Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Revisão de Crenças em Lógica de Descrição e o Plug-In para o Protégé

MAC499 - Trabalho de Formatura Supervisionado
Aluno: Fillipe Manoel Xavier Resina
Orientadora: Profª Dra. Renata Wassermann

São Paulo, 1º de dezembro de 2010

Sumário

Parte I
Parte Técnica

1 Introdução

As pesquisas na área de representação de conhecimento têm crescido constantemente, incentivadas pela necessidade cada vez maior de modelos de descrição de um mini-mundo dado a fim de desenvolver aplicações eficientes para as mais variadas finalidades. Peter Gardenfors, professor de Ciência Cognitiva na Universidade de Lund (Suécia) e renomado pesquisador da área, afirma que o problema de representação de conhecimento é um dos mais importantes nas pesquisas atuais em Filosofia, Inteligência Artificial e Ciência Cognitiva em geral. Nesse contexto, as ontologias têm um papel muito importante, pois são elas que nos fornecem uma representação formal do conhecimento, por descreverem conceitos e relacionamentos que são importantes em um domínio particular.

A partir de então, as Lógicas de Descrição ganham muito importância, tornando-se um dos principais formalismos para representação de conhecimento e o núcleo de sistemas para tal fim. Elas são um subconjunto estruturado da Lógica de Primeira Ordem e, apesar de serem menos expressivas que esta última, possuem grande capacidade de representação para sistemas baseados em conhecimento, além de existirem métodos eficientes de raciocínio e inferência associados às Lógicas de Descrição. De volta às ontologias, existem softwares para edição das mesmas e o principal tem sido o Protégé[1], software livre que permite criar e editar uma ontologia, processo este baseado em Lógicas de Descrição. Isso é feito por meio de uma interface gráfica e é possível exportar em diversos formatos a base de conhecimento criada.

Muito se discute sobre a necessidade de utilização de ontologias no desenvolvimento da web semântica. Porém, na web (e em muitos outros campos), conhecimento não é estático e é justamente por conta desse dinamismo que o estudo de revisão de crenças ganha importância. Em um âmbito mais geral, a teoria de revisão de crenças é desenvolvida e estudada a fim de descrever como um agente, munido de um conjunto de crenças, deve mudá-las quando confrontado com novas informações, sejam esses agentes seres humanos, programas de computador ou sistemas dos mais variados, baseados em procedimentos racionais. Segundo Gardenfors, solucionar o problema de

[1] http://protege.stanford.edu/

encontrar uma representação de conhecimento apropriada é de pouca ajuda se não se souber como evoluir os estados epistêmicos (de conhecimento) à luz de novas informações. Para as ontologias, aplicar um método eficiente de revisão de crenças, ou seja, que elimine inconsistências e provoque mudanças mínimas, é essencial. Assim, faz-se necessário que ferramentas tais como o Protégé tenham essa capacidade, interagindo com o usuário na revisão de sua ontologia.

2 Lógica de Descrição

Os estudos e pesquisas na área de Lógica, Inteligência Artificial e Representação de Conhecimento buscam definições de linguagens formais para representar o conhecimento e também desenvolver métodos de raciocínio e inferência associados às mesmas. Assim, é de grande interesse que os sistemas desenvolvidos nesse propósito sejam capazes de extrair informações implícitas à base de conhecimento, além de poderem armazenar conhecimentos adquiridos.

Nesse âmbito, as Lógicas de Descrição têm sido o principal formalismo para representar conhecimento, em especial conhecimento terminológico. Como subconjuntos estruturados da Lógica de Primeira Ordem, detêm uma semântica bem definida (semântica formal) e são, nos casos computacionalmente interessantes, decidíveis, ao contrário da inferência em Lógica de Primeira Ordem. Além disso, a linguagem OWL[2] (Web Ontology Language), a qual é uma recomendação da W3C desde 2004 para representação de ontologias, é baseada nas Lógicas de Descrição. A OWL está intimamente ligada à Web Semântica e é utilizada como padrão para definir e instanciar ontologias na Web, sendo que a recomendação da mesma aplica-se às suas três versões: OWL-Full, OWL-DL e OWL-Lite, aqui em ordem decrescente de poder de representação. Falaremos das mesmas mais adiante. Vale ressaltar que a OWL-Full, justamente pelo maior poder de representação, é indecidível.

Denotemos, daqui em diante, Lógica de Descrição por LD. Em LD, o conhecimento é dividido em duas partes: TBox e ABox. Uma TBox refere-se às terminologias, ou seja, o conhecimento "intencional" sobre um determinado domínio, definindo conceitos e suas propriedades. Uma ABox refere-se às asserções, ou seja, o conhecimento "extensional" do domínio, especificando indivíduos de um ou mais conceitos.

O que difere uma DL em particular de outra são, a partir de conceitos atômicos e papéis, os novos conceitos e papéis possíveis de ser formados por meio dos construtores da linguagem. A lógica $\mathcal{ALC}$ engloba, dados conceitos A e B e um papel P, união ($A \sqcup B$), intersecção ($A \sqcap B$), complementar ($\neg A$) e valores de restrição ($\forall A.P$ e $\exists B.P$). Também permite como axiomas, para conceitos, subsunção ($A \sqsubseteq B$), equivalência ($A \equiv B$) e asserção ($A(x)$), além da asserção de papéis ($P(x, y)$).

A lógica denotada por $\mathcal{SHIF}(\mathcal{D})$ engloba $\mathcal{ALC}$ e, ainda, hierarquia de papéis ($P \sqsubseteq Q$), papéis inversos, transitivos e funcionais e tipos de dados. Só para exemplificação de papéis inversos, podemos representar que o papel *hasIngredient* é o inverso do papel *isIngredientOf*. Já a lógica $\mathcal{SHOIN}(\mathcal{D})$ en-

[2] http://www.w3.org/TR/owl-guide/

globa $\mathcal{SHIF}(\mathcal{D})$ mais restrições de cardinalidade em papéis ($\leq_n P$ e $\geq_n P$), por exemplo. Foi demonstrado que OWL-Lite é equivalente a $\mathcal{SHIF}(\mathcal{D})$ e OWL-DL é equivalente a $\mathcal{SHOIN}(\mathcal{D})$[4].

Em uma TBox, utilizamos esses axiomas vistos acima para representar as terminologias. Por exemplo, modelando o domínio Pessoa, poderíamos ter:

- Pessoa $\equiv$ Homem $\sqcup$ Mulher
- Aluno $\sqsubseteq$ Pessoa
- Mulher $\equiv$ Pessoa $\sqcap$ Fêmea
- Mãe $\equiv$ Mulher $\sqcap \exists$temfilho.Pessoa
- SuperMãe $\equiv$ Mãe $\sqcap \geq_3$temfilho.Pessoa

Aqui podemos ver que toda mãe é uma pessoa do sexo feminino e tem pelo menos um filho que é pessoa. Também que uma super-mãe é definida como uma mãe que tem 3 ou mais filhos.

A principal forma de inferência sobre uma TBox denomina-se *"concept subsumption"*, isto é, verificar se um conceito é mais geral que outro, alcançando inferências da forma $C \sqsubseteq D$. Todo sistema gerenciador de conceitos em LD deve ser capaz de realizar esse tipo de inferência.

Em uma ABox, temos sentenças sobre indivíduos específicos expressos de duas formas: declaração de um indivíduo como parte de um determinado conceito ou da relação entre ele e outro(s) indivíduo(s) por meio de um "papel" (*role*, conhecido como propriedade). Por exemplo:

- Aluno(FILLIPE)
- temFilho(ELOISA, FILLIPE)

A principal forma de inferência sobre uma ABox denomina-se *"instanciação"*, isto é, verificar se um indivíduo é instância ou não de um conceito.

Vários motores de inferência foram desenvolvidos para oferecer esses tipos de serviço ao usuário (subsunção de conceitos e instanciação). Outro serviço fundamental que deve estar disponível é a verificação se uma ontologia é consistente. Inclusive, inconsistência é um conceito que necessita de uma definição precisa. Uma possibilidade de base inconsistente é a base trivial, que implica tudo. Em LD, isso significa que uma base é inconsistente se e somente se implica $\top \sqsubseteq \bot$. Outra definição possível para LD é que, dada uma base qualquer, se para qualquer conceito A explicitamente dado no TBox a

base implica $A \sqsubseteq \emptyset$, a base é inconsistente. Porém, seguiremos aqui uma outra notação que classifica este último caso como incoerência.

Exemplificando:

- Mamífero $\sqsubseteq \vdash$BotaOvo
- Mamífero(ornitorrinco)
- BotaOvo(ornitorrinco)

A contradição presente torna a base trivial. Não há interpretação ou modelo que satisfaça a base.

- Mamífero $\sqsubseteq \vdash$BotaOvo
- Batráqueo $\sqsubseteq$ Mamífero
- Batráqueo $\sqsubseteq$ BotaOvo

Esta base não é trivial, mas podemos inferir que o conceito Batráqueo é vazio, indicando um possível erro de modelagem. De acordo com a notação supracitada, esta é uma base consistente mas incoerente.

3 Revisão de Crenças

Em seu livro, Gärdenfors define que mudanças epistêmicas são revisões que ocorrem quando o agente recebe uma nova informação inconsistente com o estado epistêmico atual. Além disso, ao aplicar a revisão, deve-se manter as crenças antigas tantas quantas forem possíveis, o que caracteriza o critério de mudança mínima, isto é, não remover sentenças desncessariamente. Nesse contexto, revisão de crenças consiste no estudo dos estados epistêmicos e sua dinâmica, fornecendo uma representação para os elementos epistêmicos (crenças) e um critério de racionalidade. Em um primeiro plano, utilizaremos um modo simples de modelagem de estados epistêmicos que é a representação por conjuntos de crenças, tendo como base [3].

3.1 Para Refletir...

"To attain knowledge, add things every day. To attain wisdom, remove things every day." Lao Tzu (Tao-te Ching)

3.2 Um Exemplo (Gardenfors & Rott, 1995)

Suponhamos uma situação em que temos um pássaro capturado por meio de uma arapuca.

Crenças:

- O pássaro capturado na arapuca é um cisne
- O pássaro capturado na arapuca vem da Suécia
- A Suécia pertence à Europa
- Todos os cisnes europeus são brancos

Consequência:

- O pássaro capturado na arapuca é branco

Nova informação:

- O pássaro capturado na arapuca é preto

De qual(is) sentenças desistiríamos? Valeria a pena desistir da sentença "A Suécia pertence à Europa"? Seria interessante desistir de todas e deixar nas crenças apenas a nova informação? Nas próximas subseções, abordaremos a teoria que traz embasamento às respostas a essas questões.

3.3 Conjuntos de Crenças

Supomos definida uma lógica L com as ideias de consequência lógica e consistência definidas.

Como variáveis, usaremos letras maiúsculas A, B, C..., além dos símbolos $\top$ para uma tautologia e $\bot$ para contradição.

A atitude epistêmica ligada a conjuntos de crenças define que, para qualquer sentença A, existem 3 possibilidades:

- A é aceita
- A é rejeitada
- A é indeterminada, ou seja, nem aceita nem rejeitada

Definição 1 *Critérios de racionalidade que constituem um conjunto de crenças:*

- O conjunto de sentenças aceitas deve ser consistente
- As consequências lógicas do que é aceito também devem ser aceitas (crenças implícitas)

Definição 2 *Um conjuntos de sentenças K é um conjunto de crenças se e somente se:*

- K for consistente, ou seja, $\bot$ não é uma consequência lógica de K
- K for logicamente fechado, ou seja, se K $\vdash$ B, então B$\in$K

Esta segunda condição é a que envolve o conjunto de todas as consequências lógicas de um conjunto K, ou seja, {A: K $\vdash$ A} - denotado por *Cn(K)*. A definição acima implica que todos os conjuntos de crenças satisfazem a condição que expressa a ideia de equilíbrio dos estados epistêmicos, ou seja, para um conjunto de crenças K, $K = Cn(K)$.

Definição 3 *Para descrever as atitudes epistêmicas possíveis em relação a qualquer sentença* A *dado um conjunto* K *de crenças consistente, podemos expressá-las de 3 maneiras diferentes:*

- $A \in K$: A é aceita
- $\neg A \in K$: A é rejeitada
- $A \notin K$ e $\neg A \notin K$: A é indeterminada

3.4 Expansão, Revisão e Contração

Estes são os três tipos básicos de mudança de crença, ou seja, nesta subseção e nas próximas trataremos das dinâmicas das crenças. Considerando as atitudes epistêmicas definidas na subseção anterior, uma mudança de crença em relação a uma sentença A pode ser definida como mudar de uma das atitudes epistêmicas para uma das outras duas, ou seja:

Expansão: A era indeterminada e agora ou A é aceita ou $\neg A$ é aceita

Contração: A ou $\neg A$ era aceita e agora A é indeterminada

Revisão: A era aceita e passa a ser rejeitada ou $\neg A$ era aceita e passa a ser rejeitada

3.4.1 Expansão

A expansão é a forma de representar, entre as possíveis mudanças epistêmicas, aprender algo novo, ou seja, passar a aceitar uma sentença antes considerada indeterminada, mas sem retirar nenhuma outra crença. Ainda, além de adicionar ao conjunto de crenças a nova sentença, adicionam-se também suas consequências junto com as sentenças previamente aceitas.

Seja K um conjunto de crenças a A uma nova sentença a ser adicionada ao conjunto. A expansão é representada pela função $+$ e satisfaz:

$$K + A = Cn(K \cup A)$$

3.4.2 Revisão

A revisão é necessária em um conjunto de crenças quando uma nova crença, cuja negação era aceita anteriormente, passa a ser aceita. Assim, precisamos revisar o conjunto para manter sua consistência. Porém, diferentemente da expansão, nem todas as crenças antigas são mantidas, ou seja, esta operação é não monotônica.

Seja K um conjunto de crenças e A uma nova sentença a ser incluída, sendo a operação de revisão representada pela função $*$. Segundo o *paradigma AGM*, uma revisão deve satisfazer os seguintes postulados:

(K*1) $K * A$ é um conjunto de crenças

(K*2) $A \in K * A$ - postulado de "sucesso"

(K*3) $K * A \subseteq K + A$

(K*4) Se $\neg A \notin K$, então $K + A \subseteq K * A$

(K*5) $K * A = K \perp$ se e somente se $\vdash \neg A$

(K*6) Se $\vdash A \leftrightarrow B$, então $K * A = K * B$

Estes são os chamados postulados básicos para revisões, isto é, requisitos elementares. Os 2 postulados a seguir são uma generalização dos postulados (K*3) e (K*4) aplicada a mudanças de crença sequenciais (iteradas) e envolvendo a conjunção. Mantendo a mesma simbologia e sendo B também uma nova sentença a ser incluída:

(K*7) $K * (A \wedge B) \subseteq (K * A) + B$

(K*8) Se $\neg B \notin K * A$, então $(K * A) + B \subseteq K * (A \wedge B)$

3.4.3 Contração

Aplica-se a contração em um conjunto de crenças quando se quer desistir de uma crença sem adicionar nada. Porém, muitas vezes a simples retirada de uma crença, digamos A, não resolve o problema, uma vez que as sentenças restantes no conjunto podem implicar A. Nesse caso, para manter o conjunto logicamente fechado, é necessário desistir, também, da(s) sentença(s) que implica(m) A, levando-se em conta sempre o critério de mudança mínima.

Seja K um conjunto de crenças e A uma sentença a ser excluída do mesmo, sendo a operação de contração representada pela função $-$. Segundo o *paradigma AGM*[1], uma revisão deve satisfazer os seguintes postulados:

(K^-1) $K - A$ é um conjunto de crenças

(K^-2) $K - A \subseteq K$ - postulado de inclusão

(K^-3) Se $A \notin K$, então $K - A = K$

(K^-4) Se $\nvdash A$, então $A \notin K - A$ - postulado de sucesso

(K^-5) Se $A \in K$, então $K \subseteq (K - A) + A$ - postulado de recuperação

(K^-6) Se $\vdash A \leftrightarrow B$, então $K - A = K - B$

Assim como exposto para a revisão, estes são os postulados básicos para contração. Os 2 postulados a seguir envolvem conjunção na abordagem da relação entre $(K - A)$ e $K - (A \wedge B)$:

(K^-7) $(K - A) \cap (K - B) \subseteq K - (A \wedge B)$

(K^-8) Se $A \notin K - (A \wedge B)$, então $K - (A \wedge B) \subseteq K - A$

3.5 Relações entre Revisão e Contração

Apesar de apresentarmos revisão e contração como coisas separadas, se relacionarmos ambas operações obtemos um estudo mais simples e mais rico de relações e ligações. Há uma definição de revisão(*) em termos de contrações(-) e expansões(+) chamada de *identidade de Levi*:

Definição 4 *$(K * A) = [(K - \neg A) + A]$*

O Teorema de Levi nos mostra a satisfação dos postulados:

Teorema 1 *Se a função de contração - satisfaz $(K^{-}1)$-$(K^{-}8)$, então a função de revisão * definida acima satisfaz $(K^{*}1)$-$(K^{*}8)$.*

Se essa ideia foi apresentada por Isaac Levi, a ideia análoga foi apresentada por Harper (*identidade de Harper*):

Definição 5 *$(K - A) = K \cup (K * \neg A)$*

Novamente, isso leva ao seguinte teorema:

Teorema 2 *Se a função de revisão * satisfaz $(K^{*}1)$-$(K^{*}8)$, então a função de contração - definida acima satisfaz $(K^{-}1)$-$(K^{-}8)$.*

Com isso, temos que de uma definição podemos obter a outra e tê-las intercambiáveis.

3.6 Funções de Contração

Foram expostos até aqui postulados para *revisão* e *contração*, mas precisamos de algumas informações e ferramentas auxiliares para que construamos funções de contração capazes de satisfazê-los. Vamos primeiro à definição de subconjunto maximal:

Definição 6 *Sendo K um conjunto de crenças, um conjunto de crenças K' é um subconjunto maximal de K que não implica A se e somente se:*

K' é um subconjunto de K ($K' \subseteq K$)

$A \notin K'$

Se $K' \subset K'' \subseteq K$, então $K'' \vdash A$ (isto é, para qualquer sentença B em K que não está em K', $B \rightarrow A$ está em K')

Esse conjunto K' é denotado por $K \perp A$

Outra definição importante é de uma função de seleção:

Definição 7 *Sendo K um conjunto de crenças, tem-se que, para qualquer sentença A, uma função de seleção para K é uma função* φ *tal que se* $K \perp A \neq \emptyset$*, então* $\varphi(K \perp A) \neq \emptyset$ *e* $\varphi(K \perp A) \subseteq K \perp A$

3.6.1 Contração por Partial Meet

Informalmente, obtemos escolhendo alguns elementos de $K \perp A$ e tomando sua intersecção. Formalizando a definição:

Definição 8 *Sendo K um conjunto de crenças, A uma sentença qualquer e* φ *uma função de seleção, definimos a função de contração partial meet como o conjunto* $\bigcap \varphi(K \perp A)$*.*

Qualquer função de contração por *partial meet* deve satisfazer os postulados apresentados:

Teorema 3 *Para qualquer conjunto de crenças K, uma função de contração é partial meet se e somente se satisfizer* (K^-1)*-*(K^-6)*.*

4 Revisão em Lógica de Descrição - Bases de Crenças e Depuração de Ontologias

Vimos, na seção anterior, a forma como a Revisão de Crenças lida com o problema da exclusão de uma sentença ou da chegada de novas informações a uma base de conhecimento e o consequente ocasionamento de inconsistências. Como já destacado, isso é normal, uma vez que as ontologias, especialmente na web, não são estáticas e estão sempre evoluindo. Nesse meio, o objetivo da depuração de ontologias é encontrar os axiomas da base que levaram à inconsistência.

A maior parte da bibliografia sobre Revisão de Crenças está baseada no *paradigma AGM*, teoria que se tornou célebre no famoso artigo [1] e cujo nome deriva das iniciais dos seus autores. No entanto, assim como exposto em [8], os operadores de revisão não podem ser aplicados diretamente às Lógicas de Descrição por não satisfazerem os postulados AGM. Assim, queremos aproveitar esses serviços de depuração para encontrar formas de implementar técnicas de revisão de crenças em Lógicas de Descrição, em especial OWL-DL e Lite (já falamos sobre a indecidibilidade da OWL-Full na seção 3).

Remetendo ao título desta seção, vale também colocar que a partir de agora usaremos uma nomenclatura adotada na literatura que é base de crenças. Na Seção 3, falamos em conjuntos de crenças; porém, dado o fecho lógico, geralmente eles são infinitos, o que dificulta a abordagem computacional. Além disso, ao contrário das bases de crenças, em conjuntos de crenças não há distinção entre conhecimento explícito e conhecimento inferido. Assim, como queremos ir da teoria à implementação, consideraremos revisão em bases de crenças, definidas como bases de conhecimento não necessariamente fechadas.

O objetivo desta seção é apresentar, em termos algorítimicos, as ligações alcançadas entre as operações de depuração em Lógica de Descrição e propriedades formais em revisão de crenças. Os motores de inferência disponíveis para Lógica de Descrição são capazer de verificar a consistência de ontologias, mas isso não é suficiente e repará-las manualmente em caso de inconsistência é muito custoso. Os algoritmos, então, têm a função de encontrar os axiomas relacionados ao problema enfrentado. Não serão apresentados aqui todos os detalhes referentes à demonstração dos algoritmos; para informações mais detalhadas, favor consultar as referências bibliográficas no fim da Parte Técnica.

4.1 Kernel

A operação Kernel computa subconjuntos minimais da base, podendo ser usada para revisão ou contração:

Revisão: uma revisão de B por α tem por objetivo encontrar em $B + \alpha$ todos os subconjuntos minimais inconsistentes, chamado de *kernel* de $B + \alpha$

Contração: uma contração de B por α tem por objetivo encontrar em os subconjuntos minimais de B que implicam α

Feito isso, removemos pelo menos um elemento de cada um dos subconjuntos.

4.2 Partial Meet

A operação Partial Meet, já inicialmente exposta na Seção 3.6.1, computa subconjuntos maximais da base, também podendo ser usada para revisão ou contração:

Revisão: uma revisão de B por α tem por objetivo encontrar em $B+\alpha$ todos os subconjuntos maximais consistentes, chamado de *conjunto resíduo* de $B + \alpha$

Contração: uma contração de B por α tem por objetivo encontrar em os subconjuntos minimais de B que não implicam α

Tendo-os em mãos, podemos escolher alguns, pelo menos um, desses subconjuntos e obter sua intersecção.

4.3 Implementação

Como então encontrar o Kernel ou o conjunto resíduo?

O Kernel pode ser computado usando uma técnica conhecida na literatura como "Black-Box". O algoritmo Black-Box não depende do motor de inferências, isto é, chama-o como subrotina mas sem alterar sua estrutura interna, o que justifica o nome do algoritmo. A estratégia aplicada é conhecida como "Expande-Encolhe".

Para revisão, na expansão, axiomas (sentenças de B) são adicionados a um conjunto B' inicialmente vazio até que esse conjunto fique inconsistente. Depois, ao encolher, remove-se cada elemento de B' separadamente e verifica-se, a cada remoção, se o novo conjunto permanece inconsistente. Em caso

afirmativo, aquele axioma não era crucial para gerar a inconsistência e realmente deve ser removido. Em caso negativo, o axioma retirado deve fazer parte do Kernel e volta ao conjunto. Dessa forma, o conjunto devolvido é um conjunto minimal inconsistente de $B + \alpha$ e, consequentemente, um elemento do Kernel.

BLACK-BOX($B + \alpha$)

```
 1  ▷ Primeira Parte (Expande)
 2  B' ← ∅
 3  for β ∈ B + α
 4        B' ← B' ∪ {β}
 5        if B' é inconsistente
 6              Break
 7  ▷ Segunda Parte (Encolhe)
 8  for γ ∈ B'
 9        if B' \ {γ} é inconsistente
10              B' ← B' \ {γ}
11  return B'
```

Para contração, na expansão axiomas (sentenças de B) são adicionados a um conjunto B' inicialmente vazio até que esse conjunto implique α. Depois, ao encolher, remove-se cada elemento de B' separadamente e verifica-se, a cada remoção, se o novo conjunto continua implicando α. Em caso afirmativo, aquele axioma não era crucial para inferir α e realmente deve ser removido. Em caso negativo, o axioma retirado deve fazer parte do Kernel e volta ao conjunto. Dessa forma, o conjunto devolvido é um conjunto minimal de B que implica α e, consequentemente, um elemento do Kernel.

BLACK-BOX(B, α)

```
 1  ▷ Primeira Parte (Expande)
 2  B' ← ∅
 3  for β ∈ B + α
 4        B' ← B' ∪ {β}
 5        if α ∈ Cn(B')
 6              Break
 7  ▷ Segunda Parte (Encolhe)
 8  for γ ∈ B'
 9        if α ∈ Cn(B' \ {γ})
10              B' ← B' \ {γ}
11  return B'
```

O que garante a corretudo do algoritmo (conjuntos realmente minimais) é a segunda parte, isto é, o "Encolhe". Esta parte poderia ser aplicada diretamente na base B, mas o tamanho de B poderia tornar esta tarefa inviável e é nesse ponto que entra a importância da primeira parte.

Então, o que temos até aqui é um elemento do Kernel, obtido com o algoritmo "Black-Box". No entanto, precisamos de todos os elementos. Para isso, usaremos uma aplicação do algoritmo de Reiter[5]. Tal algoritmo calcula os os cortes mínimos de uma classe de conjuntos (conjunto de conjuntos).

Definição 9 *Um corte em uma classe* $\mathfrak{S}$ *de conjuntos é um conjunto* S *cuja intersecção com cada conjunto da classe* $\mathfrak{S}$ *não é vazia. Um corte* C *é mínimo se e somente se não existe nenhum corte* C' *propriamente contido em* C*.*

Em [10] foi apresentada uma adaptação do algoritmo de Reiter, ideia esta aplicada em Lógica de Descrição e depuração de ontologias por outros autores. Utilizaremos, aqui, a adaptação apresentada em [6], na forma de um algoritmo iterativo equivalente ao algoritmo recursivo apresentado em [7] e [8]. No caso específico de contração:

KERNEL(B, α)

```
1  Corte ← ∅
2  fila ← uma fila vazia
3  S ← BLACK-BOX(B, α)
4  Kernel ← {S}
5  for s ∈ S
6       insira {s} no começo da fila
7  while fila não está vazia
8       Hn ← primeiro elemento da fila
9       remove o primeiro elemento da fila
10      if ∃C ∈ Corte tal que C ⊆ Hn
11           continue
12      elseif α ∈ Cn(B\Hn)
13           S ← BLACK-BOX(B\Hn, α)
14           Kernel ← Kernel ∪ S
15           for s ∈ S
16                insira Hn ∪ {s} na fila
17      else
18                Corte ← Corte ∪ Hn
19      devolva Kernel
```

Como apresentado acima, iniciamos com um elemento do Kernel. Hn recebe um provável corte minimal, mas se algum corte está inteiramente

contido em Hn, este último ou não é minimal ou já havia sido encontrado, o que nos permite desconsiderá-lo e partir para o próximo elemento da fila. Caso contrário, se $B \backslash Hn$ não implicar α, Hn é um corte minimal no Kernel. Porém, se $B \backslash Hn$ implicar α, computa-se o menor subconjunto S de $B \backslash Hn$ que implica α e temos em S um novo elemento do Kernel; ainda nesse caso, sendo s cada elemento de S, temos $Hn \cup \{s\}$ um corte mínimo em potencial e por isso o adicionamos à fila.

Fazendo uma pequena adaptação para o uso no caso de revisão:

KERNEL($B + \alpha$)

```
 1  Corte ← ∅
 2  fila ← uma fila vazia
 3  S ← BLACK-BOX(B, α)
 4  Kernel ← {S}
 5  for s ∈ S
 6       insira {s} no começo da fila
 7  while fila não está vazia
 8       Hn ← primeiro elemento da fila
 9       remove o primeiro elemento da fila
10       if ∃C ∈ Corte tal que C ⊆ Hn
11            continue
12       elseif B + α \ Hn é inconsistente
13            S ← BLACK-BOX(B\Hn)
14            Kernel ← Kernel ∪ S
15            for s ∈ S
16                 insira Hn ∪ {s} na fila
17       else
18                 Corte ← Corte ∪ Hn
19       devolva Kernel
```

Com o Kernel em mãos, o conjunto resíduo pode ser obtido calculando os cortes mínimos do Kernel. Foi demonstrado em [10] que um corte mínimo do Kernel corresponde a B (da notação dos algoritmos acima) menos um elemento do conjunto resíduo e vice-versa. Assim, podemos novamente nos valer do algoritmo de Reiter para essa operação.

Na implementação, detalhada na Seção 6, focou-se na utilização do algoritmo Kernel.

Figura 1: *Logo do editor*

5 Protégé

Nesta seção, será apresentado o software Protégé[3] (já citado anteriormente na Introdução) e sua relação com o trabalho de revisão de crenças em Lógica de Descrição.

Em poucas palavras, o Protégé é um programa editor de ontologias. Utilizando as próprias definições e descrições presentes em seu *website*, além de editor de ontologias ele é um *framework* baseado em conhecimento, sendo o mesmo gratuito e de código aberto. Sua plataforma dá suporte a duas principais maneiras de modelar ontologias, *Protégé-Frames* e *Protégé-OWL*, sendo esta última a utilizada neste trabalho.

Protégé-OWL: uma extensão do programa que dá suporte à linguagem OWL *Web Ontology Language*, que é o mais recente desenvolvimento em linguagens de padronização de ontologias e desenvolvida e promovida pela W3C *(World Wide Web Consortium)*. Esta extensão permite ao usuário carregar, editar e visualizar ontologias, bem como executar motores de inferência sobre as mesmas, ou seja, aplicativos que inferem as consequências lógicas não explícitas na ontologia.

Considere este exemplo simples de um domínio:

1. Aves são Animais voadores
2. Pinguins são Aves
3. Tweety é um Pinguim

A Figura 2 ilustra a modelagem deste domínio no Protégé.

[3] http://protege.stanford.edu/

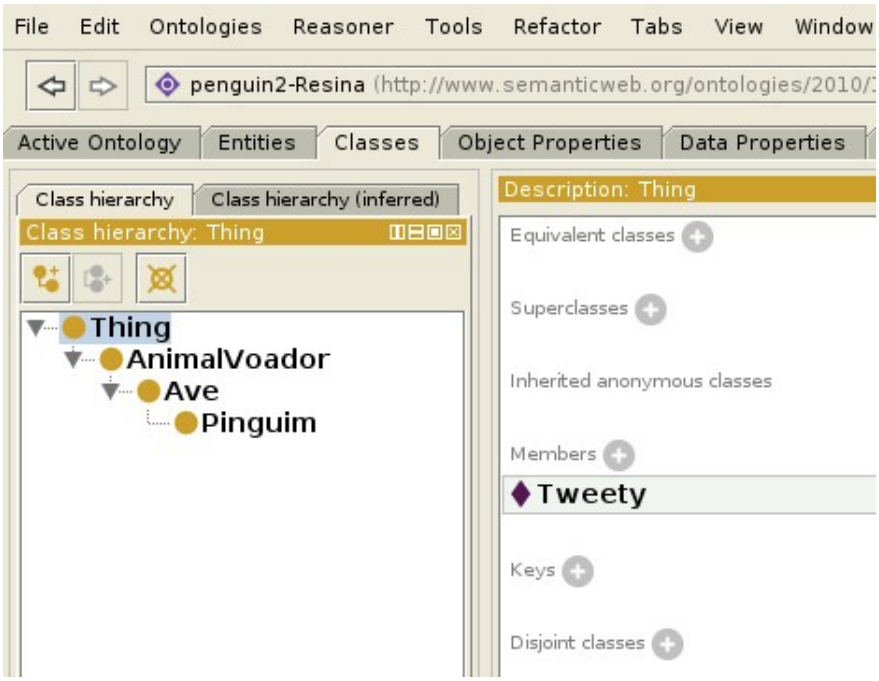

Figura 2: *Ontologia montada no Protégé*

Na mesma figura, é possível ver o menu *Reasoner*, por meio do qual podemos executar um motor de inferências sobre a ontologia que queremos. Dentre os possíveis, optou-se pelo *Pellet*[4], tanto pelo bom desempenho quanto pela boa documentação existente, fatores esses abordados na próxima seção.

Se acrescentarmos um axioma estabelecendo que pinguins não são animais voadores, ou seja, que esses conceitos são disjuntos, provocaremos uma inconsistência devido ao fato de inicialmente pinguim já ser subconceito de animal voador. Se em seguida chamarmos o *Reasoner*, a inconsistência será acusada. Apenas para exemplificar, a Figura 3 mostra a utilização de um outro motor de inferências, o *FaCT++*[5]. Uma vantagem que o *Pellet* já mostra em relação ao *FaCT++* é indicar, quando possível, uma razão para a inconsistência, ao contrário do que pode ser visto na Figura 3.

Vale acrescentar que, se mantivermos esse novo axioma de disjunção mas retirarmos o axioma indicando a asserção de Tweety ao conceito Pinguim, não será acusada nenhuma inconsistência, fato este já considerado na Seção 2 quando fizemos distinção entre inconsistência e incoerência. Neste caso aqui citado, Pinguim é inferido como conceito vazio, mostrado na Figura 4.

Vale relembrar o que foi apontado no início da Seção 4: os motores de

[4]http://clarkparsia.com/pellet/
[5]http:/owl.man.ac.uk/factplusplus/

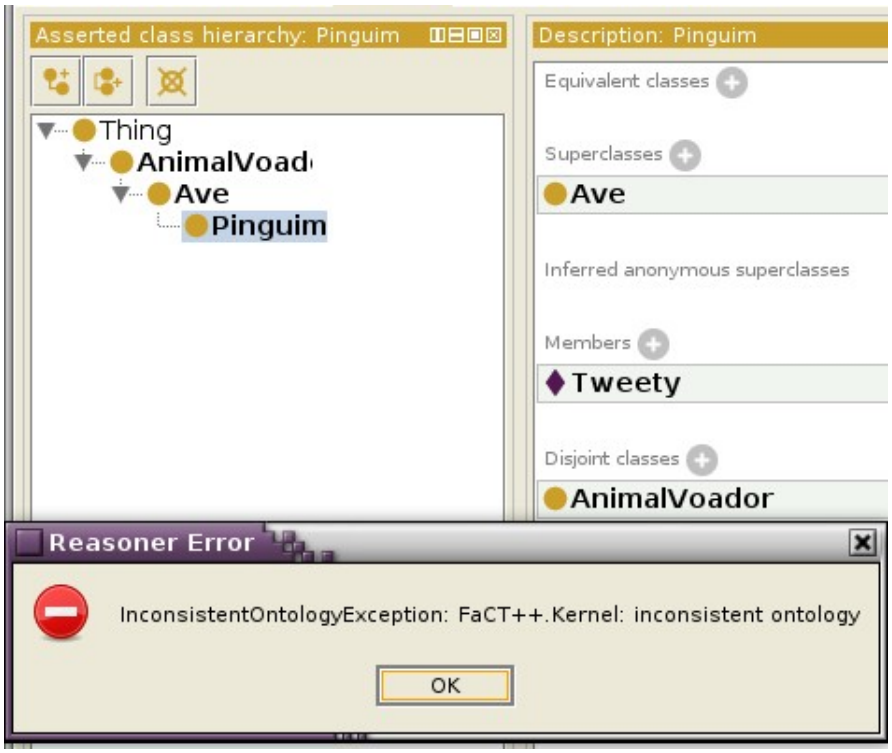

Figura 3: *Reasoner FaCT++ apontando inconsistência na ontologia*

Figura 4: *Classe Pinguim inferida como equivalente a uma classe vazia (Nothing)*

inferência verificam a consistência das ontologias, como exposto aqui, mas isso não é suficiente para revisá-las. É então que se mostra a necessidade de uma ferramenta capaz de operar eficiente e satisfatoriamente sobre essas ontologias a fim de contraí-las ou revisá-las, seguindo os conceitos, princípios e raciocínios vistos nas Seções 3 e 4.

6 Atividades Realizadas e Detalhes da Implementação do Plug-In

Uma significativa parte do tempo dedicado ao Trabalho de Formatura foi investida no estudo da bibliografia e literatura da área pertinentes a este trabalho, cujo resumo foi apresentado nas seções anteriores. Posteriormente, foram realizadas muitas pesquisas em relação à implementação.

6.1 Ambiente de desenvolvimento, API e demais recursos

A última versão do plug-in utiliza a última versão disponível do Protégé, 4.1.0 e *build* 213, lançada em 26 de outubro de 2010. Até as versões 4.0.* do editor não se dava suporte completo às especificações da linguagem OWL 2, recomendação[6] da W3C desde outubro de 2009.

Existe uma *wiki*[7] do Protégé com documentação para desenvolvedores e *links* para os recursos necessários. Foi utilizado o IDE (*Integrated Development Environment* - Ambiente Integrado de Desenvolvimento) Eclipse[8] por meio da integração do Protégé com o mesmo, de acordo com as instruções contidas na *wiki* para criar um *workspace* a partir do repositório do Protégé. Existem também diversos *sites* que explicam como iniciar um plug-in. Um muito útil foi este aqui[9].

O plug-in foi desenvolvido a partir da OWL API[10], uma API em Java e de código aberto por meio da qual podemos criar, manipular e serializar ontologias OWL. Utilizou-se, na última versão do plug-in, a última versão da API, 3.1.0, lançada em 20 de agosto de 2010 e desenvolvida para OWL 2. É importante ressaltar que a versão 3.0.0 foi a primeira versão da API em conformidade com todas as especificações da OWL 2 e tem muitas incompatibilidades com as versões anteriores, lançadas até abril de 2008. No *website* de referência estão disponíveis uma ótima documentação e também exemplos de código no trato de ontologias.

O Protégé 4.1 é baseado nessa terceira geração de versões da API e plug-ins desenvolvidos para o mesmo não serão compatíveis com as versões anteriores, inclusive pelas mudanças da API. O *reasoner* utilizado é o Pellet, também em sua última versão, 2.2.2, de 16 de setembro de 2010.

[6] http://www.w3.org/TR/owl2-syntax/
[7] http://protegewiki.stanford.edu/wiki/Protege4DevDocs
[8] http://www.eclipse.org/
[9] http://www.co-ode.org/downloads/protege-x/plugin-code-example.php
[10] http://owlapi.sourceforge.net/

6.2 O protótipo

O protótipo do plug-in revisor para o Protégé havia sido, inicialmente, desenvolvido pelo Dr. Márcio Moretto Ribeiro, cujas pesquisas estão disponível em sua tese[6]. Ele era orientando de doutorado também da Profª Dra. Renata Wassermann e se envolveu com esse projeto principalmente porque sua área de estudo tratava, também, de revisão de crenças em Lógica de Descrição. No entanto, isso ocorreu em 2007, quando o Protégé estava em sua versão anterior e no *build* 53, sem falar na OWL API, a qual estava na versão 2.0.0, e no Pellet, cuja versão na época não tem mais suporte pela organização *Clark & Parsia*.

Então, além dele não estar completo (por isso chamado aqui de protótipo), estava em desconformidade com as atualizações das ferramentas-chave utilizadas pelo mesmo. No entanto, foi muito importante estudá-lo e avaliar recursos úteis na hora de partir para a implementação, como, por exemplo, o uso do editor da *Manchester OWL Syntax*, referenciado na próxima seção.

6.3 Implementações

Inicialmente, comecei trabalhando com Renato Lundberg, atualmente aluno de mestrado orientado pela Profª Renata. Apesar dele estar trabalhando com Lógica Proposicional, também está interessado em revisão de crenças e na implementação dos algoritmos Kernel e, posteriormente, Partial Meet. O propósito desse projeto em conjunto é montar uma base comum com relação a revisão e contração para poder realizar testes e verificações e depois cada um adaptá-la para seu interesse específico; no caso deste trabalho seria o plug-in. No fim de setembro de 2010 eu diminuí minha participação no projeto para dedicação maior à adaptação do plug-in, mas pretendo voltar a trabalhar no mesmo. Ele está armazenado no repositório do *Google*, acessível para leitura neste link[11].

Abordando especificamente o plug-in, o primeiro passo foi implementar os algoritmos de revisão e contração como expostos na Seção 4. Tendo-os implementados, realizei alguns testes unitários para verificar se os mesmos estavam retornando o resultado desejado. Tendo recebido alguns resultados incorretos e outros casos que entravam em *loop* infinito, concluiu-se que era necessária uma verificação adicional, a da linha 12 do algoritmo Kernel, ao invés de chamar diretamente o algoritmo Black-Box (linha 13) para então verificar se S era vazio, forma esta anteriormente proposta mas já corretamente apresentada em [6].

[11] http://code.google.com/p/bcontractor/

Após essa fase, foi feito todo o ajuste da interface gráfica, a qual está baseada em um conjunto de componentes gráficos reutilizáveis do Protégé e do OWL Editor Kit, expostos aqui[12], e alguns componentes do próprio Java.

Para maiores detalhes com relação às adaptações realizadas ao longo do desenvolvimento com relação às mudanças de versão das ferramentas e da API, favor consultar a Seção 9 na Parte II deste mesmo trabalho.

Quando se baixa o Protégé, há um diretório interno chamado *plugins*. Para que o Protégé reconheça um plug-in adicionado nesse diretório, é necessário que, além das classes da implementação, haja um arquivo XML e um arquivo do tipo *manifest*.

No arquivo XML, definem-se pontos de extensão que, a grosso modo, são funcionalidades. No caso deste plug-in revisor, são necessários dois pontos de extensão, um para a contração e outro para a revisão. Assim como definido no XML, são extensões de *ViewComponent*, entrando no menu *View* do Protégé. Através da tag *class*, especifica-se qual classe executar quando a funcionalidade for acionada pelo usuário.

No arquivo *manifest*, especificam-se, entre outros detalhes, pacotes a serem importados (do *javax*, por exemplo), as bibliotecas de dependência (arquivos *jars* obtidos do Pellet e da OWL API), a versão do Java requisitada na execução e recursos do Protégé necessários.

Com todos esses arquivos em mãos, estamos prontos para comprimí-los em um pacote e executá-lo como um plug-in no editor Protégé.

[12] http://protegewiki.stanford.edu/wiki/P4UiComponentSummary

7 O Plug-In: Resultados Obtidos

O plug-in está na forma de um arquivo comprimido, ".jar", composto por:

lib: diretório que contém as bibliotecas de dependência

META-INF: localização do arquivo *MANIFEST.MF*

revisor: diretório que contém as classes da implementação, estando os algoritmos especificamente localizados em /revisor/ui/algorithms

plugin.xml: arquivo XML já explicado na seção anterior

Para utilizá-lo no Protégé, basta colocar o arquivo *jar* no diretório plugins do programa antes de executá-lo. Dentro do editor, tanto o recurso *Contraction* quando o *Revision* localizam-se no submenu *Misc Views* do menu *View*. Após selecionar um deles, basta clicar sobre a ontologia desejada e a interface gráfica aparecerá. As Figuras 5 e 6 mostram as interfaces de ambos.

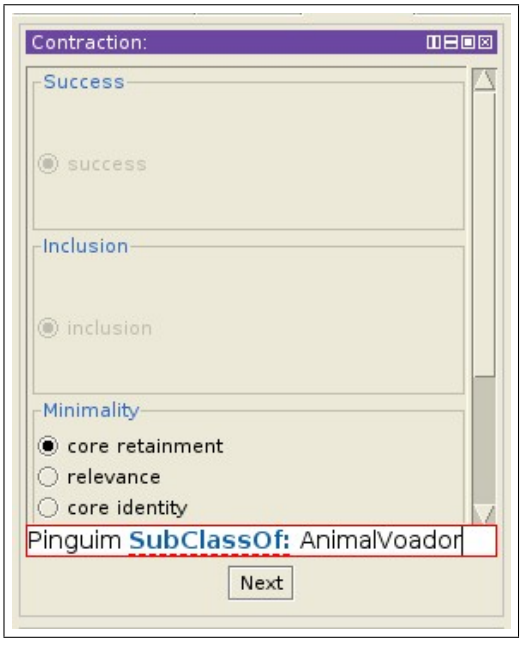

Figura 5: *Interface da ferramenta de contração*

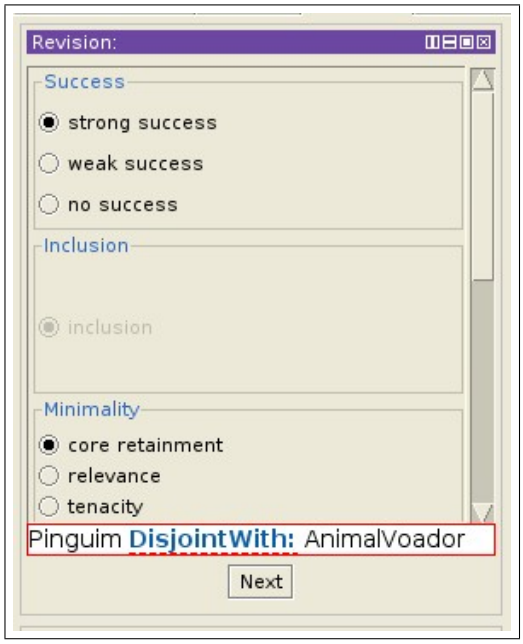

Figura 6: *Interface da ferramenta de revisão*

Testemos agora ambos recursos, baseados na ontologia utilizada na Seção 5. Expressamos os axiomas desejados no editor localizado na parte inferior

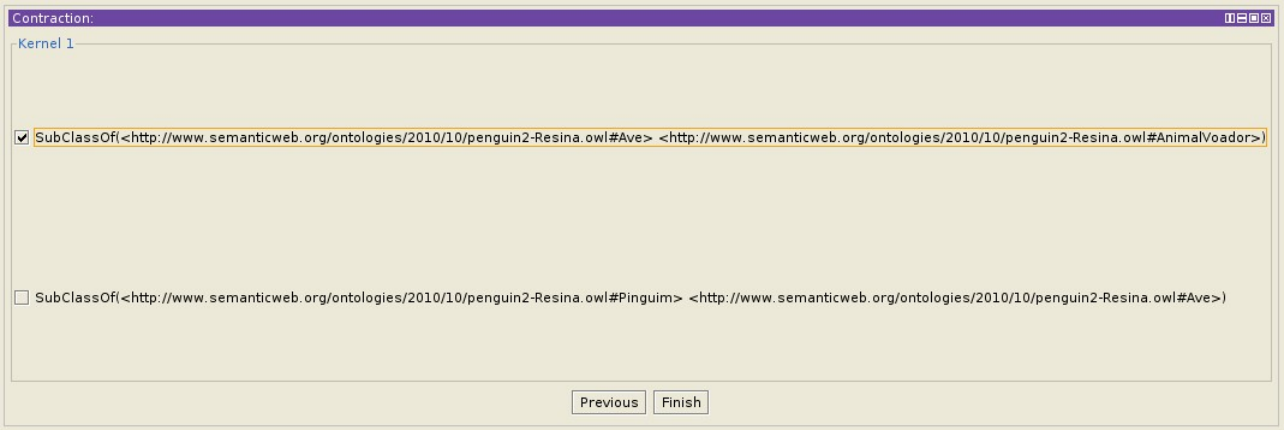

Figura 7: *Kernel da ontologia calculado de acordo com o axioma pedido na contração*

de ambas as interfaces utilizando um recurso chamado *Manchester OWL Syntax*[13], recurso de escrita para descrever classes OWL. Nas Figuras 5 e 6 já há sentenças escritas.

Para a Contração, suponhamos que desejamos desistir da crença que pinguins são animais voadores. Indicamos então este axioma pela sentença "Pinguim SubClassOf: AnimalVoador" e clicamos em *Next*. O programa devolve-nos, então, o Kernel computado para o caso, ilustrado na Figura 7. Nesse caso, só há um subconjunto minimal que implica a sentença escolhida, formado pelos axiomas "Pinguim SubClassOf: Ave" e "Ave SubClassOf: AnimalVoador". Selecionamos, como visto na figura, o(s) axioma(s) desejado(s) e clicamos em *Finish*. Nesse caso, para qualquer um dos dois que retirarmos, a ontologia deixa de implicar "Pinguim SubClassOf: AnimalVoador". A ontologia é contraída de acordo, ilustrada na Figura 8.

Para a Revisão, suponhamos que desejamos revisar a ontologia pela crença que pinguins não são animais voadores, ou seja, esses conceitos são disjuntos. Indicamos este axioma pela sentença "Pinguim DisjointWith: AnimalVoador" e clicamos em *Next*. O programa devolve-nos o Kernel computado para o caso, ilustrado na Figura 9. Nesse caso, só há um subconjunto minimal inconsistente com a sentença escolhida, formado pelos axiomas "Pinguim SubClassOf: Ave" e "Ave SubClassOf: AnimalVoador" e "Tweety types: Pinguim" (asserção). Selecionamos, como visto na figura, o(s) axioma(s) desejado(s) e clicamos em *Finish*. Nesse caso, para qualquer um dos três que retirarmos, a ontologia passa a implicar "Pinguim DisjointWith: AnimalVoador" e permanece consistente. A ontologia é revisada de acordo, ilustrada na Figura

[13]http://www.co-ode.org/downloads/manchesterowlsyntaxeditor/

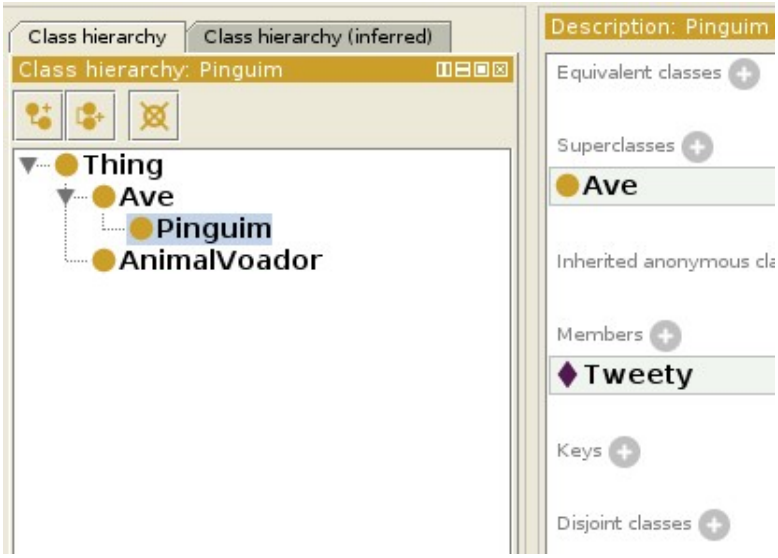

Figura 8: *Resultado da ontologia contraída de acordo com a escolha do usuário*

10.

Uma observação importante a ser feita é com relação ao editor da *Manchester OWL Syntax*. Mostramos, nas sentenças escritas nesta seção, com o sinal de dois-pontos após as palavras-chave, tal como "SubClassOf:". No entanto, é possível observar nas Figuras 5 e 6 um sublinhado vermelho sob as palavras-chave, indicando um erro e alertando para o uso das mesmas sem o sinal de pontuação; somente assim o editor é considerado como *bem formado*. Porém, dentro da implementação da *Manchester OWL Syntax* da API, o interpretador da sentença lança uma exceção de erro se a utilizarmos sem o sinal de dois-pontos, sugerindo a utilização do sinal. A solução dada a esse *bug*, pelo menos temporariamente, foi não verificar, no código, se o editor está bem formado como condição para interpretar a senteça, permitindo, assim, o uso da sentença com dois-pontos no editor do plug-in, apesar do aparente erro. O uso de uma sentença correta fica, por enquanto, a cargo do usuário.

Faltam alguns testes referentes à aplicação de outros postulados possíveis indicados na teoria e a comparações com o uso de conjuntos resíduos, apesar do plug-in já mostrá-los, como indicação de recursos a serem acrescentados. Com relação a trabalhos futuros, favor verificar a Seção 12 da Parte II.

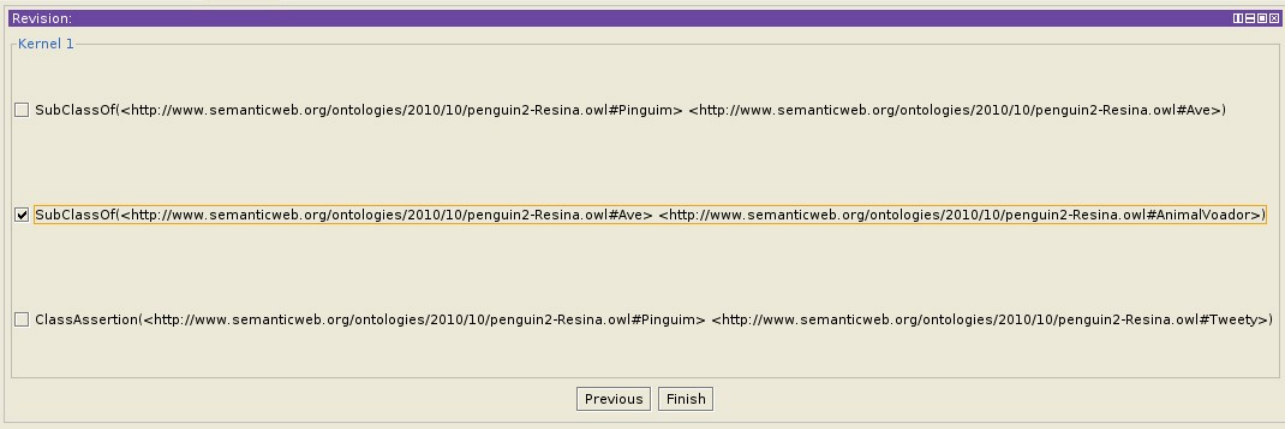

Figura 9: *Kernel da ontologia calculado de acordo com o axioma pedido na revisão*

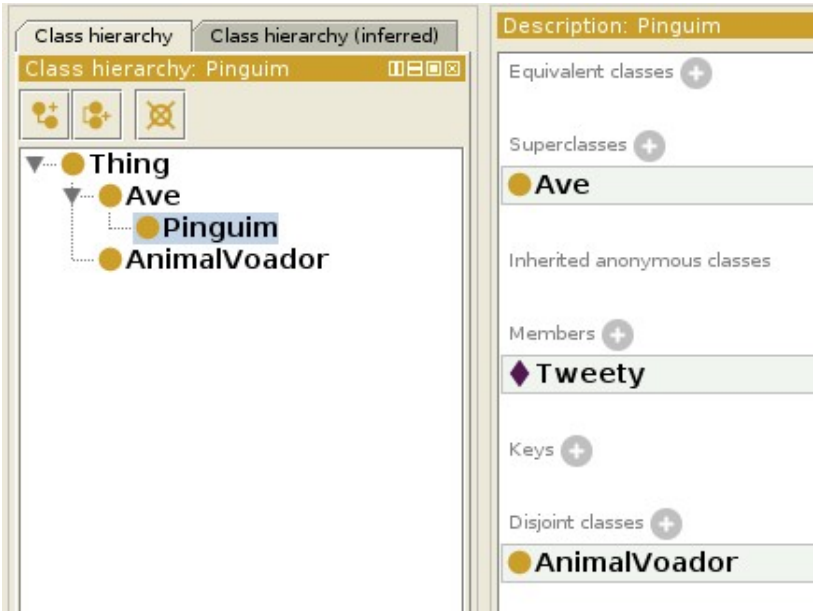

Figura 10: *Resultado da ontologia revisada de acordo com a escolha do usuário*

8 Conclusão

O Protégé é um editor que tem sido cada vez mais usado por parte daqueles que desejam modelar domínios e definir ontologias, inclusive em áreas de destaque como a Medicina, visto o aumento do desenvolvimento de ontologias médicas, muito grandes em sua maior parte. A existência, então, de uma ferramenta capaz de indicar as possíveis melhores formas para revisar ou contrair uma ontologia, e não apenas se ela é inconsistente ou não, é um grande avanço. Apesar de haver muitas referências teóricas relativas a esta área, ainda falta muito a ser implementado e este plug-in é uma considerável contribuição. A parte da proposta referente ao desenvolvimento do plug-in foi alcançada e, apesar de haver mais testes a serem feitos, os resultados obtidos foram satisfatórios.

Algo importante notado durante o desenvolvimento é que um algoritmo proposto na literatura pode carecer de um detalhe ou conter um pequeno equívoco e, às vezes, isso só será percebido ao implementá-lo, como descrito na Seção 6.3. Está aí uma importante associação entre teoria e prática no âmbito de testes e, também, avaliação de desempenho.

Ainda sobre a relação entre teoria e prática, é possível ver que, apesar da profunda pesquisa teórica ainda hoje voltada para a área de representação de conhecimento, há muitas aplicações práticas para as teorias existentes, principalmente na área de ontologias, e, quanto mais aplicações são desenvolvidas, mais recursos e bases teóricos são necessários, além da percepção de novas necessidades. Portanto, a área de lógica e representação de conhecimento ainda tem bastante a conquistar.

Referências

[1] C. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change. *Journal of Symbolic Logic*, 50, 510–530, 1985.

[2] D. Nardi, R. J. Brachman. An Introduction to Description Logics. In the Description Logic Handbook, edited by F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider. *Cambridge University Press*, 2002, pages 5-44.

[3] Peter Gardenfors. Knowledge in Flux - Modeling the Dynamics of Epstemic States. *MIT Press*, 1988.

[4] I. Horrocks and P. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In Proceedings of the 2nd International Semantic Web Conference (ISWC), pp. 17–29. Springer,New York, 2003.

[5] R. Reiter. A theory of diagnosis from first principles. Artificial Intelligence, 32(1):57–95, 1987.

[6] Márcio Moretto Ribeiro. Revisão de Crenças em Lógicas de Descrição e em Outras Lógicas não Clássicas. PhD thesis, Universidade de São Paulo, 2010.

[7] Márcio Moretto Ribeiro and Renata Wassermann. The Ontology Revisor Plug-In for Protégé. In *Proceedings of the Third Workshop on Ontologies and their Applications (WONTO 2008)*, 2008.

[8] Márcio Moretto Ribeiro and Renata Wassermann. Base Revision for Ontology Debugging. *Journal of Logic and Computation*, Vol. 19, No. 5, 721-743, 2008.

[9] Renata Wassermann. *Resource-Bounded Belief Revision*. PhD thesis, Universiteit van Amsterdan, 1999.

[10] Renata Wassermann. An algorithm for belief revision. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR'00), pages 345–352, Breckenridge, Colorado, USA, Abril, 15-20 2000. Morgan Kaufmann.

Parte II
Parte Subjetiva

9 O Início de Tudo

O campo de Lógica e Inteligência Artificial é extremamente vasto. Porém, no curso de Bacharelado em Ciência da Computação, por conta da existência de muitas áreas, a única disciplina obrigatória que realmente inicia o estudo de Lógica é a disciplina MAC 239, Métodos Formais em Programação, na qual estudamos Lógica Proposicional, Lógica de Primeira Ordem e um pouco de Métodos Formais para prova de corretude de pequenos programas. Essa disciplina, cursada por mim no segundo semestre de 2008, chamou-me muito à atenção e eu comecei a me interessar pela área.

A professora que ministrou a disciplina foi minha própria orientadora, a Profª Renata Wassermann, e, no final do semestre, fui conversar com ela sobre como funcionam as pesquisas e a Iniciação Científica nessa área. Ela disse que outros dois alunos também tinham perguntado-lhe sobre isso e que no começo do ano seguinte ela conversaria conosco. Após o início das aulas, no primeiro semestre de 2009, a professora iniciou um grupo de estudos sobre Lógica, composto por mim, outros dois colegas da minha turma e um aluno do mestrado. Ela começou a apresentar as outras áreas da Lógica não vistas no curso e passar materiais para ler e estar mais ciente do assunto. Depois de um tempo, ela apresentou os projetos abertos e em andamento e nós deveríamos escolher algum daqueles tópicos ou tentar outra área.

Desde o início eu havia pensado e comentado que queria algo ligado à Lógica mas que envolvesse programação. Escolhi a área de lógica de descrição e logo a Profª Renata pensou que eu poderia trabalhar associado ao Márcio Ribeiro, seu orientando de doutorado na época, pois ele estava envolvido com revisão de crença em lógica de descrição e eu teria que estudar sobre o assunto para poder, posteriormente, entrar com a parte de desenvolvimento. Em julho e agosto fui conhecendo o Protégé e então, no segundo semestre de 2009, cursei a disciplina MAC444 (Sistemas Baseados em Conhecimento), na qual vimos justamente lógica de descrição e fizemos um trabalho no Protégé.

Comecei a conversar com o Márcio (havia acabado de voltar da Europa e era o monitor da disciplina) sobre o projeto mas não pude avançar muito devido à grande carga das disciplinas no semestre; porém, foi o suficiente para ver que era realmente com aquilo que eu queria trabalhar. No final de setembro outro ponto colocou em dúvida a possibilidade de ingressar em um projeto de Iniciação Científica: a possiblidade de cursar o primeiro semestre

de 2010 em *King's College London*, Inglaterra, pelo CCInt da USP. Participei de todo o processo seletivo até o fim, mas não fui escolhido para uma das 5 vagas, isso já no final de outubro. Com o resultado, a Profª Renata disse: "Então vamos trabalhar!".

Daí, no início deste ano de 2010, pude realmente estudar tópicos mais específicos do assunto, discutir algoritmos com o Márcio e pensar em implementações interessantes para o problema; desde o começo já estava ciente do alto custo computacional na área de Lógica e Inteligência Artificial. Foi assim que tudo começou...

10 Desafios e Frustrações

No início, entrar nesse projeto de Lógica de Descrição e o desenvolvimento de um plug-in para o Protégé já foi um desafio. Nem o Protégé eu conhecia e, apesar de eu apreciar cada vez mais a área à medida que estudava, era algo incerto. No entanto, isso tornou-se, para mim, um grande incentivo, pois era justamente o que eu queria: envolver-me com implementação numa área que eu gostasse.

Outra pergunta que nunca saía da minha cabeça era se eu conseguiria terminar uma implementação relevante do plug-in a tempo para este Trabalho de Formatura. No palestra inicial da disciplina, o Profº Carlos Eduardo Ferreira me disse algo muito interessante: "É sempre um desafio; é como em uma empresa, você tem um projeto e procura cumprí-lo no prazo proposto." Tomei isso como um aprendizado e um desafio pessoal. Na época que esta dúvida mais me incomodou, foi minha orientadora, a Profª Renata Wassermann, quem me indicou a "luz": "Apesar de propormos um *software* e procurarmos cumprir com isso, o Trabalho de Formatura não se resume a pura implementação. Há muito estudo por trás e tudo que se aprende na caminhada é importante, isto é, muitas vezes encontramos problemas não previstos, sejam eles de qualquer natureza, e saber lidar com eles e, ainda mais, aprender com eles, é o diferencial."

Considero que o maior problema enfrentado foi com relação às versões. Como mostrado na Seção 6, as últimas versões do Protégé, da OWL API e do Pellet foram liberadas poucos meses antes da finalização deste trabalho. Como ele já estava em andamento, dei preferência à utilização da versão anterior para finalização do projeto proposto e, então, passar às atualizações da última versão.

No entanto, ainda não se tinha trabalhado muito com o *reasoner*. Quando o mesmo foi aplicado, diversas incompatibilidades surgiram. A principal era um erro no momento de instanciar o *reasoner*. Apesar de seguir exatamente

as instruções apresentadas no FAQ[14] do *website* do Pellet, o código não era aceito. Foi então que percebi a profundidade da atualização da API: além da alteração nos nomes dos pacotes importados, muitas classes tiveram seus nomes alterados para alinhar a interface com a especificação estrutural da OWL 2. O Pellet já estava associado a essa especificação e, mesmo para classes iguais nas duas versões da API, um objeto não era compatível com o outro por conta das interfaces.

Decidi, então, alterar para a última versão da API. Foi necessária uma refatoração em praticamente todos os arquivos por conta dos nomes dos pacotes e classes. Naturalmente, por fatores já explicados, os recursos dos pacotes relacionados ao Protégé também se tornaram incompatíveis. Fui, portanto, levado a atualizar, também, a versão do Protégé que estava usando.

No entanto, o Pellet começou a mostrar-se incompatível mesmo utilizando as últimas versões dos demais recursos. Sem saber como proceder, tentei começar a usar, no lugar dele, o *reasoner* HermiT[15]. No entanto, além de haver uma documentação ínfima, ele apresentou problemas quanto ao relacionamento com a ontologia à qual ele estava ligada, perdendo, muitas vezes, alterações nela. Procurando solucionar este problema encontrei a solução para o problema do Pellet: a atualização dele foi muito grande - os nomes dos pacotes foram totalmente alterados e o modo de instanciá-lo também; apesar disso, a seção FAQ do Pellet (referenciada acima) continua ensinando a usá-lo com a OWL API do modo antigo.

Na atualização da API, outro problema foi o sumiço de alguns métodos antes utilizados por mim. Recursos facilmente aplicados agora pareciam sem saída. A solução foi, então, buscar no código-fonte das versões antigas da API como o método anteriormente usado era desenvolvido, para então desenvolver uma adaptação do mesmo dentro do código do plug-in. Em um dos casos, que foi na utilização da *Manchester OWL Syntax*, a mudança foi de uma linha de código para seis, mas o importante foi a resolução do caso.

Apesar dos sustos que levei cada vez que mudava os recursos para uma versão mais nova e mais incompatibilidades surgiam, aprendi muito com isso, inclusive conhecimentos mais profundos sobre a estrutura da OWL API. Ter um programa implementado e saber avaliar se vale a pena atualizar a versão de um componente utilizado é uma importante habilidade no desenvolvimento de *softwares*.

[14] http://clarkparsia.com/pellet/faq/owl-api/
[15] http://hermit-reasoner.com/

11 Disciplinas no BCC mais Relevantes

Apesar de todas as disciplinas serem importantes na formação geral das diversas competências da área de computação, as que considero mais relevantes são:

MAC110 - Introdução à Programação, MAC122 - Princípios de Desenvolvimento de Algoritmos e MAC323 - Estrutura de Dados: disciplinas do curso introdutórias de programação, algoritmos e estrutura de dados que aumentaram minha paixão por computação e desenvolveram grandemente meu gosto por programação.

MAC239 - Métodos Formais em Programação: disciplina de Lógica responsável por despertar meu interesse pela área.

MAC444 - Sistemas Baseados em Conhecimento: disciplina optativa que cursei justamente por ser totalmente ligada ao meu trabalho, pois tratava de Lógica de Descrição e também envolveu o uso do editor Protégé.

MAC342 - Laboratório de Programação Extrema: nesta disciplina aprendi a lidar com projetos de porte maior (em relação aos visto anteriormente no curso), a utilizar recursos da IDE Eclipse até então desconhecidos por mim e iniciei o uso de testes unitários.

12 Trabalhos Futuros

O principal trabalho futuro será a finalização do desenvolvimento do plug-in no projeto de Iniciação Científica. Essa finalização envolve a aplicação e os testes de outros postulados possíveis (em relação a revisão e contração) presentes na bibliografia da área e a implementação e teste dos algoritmos utilizando conjuntos resíduos, para assim verificar se um dos métodos (por Kernel ou por Partial Meet) é sempre mais eficiente ou se depende do caso. Dessa forma, saberemos se vale a pena implementar das duas formas ou se é melhor computar o conjunto resíduo pelo algoritmo de Reiter a partir do Kernel, como exposto na Parte I.

Particularmente, continuarei este projeto em Iniciação Científica e pretendo avançar na área através da Pós-Graduação.