na base de conhecimento `kb`.
- `ResultSet ask(String query)`, linha 14: dispara um processo de raciocínio. A questão é passada como uma string que é compilada e submetida ao *middleware*. O retorno é um objeto que implementa a interface `intelligentobject.ResultSet`, a qual será abordada, juntamente com o processo de raciocínio, na Seção 5.4.
- `ResultSet ask(Term[] terms)`, linha 19: também inicia um raciocínio, porém, a questão é passada aqui como uma lista de termos. Um termo é uma expressão formada pela combinação de elementos da linguagem de regras como predicados, átomos, listas, variáveis, constantes, atributos e métodos, conectados entre si através dos operadores binários.

O conhecimento de um objeto que deriva de `IntelligentObject` pode estar codificado em qualquer parte de uma classe Java entre as marcas `/*?` e `*/`, seguindo a sintaxe da gramática descrita no Capítulo 4. Uma vez que todos os objetos que serão inteligentes foram identificados e seu conhecimento inserido, a aplicação deve ser compilada pelo compilador de regras. Trata-se de um compilador da linguagem Java que foi estendido para também compilar as regras CLP. As funções desempenhadas por este são:

1. Compilar a aplicação Java e gerar seu *bytecode* (arquivos `.class`).
2. Extrair as regras das classes que estendem de `IntelligentObject` e verificar sua sintaxe.

3. Montar a base de conhecimento para cada uma das classes e persistí-la no disco. Para cada classe inteligente será gerado um arquivo, com o mesmo nome da classe e extensão `.kb`, contendo suas regras.

4. Emitir as mensagens de erros apropriadas em cada etapa de execução.

Uma base de conhecimento é um objeto da classe `KnowlegeBase`, cujos principais métodos e atributos estão listados a seguir.

```
25  abstract public class KnowledgeBase
26     extends IntelligentComponent
27  {
28     private String className;
29     private String baseClassName;
30     private TreeMap clauses = new TreeMap();
31     static String rootKBClassName =
32          "intelligentobject.IntelligentObject";
33
34     public KnowledgeBase(String className, String baseClassName)
35     {
36        this.baseClassName = baseClassName;
37        this.className = className;
38        KnowledgeCenter.registerKnowledgeBase(this);
39     }
40     public KnowledgeBase(String className)
41     {
42        this(className, rootKBClassName);
43     }
44     final public void addClause(Clause clause)
45     {
46        String clauseName = clause.getName();
47        clauses.put(clauseName, clause);
48     }
49     final public Clause getClause(String clauseName)
50     {
51        return (Clause)clauses.get(clauseName);
52     }
53     final public String getClassName()
54     {
55        return className;
56     }
57     final public int getNumberOfClauses()
58     {
59        return clauses.size();
60     }
61     final void bindingBaseClassClauses()
62        throws IntelligenceException
63     {
64        KnowledgeBase kb =
65           KnowledgeCenter.getKnowledgeBase(baseClassName);
66        for (ClauseIterator cit = new ClauseIterator(kb.clauses);
67             cit.hasNext();)
68        {
69           Clause clause = cit.next();
70           String clauseName = clause.getName();
71           if (clause.getAccessModifier() == Clause.PUBLIC)
72           if (!clauses.containsKey(clauseName))
73              clauses.put(clauseName, clause);
74        }
75     }
76  }
```

A classe declara as variáveis de instância `className`, linha 28, e `baseClassName`, linha 29, que armazenam, respectivamente, o nome da classe inteligente que contém esta base e o nome da sua superclasse, bem como uma coleção de cláusulas `clauses`, linha 30. Declara ainda uma variável estática `rootKBClassName` que possui o nome padrão para a classe que representa a classe base de todas as classes inteligentes na aplicação. Seus métodos são:

- `KnowledgeBase(String className, String baseClassName)`, linha 34: construtor da classe. Armazena o nome da classe e da sua classe base e utiliza a classe `KnowlegeCenter` para se registrar no centro de conhecimento.
- `KnowledgeBase(String className)`, linha 40: construtor da classe. Invoca o construtor acima passando como parâmetro para classe base o nome da classe padrão armazenada em `rootKBClassName`.
- **`void`** `addClause(Clause clause)`, linha 44: adiciona uma cláusula à coleção.
- `Clause getClause(String clauseName)`, linha 49: retorna a cláusula cujo nome equivale ao recebido em `clauseName`.
- `String getClassName()`, linha 53: retorna o nome da classe.
- **`int`** `getNumberOfClauses()`, linha 57: retorna o número de cláusulas na coleção.
- **`void`** `bindingBaseClassClauses()`, linha 61: é o principal método da classe. Sua utilização e funcionamento serão vistos na Seção 5.4 quando estivermos tratando da criação de uma base de conhecimento.

Uma cláusula possui uma regra que é formada por classes que representam os não-terminais da linguagem de regras. A estrutura completa de uma cláusula pode ser vista no diagrama de classes UML da Figura 5.3. Estas classes são manipuladas internamente pelo compilador de regras e não serão descritas aqui em maiores detalhes.

5.3 Utilização do Conhecimento através do Middleware

Um *middleware* é uma camada de software que faz a comunicação entre duas tecnologias diferentes. Neste trabalho, o *middleware* deve tornar viável a comunicação entre um objeto inteligente e um mecanismo de inferência arbitrário. Para isso ele desempenha duas atividades principais:

- converte o conhecimento do objeto para que este possa ser usado pelo mecanismo em questão; e
- trata as chamadas do objeto ao mecanismo de inferência, bem como a resposta deste ao objeto.

Apesar destas operações ocorrerem em conjunto, para maior clareza iremos apresentá-las em separado. Primeiramente vamos mostrar os aspectos do *middleware* relacionados com a tradução da base de conhecimento e como implementamos isto no *middleware* que construímos para acessar o ECLiPSe. Em seguida, explicamos o funcionamento do método `ask()` e a interface `ResultSet`, os componentes para ativação do raciocínio.

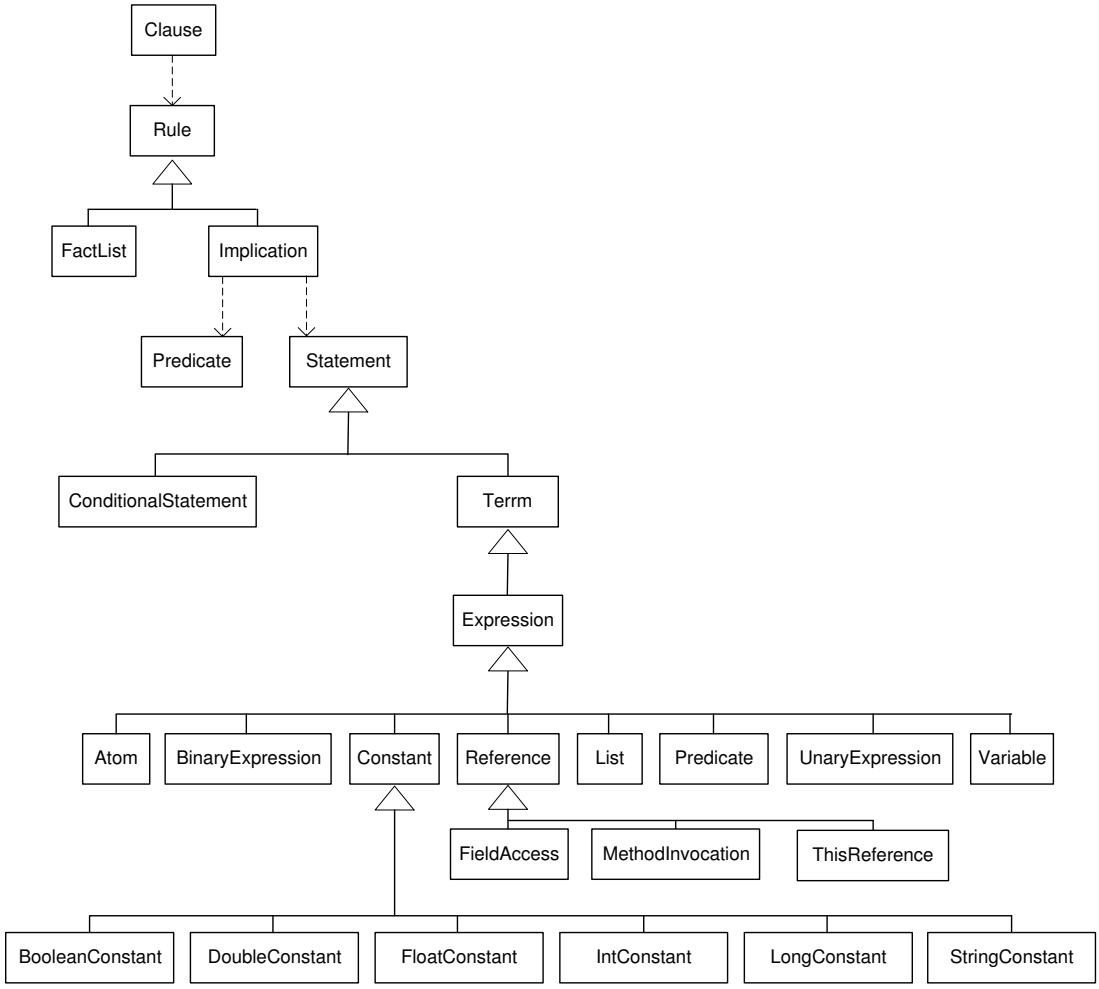

Figura 5.3: Hierarquia de classes para uma cláusula.

5.3.1 Tradução do Conhecimento

A classe `Middleware` é uma classe abstrata que provê a interface padrão para a construção de um *middleware*. Os métodos concretos e abstratos que tratam da tradução de uma base de conhecimento estão descritos abaixo.

```
77   abstract public class Middleware
78      extends IntelligentComponent
79   {
80      protected void translateKnowledgeBase(KnowledgeBase kb)
81         throws IntelligenceException
82      {
83         for (ClauseIterator cit = kb.clauseIterator();
84               cit.hasNext();)
85            translateClause(cit.next());
86      }
87      abstract public void translatePredicate(Predicate p)
88         throws IntelligenceException
89      abstract public void translateTerm(Term t)
90         throws IntelligenceException
91      abstract public void translateStatementList(StatementList sl)
92         throws IntelligenceException
93      abstract public void translateStatement(Statement s)
94         throws IntelligenceException
95      abstract public void translateList(List l)
```

```
96        throws IntelligenceException
97     abstract public void translateConstant(Constant c)
98        throws IntelligenceException
99     abstract public void translateAtom(Atom a)
100       throws IntelligenceException
101    abstract public void translateConditionalStatement(
102          ConditionalStatement cs)
103       throws IntelligenceException
104    abstract public void translateExpression(Expression e)
105       throws IntelligenceException
106    abstract public void translateBinaryExpression(BinaryExpression be)
107       throws IntelligenceException
108    abstract public void translateUnaryExpression(UnaryExpression ue)
109       throws IntelligenceException
110    abstract public void translateVariable(Variable v)
111       throws IntelligenceException
112    abstract public void translateReference(Reference r)
113       throws IntelligenceException
114 }
```

O único método concreto é o `translateKnowledgeBase()`, linha 80, que executa o que provavelmente sempre será necessário ser feito de início, iterar sobre a coleção de cláusulas para realizar a tradução das mesmas. Este é o método inicial para conversão de uma base de conhecimento e veremos quando este é efetivamente invocado mais adiante. Os outros são todos métodos abstratos que devem ser implementados na classe derivada e refletem o comportamento padrão do *middleware*. Basicamente foi criado um método capaz de traduzir cada estrutura da gramática de forma adequada.

O Middleware para o ECLiPSe

O *middleware* para o ECLiPSe está implementado na classe `EclipseMiddleware`, que deriva de `Middleware`. Abaixo segue sua definição contemplando os métodos vistos acima.

```
115 final public class EclipseMiddleware
116    extends Middleware
117 {
118    native protected void translateKnowledgeBase(KnowledgeBase kb)
119       throws IntelligenceException;
120    public void translatePredicate(Predicate p)
121       throws IntelligenceException
122    {
123       //Do nothing
124    }
125    ...
126    public void translateReference(Reference r)
127       throws IntelligenceException
128    {
129       //Do nothing
130    }
131    static
132    {
133       System.loadLibrary("middleware");
134    }
135 }
```

Neste ponto, temos que fazer alguns comentários sobre o ECLiPSe e sobre as decisões que tomamos e influenciaram a implementação do projeto. O ECLiPSe disponibiliza algumas interfaces que tornam possível utilizá-lo com outras linguagens de programação e este foi um dos fatores que nos levou a escolhê-lo como mecanismo de resolução. Apesar de existir uma

interface Java que possibilita a comunicação entre ambos, a qual num primeiro momento pareceu ser a mais adequada para nossas necessidades, constatamos que a interface C++ provê melhores recursos para integração e decidimos utilizá-la. Para que a solução em Java pudesse acessar um *middleware* escrito em C++ e vice-versa, utilizamos extensivamente a biblioteca JNI (*Java Native Interface*) [57] do Java. Esta biblioteca possibilita que se invoque métodos C++ dentro de objetos Java e também que se acesse objetos Java dentro destes métodos, enviados a estes como seus argumentos.

Pode-se observar que na classe `EclipseMiddleware` o método `translateKnowledgeBase()` foi sobreescrito por outro, especificado com o modificador **`native`**, e que todos os métodos abstratos foram implementados apenas pró-forme, sem nenhum código neles. Um método descrito como nativo indica que seu código está contido numa biblioteca em C (ou C++), que é carregada através da invocação do método estático `System.loadLibrary()`. Dessa forma, a implementação do método `translateKnowledgeBase()`, bem como de todos os outros necessários para realizar a tradução do conhecimento, está contida na biblioteca `middleware`, carregada na linha 133. Todas as funções desta biblioteca utilizam os recursos da API JNI para manipular os objetos Java e não a descreveremos aqui por conta do seu tamanho e complexidade.

A Reflexão Computacional

Como visto anteriormente, tudo que foi escrito nas regras CLP está armazenado de forma simbólica na base de conhecimento e para se ter acesso real a atributos e métodos de dentro das regras faz-se necessária a reflexão computacional. Em Java, todo objeto existente numa aplicação possui uma referência a uma instância da classe `Class`, que é criada automaticamente pela JVM (*Java Virtual Machine*) quando a classe de um objeto é carregada. Um objeto `Class` é uma *meta-classe* de uma classe Java, mantendo, em tempo de execução, informações a respeito de sua superclasse e de todos seus construtores, atributos e métodos públicos.Tais informações possibilitam as funcionalidades providas pela reflexão. Vejamos como isso funciona analisando as classes `MethodInvocation` e `FieldAccess` (vide diagrama da Figura 5.3), que representam respectivamente chamadas a métodos e acesso a atributos do objeto.

- `MethodInvocation`: esta classe da gramática armazena as informações referentes à chamada de um método em Java. Ela contém o nome do método e uma lista com seus parâmetros, a qual pode conter elementos das classes `Constant` ou `Reference`. Constantes são valores literais enquanto uma referência pode ser uma chamada a método ou acesso a algum atributo. Nesta lista de parâmetros está disponível para cada um deles a informação sobre seu tipo e valor, graças ao compilador de regras que foi embutido num compilador completo da linguagem Java. Utilizamos a reflexão do Java da seguinte maneira:
  - Obtemos a referência para a classe `Class` que representa a classe do objeto inteligente para o qual desejamos invocar algum método. Para isso usamos a referência ao objeto inteligente disponível e enviamos a este a mensagem `getClass()`. Invocamos então o método `Class.getDeclaredMethod()` o qual recebe como parâmetro uma `String` com o nome do método e um vetor com os tipos dos parâmetros.
  - Este retorna um objeto da classe `java.lang.reflect.Method` a qual define o método `invoke()` para execução do método representado pelo objeto..
  - O método `invoke()` deve receber o objeto para o qual o método está sendo invocado e uma lista com os valores dos seus argumentos.

Assim temos que todas as informações necessárias à execução de um método de forma reflexiva encontra-se na classe `MethodInvocation`. Quando o *middleware* traduz esta classe ele passa ao ECLiPSe a estrutura apropriada para a execução do método e este, no momento exato em que a encontra durante o processamento de uma regra, faz a invocação do método do objeto.

- `FieldAccess`: o acesso a um atributo do objeto funciona de forma similar à invocação de um método. A classe `FieldAccess` armazena o nome do campo e sabe responder qual é seu tipo. A reflexão aqui ocorre da seguinte forma:
  - Após obter uma referência à classe `Class` que representa a classe do objeto inteligente para o qual desejamos acessar algum atributo, invocamos o método `Class.getDeclaredField()`, o qual recebe como parâmetro uma `String` com o nome do atributo.
  - Este retorna um objeto da classe `java.lang.reflect.Field` o qual possui uma série de métodos tanto para obter como para ajustar um valor para o campo.

  Da mesma forma aqui, o *middleware* traduz a classe criando a estrutura apropriada para passar ao ECLiPSe. No momento em que este precisar usar o valor do atributo esta estrutura é acionada e obtém, através da reflexão, o valor corrente do atributo no objeto.

5.4 Centro de Conhecimento

O centro de conhecimento de uma aplicação é representado pela classe `KnowledgeCenter`. A classe não possui instância mas declara métodos estáticos utilizados para persistência e recuperação de uma base de conhecimento e para invocação do raciocínio junto ao `middleware`. Estes métodos estão listados na Tabela 5.1 e são comentados a seguir.

Classe `KnowledgeCenter`
Métodos públicos
`static` `ResultSet ask(IntelligentObject io, String query)` Dispara um processo de raciocínio.
`static` `ResultSet ask(IntelligentObject io, Term[] terms)` Dispara um processo de raciocínio.
`static void` `saveKnowledgeBase(KnowledgeBase kb)` Grava a base de conhecimento `kb` no disco.
`static` `KnowledgeBase getKnowledgeBase(String className)` Retorna a base de conhecimento da classe `className`.
Métodos privados
`static` `KnowledgeBase loadKnowledgeBase(String className)` Recupera a base de conhecimento da classe `className` do disco.
`static` `File getKBFileFor(String className)` Retorna o objeto `File` que contém a base de conhecimento da classe `className`.

Tabela 5.1: Classe `KnowledgeCenter`.

5.4.1 Persistência da Base de Conhecimento

O esquema de persistência é implementado utilizando-se os recursos de serialização do Java, os quais possibilitam a representação de qualquer objeto (cuja classe implementa a interface

java.io.Serializable [58]) em um fluxo de bytes. É este fluxo que é armazenado em disco e posteriormente recuperado para restauração do objeto. A classe KnowledgeBaseStreamer, listada a seguir, declara métodos estáticos responsáveis por gravar e recuperar uma base de conhecimento para e de um arquivo em disco.

```
136   public class KnowledgeBaseStreamer
137     extends IntelligentComponent
138  {
139     static public void save(KnowledgeBase kb, File file)
140       throws IntelligenceException
141     {
142       try
143       {
144          ObjectOutputStream os = openOutputStream(file);
145          os.writeObject(kb);
146          os.flush();
147          os.close();
148       }
149       catch (IOException e)
150       {
151          System.out.println(e.getMessage());
152          error("Unable to save knowledge base into file "+
153          file.getName());
154       }
155     }
156     static private ObjectOutputStream openOutputStream(File file)
157       throws IOException
158     {
159       return new ObjectOutputStream(new FileOutputStream(file));
160     }
161      static KnowledgeBase load(File file)
162       throws IntelligenceException
163     {
164       KnowledgeBase kb = null;
165       try
166       {
167          ObjectInputStream is = openInputStream(file);
168          kb = (KnowledgeBase)is.readObject();
169          if (!kb.baseClassName.equals(KnowledgeBase.rootKBClassName))
170             kb.bindingBaseClassClauses();
171          KnowledgeCenter.registerKnowledgeBase(kb);
172          is.close();
173       }
174       catch (IOException e)
175       {
176          System.out.println(e.getMessage());
177          error("Unable to load knowledge base into file "+
178          file.getName());
179       }
180       catch (ClassNotFoundException e)
181       {
182          error("Knowledge base file "+ file.getName() + "corrupted");
183       }
184       return kb;
185     }
186     static private ObjectInputStream openInputStream(File file)
187       throws IOException
188     {
189       return new ObjectInputStream(new FileInputStream(file));
190     }
191  }
```

Seus métodos são:

- **void** `save(KnowledgeBase kb, File file)`, linha 139: recebe como argumentos a base de conhecimento a ser salva e uma referência ao arquivo no disco que irá contê-la. É aberto então um fluxo de saída para este arquivo, linha 144, e o conjunto de bytes na memória que representam a base são copiados para este arquivo, linha 145.
- `ObjectOutputStream openOutputStream(File file)`, linha 156: abre um fluxo de saída que aceita receber um conjunto de bytes e os envia para algum repositório, neste caso um arquivo no sistema de arquivos da máquina representado pelo argumento `file`.
- `KnowledgeBase load(File file)`, linha 161: realiza o processo inverso ao `save()`, isto é, carrega uma base de conhecimento armazenada no arquivo `file` recebido como argumento para a memória. Mais detalhes sobre o funcionamento deste método será visto em seguida no processo de recuperação de uma base de conhecimento.
- `ObjectInputStream openInputStream(File file)`, linha 186: abre um fluxo de entrada que aceita receber um conjunto de bytes e os envia para algum repositório, neste caso um arquivo no sistema de arquivos da máquina representado pelo argumento `file`.

Para gravar uma base de conhecimento no disco o compilador de regras invoca o método `saveKnowledgeBase()`, linha 193, da classe `KnowledgeCenter`, passando a base a ser persistida. Este por sua vez invoca o método `save()` da classe `KnowledgeBaseStreamer`, passando, além da base, um objeto do tipo `File` que representa uma referência ao arquivo a ser criado. Estes passos estão ilustrados no diagrama de seqüencia UML da Figura 5.4.

```
192  // Método KnowledgeCenter.saveKnowledgeBase
193   static public void saveKnowledgeBase(KnowledgeBase kb)
194     throws IntelligenceException
195  {
196     KnowledgeBaseStreamer.save(kb, getKBFileFor(kb.getClassName()));
197  }
198
199  // Método KnowledgeCenter.getKBFileFor
200  static private File getKBFileFor(String className)
201  {
202     return new File(className + ".kb");
203  }
```

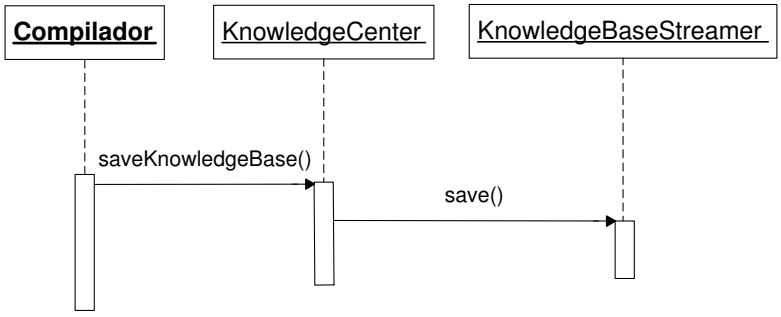

Figura 5.4: Diagrama de seqüência para gravar uma base de conhecimento no disco.

O processo de recuperação de uma base de conhecimento, ilustrado no diagrama de seqüência da Figura 5.5, ocorre da seguinte maneira:

1. Quando um objeto inteligente é instanciado na aplicação, seu construtor é executado chamando o método `getKnowledgeBase()` (linha 205) da classe `KnowledgeCenter`, o qual recebe o nome da classe do objeto como argumento.

2. O método `getKnowledgeBase()` primeiramente verifica se a base para a classe em questão já não foi carregada (linha 209). As bases já carregadas ficam num objeto da classe `java.util.TreeMap` denominado `knowledgeBases`, pertencente a `KnowledgeCenter`. Esta estrutura mapeia o nome de uma classe com sua base de conhecimento. Caso a base ainda não exista, o método `loadKnowledgeBase()` é chamado para obtê-la (linha 210) e então, caso a encontre, esta é acrescentada na `knowledgeBases` e sua referência retornada para o objeto (linhas 211 e 212).

```
204  // Método KnowledgeCenter.getKnowledgeBase
205   static public KnowledgeBase getKnowledgeBase(String className)
206     throws IntelligenceException
207  {
208     KnowledgeBase kb;
209     if ((kb = (KnowledgeBase)knowledgeBases.get(className)) == null)
210        if ((kb = loadKnowledgeBase(className)) != null)
211           knowledgeBases.put(className, kb);
212     return kb;
213  }
```

3. O método `loadKnowledgeBase()` (linha 215) cria uma referência para o arquivo da base de conhecimento (linha 218) e o repassa como parâmetro para o método `load` da classe `KnowledgeBaseStreamer` (linha 161). Este método abre o arquivo e utiliza os recursos de serialização do Java para restaurar a base num objeto em memória (linhas 167 e 168). Contudo, o processo de recuperação não pára por aí. É preciso ainda implementar o mecanismo de herança e polimorfismo. Na linha 169 verifica-se se a classe base do objeto sendo criado é do tipo definido na variável `KnowledgeBase.rootKBClassName`, ou seja, do tipo `IntelligentObject`. Caso não seja, significa que sua classe deriva de outra classe inteligente, sendo necessário então levar em consideração as regras contidas nesta superclasse. Isto é feito na linha 170 invocando-se o método `bindingBaseClassClauses()`, definido na linha 61, classe `KnowledgeBase`. O método `bindingBaseClassClauses()` inicia a recursão do processo invocando novamente `getKnowledgeBase()` (linha 65) da `KnowledgeCenter`. Somente quando um objeto que deriva diretamente de `IntelligentObject` for encontrado é que a recursão termina e as bases vão sendo montadas (quando a linha 169 do método `load()` retorna **`true`**). Primeiramente cria-se a base do objeto no n-ésimo nível da recursão, depois cria-se a do nível n-1 acrescentando na coleção de cláusulas desta base as do nível anterior. Serão acrescentadas apenas as cláusulas públicas (linha 71) e cujo nome difere de todas as cláusulas já existentes na base (linha 72). Quando este processo termina, todas as bases existentes na hierarquia de classes inteligentes do objeto chamador foram construídas e encontram-se disponíveis no centro de conhecimento.

```
214  // Método KnowledgeCenter.loadKnowledgeBase
215   static private KnowledgeBase loadKnowledgeBase(String className)
216     throws IntelligenceException
217  {
218     File file = getKBFileFor(className);
219     if (!file.exists())
220        error("Unable to find knowledge base file "+ file.getName());
221     return KnowledgeBaseStreamer.load(file);
222  }
```

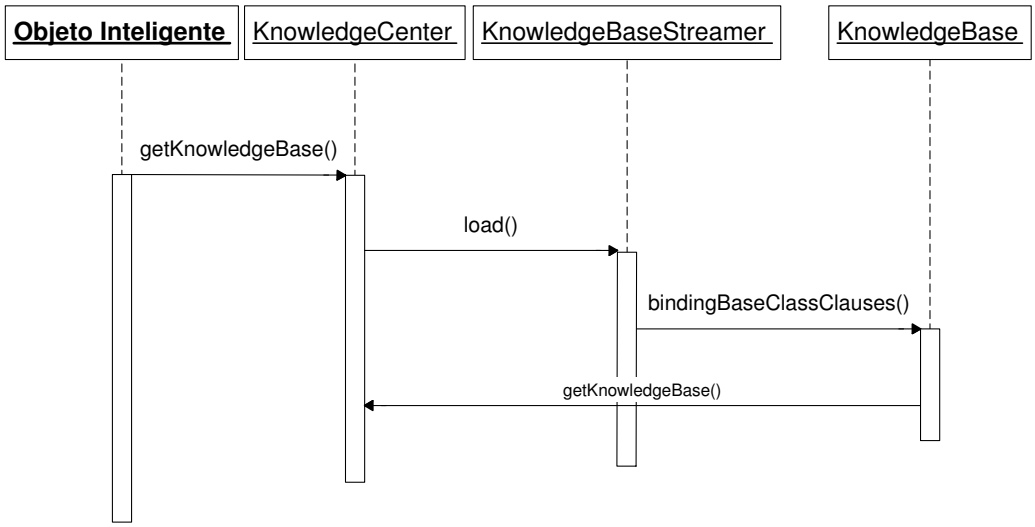

Figura 5.5: Diagrama de seqüência para restaurar uma base de conhecimento do disco.

5.4.2 Ativação do Raciocínio

O processo de ativação do raciocínio está ilustrado no diagrama de seqüência da Figura 5.6.

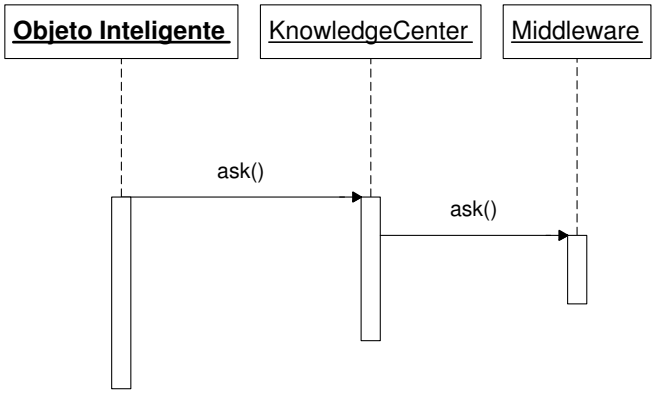

Figura 5.6: Diagrama de seqüência para ativação do raciocínio.

O processo inicia quando um objeto inteligente invoca um dos seus métodos `ask()`. Mostraremos a utilização do método que recebe a pergunta como uma string (linha 14), pois este provavelmente será o mais utilizado. Este método por sua vez invoca o método `ask()` (linha 224) da classe `KnowledgeCenter`, o qual recebe uma referência a um objeto inteligente e a string com a questão como parâmetros. A referência ao objeto que está utilizando seu conhecimento naquele momento é necessária para que o *middleware* possa selecionar a base de conhecimento corretamente e, mais importante ainda, para que o esquema de reflexão possa ser utilizado. O `KnowledgeCenter` então instancia um novo compilador de regras (linha 227) que irá transformar a pergunta num vetor de termos e então invoca efetivamente o método `ask()` (linha 230) da classe `Middleware`.

```
223  // Método KnowledgeCenter.ask
224  static public ResultSet ask(IntelligentObject io, String query)
225     throws IntelligenceException
226  {
```

```
227     return middleware.ask(io, new QueryParser().parse(io, query));
228  }
```

Neste momento é que a base de conhecimento do objeto é traduzida e encaminhada ao mecanismo de resolução através do método `postKnowledgeBase()` (linha 233). Em seguida a questão é submetida com o método `postQuery()` (linha 234).

```
229  // Middleware.ask
230  public ResultSet ask(IntelligentObject io, Term[] terms)
231     throws IntelligenceException
232  {
233     postKnowledgeBase(io);
234     return postQuery(io, terms);
235  }
```

O método `postKnowledgeBase()` (linha 237) verifica se a base para a classe daquele objeto já não foi traduzida e, caso ainda não tenha sido, realiza a tradução. Os métodos abstratos `isPosted()` e `setPosted()` devem ser implementados na classe específica do *middleware* pois suas ações variam com o mecanismo de inferência sendo utilizado. Na classe `EclipseMiddleware` eles são definidos como nativos, assim como `translateKnowledgeBase()`. Dependendo da implementação do *middleware*, a tradução de uma base de conhecimento pode ser feita toda vez que um objeto solicitar um raciocínio ou apenas na primeira vez que isso ocorrer. No nosso *middleware* em C++ isso ocorre apenas uma vez, pois este guarda as bases já traduzidas num repositório.

```
236  // Middleware.postKnowledgeBase
237  protected void postKnowledgeBase(IntelligentObject io)
238     throws IntelligenceException
239  {
240     KnowledgeBase kb = io.getKnowledgeBase();
241     if (isPosted(kb))
242        return;
243     translateKnowledgeBase(kb);
244     setPosted(kb);
245  }
```

O método `postQuery()` (linha 247) invoca dois métodos: `translateQuery()`, para traduzir a questão e submetê-la ao solucionador, e `resume()`, para de fato disparar o mecanismo de inferência. Ambos também são definidos como nativos na classe `EclipseMiddleware`.

```
246  // Middleware.postQuery
247  protected ResultSet postQuery(IntelligentObject io, Term[] terms)
248     throws IntelligenceException
249  {
250     translateQuery(terms);
251     return resume(io);
252  }
```

A Interface ResultSet e a Classe ResultSetAdapter

Tendo em vista que a maioria dos problemas resolvidos por CLP são aqueles que requerem busca num espaço de soluções (como os problemas CSP's) e que geralmente existe mais de uma, criamos uma estrutura que representa o conjunto de respostas obtidas num processamento.

Na interface `intelligentobject.ResultSet`, listada a seguir, encontram-se os métodos necessários à navegação nesta estrutura bem como àqueles utilizados para recuperação dos

valores das variáveis envolvidas. O método `next()` pode ser utilizado para iterar no conjunto de respostas, verificando, sempre que invocado, se há alguma nova solução disponível. Quando existir, este recupera os valores das variáveis lógicas de resposta retornados pelo mecanismo de resolução de restrições, converte cada um deles no tipo Java apropriado e retorna verdadeiro. Estes valores podem então ser obtidos através dos métodos `get's` disponibilizados pela interface, os quais tomam como argumento o nome da variável lógica. Se é previsível o tipo de dado que se espera receber numa variável, então é possível utilizar os métodos específicos como `getBoolean()`, `getDouble()`, `getInt()`, para recuperar valores atômicos, ou o `getList()`, quando o retorno for uma lista de valores. Todavia, se o tipo de dado a ser retornado numa variável é incerto, pode-se usar o método `getObject()`, o qual retorna o resultado num objeto da classe `java.lang.Object`.

```
253 public interface ResultSet
254 {
255    public boolean next()
256        throws IntelligenceException;
257    public boolean getBoolean(String variableName)
258        throws IntelligenceException;
259    public double getDouble(String variableName)
260        throws IntelligenceException;
261    public float getFloat(String variableName)
262        throws IntelligenceException;
263    public int getInt(String variableName)
264        throws IntelligenceException;
265    public long getLong(String variableName)
266        throws IntelligenceException;
267    public ResultsetMetaData getMetaData()
268        throws IntelligenceException;
269    public Object getObject(String variableName)
270        throws IntelligenceException;
271    public String getString(String variableName)
272        throws IntelligenceException;
273    public java.util.List getList(String variableName)
274        throws IntelligenceException;
275    public void close()
276        throws IntelligenceException;
277 }
```

A classe `ResultSetAdapter` implementa esta interface bem como o repositório que armazena as soluções. Além do método de navegação e dos de obtenção de valores, existe um para esvaziar a estrutura (`clear()`) e outro para inserir valores (`put()`).

```
278 abstract public class ResultSetAdapter
279    extends IntelligentComponent
280    implements ResultSet
281 {
282    protected TreeMap variables = new TreeMap();
283
284    final protected void clear()
285    {
286        variables.clear();
287    }
288    final protected void put(String variableName, Object value)
289    {
290       variables.put(variableName, value);
291    }
292    ...
293 }
```

5.5 Exemplos

Nas seções anteriores descrevemos a implementação e funcionalidade dos principais componentes da arquitetura do sistema de objetos inteligentes. Nesta seção, apresentamos três aplicações que fazem uso de objetos inteligentes para resolverem, na prática, problemas do mundo real. O primeiro deles é o clássico problema de busca das n-rainhas, abordado no Capítulo 3. O segundo é um problema geométrico combinatório, enquanto o terceiro, extraído da computação gráfica, envolve a renderização de objetos poligonais em duas dimensões.

5.5.1 O Problema das N-Rainhas

O problema das n-rainhas consiste em posicionar n rainhas em um tabuleiro de xadrez n × n de tal forma que qualquer rainha não ataque nenhuma outra. Para solucioná-lo, desenvolvemos um programa em Java cujas classes estão no diagrama de classes ilustrado na Figura 5.7.

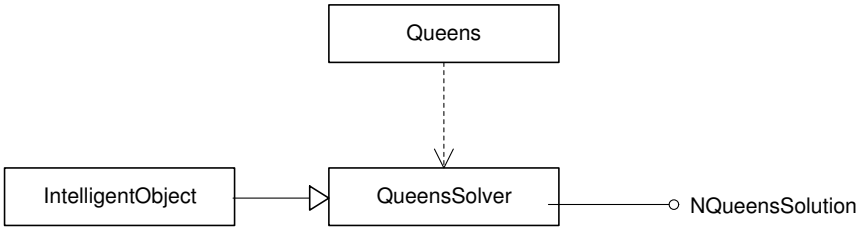

Figura 5.7: Diagrama de classes para o programa das n-rainhas.

A classe `Queens` utiliza os recursos para construção de interfaces gráficas do Java para implementar um tabuleiro de xadrez de tamanho arbitrário, como o apresentado na Figura 5.8. A interface possibilita a definição do tamanho do tabuleiro, que pode ser recriado acionando-se o botão *Generate*. Uma vez solicitado um tabuleiro, pode-se iterar nas possíveis soluções para posicionamento das rainhas através do botão *Next*. Cada vez que este é pressionado, as rainhas se movimentam no tabuleiro numa nova configuração que, possivelmente, satisfaz a definição do problema.

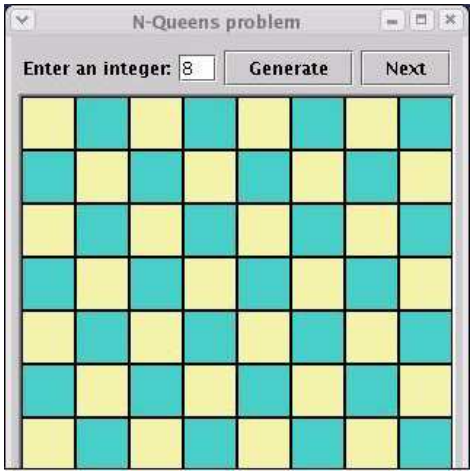

Figura 5.8: Tabuleiro de xadrez 8 x 8 implementado por `Queens`.

O construtor da classe `Queens` deve receber como argumento um objeto de uma classe que implemente a interface `NQueensSolution`. Esta interface, listada a seguir, define os

métodos que uma classe deve implementar para prover a solução do problema das n-rainhas e poder ser utilizada pela interface gráfica.

```
294  public interface NQueensSolution
295  {
296     public void run(int) throws Exception;
297     public boolean hasNext() throws Exception;
298     public int[][] next() throws Exception;
299  }
```

A funcionalidade destes métodos é a seguinte:

- **void** `run(`**int** `n)`: recebe como parâmetro o número n de rainhas e o utiliza para disparar o cálculo das soluções.
- **boolean** `hasNext()`: verifica se há (mais) alguma solução para o problema. Retorna verdadeiro caso afirmativo.
- **int**`[][] next()`: retorna uma solução para posicionamento das rainhas no tabuleiro na forma de uma matriz de inteiros. O tamanho da matriz é n x 2, onde n é o número de rainhas. Para cada linha tem-se dois valores inteiros que correspondem à posição de linha e coluna de uma rainha.

Qualquer classe que implementar adequadamente estes métodos pode fazer uso da interface gráfica provida por `Queens` para apresentar seus resultados. A estratégia utilizada para solucionar o problema não importa, contanto que a interface dos métodos sejam respeitadas. Sendo assim, optamos por construir uma classe inteligente e fazer uso de seus recursos especiais para solucionar o problema das n-rainhas. A classe `QueensSolver`, descrita a seguir, é o resultado desta implementação.

```
300  public class QueensSolver extends IntelligentObject
301     implements NQueensSolution
302  {
303     private int numberOfQueens;
304     private ResultSet rs;
305     private final String SOLUTION_NAME = "Result";
306
307     public int getNumberOfQueens()
308     {
309        return numberOfQueens;
310     }
311     public void run(int numberOfQueens) throws IntelligenceException
312     {
313        try
314        {
315           this.numberOfQueens = numberOfQueens;
316           if (rs != null)
317           {
318              rs.close();
319              rs = null;
320           }
321           rs = ask("queens_list("+ SOLUTION_NAME + ')');
322        }
323        catch(IntelligenceException e)
324        {
325           throw new IntelligenceException("Unable to run!  "+
326              e.getMessage());
327        }
328     }
```

```
329     public boolean hasNext() throws IntelligenceException
330     {
331        try
332        {
333           return rs.next();
334        }
335        catch(IntelligenceException e)
336        {
337           throw new IntelligenceException("Unable to verify if
338              has more solutions!  "+ e.getMessage());
339        }
340     }
341     public int[][] next() throws IntelligenceException
342     {
343        try
344        {
345           java.util.List list = rs.getList(SOLUTION_NAME);
346           int[][] aux = new int[numberOfQueens][2];
347           int i = 0;
348           for (java.util.ListIterator it = list.listIterator();
349              it.hasNext();)
350           {
351              aux[i][0] = i + 1;
352              aux[i][1] = ((Long) it.next()).intValue();
353              i++;
354           }
355           return aux;
356        }
357        catch(IntelligenceException e)
358        {
359           throw new IntelligenceException("Unable to obtain next
360              solution!  "+ e.getMessage());
361        }
362     }
363
364     /*?
365     public rule queens queens_list(Board) :-
366        N is (this.getNumberOfQueens()),
367        length(Board, N),
368        Board ::  1 ..  N,
369        ( (fromto(Board, [Q1|Cols], Cols, [])) do
370           ( (foreach(Q2, Cols), param(Q1), count(Dist,1,_)) do
371           (Q2 #= Q1,
372           Q2 - Q1 #= Dist,
373           Q1 - Q2 #= Dist))
374        ),labeling(Board);
375     */
376  }
```

A classe inteligente `QueensSolver` possui uma única regra, `queens`, que descreve um programa CLP para calcular a solução para um certo número de rainhas. Este número é obtido no interior da regra através da expressão `N is this.getNumberOfQueens()` (linha 366), a qual, quando avaliada pelo ECLiPSe, invoca o método `getNumberOfQueens()` do objeto. Quando esta regra é executada, realiza a busca por uma configuração válida de rainhas posicionadas num tabuleiro `N` x `N`. Quando uma solução é encontrada, esta é armazenada na variável lógica `Board` como uma lista de números inteiros que indicam a posição de cada rainha.

Quando um objeto da classe `QueensSolver` é instanciado, o método `run()` (linha 311) é o que deve ser primeiramente executado. Este recebe o número de rainhas e ajusta a variável de instância `numberOfQueens` (linha 315). Depois invoca o método `ask()` (linha 321) do

objeto inteligente, passando o nome da cabeça da regra que deverá ser executada, bem como a string `SOLUTION_NAME`, que será utilizada para recuperar o valor de retorno armazenado em `Board`. Como visto, após a execução deste método a base de conhecimento do objeto foi traduzida e submetida ao ECLiPSe, bem como a questão a ser resolvida. O retorno do `ask()`, um objeto do tipo `intelligentobject.ResultSet`, é armazenado na variável de instância `rs` e pode ser utilizado então para obtenção das soluções existentes.

Uma vez que o `run()` tenha sido executado, deve-se sempre verificar se há alguma solução disponível através do método `hasNext()`. Este apenas invoca o método `next()` do `result set`, o qual retorna verdadeiro caso ainda haja alguma solução disponível no espaço de soluções, ou falso caso contrário. Por fim, o método `next()` (linha 314) pode ser utilizado para obter uma solução. Isto é feito quando o método `rs.getList(SOLUTION_NAME)` do `result set` é invocado. Este método transforma a lista de inteiros vinda do ECLiPSe, para um objeto do tipo `java.util.List` do Java.

A Figura 5.9 apresenta os dois resultados existentes para o problema com quatro rainhas, enquanto a Figura 5.10 traz as quatro primeiras soluções para o problema com oito rainhas.

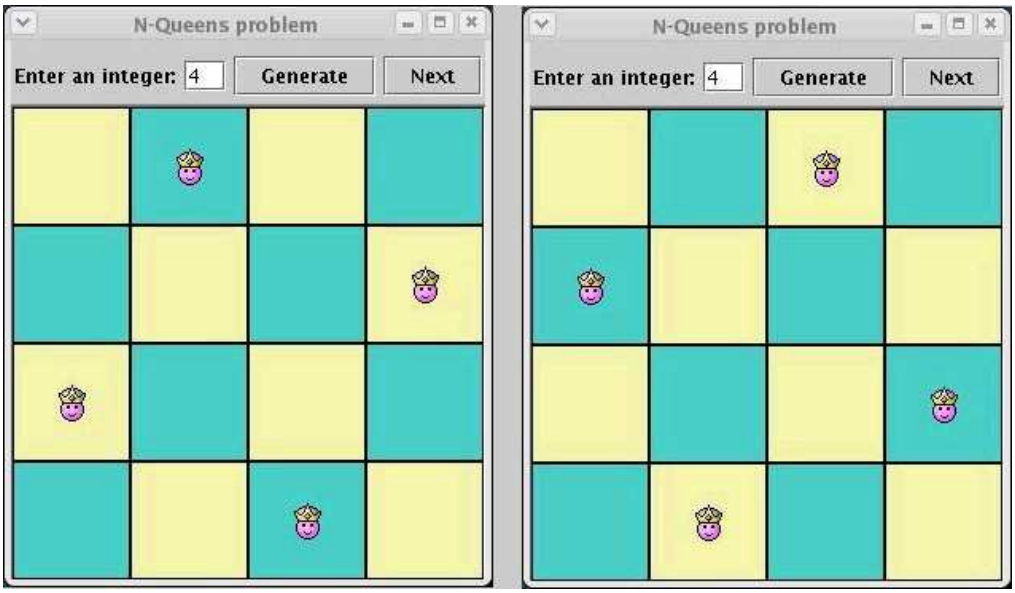

Figura 5.9: Resultado para o problema com 4 rainhas.

5.5.2 O Problema dos Azulejos

O problema dos azulejos (*square tiling problem*) consiste em descobrir, dada uma área quadrada e um conjunto de azulejos também quadrados e de tamanhos variados, quais as possíveis combinações para acomodação dos azulejos de forma a cobrir completamente a área definida.

Por exemplo, para um quadrado de $1m^2$ de área, ou seja, seus lados medem $1m$ cada, e quatro azulejos com $50cm$ de lado cada um, a única e evidente solução é acomodar dois na parte superior e dois na parte inferior. Este exemplo é trivial, porém, no caso de muitos azulejos com tamanhos variados a solução passa ser muito mais trabalhosa e difícil de ser obtida sem algum auxílio.

Para resolver este problema, desenvolvemos a classe inteligente `SquareTiling`, apresentada a seguir.

```
377  public class SquareTiling extends IntelligentObject
378  {
```

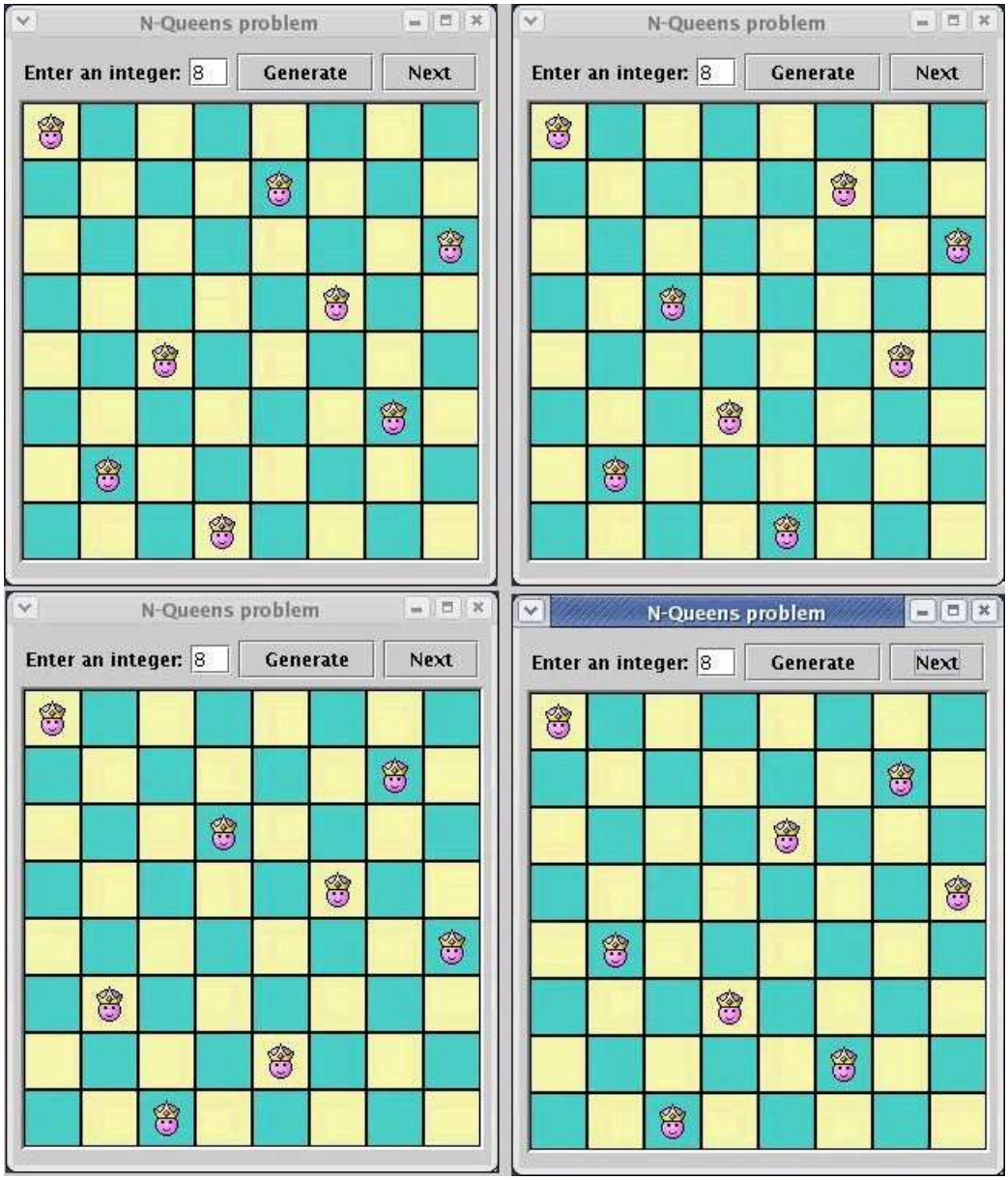

Figura 5.10: As quatro primeiras soluções para o problema das 8 rainhas.

```
379     public int areaSize;
380     public int [] tiles;
381     private ResultSet rs = null;
382
383     public SquareTiling(int areaSize, int[] tiles)
384        throws IntelligenceException
385     {
386        this.areaSize = areaSize;
387        this.tiles = tiles;
388     }
389     public void run() throws IntelligenceException
390     {
391        try
392        {
393           if (rs != null)
394              rs.close();
395           rs = ask("squares(this.areaSize, this.tiles, Xs,Ys)");
```

```
396       }
397       catch(IntelligenceException e)
398       {
399          throw new IntelligenceException("Unable to run!  Error:  "
400             + e.getMessage());
401       }
402    }
403    public boolean hasNext() throws IntelligenceException
404    {
405       try
406       {
407          return rs.next();
408       }
409       catch(IntelligenceException e)
410       {
411          throw new IntelligenceException("Unable to obtain next
412             solution!  "+ e.getMessage());
413       }
414    }
415    public void next() throws IntelligenceException
416    {
417       try
418       {
419          printSquare(rs.getList("Xs"), rs.getList("Ys"));
420       }
421       catch(IntelligenceException e)
422       {
423          throw new IntelligenceException(e.getMessage());
424       }
425    }
426
427    /*?
428    public rule square squares(Size, Sizes, Xs, Ys) :-
429       ( (foreach(X,Xs), foreach(Y,Ys), foreach(S,Sizes)
430       , param(Size)) do
431          ( Sd is Size - S + 1,
432          [X,Y] ::  1 ..  Sd)
433       )),
434       no_overlap(Xs, Ys, Sizes),
435       capacity(Xs, Sizes, Size),
436       capacity(Ys, Sizes, Size),
437       mylabeling(Xs) , mylabeling(Ys);
438
439    public rule no_overlap3 no_overlap(Xs, Ys, Sizes) :-
440       ( (fromto(Xs, [X|XXs], XXs, []),
441       fromto(Ys, [Y|YYs], YYs, []),
442       fromto(Sizes, [S|SSs], SSs, []))
443       do
444       ( (foreach(X1,XXs), foreach(Y1,YYs), foreach(S1,SSs)
445       , param(X,Y,S)) do
446          (no_overlap(X, Y, S, X1, Y1, S1))
447       )
448    );
449
450    public rule no_overlap6 no_overlap(X1, Y1, S1, X2, Y2, S2) :-
451       1 #= ( (X1+S1 #=< X2) or (X2+S2 #=< X1) or (Y1+S1 #=< Y2)
452       or (Y2+S2 #=<Y1));
453
454    public rule capacity capacity(Cs, Sizes, Size) :-
455       (for(Pos,1,Size), param(Cs,Size,Sizes)) do
456          (( (foreach(C,Cs), foreach(S,Sizes), param(Pos),
457          fromto(Sum, S * B + Sum1, Sum1, 0))
458          do
```

```
459             (operator::(C, Pos - S + 1 ..  Pos, B))
460           ),
461         Sum #= eval(Size));
462
463     public rule mylabeling mylabeling(InitialList) :-
464         search(InitialList, 0, 'smallest', 'indomain', 'complete',[]);
465     */
466   }
```

A base de conhecimento da classe `SquareTiling` é composta por cinco regras CLP. Diferente da regra do exemplo anterior, estas não fazem uso de métodos ou atributos do objeto, estando codificadas apenas com variáveis lógicas, restrições e operadores. A regra `square`, cuja cabeça é definida por `squares(Size, Sizes, Xs, Ys)`, é o ponto de partida para início de um processamento. `Size` e `Sizes` são parâmetros de entrada e representam, respectivamente, o tamanho do quadrado a ser preenchido e uma lista com os tamanhos dos azulejos a serem utilizados. A soma total das áreas dos azulejos deve ser igual a área total a ser preenchida, isto porque o programa CLP sempre tentará encontrar uma solução fazendo uso de todos os azulejos disponíveis. `Xs` e `Ys` são parâmetros de saída e significam as coordenadas do canto superior esquerdo de cada um dos azulejos contidos em `Sizes`.

A utilização da classe é simples e seus principais métodos são:

- `SquareTiling(int areaSize, int[] tiles)` (linha 383): o construtor da classe deve receber o tamanho de um dos lados do quadrado a ser coberto e um vetor de inteiros com o tamanho dos lados de todos os azulejos que devem ser utilizados. A área total formada pelos azulejos deve ser a mesma que a área a ser preenchida por estes. Os valores recebidos são atribuídos para variáveis de instância da classe.
- `run()` (linha 389): executa o processamento das regras invocando o método `ask` com a string `"squares(this.areaSize, this.tiles, Xs,Ys)"`. Pode-se observar aqui atributos do objeto sendo utilizados como parâmetros para a cabeça da regra. Tanto o inteiro `Size` como o array de inteiros `Sizes` terão seus valores convertidos para valores equivalentes no ECLiPSe.
- `hasNext()` (linha 403): verifica se há alguma solução disponível.
- `next()` (linha 415): obtém e imprime a solução corrente.

Como exemplo de utilização da classe, considere o programa `Squares` abaixo:

```
public class Squares
{
   public static void main(String[] args)
   {
      int[] tiles = {10,9,7,6,4,4,3,3,3,3,3,2,2,2,1,1,1,1,1,1};
      try
      {
         SquareTiling st = new SquareTiling(19, tiles);
         st.run();
         while (st.hasNext())
          st.next();
      }
      catch(IntelligenceException e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

Neste exemplo, a área a ser preenchida é um quadrado de $19m^2$. Assim sendo, o constructor da classe é invocado da seguinte forma: `SquareTiling(19, tiles)`, onde `tiles` é um vetor de inteiros com o conjunto de azulejos selecionados. A Figura 5.11 apresenta alguns resultados produzidos na execução do programa.

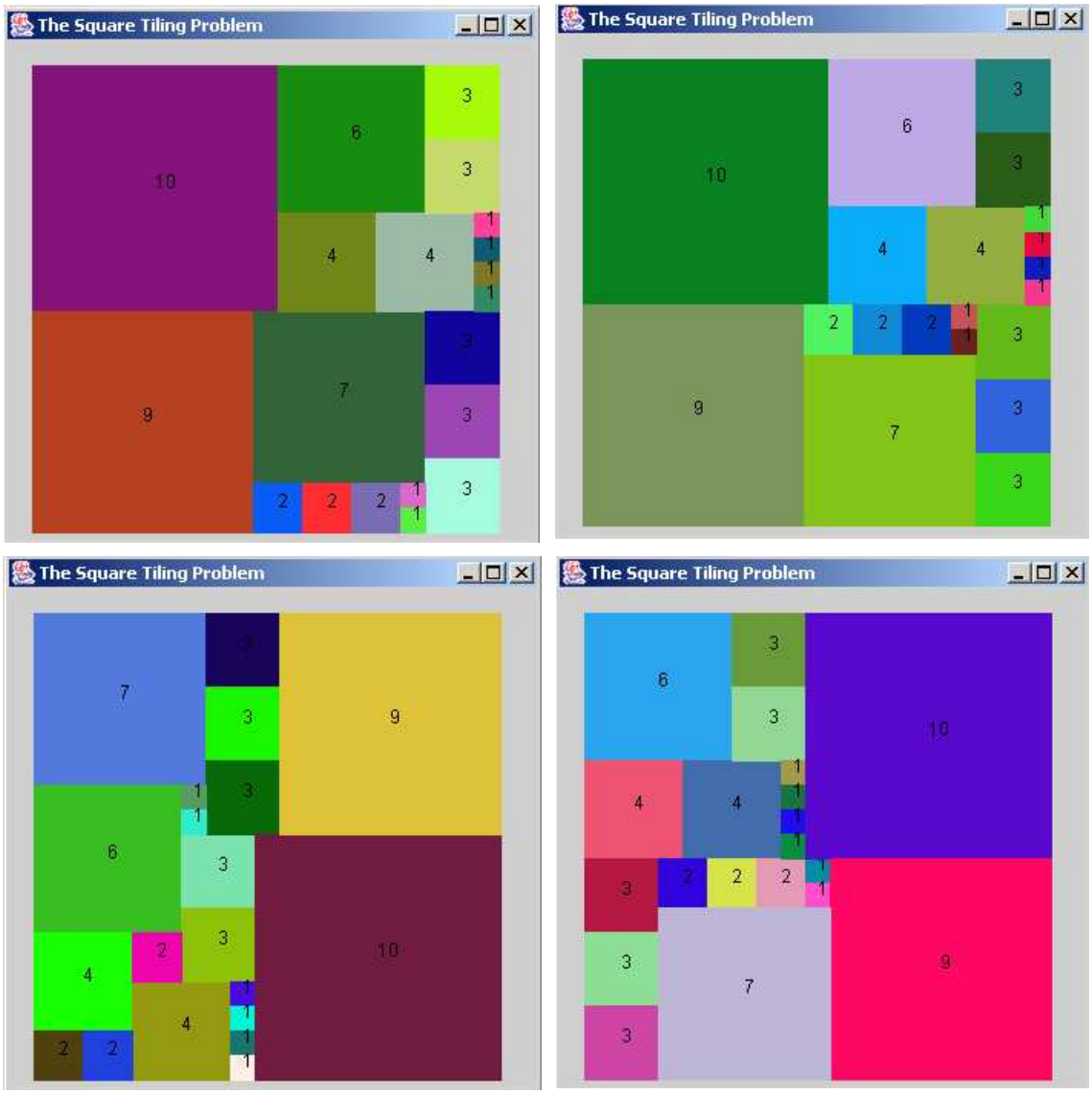

Figura 5.11: Quatro configurações diferentes de azulejos produzidas na execução de `Squares`.

5.5.3 A Aplicação I2D

I2D (*Intelligent 2D*) é uma aplicação gráfica Java destinada à criação de cenas compostas de objetos geométricos bidimensionais fechados. Com esta um usuário pode:

- compor uma cena com objetos cujos contornos são constituídos de segmentos de linhas retas e curvas quádricas e cúbicas (Bezier). Círculos, retângulos e polígonos são objetos primitivos (a partir dos quais outros são criados);
- selecionar objetos de uma cena;
- mover objetos selecionados (transformação de translação);

- rotacionar objetos selecionados (transformação de rotação);
- colorir objetos;
- apagar objetos.

A Figura 5.12 apresenta uma tela do aplicativo com alguns objetos simples criados.

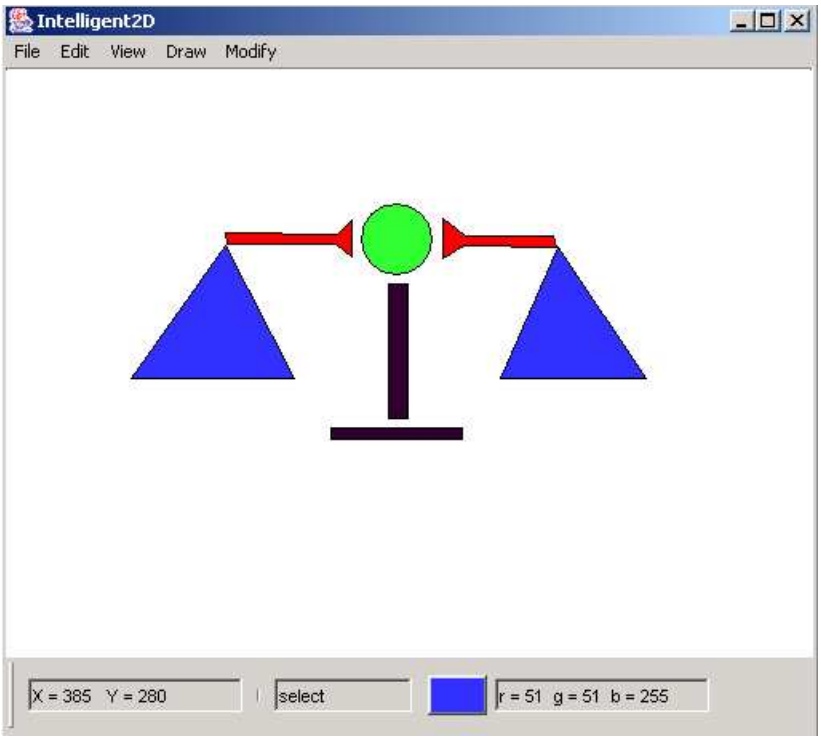

Figura 5.12: Uma cena criada com o aplicativo I2D.

A aplicação foi desenvolvida e utilizada como exemplo neste trabalho somente para ilustrar como o emprego de inteligência computacional pode ser convenientemente útil na implementação de estratégias de tomada de decisões em programas científicos (fora isso, a aplicação não é mais que um exercício acadêmico de computação gráfica bidimensional). O Grupo de Computação Gráfica e Mecânica Computacional (GCGMC) da UFMS tem desenvolvido programas gráficos e numéricos que poderiam ser mais apropriadamente empregados para demonstrar o conceito de objetos inteligentes e sua utilidade em aplicações dessa natureza, por exemplo OSW-Solid [14]. Este é um modelador de sólidos no qual objetos tridimensionais são construídos através de operações booleanas regularizadas e de sólidos primitivos tais como esferas, cilindros, caixas, etc. Questões como o que acontece quando um novo sólido é adicionado a uma cena em um espaço já ocupado por outros objetos, ou qual é a reação de outros objetos de uma cena quando um conjunto de sólidos da cena é transformado de alguma maneira, não obstante o processamento geométrico envolvido, podem ter respostas codificadas através de regras. Com isso, certas restrições seriam resolvidas através da invocação do raciocínio de objetos inteligentes. A vantagem, neste caso, é que modificações ou extensões dessas estratégias ocasionariam a alteração de regras existentes ou a definição de novas regras, sem modificações no código fonte, reduzindo portanto o trabalho de manutenção da aplicação. Contudo, a utilização de um programa complexo como OSW-Solid como exemplo envolveria a codificação de centenas de classes de objetos, originalmente escritas em C++, em Java. Optou-se, então, pelo desenvolvimento de I2D. Embora muito mais simples, podemos, com as regras embutidas nas classes de objetos inteligentes da aplicação, vislumbrar algumas possibilidades quando extrapolamos para 3D.

Uma cena na I2D é um objeto da classe `Scene2D`. Uma cena é composta por uma coleção de objetos derivados da classe inteligente abstrata `Object2D` e armazenados num objeto da classe `Object2DSet`. As classes concretas que derivam de `Object2D` são: `Circle2D`, `Rect2D` e `Polygon2D`. A Figura 5.13 apresenta o diagrama de classes que contempla esta modelagem.

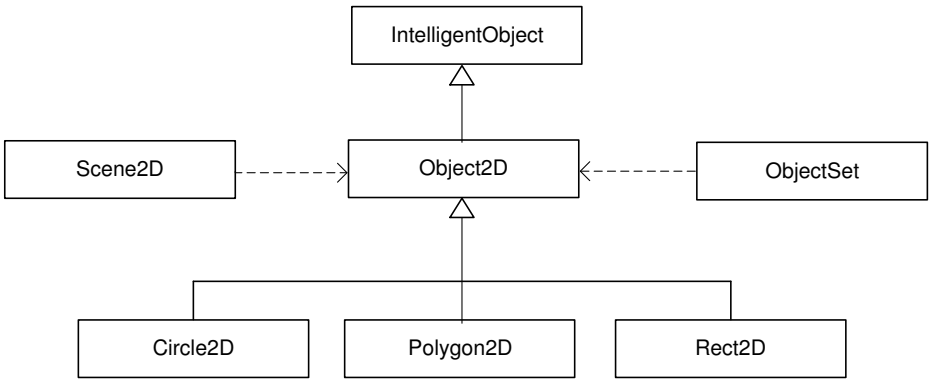

Figura 5.13: Diagrama de classes simplificado para a aplicação I2D.

Os principais métodos da classe `Object2D` e `Scene2D` estão listados nas Tabelas 5.2 e 5.3.

Classe `Object2D`
Métodos públicos
`void` `transform(AffineTransform t)`
Aplica a transformação `t` no objeto.
`void` `invLastTransformation()`
Desfaz a última transformação aplicada ao objeto.
`boolean` `intersects(Object2D other)`
Retorna **`true`** se há intersecção entre o objeto e `other`.
`int` `isIntersectingOtherObject()`
Verifica se o objeto intercepta algum outro na cena.
Retorna 1 caso haja intersecção e 0 caso contrário.
`void` `abortCreation()`
Atribui **`true`** para a atributo `abortCreation`.
`ResultSet ask(String query)`
Dispara o raciocínio do objeto baseado na consulta contida em `query`.

Tabela 5.2: Classe `Object2D`.

A inteligência de um `Object2D` é aplicada em duas situações: na transformação e na criação de objetos em uma cena. Quando objetos são deslocados de sua posição atual através das operações de transformação, descritas em maiores detalhes na próxima seção, e passam a sobrepor outros objetos da cena, consideramos isso como indesejável e passível de uma ação corretiva. Da mesma forma ocorre quando objetos são criados sobre outros objetos da cena. A ação a ser executada quando estas inconsistência são detectadas na I2D está codificada na base de conhecimento de um objeto que, utilizando seu conhecimento, decidirá o que deve ser feito.

Transformação de Objetos

Com a I2D um usuário pode selecionar objetos numa cena e escolher arrastá-los ou rotacioná-los a fim de alterar suas posições. Estas ações correspondem respectivamente às transformações de translação e rotação e o deslocamento que produzem pode significar que um

Classe `Scene2D`
Métodos públicos
void `addObject(Object2D object)` Inclui `object` na coleção de objetos da cena. Utiliza o conhecimento do objeto para detectar se este foi criado sobre algum outro e executar a ação apropriada caso afirmativo.
void `removeObject(Object2D object)` Remove `object` da coleção de objetos da cena.
boolean `transform(AffineTransform t, Object2DSet objects)` Aplica a transformação `t` a todos os objetos contidos em `objects`. Retorna **true** caso não tenha ocorrido nenhum problema com as transformações.
boolean `transformAll(AffineTransform t)` Aplica a transformação `t` a todos os objetos da cena. Retorna **true** não tenha ocorrido nenhum problema com as transformações.
boolean `verifyTransformation(Object2DSet objects)` Utiliza o conhecimento de cada objeto contido em `objects` para verificar se este intercepta algum outro e executar a ação apropriada caso afirmativo. Retorna **true** caso não haja intersecção.
`Object2DSet getIntersectionSet(Object2D object)` Retorna o conjunto de objetos que `object` intercepta na cena.

Tabela 5.3: Classe `Scene2D`.

ou mais desses objetos passaram a sobrepor outros objetos da cena. Contudo, consideramos desejável na I2D que esta situação seja tratada a fim de corrigir tais sobreposições. O método `transform()` da `Scene2D` é o responsável por aplicar a transformação escolhida nos objetos selecionados e por verificar se o seu resultado está consistente. Isto é feito no método `verifyTransformation()` descrito abaixo.

```
467  //Scene2D.verifyTransformation
468  public boolean verifyTransformation(Object2DSet objects)
469     throws NoninvertibleTransformException
470     throws IntelligenceException
471  {
472     boolean verified = true;
473     for (Object2DSet.Iterator it = objects.iterator(); it.hasNext();)
474     {
475        Object2D object = it.next();
476        selectedObjectsForTransformation = objects;
477        ResultSet rs = object.ask("verifyTransformation(Verified)");
478        rs.next();
479        if (rs.getLong("Verified"== 1)
480           verified = false;
481        rs.close();
482     }
483     return verified;
484  }
```

Este método recebe como parâmetro os objetos selecionados numa cena em um objeto da classe `Object2DSet`. Para cada objeto que sofreu a transformação é acionada a regra `verifyTransformation` (linha 477). O resultado da verificação é obtido através da variável `Verified` (linha 479), sendo um se o objeto está sobre outro e zero caso contrário. O método retorna verdadeiro se nenhum dos objetos selecionados está sobrepondo outro e falso caso exista alguma sobreposição.

A classe `Object2D` possui a implementação padrão da regra `verifyTransformation`, descrita na seguinte base de conhecimento:

```
485 //Base de conhecimento da classe Object2D
486 /*?
487 //This rule is called in the method verifyTransformation.  It checks
488 //if this Object2D is intercepting another after some transformation
489 //applied on it and, if it's true, asks to the object undo the last
490 //transformation.
491
492 public rule verifyTransformation verifyTransformation(Verified) :-
493    if (this.isIntersectingOtherObject() is 0)
494       this.scene.objects.extents.inflate(
495          this.scene.selectedObjectsForTransformation.extents),
496       Verified is 0
497    else
498       this.invLastTransformation(), Verified is 1;
499 */
```

O corpo da regra possui apenas uma declaração `if` que testa se o objeto em questão está interceptando algum outro. Caso não esteja, isto é, o método `isIntersectingOtherObject()` retornou 0, então a transformação é efetivada, caso contrário o objeto desfaz a última transformação nele aplicada.

A regra `verifyTransformation` é pública, sendo assim é herdada pelas classes que derivam de `Object2D`, isto é, `Rect2D`, `Polygon2D` e `Circle2D`. Entretanto, apenas `Rect2D` mantém o comportamento da implementação original, `Polygon2D` e `Circle2D` sobrescrevem a regra para fornecer seu próprio comportamento quando detectarem que estão sobre algum outro objeto. Um `Circle2D` realiza uma junção com o(s) objeto(s) que está sobrepondo, enquanto um `Polygon2D` tenta se posicionar o mais perto possível da posição final gerada pela transformação sem, no entanto, sobrepor nenhum outro objeto. As bases de conhecimento para estas classes estão listadas a seguir.

```
500 //Base de conhecimento da classe Circle2D
501 /*?
502 //This rule is called in the method verifyTransformation.  It checks
503 //if this Circle2D is intercepting another after some transformation
504 //applied on it and, if it's true, asks to the object to merge itself
505 //with all others it is intercepting.
506
507 public rule verifyTransformation verifyTransformation(Verified) :-
508    if (this.isIntersectingOtherObject() is 0)
509       this.scene.objects.extents.inflate(
510          this.scene.selectedObjectsForTransformation.extents),
511       Verified is 0
512    else
513       this.merge(this.scene.getIntersectionSet(this)), Verified is 1;
514 */

515 //Base de conhecimento da classe Polygon2D
516 /*?
517 //This rule is called in the method verifyTransformation.  It checks
518 //if this Polygon2D is intercepting another after some transformation
519 //applied on it and, if it's true, asks to the object to position
520 //himself as near as possible.
521
522 public rule verifyTransformation verifyTransformation(Verified) :-
523    if (this.isIntersectingOtherObject() is 0)
524       this.scene.objects.extents.inflate(
525          this.scene.selectedObjectsForTransformation.extents),
```

```
526       Verified is 0
527    else
528       this.posAsNearAsPossible(this.scene.getIntersectionSet(this)),
529       Verified is 1;
530 */
```

As Figuras 5.14, 5.15 e 5.16 ilustram passa a passo o resultado de uma transformação que resulta em sobreposição aplicada aos três tipos de objetos. Para cada um deles pode-se verificar o comportamento específico após a sobreposição ter sido detectada.

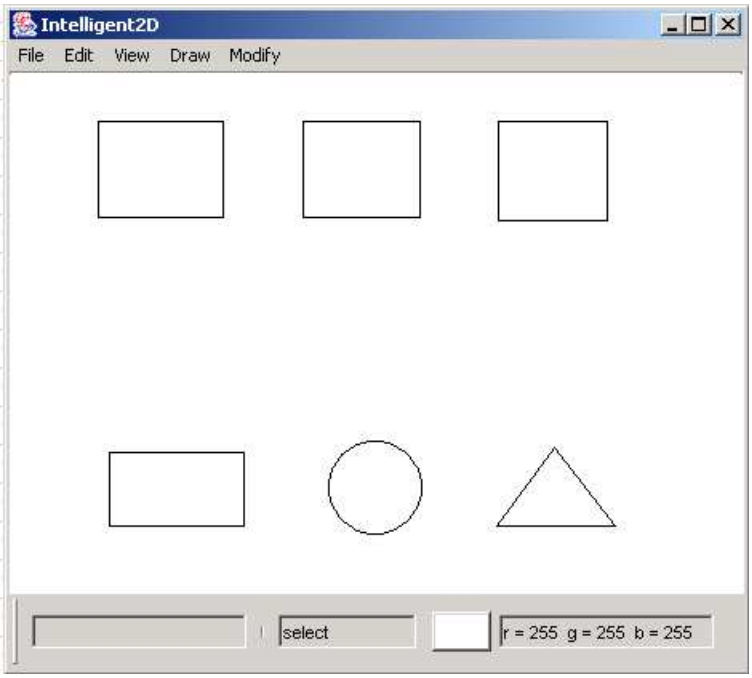

Figura 5.14: Alguns objetos criados com aplicação I2D.

Criação de Objetos

Quando um novo objeto é criado em uma cena, nada impede que o usuário o posicione sobre outros objetos já existentes. A aplicação I2D trata dos problemas de sobreposição que podem ocorrer na criação de objetos de forma similar às transformações.

Um novo objeto é adicionado a uma cena no método `addObject()` da classe `Scene2D` listado a seguir.

```
531 //Scene2D.addObject
532 public void addObject(Object2D object)
533    throws IntelligenceException
534 {
535    if (object.scene != null)
536       object.scene.removeObject(object);
537    ResultSet rs = object.ask("verifyCreation()");
538    rs.next();
539    rs.close();
540    if (!object.abortCreation)
541    {
542       objects.add(object);
543       object.scene = this;
544    }
545    else
```

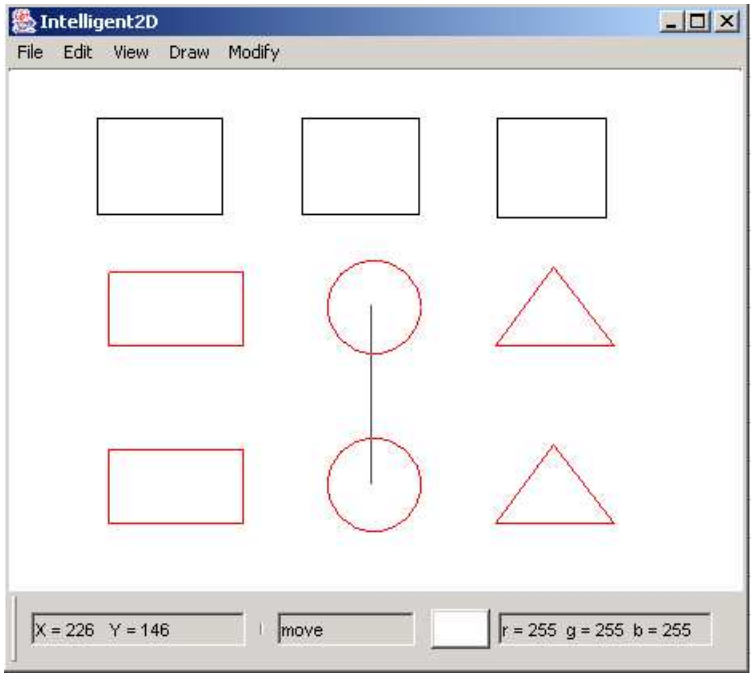

Figura 5.15: O retângulo, círculo e triângulo estão sendo transladados para sobreporem os quadrados acima deles.

```
546         object.abortCreation = false;
547   }
```

Na linha 535 é verificado se o objeto sendo adicionado àquela cena já pertence a alguma outra e, caso afirmativo, este é removido da mesma. Nas linhas 537 e 538 a regra `verifyCreation` é submetida e executada. É esta regra que irá detectar e tratar o problema da sobreposição na criação de um `Object2D`. Nas linhas 542 e 543, caso a criação do objeto esteja liberada, este é adicionado à lista de objetos da cena e recebe um ponteiro para esta.

A classe `Object2D` implementa na regra `verifyCreation` (linha 554) a estratégia padrão para tratar da sobreposição na criação de objetos da classe.

```
548   //Base de conhecimento da classe Object2D
549   /*?
550   //This rule is called in the method addObject.  It checks
551   //if this Object2D is intercepting another when created
552   //and, if it's true, asks to the object to abort it's creation.
553
554   public rule verifyCreation verifyCreation() :-
555      if (this.isIntersectingOtherObject() is 1)
556         this.abortCreation();
557   */
```

O corpo da regra possui apenas uma declaração `if` que testa se o objeto em questão está interceptando algum outro. Caso esteja, isto é, o método `isIntersectingOtherObject()` retornou 1, então o objeto é solicitado a abortar sua criação, o que significa que este não será adicionado à lista de objetos daquela cena.

Assim como na transformação de objetos, apenas a classe `Rect2D` mantém a estratégia padrão implementada por `Object2D`. As classes `Circle2D` e `Polygon2D` sobrescrevem a regra para fornecer outro comportamento a esta. A regra `verifyCreation` (linha 565) de `Circle2D` tenta posicionar o círculo o mais perto possível da posição original da sua criação

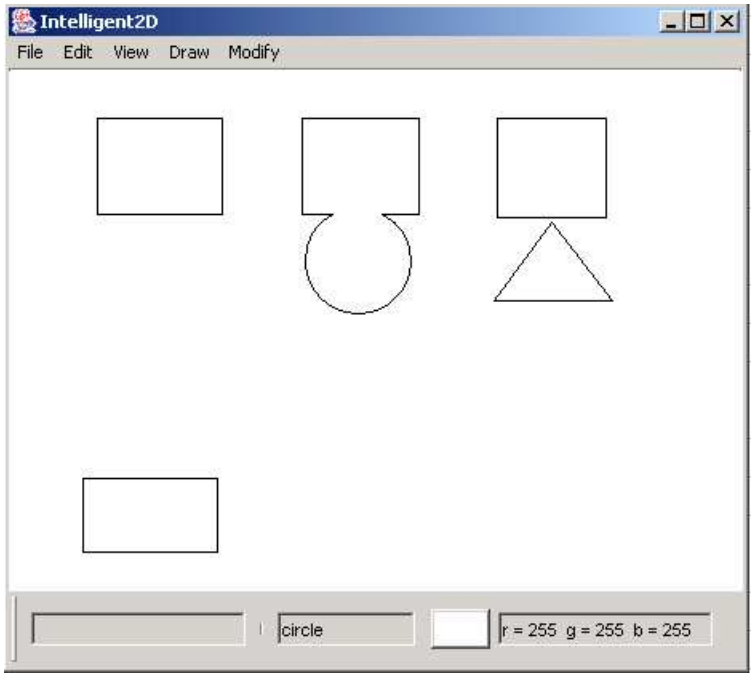

Figura 5.16: Resultado após a translação. O retângulo voltou para sua posição de origem, o círculo se fundiu com o quadrado e o triângulo se aproximou o máximo que pôde.

sem sobrepor nenhum outro objeto, enquanto a `verifyCreation` (linha 576) de `Polygon2D` faz o polígono se fundir com os objetos que está sobrepondo.

```
558  //Base de conhecimento da classe Circle2D
559  /*?
560  //This rule is called in the method addObject.  It checks
561  //if this Circle2D is intercepting another when created
562  //and, if it's true, asks to the object to position itself
563  //as near as possible.
564
565  public rule verifyCreation verifyCreation() :-
566     if (this.isIntersectingOtherObject() is 1)
567        this.posAsNearAsPossible(this.scene.getIntersectionSet(this));
568  */

569  //Base de conhecimento da classe Polygon2D
570  /*?
571  //This rule is called in the method addObject.  It checks
572  //if this Polygon2D is intercepting another when created
573  //and, if it's true, asks to the object to merge itself
574  //with all others that cause the interceptions.
575
576  public rule verifyCreation verifyCreation() :-
577     if (this.isIntersectingOtherObject() is 1)
578        this.merge(this.scene.getIntersectionSet(this));
579  */
```

Conclusão

Na abordagem que adotamos para utilização do conhecimento na aplicação I2D, delegamos às regras implementarem estratégias de ações a serem adotadas mediante alguma situação que ocorre com um objeto inteligente. Utilizando herança e polimorfismo de regras, foi possível

especializar estratégias de acordo com as características dos objetos e assim criar comportamentos genéricos e específicos dentro do conhecimento dos diversos objetos inteligentes da aplicação.

O fato dessas estratégias serem implementadas como parte do conhecimento dos objetos e não no código da linguagem hospedeira, resulta em um programa mais flexível e facilmente manutenível. Se alguma estratégia precisar ser alterada, basta que se altere a base de conhecimento da classe de objetos, evitando assim possíveis erros decorrentes da modificação do código fonte da aplicação. Se a base estiver centralizada e compartilhada, evita-se também a necessidade de redistribuição do aplicativo.

5.6 Comentários Finais

Concluímos com este capítulo a parte final do nosso trabalho. Descrevemos aqui as partes mais relevantes da implementação de todos os componentes necessários ao funcionamento de um objeto inteligente. Abordamos a criação, gerenciamento e persistência de bases de conhecimento. Discutimos sobre a criação de *middlewares* para integração com mecanismos de resolução de restrições de terceiros e mostramos a nossa implementação de um para integração com o ECLiPSe. Explicamos em detalhes a questão da reflexão computacional e como esta foi utilizada na solução. Apresentamos também como um processo de raciocínio é disparado e como os resultados de seu processamento podem ser obtidos através de classes Java bastante intuitivas e de fácil manipulação. Por fim, os exemplos fornecem informações adicionais da aplicabilidade e facilidade de uso desta implementação.

CAPÍTULO 6

Conclusão

6.1 Discussão dos Resultados Obtidos

O objetivo deste trabalho foi definir, implementar e validar um conceito de objeto inteligente baseado em CLP para ser utilizado na construção de aplicações computacionais numéricas e gráficas. Usamos a orientação a objetos pois esta nos permite melhor gerenciar a complexidade de aplicações desta natureza, tanto na fase de modelagem como na de implementação. Um objeto inteligente engloba dados e métodos, tal qual um objeto convencional, mais conhecimento declarativo e mecanismos de inferência, que possibilitam a este produzir comportamentos inteligentes.

A idéia de integrar conhecimento com objetos é explorada desde a época do surgimento dos sistemas baseados em conhecimento e inúmeros esquemas de integração de regras de produção com objetos já foram propostos [5, 21, 13, 40, 45, 46]. Dentre estes, consideramos os mais interessantes os apresentados em [13] e [46], pois suas abordagens possibilitam uma integração natural das regras com a orientação a objetos. Isto deve-se ao fato das regras serem tratadas como pseudo-métodos da linguagem e, por isso, respeitarem as principais características da orientação a objetos — encapsulamento, herança e polimorfismo.

O conhecimento dos objetos inteligentes que propomos, diferentemente dos sistemas especialistas, é formulado através de regras CLP (*programação lógica com restrições ou constraint logic programming*). Escolhemos a linguagem lógica com restrições como forma de representação do conhecimento pois suas características a tornam mais adequada para ser empregada no auxílio da resolução de problemas científicos. Um dos motivos disto é que as restrições embutidas suplementam a natureza puramente abstrata da lógica, introduzindo símbolos que possuem significados pré-definidos num determinado domínio de aplicação. Outro fator é que o conhecimento que estas restrições codificam pode ser usado para restringir o espaço de busca das soluções, possibilitando uma performance de busca superior que a de outras linguagens lógicas.

Os objetivos específicos que visamos atingir ao longo deste trabalho a fim de tornar viável sua compreensão e execução foram:

1. *Revisão dos fundamentos da lógica proposicional e de primeira ordem, necessários à compreensão dos princípios da programação lógica com restrições.* No Capítulo 2 fizemos uma revisão sobre a lógica e sua adequação como método de representação do conhecimento. Seu estudo levou a criação de métodos sistemáticos para provar asser-

tivas lógicas, contribuindo para o surgimento das linguagens lógicas de programação e dos mecanismos de inferência que atuam sobre elas. O Prolog é um dos principais representantes desta classe de linguagens de programação e serve como base de estudo para outras propostas e extensões. O entendimento da representação de conhecimento através de sistemas lógicos, dos mecanismos de inferência que permitem processá-lo e de linguagens lógicas de programação são necessários para uma boa compreensão da tecnologia CLP que adotamos.

2. *Estudo dos conceitos e técnicas da CLP e suas aplicações.* No Capítulo 3 apresentamos a programação lógica com restrições. Inicialmente fizemos um resumo das idéias que influenciaram seu surgimento o qual, de maneira geral, deve-se a observância de que a *programação lógica* era apenas um tipo particular de *programação com restrições* e que um novo paradigma para modelar e resolver problemas poderia ser criado com a unificação dessas tecnologias. Depois disso descrevemos as bases do esquema de programação lógica com restrições, chamado CLP(X). O principal objetivo deste esquema é prover ao usuário mais expressividade e flexibilidade no que tange aos objetos primitivos que a linguagem pode manipular. Para isso, símbolos especiais (restrições) são introduzidos e, por possuírem um significado fixo dentro de um determinado domínio (como o aritmético linear e não linear), possibilitam modelar as relações existentes em um domínio particular. Apesar das restrições poderem ser combinadas com diferentes linguagens lógicas, os trabalhos realizados estão quase que exclusivamente dedicados à introdução de métodos de resolução de restrições em linguagens lógicas baseadas no Prolog. Por isso a sintaxe e semântica, declarativa e operacional do esquema, serem fortemente influenciadas por este. Em seguida, abordamos as técnicas de consistência e seus algoritmos para aperfeiçoar a performance da busca através da manipulação de restrições. Devido à combinação destas técnicas ao modelo operacional das linguagens CLP, estas atingiram performance de execução superior a de outras linguagens lógicas e, em algumas situações, equivalente à execução de programas procedimentais. Isso estimulou sua utilização e as solidificou como opção eficaz para solução de problemas que envolve busca.

3. *Definição de objeto inteligente baseado em CLP.* No Capítulo 4 apresentamos nosso projeto para objetos inteligentes com CLP. Inicialmente explicamos as características que tornam um objeto convencional um objeto inteligente. Evidenciamos as vantagens da arquitetura que adotamos para organização do conhecimento orientado a objetos, onde as regras, por estarem acopladas aos objetos na forma de pseudo-métodos que respeitam as características da orientação a objetos, possuem um estilo de expressão uniforme e uma hierarquia equivalente à hierarquia de classes da aplicação. Em seguida, descrevemos em alto nível os componentes que projetamos para a solução. Primeiro, mostramos como as regras CLP do objeto são criadas, avaliadas e armazenadas na sua base de conhecimento, de forma a respeitar todas as características da organização do conhecimento antes definidas. Depois, explicamos nossa proposta para processamento do conhecimento e a opção que fizemos por não implementar uma máquina de inferência, mas sim, integrar outras já existentes. Projetando a solução para utilizar um software intermediário (um *middleware*) que faz a ligação entre o objeto e o mecanismo de inferência, seria possível, em tese, obter uma certa independência do mecanismo de inferência, o qual poderia ser trocado por outro sem grandes alterações no conhecimento já codificado dos objetos, contanto que existisse o *middleware* apropriado.

4. *Descrição da implementação e exemplos de utilização dos objetos inteligentes.* No Capítulo 5, descrevemos a implementação do sistema baseado em objetos inteligentes. Primeiramente, abordamos todos os passos para geração da base de conhecimento

dos objetos inteligentes, desde a codificação textual das regras na classe dos objetos na forma de comentários especiais da linguagem, extração e validação destas pelo compilador de regras, que cria uma base de conhecimento temporária para cada objeto e a torna persistente gravando-a no disco, até a posterior recuperação dessas bases e sua criação, que deve levar em conta a hierarquia de classes inteligentes de cada objeto. Depois disso, discutimos sobre os aspectos funcionais que um *middleware* para comunicação com um mecanismo de inferência deve contemplar e explicamos nossa implementação de um para o ECLiPSe [32, 66]. O ECLiPSe é uma plataforma para integrar várias extensões da programação lógica, em particular a programação lógica com restrição. O escolhemos como nosso mecanismo solucionador de restrições porque, além de bastante completo no que tange aos domínios que possibilita tratar, possui interfaces para integração com outras linguagens de programação como Java e C++. Em seguida, descrevemos como ocorre a inicialização de um processo de inferência e como seu resultado pode ser obtido e utilizado por um objeto inteligente. Por último, demonstramos a solução através da implementação de aplicações que fazem uso de objetos inteligentes para solução de algum problema. A primeira resolve um problema clássico de busca, o problema das n-rainhas. A segunda, implementa a solução de um problema geométrico combinatório: dada uma grande área quadrada e um conjunto de quadrados menores com tamanhos variados, quais as maneiras possíveis de cobrir esta área utilizando os quadrados disponíveis. A terceira é uma aplicação gráfica que permite a geração de figuras poligonais fechadas em duas dimensões.

Com isso cobrimos todas as etapas do nosso trabalho. O resultado da integração está, como havíamos proposto, bastante natural e intuitivo para aqueles já familiarizados com a programação orientada a objetos. Originalmente nossa intenção era a de propor uma integração com o C++ devido às suas características como suporte à orientação a objetos e disponibilização de mecanismos eficazes que possibilitam a obtenção dos recursos computacionais que em geral aplicações numéricas e gráficas requerem. Contudo, neste trabalho utilizamos a linguagem Java, pois, além de possibilitar o desenvolvimento mais rápido de um protótipo com o qual pudéssemos validar nossa proposta, os ambientes de execução padrão de Java disponibilizam os recursos de reflexão computacional e serialização que em C++ teríamos que implementar. Apesar disso, a solução consegue capturar completamente os benefícios que a utilização de uma técnica de inteligência artificial como esta pode trazer na implementação de aplicações orientadas a objetos e concluímos que todos os objetivos do trabalho foram alcançados.

Os exemplos apresentados no Capítulo 5 são simples mas oferecem uma boa idéia da utilização e funcionamento do sistema de objetos inteligentes proposto. Pretendíamos, no início, adaptar algumas aplicações numéricas e gráficas desenvolvidas com base na biblioteca de classes C++ OSW (*Object Structural Workbench*) [47] para utilizarem objetos inteligentes. OSW é um *toolkit* destinado ao desenvolvimento de aplicações de modelagem em Ciências e Engenharia. Uma aplicação de modelagem, no contexto do OSW, é um programa orientado a objetos de análise e visualização de modelos estruturais. Estas aplicações são bastante complexas e forneceriam um excelente estudo de caso; no entanto, justamente devido ao seu tamanho e complexidade, tornou-se inviável sua migração para Java num tempo hábil.

OSW-Solid é um exemplo de aplicação de modelagem OSW na qual poderíamos empregar objetos inteligentes, não apenas para modelagem geométrica de uma cena, conforme comentado no Capítulo 5, mas também na determinação de seu comportamento mecânico como discutido a seguir. Em OSW-Solid, o método de análise numérica empregado na resolução das equações diferenciais que regem o comportamento mecânico de um sólido é o *método dos elementos de contorno* (MEC) [16]. Este é baseado na divisão do contorno (neste caso, a superfície de um sólido) em elementos discretos denominados *elementos de contorno*. No

MEC, a solução aproximada de determinado problema envolve funções (que dependem do tipo de problema) chamadas *soluções fundamentais*. Uma solução fundamental fornece, em um ponto de amostragem q qualquer, a resposta a uma ação aplicada em um ponto fonte p de um domínio imaginário infinito ou semi-infinito. A contribuição de cada elemento de contorno para solução do problema é obtida integrando-se as soluções fundamentais sobre todos os pontos de contorno (considerados como pontos de amostragem) de todos os elementos de contorno resultantes da discretização. O tipo de integração pode ser classificada em não-singular, quase-singular e singular, de acordo com critérios que dependem, por exemplo, da distância do ponto fonte ao ponto de amostragem e da geometria do elemento de contorno. Para cada tipo de integração pode-se empregar técnicas (numéricas ou analíticas) distintas, cuja eficiência e precisão dependem de outros critérios. Para integração não-singular, por exemplo, a técnica mais usada é e integração Gaussiana. Neste caso, deve-se escolher quantos serão os pontos de Gauss (de amostragem) a serem considerados, em função, digamos, da relação entre a distância entre o ponto fonte p e o centróide do elemento de contorno contendo q e o comprimento da maior aresta do elemento. Em programas de elementos de contorno estes critérios são codificados diretamente no código. Qualquer nova consideração ou modificação de estratégia que possa resultar em ganho de eficiência e/ou precisão requer alteração no código fonte, o que não aconteceria se tais regras fossem codificadas na base de conhecimento de elementos de contorno inteligentes.

Em relação ao mecanismo de inferência, a opção em utilizar um desenvolvido por terceiros ao invés de implementarmos um próprio, deveu-se ao fato de que o amplo escopo da nossa proposta e a complexidade inerente ao trabalho a ser realizado já era grande o suficiente para ocupar todo o tempo disponível para sua execução. O esforço adicional para construirmos um mecanismo de resolução de restrições neste momento seria impraticável. Além disso, pareceu-nos uma boa idéia projetar a solução independente de um mecanismo de resolução particular. Uma aplicação que fizesse uso de objetos inteligentes poderia, sem muita dificuldade, trocar seu mecanismo de resolução por outro que atendesse mais apropriadamente suas necessidades sem, no entanto, precisar alterar o conhecimento codificado nos objetos. Para isso, construímos uma linguagem CLP neutra e incluímos um software capaz de converter as regras CLP na nossa linguagem para regras correspondentes no mecanismo de inferência sendo utilizado.

Contudo, observamos no decorrer do trabalho a dificuldade em se criar uma linguagem realmente neutra, com a qual qualquer conjunto de regras CLP de qualquer mecanismo de resolução pudesse ser escrita. Cada mecanismo de resolução possui uma sintaxe própria, com seus próprios nomes de restrições, operadores e domínios. Cada restrição espera receber argumentos específicos e os operadores possuem sua própria precedência e associatividade. Enfim, a codificação do conhecimento acaba estreitamente relacionada com o mecanismo de resolução, tornando-se inviável seu reaproveitamento.

A introdução de um *middleware*, além de não ter servido para tornar o sistema mais flexível e evidentemente causar uma diminuição na performance quando comparado a uma solução onde não se precise de um, traz outra desvantagem. Todo o código para a comunicação nele implementado fica à mercê dos recursos para integração disponíveis no mecanismo de inferência. Dependendo como são estes recursos, pode-se ter um trabalho muito grande para implementar as funcionalidades de um *middleware*, ou até mesmo não ser possível implementá-las. No ECLiPSe por exemplo, tivemos um trabalho considerável para conseguir fazer funcionar o acesso a atributos e a invocação de métodos através da reflexão. Nos métodos que retornam valor, também houveram dificuldades para fazer com que o valor retornado pudesse ser atribuído a uma variável. Isto é bastante útil numa expressão como `(x = this.método1()) + y`, onde o retorno de um método é utilizado para computar uma expressão.

Outra limitação está relacionada com a base de conhecimento. Nesta implementação de objetos inteligentes a base de conhecimento de um objeto fica por completo armazenada em memória durante seu ciclo de vida. Para aplicações com um número considerável de objetos inteligentes e/ou objetos com um conhecimento muito extenso, isto pode tornar-se proibitivo. Poder-se-ia pensar então na utilização de mecanismos auxiliares para armazenamento do conhecimento, o qual deve continuar sendo acessado com eficiência, porém sem comprometer a memória do sistema.

Apesar da integração realmente facilitar a utilização da CLP em ambientes orientados a objetos, possibilitando até mesmo o acesso a atributos e a execução de métodos durante a ativação das regras, a codificação do conhecimento lógico por aqueles que não conhecem linguagens lógicas e seus mecanismos de execução é ainda uma atividade difícil. Faz-se necessário então a presença de um especialista em sistemas lógicos, preferencialmente um que conheça a linguagem do mecanismo de inferência a ser utilizado, ou ainda a utilização de ferramentas que facilitem a aquisição do conhecimento.

6.2 Trabalhos Futuros

Tendo em vista as considerações feitas sobre a utilização de um *middleware* neste trabalho, queremos propor como trabalho futuro a implementação do conceito de objeto inteligente aqui apresentado, mantendo-se todas as suas características quanto organização e utilização do conhecimento, porém fazendo-se uso de uma máquina de inferência própria. Isto propiciaria uma integração mais adaptada às características dos objetos inteligentes, o acesso a atributos e métodos durante o processamento das regras poderia ser revisto e melhor implementado e todo processamento extra para recodificação das regras poderia ser evitado, resultando certamente em uma performance de execução superior que a da proposta original.

Outra sugestão é a extensão deste trabalho ou a adaptação da proposta de trabalho futuro para a utilização de um sistema gerenciador de banco de dados para o armazenamento do conhecimento. Isto irá tornar a solução mais robusta em relação ao número de regras que poderá tratar, já que estes dispositivos são especializados em manter e recuperar com eficiência grandes quantidades de dados em meios físicos não voláteis, como por exemplo, o disco rígido de um computador.

Sobre a aquisição do conhecimento, apesar da maior facilidade trazida pelas linguagens CLP para se descrever problemas, numa pesquisa realizada em 2000 [53] ficou constatado que uma das principais necessidades para esta tecnologia é torná-la mais fácil de se aprender e utilizar. Além disso, constatamos também aqui a dificuldade em se criar um programa CLP com pouco conhecimento sobre linguagens lógicas de programação. Assim sendo, trabalhos poderiam ser direcionados na tentativa de tornarem a modelagem dos problemas mais fácil e natural, possivelmente através da utilização de interfaces gráficas que possibilitassem a definição visual das regras CLP.

Apêndice A

Gramática da Linguagem CLP

Abaixo segue a sintaxe, em notação BNF (*Backus-Naur Form*), da linguagem CLP.

```
program ::=
   '/*?'  clause_list '*/'

clause_list ::=
   clause
   | clause_list clause

clause ::=
   modifier?  RULE IDENTIFIER rule';'

modifier ::=
   PUBLIC
   | PRIVATE

rule ::=
   fact_list
   | head IMP_OP statement_list

fact_list ::=
   predicate
   | fact_list ',' predicate

head ::=
   predicate

statement_list ::=
   statement
   | statement_list ',' statement

predicate ::=
   IDENTIFIER'(' term_list ')'

statement ::=
   conditional
   | term
```

```
conditional ::=
   IF '(' expression ')' term_list
   | IF '(' expression ')' term_list ELSE term_list

term_list ::=
   term
   | term_list ',' term

term ::=
   expression

expression ::=
   primary_expression
   | unary_expression
   | binary_expression

binary_expression ::=
   expression BIN_OP expression

unary_expression ::=
   UN_OP expression

primary_expression ::=
   predicate
   | ATOM
   | list
   | reference
   | '(' term_list ')'
   | variable
   | constant

reference ::=
   THIS
   | reference '.'  IDENTIFIER
   | reference '(' expression_list ')'
   | reference '(' ')'

list ::=
   '[' term_list ']'

expression_list ::=
   expression
   | expression_list ',' expression

constant ::=
   STRING
   | NUMBER

variable ::=
   IDENTIFIER

IDENTIFIER =
   letter(letter|digit)*

letter =
   [_A-za-z]

digit =
   [0-9]

STRING =
   "char*"
```

```
ATOM =
   'char*'

char =
   qualquer caractere imprimível

NUMBER =
   (+|-)?digit+(.digit+)?((e|E)(+|-)?digit+)?

IMP_OP =
   :-
```

Referências Bibliográficas

[1] Abdennadher, S.; Krämer, E.; Saft, M.; Schmauss M. JACK: A Java Constraint Kit *http://www.pms.informatik.uni-muenchen.de/software/jack*, l.a. 19/02/2004.

[2] Aiba, A.; Hasegawa, R. Constraint Logic Programming Systems - CAL, GDCC and Their Constraint Solvers *Proceedings of International Conference on Fifth Generation Computer Systems (FGCS-92)*, p.113-131, 1992.

[3] Aiba, A.; Sakai K.; Sato, Y.; Hawley, D.J.; Hasegawa R. Constraint Logic Programming Language CAL *Proceedings of International Conference on Fifth Generation Computer Systems (FGCS-88)*, Tokyo, p.263-276, 1988.

[4] Aït-Kaci, H.; Podelski, A. A General Residuation Framework *manuscript*, 1993.

[5] Albert, P. Ilog rules, Embedding rules in C++: results and limits *Proceeding of the OOPSLA94 Workshop on Embedded Object-Oriented Production Systems*, 1994.

[6] Backer, B. De; Beringer, H. Intelligent Backtracking for CLP languages, An Aplication to CLP(R) *Proceedings of the 10th International Conference on Logic Programming*, p.550-563, 1983.

[7] Barták, R. Constraint (Logic) Programming: A Survey on Research and Applications In: Figwer, J. ed. *Proceedings of the 3rd workshop on Constraint Programming for Decision and Control*, 2001.

[8] Barták, R. Constraint Programming : In Pursuit of the Holy Grail *In Proceedings of the Week of Doctoral Students (WDS)*, 1999.

[9] Barták, R. On-Line Guide to Constraint Programming *http://kti.mff.cuni.cz/ bartak/constraints/*, l.a. 18/02/2004, 1998.

[10] Beldiceanu, N.; Contejean, E. Introducing Global Constraints in CHIP *Mathematical Computer Modelling*, v.20, n.12, p.97-123, 1994.

[11] Bell-Northern Research Ltd. *BNR-Prolog User Guide*, 1989.

[12] Bittencourt, G. *Inteligência Artificial: Ferramentas e Teorias*. Editora da UFSC, 1998.

[13] Bomme, P. *Intelligent Objects in Object-Oriented Engineering Environmentes*. École Polytechnique Fédérale de Lausane, 1998. Doctor Thesis.

[14] Borin, E.; Colombo, J.A.; SACCHI, R.P. *Um Sistema de Modelagem de Sólidos*, Campo Grande, 2000. Monografia (Projeto de Graduação) – Universidade Federal de Mato Grosso do Sul.

[15] Borning, A. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory *ACM Transactions on Programming Languages and Systems*, ACM Press, v.3, n.4, p.353-387, 1981.

[16] Brebbia, C.A.; Telles, J.C.F; Wrobel, L.C. *Boundary Element Techniques: Theory and Applications in Engineering*. Springer-Verlag, 1984.

[17] Codognet, P.; Diaz, D. Compiling Constraints in clp(FD) *Journal of Logic Programming*, v.27, n.3, p.185-226, 1996.

[18] Colmerauer,A. An Introduction to Prolog-III. *Communications of the ACM*, p.69-90, 1990.

[19] Dincbas, M.; Hentenryck,P.V.; Simonis, H.; Aggoun,A.; Graf, T.; Berthier, F. The Constraint Logic Programming Language CHIP. *Proceedings on the International Conference on Fifth Generation Computer Systems*, 1988.

[20] Figueira, C.S. JEOPS - Uma ferramenta para o desenvolvimento de aplicações inteligentes em Java *http://www.cin.ufpe.br/ jeops/*, l.a. 23/02/2004, 1999.

[21] Forgy, C.L. RAL/C and RAL/C++: Rule-Based Extensions to C and C++ *Proceeding of the OOPSLA94 Workshop on Embedded Object-Oriented Production Systems*, 1994.

[22] Frühwirth, T. Theory and Practice of Constraint Handling Rules *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, v.37, n.1-3, p.95-138, 1998.

[23] Frühwirth, T.; Abdennadher, S.; Meuss, H. Confluence and Semantics of Constraint Handling Rules *Proceedings of Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, 1996.

[24] Frühwirth, T.; Herold,A.; Küchenhoff, V.; Provost, T.L.; Lim, P.; Monfroy, E.; Wallace, M. Constraint Logic Programming - An Informal Introduction. *European Computer-Industry Research Centre - ECRC*, n.5, 1993.

[25] Graham, I. The SOMA method: Adding rules to classes *Object development methods*, p.199-210, 1994.

[26] Haridi, S.; Janson, S. *Kernel Andorra Prolog and its Computational Model. In Proceedings of the International Conference on Logic Programming*. The MIT Press, 1990.

[27] Hentenryck, P. V. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, MIT Press, 1989.

[28] Hentenryck, P.V.; Saraswat, V. Special issue on Strategic Directions on Constraint Progamming *CONSTRAINTS: An International Journal*, v.2, 1997.

[29] Hentenryck, P.V.; Simonis, H.; Dincbas, M. Constraint Satisfaction Using Constraint Logic Programming *Artificial Intelligence*, 1992.

[30] Henz, M.; Smolka, G.; Wurtz J. Oz - a Programming Language for Multiagent Systems *13th International Joint Conference on Artificial Intelligence*, Chamb'ery, France, Morgan Kaufmann Publishers, v.1, p.404-409, 1993.

[31] Hermenegildo, M. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System *Proceedings of Principles and Practice of Constraint Programming*, n.874, p.123-133, 1994.

[32] IC-PARC. The Eclipse Library Manual *http://www.icparc.ic.ac.uk/eclipse*, l.a.19/02/2004, 1999.

[33] ILOG SA. *ILOG Solver : Object Oriented Constraint Programming*. ILOG SA, 12, Av. Raspail, BP 7, 94251 Gentilly cedex, France, 1995.

[34] Jaffar, J.; Lasses, J.L. Constraint Logic Programming *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, p.111-119, 1987.

[35] Jaffar, J.; Maher, M.J. Constraint Logic Programming : A Survey *Journal of Logic Programming*, v.19/20, p.503-581, 1994.

[36] Jaffar, J.; Michaylov, S.; Stuckey, P.; Yap, R. The CLP(R) Language and System *ACM Transactions on Programming Languages and Systems*, 1992.

[37] Jouannaud, J.P. *Proceedings of the 1st Conference on Constraints in Computational Logic (CCL)*, 845 em LNCS 1994.

[38] Mackworth, A.K. Consistency in networks of relations *Artificial Intelligence*, v.8, p.99-118, 1977.

[39] Mackworth, A.K. Constraint Satisfaction *Artificial Intelligence*, v.1, p.285-293, 1986.

[40] McAffer, J.; O'Grady, T.; Barry, B. ENVY/Expert: An embedded reasoning system in Smalltalk *Proceeding of the OOPSLA94 Workshop on Embedded Object-Oriented Production Systems*, 1994.

[41] Monfroy, E. A Survey of Non-Linear Solvers *European Computer-Industry Research Centre - ECRC*, 1992.

[42] Montanary, U. Networks of constraints: fundamental properties and applications to picture processing *Information Sciences*, p.95-132, 1974.

[43] Mozart Consortium. The Mozart Project *http://www.mozart-oz.org*, l.a. 19/02/2004, 1999.

[44] Object Management Group. *http://www.uml.org*, l.a. 08/03/2005.

[45] Pachet, F. *Représentation de connaissance par objects et règles: le système NéOPUS*. Université Paris VI, 1992. Doctor Thesis – Institut Blaise Pascal.

[46] Pachet, F. *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems*. 1994.

[47] Pagliosa, P.A. *Um Sistema de Modelagem Estrutural Orientado a Objetos*. São Carlos, 1998. Tese (Doutorado) – Escola de Engenharia de São Carlos – USP.

[48] Pountain, D. Constarint Logic Programming *Byte Magazine*, 1995.

[49] Preparata, F.P.; Shamos, M.I. Computational Geometry : An Introduction *Springer-Verlag*, 1985.

[50] Prestwich, S. An Informal Tutorial on Search Techniques in Constraint Programmimg *National University of Ireland at Cork*, .

[51] Puget, J.F. A C++ Implementation of CLP *Proceedings of the Second Singapore International Conference on Intelligent Systems*, 1994.

[52] Régin, J.Ch. A filtering algorithm for constraints of difference in CSPs *Proceedings of the AAAI 12 th National Conference on Artificial Intelligence*, p.362-367, 1994.

[53] Rossi, F. Constraint (Logic) Programming: A Survey on Research and Applications *In Proceedings of ERCIM/Compulog Net workshop on constraints*, 2000.

[54] Russel, S.; Norvig, P. *Artificial Intelligence - A Modern Approach* Prentice Hall Series, 1995.

[55] Saraswat, V.A. *Concurrent Constraint Programming*. The MIT Press, 1993.

[56] Steele, G.L. The definition and implementation of a computer programming language based on constraints *Technical Report MIT-AI TR 595*, Dept. of Electrical Engineering and Computer Science, M.I.T., 1980.

[57] Sun Network *http://java.sun.com/j2se/1.3/docs/guide/jni/*, l.a.08/03/2005.

[58] Sun Network *http://java.sun.com/docs/books/tutorial/essential/io/providing.html*, l.a.03/11/2004.

[59] Sutherland, I. A Man-Machine Graphical qCommunication System *Proceedings of the Spring Joint Computer Conference*, v.23, p.329-346, 1963.

[60] The Constraint Programming Working Group. Constraint Programming. *ACM-MIT SDCR Workshop* , 1996.

[61] Tsang, E.P.K. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1995.

[62] Van Hentenryck, P.; Deville, Y. Operational Semantics of Constraint Logic Programming Over Finite Domains *Proceedings of the Symposium on Programming Language Implementation and Logic Programming*, n.528, p.395-406, 1991.

[63] Van Hentenryck, P.; Saraswat, V.; Deville, Y. Design, Implementations and Evaluation of the Constraint Language cc(FD) *Constraint Programming: Basics and Tools*, v.910, p.293-316, 1995.

[64] Voda, P. *The constraint language trilogy: Semantics and computations*. Technical Report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.

[65] Wallace, M. Constraint Programming *http://www.icparc.ic.ac.uk/eclipse/reports /handbook/handbook.html*, l.a.18/02/2004, 1995.

[66] Wallace, M.; Novello, S,; Schimpf, J. *Eclipse : a plataform for constraint logic programming*. IC-Parc, Imperial College, London, 1997.

[67] Waltz, D. Generating Semantic Descriptions from Drawings of Scenes with Shadows *Technical Report AI271*, Massachusetts Institute of Technology, 1972.