O3F Ontology Representation Framework

O3 Model Description

(*part in Portuguese; part in English*)

Versão 4

Data inicial desta versão: 2010-01-27

Data da última alteração desta versão: 2016-04-13

Conteúdo

1 Principais diferenças em relação à versão anterior

Esta secção descreve sumariamente as diferenças principais desta versão (iniciada a 2010-01-27) em relação à versão de 2009-07-14.

As diferenças introduzidas nesta versão foram originadas essencialmente num conjunto de sugestões e decisões tomadas na reunião do grupo O3F (Luís Botelho, Jairo Avelar e Carlos Correia) que decorreu 2010-01-27. Esta reunião teve por base a resolução de problemas relativos quer ao próprio *O3 Model*, quer às teses de mestrado do Jairo e do Carlos.

Na versão anterior, a aplicação de facetas como *Minimum_value*, *Maximum_value*, *Minimum_size* e *Maximum_size* a funções ou a métodos funcionais gerava uma ambiguidade. A intenção da aplicação de facetas desse tipo a funções (e métodos funcionais) era a de caracterizar o valor retornado por essas funções (métodos funcionais). No entanto, o *O3 Model* enfatiza o facto de que funções, métodos funcionais e atributos denotarem conjuntos de túpulos formados por elementos do domínio e pelos elementos correspondentes do contradomínio. Assim, a aplicação directa de uma faceta a um função, a um método funcional, ou a um atributo caracterizaria um conjunto de túpulos, o que não coincidiria com a intenção de caracterizar o valor da função, método funcional ou atributo. Na versão anterior da linguagem, foi proposto o operador *Range/1* que, aplicado a uma função, denota o seu contradomínio. As facetas referidas poderiam assim ser aplicadas ao contradomínio da função o que já coincidia com a intenção original. Criou-se também o operador *Domain/1* que denotava o domínio da função a que se aplica. A utilização de *Range/1* e de *Domain/1* acarretou subtilezas formais (descritas na versão 2009-07-14) que conduziram à introdução do operador *Eval/1* usado para forçar a avaliação de expressões.

Argumentos de natureza diferente conduziram, nesta nova versão, a uma solução diferente para os problemas descritos:

- Denotando a função F um conjunto de pares ordenados, não seria nada óbvio falar do contradomínio de F – um conjunto não tem contradomínio;

- Quando se insere, na base de dados onde as ontologias estão armazenadas, o facto de que uma faceta está aplicada a uma entidade da ontologia, essa entidade tem de existir na base de dados. Mas, enquanto que a função F pode ser uma entidade de uma ontologia, Range(F) não é (i.e., o modelo não tem a entidade contradomínio de uma função); e

- Em termos de utilização da linguagem por pessoas com background insuficiente, seria difícil perceber todas estas subtilezas.

A principal diferença desta versão em relação à versão de 2009-07-14 é a aplicação da faceta *Minimum_value* e outras da sua família a funções, métodos funcionais e atributos, no sentido de caracterizar os seus valores e não os conjuntos denotados.

A perspectiva adoptada tem dois componentes. Em primeiro lugar, acrescentou-se semântica às facetas *Minimum_value*, *Maximum_value*, *Minimum_size*, *Maximum_size* e outras de modo que a sua aplicação a uma função, método funcional ou atributo tenha um significado coincidente com a intenção original. Estas facetas aplicam-se apenas a funções, métodos funcionais e atributos e caracterizam os seus contradomínios. Em segundo lugar, criaram-se as facetas *Smallest_instance*, *Largest_instance*, *Instance_maximum_size* e *Instance_minimum_size*, entre outras, que se aplicam a quaisquer conjuntos, incluindo os denotados por funções, métodos funcionais e atributos.

Com esta nova perspectiva, quando a faceta *Minimum_value* se aplica a uma função ou a um método funcional, ela caracteriza o valor retornado pela função ou pelo método (i.e., o seu contradomínio). Quando se aplica a um atributo, ela caracteriza o valor do atributo. Se pretendêssemos caracterizar os túpulos do conjunto denotado, por exemplo, por um atributo de uma classe, usar-se-ia a faceta *Largest_instance* ou *Smallest_instance*.

Por exemplo, se pretendermos dizer que o menor valor de uma idade é 0, podemos usar a seguinte proposição, assumindo que Idade é uma função que devolve uma idade:

EntityFacet(Idade, Minimum_value, 0).

Consideremos a função Ordem que aplica o conjunto {a, e, i, o, u} no conjunto {1, 2, 3, 4, 5}, estabelecendo o número de ordem das vogais. Podemos dizer que o menor dos túpluos do conjunto denotado pela função Ordem é <a, 1>, usando a seguinte proposição:

EntityFacet(Ordem, Smallest_instance, Sequence(a, 1)).

Ao contrário do que acontecia com a versão de 2009-07-14, esta solução não obriga a assumir que as expressões da linguagem não são avaliadas a não ser que essa avaliação seja forçada. Pareceria ter deixado de surgir a necessidade da utilização do operador *Eval/1*.

No entanto, surgiu agora (2010/08/23) uma nova razão relativa à avaliação de expressões funcionais. Esta razão prende-se com a necessidade de dizer que os papeis desempenhados por dois argumentos de dois operadores diferentes são iguais. Teria de se usar um axioma como por exemplo

Op1.ARGi = Op2.ARGj

Mas o axioma anterior significaria que os valores dos dois argumentos são iguais.

Talvez o problema possa ser resolvido com o operador Quote/1 que evita que a expressão seja avaliada. Neste caso, bastaria escrever Quote(Op1.ARGi) = Quote(Op2.ARGj).

Por enquanto, o operador *Eval/1*, e os operadores *Range/1* e *Domain/1* permanecem na linguagem. Cria-se o operador Quote/1.

A outra alteração mais significativa desta nova versão (2010-01-27) em relação à versão de 2009-07-14 é a representação das relações de dependência. A versão anterior não permitia a representação de relações de dependência. Nesta nova versão, podem especificar-se dependências entre entidades de qualquer natureza. As relações de dependência são captadas pelo predicado *Dependency/1* que declara a existência de uma relação de dependência, pelo predicado *DependencyArgument/3* que se usa para especificar as entidades relacionadas na relação de dependência. Finalmente, usa-se a faceta *Dependence* para indicar quais das entidades relacionadas numa relação de dependência são dependentes e quais as independentes. Para isso, foi necessário acrescentar o valor "Dependent" à faceta *Dependence*.

Nesta nova versão (2010-01-27), foram incluídos dois dispositivos relativos à definição de facetas novas. O predicado *ValidFacetElement/2* usa-se para especificar os elementos a que uma faceta se pode aplicar. *ValidFacetType/2* especifica os valores válidos de uma faceta.

Finalmente, na versão anterior (2009-07-14), havia as duas facetas *Navigable* e *NonNavigable*, as quais eram aplicadas a relacionamentos (associações e relações de dependência) para indicar os argumentos desse relacionamento que podem ser acedidos a partir dos outros (*Navigable*) e os argumentos que não podem ser acedidos a partir dos outros (*NonNavigable)*. Na nova versão (2010-01-27), em vez das duas facetas *Navigable* e *NonNavigable* aplicadas a relacionamentos, passou a haver a faceta *Navigability*, aplicada aos argumentos de um relacionamento, com valores *Navigable*, *Non_navigable* e *Non_specified*.

2 *O3 Model*: Modelo O3F de Representação de Ontologias

O modelo de representação de ontologias usado nestes apontamentos chama-se *O3 Model*, foi originalmente definido em [Mota et al 2003] e posteriormente modificado e melhorado ao longo do tempo. Trata-se de um modelo em que os conceitos do domínio são descritos através de objectos, classes, atributos, e métodos, mas também de funções, ações e predicados, entre outros tipos de entidade.

O *O3 Model* inclui diversos conceitos típicos dos modelos de objectos, entre os quais, classes, associações, atributos, e métodos. Um dos conceitos mais importantes do *O3 Model*, o qual é usado numa grande variedade de definições, é o tipo de dados. Este conceito é representado, no *O3 Model*, pela classe *Type*. Importa salientar que, no âmbito do O3F, um tipo é um conjunto. Embora não seja invulgar noutros contextos, esta realidade nem sempre é realçada. Por exemplo, o tipo *Integer* é o conjunto dos inteiros, o

tipo *Char* é o conjunto dos caracteres e *Pessoa* (muito provavelmente, uma classe), é igualmente um conjunto. Inteiros, naturais, caracteres e outros tipos básicos integram-se na classe *Scalar* do diagrama de classes que descreve *O3 Model*. As classes são representadas pela classe *Class* do diagrama.

Talvez menos evidente seja o facto de que as associações, as quais relacionam classes e/ou outras associações, são igualmente conjuntos, ou seja, também são tipos. Por exemplo, a associação binária *CarroDaPessoa* (relaciona um carro com as pessoas que o detêm e relaciona uma pessoa com os seus carros) é o conjunto dos conjuntos {carro = Carro, dono = Pessoa} tal que *Pessoa* é dona de *Carro*. A classe *Association* do diagrama de classes representa as associações.

O *O3 Model* contempla também predicados, os quais são relações com algumas características diferentes das características das associações. Os predicados são representados, no *O3 Model*, pela classe *Predicate*. Um predicado pode relacionar um número indefinido de argumentos, os quais podem ter qualquer tipo de dados, enquanto que as associações podem apenas relacionar classes e/ou associações. Ao contrário das associações, os predicados não podem ter nem atributos nem métodos. Os predicados representam igualmente conjuntos, por isso são igualmente tipos. Por exemplo, o predicado *Distancia*, o qual relaciona duas localizações num dado espaço e a distância entre elas, representa o conjunto dos conjuntos {localizacao1 = Loc-1, localização2 = Loc-2, distância = Distância} tal que *Distância* é o valor da distância entre as localizações *Loc-1* e *Loc-2*.

O *O3 Model* permite também a modelação de funções, as quais são captadas pela classe *Function* do diagrama de classes. Uma função representa igualmente um conjunto, ou seja, uma função é também um tipo. Por exemplo, a função *Capital*, que aplicada a um país denota a sua capital, representa o conjunto de pares {país = País, capital = Capital} tal que o segundo elemento do par é a capital do primeiro elemento.

Há um último tipo de dados no *O3 Model*, o qual surge devido à possibilidade de representar ações no modelo. Ao contrário das associações, dos predicados, e das funções, as ações são mais do que tipos de dados. O aspecto mais importante de uma ação é o efeito que produz no mundo. Quando uma ação é executada, o mundo sofre uma alteração. Por exemplo, se a ação abrir a porta for executada, aplicada a uma porta particular, essa porta deixa de estar fechada e passa a estar aberta. O mundo altera-se. Associações, predicados e funções não causam qualquer alteração no estado do mundo. Mas, tal como as associações, os predicados, e as funções, as ações têm igualmente argumentos e, por vezes, também devolvem valores. Diremos que as ações têm dois aspectos: um aspecto procedimental e um aspecto declarativo. O aspecto procedimental diz respeito aos efeitos da ação sobre o mundo, as alterações que a execução da ação causa no estado do mundo. O aspecto declarativo diz respeito aos seus argumentos, aos seus tipos de dados e, no caso de haver, aos valores de retorno. Por agora, esqueceremos o aspeto procedimental das ações. Ao aspecto declarativo de uma ação chamaremos interface da ação (a que também poderíamos chamar assinatura da ação). As interfaces de ação estão representadas, no diagrama de classes, através da associação entre a classe *Action* e a classe *ActionInterface*, das suas particularizações (*RelationalActionInterface* e *FunctionalActionInterface*), as quais são por sua vez particularizações de *Predicate* e de *Function*. Se uma ação não devolver nada, diz-se que tem uma interface relacional. Se, pelo contrário, a ação devolver um valor, diz-se que tem uma interface funcional. Para todos os efeitos, uma interface relacional é um predicado e uma interface funcional é uma função. Tratando-se de um predicado ou de uma função, uma interface de ação define igualmente um tipo de dados. Para melhor clarificar a ideia, vamos supor a existência da ação *ComprarBilhete* de um serviço de aquisição *on-line* de bilhetes de cinema que tem, como argumentos, o cinema, o filme pretendido, o horário a que o filme é exibido, e o número de contribuinte do utilizador. Quando a ação *ComprarBilhete* é executada, o bilhete fica reservado para o utilizador e a ação devolve um número de reserva que pode ser usado quando o utilizador levantar os bilhetes no cinema. Dado que há um valor de retorno, *ComprarBilhete* tem uma interface funcional. Esta interface funcional aplicada a um cinema, um filme, um horário, e um número de contribuinte denota o número de reserva correspondente. Dito de outra forma, a interface funcional da ação *ComparBilhete* representa um conjunto de conjuntos {cinema = Cinema, filme = Filme, horário = Horario, cliente = NIF, num_reserva = Reserva}, ou seja, define um tipo de dados cujos membros são conjuntos com essa composição.

Os tipos de dados, no *O3 Model*, podem organizar-se em hierarquias. Isto significa que podemos estabelecer, por exemplo, hierarquias de classes e de tipos básicos (como é habitual), mas também hierarquias de funções, e de predicados, o que não é assim tão habitual. Embora não esteja explicitamente captado no diagrama de classes, as ações também se podem hierarquizar através do estabelecimento de uma hierarquia das interfaces de ação. Imaginemos que temos uma função que devolve o bilhete de identidade de uma pessoa passada como argumento, e outra função que devolve o número de identificação fiscal da pessoa. Ambas são casos particulares da função que devolve a identificação da pessoa. Embora pouco habitual, o conceito de hierarquia de funções, de predicados, de associações e de

ações é uma ferramenta de modelação extremamente poderosa que permite representar mais informação sobre o domínio que é modelado. Isto é captado pela classe *Hierarchy* e pelas suas associações com a classe *Type*, no diagrama de classes do *O3 Model*. Além do grande poder de expressão que as hierarquias de tipos conferem ao modelo, esta possibilidade tem ainda a vantagem de aproximar o O3F do OWL, a linguagem mais divulgada actualmente para a representação de ontologias, a qual é um standard de facto suportado pelo W3C (World Wide Web Consortium).

O conceito tipo de dados do *O3 Model* é muito importante porque é usado na definição de muitos outros conceitos.

As classes e as associações no *O3 Model* têm atributos e métodos, os quais podem ser especificados de duas maneiras equivalentes. Para melhor compreender essas duas formas de especificação de atributos e métodos de uma classe, devemos notar que, se uma classe tem o atributo A, então, se x for um elemento da classe, podemos escrever *x.A* para especificar o valor do atributo A, quando aplicado a x. Escrever *x.A*, na notação de objectos, é equivalente a escrever *A(x)* numa notação funcional. *A(x)* é o valor de A quando aplicado a x. Ou seja, um atributo de um objecto, na notação de objectos, é exactamente o mesmo que uma função de aridade 1 aplicada a esse objecto, na notação funcional. Por exemplo, a data de nascimento de uma pessoa x pode escrever-se, na notação de objectos, *x.DataNascimento*; e na notação funcional, *DataNascimento(x)*. Na notação de objectos, *DataNascimento* é um atributo; na notação funcional, *DataNascimento* é uma função unária (aridade 1).

Do mesmo modo que *x.A*, na notação de objectos em que *A* é um atributo de x, é o mesmo que a A(x), na notação funcional em que A é uma função unária, existe também um paralelo entre métodos, na notação de objectos e funções, predicados e ações quando não se usa uma notação de objectos. Podem definir-se vários tipos de métodos numa classe ou numa associação: métodos funcionais, métodos relacionais e métodos de ação. Os métodos funcionais não operam qualquer alteração no estado do mundo e, quando aplicados aos objectos da classe em que estão definidos (e eventualmente aos seus argumentos, se os houver), denotam um valor. Os métodos relacionais também não operam alterações no estado do mundo e, quando aplicados aos objectos da classe em que se definem (e eventualmente aos seus argumentos, se os houver) denotam uma proposição verdadeira ou falsa. Os métodos de ação, quando aplicados aos objectos da classe em que são definidos (e eventualmente aos seus argumentos, se os houver), efectuam alterações ao estado do mundo. Há métodos de ação que computam valores que retornam e há métodos de ação que não computam qualquer valor de retorno. Os paralelos que se podem traçar são entre métodos funcionais e funções, entre métodos relacionais e predicados, e entre métodos de ação e ações. Imaginemos o método funcional F com aridade 1, definido na classe C. Se x for um objectos da classe C e se α for o argumento de F numa dada invocação, a expressão *x.F(α)* em notação de objectos representa exactamente a mesma coisa que *F(x, α)* numa notação funcional, assumindo a convenção de que a instância da classe a que F/1 se aplica é o primeiro argumento de F/2. Consideremos a classe Pessoa, a qual possui o método Anos/1 que recebe uma data e determina o número inteiro de anos que a pessoa tem na data especificada. Numa notação de objectos, *x.Anos(2008/03/04)* significa os anos da pessoa x em 2008/03/04. Usando uma notação funcional, poderíamos escrever Anos(x, 2008/03/04) exactamente com o mesmo significado, desde que se assuma que Anos tem agora dois argumentos, e que o primeiro argumento de Anos é a pessoa e o segundo é uma data. Ou seja, um método funcional com N argumentos, em notação de objectos, funciona como uma função de N+1 argumentos em notação funcional, em que o primeiro argumento é exactamente o objecto a que a função se aplicaria no modelo de objectos.

Imaginemos agora o método relacional R com aridade 1, definido para os elementos da classe C. Sendo x, um elemento de C, a expressão x.R(α) é uma proposição que, informalmente se leria "x está na relação R com α". Em notação relacional, poderia escrever-se R(x, α) exactamente com o mesmo significado, sendo que agora a relação R tem dois argumentos. Tomando como exemplo o método relacional NomeDeDescendente da classe Cão, o qual relaciona um dado cão com os seus descendentes. Assim, se x for um membro da classe Cão e se Simba e Carlota forem os nomes de dois dos descendentes de x, *x.NomeDeDescendente(Simba)* representa a afirmação "Simba é nome de um descendente de x" e *x.NomeDeDescendente(Carlota)* significa "Carlota é nome de um descendente de x". Se, em vez de notação de objectos, usarmos uma notação relacional, poderíamos escrever as seguintes proposições exactamente com os mesmos significados que as anteriores: *NomeDeDescendente(x, Simba)* e *NomeDeDescendente(x, Carlota)*.

Algo semelhante se passa em relação a métodos de ação. Imaginemos que temos a classe Restaurante com o método de ação ReservarMesa, o qual aplicado a um restaurante, reserva uma mesa para um número especificado de pessoas, para uma dada data e hora, em nome de uma dada pessoa (especificada através do seu bilhete de identidade). Sendo x, um restaurante, *x.ReservarMesa(6, 2008/03/04, 21:00, 5460776)* significa a execução de uma reserva de mesa, no restaurante x, para 6 pessoas, para as 21 horas

de dia 2008/03/04, em nome da pessoa com o bilhete de identidade número 5460776. Se não usarmos a notação de objectos, escrevemos *ReservarMesa(x, 6, 2008/03/04, 21:00, 5460776)*. Isto significa que um método de ação com N argumentos funciona como uma ação com N+1 argumentos em que um deles é o objecto a que o método seria aplicado no modelo de objectos. Por comodidade, convenciona-se frequentemente que este é o primeiro argumento da ação.

Após a apresentação das explicações precedentes e respectivos exemplos, estamos em condições de avançar com a apresentação de uma das formas de especificar atributos e métodos em classes e associações, usando o *O3 Model*. Comecemos pelos atributos. Dado que atributos, no modelo de objectos, são funções unárias no modelo funcional, a especificação de um atributo da classe ou da associação C envolve os seguintes passos:

1 – Declaração da função unária que fará o papel de atributo dos elementos de C, por exemplo F/1.

2 – Declaração do atributo A, da classe ou da associação C, dizendo que A se define à custa da relação de C com F. Esta especificação faz-se à custa da associação *AttributeFunction* do diagrama de objectos do *O3 Model*.

Como resultado deste procedimento, fica definida a função F, e fica definido o atributo A de C, à custa de F. F e A poderão ter nomes diferentes um do outro, ou o mesmo nome.

A especificação de um método funcional FM com N argumentos na classe ou na associação C segue um procedimento análogo ao da definição de um atributo:

1 – Declaração da função F com N+1 argumentos que fará o papel de método funcional dos elementos de C.

2 – Declaração do método funcional FM, da classe ou da associação C, dizendo que FM se define à custa da relação de C com F. Esta especificação faz-se à custa da associação *FMethodFunction* do diagrama de classes do *O3 Model*. O atributo *Self* de *FMethodFunction* permite especificar o argumento de F que faz o papel do objecto a que FM é aplicado.

Como resultado deste procedimento, fica definida a função F, e fica definido o método funcional FM de C, à custa de F. F e FM poderão ter nomes diferentes um do outro, ou o mesmo nome.

A especificação de um método relacional RM com N argumentos na classe ou associação C faz-se à custa da especificação do predicado (i.e., da relação) R com N+1 argumentos:

1 – Declaração do predicado R com N+1 argumentos que fará o papel de método relacional dos elementos de C.

2 – Declaração do método relacional RM, da classe ou da associação C, dizendo que RM se define à custa da relação de C com R. Esta especificação faz-se à custa da associação *RMethodPredicate* do diagrama de classes do *O3 Model*. O atributo *Self* de *RMethodPredicate* permite especificar o argumento de R que faz o papel do objecto a que RM é aplicado.

Como resultado deste procedimento, fica definida o predicado (i.e., a relação) R, e fica definido o método relacional RM de C, à custa de R. R e RM poderão ter nomes diferentes um do outro, ou o mesmo nome.

Finalmente, a especificação do método de ação AM com N argumentos, na classe ou na associação C, é feita à custa de especificação da ação A com N+1 argumentos:

1 – Declaração da ação A com N+1 argumentos que fará o papel de método de ação dos elementos de C.

2 – Declaração do método de ação AM, da classe ou da associação C, dizendo que AM se define à custa da relação de C com A. Esta especificação faz-se à custa da associação *AMethodAction* do diagrama de objectos do *O3 Model*. O atributo *Self* de *AMethodAction* permite especificar o argumento de A que faz o papel do objecto a que AM é aplicado.

Como resultado deste procedimento, fica definida a ação A, e fica definido o método de ação AM de C, à custa de A. A e AM poderão ter nomes diferentes um do outro, ou o mesmo nome.

Os nomes dos métodos especificados por este processo são únicos apenas dentro da classe ou da associação em que são definidos; mas os nomes das funções, predicados e ações à custa de quem os métodos são definidos têm nomes únicos em toda a ontologia.

A segunda forma de especificar atributos e métodos de classes e de associações é mais tradicional (e mais típica de um modelo de objectos). Nesta alternativa, os atributos e métodos são directamente especificados dentro da sua classe ou associação. No entanto, como as duas formas de especificar devem

ser o mais equivalentes que for possível, sendo C uma classe ou associação, a especificação do atributo C.A cria implicitamente a função unária A; a especificação do método funcional C.FM com N argumentos cria implicitamente a função FM com N+1 argumentos; a especificação do método relacional C.RM com N argumentos resulta na criação implícita do predicado RM com N+1 argumentos; e a especificação do método de ação C.AM com N argumentos resulta na criação implícita da ação AM com N+1 argumentos. Em todos estes casos, o primeiro argumento dos operadores criados implicitamente fará sempre o mesmo papel que as instâncias do *classifier* a que o método é aplicado. O papel do primeiro argumento do operador criado implicitamente será *main_argument*. Os papéis dos outros argumentos serão iguais aos dos argumentos correspondentes do método definido no *classifier*. Como exemplo, voltemos ao método Restaurante.ReservarMesa da classe Restaurante, o qual tem 4 argumentos: número de pessoas para quem a mesa é reservada, data e hora a que a mesa será ocupada pelas pessoas para quem é reservada, e nº do BI da pessoa em nome de quem a mesa é reservada. Depois da especificação deste método na classe Restaurante, passará a existir a ação ReservarMesa com 5 argumentos, sendo o primeiro o restaurante (main_argument) e os outros quatro os mesmos que os do método Restaurante.ReservarMesa.

As funções, os predicados e as ações implicitamente criados pela especificação directa de atributos e métodos numa classe ou numa associação são funções, predicados e ações de pleno direito, tendo todas as suas propriedades e prerrogativas, por exemplo, definem tipos e garante-se que os seus nomes são únicos em toda a ontologia.

Apesar das regras de inferência do *O3 Model* originarem a criação automática de certos operadores quando são criados atributos e métodos de um *classifier*, é possível que ferramentas computacionais associadas ao *O3 Model* não operem essas inferências.

A respeito da especificação de atributos e métodos, resta-nos frisar que a definição por exemplo da função F com domínio C1×C2 não resulta na definição implícita dos métodos ou dos atributos correspondentes de C1 e de C2, pois, sendo x um objecto de C1 e y um objecto de C2, seria impossível saber se deveríamos escrever x.F(y) ou y.F(x). A definição da função unária F com domínio C resultaria numa ambiguidade: não poderíamos decidir se os elementos de C teriam um método funcional chamado F sem argumentos, o qual se evocaria através da expressão x.F(), ou um atributo chamado F, o qual se evocaria através da expressão x.F.

A primeira alternativa para a especificação de atributos e métodos é muito útil do ponto de vista da possibilidade de traduzir ontologias escritas em OWL em ontologias O3F pois o OWL também define os atributos de classes à custa de uma entidade chamada propriedade (*Property*) a qual é semelhante ora a funções ora a predicados.

Um dispositivo extremamente flexível e fácil de usar no *O3 Model* é a utilização de facetas. Uma faceta pode ser usada para caracterizar quase tudo quanto se possa escrever numa ontologia, desde classes e associações até aos argumentos e valores retornados por métodos passando por atributos, entre outras coisas. Existe um conjunto pre-definido de facetas no *O3 Model* e, havendo necessidade, podem definir-se novas facetas. De entre as facetas pre-definidas, exemplificam-se *maximum_value*, *mandatory*, *default_value*, *distinct*, *scope* e *values_Set*, entre muitas outras. *maximum_value* permite definir o valor máximo, por exemplo de um atributo ou de um argumento de um operador ou método. *mandatory* é uma faceta que especifica, por exemplo, se é obrigatório ou não que um dado atributo tenha valor. *default_value* pode ser usada para especificar o valor em caso de omissão (*by default*) de um atributo ou de um argumento. *distinct* permite especificar se o valor do atributo de uma dada instância tem de ser diferente do valor do mesmo atributo de uma instância diferente. Se um atributo for obrigatório e distinto, permite identificar o objecto a que pertence. *scope* indica se um dado atributo ou método se aplica a todos os objectos da classe ou da associação, ou se apenas se aplica à própria classe ou associação. Finalmente, *values_Set* permite definir o conjunto de possíveis valores, por exemplo de um atributo ou de um argumento.

Outro dos aspectos mais relevantes do *O3 Model* é a possibilidade de criar axiomas para restringir ou definir outros elementos da ontologia. Por exemplo, tendo uma classe, podemos definir outra como sendo uma extensão ou uma restrição da primeira. Essa extensão ou restrição podem ser captadas por axiomas. Actualmente, não foi ainda definida uma linguagem para a representação de axiomas.

3 Brief description of the O3 Model UML Diagram

This section presents a brief description of the UML Class Diagram of the *O3 Model*, as depicted in http://dcti.iscte.pt/O3F/model/UML_diagram.htm.

Types	
Type	This class generalizes all the O3F Types. O3F enphasizes the fact that types are sets (e.g, type *Integer* is the set of all the integers and the type Person - most likely a class - is also a set of people). Hence, in O3F, most sets are also treated as types.
Scalar	This class represents the basic O3F datatypes, e.g., Number, Integer, Float, Natural, Word, Char, String, Date, and Date_and_time
Collection	A class representing collections of elements of the same type or different types
Number	Represents all numbers in O3F. It generalizes Integer, Float and Natural
Integer	Represents all integer numbers
Float	Represents floating point numbers
Natural	Represents positive integers
Word	Represents character sequences, without spaces, starting with a letter
Char	Represents single characters
String	Represent sequences of alphanumeric characters
Time	Represents instants of time, in a 24 hour clock
Date	Represents dates
Date-and-time	Represents dates and times
URL	Represents an URL (e.g., http://iscte.pt)
eMailAddress	Represents an eMail Address (e.g., luis.botelho@iscte.pt)
Bag	This class represents collections of elements in which repetitions are allowed but the order is not relevant. The elements in a bag may have the same or different types
Sequence	Represents collections of elements in which repetitions are allowed and the order is important
Ordered Set	Represents collections of elements without repetitions, for which the order is important
Set	This class represents collections of elements in which repetitions are not allowed and the order is not relevant

Classifiers	
Classifier	A class that generalizes the two types of classifiers in O3F, Class and Association. Classifier is also an O3F Type and denotes sets of sets of name/value pairs, therefore Classifier is generalized by the class Set
Class	This class represents sets of individuals (Object class) with a similar structure.
Association	This class represents relations between classifiers
Argument	This class is used to represent arguments of an association, method or operator. It specifies the name (i.e., the role) of the argument and its type. The argument multiplicity may be specified with the Multiplicity facet
Attribute	This class represents the attributes of a classifier, including attribute name and type. Attributes may be applied to the instances of the classifier or to the classifier itself
UnambiguosKey	This class represents the sets of attributes or arguments of structured objects (e.g., Class, Association, Predicate/Relational Method, and Function/Functional Method) that uniquely identify them.

Methods	
Method	Beside attributes, in O3F a classifier can also be characterized with methods. This class represents methods. It generalizes RelationalMethod, FunctionalMethod and ActionMethod. Relational and functional methods denote sets therefore they are also types. Methods may be applied to the instances of the classifier or to the classifier itself.
RelationalMethod	This class represents relational methods. These methods do not cause changes in the world and, when applied to an instance of the classifier in which they are defined and to their arguments, form a true or false proposition.
FuncationalMethod	This class represents functional methods. These methods do not cause changes in the world. When applied to an object of the classifier in which they are defined and to their arguments denote a value (return value). This can be any kind of value – simple values (scalar) or compound values (e.g., objects).
ActionMethod	This class represents action methods. These methods change the state of the world. When applied to the object of the classifier in which they are defined and to their arguments (if any) they may return value.
Argument	This class represents the arguments of associations, methods and operators.

Operators	
Operator	The Operator class represents operators – Predicate, Function and Action. As methods, operators are also denote sets, therefore they are types. Operator is a subclass of the class Set
Predicate	This class represents predicates. A predicate can relate any number of arguments of any type but, contrarily to associations, they cannot have attributes or methods.
Function	This class represents functions. Functions apply an input set (domain) into a output set (range), In O3F, functions denote sets whose elements are sets of pairs name=value in which one represents the value returned by the function.
Action	This class represents Actions. An action has two distinct aspects, a declarative aspect and a procedural aspect. The declarative aspect defines the arguments, datatypes and return value (if any) of the action (See also ActionInterface). The procedural aspect defines the sequence of steps that are executed when the action is performed.
ActionInterface	This class represents action interfaces (also known as action signature), which are their declarative aspect. ActionInterface generalizes two kinds of action signatures – RelationalActionInterface and FunctionalActionInterface
RelationalActionInterface	The RelationalActionInterface is used to represent the relational action interfaces (relational action signatures). No value is returned. RelationalActionInterface is a particularization of the Predicate class
FuncationalActionInterface	The FunctionalActionInterface is used to represent the functional action interfaces (functional action signatures). A value is returned. FunctionalActionInterface is a particularization of the Function class
PropositionalSymbol	This class defines propositional symbols in O3F. A propositional symbol is not really an operator. Whereas predicates must have at least one argument, propositional symbols can be defined as a sequence of characters without any arguments. Propositional symbols are de simplest existing propositions.

Relation between the object oriented and the relational / functional paradigms	
RMethodPredicate	This class represents the parallelisms that can be established between predicates and relational methods. A relational method of N arguments can be seen as a predicate of N+1 arguments. The extra argument of the predicate corresponds to the object of the classifier in which the relational method is defined.
FMethodFunction	This class represents the parallelisms that can be established between functions and functional methods. A functional method of N arguments can be seen as a function of N+1 arguments. The extra argument of the function corresponds to the object of the classifier in which the functional method is defined.
AMethodAction	This class represents the parallelisms that can be established between actions and action methods. An action method of N arguments can be seen as an action of N+1 arguments. The extra argument of the action corresponds to the object of the classifier in which the action method is defined.

Individuals	
Individual	This class represents individuals of the ontology. An individual can be an instance of one or more types, but it may also exist outside of any defined type. A common use of individuals is to represent objects of class.

Hierarchies	
Hierarchy	The Hierarchy class represents hierarchies. Hierarchies may be defined of any kind of types (e.g., class hierarchies, predicate hierarchies).
HierarchicRelation	This class represents a hierarchic relation between types.
SubHierarchicRelation	This class is used to represent a sub hierarchic relation of one hierarchic relation. It can represent another hierarchic relation (a node in the hierarchic tree) or a single type (a leaf of the hierarchic tree)

Facets	
Facet	This class represents facets (built in as well as the defined). A facet is a versatile way of further describing ontology elements.
ElementFacet	This associative class represents the application of a facet to an element in the ontology.

Dependencies	
Dependency	This class represents dependency relations between two or more elements in the ontology.
DependencyArgument	This class represents dependency arguments. Dependency arguments are the elements in the ontology that are related by means of a dependency relation.

Ontologies	
Ontology	The class Ontology is the main class of the diagram. It represents ontologies and their parameters (name, version, owner, location, initial date and last modification date)
Element	This class generalizes all the other classes in the model. It represents any element that an ontology can include.

Axioms	
Axiom	This class represents axioms. In the future this functionality will be improved.

4 Tarefas para fazer / Decisões a tomar

Notas sobre as hierarquias:

- No *O3 Model* tem de existir um conjunto de meta-tipos: Datatype, Class, Association, Predicate, Function...
- Os elementos da ontologia devem ter um atributo com a classe de elemento a que pertencem (class, hierarchy, association, ...). Para isso, basta criar um axioma que diz que o ElementClass da class Element é igual ao nome da classe.
- Cada tipo deve ter o seu meta-tipo (acrescentei o atributo MetaType à classe Type)
- Cada nó de uma hierarquia deve ser do mesmo meta-tipo ou de um sub-meta-tipo do nó imediatamente acima dele na hierarquia.

Tem que se fazer diversas outras alterações no modelo:

- É necessário criar a lista de meta-tipos e representar a relação hierárquica que existe entre eles.
- Tem que se dizer que os argumentos de um método são os argumentos do operador correspondente, excepto o argumento Self.
- *Dizer que os valores de retorno de um método são os mesmos retornados pelo operador correspondente (ou, alternativamente, remover o valor de retorno de um método)*

Não se pode passar para o servidor de ontologias, a responsabilidade de avaliar expressões que envolvam operadores definidos numa ontologia (i.e., exteriores à linguagem CO3L) porque o servidor da ontologia não pode conhecer a implementação de todos esses operadores. Isto talvez imponha a necessidade de armazenar as expressões no servidor de ontologias. Quem consulta a ontologia terá a responsabilidade de fazer avaliar as expressões, o que significa que terá de se optar por efectuar a avaliação de expressões em circunstâncias diferentes.

Tem que se decidir o que acontece quando um colecção puder ser especificada por compreensão, por exemplo algo como Set(?x : (?x $\in$ Natural $\wedge$?x mod 2 = 0)). Se não houver avaliação, este será um conjunto singular cujo único elemento será a expressão simbólica ?x : (?x $\in$ Float $\wedge$?x mod 2 = 0). Se a expressão simbólica for avaliada, este é o conjunto dos números pares positivos. Como é que expressões destas podem ser avaliadas se a cardinalidade do conjunto correspondente é infinita?

Será que é necessário o operador Eval/1? Será que agora se pode assumir que as expressões são sempre avaliadas (quando)?