Universidade Federal de Pernambuco

Graduação em Ciência da Computação

Centro de Informática

CEOPS++ - Integração entre Objetos e Regras de Produção em C++

Aluno: Pablo de Santana Barbosa (psb@cin.ufpe.br)

Orientador: Geber Lisboa Ramalho (glr@cin.ufpe.br)

Recife, Outubro de 2006.

Resumo

Nos últimos tempos o uso de Inteligência artificial em jogos eletrônicos tem se intensificado cada vez mais. A qualidade da IA presente influi diretamente na imersão do jogador no ambiente de jogo. E apesar dos sistemas baseados em regras serem os mecanismos prediletos dos desenvolvedores para a implementação dessa IA, o número de motores de inferência *open source* é bastante reduzido e as implementações existentes possuem alguns inconvenientes.

O intuito desse trabalho foi o de construir um motor de inferência, em C++, baseado no Jeops. O Jeops, desenvolvido no CIn e baseado em Java, promove uma excelente integração entre regras e objetos. Com isso achamos que seria muito interessante levar essas facilidades ao mundo de C++, que é a linguagem padrão para desenvolvimento de jogos.

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

Há algum tempo os jogos eletrônicos já não são mais vistos como meros passatempo. Hoje eles fazem parte de um mercado milionário de entretenimento, chegando a concorrer com a indústria cinematográfica, seja em faturamento, utilização de efeitos especiais e gastos com publicidade. Esse mercado teve um faturamento de US$7 bilhões em 2005, e existem estimativas de que esse valor possa chegar a US$47 bilhões em 2010. A próxima geração de consoles é apontada como fator primordial para expansão desse mercado [17].

Esses valores também se refletem nos investimos que empresas como Sony e Microsoft fazem no desenvolvimento de consoles. A Sony e a Toshiba, por exemplo, gastaram em torno de US$2 bilhões para desenvolver o Playstation 2, enquanto que a Microsoft investiu em torno de US500 milhões só no marketing do seu Xbox, ambos consoles da geração passada.

As melhorias mais notáveis em decorrência desses investimentos foi com relação aos gráficos dos jogos, com o número de polígonos renderizados em uma cena crescendo quase que exponencialmente a cada ano. O impacto dessas melhorias é evidenciado no realismo dos ambientes dos jogos atuais. Esses ambientes são populados tanto com personagens controlados por humanos quanto por computadores (NPC's).

Entretanto de uns tempos pra cá, os desenvolvedores começaram a se preocupar tanto com esse relacionamento entre personagens controlados por humanos e NPC's, quanto com a qualidade visual do jogo. Eles começaram a perceber que a qualidade da inteligência artificial assume papel decisivo na imersão do jogador no ambiente lúdico de jogo, aumentando também seu divertimento ao jogar. A IA passou então a ser uma exigência cada vez maior do mercado e um dos pontos cruciais para o sucesso do produto.

Para implementar mecanismos de raciocínio de forma fácil, é fundamental utilizar um motor de inferência. O motor mais comum é aquele baseado em regras de produção. Neste caso, basta o desenvolvedor escrever as regras, pois o motor cuida de como e quando elas serão disparadas.

Existem alguns motores de código aberto hoje, incluindo o Jeops, desenvolvido no CIn e baseado em Java, que promove uma excelente integração entre regras e objetos. Nesta mesma abordagem de integração não existem motores em C++, que é a linguagem padrão para desenvolvimento de jogos, devido à exigência de alto desempenho.

O objetivo deste trabalho é, baseado no Jeops, propor e implementar um motor cuja linguagem hospedeira seja C++ de forma a facilitar a incorporação de IA em jogos digitais.

2 Estado da arte

Este capítulo apresenta alguns motores de inferência de código aberto disponíveis no mercado e tenta explicar um pouco do funcionamento destes motores. Mas antes apresentaremos brevemente uma introdução sobre sistemas de produção, orientação a objetos e uma proposta de integração entre esses paradigmas.

2.1 Raciocínio a base de regras e Jogos

Conforme os jogos foram ficando mais complexos, seus consumidores passaram a exigir agentes controlados por computador mais sofisticados, obrigando os desenvolvedores a prestarem mais atenção nos aspectos de inteligência artificial de seus jogos. Atualmente todos os jogos possuem algum tipo de IA, sendo as técnicas utilizadas as mais diversas: *scripts*, sistemas a base de regras, redes neurais, algoritmos genéticos, *fuzzy*, etc.

Um dos mecanismos prediletos dos desenvolvedores de jogos são os sistemas baseados em regras (RBS). RBS's podem ser utilizados para codificar o conhecimento a respeito de um cenário particular dentro do ambiente de jogo e o comportamento dos personagens quando inseridos nesse ambiente. No entanto, mesmo em jogos simples, as regras que governam o comportamento das entidades com o mundo virtual precisam expressar relações complexas com um número potencialmente grande de aspectos do estado jogo, que por sua vez pode estar mudando constantemente.

Sendo assim pode se dizer que um RBS's para jogos deve primar pela facilidade de uso e performance, tendo em vista as inevitáveis dificuldades que os desenvolvedores devem enfrentar e essas complexas relações com os vários aspectos do jogo.

Alguns princípios que atendem essas necessidades são:

1) Uniformidade de integração: quanto mais fácil e natural for a integração entre o motor e o ambiente de desenvolvimento, menos trabalho terá o programador que terá mais tempo em se preocupar com outros aspectos importantes do jogo.
2) Performance: o tempo de resposta é fator crucial para imersão do jogador no ambiente lúdico de jogo, portanto a performance é um requisito fundamental.

2.2 Sistemas a base de regras

Programação baseada em regras é uma técnica bastante usada no desenvolvimento de sistemas especialistas[1]. Nesse paradigma, regras são utilizadas para representar conhecimento heurístico do mundo, especificando um conjunto de ações a serem executadas em uma situação específica. Por isso, a modelagem da informação acontece num nível mais alto de abstração, quando comparado com as linguagens convencionais de programação.

Sistemas que utilizam regras de produção como forma de representar o conhecimento recebem uma denominação genérica de Sistemas de Produção[1]. Na verdade é possível definir um sistema de produção como:

1) um conjunto de produções, regras, que representam a principal maneira de armazenar o conhecimento heurístico do mundo;
2) uma memória de trabalho, onde são armazenados os fatos, que são usados para representar informação;
3) e por fim um algoritmo responsável por produzir novos fatos a partir dos antigos.

A memória de trabalho é a estrutura responsável por armazenar e manipular um conjunto de fatos. Estes fatos são utilizados para armazenar informações, sendo também a unidade fundamental de dados utilizada pelas regras de produção.

As regras de produção, como já foi dito, são utilizadas para armazenar o conhecimento heurístico do mundo (também é possível representar outras formas de conhecimento) [1]. Elas especificam um conjunto de ações a serem tomadas com base num conjunto de percepções, ou fatos na memória de trabalho. As regras são formadas por um antecedente e um conseqüente. O antecedente, também chamado de parte *if* ou *left-hand-side*, define um conjunto de condições que devem ser satisfeitas. O conseqüente, também chamado de parte *then* ou *right-hand-side*, define um conjunto de ações a serem tomadas. O conseqüente é unificado com o conjunto de fatos da memória de trabalho e, quando todas as condições são satisfeitas, a regra é disparada. O resultado da execução de uma regra será de atualizar, remover ou adicionar dados à memória de trabalho. Isso de acordo com o especificado pelo seu conjunto de ações. A Tabela 1 mostra um exemplo de regra.

```
Se
     Está chovendo
     E possuo um guarda-chuva
Então
    Usar guarda-chuva
```

Tabela 1 Exemplo de Regra em português

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

1) Unificação

Os fatos, presentes na memória de trabalho, são unificados com as condições das regras. Quando um determinado conjunto de fatos torna verdade todas as condições de uma regra, a unificação foi bem sucedida e esse conjunto de fatos mais a regra são inseridos no conjunto de resolução de conflitos.

2) Resolução de Conflitos

Podemos chamar de **Conjunto de Conflitos** o conjunto formado por pares <regra, fatos>, onde uma regra tem todas as suas pré-condições satisfeitas por um determinado conjunto de fatos. Então se faz necessário o uso de algum tipo de estratégia de resolução de conflitos para decidir qual regra deverá ser disparada. A estratégia pode ser tão simples quanto escolher a primeira regra da lista e dispará-la, ou utilizar alguma heurística complicada para a escolha da melhor regra.

3) Disparo da Regra

Onde todas as ações definidas pela regra escolhida pela política de resolução de conflitos são executadas.

Os passos são repetidos até que o conjunto de conflitos esteja vazio. A Figura 1 ilustra esses passos de execução.

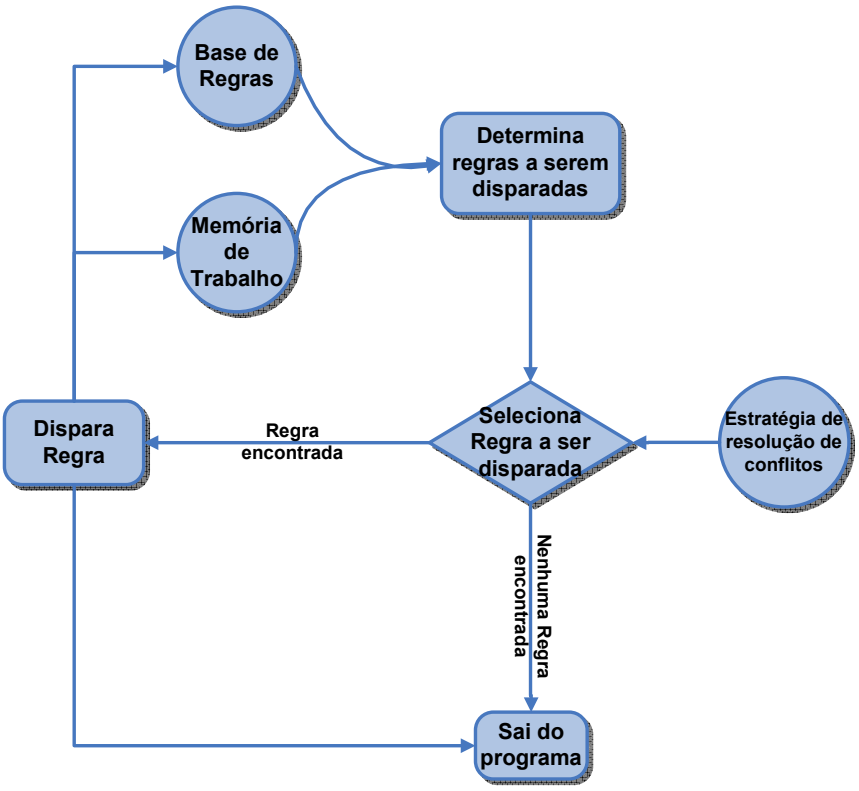


Figura 1 Ciclo de execução de um motor de inferência com encadeamento progressivo

2.3 Orientação a Objetos

Programação orientada a objetos é um paradigma de programação que surgiu no final da década de 60. Por essa época o nascente campo da engenharia de software começava a discutir a idéia da "crise do software" [5]. Esse termo foi usado para descrever o impacto do rápido aumento do poder computacional e da complexidade dos softwares desenvolvidos. O que houve foi que o grande aumento da capacidade computacional tornava viável também o aumento da complexidade das aplicações que eram desenvolvidas para esses computadores[6]. O software resultante era então bem mais complexo que os precedentes.

Devido à imaturidade da engenharia de software e dos processos de desenvolvimento, os programas desenvolvidos naquela época eram extremamente difíceis de escrever, corrigir e modificar.

A crise de software se manifestou de diversas maneiras:

- Os projetos facilmente extrapolavam seus orçamentos;
- O tempo de desenvolvimento, com muita freqüência, ultrapassava o estimado;
- Baixa qualidade dos softwares;
- Os softwares não estavam de acordo com os requisitos;
- Difícil gerenciamento dos projetos e manutenção do código.

A técnica de programação orientada a objetos foi uma das propostas surgidas à época e se apresenta atualmente como uma boa alternativa para o desenvolvimento de grandes aplicações.

O modelo de orientação a objetos é baseado em quatro elementos principais (Abstração, Encapsulamento, modularidade, hierarquia) e três secundários (tipagem, concorrência e persistência) [7]. Como secundário podemos entender que são os elementos que são úteis, mas não essenciais ao modelo.

- **Abstração**

Abstração consiste na prática em reduzir certos detalhes, de maneira a podermos trabalhar focados em poucos conceitos.

Em orientação a objetos, o processo de abstração envolve identificar o comportamento crucial de um objeto e eliminar detalhes irrelevantes e tediosos[8]. O objetivo é simplificar a complexidade da realidade, modelando classes que sejam apropriadas para resolução do problema. Essas classes definem um conjunto de elementos do mundo que apresentam características comuns. Uma classe define uma abstração de alguma coisa, incluindo suas características (atributos) e seu comportamento (métodos).

Então é possível dizer que a complexidade é gerenciada através da abstração, pois seu foco está na visão externa dos objetos, separando seu comportamento da implementação.

- **Encapsulamento**

Em programação orientada a objetos, as características (atributos) e o comportamento estão encapsulados dentro das classes. Embora objetos possam se comunicar através de interfaces bem definidas, eles não têm permissão para saber como outros objetos são implementados. Esses detalhes ficam ocultos dentro dos próprios objetos.

O encapsulamento é importante para a manutenção de um estado interno consistente, o que não

seria possível caso suas informações pudessem ser acessadas diretamente[1].

- **Modularidade**

Modularidade pode ser definida como a capacidade de particionar um programa em componentes individuais na tentativa de reduzir sua complexidade em algum grau[9]. Um módulo agrupa um conjunto coeso de subprogramas e estruturas de dados. Cada módulo pode ser compilado separadamente, o que possibilita seu reuso e também possibilita que vários programadores trabalhem em diferentes módulos separadamente. Módulos também provêm encapsulamento, facilitando o entendimento de sistemas mais complexos.

- **Hierarquia**

Hierarquia pode ser definida simplesmente como uma ordenação de abstrações[7]. Uma das formas de obtermos hierarquia é através de herança, pela qual podemos definir uma nova classe que herda comportamento de uma classe já existente. Como exemplo podemos dizer que um carro e uma bicicleta são veículos. Portanto, em programação orientada a objetos, veículo define o comportamento comum a todos os veículos e cada subclasse (carro e bicicleta) deve prover o comportamento específico de cada tipo de veículo.

2.4 EOOPS

O termo EOOPS (de *Embedded Object Oriented Production Systems*) é usado para caracterizar os sistemas criados que integram linguagens orientadas a objetos com sistemas de produção[1]. A idéia por trás da integração de objetos com os sistemas de produção é juntar as vantagens de sistemas inteligentes (que historicamente estavam restritos a aplicações acadêmicas), com toda a estrutura adquirida pelas linguagens orientadas a objetos durante todos os anos de pesquisa da Engenharia de Software.

Com a integração entre estes dois paradigmas distintos, os fatos são substituídos por objetos, e a resolução de um problema passa a utilizar conceitos de ambos os mundos, dos objetos e das regras. Com essa mudança o casamento entre regras e fatos, que era realizado de maneira estrutural (caractere por caractere), passa a ser realizado de maneira comportamental (envio de mensagens). Assim, o usuário pode tirar proveito dessa característica pois é possível encapsular a complexidade.

Isso se deve ao fato de ser utilizada uma chamada de método de um objeto, que pode se desdobrar em várias outras, além dele poder usufruir todos os serviços embutidos nos objetos, que por sua vez podem ter sidos reaproveitados de outros projetos do próprio usuário ou de terceiros.

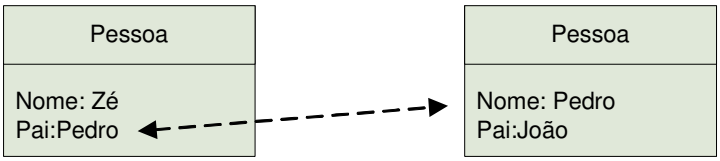


Figura 2 Exemplo de ligação estrutural entre os fatos

Um dos problemas existentes com essa abordagem é que devido aos objetos serem estruturas encapsuladas, não é possível saber quando eles são modificados. Esse problema é conhecido como Problema da modificação dos objetos.

2.5 Motores Open Source

Nesta seção apresentamos alguns motores de inferência *open source* disponíveis no mercado. Para esse fim escolhemos três motores: o CLIPS, por ser um projeto que tenta unir regras e objetos e, apesar de ter tido seu desenvolvimento descontinuado, ainda é bastante utilizado; o Soar, que apesar na noção de objeto ser tratada de maneira diferente por este motor, existem muitos projetos de jogos que o utilizam; e por fim o Jeops, por considerarmos que possui a melhor abordagem entre objetos e regras e por ser base para esse trabalho.

2.5.1 CLIPS

Até 1984 o setor de Inteligência Artificial da NASA já havia produzido dúzias de protótipos de sistemas especialistas. Entretanto, apesar do potencial demonstrado por esses sistemas, poucas destas aplicações entraram realmente em uso. O problema estava relacionado ao uso de LISP como linguagem base de praticamente todos os sistemas especialistas desenvolvidos. Em particular, três problemas estavam relacionados ao uso de LISP: a baixa disponibilidade da linguagem para a maioria dos computadores convencionais, o alto custo das ferramentas para se desenvolver nessa linguagem e a fraca integração entre LISP e outras linguagens[10].

O setor de Inteligência artificial percebeu que o uso de linguagens mais convencionais, como C, poderia resolver boa parte de seus problemas. Então, em 1985 surgiu o primeiro protótipo

do CLIPS (C *Language Integrated Production System*). Sua sintaxe é baseada em LISP e nada se assemelha com a linguagem hospedeira C.

Existem três maneiras de representar o conhecimento no CLIPS: regras, usadas principalmente para a representação de conhecimento heurístico, baseado em experiência; funções específicas e funções genéricas, para representação de conhecimento procedimental; ou programação orientada a objetos, também usada na representação de conhecimento procedimental.

CLIPS foi desenvolvido para facilitar a integração entre sistemas de produção e linguagens convencionais (como C, C++ e Ada). É possível, por exemplo, executar o CLIPS a partir de uma função ou método em C++, e quando o motor terminar de executar ele retornará o controle de volta ao programa principal. Também é possível definir código procedimental como funções externas, que podem ser chamadas a partir do CLIPS.

Apesar de desenvolvido para facilitar a integração entre sistemas de produção e linguagens convencionais, o CLIPS define uma linguagem própria chamada de CLIPS *Object-Oriented Language* (COOL). Isso implica que é possível que sistemas sejam inteiramente desenvolvidos usando essa linguagem, que também provê funções de linguagens convencionais.

No CLIPS há dois tipos de dados que podem estar presentes na memória de trabalho, fatos e classes. Fatos são definidos pelo comando *deftamplate*, que define um agrupamento de valores.

```
(deftemplate pessoa
    (slot name)
    (slot age)
    (slot weight)
    (slot height)
    (multislot blood-pressure) )
```

Tabela 2 Definição de template no CLIPS

Fatos são adicionados à memória de trabalho com o comando *assert*, são removidos com o comando *retract* e modificados com o comando *modify*.

```
(assert (pessoa (name Andrew) (age 20) (weight 80)
    (height 188) (blood-pressure 130 80)))
```

Tabela 3 Inserção de um fato na memória de trabalho

As classes são construídas com o comando *defclass*, através do qual é possível informar vários detalhes sobre a classe que está sendo criada, como o nome da classe, as propriedades dos

objetos (*slots*), seu comportamento (*message-handlers*), uma lista de superclasses, informações se é permitido ou não uma criação direta da classe (classes abstrata).

```
(defclass pessoa (is-a USER)
   (role concrete)
   (slot name (create-accessor read-write)
   (slot age (type NUMBER)
   (visibility public))
   (defmessage-handler pessoa print-name ()
     printout t "Name: " (send ?self get-name) crlf))
```

Tabela 4 Definição de uma classe no CLIPS

No CLIPS as classes são instanciadas através do comando *make-instance*. Cada objeto possui um nome, que pode ser usado para referenciar esse objeto.

```
(make-instance [Pessoa1] of pessoa
   (name "Joao") (age 18))
```

Tabela 5 Inserção de um novo objeto na memória de trabalho

As regras são definidas através do comando *defrule*. O CLIPS permite redefinir uma regra, nesse caso a regra anterior de mesmo nome é removida. O código abaixo ilustra um exemplo de regra:

```
(defrule aniversario
    ?fato1 <- (aniversario ?nome)
    ?fato2 <- (pessoa (name ?nome) (age ?idade))
    =>
    (modify ?fato2 (idade (+ ?idade 1)))
    (retract ?fato1) )
```

Tabela 6 Exemplo de regra no CLIPS

Uma vez que as regras já foram definidas e a memória de trabalho já está preparada, o CLIPS está pronto para ser executado. Todas as regras que têm suas pré-condições satisfeitas são colocadas numa lista chamada de *agenda*. A agenda é similar a uma pilha. Quando uma regra é ativada, ela é posta na agenda ordenada por algum fator. O CLIPS provê sete estratégias de resolução de conflitos*: Depth Strategy*, *Breadth Strategy*, *Simplicity Strategy*, *Complexity Strategy*, *LEX Strategy*, *MEA Strategy*, *Random Strategy*.

2.5.2 Soar

O Soar foi desenvolvido para ser uma arquitetura para construção de sistemas inteligentes. Está em uso desde 1983, e atualmente se encontra na versão 8.6. Os objetivos dos desenvolvedores do Soar era que ele fosse uma arquitetura na qual seria possível [12]: a) ser utilizado na construção de sistemas que realizam uma grande variedade das tarefas realizadas por agentes inteligentes, desde a rotina mais simples até a resolução de problemas extremamente difíceis; b) interagir com o mundo externo; c) aprender com os vários aspectos das tarefas executadas e sua performance sobre essas tarefas.

Para atingir esses objetivos, os designers do Soar se basearam em dois princípios:

1) O número de mecanismos distintos na arquitetura deve ser minimizado. No Soar há apenas uma representação do conhecimento permanente (produções), uma única representação do conhecimento temporário (objetos com atributos e valores), um único mecanismo para geração de objetivos (*automatic subgoaling*), e um único mecanismo de aprendizado (*chunking*).
2) Todas as decisões são tomadas com base em um conhecimento relevante, em tempo de execução. Cada decisão é baseada na interpretação atual dos dados percebidos e em qualquer conhecimento relevante recuperado da memória permanente.

Para resolução de um problema, o Soar usa o conhecimento a respeito do estado inicial do mundo, o conhecimento a respeito do estado ou estados a serem alcançados e o conhecimento a respeito de quais operadores podem ser utilizados. Seu design é baseado na hipótese de que o comportamento guiado por objetivos pode ser mapeado como uma seleção e aplicação de operadores a estados. O estado representa a situação atual do problema. Os operadores modificam esse estado. Então, durante a execução, operadores são continuamente selecionados e aplicados, de forma a transformar o estado inicial em um estado desejado [12]. A Figura 1 mostra o processo de execução dos Soar.

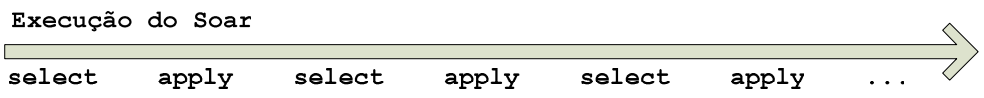


Figura 3 Processo de execução do Soar

As proposições no Soar são organizadas em torno da idéia de seleção e aplicação de operadores. Existem então cinco tipos de proposições:

1) ***Operator Proposal*:** Conhecimento para definir que operador é apropriado na situação atual.

```
sp {agent*propose*usar-guarda-chuva
   (state <s> ^name agent
              ^io.input-link <input>)
   (<input> ^clima <clima>)
   (<clima> ^tempo << CHUVA NEVE >>)
-->
   (<s> ^operator <op> +)
   (<op> ^name usar-guarda-chuva)
}
```

Tabela 7 Proposição de operador no Soar

2) ***Operator Comparison*:** Conhecimento para comparar os operadores candidatos, utilizado para definir qual o melhor operador a ser utilizado na situação atual.

```
sp {agent*compare*usar-guarda-chuva
   (state <s> ^name agent
              ^operator <op1> +
              ^operator <op2> +)
   (<op1> ^name usar-guarda-chuva)
   (<op1> ^name usar-capa)
-->
   (<s> ^operator <op1> > <op2>)
}
```

Tabela 8 Comparação de operadores no Soar

3) ***Operator Selection*:** Conhecimento para selecionar um único operador baseado nas comparações. Não existe uma produção para essa etapa, ela é realizada pelo motor de inferência.

4) ***Operator Application*:** Conhecimento de como um operador específico modifica o estado.

```
sp {agent*apply*usar-guarda-chuva
   (state <s> ^name agent
              ^operator <op>
              ^io.output-link <output>)
   (<op> ^name usar-guarda-chuva)
-->
   (<output> ^nova-acao <acao>)
   (<acao> ^name usar-guarda-chuva)
}
```

Tabela 9 Aplicação de operador no Soar

5) ***State Elaboration*:** Modificam o estado, afetando indiretamente como os operadores são selecionados, comparados e aplicados.

```
sp {elaborate*state*io
   (state <s> ^superstate.io <io>)
-->
   (<s> ^io <io>)
}
```

Tabela 10 Exemplo de elaboração de estado no Soar

Os resultados dessas produções sobre a memória de trabalho podem ser permanentes ou não. As produções dos tipos *operator proposal*, *operator comparison* e *state elaboration* são ditas do tipo *I-support* (*instantiation support*). Isso quer dizer que quando as condições que fizeram com essas regras fossem disparadas deixam de ser válidas, o resultado dessas regras sobre a memória de trabalho é desfeito. Já o resultado de uma proposição do tipo *O-support* (*operator support*) persiste mesmo quando as condições que fizeram com que a regra fosse disparada são invalidadas. *Operator application* é a única regra do tipo *o-support*.

A memória de trabalho representa a situação atual do mundo. Ela contém elementos chamados *Working Memory Elements* (*WME*'s). Cada *WME* representa um pedaço de informação. Um *WME* é uma lista consistindo de três símbolos: identificador, atributo e valor. A Tabela 11 mostra como se organizam esses símbolos em um *WME*.

```
(identifier ^attribute value)
```

Tabela 11 Símbolos de um *WME*

O identificador é um símbolo interno gerado pelo Soar. Atributos e valores podem ser identificadores ou constantes. O grupo de *WME*'s que compartilham o mesmo identificador é

chamado de objeto. Assim cada *WME* pode ser visto como um atributo de um objeto. Todos os objetos na memória de trabalhos estão ligados por algum *WME*, que por sua vez devem estar ligados a algum estado, direta ou indiretamente. Todos esses conceitos estão ilustrados na Figura 4.

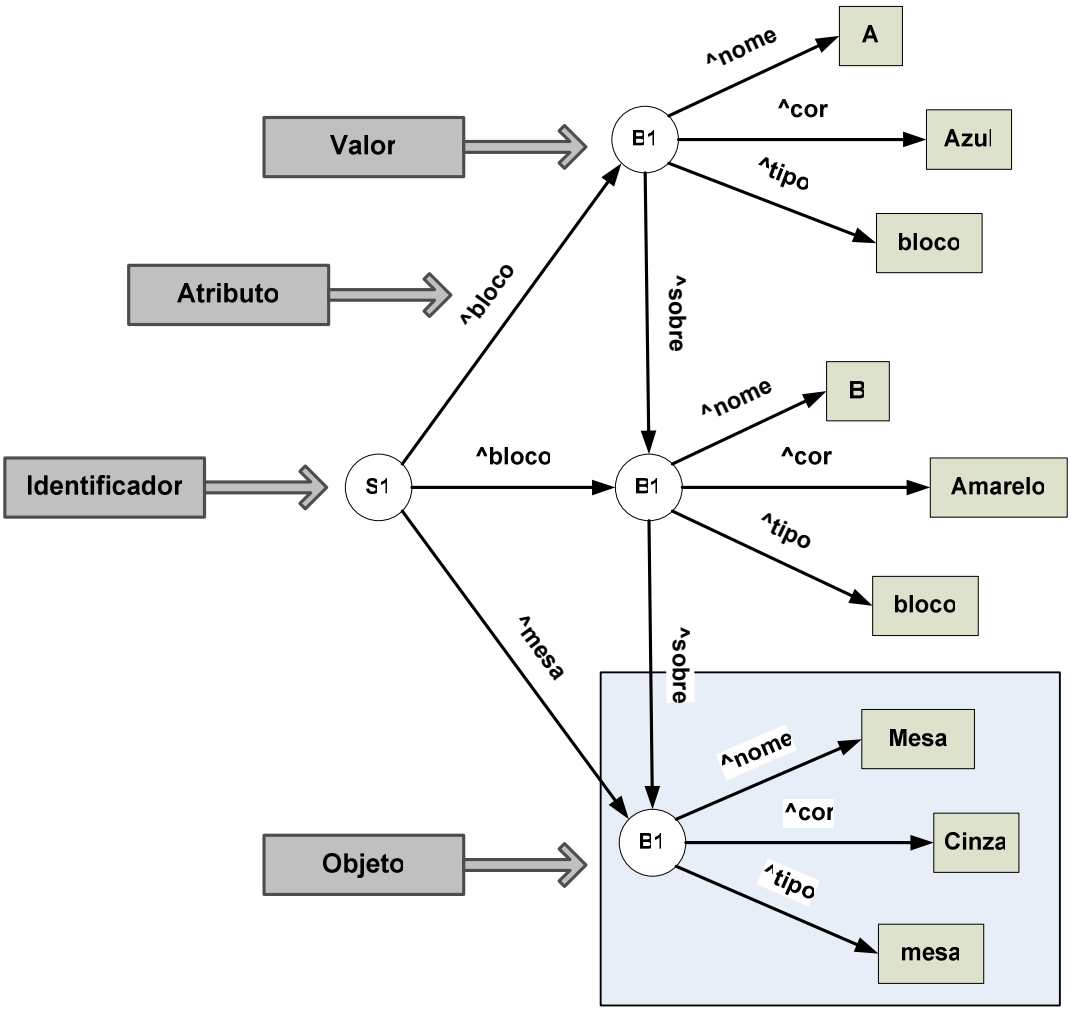


Figura 4 Elementos presentes na memória de trabalho do Soar

A execução do Soar se dá através de um determinado número de ciclos. Se considerarmos a execução sem sub-estados, cada ciclo possui cinco fases:

1) ***Input*:** Novos dados, percebidos pelos sensores, são acrescentados à memória de trabalho.
2) ***Proposal*:** Produções são disparadas (ou removidas) de acordo com esses novos dados (*state elaboration*). Remover uma regra significa desfazer o seu resultado sobre a memória de trabalho. Isso acontece porque todas as regras dessa fase são do tipo *I-support*. Operadores são propostos (*operator proposal*) e em seguida comparados (*operator comparison*). Todas

as produções ativas são disparadas em paralelo (a remoção das regras também é feita em paralelo). A fase continua até que nenhuma regra seja disparada ou removida.

3) ***Decision*:** Um novo operador é selecionado, ou um impasse é detectado e um novo estado é criado.

4) ***Application*:** Produções disparam para aplicar operadores (*operator application*). A ação dessas produções é do tipo *O-support*. Regras do tipo *elaborate state* também podem ser disparadas ou removidas em decorrência das modificações feitas pelas regras do tipo *operator application*.

5) ***Output*:** Comandos de saída são enviados para o ambiente externo.

A Figura 5 a seguir exemplifica os passos de execução descritos anteriormente.

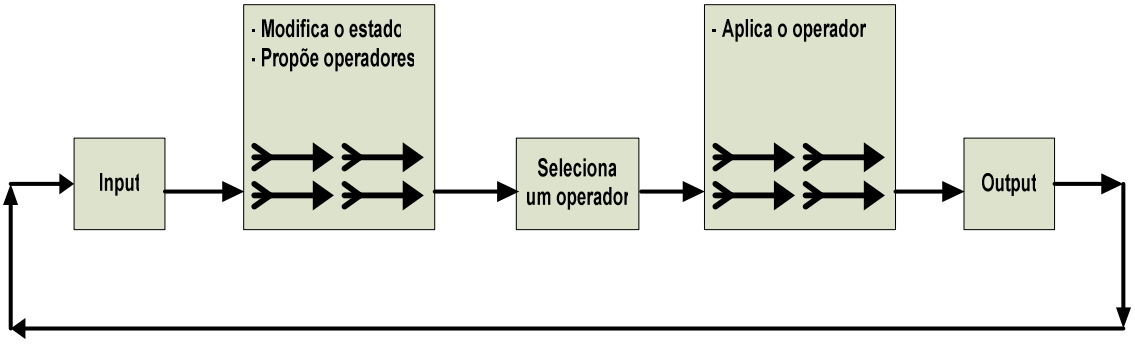


Figura 5 Fases da execução do Soar

2.5.3 Jeops

O Jeops (Java *Embedded Object Production System*) foi desenvolvido com o intuito de fornecer a linguagem Java o poder dos sistemas de produção. A idéia inicial surgiu em 1997, em uma disciplina da graduação do Centro de Informática (CIn) da UFPE. Ao término da disciplina, Figueira, autor do JEOPS, se interessou pelo projeto e continuou seu desenvolvimento sob orientação do professor Geber Ramalho, tendo início a partir daí um projeto de iniciação científica que em seguida se transformou em um trabalho de graduação. Tendo em vista que muito já se avia

desenvolvido, mas que ainda havia muita coisa por fazer, principalmente para que o sistema pudesse ser utilizado em aplicações de grande porte, Figueira se utilizou de um curso de mestrado como forma de aprofundar os diversos conceitos que fundamentavam seu trabalho [1], tendo surgido a partir deste último a versão atual do Jeops.

A idéia de Figueira era de criar um sistema de fácil uso e de acordo com o paradigma de orientação a objetos, tentando obter o máximo de integração entre as regras e a linguagem hospedeira.

Os elementos que podem ser inseridos na memória de trabalho são quaisquer objetos da linguagem Java, não havendo restrição aos objetos que podem ser inseridos na memória.

O Jeops utiliza uma sintaxe bastante parecida com Java para expressar as regras de produção. Na definição de uma regra há uma separação explícita de seus campos: declarações de variáveis, condições e ações. As regras são organizadas dentro de uma base de regras, que também pode conter métodos e declarações de variáveis. A Tabela 12 mostra um exemplo de regra.

```
rule BaseCase {
  declarations
    Fibonacci f;
  conditions
    f.getN() <= 1;
    f.getValue() == -1;
  actions
    f.setValue(f.getN());
    modified(f);
}
```

Tabela 12 Exemplo de regra no JEOPS

Como as regras utilizam objetos, é preciso que estes objetos já estejam definidos quando esta for pré-compilada (o processo de pré-compilação será descrito mais adiante). As declarações das regras definem as variáveis que devem ser unificadas com fatos da base de objetos.

As condições da regra podem ser quaisquer expressões booleanas de Java. Mas atualmente apenas um subconjunto dessas expressões é aceito pelo Jeops. Em geral, essas condições são comparações ou chamadas de métodos que retornam valores booleanos. Os métodos chamados nas pré-condições de uma regra não devem alterar o estado de seu objeto (devem apenas consultar o estado do objeto). Isso porque as diversas avaliações dos métodos podem ser feitas sem que a regra seja disparada uma única vez com o objeto (porque uma outra pré-condição pode ser falsa, por exemplo).

Na parte de ações das regras podem ser utilizadas quaisquer expressões Java, além da manipulação dos objetos da base de fatos. Variáveis locais podem ser declaradas neste campo, para melhor estruturar suas ações. Através dos comandos *insert* e *retract*, objetos podem ser inseridos e removidos da base, respectivamente. Modificações nos objetos devem ser informadas ao sistema pelo usuário através do comando *modify*.

Uma vez definido o conjunto de regras, é necessário realizar a pré-compilação do arquivo da base de regras. Após essa etapa é gerado um arquivo que contém o mesmo nome da base de regras, representando a base de conhecimentos, e uma classe representando a base interna de regras. A Figura 6 ilustra os seguintes passos utilizados para o processo de pré-compilação: 1) Primeiro o compilador Jeops verifica se existe alguma classe utilizada pelas regras que ainda não tenha sido compilada, compilando-as se for o caso; 2) Em seguida o arquivo de regras (extensão "rules"), passado como parâmetro, é convertido para um arquivo Java contendo a base de regras e a base de conhecimento; 3) Por fim o arquivo Java gerado é compilado.

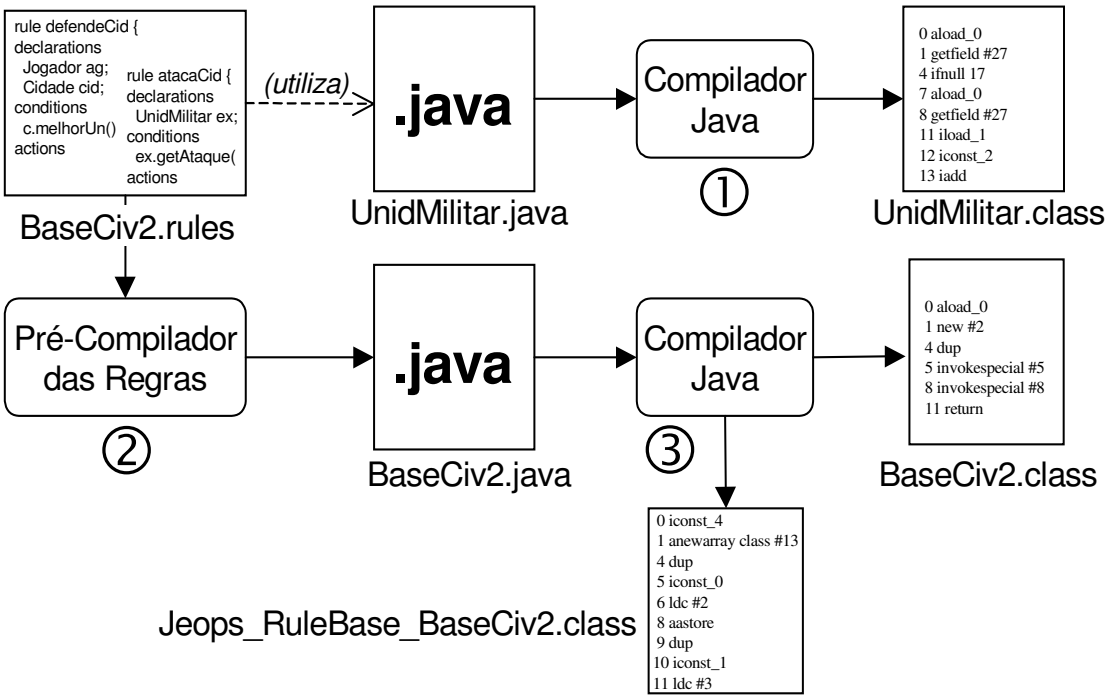


Figura 6 Etapas da pré-compilação do arquivo de regras. Fonte [1]

Na arquitetura do Jeops existe uma clara separação entre a memória de trabalho e a base de regras. As regras que têm todas as suas pré-condições verdadeiras são inseridas em um conjunto de

conflito juntamente com os objetos que a ativaram. Quando o motor de inferência é executado, ele solicita continuamente um elemento do conjunto de conflitos e executa a regra com seus objetos correspondentes. O processo continua até que o conjunto de conflito esteja vazio.

3 CEOPS++

Como foi visto, atualmente existem vários componentes de software que provêem a integração entre regras de produção e linguagens orientadas a objetos. Na programação orientada a objetos os programas são modularizados em classes, que definem a estrutura e o comportamento das instâncias dessas classes, os objetos. A idéia principal dessa integração é juntar as vantagens de sistemas inteligentes com toda a estrutura adquirida pelas linguagens orientadas a objetos [1].

As várias bibliotecas ou componentes de software mostrados na seção 2.5 foram desenvolvidos com o intuito de prover essa integração entre regras de produção e linguagens orientadas a objetos. Dentre estes, o JEOPS, desenvolvido no CIn-UFPE, foi um dos que conseguiu atingir esse objetivo de maneira mais natural, obtendo um excelente nível de integração entre regras, fatos e a linguagem de programação, nesse caso, a linguagem Java. No Jeops, a sintaxe para definição de regras é muito parecida com a da linguagem hospedeira, o que facilita muito a vida do programador que não precisa aprender outra linguagem. Outra característica marcante nesse motor é que os fatos, que são unificados com as regras, são também objetos da própria linguagem.

Java possui diversas facilidades, como uma arquitetura bastante voltada para redes, o que a torna muito boa para o desenvolvimento de *web services*. Entretanto, por ser uma linguagem interpretada, o desempenho das aplicações Java fica abaixo do desempenho de aplicações que rodam com código nativo. Este fato torna essa linguagem imprópria para o desenvolvimento de certos tipos de sistemas, que exijam menor tempo de resposta, como jogos eletrônicos, para os quais, nos últimos tempos, o uso de agentes inteligentes tem sido cada vez mais exigido como forma de aumentar o *gameplay*.

Jogos podem ser considerados como aplicações de tempo real, pois um baixo tempo de resposta é requisito essencial para a imersão do jogador no ambiente de jogo. Por ser bastante eficiente, além suportar orientação a objetos, o que propicia o desenvolvimento de sistemas de grande escala, C++ acabou sendo adotada como padrão para a produção de aplicações deste tipo.

Infelizmente existem poucos motores, *open source*, para C++ disponíveis no mercado. Além do mais os motores de inferência disponíveis possuem alguns inconvenientes:

a) as regras de produção são interpretadas, o que torna a execução do motor mais lenta quando comparada àqueles que usam regras compiladas;

b) há necessidade de aprender uma nova linguagem, específica do motor em uso, para a definição das regras e instanciação dos fatos;

c) não há uma boa integração entre o motor e a linguagem hospedeira, por exemplo no tratamento unificado dos objetos.

Esses problemas dificultam o uso desses motores de inferência no desenvolvimento de jogos.

3.1 O Jeops como ponto de partida

Com base em tudo que vimos até agora, consideramos que seria bastante interessante levar as facilidades providas pelo Jeops a outras linguagens, como C++, que existe há mais de vinte anos e que atualmente é utilizada por centenas de milhares de programadores em vários domínios de programação [3], além de ser a linguagem padrão para o desenvolvimento de jogos.

C++ foi desenvolvida tendo C como linguagem base. Seus programas são compilados para código nativo, e por isso é adequada para a maioria das tarefas de programação de sistemas, inclusive aquelas em que o tempo de resposta é um fator crítico.

O objetivo deste trabalho foi de desenvolver uma ferramenta em C++, com as mesmas características do Jeops. Nos capítulos seguintes demonstraremos os resultados obtidos.

3.2 Adaptando o Jeops para C++

Nas próximas seções falaremos das principais adaptações do Jeops para o CEOPS++. Será apresentada a arquitetura do Jeops e como ficou o mesmo componente no CEOPS++.

3.2.1 Fatos

3.2.1.1 JEOPS

Uma característica muito importante do Jeops é que os fatos, que são unificados com as regras, são também objetos da própria linguagem Java. Essa integração entre o motor e linguagem hospedeira no tratamento unificado dos objetos não é vista, por exemplo, no Soar ou no CLIPS (seções 2.5.1 e 2.5.2).

```
public class Pessoa {
      private String nome;
      private Pessoa pai;
      private Pessoa mae;

      Pessoa(String nome) {
            this.nome = nome;
      }

      public Strint getNome() { return this.nome; }
      public void setPai(Pessao pai) { this.pai = pai; }
      public Pessoa getPai() { return this.pai; }
      public void setMae(Pessao mae) { this.mae = mae; }
      public Pessoa getMae() { return this.mae; }
}
```

Tabela 13 Exemplo de fato no JEOP (Classe em Java).

3.2.1.2 CEOPS++

No CEOPS++ os fatos são representados por ponteiros para objetos ou estruturas. A Tabela 14 e mostra um exemplo de classe em C++.

```
class Pessoa {
private:
       string nome;
       Pessoa* pai;
       Pessoa* mae;
public:
       Pessoa(const string& nome) {
         this->nome = nome;
         this->pai = NULL;
         this->mae = NULL;
       }

       string& GetNome() { return nome; }

       Pessoa* GetPai() { return pai; }
       Pessoa* GetMae() { return mae; }
       void SetPai(Pessoa* pai) { this->pai = pai; }
       void SetMae(Pessoa* mae) { this->mae = mae; }
};
```

Tabela 14 Exemplo de fato no CEOPS++ (Classe em C++) .

3.2.2 Base de Objetos

3.2.2.1 Jeops

A base de objetos guarda referências para todos os objetos armazenados na memória de trabalho. Quando um objeto é inserido na memória, antes dele ser enviado à rede Rete, ele deverá ser inserido na base de objetos. Fica sob responsabilidade dela verificar se objeto já existe na memória de trabalho, impedindo que existam duas referências para o mesmo.

A base de Objetos também trata relação de herança entre classes, de forma que a chamado ao método `objects(String)` retorna todas as instância diretas e indiretas da classe cujo nome é passado como parâmetro.

3.2.2.2 CEOPS++

Da mesma forma que o Jeops, a base de objetos no CEOPS++ armazena o endereço de memória de todos os objetos presentes na memória de trabalho. Esses endereços são armazenados em ponteiros. Também fica sob responsabilidade desta verificar se já não existe um ponteiro para um objeto que está sendo inserido na base, impedindo que existam duas referências para a mesma instância de classe.

No CEOPS++ a base de objeto armazena informações sobre herança entre classes, mas essas informações de herança são obtidas por meio da função `TypeCheckFunction` (seção 4.2). Os detalhes de implementação serão mostrados na seção 4.1.

3.2.3 Regras

Antes de falarmos das regras Jeops e CEOPS++, vale a pena explicar a diferença no processo de pré-compilação e interpretação de Regras.

3.2.3.1 Regras Pré-compiladas e Interpretadas

Como já foi dito, as regras de produção são entidades de representação do conhecimento descritas em alto nível. Cada um dos motores que descrevemos na seção 2.5 possui uma sintaxe própria para definição de suas regras, sendo esta definida pelos projetistas desses sistemas com base em algum critério arbitrário. Então é necessário o mapeamento dessas regras para a linguagem

hospedeira do sistema de produção. Basicamente existem duas maneiras de fazer esse mapeamento: via interpretação das regras ou via pré-compilação das regras.

No processo de interpretação das regras, o sistema de produção cria em memória uma estrutura representando a regra lida. Essa estrutura poderá em seguida ser utilizada pelo motor de inferência na unificação, resolução de conflitos e disparo das regras. Esse mecanismo de interpretação torna possível que a base de regras seja modificada dinamicamente durante a execução do sistema. Outra grande vantagem da interpretação é que não há a necessidade do uso do compilador da linguagem.

Já na pré-compilação, as regras são transformadas em código fonte da linguagem hospedeira do sistema de produção. Em seguida esse código deve ser compilado para poder ser utilizado pelo motor de inferência. Esta estratégia facilita a implementação do sistema, pois algumas tarefas, como a avaliação de expressões aritméticas ou como a definição e utilização de variáveis, passam a ser responsabilidade do compilador. Outra vantagem da pré-compilação das regras é que o próprio compilador pode realizar algumas otimizações no código, além do código gerado estar em um nível bem mais baixo, o que pode significar alguma melhoria no desempenho do motor. A desvantagem dessa abordagem é que não é possível modificar a base de regras em tempo de execução.

3.2.3.2 Jeops

Baseado no princípio de uniformidade de integração, o Jeops utiliza sintaxe muito próxima de Java para definir suas produções. Na definição de uma regra há uma separação explícita entre os campos para as declarações de variáveis, condições e ações. O conjunto de produções é organizado em uma base de regas. Além de produções, uma base de regras pode conter métodos e declarações de variáveis. Também vale lembrar que o Jeops utiliza o sistema de pré-compilação. A sintaxe dessa base de regras pode ser encontrada no Anexo A - .

```
rule GoUp {
  declarations
    Fibonacci f,f1,f2;
  conditions
    f.getN() > 1;
    f.getSon1() == f1;
    f.getSon2() == f2;
    f1.getValue() != -1;
    f2.getValue() != -1;
  actions
    int v = f1.getValue() +
            f2.getValue());
    f.setValue(v);
    modified(f);
}
```

Tabela 15 Regra JEOPS

3.2.3.3 CEOPS++

Para o CEOPS++ decidimos manter essa uniformidade. Assim tentamos manter a sintaxe de definição de regras o mais próximo possível de C++. E da mesma forma que o Jeops, aqui também há uma separação entre os campos para as declarações de variáveis, condições e ações, além do conjunto de regras também estar organizado dentro de uma base de regras. Mas para essa primeira versão do sistema resolvemos fazer algumas simplificações. Na versão atual, por exemplo, não é possível fazer declarações de métodos e variáveis dentro do corpo de uma base de regras. No CEOPS++ as regras também são pré-compiladas. Um exemplo de regra é mostrado a seguir.

```
rule GoUp {
    declarations
        Fibonacci* f_0;
        Fibonacci* f_1;
        Fibonacci* f_2;
    conditions
        f_0->getN() > 1;
        f_0->getValue() == -1;
        f_0->getSon1() == f_1;
        f_0->getSon2() == f_2;
        f_1->getValue() != -1;
        f_2->getValue() != -1;
    actions
       int v = f_1->getValue() + f_2->getValue();
       f_0->setValue(v);
       Retract(f_1);
       Retract(f_2);
       delete f_1;
       delete f_2;
       Modified(f_0);
}
```

Tabela 16 Regra CEOPS++

Na parte de declarações da regra só é possível declarar ponteiros para classes ou estruturas. Essa decisão foi tomada para facilitar a implementação do interpretador. Para entender como essa implementação pôde ser simplificada, veja os exemplos de declarações da Tabela 17. Considere que exista uma classe "Pessoa".

```
1) Declaração 1:
      Pessoa p1;

2) Declaração 2:
      Pessoa& p2;

3) Declaração 3:
      Pessoa* p3;
```

Tabela 17 Tipos de declarações

A Tabela 18 representa um possível código que o interpretador poderia gerar para definição e modificação dos valores dessas variáveis em uma regra. Considere o código abaixo como parte do corpo de uma definição de classe. Vale salientar que o código que é realmente gerado é um pouco mais complexo que o mostrado, pois leva em consideração a possibilidade de serem definidas várias variáveis.

1) Código gerado a partir da declaração 1:

```
Pessoa p1;
void SetObject(Pessoa p) {
      p1 = p;
}
```

2) Código gerado a partir da declaração 2:

```
Pessoa& p2;
void SetObject(Pessoa& p) {
      p2 = p;
}
```

3) Código gerado a partir da declaração 3:

```
Pessoa* p3;
void SetObject(Pessoa* p) {
      p3 = p;
}
```

Tabela 18 Possível código gerado pelas declarações da Tabela 17

O código gerado a partir da "declaração 1" possui uma declaração na forma "Pessoa p". Esse tipo de declaração gera automaticamente uma instância da classe pessoa, e caso a declaração dessa classe não possua um construtor default (construtor sem parâmetros) o código conteria um erro de compilação. Além disso, na assinatura do método "`SetObject(Pessoa p)`" a variável "`p`" recebe uma cópia do objeto que está sendo passado como parâmetro, e da mesma forma a igualdade "`p1=p`", presente no corpo do método, faz uma cópia de "`p`" em "`p1`". Isso implica que qualquer alteração em "`p1`", por parte das ações da regra por exemplo, não seria refletida no objeto original, pois "`p1`" é apenas uma cópia da cópia daquele objeto.

Em C++ uma declaração do tipo "`Pessoa& p`", presente no código gerado para "declaração 2", significa referência para classe "Pessoa". Uma referência nada mais é do que um nome alternativo para o objeto. Para assegurar que uma referência sempre está vinculada a algum objeto, o compilador obriga que esta seja inicializada. Assim podemos concluir que esse código está errado, pois a declaração feita dessa forma geraria um erro de compilação (não há inicialização da variável "`p2`"). Além do mais, referências são mais comumente utilizadas para especificar argumentos e valores devolvidos por funções, em particular para operadores sobrecarregados.

Por último uma declaração do tipo "`Pessoa* p`", presente no código gerado para "declaração 2", significa ponteiro para objeto da classe "Pessoa". Isto quer dizer que uma variável desse tipo pode armazenar o endereço de um objeto do tipo "Pessoa". O método "`SetObject(Pessoa* p)`", definido dessa maneira, significa que "`p`" recebe uma cópia do endereço do objeto passado como parâmetro, e esse endereço é em seguida, no corpo do mesmo

método, copiado para "p3". Logo qualquer alteração realizada através de "p3" será refletida no objeto original.

É claro que o código dos exemplos poderiam ter sido gerados de forma que todos fossem válidos e apresentassem os resultados esperados. Mas o que queríamos demonstrar aqui é que não poderia haver uniformidade na geração desses códigos, caso todas as construções fossem permitidas. Isso levaria a um aumento na complexidade do interpretador sem que houvesse ganho significativo para quem escreve as regras.

Para concluir, a parte de condição da regra pode ser utilizada qualquer expressão booleana válida em C++, e na parte de ações pode ser utilizado qualquer comando válido em C++, além dos comandos *Retract*, *Insert* e *Modify*.

3.2.4 Base de Regras

3.2.4.1 Jeops

A base de regras é gerada a partir da pré-compilação do arquivo de regras, sendo a classe gerada uma subclasse de `jeops.AbstractRuleBase`. Ela representa o conjunto das produções definidas no arquivo de regras, ficando responsável por responder à base de conhecimento ou à rede Rete se as condições de uma regra são verdadeiras, quantas e quais regras foram definidas no arquivo pelo usuário, quais as declarações de cada regra, etc. Ela também é responsável por disparar uma regra quando necessário.

De maneira geral podemos dizer que a base de regras no Jeops possui todas as informações necessárias à base de conhecimento para que esta possa fazer a unificação entre objetos e regras.

3.2.4.2 CEOPS++

No CEOPS++ as funcionalidade presentes em `jeops.AbstractRuleBase` foram divididas em duas classes, `ceops::AbstractRuleBase` e `ceops::AbstractRule`. A classe `ceops::AbstractRuleBase` funciona como um repositório de regras (armazena um conjunto de classes `ceops::AbstractRule`), possuindo basicamente os métodos necessários para acessar o conjunto de regras armazenadas.

A classe `ceops::AbstractRule` representa uma produção definida no arquivo de

regras. Ela contém todos os métodos necessários para obtenção de informações sobre a regra declarada e manipulação da mesma. Exemplos de informações que podem ser obtidos por essa classe são: os números de declarações, condições e ações da regra; nome da regra; classes das declarações; se as condições são verdadeiras para um determinado conjunto de fatos; etc.

As classes concretas de `ceops::AbstractRuleBase` e `ceops::AbstractRule` são geradas a partir da pré-compilação do arquivo de regras. O processo de pré-compilação gera um arquivo chamado `<Nome da base de regras>_rules.h`, e vários arquivos `<Nome da base de regras>_Rule_<nome da regra>.h`, sendo este último um arquivo para cada regra que foi definida.

A decisão da separação do código em vários arquivos foi tomada por considerarmos essa divisão como sendo algo mais natural (com cada regra representada por uma classe e a base de regras como um conjunto dessas classes) e também para facilitar a leitura do código gerado. A princípio o usuário que estivesse definindo as regras não teria necessidade de olhar o código gerado. Mas como veremos mais adiante, o processo de pré-compilação não faz uma análise contextual das condições e das ações definidas em uma regra, o que pode ocasionar erro de compilação. Sendo o código de fácil leitura, a identificação e correção de erros ficam também bastante facilitadas.

Para exemplificar o que queremos dizer, considere as classes Pessoa e Objetivo mostradas na Tabela 19 e a regra para encontrar ancestrais mostrada na Tabela 20.

```
class Pessoa {
private:
       string nome;
       Pessoa* pai;
       Pessoa* mae;
public:
       Pessoa(const string& nome) {
         this->nome = nome;
         this->pai = NULL;
         this->mae = NULL;

       }

       string& GetNome() { return nome; }

       Pessoa& GetPai() { return *pai; }
       Pessoa& GetMae() { return *mae; }
       void SetPai(Pessoa* pai) { this->pai = pai; }
       void SetMae(Pessoa* mae) { this->mae = mae; }
};

class Objetivo {
private:
       Pessoa* objetivo;
public:
       Objetivo(Pessoa* objetivo) {
         this->objetivo = objetivo;
       }

       Pessoa* GetObjetivo() { return objetivo; }
};
```

Tabela 19 Definição das classes Pessoa e Objetivo

```
rule Ancestral {
         declarations
             Objetivo* o_1;  //d1
             Pessoa* p_1;    //d2
         conditions
             p_1 == o_1->GetObjetivo();  //c1
             p_1->GetPai() != NULL;      //c2
             p_1->GetMae() != NULL;      //c3
         actions
             Retract(o_1);       //a1
             Insert(new Objetivo(p_1->GetPai()));  //a2
             Insert(new Objetivo(p_1->GetMae()));  //a3
}
```

Tabela 20 Regra para encontrar ancestrais

O código gerado para as ações “a2” e “a3” conteria um erro de compilação, pois os métodos `GetPai` e `GetMae` retornam referências de “Pessoa”, enquanto o construtor de “Objetivo” recebe

um ponteiro para “Pessoa”. Esse erro só é percebido quando o usuário tenta compilar o arquivo de saída do pré-compilador. Essa é a razão do cuidado com a legibilidade do código gerado.

Detalhes sobre a implementação de `AbstractRule` e `AbstractRuleBase` podem ser encontrados nas seções 4.3 e 4.4.

3.2.5 RETE

Rete é um eficiente algoritmo de casamento de padrões utilizado na implementação de sistemas produção. Ele foi desenvolvido em 1979 pelo Dr. Charles L. Forgy e tem sido usado desde então como base de muitos sistemas especialistas. Inclusive todos os motores citados na seção 2.5 fazem uso desse algoritmo [14].

Em uma implementação simples do motor de inferência, durante a unificação, cada regra poderia ser unificada com todos os fatos na memória de trabalho, o que poderia se tornar algo bastante ineficiente caso tivéssemos um grande número de regras e fatos.

O algoritmo Rete provê uma base para uma implementação mais eficiente de sistemas de produção. Sistemas baseados nesse algoritmo constroem uma rede de nós que correspondem aos padrões presentes nas condições de das regras. Quando novos fatos são adicionados ou modificados, eles são propagados pela rede e cada nó armazena informações sobre as avaliações das condições de uma regra de forma que essas avaliações não precisem ser refeitas.

3.2.5.1 Jeops

Jeops implementa uma variação do algoritmo original, possuindo apenas a funcionalidade de memorização das avaliações já realizadas e deixando de lado a utilização da rede para eliminar avaliações redundantes. Essa adaptação torna possível que cada regra seja tratada individualmente, reduzindo assim o esforço na implementação do algoritmo [1].

Como as regras são tratadas de forma independentes, o que acontece na verdade é que para cada Regra, o Jeops cria uma mini-rede por onde cada novo fato inserido na memória de trabalho deverá ser propagado.

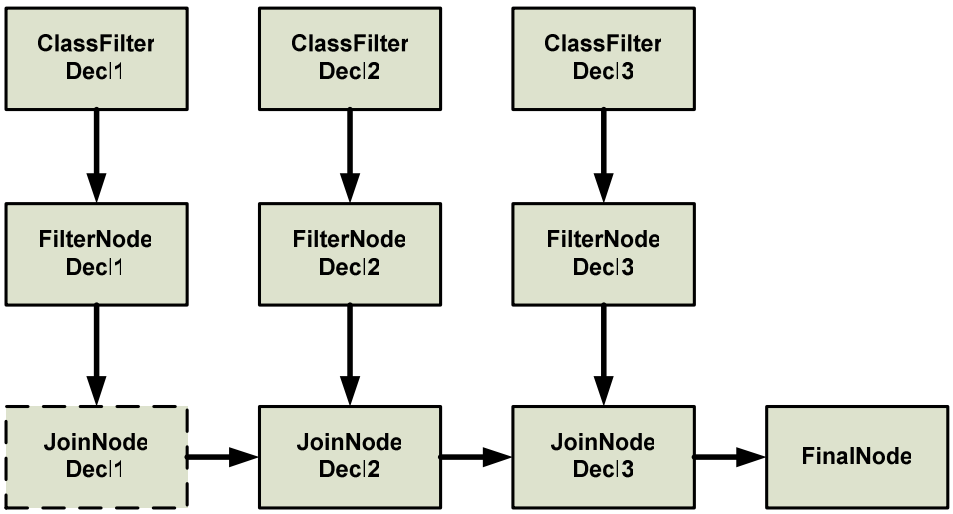


Figura 7 Implementação do Rete no JEOPS

A seguir a descrição de cada um dos elementos presentes na figura acima.

- **ClassFilter**

Esse nó é criado para cada declaração presente na regra. Ele filtra os elementos que são do mesmo tipo da classe da variável declarada. Se um elemento for instância da classe representada por esse nó, então ele será propagado para seu sucessor, caso contrário ele não será propagado.

- **FilterNode**

Ele agrupa todas as condições que dependem de uma única declaração, sendo criado, então, um nó desse tipo para cada declaração. Se todas as condições forem satisfeitas para um determinado objeto, então este será propagado para o seu sucessor.

- **JoinNode**

Os nós de junção também estão associados com cada declaração. Ele fica responsável por avaliar as condições que dependem da condição que ele representa e de todas as condições anteriores. Quando um grupo de objetos passa por esse nó quer dizer que todas as condições que eles dependem já foram satisfeitas.

- **FinalNode**

Toda vez um conjunto de objetos atinge esse nó, é criado um novo elemento para ser adicionado ao conjunto de conflitos.

3.2.5.2 CEOPS++

No CEOPS++ a rede é praticamente idêntica a rede descrita para o Jeops, com duas pequenas alterações. Para melhor exemplificar o funcionamento das redes e as diferenças entre elas vamos fazer uso de um exemplo.

```
rule atacar {
   declarations
      AgenteJogador agente;     //d1
      Unidade unidade1;         //d2
      Unidade unidade2;         //d3
   Conditions
      agente.ehDono(unidade1);            //c1
      agente.ehInimigo(unidade2);         //c2
      unidade2.getPosicao().distancia( unidade1.getPosicao() ) <= unidade1.getAlcanceAtaque();   //c3
      unidade1.getAtaque() > unidade2.getAtaque();          //c4
      unidade1.tirosRestantes > 0;            //c5
   actions
      unidade1.atacar(unidade2);       //a1
}
```

Tabela 21 Regra Atacar

A regra da Tabela 21 está defina segunda a sintaxe do Jeops. A Figura 8 mostra a rede gerada para a regra representada acima. Repare que como não há qualquer condição que dependa apenas da primeira e terceira condições, os objetos do tipo `AgenteJogador` e `Unidade`, este último desde que esteja relacionado à terceira condição, serão sempre propagados para os nós de junção. O primeiro nó de junção aparece pontilhado na figura porque ele não é construído realmente. Os objetos do primeiro nó de filtro são propagados diretamente para o nó de junção da segunda declaração, onde, junto com todos os objetos provenientes do segundo nó de filtro, serão testados contra a condição c1.

Os nós de junção ficam responsáveis por armazenar todas as unificações parciais realizadas, de forma que os outros tipos nós não possuem qualquer memória.

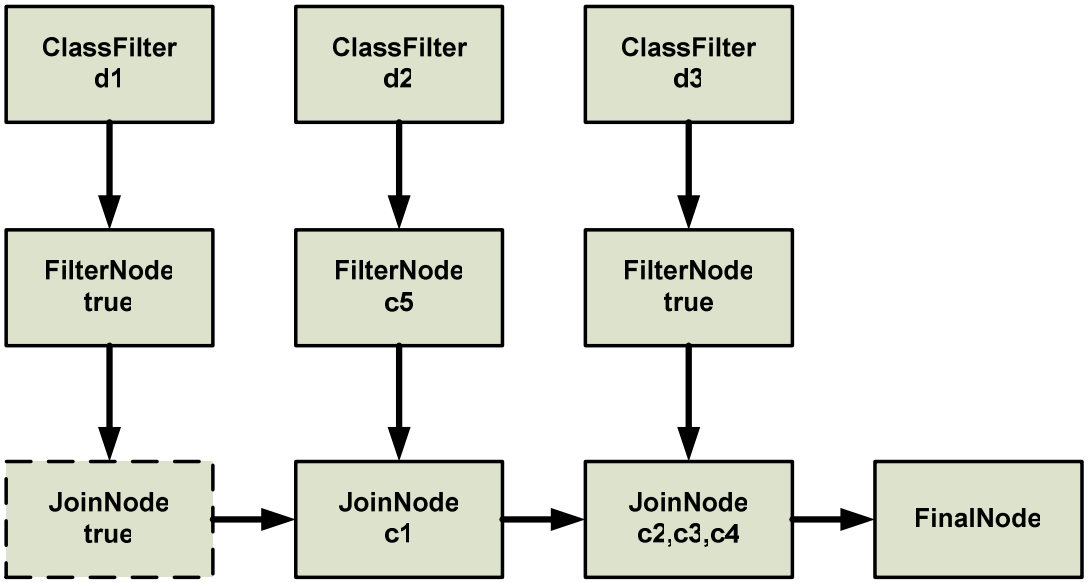


Figura 8 Rede Rete do JEOPS gerada para a regra da Tabela 21

Agora vejamos o mesmo exemplo aplicado ao CEOPS++. Devemos considerar que existe uma regra equivalente a anterior adaptada para este motor de inferência.

Como podemos notar na Figura 9, a principal diferença está relacionada aos filtros de classes. No CEOPS++ existe um filtro de classe para cada tipo de classe e não para cada declaração como no Jeops. Essa modificação reduz significativamente o número de nós que são percorridos ao se adicionar um novo fato à rede.

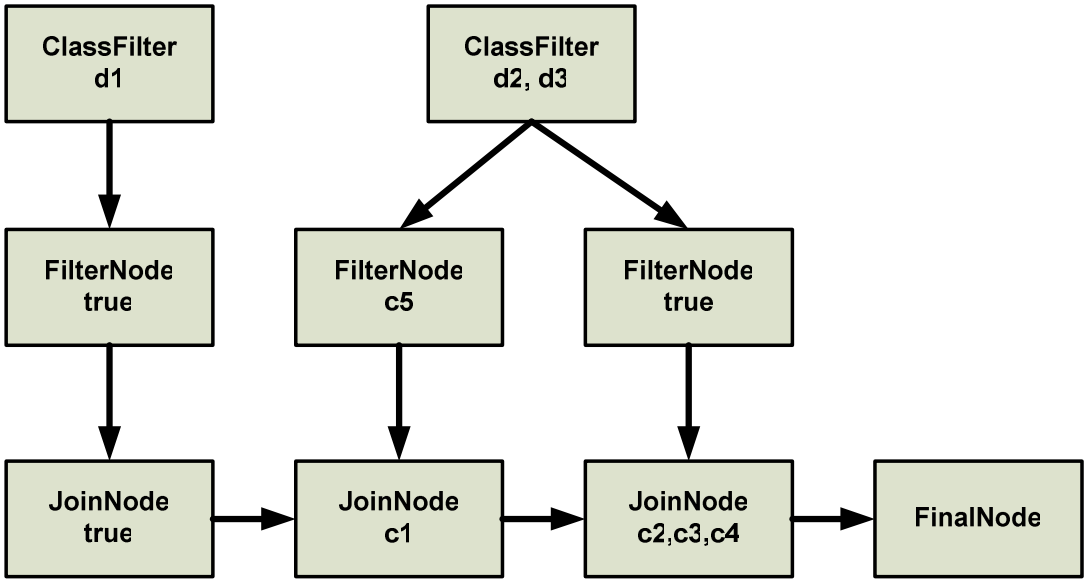


Figura 9 Rede Rete gerada para o CEOPS++

3.2.6 Base de Conhecimento

3.2.6.1 Jeops

A base de conhecimentos atua como fachada do sistema, possuindo os métodos necessários para o acesso do usuário ao motor de inferência. Toda base de conhecimento é subclasse de jeops.AbstractKnowledgeBase, sendo a classe concreta criada a pelo mecanismo de pré-compilação. Seus principais métodos são para manipulação dos objetos presentes na memória de trabalho (inserção, remoção, modificação e acesso). Seguem esses métodos:

- **void insert(Object)**

Insere um novo objeto na memória de trabalho.

- **void retract(Object)**

Remove um objeto da memória de trabalho. Também verifica se existe algum elemento do conjunto de conflitos que utilize esse objeto, se for o caso, esse elemento será removido do conjunto.

- **Vector objects(String)**

Retorna todos os objetos que são instâncias diretas e indiretas da classe passada como parâmetro.

- **void flush()**

Limpa todos os objetos da memória de trabalho e apaga todos os elementos do conjunto de conflitos.

- **void run()**

Coloca o motor de inferência para funcionar, disparando todas as regras presentes no conjunto de conflitos até que este esteja vazio.

3.2.6.2 CEOPS++

Assim como no Jeops, aqui a base de conhecimento também serve como fachada para acesso às funcionalidades do motor de inferência. Entretanto no CEOPS++ toda base de conhecimento é uma instância de `ceops::KnowledgeBase`, que é uma classe concreta. Os principais métodos dessa classe são os mesmo listados na seção anterior, com pequenas modificações em suas assinaturas devido a particularidades da linguagem hospedeira.

Uma mudança significativa é que a `ceops::KnowledgeBase` permite a execução das ações *Insert*, *Retract* e *Modify* sejam executadas por meio de eventos. Detalhes de implementação na seção 4.7.

4 Implementação do CEOPS++

Neste capítulo serão apresentados detalhes da implementação dos diversos itens listados no capítulo anterior.

4.1 Base de Objetos

A base de objetos é representada pela classe `ceops::ObjectBase`. Essa classe possui dois atributos, `objects` e `subclasses`. O primeiro é um mapeamento de um nome de classe em um conjunto de ponteiros do tipo `void` (`vector<void*>`). O segundo atributo é um mapeamento de um nome de classe em um conjunto contendo o nome de suas subclasses. O atributo subclasses foi criado com intuito de aperfeiçoar a busca de todos os subtipos de uma determinada classe. Na seção 4.2 demonstramos como obtemos essas subclasses.

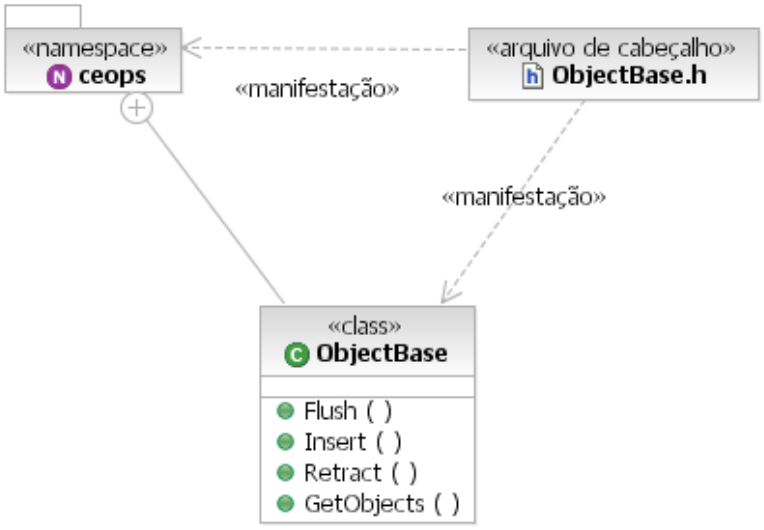


Figura 10 Classe ObjectBase

- **void Flush()**

O método `Flush()` remove todos os objetos da base de objetos. Esse método faz simplesmente um chamada ao método `clear()` dos mapeamentos.

- **bool Insert<T>(T* obj)**

Esse método insere um novo objeto na base de objetos. O valor do tipo bool retornado por ele

serve para indicar se o objeto foi realmente adicionado na base.

Primeiro é feito uma pesquisa para verificar se já não há uma referência do objeto na memória de trabalho. Se a referência ao objeto já existe nenhuma alteração é feita e o método retorna falso.

Caso seja a primeira vez que uma determinada classe esteja sendo inserida na memória de trabalho, o método faz uma chamada à função `TypeCheckFunction`, que retornará toda informação de herança da classe que está sendo armazenada.

- **bool Retract<T>(T* obj)**

Remove um objeto da memória de trabalho. O método verifica se há na memória de trabalho alguma referência ao objeto passado como parâmetro. Caso ele encontre, o objeto é removido e é retornado true.

Como os objetos estão agrupados por tipo, não é necessária uma busca por toda a memória de trabalho, apenas um subconjunto dos objetos é verificado. Esse detalhe melhora a eficiência na remoção de fatos.

- **void GetObjects<T>(vector<T*>& objs)**

Retorna todas as instâncias de uma determinada classe. Também são retornadas todas as subclasses da classe pesquisada.

Como as informações de herança já estão presentes no atributo subclasses, a implementação desse método ficou bastante facilitada.

- **GetSubclasses<T>(vector<string>& subclasses)**

Retorna todas as subclasses de uma determinada classe.

4.2 Checagem dos Subtipos (TypeCheckFuncion)

Em programação, reflexão é usada para investigar as formas dos objetos em tempo de execução, podendo até invocar métodos e acessar campos desses objetos. Esse mecanismo é utilizado para implementar códigos genéricos que possam trabalhar com objetos de tipos desconhecidos.

O Jeops faz o uso de reflexão para obter informações sobre a hierarquia de classes dos

objetos inseridos na base de objetos. Por exemplo, se existisse um `ClassFilter` que filtrasse objetos do tipo Automóvel, um objeto do tipo Carro (considerando que Carro é subclasse de Automóvel) seria propagado para seu sucessor caso ele fosse adicionado a esse nó. A informação de que Automóvel é uma generalização de carro é obtida em tempo de execução fazendo uso da API de reflexão de Java (`java.lang.reflect`).

Infelizmente C++ não suporta reflexão. Pelo menos não como gostaríamos. Existe suporte limitado a RTTI (*Runtime Type Information*) na linguagem. É possível obter o tipo de um objeto em tempo de compilação e em tempo de execução. Entretanto, o tipo de um objeto só pode ser obtido em tempo de execução se ele contém pelo menos uma função virtual. O exemplo abaixo ilustra esse fato.

```
class A {
public:
	void metodo() {
		cout << "Metodo da classe A" << endl;
	}
};

class B : public A {
	void metodo() {
		cout << "Metodo da classe B" << endl;
	}
};

int main() {
	A* a = new A();
	B* b = new B();

	A* a2 = b;

	cout << "Nome de a: " << typeid(*a).name() << endl;
	cout << "Nome de b: " << typeid(*b).name() << endl;
	cout << "Nome de a2: " << typeid(*a2).name() << endl;

	cin.get();
}
```

Tabela 22 Exemplo de RTTI em C++

Nesse exemplo temos duas classes A e B, sendo que B é subclasse de A. Note que a variável "`a2`" é um ponteiro para um objeto do tipo A e recebe o endereço de um objeto do tipo B, mas ao executar o programa temos a seguinte saída.

Figura 11 Saída do programa da Tabela 22

Repare que na saída é impressa a informação de que "a2" aponta para um objeto do tipo A, informando assim o tipo declarado para esse ponteiro e não o tipo do objeto que ele realmente aponta.

Agora modificamos o código da classe A para que ela contenha um método virtual.

```
class A {
public:
	virtual void metodo() {
		cout << "Metodo da classe A" << endl;
	}
};
```

Tabela 23 Classe A com um método virtual

Quando rodamos o programa temos a seguinte saída:

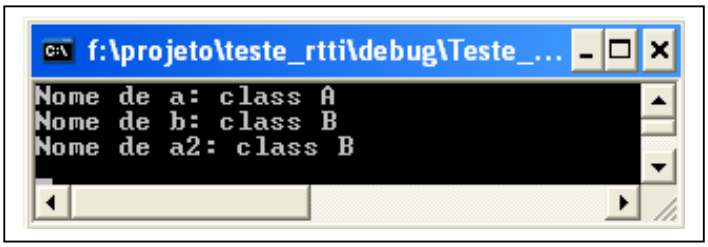


Figura 12 Saída do programa quando A possui um método virtual

Agora repare que na saída está impresso corretamente que "a2" aponta para um objeto do tipo B.

Comparar tipos e obter nomes de tipos é tudo o que é possível fazer usando RTTI em C++. E mesmo para essas ações existem algumas restrições, como pôde ser visto. O problema é que, assim como o Jeops, o CEOPS++ também precisa de informações sobre hierarquia de classes para que os filtros de classe da rede Rete possam funcionar corretamente.

Existem diversas API's de código aberto que de alguma forma tentam prover algum mecanismo de reflexão a C++ [15] [16]. Algumas abordagens dessas API's obrigam que o usuário

redefina as classes que ele pretende obter informações em tempo de execução, para que ela seja por exemplo subclasses de uma classe específica. Infelizmente esse tipo de abordagem restringe o uso de código de terceiros.

Outra abordagem, essa mais interessante, obtém informações a partir dos *headers* das classes (arquivos com extensão ".h"), compilando um conjunto de classes que contêm todas as informações necessárias para o mecanismo de reflexão. Entretanto modificações nos arquivos ".h" , ou se houver necessidade de acrescentar novas classes, exigem que o usuário recompile o código para que as novas informações seja adicionadas.

Para essa primeira versão do CEOPS++ resolvemos utilizar uma solução própria, que a princípio corresponde a todas as nossas expectativas. Essa solução faz uso de uma função chamada `TypeCheckFuncion`, sua assinatura é mostrada logo abaixo.

```
template<typename T>
inline void util::TypeCheckFunction(T* type, vector<string>& subclasses);
```

Tabela 24 Assinatura da função `TypeCheckFunction`

Essa função recebe como parâmetros um ponteiro para objeto e uma referência para um vetor, e tem como finalidade preencher o vetor com o nome das subclasses do primeiro parâmetro. Na implementação padrão da função, atualmente disponível no motor de inferência, ela retorna apenas o nome da própria classe passada como parâmetro. Para utilizar informações sobre hierarquia de classes o usuário deve redefinir essa função e utilizá-la em conjunto com a classe `TypeCheck`.

```
template<typename T>
class TypeCheck {
public:
      TypeCheck() {}

      bool check(T* t) {
            return true;
      }

      bool check(T& t) {
            return true;
      }

      bool check(void* v) {
            return false;
      }
};
```

Tabela 25 Classe TypeCheck

A finalidade da classe TypeCheck é obter informações sobre o tipo de um objeto estaticamente. Seu uso é bem simples: ela é parametrizada com um tipo específico, e ao fazermos uma chamada ao método `check` ele retornará verdadeiro ou falso dependendo do tipo do objeto passado como parâmetro. Esse comportamento é baseado no fato do compilador escolher qual o método correto a ser chamado. A tabela abaixo mostra um exemplo de uso dessa classe.

```
class A {
public:
	void metodo() {
		cout << "Metodo da classe A" << endl;
	}
};

class B : public A {
	void metodo() {
		cout << "Metodo da classe B" << endl;
	}
};

class C : public B {
	void metodo() {
		cout << "Metodo da classe C" << endl;
	}
};
int main() {
	A* a = new A();
	B* b = new B();
	C* c = new C();

	TypeCheck<A> typeA;
	TypeCheck<B> typeB;
	TypeCheck<C> typeC;

	cout << "a eh do tipo A?: " << (typeA.check(a)?"Sim":"Nao") << endl;
	cout << "a eh do tipo B?: " << (typeB.check(a)?"Sim":"Nao") << endl;
	cout << "a eh do tipo C?: " << (typeC.check(a)?"Sim":"Nao") << endl;
	cout << "b eh do tipo A?: " << (typeA.check(b)?"Sim":"Nao") << endl;
	cout << "b eh do tipo B?: " << (typeB.check(b)?"Sim":"Nao") << endl;
	cout << "b eh do tipo C?: " << (typeC.check(b)?"Sim":"Nao") << endl;
	cout << "c eh do tipo A?: " << (typeA.check(c)?"Sim":"Nao") << endl;
	cout << "c eh do tipo B?: " << (typeB.check(c)?"Sim":"Nao") << endl;
	cout << "c eh do tipo C?: " << (typeC.check(c)?"Sim":"Nao") << endl;

	cin.get();
}
```

Tabela 26 Exemplo de uso da classe TypeCheck

A figura abaixo mostra a saída do programa acima.

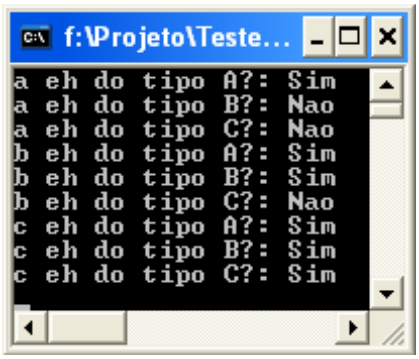


Figura 13 Saída do programa da Tabela 26

Ao redefinir a função `TypeCheckFunction`, o usuário deve ter em mente todos os tipos de classes presentes no arquivo de regras. Por exemplo, supondo que as classes A, B e C estivessem presentes nas declarações das regras, o usuário poderia implementar essa função da seguinte forma:

```
template<typename T>
inline void TypeCheckFunction(T* type, vector<string>& subclasses) {
	TypeCheck<A> typeA;
	TypeCheck<B> typeB;
	TypeCheck<C> typeC;

	if(typeA.check(type)) { //verifica se eh subclasse de A
		subclasses.push_back(typeid(A).name());
	}

	if(typeB.check(type)) { //verifica se eh subclasse de B
		subclasses.push_back(typeid(B).name());
	}

	if(typeC.check(type)) { //verifica se eh subclasse de C
		subclasses.push_back(typeid(C).name());
	}
}
```

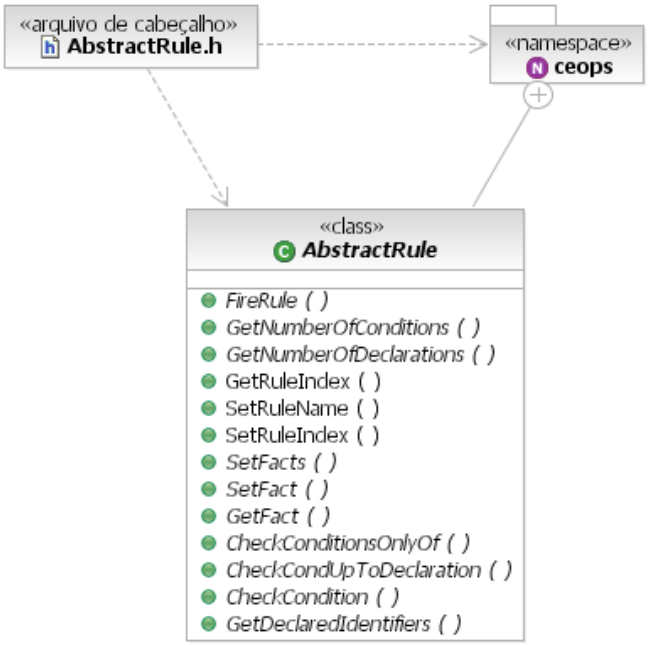
Tabela 27 Exemplo de implementação da função TypeCheck

Repare que não são obtidas todas as subclasses de uma determinada classe, mas apenas aquelas cujos tipos estão presentes nas declarações das regras. Mas essa informação já é suficiente para o motor de inferência.

4.3 AbstractRule

Essa classe representa uma produção definida pelo usuário no arquivo de regras. Ela foi definida como uma classe abstrata e sua interface contém o conjunto de métodos necessários para

obtenção de informações e manipulação de uma regra. As classes concretas de AbstractRule são geradas automaticamente pelo pré-compilador.



A seguir uma lista com os principais métodos dessa classe:

- **void SetRuleIndex(unsigned int ruleIndex)**

Modifica o índice dessa regra.

- **virtual void SetFacts(vector<void*>& facts)**

Mapeia o conjunto de objetos contidos no vetor passado como parâmetro no conjunto de declarações da Regra. Esse método é muito utilizado pelos nós da rede Rete para checar a validade das condições da regra.

- **virtual void SetFact(unsigned int dclIndex, void* fact)**

Mapeia o objeto passado como parâmetro na declaração cujo índice é igual a `dlcIndex`.

- **virtual void* GetFact(unsigned int dclIndex)**

Retorna um ponteiro para o objeto que está atualmente mapeado em uma das declarações.

- **virtual bool CheckConditionsOnlyOf(unsigned int declIndex)**

Verifica se as condições que dependem apenas de uma dada declaração são verdadeiras. É

utilizado pela classe `FilterReteNode`.

- **virtual bool CheckCondUpToDeclaration(unsigned int declIndex)**

Verifica se as condições que dependem da declaração de índice dado e das declarações de índice menor ou igual ao índice dado são verdadeiras. É utilizado pela classe `JoinReteNode`.

- **virtual bool CheckCondition(unsigned int condIndex)**

Verifica se uma condição é verdadeira, tendo em vista os objetos que atualmente estão mapeados nas declarações.

- **virtual void FireRule()**

Executa o conjunto de ações definidas pelo usuário para esta regra. Este método é utilizado pela Base de conhecimento para disparar a regra.

- **virtual const string& GetDeclaredClassName(unsigned int index)**

Retorna o nome do tipo de uma determinada declaração.

- **virtual void GetDeclaredIdentifiers(vector<string>& identifiers)**

Retorna o nome dos identificadores declarados pelo usuário no arquivo de regras.

- **virtual int GetNumberOfConditions()**

Retorna o número de condições que a regra possui.

- **virtual int GetNumberOfDeclarations()**

Retorna o número de declarações que a regra possui.

- **unsigned int GetRuleIndex()**

Retorna o índice da regra. Esse índice indica a ordem que o usuário definiu a regra no arquivo de regras.

- **const string& GetRuleName()**

Retorna o nome da regra.

4.4 Base de Regras

A base de regras é representada pela classe `ceops::AbstractRuleBase`, e armazena

uma coleção de objetos do tipo `ceops::AbstractRule`. Além dos métodos necessários para acessar e manipular esse conjunto de regras, essa classe também define todos os métodos presentes em `AbstractRule` com um parâmetro a mais: o índice da regra que se deseja obter a informação.

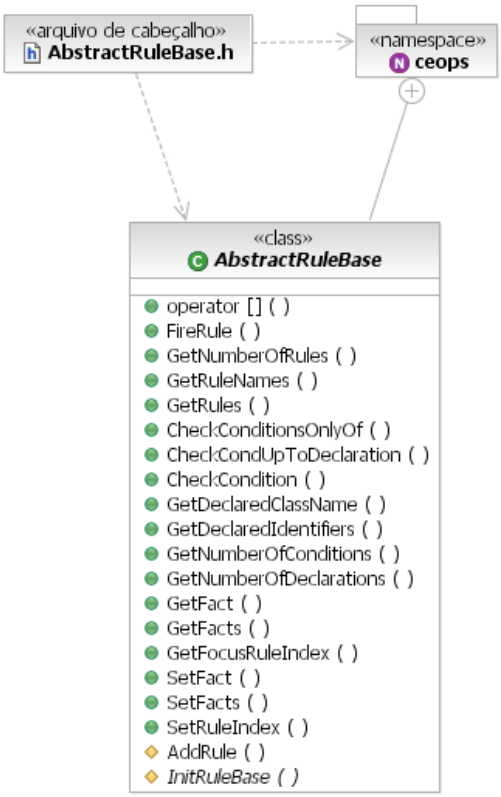


Figura 14 Classe AbstractRuleBase

Seguem os principais membros dessa classe:

- **AbstractRule& operator[](unsigned int ruleIndex)**

Retorna uma referência para regra cujo índice é passado como parâmetro.

- **void FireRule(unsigned int ruleIndex)**

Dispara a regra com índice igual ao índice passado como parâmetro.

- **int GetNumberOfRules()**

Retorna o número de regras armazenadas na base de regras.

- **void GetRuleNames(vector<string>& ruleNames)**

Retorna um vetor contendo o nome de todas as regras armazenadas na base de regras.

- **void GetRules(vector<AbstractRule*>& rules)**

Retorna um vetor contendo todas as regras armazenadas na base de regras.

- **void AddRule(AbstractRule* rule)**

Adiciona uma nova regra na base de regras. Esse método é protegido, de forma que só pode ser chamado a partir de uma subclasse.

- **virtual void InitRuleBase()**

Função membro utilizada para inicializar a base de regras. Esse método é chamado no construtor da classe e sua sintaxe obriga que ele seja implementado por alguma subclasse dessa classe (método abstrato). Ele é gerado automaticamente pelo pré-compilador quanto constrói a classe concreta de `AbstractRuleBase`, sendo utilizado para popular a base de regras com várias instâncias das regras definidas pelo usuário.

4.5 Rete

A classe Rete possui conjuntos de objetos `ClassFilter`, `FilterNode`, `JoinNode` e `FinalNode`, representando todas as mini-redes geradas para cada regra definida pelo usuário. Assim, essa classe funciona como uma interface entre essas mini-redes e a base de conhecimento. Essa classe também fica responsável por construir todos os nós da rede, o que é feito pela chamada do método `AddRule`. A chamada desse método faz com sejam tomadas as seguintes ações:

```
void Rete::AddRule(AbstractRule* rule) {
    para cara declaração d de rule
        Se existir um nó filtro de classe c correspondente à classe de d
            utilizar o nó c
        caso contrário
            criar o nó c

        Criar um nó filtro f que avalia as condições que dependem apenas de d
        Associar f como sucessor de c
        Criar nó de junção j referente a declaração d
        Associar j como sucessor de f
        Se já existe algum nó de junção, associar j como sucessor do último
        nó de junção criado antes dele

    Por fim criar um nó final e adicionar esse nó como sucessor do último nó
de junção criado
```

Tabela 28 Passos utilizados para criar a rede Rete

Cada `ClassFilterNode` está ligado a um conjunto de objetos `FilterNode`. Quando um novo fato chega ao filtro de classe, pela chamada ao método `NewObject`, ele é imediatamente propagado para os nós de filtro ligados a ele. Isso porque a classe `Rete` se encarrega de enviar as classes corretas para os filtros de classes corretos. Nos `FilterNodes` são testadas as condições que dependem apenas da declaração associada ao filtro, e caso sejam verdadeiras, o fato é propagado para o nó de junção correspondente. Nos nós de junção são testadas as condições que dependem da declaração associada aos nós e de quaisquer das declarações de índice menor. Os nós desse tipo também armazenam todos os objetos que chegam até eles. Caso as condições sejam todas validadas, os objetos são propagados até o `FinalReteNode` que cria um `ConflitSetElement` e informa a base de conhecimentos para que esse elemento seja colocado no conjunto de conflitos.

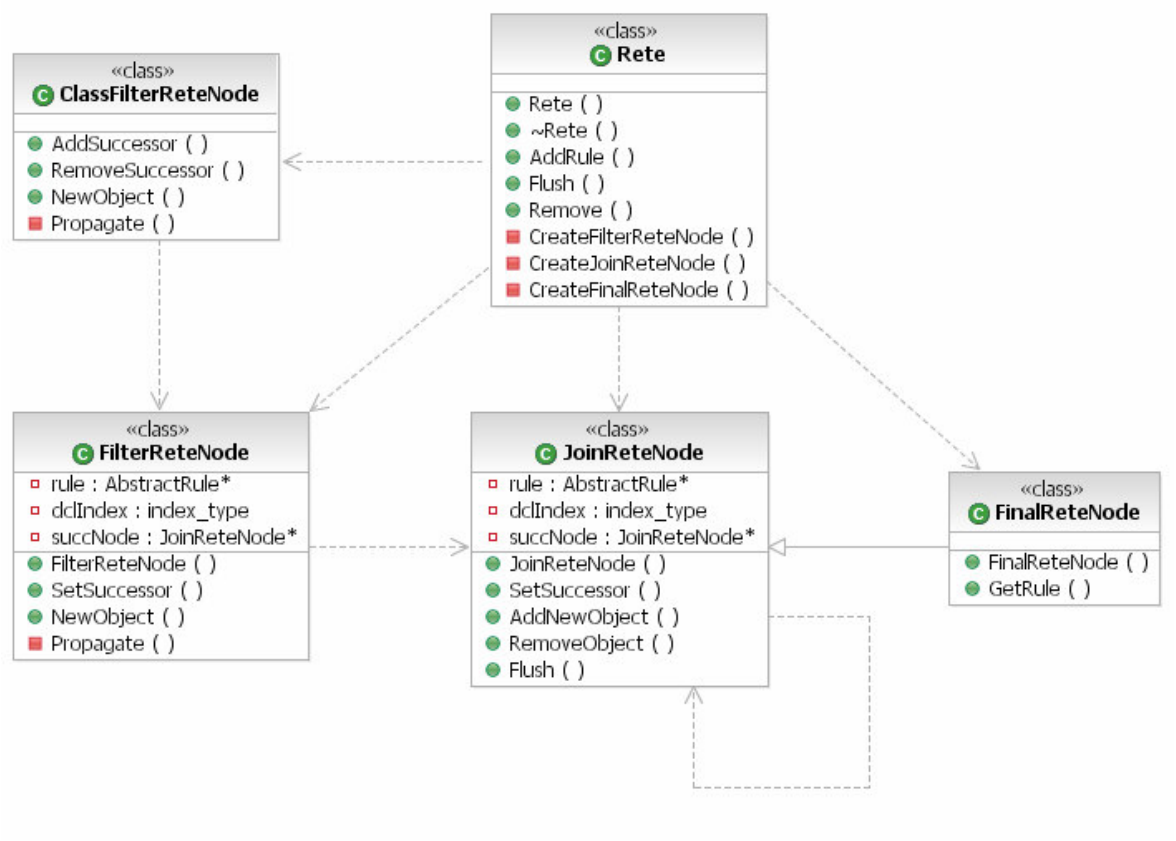


Figura 15 Classes do Namespace ceops

4.6 Conjunto de Conflitos

Esse conjunto armazena os pares constituídos de todas as regras ativas juntamente com os objetos que as ativaram. A classe `conflict::ConflictSet` define uma interface (classe com todos os métodos abstratos) que pode ser utilizada pelo usuário para implementação da sua própria política de resolução de conflitos.

Os métodos que devem ser definidos pelo usuário são os seguintes:

- **virtual void Flush()**

Apaga todos os elementos do conjunto de conflitos.

- **virtual void InsertElement(ConflictSetElement* element)**

Adiciona um novo elemento ao conjunto de conflitos.

- **virtual bool IsEmpty()**

Verifica se o conjunto de conflitos está vazio.

- **virtual ConflictSetElement* NextElement()**

Retorna o próximo elemento do conjunto de conflito, que possui a regra a ser disparada.

- **void RemoveElementsWith(void* obj)**

Remove do conjunto de conflitos todos os elementos que contenham o objeto passado como parâmetro.

Essa classe deve armazenar um conjunto de objetos do tipo `conflic::ConflitSetElement`. A classe `ConflitSetElement` possui os seguintes membros:

- **AbstractRule* GetRule()**

Retorna a regra representada pelo elemento.

- **vector<Fact>& GetFacts()**

Retorna o conjunto de fatos que ativam a regra representada pelo elemento.

- **unsigned long GetTime()**

Retorna um número representando o instante em que o elemento foi criado.

4.7 Base de Conhecimentos

A base de conhecimentos é a interface entre o usuário e as funcionalidades do motor de inferência. Ela é representada pela classe `ceops::KnowledgeBase`, e para criar uma instância dessa classe o usuário deve fornecer como parâmetros do construtor objetos do tipo `ConflitSet` e `AbstractRuleBase`. O conjunto de regras encapsulado pela base de regras é passado para um atributo do tipo `rete::Rete` que se encarrega de construir a rede Rete.

A finalidade principal dessa classe é permitir a inclusão, modificação, pesquisa e remoção dos objetos da memória de trabalho.

- **bool Insert<T>(T* fact)**

  Insere um novo objeto na memória de trabalho. Primeiro o objeto é inserido na base de objetos, e se esta retornar true, a base de conhecimentos pede então informações sobre as subclasses dele, passando essa informação mais o objeto inserido para a Rete.

- **void Modify<T>(T* fact)**

  Informa ao motor de inferência que um objeto foi modificado. Para implementar esse método, o motor faz simplesmente chamadas aos métodos `Retract` e `Insert`, passando como parâmetro o objeto modificado.

- **void Retract<T>(T* fact)**

  Remove um objeto da memória de trabalho. O objeto deve ser removido da base de objeto e da rede Rete. Se houver algum `ConflitSetElement` que possua uma referência para esse objeto, esse elemento também deve ser removido do conjunto de conflitos.

4.8 Resultados obtidos

Durante o desenvolvimento do sistema vários testes foram realizados a fim de validar o CEOPS++. Nessa seção apresentaremos o resultado obtido para o problema de encontrar o enésimo número da série de Fibonacci. Essa seqüência é definida pela fórmula mostrada na Figura 16.

$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

Figura 16 Fórmula recursiva da séria de Fibonacci

A solução do problema aqui apresentada foi adaptada de [1].

Para resolução do problema foi utilizada a classe da Figura 17 que representa o enésimo número na seqüência. Essa classe possui quatro atributos: 1)n – a posição do elemento na seqüência; 2) value – o valor do elemento na seqüência; 3) os seus dois antecessores.

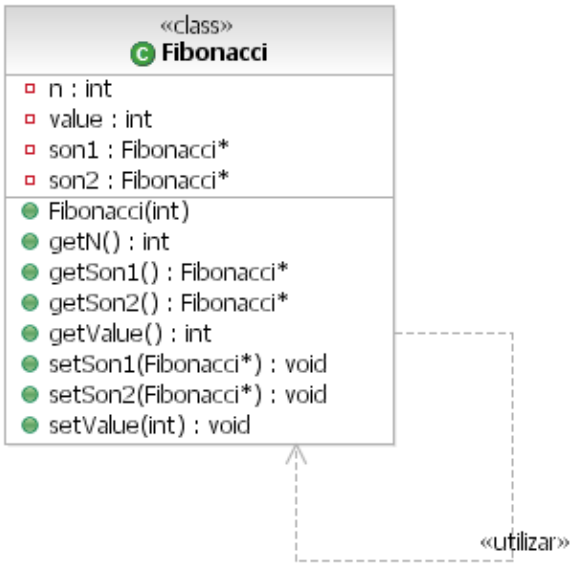


Figura 17 Classe Fibonacci

As regras utilizadas para o cálculo do número de Fibonacci são apresentadas nas tabelas abaixo.

```
rule BaseCase {
    declarations
        Fibonacci* f0;
    conditions
        f0->getN() <= 1;
        f0->getValue() == -1;
    actions
        f0->setValue(f0->getN());
        Modified(f0);
}
```

Tabela 29 Regra BaseCase

```
rule GoDown {
    declarations
        Fibonacci* f0;
    conditions
        f0->getN() > 1;
        f0->getValue() == -1;
        f0->getSon1() == NULL;
    actions
        int n = f0->getN();
        Fibonacci* s1 = new Fibonacci(n - 1);
        Fibonacci* s2 = new Fibonacci(n - 2);
        f0->setSon1(s1);
        f0->setSon2(s2);
        Insert(s1);
        Insert(s2);
        Modified(f0);
}
```

Tabela 30 Regra GoDown

```
rule GoUp {
    declarations
        Fibonacci* f_0;
        Fibonacci* f_1;
        Fibonacci* f_2;
    conditions
        f_0->getN() > 1;
        f_0->getValue() == -1;
        f_0->getSon1() == f_1;
        f_0->getSon2() == f_2;
        f_1->getValue() != -1;
        f_2->getValue() != -1;
    actions
       int v = f_1->getValue() + f_2->getValue();
       f_0->setValue(v);
       Retract(f_1);
       Retract(f_2);
       delete f_1;
       delete f_2;
       Modified(f_0);
}
```

Tabela 31 Regra GoUp

Para utiliza o sistema basta criar uma base de regras e adicioná-la a uma instância da base de conhecimentos. Então é só inserir um novo objeto da classe Fibonacci na memória de trabalho e rodar o motor, conforme mostrado na Tabela 32.

```
int main() {

      AbstractRuleBase* ruleBase = new Fibonacci_Rules();
      KnowledgeBase base;
      base.SetRuleBase(ruleBase);
      Fibonacci f(20);
      base.Insert(&f);
      base.Run();

      delete ruleBase;

      return 0;
}
```

Tabela 32 Utilização da base de regras Fibonacci

Esse código foi executado várias vezes e os resultados estão compilados na tabela abaixo. O programa foi testado em um AMD Athlon 2.4+ com 1GB de Memória RAM.

	CEOPS++	**Jeops**
Fibonacci(20):	6765	6765
Número de disparos:	32836	32836
Tempo de criação da KB	Menos de 1 milisegundos	Em torno de 15 milisegundos
Tempo de Execução do programa	Média de 565 milisegundos	Média em torno de 2min e 30 segundos

Tabela 33 Resultados do programa Fibonacci

5 Conclusão

Conforme os jogos de computadores ficavam cada vez mais complexos, seus consumidores demandavam agentes controlados por computador cada vez mais sofisticados. Uma das maneiras mais utilizadas pelos desenvolvedores para prover raciocínio aos agentes em um jogo é fazendo uso de algum tipo de mecanismo de inferência.

Entretanto as soluções de código aberto disponíveis no mercado possuem algumas características que dificultam seu uso nesse tipo de aplicação. O Jeops, desenvolvido no CIn-UFPE, se mostrou um ótima ferramenta, pois ele possui um excelente nível de integração entre regras, fatos e a linguagem de programação, nesse caso, a linguagem Java.

Mas jogos são aplicações de tempo real, e o tempo de resposta é um fator importante para imersão do jogador no ambiente de jogo. Por isso linguagens interpretadas, como Java, acabam se tornando inadequadas para esse tipo de aplicação.

Com esse trabalho tentamos construir um ferramenta, em C++, que reunisse as principais vantagens apresentadas pelo Jeops. C++ foi escolhida por ser a linguagem padrão utilizada pelos desenvolvedores de jogos. Tentamos então reunir todas as vantagens daquele motor de inferência, como por exemplo: definindo uma a sintaxe para definição de regras muito próxima da linguagem hospedeira; permitindo que objetos da linguagem fossem utilizados como fatos; etc.

Com isso conseguimos fazer um sistema que é fácil de utilizar, além de possuir um bom desempenho (o uso de Rete garantiu boa performance ao sistema). No entanto, essa é a primeira versão do CEOPS++, e portanto há ainda muito que se fazer para melhor a aplicação.

O *parser* do sistema de pré-compilação de regras ainda não exibe muita informação a respeito dos erros de sintaxe presentes no arquivo de regras, sendo este um dos principais pontos a serem melhorados. A construção da classe `TypeCheckFuncion` também poderia passar a ser responsabilidade do pré-compilador e não do usuário, como acontece atualmente.

Outra questão importante é com relação às políticas de resolução de conflitos. A ferramenta é flexível o suficiente para que o usuário defina a política que ele desejar, mas seria interessante que houvesse um conjunto maior de implementações prontas.

Há também a necessidade de garantir a portabilidade do código, que atualmente tem seu funcionamento garantido apenas para o compilador da Microsoft. É preciso garantir que o código possa ser compilado pelo menos para gcc e turboC++, que são outros compiladores bastante

utilizados.

Mesmo precisando de algumas melhorias acreditamos que o CEOPS++ está apto a ser utilizado em vários projetos. Com a divulgação e disponibilização do resultado desse trabalho, esperamos que ele possa ser utilizado por um número cada vez maior de pessoas, que darão o *feedback* necessário para melhorarmos cada vez mais o sistema.

6 Referências Bibliográficas

[1] Figueira, Carlos. *JEOPS – Integração entre objetos e Regras de produção em Java*. Dissertação de Mestrado, CIn – UFPE, 2000.

[2] Russel, Stuart; Norvig, Peter. *Inteligência Artificial*. Editora Campus: ISBN 8535211772, 2004.

[3] Stroustrup, Bjarne. *A linguagem de programação C++*. Bookman: ISBN 85-7307-699-2, 2000.

[4] Riley, Gary. *What are Expert Systems?***.** Dinponível em http://www.ghg.net/clips/WhatIsCLIPS.html. Acessado em 10 de outubro de 2006.

[5] Sommerville, Ian. *Engenharia de Software***.** 6ª edição. Editora: Addison Wesley, 2003

[6] Dijkstra, Edsger. *The Humble Programer***.** ACM 15 (1972).

[7] Booch, Grady. *Object-oriented Analysis and Design*. 2ª edição. Addison Wesley, 1998.

[8] Hamilton. *Object-Oriented Programming*. O'Reilly.

[9] Myers, Glenford. *Composite/Structured Design*. Van Nostrand Reinhold, 1978.

[10] *CLIPS Basic Programming Guide*. Disponível em http://www.ghg.net/clips/download/documentation/bpg.pdf. Acessado em 10 de setembro de 2006.

[11] *CLIPS Advanced Programming Guide*. Disponível em http://www.ghg.net/clips/download/documentation/apg.pdf. Acessado em 10 de setembro de 2006.

[12] *Soar 8 Manual*. Disponível em http://ai.eecs.umich.edu/soar/sitemaker/docs/manuals/Soar8Manual.pdf. Acessado em 10 de setembro de 2006.

[13] Hasting, Peter. *The Soar Coloring Book*. 1997.

[14] Doorenbos, Robert. *Production Matching for Large Learning Systems*. Tese de doutorado, Carnegie Mellon University, 1995.

[15] Vollmann, Detlef. *Metaclasses and Reflection in C++*. Disponível em http://vollmann.ch/en/pubs/meta/meta/meta.html. Acessado em 15 de setembro de 2006.

[16] *Reflex*. Disponível em http://seal-reflex.web.cern.ch/seal-reflex/. Acessado em 15 de setembro de 2006.

[17] PricewaterhouseCoopers. Disponível em http://www.pwc.com/uk/eng/main/home/index.html. Acessado em 01 de outubro de 2006.

Anexo A – BNF DAS REGRAS JEOPS

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

Anexo B – BNF DAS REGRAS CEOPS++

```
Base de Regra ::= ("#define" <ident>)*
                  ("#include" ("<"<ident>">" | "<ident>") )*
                  ("using namespace" <ident> ";")*
                  "ruleBase" <ident> "{" <Corpo da Base> "}"

Corpo da Base ::= ( <Regra> )*

Regra ::= "rule" <ident> "{" <Corpo da Regra> "}"

Corpo da Regra ::= <Declarações>
                     (<Declarações Locais>)?
                    <Condições>
                    <Ações>

Declarações ::= "declarations"
                  (<Ponteiro> <ident>";" )*

Condições ::= "conditions" (<expressão booleana>)*

Ações ::= "actions" (<Ação>)+

Ação ::= "Assert" "(" <expressão> ")"
       | "Retract" "(" <expressão> ")"
       | "Modify" "(" <expressão> ")"
       | <comando>

<Ponteiro> ::= <ident> ("::" <ident>)* "*"

<ident> ::= <Identificador válido em C++>

<expressão booleana> ::= <Expressão boolean válida em C++>

<expressão> ::= <Expressão válida em C++ e que retorna um ponteiro>
```

_____

Geber Lisboa Ramalho

_____

Pablo de Santana Barbosa