Universidade Federal de Pernambuco
Graduação em Ciência da Computação
Centro de Informática

Trabalho de Graduação

R#

Integração de objetos da linguagem C# com regras de produção

Aluno: **Victor Cavalcanti Rodrigues** vcr2@cin.ufpe.br
Orientador: **Geber L. Ramalho** glr@cin.ufpe.br

Recife, Junho de 2008.

“Any sufficiently advanced technology is indistinguishable from magic.”
Arthur C. Clarke

"A atmosfera estava tão úmida que os peixes poderiam entrar pelas portas e sair pelas janelas, navegando no ar dos aposentos."
Regra escrita por
Gabriel G. Marquez,
compilável em R#

Agradecimentos

À minha família, especialmente meus pais, que me deram o apoio necessário durante esta e outras jornadas da minha vida.

Aos amigos da Commit Solutions, assim como ao dissidente João Augusto e à de outro departamento Thaíza: se vocês não perguntassem tanto como andava meu TG, eu não teria me dedicado tanto pra deslanchar com esse trabalho aqui. Valeu!

Ao meu orientador Geber, e também ao meu provável avaliador André Santos, que me ajudou no início deste trabalho com seus conhecimentos em .NET. Também a Pablo, que me disponibilizou o material do seu trabalho e também sua atenção.

Resumo

Soluções tecnológicas freqüentemente precisam automatizar processos cuja linha de raciocínio utiliza conhecimento implícito. Este conhecimento pode ser representado na forma de regras, que em conjunto formam uma base de conhecimento capaz de inferir do mundo informações de relevância e tomar determinadas ações quando um conjunto de premissas é satisfeito.

O presente trabalho apresenta uma proposta de linguagem para desenvolver conhecimento baseados em regras como uma extensão simbiótica para C#, uma das linguagens mais relevantes dentro do paradigma orientado a objetos.

Palavras-chave: C#, .NET, regras, integração, orientação a objetos, EOOPS.

Sumário

1. Introdução 7

2. Contextualização 8

2.1 RBS: Sistemas baseados em regras 8

2.2 Integração entre regras e linguagens orientadas a objetos 9

2.2.1 Compatibilidade Mútua 9

2.2.2 Encapsulamento 10

2.2.3 Herança 11

2.3 Avaliação de trabalhos relacionados 12

2.3.1 C# 12

2.3.2 JEOPS 15

2.3.3 NéOPUS 19

2.3.4 R++ 20

2.4 Resultado da análise 24

3. R# 27

3.1 Linguagem R# 28

3.1.1 Rulebase 28

3.1.2 Ativar ou desativar regras e *rulebases* 31

3.1.3 Choose 33

3.1.4 Declaração 34

3.1.5 Outras considerações sobre o uso da linguagem 35

3.2 Mecanismo de inferência de regras 36

4. Implementação de R#: *Proof-of-Concept* 41

5. Conclusões 45

6. Referências 46

Assinaturas 47

Índice de Figuras

Imagem 1.....8
Imagem 2.....13
Imagem 3.....13
Imagem 4.....13
Imagem 5.....14
Imagem 6.....14
Imagem 7.....14
Imagem 8.....16
Imagem 9.....17
Imagem 10.....18
Imagem 11.....19
Imagem 12.....21
Imagem 13.....27
Imagem 14.....28
Imagem 15.....29
Imagem 16.....31
Imagem 17.....33
Imagem 18.....35
Imagem 19.....36
Imagem 20.....38
Imagem 21.....39
Imagem 22.....40
Imagem 23.....41
Imagem 24.....42
Imagem 25.....42
Imagem 26.....43
Imagem 27.....44

1. Introdução

Soluções tecnológicas freqüentemente precisam automatizar processos cuja linha de raciocínio utiliza conhecimento implícito. Enquanto a forma explícita de conhecimento pode encontrar expressividade eficaz em mecanismos formais amplamente disponíveis, aspectos do conhecimento tácito podem continuar implícitos quando há um uso puramente funcional dessas representações [1]. Este conhecimento encontra uma representação mais eficaz na forma de regras, que em conjunto são capazes de construir uma base de conhecimento que tenha o poder de inferir do mundo percebido pela aplicação informações de relevância e tomar determinadas ações quando um conjunto de premissas é satisfeito.

Existem abordagens de sistemas de conhecimento baseados em regras para praticamente todas as linguagens do mercado que seguem o paradigma de programação orientado a objetos, que desde os anos 90 é o paradigma dominante na indústria de software e o que mais cresce em uso [2]. A relativa abrangência de tais sistemas mostra que há interesse da comunidade tecnológica na área, no entanto nem sempre a combinação de regras e objetos satisfaz na tentativa de promover um sistema híbrido integrado.

Há uma dificuldade em suportar ambos os paradigmas de forma harmônica. Os dois são adequados para diferentes tipos de tarefas e uma integração satisfatória precisa resolver algumas questões conceituais a respeito desses conflitos de paradigma.

Faz parte da solução para integrar de maneira bem sucedida ambos os paradigmas encontrar a simbiose lingüística [3] entre as duas linguagens, isto é, permitir que os programas escritos em uma linguagem possam integrar-se a programas implementados na outra de forma transparente e automática. A maior justificativa para haver esforço na direção da simbiose lingüística está na necessidade do usuário especialista que definirá tais regras de vivenciar uma experiência de especificação sem se envolver com particularidades de um ambiente de codificação para desenvolvedores de software, apenas preocupado com a definição do domínio o qual ele reúne conhecimento implícito valioso.

O presente trabalho apresenta a proposta de criação de um novo sistema de produção de regras, e descreve entre os vários componentes da solução uma linguagem criada para especificar conhecimento baseados em regras como um mecanismo simbiótico com C#, uma das linguagens mais relevantes dentro do paradigma orientado a objetos.

2. Contextualização

Esta seção é destinada ao entendimento geral do problema, o que inclui um esclarecimento breve sobre termos e particularidades importantes do domínio da solução descrita neste trabalho, realizado nas seções 2.1 e 2.2, assim como inclui também a avaliação de trabalhos relacionados, um dos principais critérios utilizados na concepção da solução, descrita na seção 2.3.

2.1 RBS: Sistemas baseados em regras

Sistemas baseados em regras – também chamados de Sistemas de produção ou RBS, *Rule Based Systems* – são aplicações de inteligência artificial desenvolvidos a partir de elementos que, juntos, representam um conjunto de conhecimentos heurísticos em relação a um domínio. Esses elementos são chamados de regras de produção. Cada regra de produção descreve, dada uma condição do ambiente avaliado, como o sistema deverá se comportar. Por permitir um modo diferenciado de codificação de conhecimento, tornou-se um método bastante utilizado para a definição de sistemas especialistas [4] e agentes inteligentes [5].

Uma regra de produção é composta por uma condição e um conjunto de ações a serem realizadas caso a condição seja satisfeita. À condição também se denomina antecedente, parte *if* ou *left-hand-side*, enquanto o conjunto de ações é freqüentemente chamado de conseqüente, parte *then* ou *right-hand-side*. A Imagem 1 ilustra o conceito de uma regra.

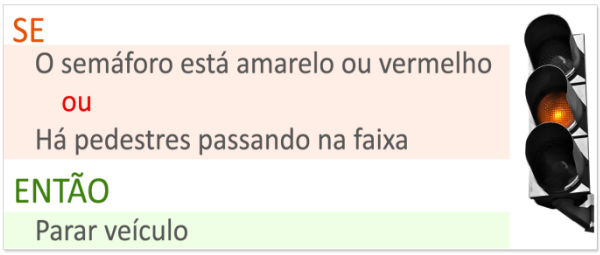

Imagem 1

Memória de trabalho é um conceito bastante necessário na compreensão e desenvolvimento (freqüentemente também no uso) de um sistema de produção. Ela constitui um espaço alocado pelo sistema que contém os fatos, ou seja, o que é considerado verdade sobre o ambiente. Os fatos serão avaliados em relação as condições descritas nas regras de produção. Eles podem ser afetados pelas ações

previstas nas regras cujas condições sejam satisfeitas, ou seja, um fato pode ser removido, modificado ou adicionado.

Um sistema de produção, em linhas gerais de funcionamento, contém regras e uma memória de trabalho. Durante sua execução, confronta as regras e os fatos: as regras que possuem condições verdadeiras em relação aos fatos apresentados são selecionadas. Se houver apenas uma regra selecionada, suas ações são realizadas. Se houver mais de uma, como seus distintos conjuntos de ações podem configurar um paradoxo, há o procedimento de seleção da regra prioritária, uma resolução para este tipo de conflito. O ciclo de execução de um sistema de produção se resume a estes passos, que podem ser denominados de **Unificação** – o confrontamento dos fatos com as condições das regras – **Resolução de Conflitos** e **Disparo** da regra selecionada. Enquanto houver regra selecionada na etapa de unificação, o ciclo continua.

2.2 Integração entre regras e linguagens orientadas a objetos

Entre os paradigmas de regras e de objetos, não deve-se estabelecer somente um cenário de escolha. Ambos os paradigmas são adequados para diferentes tipos de tarefas em programação e, em razão desta complementaridade, a integração entre os paradigmas pode ser vista como uma necessidade evolutiva. Programadores orientados a objeto precisam algumas vezes de formas mais abertas de controle para representação de conhecimento, por vezes também necessitam de mecanismos não-determinísticos. Programadores de bases de regras necessitam realizar representações mais sofisticadas dos fatos do mundo sobre os quais as regras atuam. Implementações estranhas em um paradigma podem encontrar no outro uma forma mais apropriada, como regras difíceis de manter ou inadequadas que podem ser substituídas por um método ou por um novo módulo de classes.

Algumas dimensões qualitativas desta integração, no entanto, devem ser consideradas, de modo a garantir que, do ponto de vista conceitual dos paradigmas a serem integrados, não houve conflitos ou concessões que prejudiquem o andamento das atividades de programação pelo uso do resultado obtido. O restante desta seção trata de avaliar algumas destas dimensões.

2.2.1 Compatibilidade Mútua

Avaliar o grau de compatibilidade mútua entre as linguagens é crucial, pois uma das partes pode perder sua força natural quando utilizada através dos mecanismos de integração. Conseguir esta compatibilidade entre regras e objetos não se mostra um esforço trivial de ser realizado sem concessões. Um bom exemplo desta dificuldade

está no fato de que as linguagens que descrevem regras historicamente se basearam na possibilidade de ter acesso à estrutura dos fatos contidos nos elementos da memória de trabalho. De acordo com a filosofia orientada a objetos, esta atitude é vista como totalmente contrária a seus princípios, sua tendência é de esconder a estrutura de dados no interior dos objetos.

Outro ponto corroborador com o aumento desta complexidade de conseguir um bom grau de compatibilidade mútua entre os paradigmas de regras e objetos é a integração disponibilizar todas as funcionalidades sintáticas da linguagem de objetos para a linguagem de regras, como variáveis locais e expressões diferenciadas, esta última sendo atualmente uma atitude bastante comum de linguagens de alto nível. A linguagem C#, que na sua versão 3.0 disponibiliza, além dos tradicionais constructos como “if”, “while”, “for”, outros mais elaborados como “foreach” e “using”, também dá suporte ao uso de *lambda-expressions* e é inerentemente integrada com uma linguagem avançada de consultas a dados chamada LINQ.

2.2.2 Encapsulamento

O tratamento da questão do encapsulamento é uma dimensão importante a ser considerada na integração das linguagens de regras e objetos. Apesar de ter origens relacionadas com o grau de compatibilidade mútua, como qualquer dimensão em última instância teria, vale a pena considerar separadamente a situação do encapsulamento em razão dos problemas gerados por ela serem de natureza mais específica, por este motivo demanda um esclarecimento mais dedicado.

Um dos pontos principais a respeito do encapsulamento trata do conflito entre as filosofias: objetos são projetados a princípio para esconder a estrutura de dados nele contida, assim como detalhes de implementação; regras são produzidas, à parte da integração com a filosofia orientada a objetos, voltadas para a manipulação explícita da estrutura de dados. Unir as duas perspectivas resulta em contradição neste ponto, e partindo do princípio que um programador de base de regras não deveria referenciar na sua parte do código detalhes de implementação e utilizar a estrutura de dados explicitamente, este conflito, neste trabalho, será tratado com uma concessão do paradigma de regras em relação a estes fatores. Embora haja a concessão para resolver o conflito, pelo fato desta ser tomada em favor de uma boa prática de engenharia de software ela pode ser vista não tão somente como uma concessão, mas como uma melhoria no paradigma de regras.

Outro fator relacionado a encapsulamento a ser ponderado na elaboração ou avaliação de um mecanismo de integração entre regras e objetos reside na consideração de que modificações são necessárias para permitir a uma aplicação o

acoplamento de uma base de regras. Um exemplo de como este fator é relevante: imagine que, para uma classe de objetos poder ter conhecimento sobre si representado em uma base de regras, precise herdar de uma certa superclasse pertencente ao motor de inferência da base de regras. Essa necessidade resulta num acoplamento muito forte, e pode impor restrições prejudiciais.

Quando os sistemas de inferência de regras ou a própria linguagem impõem restrições nos objetos da aplicação, de modo a interferir fortemente a ponto de violar a arquitetura de uma solução orientada a objetos, esta integração não ocorre da forma mais adequada. Esta afirmação pode sugerir que o sistema de regras adequado em relação a este fator não causa interferências na solução orientada a objetos, não precisa modificar a aplicação. Alguns pontos de acoplamento sempre serão necessários, a partir do código orientado a objetos, o que implica numa modificação da aplicação. Como não é possível associar de forma totalmente desacoplada os dois paradigmas, pode ser entendido que este fator de qualidade é impossível de ser alcançado. No entanto, trata-se de evitar interferência que causa violação arquitetural ou impede algumas atividades na programação orientada a objetos. É possível realizar uma associação de qualidade utilizando o mínimo de acoplamento necessário, escolhendo as alternativas menos invasivas e, quando possível, realizando automaticamente a maior parte das alterações requeridas.

2.2.3 Herança

A maneira como regras enxergam a herança entre classes é um aspecto muito importante da integração, devido a ser uma característica inerente da orientação a objetos que não deve ser desconsiderada. Durante a definição de uma linguagem de regras orientada a objetos, a questão da herança é encontrada em várias decisões:

a) Quando se decide de que maneira variáveis são tipadas nas regras;

b) Se há um quantificador universal (por exemplo, uma regra contendo: *"para qualquer objeto x, verifique essas condições"*);

c) Se ao definir uma regra para uma classe, ela será aplicada a objetos pertencentes a subclasses desta;

d) Se a integração implica na necessidade da classe especificada herdar de alguma classe do motor de inferência, este é um problema de violação de encapsulamento, como já citado, mas também de herança, pois impede que a prática seja utilizada nas classes que estão sob a regulamentação de uma base de regras.

2.3 Avaliação de trabalhos relacionados

Esta seção seleciona a análise de trabalhos relacionados, resultando na avaliação de C#, linguagem hospedeira do R#, e de algumas soluções para especificação e inferência de regras. Existe seguramente mais de uma dezena de projetos interessantes na área, no entanto foi dado foco de análise nos sistemas que mais apresentassem pontos críticos e de confluência para a criação deste trabalho.

2.3.1 C#

A seção anterior tratou das questões que envolvem a integração de um mecanismo de regras em uma linguagem orientada a objetos. Ao passo que muitas das considerações levantadas colocam o suporte à filosofia e às funcionalidades da linguagem hospedeira como principal fator de adequação na integração, esta seção foi criada com o objetivo de avaliar C# principalmente em relação a esses aspectos.

Em junho de 2000, a Microsoft anunciou tanto a plataforma .NET quanto C#, uma nova linguagem de programação. C# é uma linguagem orientada a objetos de propósito geral fortemente tipada, concebida tendo como principal objetivo *"combinar de forma ótima simplicidade, expressividade e performance"* [6].

A sintaxe de C# baseia-se bastante em C++ e também é influenciada por aspectos de outras linguagens, como Delphi e Java. A plataforma .NET tem como núcleo uma *Common Language Runtime*, máquina virtual para aplicativos escritos em uma linguagem intermediária denominada IL. O código escrito na linguagem C#, assim como em outras linguagens suportadas pela plataforma .NET é compilado para IL e pode ser executado em qualquer máquina que tenha a CLR da plataforma .NET.

As linguagens orientadas a objetos mais recentes provêem uma série de funcionalidades de linguagem de alto nível, que permitem o desenvolvedor executar atividades de maneira mais eficiente, sem ferir completamente o paradigma hospedeiro, no caso, o de orientação a objetos. Uma linguagem como C# dá acesso ao uso de *lambda-expressions, extension methods*, tipos anônimos, entre outras inovações dentro da linguagem, que não possuem necessariamente aderência ao paradigma orientado a objetos. As linguagens dominantes do mercado precisam, a cada nova versão, aumentar suas possibilidades de expressão e, para isso, podem realizar uma integração de itens que a princípio seriam considerados estranhos ao paradigma, mas há um esforço focado na adequação, e quando há uma incorporação destes itens de maneira que sua inserção não é causadora de não-conformidades na

especificação da linguagem e no seu uso e sua integração é bem-sucedida nas dimensões qualitativas.

Assim sendo, a seguir será listada uma seleção das funcionalidades mais inovadoras da linguagem C#, considerando como critério de seleção o diferencial das funcionalidades e o potencial de conflito de integração destas funcionalidades com o mecanismo de regras. O critério de seleção ajuda porventura na elicitação de uma impossibilidade completa de integração de alguma funcionalidade, como também pode apontar para uma combinação satisfatória com o mecanismo de regras.

Delegate: é um tipo que referencia métodos, encapsulando o acesso a métodos anônimos ou nomeados, mas são *type-safe*. A linguagem apóia-se em *delegates* para a manipulação de eventos.

```
public delegate int OperacaoAritmetica(int a, int b);
...
static void Main()
{
   OperacaoAritmetica op1 = Soma;
   OperacaoAritmetica op2 = delegate(int a, int b)
   {
      return a - b;
   };

   int a = 10, b = 5;
   int soma = op1(a,b);
   int diferenca = op2(a,b);
}
...
```

Imagem 2

Object Initializer: é possível inicializar um objeto substituindo o construtor tradicional por uma inicialização apenas das propriedades que o desenvolvedor quiser inicialmente. Ideal para evitar a construção de muitos construtores.

```
Customer mark = new Customer {Id=12, Kids=2, Money=1000};
```

Imagem 3

Type Inference: é possível inicializar uma variável sem declarar a priori o seu tipo, contudo em tempo de compilação seu tipo é descoberto.

```
var message = "Hello World!";
```

Imagem 4

Anonymous Type: permite criar tipos implicitamente em código. Uma vez que eles não tem um nome, são guardados através da palavra-chave *var*, o que pede ao compilador C# que use *Type Inference* para checar o tipo da variável.

```
var person = new {FirstName = "John", LastName = "Smith"};
```

Imagem 5

Extension Methods: disponível a partir da versão 3.0 da linguagem, permite "acrescentar" métodos a uma classe sem criar um tipo derivado ou recompilar esta classe, tampouco modificar a própria classe. Na sua declaração, um *extension method* é um método estático, que recebe no primeiro parâmetro a palavra-chave *this*.

```
public static int Soma(this int i, int a)
{
   return i + a;
}
...
int n = 4;
int m = n.Soma(n);
```

Imagem 6

Lambda Expressions: é uma função anônima que pode conter expressões e comandos, podendo ser utilizada para a criação de delegates, por exemplo. Toda *lambda expression* utiliza o operador =>, onde o lado à esquerda do operador representa a entrada e o lado à direita do operador representa o retorno da função.

```
delegate int del(int i);
del myDelegate = x => x * x;
int j = myDelegate(5); //j = 25
```

Imagem 7

Realizando um balanço geral das funcionalidades apresentadas, *extension methods* é uma funcionalidade que em nada interfere na integração com regras. A nova maneira de inicializar objetos sem o uso de um construtor explícito acrescenta uma complexidade ao compilador da integração apenas, mas não impede a integração. O uso de *delegates*, por conseguinte de *lambda expressions*, ao contrário do que possa parecer, pode facilitar o trabalho de especificação das regras e sua posterior compilação, de qualquer forma impactando na complexidade do compilador. O uso de tipos anônimos pode não ser considerado um grande problema se levarmos em conta que tipos anônimos não podem ser regulamentados por uma base de regras.

2.3.2 JEOPS

O JEOPS (Java Embedded Object Production System) é um motor de inferência para regras integrado à Java. Teve seu lançamento no ano de 1997, fazendo em 2000 parte da dissertação de mestrado do Centro de Informática da Universidade Federal de Pernambuco pelo então aluno de pós-graduação Carlos Santos da Figueira Filho, orientado pelo Prof. Dr. Geber Lisboa Ramalho [7]. No ano de 2006, o JEOPS recebeu uma versão equivalente de sua estrutura integrado à linguagem C++, que foi batizado de CEOPS++, elaborado por Pablo de Santana Barbosa em seu trabalho de conclusão de curso de graduação [8].

O presente trabalho compartilha com os projetos JEOPS e CEOPS++ não só o professor orientador e a instituição, mas também a proposta de complementaridade lingüística para o paradigma de orientação a objetos, no caso para a plataforma .NET, especificamente a linguagem C#. Java e C# guardam entre si muitas semelhanças, entre elas muitos pontos de confluência a respeito de suas filosofias.

A idéia norteadora da concepção do JEOPS foi de *"criar um sistema que primasse sobretudo pela facilidade de uso e pelo respeito às características do paradigma orientado a objetos, o que seria conseguido com a maior integração possível entre as regras e a linguagem hospedeira"* [7]. Performance também foi um aspecto considerado, dado o objetivo de utilizar o JEOPS em aplicações reais, sobretudo para a implementação de agentes, freqüentemente utilizados em aplicações que necessitam de um tempo curto de resposta.

Da idéia inicial surgiu, como princípio primeiro e mais importante, o que foi chamado por Carlos Figueira de "uniformidade da integração", isto é, a sintaxe das regras em JEOPS buscaria semelhança máxima à sintaxe da linguagem hospedeira, Java. Como o atingimento de uniformidade na integração implica na consideração de vários outros fatores (alguns destacados neste relatório, na seção **2.2**), houve também a preocupação no desenvolvimento da linguagem JEOPS de suportar todos os elementos de sintaxe da linguagem Java, como por exemplo, permitir chamadas aninhadas de métodos e acessar atributos públicos. Contudo, dado que o JEOPS trabalha baseado na versão 1.4 de Java, vários elementos da sintaxe atual certamente não são suportados, o que causa uma incompatibilidade entre JEOPS e as novas funcionalidades de linguagem providas pelas versões recentes de Java.

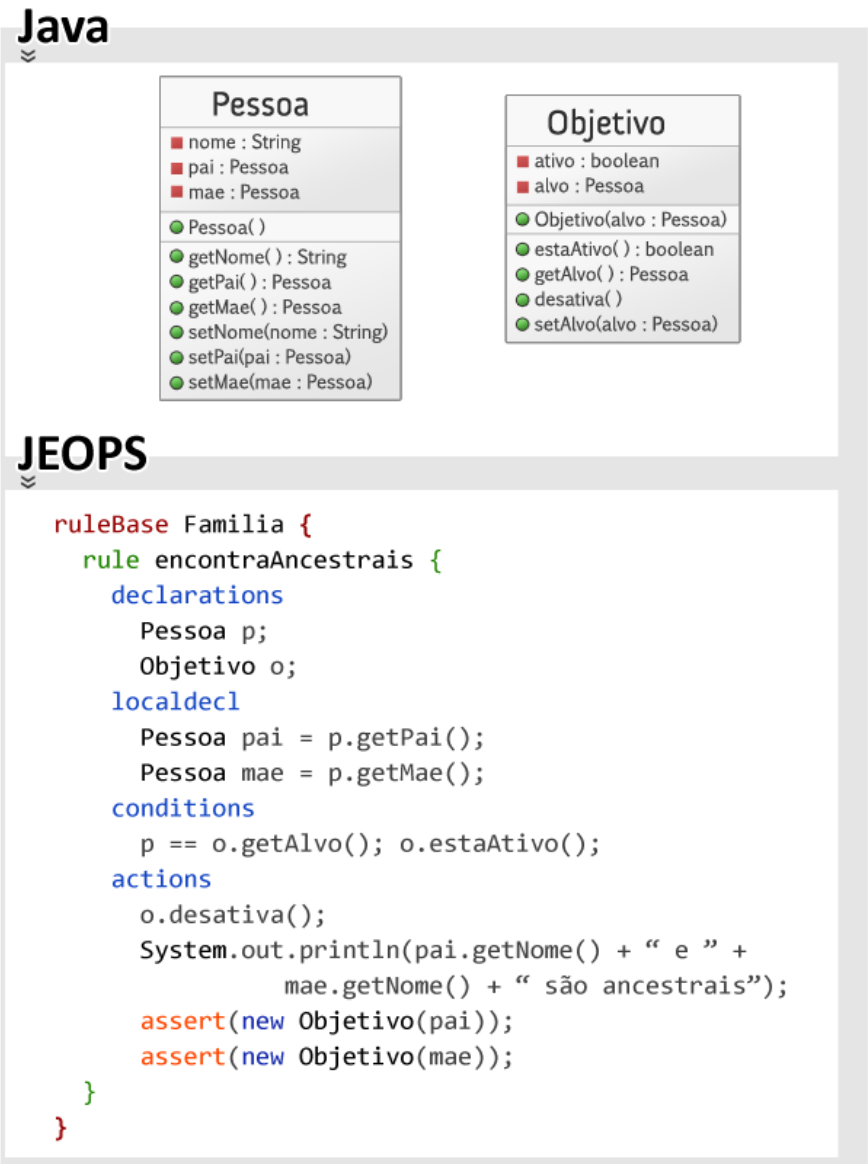

Imagem 8

As regras JEOPS são armazenadas em um arquivo que representa uma base de regras, este arquivo pode conter várias regras, assim como métodos e declarações de variáveis. O diagrama na Imagem 8, baseado num exemplo encontrado na dissertação de Carlos Figueira [7] ilustra uma base de regras ***Familia***, que contém uma regra apenas, chamada ***encontraAncestrais***, esta descrevendo conhecimento a respeito de objetos das classes em Java ***Pessoa*** e ***Objetivo***. As classes Java, na parte superior do diagrama, estão descritas de acordo com a modelagem UML, enquanto o código correspondente à especificação da regra foi colocado na íntegra, na parte inferior do diagrama. As cores atribuídas às palavras-chave da linguagem não necessariamente correspondem à realidade do JEOPS, foram assim atribuídas neste diagrama para possibilitar uma melhor leitura do código explicativo.

O processo de compilação de uma base de regras no JEOPS funciona de acordo com a Imagem 9. Inicialmente, o JEOPS realiza uma compilação dos arquivos que contém bases de regras (no exemplo, Familia.rules) e os transforma num arquivo Java (Familia.java). Em seguida, o compilador de Java gera os arquivos .class que serão interpretados pela máquina virtual Java (JVM). O arquivo Java compilado pelo JEOPS gera dois arquivos .class, pois dentro dele estavam definidas duas classes: uma representa a base de conhecimentos (Familia.class) e a outra representa a base interna de regras (Jeops_RuleBase_Familia.class). O acesso à base interna de regras não precisa ser efetuado pelo usuário do JEOPS. É automaticamente gerado pelo seu compilador e serve apenas para o próprio mecanismo de inferência da ferramenta.

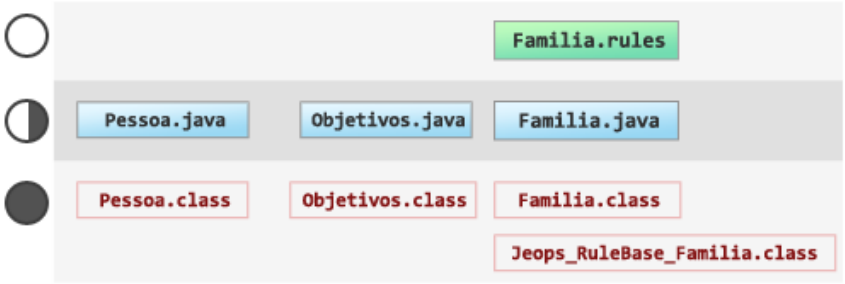

Imagem 9

A arquitetura do JEOPS (em alto nível) funciona de acordo com o diagrama exibido na Imagem 10. O agente comunica-se com o JEOPS através de uma fachada. Internamente, o JEOPS contém: uma **base de objetos**, onde são armazenadas as referências para os objetos que fazem parte da memória de trabalho; **uma base interna de regras**, que contém as regras e é responsável por responder à rede RETE se as condições de uma regra são verdadeiras; a rede **RETE** [11], que é o algoritmo de casamento de padrões que identificará, a partir de fatos modificados, possíveis conjuntos de objetos que, de acordo com algumas regras, devem ser enviados para o conjunto de conflitos e, por fim, **o conjunto de conflitos**, onde são armazenados os pares regra-objetos verificados na base de conhecimentos através do RETE e do acesso à base interna de regras. O conjunto de conflitos ficou separado do mecanismo do motor para permitir uma maior modularidade e extensibilidade em relação a como o usuário gostaria de resolver os conflitos. Além de um conjunto de estratégias de resolução de conflitos providas pelo JEOPS que podem ser escolhidas pelo usuário, também é fornecida uma interface para o usuário especificar uma estratégia diferenciada, caso seja necessário.

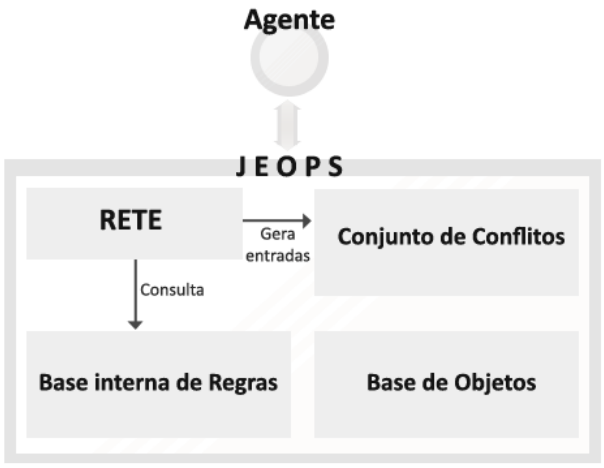

Imagem 10

O projeto do JEOPS considerou aspectos de simbiose lingüística, propôs estratégias que levam em consideração uma integração das tecnologias em um nível lingüístico satisfatório. Atingiu em grande parte seus objetivos nessa área, pois foi criada uma linguagem de regras complementar a Java que compartilha a mesma filosofia de codificação, o que permite que desenvolvedores com experiência em Java desenvolvam regras em JEOPS conservando a simbologia da linguagem hospedeira e mantendo vantagens do paradigma orientado a objetos. No entanto, o motor não detecta automaticamente quando um novo fato existe na base de conhecimento, deixando a cargo do desenvolvedor avisar quando há esta modificação. Isso resulta numa fraca transparência na integração entre as duas linguagens, o que é ruim do ponto de vista do acoplamento, podendo implicar inclusive em problemas de modularidade entre o código da aplicação e a manipulação do mecanismo de regras.

Como crítica principal ao trabalho, mesmo que o JEOPS tenha como principal objetivo sanar este problema, percebe-se a necessidade de resolver problemas de integração entre regras e objetos, pois além de não haver uma simbiose lingüística completa, existem outras abordagens de integração de regras (explicadas nos próximos trabalhos) que são mais aderentes ao paradigma orientado a objetos.

A disponibilização da atividade de especificação de regras deve ser realizada de tal forma que o período de aprendizado para o uso da linguagem seja curto. O esforço que uma linguagem demanda para ser aprendida e proveitosamente assimilada tem grande impacto no uso que os profissionais farão da linguagem, portanto, no resultado qualitativo da solução resultante. No tocante ao aprendizado, o JEOPS foi muito satisfatório, por ter sua sintaxe bastante similar a de Java.

2.3.3 NéOPUS

O NéOPUS é a proposta de integração entre regras e objetos para SmallTalk, realizado por François Pachet em 1992. Baseado no sistema OPUS [9], foi pioneiro na consideração de fatores qualitativos na integração de um mecanismo de regras a uma linguagem orientada a objetos, obtendo sucesso na busca de suavizar a transição entre a linguagem hospedeira e as regras. Sua existência provocou uma evolução na maneira de se enxergar a integração entre os paradigmas, e pode ser considerada como precursora direta do JEOPS, que foi bastante influenciado por esta solução [7].

Por prezar sobretudo por valores como transparência e manutenção total das funcionalidades da linguagem hospedeira Smalltalk, pode ser vista como um grande avanço e como uma estratégia que investiu adequadamente na compatibilidade mútua. A Imagem 11 contém a implementação em NéOPUS da mesma regra mostrada na seção destinada ao JEOPS. Assim como o JEOPS, a especificação de regras em NéOPUS faz uso de objetos respeitando o encapsulamento.

NéOpus

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

Imagem 11

O maior problema no uso do NéOPUS está na linguagem hospedeira. Smalltalk há algum tempo deixou de ser uma linguagem utilizada amplamente, fato que distancia a solução de integração do público utilizador. Soluções como o JEOPS/CEOPS++ estão, neste sentido, mais próximas do estado da arte e do conjunto de linguagens dominantes na atualidade, sendo entre elas também pertencente C#. Foi selecionada para esta análise por ter um papel importante como precursora do JEOPS e de grande parte das soluções de integração entre regras e objetos.

2.3.4 R++

A linguagem R++ nasceu no Bell Labs em 1994. Sua idéia era construir uma extensão da linguagem C++ adicionando o paradigma de regras. No entanto, essa extensão foi feita de forma diferente das realizadas até então. Ao invés de ter motores de inferência que se baseavam no casamento de padrões (como NéOPUS, JEOPS) realizados por algoritmos como o RETE para tratar que regras deveriam ser verificadas, R++ propôs uma alternativa radical: as regras são traduzidas na linguagem hospedeira e não há nenhum motor de inferência central sendo executado. Para a especificação de regras, R++ modificou a linguagem C++ simplesmente acrescentando um novo constructo: a regra, vista como um membro de uma classe.

Não eram, todavia, regras tradicionais de produção como as outras. R++ baseou-se em um tipo de regras diferenciado, denominado *path-based rules* [10]. Este tipo de regra surgiu em 1989 em uma pesquisa conduzida por James Crawford e Benjamin Kuipers, na *University of Texas*, em Austin, que gerou uma linguagem precursora de R++ chamada Algernon. Embora haja espaço de argumentação que as linguagens tradicionais de regras teriam uma expressividade maior para tratar conhecimento, esta decisão foi justificada por menores interferências do paradigma de regras *path-based* com o paradigma orientado a objetos, e pelo ganho considerável de performance observado. As regras *path-based* provaram, de qualquer modo, serem um modelo viável para integração entre regras e objetos.

As regras em R++ possuem sintaxe e semântica similar a *member functions* em C++, conforme pode ser visualizado na Imagem 12. São declaradas dentro das classes, operam diretamente nos objetos C++, respeitam controladores de acesso (*public / protected / private*), são membros herdados por subclasses, e possuem acesso à instância '*this'*, assim como métodos ou funções em C++. Ao contrário de funções, as regras são disparadas automaticamente ao invés de chamadas.

São denominadas *path-based rules* porque seguem estritamente um caminho de ponteiros entre objetos, a partir de 'this' para objetos relacionados. Desta forma, é dado aos objetos apenas a capacidade de acesso que eles possuem inerentemente, de tal forma que eles não podem suplantar os relacionamentos especificados na arquitetura orientada a objetos, como numa regra tradicional que poderia relacionar dois objetos que não têm nenhuma relação pré-estabelecida entre si, ou recursivamente entre suas relações.

O estudo responsável pela concepção de R++ afirma que o fato de *"outras linguagens de regras poderem realizar relacionamentos arbitrários entre objetos não relacionados"* [12] viola o princípio da localidade de referência projetado num modelo orientado a objetos, e que isso torna uma base de regras mais complexa, por ser mais

difícil de prever e entender seus caminhos. Principalmente por este motivo, R++ consegue uma integração com maior conformidade em relação ao paradigma orientado a objetos que as linguagens de regras de produção tradicionais.

R++

```
class Employee {
friend Manager;
   public: Employee (int salary);
   protected: monitored int salary;
};

class Manager : public Employee {
   rule maintainSalary;
   public: Manager (int salary);
   protected: monitored Set_of_p<Employee> staff;
};

rule Manager::maintainSalary {
   Employee * emp @ staff &&
   emp->salary > salary
=> set_salary(emp->salary);
}

Employee::Employee(int sal) : salary(sal) { }
Manager::Manager(int sal) : Employee(sal) { }
```

Imagem 12

Esse tipo de regra é chamado de *path-based*, uma tradução literal levaria a um "orientado a caminhos", pois as referências checáveis em suas condições atravessam caminhos entre objetos a partir de um objeto raiz (a instância '*this*'). No entanto o título dado a este tipo de regra parece ter sido atenuado por receio de soar pouco modesto em relação à aderência obtida com sua proposta de integração. Um nome mais adequado, porém talvez mais ambicioso, seria regras "orientadas a objetos", uma vez que é exatamente nisso que a regra se transforma, como um elemento que, de tão orientado a objetos, tornou-se um membro de classe e respeita, em R++, todo o paradigma orientado a objetos.

A maneira que R++ dispara regras está relacionada ao monitoramento das chamadas a métodos que modificam o estado do objeto e dos construtores. As regras são controladas por mecanismos naturais do paradigma orientado a objetos. Quando uma instância de uma classe é criada, o objeto correspondente é verificado pelas regras definidas para aquela classe. O único modo de desconsiderar regras em um objeto é destruí-lo, pois uma regra está sempre ativa em todos os objetos de uma classe, incluindo objetos de classes derivadas.

Sobre a decisão de tornar as regras membros de classe, R++ enxerga esta associação uma maneira natural de integrar regras com os outros elementos do paradigma orientado a objetos, colocando regras para trabalhar diretamente com objetos que são instâncias da classe. Assim como atributos e métodos de acesso a dados definem a *representação* de uma classe e funções e métodos definem a *interface* de uma classe, regras definem o *comportamento automático* de uma classe.

Outras considerações sobre como R++ trata regras são:

1. Uma vez que regras são membros de classe, então são definidos pelos desenvolvedores da classe, não por usuários dela. O fato de descrever comportamento de um conjunto de objetos do mesmo tipo, ainda que seja na forma de regras, quando esta especificação é realizada além dos limites da classe que os descreve, é considerado segundo a visão desta solução uma violação ao paradigma orientado a objetos.

2. Regras possuem acesso aos membros visíveis da sua classe imediata, ou seja, todos os ali declarados, sejam *public*, *protected* ou *private*, assim como os membros visíveis externamente de classes associadas.

   Apesar de estar em conformidade com a decisão defendida pela solução, esta característica desenha uma visão diferente da que considera que uma regra deve especificar conhecimento sobre um objeto apenas pelo seu comportamento externo [2].

   No entanto, a tecnologia da linguagem C++ permite que esta modularidade seja conseguida caso a aplicação a tenha como crítica. Em tais aplicações, o projetista da classe pode encapsular a estrutura de dados e definir, através de uma interface pública, as regras do objeto.

3. Regras definidas em uma classe que é herdada também são herdadas, isto é, também funcionam em instâncias de classes derivadas.

4. Uma regra em uma classe derivada pode sobrescrever uma regra de mesmo nome em uma classe básica.

5. Regras não podem ser chamadas, então não possuem parâmetros de entrada, nem tipo de retorno, tampouco especificação de acesso *public*, *protected* ou *private*.

   Em relação à especificação de acesso à regra, não parece fazer realmente muito sentido sua existência, uma vez que ela não pode ser chamada. No entanto, permitir que regras (todas ou algumas em específico) possam ser desativadas durante a execução da aplicação poderia ser uma alternativa interessante para o desenvolvimento de algumas aplicações, tendo em vista que regras em R++ estão sempre ativas em instâncias de

uma classe, e sua única maneira de impedir o comportamento automático das regras é destruir o objeto. O uso de um modificador poderia indicar que regras podem ter seu estado alterado apenas pela classe hospedeira (*private*), também por todas as classes friend (*protected*), ou também por classes derivadas (*public*), permitindo também uma quarta opção, onde a regra não pode ter seu estado alterado (*readonly*).

Também falta um mecanismo de acesso para especificar que regras podem ser sobrescritas (*virtual*), as que são definidas apenas em parte em uma classe abstrata, isto é, não instanciável e as classes que herdarem precisariam especificá-las (*abstract*), assim como as que nenhuma classe derivada poderia sobrescrever (*final*).

Realizando uma análise de R++ a respeito do seu "motor de inferência", na solução, ao contrário das linguagens de regras tradicionais, não é centralizado e está distribuído na implementação de cada objeto *regulamentado*. O algoritmo utilizado para o processamento de regras *path-based* é mais simplificado que um algoritmo de casamento de padrões, uma vez que regras em R++ simplesmente se transformam em funções em C++ quando compiladas, e não é necessário um interpretador em tempo de execução para realizar esta tarefa, tampouco um motor.

Para cada dado monitorado por uma regra, o compilador de R++ adiciona um dado de controle específico que, durante a execução, referenciará os objetos raízes. Quando uma mudança relevante é realizada em um dado de um objeto que não é raiz, cada regra disparada seguirá as referências aos objetos raízes e então são avaliadas suas condições em todos os caminhos que podem ser formados a partir daquela raiz e que passam pelo objeto modificado.

O algoritmo utilizado para o raciocínio das regras *path-based* é geralmente mais eficiente que o motor de inferência baseado em RETE. De acordo com um estudo comparativo [10], no pior caso o custo em R++ é de $O(rn^2)$ enquanto numa implementação RETE é de $O(rn^v)$, onde r = quantidade de regras, n = quantidade de objetos, v = quantidade de variáveis. Embora o pior caso seja raramente atingido na prática, outros resultados empíricos suportam a eficiência das regras *path-based*.

Um problema administrativo relacionado aos direitos comerciais da linguagem R++ a impede de ser utilizada. A empresa Lucent possui a patente da tecnologia, porém a AT&T é proprietária do software que realiza a tecnologia. R++ foi desenvolvida pelo Bell Labs, no entanto quando a Bell foi dividida, a propriedade intelectual de R++, desenvolvida nos laboratórios entre a AT&T e a Lucent, não teve uma definição, pois as duas companhias disputaram a sua propriedade sem acordo.

2.4 Resultado da análise

Esta seção descreve, em linhas gerais, a reflexão tida como conseqüência da análise dos trabalhos relacionados. As três soluções de integração entre regras não representam o universo total de soluções em mecanismos de inferência, mas são bons exemplares e criaram uma base suficiente para o presente trabalho. Mais especificamente o JEOPS e o R++ têm gerado, no confrontamento de suas estratégias bastante distintas, a principal fonte de decisões para a concepção da solução R#.

Ao verificar que a linguagem R# não teria dificuldades inviabilizadoras da incorporação da sintaxe diferenciada de C# e que pretende ser coerente do ponto de vista do paradigma de orientação a objetos, isso não é de todo suficiente para uma simbiose lingüística adequada. A linguagem de regras também precisa se alinhar com a visão de C#, sua filosofia, que encoraja a criação crescente de funcionalidades para programação em alto nível. Desta forma, R# também deve, em tese, prezar pela criação de constructos de linguagem inovadores que permitam uma expressividade maior do especificador de regras quando em contato com R#.

A maneira com a qual C# visualiza simplicidade também deve ser considerada. A linguagem disponibiliza constructos simples, mas que em conjunto, aninhados e acessados um pelo outro podem levar a aplicação a um nível altíssimo de complexidade. A complexidade deve ser permitida, pois algumas aplicações precisam fazer uso de mecanismos mais sofisticados. Prezar cegamente pela simplicidade pode tirar o poder de expressão e resolução de problemas de uma linguagem.

O restante desta seção foca na atividade de confrontar os paradigmas levantados por JEOPS e R++, na direção de obter uma decisão de paradigma de regras que permita uma uniformidade de integração com C#.

Regras *path-based* podem ser consideradas mais aderentes ao paradigma orientado a objetos do que regras que promovem o casamento de padrões, no entanto há uma questão aberta em relação ao poder de expressividade das regras. As regras de casamento de padrão podem relacionar quaisquer objetos que estejam no momento na base de objetos, enquanto uma regra *path-based* é direcionada a uma instância e só pode relacionar o comportamento automático deste objeto a um outro que ele possua, diretamente, uma relação.

Isto não é considerado, do ponto de vista das regras *path-based*, um problema, uma vez que relacionar em uma regra arbitrariamente objetos que não têm uma relação entre si não é uma atitude esperada de um programa doutrinado na boa prática da engenharia de software com orientação a objetos.

O foco deste trabalho não concentra-se na implementação pura de um mecanismo de regras para C#, mas na proposta de uma solução que consiga integrar

regras a C# da forma mais adequada possível do ponto de vista do paradigma no qual a linguagem hospedeira se baseia. Há uma inclinação, portanto, para trabalhar com a alternativa proposta pelo R++ e fazer uso no R# de regras path-based. C# possui mecanismos que permitem realizar o equivalente ao que o compilador de R++ faz com o código C++, transformando-o de modo a inserir as regras dentro das classes em C++. Bibliotecas voltadas para instrumentação de código possibilitam este tipo de implementação, e os mecanismos de reflexão são inclusive mais sofisticados do que em C++. A grande perda em não utilizar o paradigma do JEOPS está na expressividade das regras, como já dito. Programadores de bases de regras acostumados com o paradigma de regras tradicional podem ter dificuldades ao tentar especificar certos tipos de conhecimento, inclusive alguns que, declarados como na forma tradicional, não trariam grandes violações ao paradigma.

Suponha que existem dois elementos em uma lista e é válido realizar uma verificação entre eles. Eles não possuem, a priori, uma relação entre si, explícita, mas estão na mesma coleção. Isto é, não há uma relação explícita necessariamente entre eles, mas eles estão relacionados a um mesmo objeto, que possui essa coleção. E devido à qualidade da relação dos dois elementos a este objeto que contém a coleção ser a mesma – eles estão na mesma coleção, afinal –, eles são potencialmente relacionáveis, por mais que eles não se relacionem, eles poderiam. O esforço de ter essas relações mapeadas em ambos os objetos previamente, na etapa de arquitetura, tem pouco valor para a orientação a objetos. Para regras, que tratam de conhecimento de uma forma diferenciada de métodos, por exemplo, este tipo de relacionamento potencial pode ser aproveitado e gerar um ganho na expressividade da linguagem.

O paradigma de *pattern-matching*, se aproveitado em situações específicas como esta unido ao *path-based*, consegue ser complementar de forma coerente. Obviamente, R# não deve permitir que a expressividade obtida permita a relação entre objetos que são totalmente desacoplados um do outro, e também não apresentam potenciais relações, através de relações em comum com outros objetos.

Em R++, as regras são um membro da classe a que se referenciam. O fato de dizerem respeito ao comportamento de um objeto torna interessante essa decisão de colocá-las como membro de classe, e parece inclinar para a afirmação que do contrário há uma violação do encapsulamento. No entanto, para R# fazer o mesmo, precisaria não somente trabalhar junto a um projeto C# para criar regras, mas tornar-se uma extensão de C# que contivesse esse novo constructo: a regra. Fazer isso impediria compatibilidade de R# com novas versões de C#, além de gerar um acoplamento muito forte, de tal modo que um projeto C# que quisesse utilizar regras precisaria abdicar de certas decisões para tornar-se um projeto R#.

O fato de colocar regras como membro de classe é também geradora da seguinte situação indesejada: desenvolvedores de áreas diferentes, como regras e código C# convencional, estariam utilizando o mesmo arquivo, o que abre margem para uma série de inconvenientes de controle de versão, ainda que estas possuam soluções que as resolvam com eficiência. O fato de eles estarem trabalhando em um arquivo onde a totalidade do código deixa de ser de interesse mútuo de ambos abre espaço para, sem problemas, optar-se pela maneira de especificação de regras do JEOPS, isto é, fora de classes. Na própria linguagem C#, códigos de uma mesma classe que possuem objetivos bastante distintos, como descrição visual dos objetos de uma interface gráfica e o raciocínio lógico que os mantém são definidos em arquivos separados. Dessa forma, pode tornar-se até aderente à filosofia de C# que bases de regras, separadas dos objetos, os referenciem e especifiquem seu comportamento automático. Assim como C# aceitou e fundamentou a partir da sua versão 3.0 o uso de *extension methods*, que permitem a extensão do comportamento de um objeto declarada fora de sua classe, as bases de regras fazem o mesmo.

A crítica realizada em R++ a respeito da criação de modificadores de acesso e de sobrescrita são válidas e podem incrementar de maneira significativa o poder do mecanismo de regras. Modificadores de permissão à sobrescrita de regras em classes derivadas podem gerar melhoria no sentido de delimitar as formas que a herança em reespecificação de regras pode ocorrer, podendo fixar um conhecimento perenemente ou permitir que ele possa ser reespecificado. Também deve ser permitido que a sobrescrita apenas acrescente uma pré-condição a mais, uma ação a mais, ou alguns itens a mais. Suponha que existem duas classes Empregado e Gerente, onde Gerente herda o comportamento de Empregado, e Empregado possui uma regra FimExpediente, que verifica o horário e, caso tenha terminado o expediente vai embora. O Gerente também possui esta regra, no entanto, antes de ir embora precisa fazer mais algumas verificações. Em favor da prática do reuso, o Gerente não deve, ao sobrescrever esta regra, repetir as partes que já estão especificadas na regra do Empregado, mas apenas acrescentar suas verificações à regra base. Apesar de a implantação de modificadores de acesso em regras parecer uma solução de pouco valor prático, essa característica pode permitir o surgimento de arquiteturas bastante sofisticadas em bases de conhecimento. Ativar ou desativar regras específicas de uma base de conhecimento pode ser interessante em termos de *refactoring*, pois quando várias regras possuem uma mesma condição, entre outras, talvez torne-se mais viável ter uma regra que, possuidora desta condição, as ativa quando ela ocorre.

A seção a seguir descreve a proposta da solução R#, elicitada, principalmente, a partir das reflexões realizadas durante a análise dos trabalhos relacionados.

3. R#

Esta seção apresenta a proposta da solução R#, que pretende ser um mecanismo de regras integrado com a linguagem C# através de uma síntese cuidadosa, de modo a otimizar o grau de compatibilidade mútua entre as linguagens, seguir a filosofia da família .NET, especificamente da linguagem C#, respeitando as práticas e princípios relacionados ao paradigma orientado a objetos.

A Imagem 13 ilustra uma regra R# que especifica comportamento automático para a classe em C# *Employee*. A regra *AskForSalaryIncrease* está dentro da base de regras *EmployeeRules*, que declara que todas as regras pertencentes a esta base diz respeito a instâncias de *Employee*, ao fazer *"EmployeeRules : Employee"*. Cada base de regras está contida em um arquivo, podendo uma *rulebase* conter várias regras.

Imagem 13

A regra *AskForSalaryIncrease* da Imagem 13 já está suficientemente bem definida a ponto de ser considerada uma regra compilável em R#. Uma regra é denominada pela palavra-chave '*rule*' e possui um identificador, no caso *AskForSalaryIncrease*. Dentro das chaves por ela abertas encontram-se, basicamente, o *left-hand-side* ou lado *if*, que possui as condições que precisam estar verdadeiras para a regra ser disparada, e, após o divisor denotado pelo marcador '=>' está o *right-hand-side* ou lado *then*, que possui a ação ou o conjunto de ações a serem executadas caso a regra seja disparada.

Com um curto período necessário de aprendizado é possível compreender como descrever uma regra simples em R#. As funcionalidades apresentadas neste único exemplo já tornam possível especificar uma grande quantidade de regras. No entanto, R# tem uma proposta mais sofisticada que permite ao programador de bases de regras mais poder de especificação, além de possibilitar uma flexibilidade grande na arquitetura de regras a ser definida. Na seção a seguir, serão apresentados constructos de linguagem diferenciados de R# e situações nas quais eles podem ser utilizados.

3.1 Linguagem R#

Esta seção apresenta os elementos da linguagem R# de uma maneira mais detalhada, de modo a cobrir suas funcionalidades e mostrar como ela resolve, no nível lingüístico, os problemas que se propôs a solucionar.

3.1.1 Rulebase

Uma base de regras é definida num arquivo localizado dentro de um projeto R# e que contenha o mesmo nome que a base de regras, acrescido da extensão ".rs". Por exemplo, a base de regras *EmployeeRules*, que tem o objetivo de especificar as regras do objeto *Employee*, está localizada no arquivo "*EmployeeRules.rs*", que por sua vez está dentro de um projeto R#.

A sintaxe básica para a definição de uma base de regras é: *"rulebase NomeDaBase { regras }"*, entendendo-se por 'regras' a declaração de todas as que estão contidas na base de regras. Um arquivo de base de regras, por estar na plataforma .NET, precisa referenciar explicitamente o *namespace* das classes C# que utiliza, e isto é feito através da inclusão de elementos de sintaxe *using* no início do arquivo, antes de iniciar a declaração da base de regras. O exemplo A da Imagem 14 mostra a base de regras declarando usar o namespace *BusinessCompanySolution*, pois este contém a classe *Employee,* a qual a base de regras regulamenta.

A

```
using BusinessCompanySolution;

rulebase EmployeeRules : Employee { ... }
```

B

```
using BusinessCompanySolution;
using SyntheticAgentRules;

rulebase EmployeeRules : Employee { ... }
```

Imagem 14

É possível dentro de uma base de regras referenciar outra base de regras, para a realização de algumas operações mais sofisticadas que veremos a seguir. Caso a outra base de regras esteja em outro projeto R#, a declaração da base de regras deve fazer de acordo com o exemplo B da Imagem 14 ao projeto R# "*SyntheticAgentRules*".

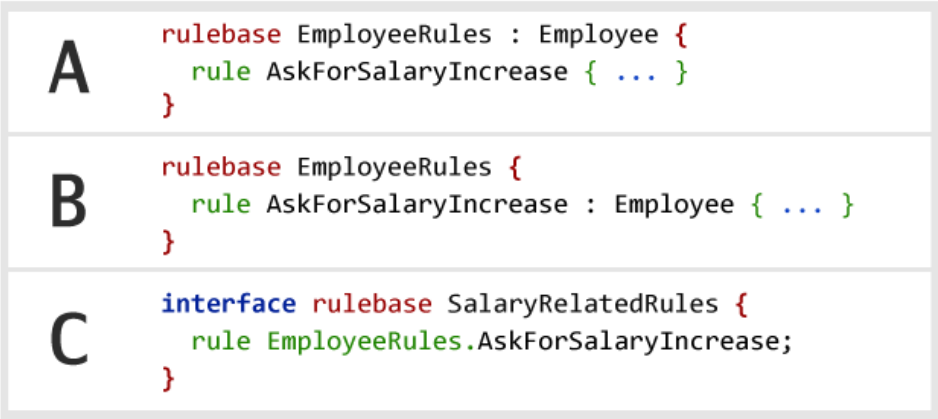

Imagem 15

A Imagem 15 tem a intenção de mostrar como as *rulebases* podem ser especificadas de modo flexível, pois dependendo da organização das regras que o projetista achar mais conveniente, podem ser realizadas especificações distintas.

Se uma base de regras faz uma declaração como no item A, descrevendo após o identificador da *rulebase* o "*: Employee*", na assinatura da *rulebase*, obriga todas as regras declaradas nela a regulamentarem a classe especificada pela *rulebase*, assim como não as permite mais declarar na assinatura da regra esta relação. Se o projetista das regras quiser dividir as bases de regras com relação às classes cujo comportamento automático será especificado, o exemplo A é um modelo adequado de declaração das bases de regras. A regra *AskForSalaryIncrease*, assim como todas as outras descritas nesta *rulebase* necessariamente dizem respeito ao comportamento de *Employee*.

O caso do exemplo B mostra que quando uma *rulebase* não define previamente que classe está sendo regulamentada, as regras devem fazê-lo. Sendo assim, regras numa mesma *rulebase* podem regulamentar classes diferentes, desde que a base de regras não defina que uma classe específica está sendo regulamentada através dela. O exemplo B mostra-se mais adequado quando o projetista da solução de regras acha melhor agrupar nas bases regras que não possuem necessariamente a mesma classe regulamentada, porém possuem uma relação semântica que torna mais viável aglutinar num mesmo arquivo regras relacionadas entre si. Também é uma boa opção caso o projetista não queira amarrar à base de regras – que pode funcionar como um mero agrupamento – a responsabilidade desta definição.

O exemplo C ilustra uma possibilidade na definição de bases de regras, que é declarar uma *rulebase* abstrata, que serve apenas como uma interface de acesso a regras, mas não as define. É adequada quando várias bases de regras são definidas como no exemplo A e a solução necessita visualizar uma base de regras através de uma outra organização. Como sugere o nome do exemplo C, esta *rulebase* abstrata aglutina todas as regras, de várias bases de regras, relacionadas a salários. A palavra-chave '*interface*' escrita antes da palavra-chave '*rulebase*' define uma base de regras abstrata e em tal tipo de *rulebase* só podem constar referências a regras e não

declarações, como a referência vista no exemplo C, onde só há a assinatura da regra, contendo antes do seu nome o nome de sua base de fato. No entanto, bases de regras, mesmo que não sejam abstratas, podem referenciar regras de outras *rulebases*. R# permite algumas operações com bases de regras, como já foi comentado, o que será explicado em maiores detalhes na próxima seção e introduzido nos próximos parágrafos. A criação deste tipo de base de regras é bastante útil para a concretização destas operações.

Uma das preocupações com relação ao mecanismo de regras *path-based* em R++ está no fato das regras estarem sempre funcionando, desde o momento da criação do objeto, e nada, a não ser destruir o objeto, pode parar a avaliação constante das regras em relação aos objetos. Foi observado em R++ a necessidade de permitir que regras possam ser ativadas ou desativadas. Muitas aplicações podem ser definidas sem este recurso, no entanto a possibilidade de ativar ou desativar regras, todas ou algumas em específico, pode ser uma funcionalidade interessante tanto no quesito expressividade quanto de um ponto de vista de arquitetura.

Permitir que uma regra ative uma outra (ou um conjunto de outras regras) quando sua condição for satisfeita é habilitar a linguagem de regras para o reuso e o *refactoring*, além de permitir uma melhor performance na inferência. Imagine que dezenas de regras possuem, pelo motivo de haver uma relação semântica entre elas, uma mesma condição em comum. Uma nova regra pode definir esta condição e, quando esta condição for satisfeita, ativar as outras regras associadas para que elas possam verificar suas condições diferentes.

Em R# é possível fazer esse tipo de operação e, pelo fato de uma base de regras agrupar regras que têm uma relação em comum, é possível também ativar ou desativar *rulebases*, o que significa, em última instância, efetuar essas modificações de estado nas regras associadas à base de regras em questão.

A partir desta explicação, justifica-se a necessidade de bases de regras abstratas, pois estas criam bases virtuais que referenciam regras de acordo com uma relação definida pelo projetista, de tal modo que elas não são bases declaradoras de regras, mas agrupadoras de regras que tem relações em comum mas estão definidas em outras bases. *Rulebases* abstratas também habilitam aplicações futuras que possam utilizar o R# a executarem estatísticas mais aprofundadas sobre o comportamento dos objetos segundo as regras definidas.

Há uma maneira de declarar, tanto nas regras quanto nas *rulebases*, se elas podem ou não serem ativadas ou desativadas. Visando guardar este recurso para usuários avançados da linguagem, por padrão elas não podem, e caso o projetista tenha o desejo de fazê-lo, deve colocar em uma linha acima da declaração da *rulebase* (ou em uma linha acima da declaração de uma regra) o atributo ‘[Lockable]’, para

definir que esta base de regras ou esta regra possui a propriedade de ser ativada ou desativada pela ação de uma regra específica.

Modificadores de acesso *public* ou *private* são possíveis em *rulebases*. Modificadores *private* impedem regras de outras *rulebases* referenciarem esta *rulebase* (como *interface*) ou alterarem seu estado (ativar ou desativar, assim como uma ou outra regra de dentro desta base de regras especificamente).

Modificadores *public* permitem que outros projetos R# possam realizar estas operações na base de regras. O modificador padrão (implícito, é válido quando não há nada escrito) permite que bases de regras num mesmo projeto possam se referenciar, e que regras em uma base possam modificar o estado de regras da outra, esta última quando o atributo [Lockable] está declarado. Não é possível ter regras que estão declaradas em *rulebases private* referenciadas em outras *rulebases* abstratas.

3.1.2 Ativar ou desativar regras e *rulebases*

A Imagem 16 mostra, em quatro exemplos, como ativar ou desativar regras e *rulebases* em R#. Os exemplos foram pensados não só para exibir a sintaxe da linguagem em relação a estas operações, como também para mostrar situações onde a aplicação deste recurso seja vantajoso.

A
```
rule IncreaseDifficultyToSecondLevel : Player {
    Skills.TotalSkills > 500000;
  =>
    activate rulebase SecondLevelRules;
}
```

B
```
rule SetFloatingOn : Player {
    FloatingEnabled;
  =>
    deactivate rulebase GravityRules;
}
```

C
```
rule SetFloatingOnToAllPlayers : Player {
    MagicButtonPressed;
  =>
    deactivate rulebase GravityRules forall;
}
```

D
```
rule BossConfrontationStarted : GameLogic {
    BossArrivedOnScenario;
  =>
    activate rulebase FightWithBossRules forall Players;
}
```

Imagem 16

O exemplo A descreve uma regra cuja definição é que, caso as habilidades gerais de um jogador passem de determinado ponto, habilita-se um conjunto de outras regras para promover um nível de dificuldade maior. Isto é, "se os seus *skills* aumentarem, ative as regras definidas na base de regras '*SecondLevelRules*' para este jogador". Sua sintaxe mais simples é composta somente pela palavra-chave '*activate'*, seguida do tipo de elemento que se deseja ativar (pode ser uma base de regras ou uma regra, e no exemplo é uma base de regras), seguida finalmente pelo nome do elemento a ser ativado. Praticamente a mesma sintaxe é utilizada para desativar um elemento, apenas trocando-se '*activate'* por '*deactivate'*. Ao ativar a base de regras *SecondLevelRules* para esta instância da classe *Player* , por motivos óbvios, apenas as regras em *SecondLevelRules* que regulamentam a classe *Player* terão efeito nesta operação. Se nesta base houver regras especificando outras classes, elas não serão ativadas ou desativadas, por não fazer sentido esta operação neste escopo, uma vez que elas não têm nenhum poder de inferência sobre a classe *Player*.

No caso B, regras relacionadas a gravidade são desabilitadas para um jogador quando ele tem a propriedade de flutuar válida. Da forma que está escrita, apenas a instância de *Player* que tiver a propriedade de flutuar válida terá a gravidade suspensa.

O exemplo C considera como poderia funcionar o mecanismo num cenário *multiplayer*, se houvesse uma regra que, quando um dos jogadores apertasse uma combinação de botões, o permitisse desabilitar as regras de gravidade para todos os outros jogadores. A mudança na chamada do *deactivate* foi apenas no final da sua declaração, onde foi escrito um '*forall'*. O *forall*, colocado desta maneira, significa que para todas as instâncias de *Player* esta base de regras será desativada.

Há a possibilidade de, ao invés de *forall*, utilizar um *for* <*instance*>, onde substitui-se <*instance*> por alguma instância específica que se queira desabilitar aquela base de regras. Há implicitamente em todas as chamadas que não tiverem *forall* ou *for* <*instance*> um "*for this*", o que quer dizer que estas operações terão reflexo apenas na instância que a regra que contém a operação regulamenta.

O caso D mostra que também é possível usar *forall* acompanhado de uma referência para uma coleção de objetos de uma mesma classe, obrigatoriamente também sendo da mesma classe regulamentada pela base de regras ou pela regra que está sendo ativada ou desativada nesta chamada. No exemplo D todos os jogadores que estão na coleção *Players*, encontrada na classe *GameLogic*, têm algumas regras ativadas após o confronto com o "mestre" da fase começar.

O mesmo atributo [*Lockable*], que precisa ser relacionado a uma rulebase ou regra para definir se elas podem ter seu estado de ativação alterado, tem uma propriedade que precisa ser modificada para permitir que seja possível ativar ou desativar uma instância sem ser ela própria, caso que acontece no uso de *forall* em

qualquer um de seus modos e no uso de *for* <*instance*> onde a instância não for *this*. Para permitir tal funcionalidade, deve-se escrever [Lockable(Globally=true)].

3.1.3 Choose

A parte destinada a declaração de que ações serão executadas caso uma regra seja disparada deve permitir o uso de qualquer tipo de comando C# dentro dela, porém foi levantada a necessidade de outro elemento específico para regras. A Imagem 17 mostra um constructo específico da linguagem R# que pode ser adicionado na parte de ações em uma regra: *choose*. A principal idéia que motiva sua existência é facilitar a implementação do não-determinismo através do uso de probabilidade.

```
rule AskForSalaryIncrease {
     Wage < 2500;
     Satisfaction["job"] < 5;
  =>
     choose
     {
      option[0.50]:
        Manager.Request(this, Request.SalaryIncrease);

      option[0.40]:
        if(Satisfaction["job"] < 3)
           Manager.Request(this, Request.Quit);
     }
}
```

Imagem 17

Mesmo que um determinado conjunto de pré-condições seja satisfeito, isso não quer dizer que o especificador deseje que uma regra sempre execute o mesmo tipo de ação. Através do uso de *choose*, é possível escolher entre duas ou mais ações com uma probabilidade ponderada entre elas.

No exemplo ilustrado pela Imagem 17, a regra *AskForSalaryIncrease*, quando detecta que o salário está abaixo de um determinado nível e a satisfação do empregado está baixa, não toma uma decisão imediatamente. Com uma probabilidade de 50%, ele escolherá a primeira opção contida no bloco *choose*, que será enviar um pedido ao gerente de aumento de salário. Com uma probabilidade de 40% ele entrará na segunda opção declarada no *choose*, e dentro dela verificará se a satisfação está criticamente baixa. Se estiver, pede ao gerente demissão. Como a soma dos valores atribuídos a opção é inferior a 1 (0.50 + 0.40 = **0.90**), há uma probabilidade de 10% de o empregado não tomar nenhuma decisão, nenhuma das alternativas ser escolhida.

Ao invés de valores definidos na própria regra como "0.50", é possível associar à opção uma referência para um valor em uma propriedade ou numa constante de uma classe, por exemplo. Caso a soma dos valores ultrapassar 1 é ignorada a opção de não decidir por nenhuma e há uma reponderação da probabilidade tomando por valor

base a nova soma dos valores. Há também a possibilidade de não colocar nenhum valor de probabilidade, descrevendo apenas "*option*:". Este caso é adequado quando o desenvolvedor queira dar igual oportunidade entre as escolhas.

Se houver dentro do bloco *choose* apenas um "*option*:", a linguagem atribui 50% de chance desta única opção ser escolhida, uma vez que não faria sentido criar uma estrutura choose-option caso se quisesse com probabilidade de 100% executar um comando. No caso onde se declara em uma ou mais cláusulas (options) a probabilidade e existem outras sem valor associado, o valor restante é dividido para estas. A próxima seção apresenta outro detalhe da linguagem, todavia introduz outro exemplo que também utiliza o bloco *choose*, dessa vez de maneira mais avançada, de tal modo que a leitura da próxima seção complementa esta explicação.

3.1.4 Declaração

Todas as regras tradicionais, baseadas em casamento de padrões, precisam realizar a etapa de declaração. É através dela que essas regras descrevem que tipos de objetos da memória de trabalho estão sendo regulamentados por ela. Em regras *path-based* esta etapa não é sempre necessária. Pelo fato de uma regra *path-based* sempre dizer respeito a uma instância específica de uma classe, ela pode fazer o uso (implícito ou explícito) de *this*, e referenciar diretamente propriedades do objeto.

No entanto, quando a verificação de uma regra envolve uma propriedade de um objeto que está contido em uma coleção pertencente à instância regulamentada, não é possível acessar este objeto sem utilizar um quantificador, o que precisa ser declarado na definição da regra, algo como "*considerando algum elemento x desta coleção, quando x tiver determinada condição verdadeira, aplicar ação*". Por este motivo, as regras em R# possuem, opcionalmente, espaço para declaração, o que está ilustrado no exemplo da Imagem 18.

A regra é para a classe *Manager*, e tem sua condição verdadeira quando um empregado – que está listado em *Employees*, propriedade que referencia uma coleção em *Manager* – possui salário menor que o merecido. É criada uma etapa de declaração, que cria uma variável '*employee'*, definindo a que classe pertence e em que coleção está inserida. É também criado neste exemplo uma variável '*deservedWage'*, que utiliza o método de um objeto de *Manager* que recebe um empregado e calcula o salário merecido. Esta variável foi criada com o objetivo de mostrar que é possível usar esta etapa para declarar outros tipos de informação e usá-las na etapa de condição ou ação.

Também é interessante permitir a operação entre itens diferentes numa mesma coleção, bastando na declaração fazer algo como *"Employee emp1, emp2 in Employees"* e em seguida compará-los, e a linguagem não comparará uma instância com ela mesma nesta situação. Esta operação recupera a expressividade de pattern-matching em potencial, com a vantagem de garantir que não serão confrontados elementos que realmente não se relacionam, nem diretamente nem possuem uma relação com um objeto em comum.

```
rule DoEmployeeSalaryIncrease : Manager {
[
     Employee employee in Employees;
     double deservedWage = MgmtTools.CalcDeservedWage(employee);
]
     employee.Wage < deservedWage;
  =>
     choose
     {
      option[ActionProbabilities.SalaryIncrease]:
        employee.Wage = deservedWage;

      option:
        if(employee.Wage < 0.7 * deservedWage)
        choose
        {
           option[ActionProbabilities.UrgentSalaryIncrease]:
             employee.Wage = deservedWage;
        }
     }
}
```

Imagem 18

Esta regra foi definida desta forma também para exemplificar o uso mais avançado de *choose*. Há um aninhamento de *chooses* para representar o seguinte raciocínio: quando o salário que o empregado tem é menor que o merecido, a regra dispara. De acordo com uma probabilidade "*ActionProbabilities.SalaryIncrease*", realiza o aumento para o salário merecido. Com probabilidade complementar, faz o seguinte: verifica se o salário do empregado está criticamente abaixo do merecido (no caso, estabelecido como limite 70% do merecido). Caso isso ocorra, com probabilidade "*ActionProbabilities.UrgentSalaryIncrease*" ele aumenta. Ou seja, há uma possibilidade de ele não aumentar o salário ainda assim. *ActionProbabilities* neste caso representa uma classe que contém pelo menos duas propriedades de acesso global: *SalaryIncrease* e *UrgentSalaryIncrease*.

3.1.5 Outras considerações sobre o uso da linguagem

É possível utilizar dentro do código C# que está nas regras R# referências para classes que utilizam o *design pattern Singleton*, assim como acessar constantes e propriedades estáticas em geral em classes, em todas as etapas. É possível chamar métodos estáticos na etapa de ação de qualquer regra R#. Também é possível, na declaração de uma regra, o uso de *final* e *override*, para controle de herança entre regras. Ao usar *final* na declaração de uma regra, impede-se que ela seja modificada em uma classe derivada. E ao usar *override*, realiza-se a sobrescrita.

3.2 Mecanismo de inferência de regras

Esta seção explica como deve ser o funcionamento do mecanismo de inferência de regras em R#. Ao contrário do JEOPS e a maioria das outras soluções de regras, não há, em R#, um único motor centralizado. O código R# compilado está distribuído nas classes compiladas de cada objeto regulamentado em C#, processo ilustrado pela Imagem 19.

Imagem 19

O processamento de regras em R# pode ser entendido através de três fases: disparo, avaliação e execução. Uma regra é disparada quando um objeto regulamentado tem um dado modificado e a propriedade que encapsula o acesso a este dado é considerada nas condições desta regra. Para o disparo ocorrer, é necessário um monitoramento da propriedade que pode modificar este dado, que pode ser considerado como uma "fase 0" que desencadeia o início das fases de processamento de regras. Uma regra também pode ser disparada quando uma nova instância de sua classe associada é criada. No paradigma *path-based* este tipo de disparo é denominado *relevant construction* (o outro acima citado é chamado de *relevant change*), e seu monitoramento implica na necessidade de, a cada nova instância criada, verificar todas as regras.

Quando uma regra é disparada, segue-se a sua fase de avaliação. Se todas suas condições forem verdadeiras, esta passa para a fase de execução e tem suas ações executadas. A execução da ação pode disparar outras regras, causando um efeito de "*forward chaining*" na execução de regras. Considerando este efeito, é preciso garantir que uma regra não seja executada baseada em valores antigos ou mais de uma vez sobre a mesma modificação.

Por exemplo, se uma mudança num dado de um objeto dispara duas regras de forma que as duas têm suas condições satisfeitas, após a execução da primeira, a

segunda pode não ter mais como verdadeiro o resultado da avaliação de suas condições. O mecanismo deve agir de forma segura com relação ao *forward chaining*.

O uso de um contador sobre as mudanças de cada propriedade é a solução usada por R++, que pode ser aplicada em R#. Cada propriedade monitorada tem um contador de modificações, e cada regra tem um outro contador para cada propriedade de sua condição, guardando as versões avaliadas ou executadas por ela, de tal forma que comparar os dois valores é suficiente para resolver o problema.

A Imagem 19 ilustra que partes previamente definidas da classe *Employee* uma regra *AskForSalaryIncrease* deve monitorar para tornar a inferência sensível aos dados especificados em suas condições.

No caso do construtor *Employee*, porque um novo elemento deve, de qualquer maneira, sempre ser verificado em relação às condições de todas as regras que o regulamentam. Em relação à *property Wage*, quando, em algum lugar do sistema, usarem este encapsulamento para modificar o valor de *wage*, esta deverá avisar às regras relacionadas a mudança, o que tem como conseqüência a verificação das condições das regras que consideram *Wage* no seu *left-hand-side*.

No tocante à *property Satisfaction*, pelo fato de ela ser um encapsulamento para uma outra estrutura, o uso combinado do monitoramento da *property* com a criação de *triggers* para os métodos *Add*, *Remove* e dos operadores de mudança do valor de uma chave em uma coleção *Dictionary* são suficientes para fazer as regras devidas serem verificadas quando houver mudança. Esta última alteração, apesar de não ser trivial devido ao encapsulamento do comportamento das coleções em C#, pode ser realizada de várias formas, desde que os dados só estejam disponíveis a classes externas através do encapsulamento. Uma dessas formas seria a substituição de Dictionary por uma estrutura que estendesse seu comportamento, sendo esta alteração realizada automaticamente, em tempo de compilação, por R#.

Algumas novas estruturas são necessárias para o controle de tais regras, dada a necessidade de armazenar informações como *"esta regra está ativada para esta instância"* ou *"esta regra está ativada para instâncias desta classe"*, assim como também informações sobre que regras devem ser verificadas quando determinadas propriedades são modificadas. Todas elas não precisam, no entanto, estar guardadas nas classes as quais as regras se referem. Podem ser armazenadas em gerenciadores deste tipo de informação disponibilizados por uma biblioteca fornecida pelo R#, ao invés de dispersas nas classes regulamentadas. Esta decisão impactaria numa menor alteração no código C#, também trazendo como conseqüência uma centralização parcial dos dados de inferência. Como estas informações dizem respeito unicamente ao mecanismo de raciocínio baseado em regras implementado, pode ser considerada

adequada esta centralização resultante, pois faz com que a instrumentação de código C#, que causa modificações no código base, só seja necessária para a etapa de monitoramento dos dados, isto é, dos "fatos".

Outros dados, no entanto, encontram uma solução mais eficiente se estiverem nas classes regulamentadas. O exemplo ilustrado pela Imagem 20 considera a situação em que uma regra tem em sua condição não apenas as propriedades diretas de uma classe, mas propriedades de outras classes pertencentes à instância da classe regulamentada. A propriedade *Happiness* está na classe *Mood*, que mesmo não sendo a classe diretamente regulamentada pela regra *AskForBreak*, tem a modificação no dado encapsulado por *Happiness* monitorada. Para este monitoramento dar origem ao disparo da regra *AskForBreak* em relação a instância de *Employee* que tiver *Mood.Happiness* modificado, a classe *Mood* precisa armazenar, para cada instância sua, uma referência para o objeto raiz deste caminho "*Mood – Happiness*".

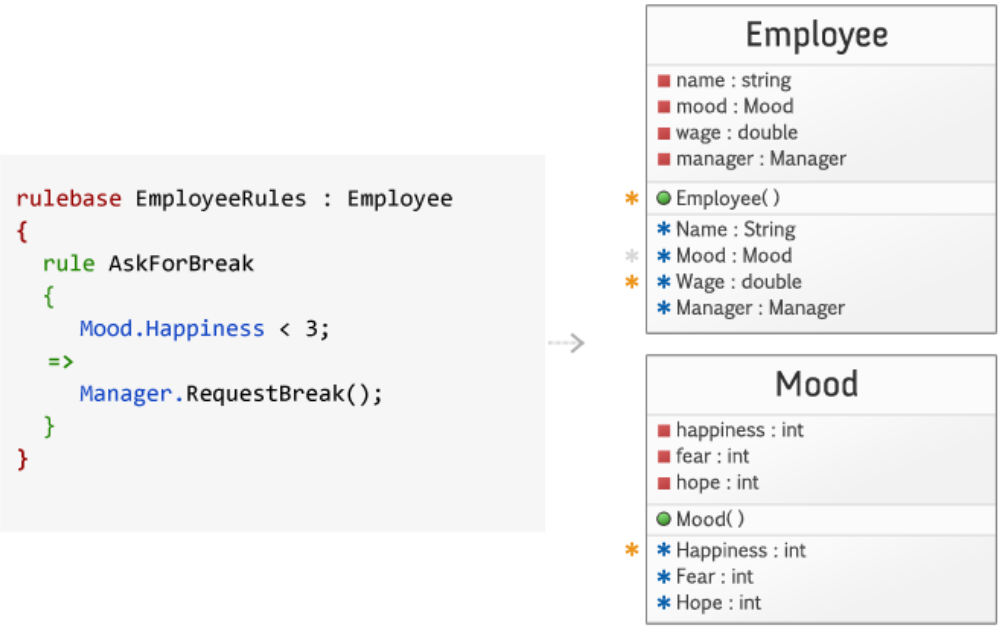

Imagem 20

Considerando um caminho maior, como *"Department – Director – Mood – Happiness"*, cada um destes, menos o *"this"*, terão uma referência para sua raiz imediata, ou seja: no caso de *Happiness*, *Mood*; no caso de *Mood*, *Director*. Por este tipo de dado estar relacionado diretamente à etapa de monitoramento, encontra sua melhor maneira de armazenamento dentro das classes direta ou indiretamente consideradas na etapa de avaliação das regras.

A integração de um projeto de regras R# aos elementos regulamentados na solução C#, devido às possibilidades permitidas pela plataforma .NET, podem ser realizadas de duas maneiras: em tempo de compilação ou em tempo de execução. As duas formas apresentam vantagens e desvantagens.

A Imagem 21 mostra a maneira de realizar a integração em tempo de compilação. Explicando o exemplo, trata-se de uma solução com dois projetos: um deles em C# chamado *Game*, e o outro sendo um projeto em R# que especifica regras para *Game*, chamado *GameRules*. Inicialmente, o projeto Game é compilado, o que gera um assembly em .NET.

Este *assembly* tem seu código modificado na etapa de compilação de R#, que adiciona ao código existente o comportamento específico de monitoramento, isto é, acrescenta código nas propriedades consideradas nas condições, nos construtores de objetos relacionados às condições das regras e cria nas classes de objetos indiretamente referenciados na etapa de avaliação referência para o(s) objeto(s) raiz(es) da sua propriedade. O plural é válido pois um mesmo objeto 'a' pode ser referenciado por dois outros 'b' e 'c', e estes estarem sobre a influência de uma mesma regra que considere 'a'.

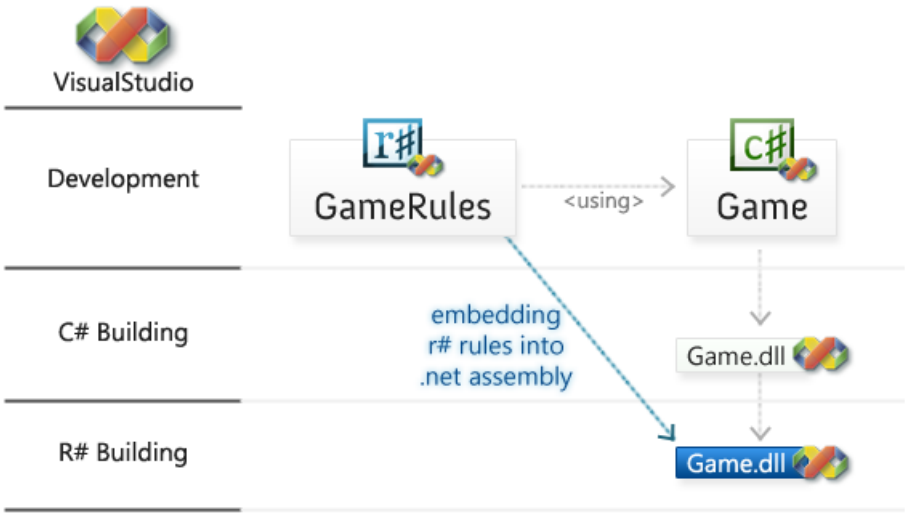

Imagem 21

Também faz parte da modificação uma operação, esta mais delicada, com o tratamento em relação ao monitoramento de coleções. Ao invés de usar as coleções normais de C#, é possível, como estratégia para monitorar a inserção ou remoção de um elemento na coleção, estender o comportamento das coleções existentes em C# adicionando o código de monitoramento nos métodos que realizam estas operações. Estas coleções estendidas podem estar disponíveis na biblioteca disponibilizada por R#, e a instrumentação apenas substituir as estruturas iniciais pelas estendidas. Ressaltando que esta substituição é uma alternativa possível apenas se for respeitado o encapsulamento no acesso a estas estruturas. A plataforma .NET apresenta outras alternativas que envolvem o uso da .NET Profiling API, no entanto utilizar esta

biblioteca, ao menos no momento da análise decorrente do presente estudo, pareceu uma solução menos viável por conta da sua complexidade.

A Imagem 22 mostra a maneira de realizar a integração em tempo de execução. Durante a etapa de compilação, ao invés de modificar o *assembly* do projeto *Game*, o projeto *GameRules* gera um novo *assembly*. Este contém o gerador de mudança de código que só é realizada em tempo de execução.

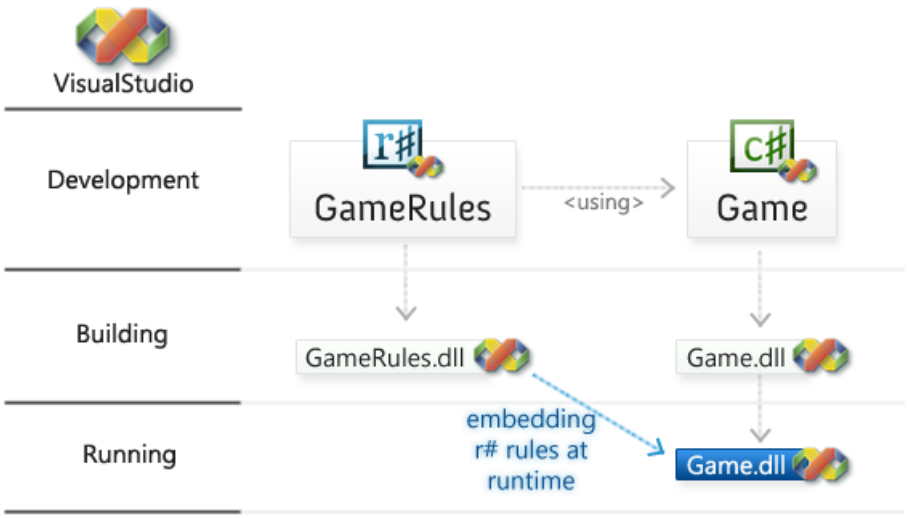

Imagem 22

A vantagem do segundo modelo de integração está em não modificar o código compilado do projeto C# em tempo de compilação. Assim, seu *assembly* não fica necessariamente submetido às regras do projeto *GameRules*. O *release* deste *assembly* fica mais limitado à produção resultante do projeto C#, sem sofrer modificações realizadas por outros agentes externos.

No entanto, o fato de realizar este processo em tempo de execução implica na necessidade de se chamar explicitamente, do projeto executável na solução, um controlador da integração (fornecido pelo compilador de C#), que carregará o projeto R# contido no .dll informado e integrará suas alterações no *assembly* de *Game*. Isto também pode ser causador – a depender do tamanho das bases de regras contidas no projeto e da performance da implementação do compilador – de uma lentidão na inicialização da solução que usar este tipo de estratégia.

O primeiro modelo surge mais adequado quando não há uma preocupação aguda em relação a questões de acoplamento, de modo que quando for executado este tipo de integração, o assembly do projeto C# nunca funcionará sem R#. Apresenta a vantagem de, no máximo, transferir uma possível lentidão na compilação para o tempo de compilação, mais aceitável que uma demora na sua execução.

Não há uma melhor escolha genericamente. As duas formas são complementares, pois cobrem os dois tipos de decisão de projeto possíveis nesta situação.

4. Implementação de R#: *Proof-of-Concept*

Devido à alta complexidade de desenvolver a solução R# em um período de tempo tão curto como o deste projeto, foi realizada uma implementação sem o objetivo de lançar R# como produto, mas com a intenção de provar o conceito da proposta, mostrando que é possível realizar esta integração em C#, da forma que foi apresentada. O foco do trabalho não era implementar uma solução de regras, foi estabelecido desde o princípio que o foco estava na análise dos conflitos existentes e proposta de uma solução otimizadora da resolução destes. Portanto, a elaboração da solução criou uma proposta de escopo amplo. Somente a implementação completa do compilador de R# já seria demasiado extensa, pelo simples fato de haver uma aceitação na integração de todos os constructos da linguagem C#.

A implementação PoC (*proof-of-concept*) de R# tem sua solução dividida em dois projetos principais, ilustrados na Imagem 23.

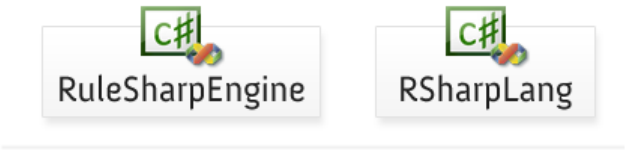

Imagem 23

O projeto *RuleSharpEngine* contém classes que provêem funcionalidade para as classes C# regulamentadas. O projeto *RSharpLang* contém a implementação PoC do compilador de R#, utilizando a plataforma *ManagedBabel* do *Software Development Kit* do *Visual Studio* 2005 [13]. A decisão de implementação do projeto PoC levou em consideração a centralização do máximo possível de código no *framework* de R# (contido em *RuleSharpEngine*), deixando a cargo da instrumentação programada pelo compilador no código C# apenas o código de monitoramento que é necessário para disparar as regras (fase 0). Alguns dados de controle deveriam ser inseridos no objeto, mas por simplificação estes dados estão centralizados no motor. Este fato criou a necessidade de um mecanismo, como existente no JEOPS, de conter uma base de objetos regulamentados. Para este projeto foi apenas considerada a integração das regras com os projetos C# envolvidos em tempo de execução.

A Imagem 24 contém o código necessário para realizar esta integração PoC entre regras e objetos. No método *main* do projeto C#, inicializou-se um objeto *RuleEngine*, motor responsável pelo controle da inferência, na linha seguinte adicionou-se as *rulebases* contidas no *assembly ProjectRules.dll*, sendo necessário colocar o caminho completo do arquivo para ser carregado, informação omitida na figura por questões de legibilidade e espaço.

```
public class Program
{
    public static void Main()
    {
        RuleEngine ruleEngine = new RuleEngine();
        ruleEngine.AddRuleBase(@"ProjectRules.dll");

        SyntheticAgent agent = new SyntheticAgent();
        agent.Wage = 100;
        agent.Angry = false;

        ruleEngine.Include(agent);

        //@
        agent.FirePropertyChanged =
            new PropertyChanged(ruleEngine.FirePropertyChanged);
        //@

        agent.Wage = 75;

        if (agent.Angry)
        {
            Console.WriteLine("Agent got angry");
        }
        else
        {
            Console.WriteLine("Agent is not angry");
        }
    }
}
```

Imagem 24

Em seguida, criou-se um objeto de uma classe regulamentada pelas regras de *ProjectRules*, chamada *SyntheticAgent*. Inicialmente, *Wage* tem valor 100 e *Angry* é *false*. O que este teste fez foi baixar o valor de *Wage* para 75, e o motor de inferências de regras modificou, por conter a regra descrita na Imagem 25, *Angry* para *true*. No final da execução deste método, é impresso *"Agent got angry"*.

O código que está entre '//@' significa que é autogerado por R# em tempo de execução. Portanto, apenas no momento de runtime será adicionado esta linha, que associa ao *agent* a referência para o método de *ruleEngine* que ele deverá chamar quando tiver detectado uma mudança em alguma propriedade sua. Quando *Wage* for modificada, o agent chamará este método, passando como parâmetro sua instância e especificando que propriedade foi modificada.

```
rule GetAngry : SyntheticAgent {
     Wage < 80;
     !Angry;
   =>
     Angry = true;
}
```

Imagem 25

A Imagem 26 exibe os membros públicos das principais classes que compõem o projeto *RuleSharpEngine*. A intenção não é de cobrir toda a implementação contida nelas, mas explicar como se comportam e para que servem suas instâncias.

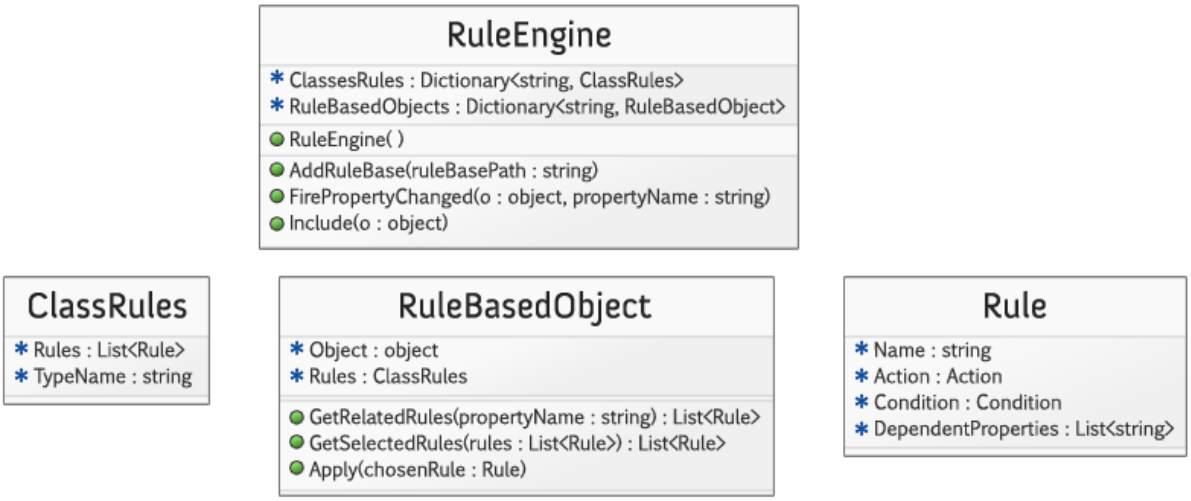

Imagem 26

A classe principal do projeto é *RuleEngine*. Ela é responsável por controlar os objetos regulamentados e conectá-los ao comportamento automático descrito através das regras, se houver. Provê funcionalidades para incluir bases de regras dentro de um projeto (podendo ser várias), podendo receber vários *assemblies* de projetos de regras. A maior parte do ciclo de inferência de regras é nela definido.

ClassRules é uma classe que contém um conjunto de regras descrito para uma classe em específico. Serve de estrutura para as classes *RuleEngine* e *RuleBasedObject*. A classe *Rule* representa uma regra, com suas condições e ações, assim como um conjunto pré-processado de que propriedades que, quando modificadas, podem afetar o resultado da verificação de suas condições. As propriedades *Condition* e *Action* são de tipos *delegates* definidos, de modo que dentro do *delegate Condition* pode constar qualquer código C# de avaliação desde que se retorne no final um resultado booleano, e dentro do *delegate Action* pode constar qualquer código, e não há retorno.

A classe *RuleBasedObject* representa um objeto submetido à inferência de regras no *RuleEngine*. Guarda uma referência para o objeto no mundo real e as regras que são válidas para ele. Tem um método *GetRelatedRules*, que ao receber o nome de uma propriedade sua (ou de um objeto seu em última instância), retorna as regras nas quais esta propriedade ocorre no *left-hand-side*. Contém ainda um método *GetSelectedRules*, que tem como entrada um conjunto de regras e retorna as regras que possuem sua condição verificada como verdadeira para aquele objeto. A *RuleEngine* realiza esta operação, mediada pelo *RuleBasedObject*. Quando o *GetSelectedRules* retorna mais de uma regra, é acionado uma estratégia de resolução

de conflito, que pode ser customizada pelo usuário (ver Imagem 27), e em seguida, após solicitar à estratégia de resolução de conflitos a regra escolhida para ser executada, é chamado no *RuleBasedObject* o método *Apply*, que recebe a regra escolhida e a aplica na instância cujo método foi chamado.

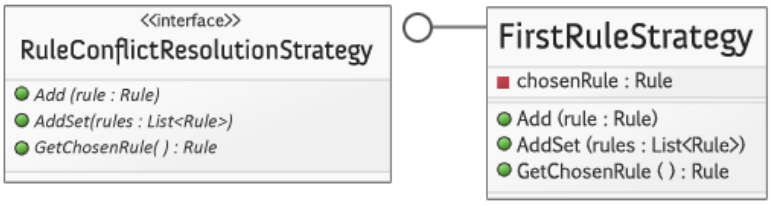

Imagem 27

O projeto *RSharpLang*, por sua vez, contém o material necessário, de acordo com o *framework Managed Babel* [13], para a compilação de um projeto R#. Tem como seus recursos principais:

- Um arquivo **"lexer.lex"**, que especifica os *tokens* da linguagem R# (numa implementação real, também especifica os *tokens* de C#, por extensão), e é, através do gerador *MPLEX* [14], base para a geração automática de um *Scanner*.
- Um arquivo **"parser.y"**, que define a gramática da sintaxe R#. Também é necessário numa implementação completa de R# definir toda a gramática da sintaxe de C#, além de possíveis restrições para a aplicação desta gramática em cada parte de uma regra em R# (exemplo: no *left-hand-side* de uma regra só podem estar expressões de avaliação de C#). O arquivo é, através do gerador *MPPG* de *Managed Babel* [15], transformado em uma classe em C# que realiza o *parsing*.
- Uma classe ***Configuration***, que define, além de nome e extensão da linguagem, outras informações, como por exemplo a coloração das palavras-chave que terão destaque na linguagem.
- Uma classe ***Resolver***, que, a partir da árvore gerada pelo *parser*, realiza a compilação da forma definida pelo usuário do *framework*.

O material gerado pelos dois projetos não foi suficiente, como já dito, para uma versão de R# equivalente à proposta realizada. Também não é possível, a partir do material implementado, tornar o PoC de R# viável para utilização em projetos reais. Sua aplicação pode ser realizada em projetos experimentais, que tenham como objetivo principal não seu usufruto simplesmente, mas a continuação do seu desenvolvimento numa direção de finalização do produto R#.

5. Conclusões

A realização deste trabalho foi muito importante para o entendimento claro dos conflitos existentes entre os paradigmas de regras e de orientação a objetos. A análise de duas alternativas mais significativas de integração – o JEOPS, sucessor do NéOPUS, e o R++ – possibilitou, no confrontamento de suas abordagens e crenças, a definição de uma solução coerente em relação aos resultados obtidos da avaliação.

Embora a implementação de R# tenha sido apenas uma prova de conceito e não tenha, na prática, o poder de especificar as regras R# tais como elas foram propostas neste trabalho, esta implementação foi suficiente para delinear os passos a serem seguidos num futuro desenvolvimento real da solução em C#.

As decisões tomadas pela solução R# poderão implicar numa aceitação pouco unânime por parte da comunidade de programadores de bases de regras, pois estes muitas vezes estão acostumados a especificar conhecimento a partir de outros paradigmas que R# não suporta. Todavia, o fato da linguagem ter seu objetivo alcançado na dimensão da uniformidade de integração é considerado mais relevante.

Diversos trabalhos futuros são possíveis a partir deste que foi realizado. O primeiro deles seria a implementação de fato da solução R#, lançá-la como produto. A criação de uma solução integrada ao *Visual Studio* de *Debug* também ajudaria bastante no desenvolvimento de regras. Aplicações que criem diagramas que exibam visualmente a modelagem de regras, preferencialmente contendo a relação com as classes C# regulamentadas, pode ser de grande valia para um projeto que utilize R#. Pelo fato de C# ser uma linguagem sempre em evolução, trabalhos futuros também incluiriam a adequação de R# a novos elementos da linguagem hospedeira.

A proposta da solução R#, além de resolver de maneira cuidadosa as diferenças entre os paradigmas integrados, também promoveu, por aderir não só tecnicamente à linguagem, o seguimento mais amplo da filosofia de C#. Esta adesão resultou num mecanismo de definição de regras de alto nível, com constructos que permitem uma expressividade maior na especificação de conhecimento implícito, habilitam programadores de bases de regras a tomarem decisões de arquitetura mais elaboradas e ampliam as possibilidades de inserção de não-determinismo através de regras. Com R# é possível definir uma base de conhecimento dinâmica, onde conhecimentos específicos têm sua inferência ativada apenas a partir de um dado momento, e este fator é um dos maiores diferenciais da solução. A criação do constructo *choose* também tem grande valor no incremento de expressividade em regras. O trabalho, desta forma, apontou com um bom grau de inovação e originalidade sobre o estado da arte dos mecanismos existentes para a integração entre regras e objetos.

6. Referências

[1] ROLF, Bertil (2004) **Two Theories of Tacit and Implicit Knowledge**. *Blekinge Institute of Technology, Ronneby Sweden.*

[2] BOUAUD, J.; VOYER, R. (1994) **Behavioral match: embedding production systems and objects.** *Proceedings of OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems. Paris: Laforia.*

[3] D'HONT, Maja; GYBELS, Kris; JONCKERS, Viviane (2004) **Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis.** *SAC '04, March 1417.*

[4] CLANCEY, William J. (1981) **The Epistemology of a rule-based expert system: a framework for explanation.** *Report No. STAN-CS-81-896. Stanford University, Stanford CA.*

[5] SHOHAM, Y. (1993) **Agent Oriented Programming.** *Artificial Intelligence 60 51–60.*

[6] ALBAHARI, Ben (2000) **A comparative overview of C#.** *Disponível em http://genamics.com/developer/csharp_comparative.htm . Acessado em junho de 2008.*

[7] FIGUEIRA, Carlos (2000) **JEOPS - Integração entre objetos e Regras de produção em Java**. *Dissertação de Mestrado, CIn – UFPE.*

[8] BARBOSA, Pablo (2006) **CEOPS++ - Integração entre Objetos e Regras de Produção em C++**. *Trabalho de Graduação, CIn - UFPE.*

[9] PACHET, François (1995) **On the Embeddability of Production Rules in Object-Oriented Languages.** *Journal of Object-Oriented Programming, vol. 8, n. 4.*

[10] CRAWFORD, James M., et al. (1996) **Path-Based Rules in Object-Oriented Programming.** *Thirteenth National Conference on Artificial Intelligence (AAAI-96).*

[11] FORGY, Charles (1982) **Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem.** *Artificial Intelligence 19 17–37.*

[12] LITMAN, Diane, et al. (2002) **R++: Adding Path-Based Rules to C++.** *Knowledge and Data Engineering, vol. 14 n. 3 638-658.*

[13] GOUGH, John (2006) **Managed Babel Overview.** *Queensland University of Technology, Brisbane Australia. July 21, 2006*

[14] GOUGH, John (2006) **The MPLEX Scanner Generator.** *Queensland University of Technology, Brisbane Australia. August 21, 2006*

[15] GOUGH, John; KELLY, Waine (2006) **The MPPG Parser Generator.** *Queensland University of Technology, Brisbane Australia. August 21, 2006*

Assinaturas

_____

Geber L. Ramalho (Orientador)

_____

Victor Cavalcanti Rodrigues (Aluno)