O3F Ontology Representation Framework

CO3L Detailed Description

(in Portuguese)

Versão 4

Data inicial desta versão: 2010-01-27

Data da última alteração desta versão: 2017-03-15

Data final desta versão:

Este documento descreve a linguagem CO3L.

Conteúdo

Linguagem CO3L

As ontologias O3F são descritas textualmente através da linguagem CO3L, a qual tem uma sintaxe e pragmática baseada nas da lógica de predicados de primeira ordem. Poderiam igualmente criar-se outras versões da linguagem de especificação de ontologias, adoptando por exemplo uma sintaxe diferente, como a sintaxe XML. Podemos igualmente escrever ontologias O3F, usando qualquer linguagem cujo modelo subjacente seja um subconjunto do modelo O3F. Isto significa que o desenvolvimento, processamento e partilha de ontologias O3F pode ser efectuado com diversas linguagens de especificação. A análise destas possibilidades está, no entanto, fora do âmbito de TSI.

Esta secção descreve, com detalhe, a linguagem CO3L.

1 Apresentação do CO3L

A descrição textual de uma ontologia inicia-se com a palavra *"Ontology"* seguida do nome lógico da ontologia, de uma chaveta, de um conjunto de parâmetros, de um conjunto de comandos, de um conjunto de asserções de descrição da ontologia, e é terminada com uma chaveta.

Descrição da Ontologia =
"Ontology" <nome da ontologia> "{"
[<parâmetros da ontologia>]
[<comandos de ontologias>]
<asserções de descrição da ontologia>
"}"

O nome da ontologia é uma designação única. Os parâmetros da ontologia são constituídos pelo nome do parâmetro, pelo sinal de dois pontos (":") e pelo valor do parâmetro. Os parâmetros da ontologia podem ser especificados em qualquer ordem. Todos os parâmetros são opcionais. Os comandos são todos opcionais e podem aparecer em qualquer ordem. Finalmente, tem de aparecer pelo menos uma asserção de descrição da ontologia. As asserções existentes podem aparecer por qualquer ordem.

Se existirem, os parâmetros, os comandos, e as descrições têm de aparecer por esta ordem.

Os parâmetros da ontologia existentes na versão actual da linguagem são *Owner*, *Initial_date*, *Last_modification_date* e *Location*.

Actualmente só existe o comando *import*, o qual serve para importar ontologias previamente definidas. O comando *import* usa-se da seguinte forma:

import(<nome da ontologia>, <Especificação de Localizações>).

O nome da ontologia identifica univocamente a ontologia a importar. A Especificação das Localizações pode ser um URL de onde a ontologia deve ser importada ou uma sequência de URLs. Neste caso, informa-se o sistema de gestão da ontologia que deve tentar importar a ontologia usando o primeiro URL especificado. Se esse falhar, poderá usar-se o segundo. O processo continua até que a importação da ontologia tenha sucesso, ou até esgotar a lista especificada.

A identificação da ontologia (o seu nome) é um nome lógico com a identificação do *namespace* a que a ontologia pertence. O operador quatro pontos (::) é usado para separar *namespaces* de sub *namespaces*. Por exemplo:

edu::cmu::ontologies::living::People

pt::iscte-iul::ontologias::Cursos

As asserções de descrição da ontologia são um conjunto de factos que declaram e definem todos os elementos da ontologia. As descrições são feitas à custa dos seguintes predicados:

- Datatype/2: Declara um novo tipo de dados básico, e especifica o tipo de dados já existente em que o novo se baseia.

- Class/1: Delara uma classe.
- Association/1: Declara uma associação entre classes / associações
- Hierarchy/1: Declara o nome de uma hierarquia de tipos
- Dependency/1: Declara a existência de uma relação de dependência
- PropSymbol/1: Declara um símbolo proposicional. *Nota: Os predicados terão de ter pelo menos um argumento.*
- Predicate/1: Declara a existência de um predicado
- Function/2: Declara a existência de uma função e do tipo de dados que retorna (contradomínio). Se for desejado, pode especificar-se o papel desempenhado pelo valor retornado, através da faceta *return_role*. Se a faceta *return_role* não for especificada, assume-se que o nome da função descreve o papel desempenhado pelo valor retornado.
- Action/1: Declara a existência de uma acção. Se a acção devolver um valor, é necessário especificar o seu tipo de dados e o papel desempenhado por esse valor, através das facetas *return_type* e *return_role*.
- Facet/1: Declara a existência de uma nova faceta
- ValidFacetElement/2: Especifica o tipo de entidades a que a faceta se pode aplicar. Os tipos de entidades válidos podem ser especificados através de uma meta-entidade (e.g., *class*, *individual*, *attribute*, *predicate*) ou através de um conjunto de entidades concretas.
- ValidFacetType/2: Especifica o tipo de dados da faceta, isto é, os valores que ela pode tomar. Os tipos de dados podem ser quaisquer dos pré-definidos no modelo ou que venham a ser definidos na ontologia. Os valores possíveis de uma faceta podem também ser especificados através de um conjunto explicitamente definido.
- Axiom/2: Define um axioma e o seu nome. Se for necessário especificar que o axioma pertence a uma classe, associação, ou a outro tipo, usa-se a faceta *Owner* (*Nota: a definição de axiomas não faz parte da matéria de TSI*).
- ObjectDef/2: Especifica um indivíduo e o seu nome. O indivíduo especificado pode ser ou não declarado como uma instância de um dado tipo de dados. O nome do indivíduo tem de ser um nome único na ontologia. Uma das utilizações mais comuns de ObjectDef/2, é a especificação de objectos estruturados compostos por atributos e correspondentes valores. O objecto definido não tem de ter uma composição guiada pela definição de alguma classe ou associação.
- Attribute/3: Define o atributo de uma classe ou associação e especifica o seu tipo de dados
- Key/3: Define um mecanismo de identificação das instâncias de uma classe ou de uma associação.
- RelationalMethod/2: Declara a existência de um método relacional e a classe ou associação a que pertence.
- FunctionalMethod/3: Declara a existência de um método funcional, o tipo de dados que retorna e a classe ou associação a que pertence. Se for desejado, pode especificar-se o papel desempenhado pelo valor retornado, através da faceta *return_role*. Se a faceta *return_role* não for especificada, assume-se que o nome do método funcional descreve o papel desempenhado pelo valor retornado.
- ActionMethod/2: Declara a existência de um método de acção e indica a classe ou associação a que pertence. Se o método de acção devolver um valor, é necessário especificar o seu tipo de dados e o papel desempenhado por esse valor, através das facetas *return_type* e *return_role*.
- AttributeFunction/3: Especifica o nome do atributo de uma classe ou associação, e o nome da função que faz o papel de atributo.
- FMethodFunction/4: Especifica o nome de um método funcional de uma classe ou associação, o nome da função que faz o papel do método funcional, e o argumento da função que faz o papel da instância a que o método é aplicado.
- AMethodAction/4: Especifica o nome de um método de acção de uma classe ou associação, o nome da acção que faz o papel do método de acção especificado, e o argumento da acção que faz o papel da instância a que o método é aplicado.

- RMethodPredicate/4: Especifica o nome de um método relacional de uma classe ou associação, o nome do predicado que faz o papel do método relacional especificado, e o argumento do predicado que faz o papel da instância a que o método é aplicado.
- Subtype/3: Especifica um dos subtipos de um dado tipo, de acordo com uma hierarquia especificada. Tanto o tipo como o subtipo podem ser quaisquer (Type), por exemplo classes, associações, datatypes, funções, e predicados.
- Argument/3: Especifica o argumento de uma associação ou de um operador, definindo o seu tipo e o seu papel (role), e a associação ou o operador a que pertence, o qual pode ser um predicado ou método relacional, uma acção ou método de acção, e uma função ou método funcional. No caso de métodos, a classe ou a associação a que o método pertence é indicada juntamente com o nome do método, por exemplo, pessoa.idade. Se for desejado especificar a multiplicidade de um argumento, como no caso dos argumentos de predicados e de associações, usa-se a faceta *Multiplicity*.
- DependencyArgument/3: Especifica um argumento de uma relação de dependência, o qual pode ser qualquer elemento da ontologia, e especifica o seu papel (*Role*) na relação de dependência especificada.
- Instance/2: Declara a existência de uma instância de um tipo de dados qualquer, em particular um Classifier (classe ou associação).
- EntityFacet/3: Especifica o valor de uma faceta de qualquer entidade da ontologia, por exemplo, tipos de dados, classes, associações, operadores e métodos, atributos, e argumentos. A especificação de facetas de entidades cuja identificação necessite de vários componentes recorre a identificações compostas. *Host.Entity* identifica uma entidade de uma classe, de uma associação, de um método ou de um operador. Finalmente, *Hierarchy*!*SuperType* identifica uma generalização.

Seguidamente descrevem-se, detalhadamente e com exemplos, todos estes predicados. Antes convém realçar que qualquer dos argumentos dos predicados usados para a descrição da ontologia (os predicados acabados de enumerar) pode, em princípio, ser especificado através de uma expressão funcional.

Datatype/2: Datatype(TipoNovo, TipoExistente)

Define um novo tipo de dados com base num tipo básico já existente. Podem acrescentar-se facetas e axiomas para tornar a definição mais precisa.

Exemplo

Datatype(TipoPreço, Float). Define o novo tipo chamado TipoPreço, o qual é um Float.

EntityFacet(TipoPreço, smallest_insance, 0). O valor mínimo do tipo de dados *TipoPreço* é 0. Ou dito de outra forma, a menor instancia do conjunto TipoPreço é 0.

Class/1: Class(Class)

Declara a existência de uma nova classe.

Exemplo: Class(Pessoa). Declara a classe chamada *Pessoa*

Association/1: Association(Association)

Declara a existência de uma associação. Os argumentos da associação são declarados através do predicado Argument/3.

Exemplo: Association(CarroPessoa). Declara a associação chamada *CarroPessoa*

Hierarchy/1: Hierarchy(Hierarchy)

Declara a existência de uma hierarquia. A sua identificação é independente dos tipos que relaciona. A especificação das relações hierárquicas entre tipos é feita através do predicado Subtype/3.

Na descrição do predicado Subtype/3 são apresentadas explicações mais detalhadas sobre a relação entre e utilização dos predicados Hierarchy/1 e Subtype/3.

Exemplo:

Hierarchy(Tipo_de_documento).

Subtype(Tipo_de_documento, Documento, DocumentoDeIdentificação). *DocumentoDeIdentificação* é uma subclasse da classe *Documento*, de acordo com a hierarquia *Tipo_de_documento*. Sabe-se que *Documento* e *DocumentoDeIdentificação* são classes (e não associações, datatypes, predicados ou funcções) pelos seus nomes, os quais têm de ser declarados com o predicado Class/1.

Dependency/1: Dependency(DependencyRelation)

Declara a existência de uma relação de dependência. Os argumentos da relação de dependência são especificados através do predicado *DependencyArgument/3*. Uma relação de dependência pode ter dois ou mais argumentos.

Exemplo:

Dependency(UtilizaMetodo).

Esta asserção declara a existência de uma relação de dependência chamada *UtilizaMetodo*.

PropSymbol/1: PropSymbol(Symbol)

Declara um símbolo proposicional. Os predicados terão de ter pelo menos um argumento, enquanto que os símbolos proposicionais são formados por uma sequência de caracteres sem quaisquer argumentos.

Exemplo: PropSymbol(SistemaIndisponvel). Declara a existência do símbolo proposicional *SistemaIndisponivel*. O valor de verdade do símbolo não fica definido.

ObjectDef/2: ObjectDef(ObjectIdentifier, ObjectDefinition)

Especifica um indivíduo e o seu nome. O indivíduo especificado pode ser ou não declarado como uma instância de um dado tipo de dados. O nome do indivíduo tem de ser um nome único na ontologia. Uma das utilizações mais comuns de ObjectDef/2 é a especificação de objectos compostos por atributos e correspondentes valores. O objecto definido não tem de ter uma composição guiada pela definição de alguma classe ou associação.

Utilização: ObjectDef(ObjectIdentifier, ObjectDefinition)

Em que *ObjectIdentifier* é uma designação, única na ontologia, do individuo declarado; e *ObjectDefinition* é a definição do indivíduo declarado. Se o indivíduo declarado for simples, *ObjectDefinition* será o seu valor (e.g., 5, "Ana Maria"). Se o indivíduo declarado for uma colecção, *ObjectDefinition* será essa colecção (e.g., Sequence(10, 20)). Se o indivíduo especificado for um objecto composto, estruturado, *ObjectDefinition* será um conjunto cujos elementos são pares atributo valor, com o formato attributo=valor. Não é forçoso que a constituição do objecto composto estruturado (i.e., os seus atributos) corresponda à definição dos atributos de alguma classe ou associação.

Exemplo

ObjectDef(p001, Set(bi = 7369766, nome = "Ana Sofia", cor_preferida = vermelho))

Esta asserção define um objecto com os pares atributo/valor especificados. Com esta definição, a aplicação do operador # ao identificador p001 (#p001) é uma forma abreviada de escrever o objecto por extenso:

Set(bi = 7369766, nome = "Ana Sofia", cor_preferida = vermelho)

Os atributos do objecto criado não têm de coincidir com os atributos de alguma classe ou associação. Pode acontecer também que alguns dos atributos, por exemplo bi e nome, coincidam com os atributos de uma dada classe ou associação e outros (e.g., cor_preferida) não correspondam a nenhuma classe ou associação.

Pode declarar-se que um indivíduo definido com o predicado ObjectDef/2 é uma instância de qualquer tipo de dados. Em particular, pode declarar-se que o objecto definido é uma instância de uma classe ou de uma associação. Para isso, basta que o objecto tenha os atributos obrigatórios dessa classe ou associação, mesmo que não tenha outros atributos da definição da classe ou da associação, e mesmo que tenha outros atributos que não correspondam à definição da classe ou da associação. Como exemplo, podemos considerar o seguinte excerto de uma ontologia com a classe Pessoa e o objecto p001 (já definido).

Class(Pessoa).

Attribute(Pessoa, bi, Natural).

Attribute(Pessoa, nome, String).

Attribute(Pessoa, telefone, Natural).

EntityFacet(Pessoa.bi, mandatory, true).

EntityFacet(Pessoa.nome, mandatory, true).

Instance(#p001, Pessoa).

Apesar do objecto p001 não ter o atributo telefone da classe Pessoa (o qual não é obrigatório) e de ter um atributo adicional (i.e., cor_preferida), p001 pode ser uma instância de Pessoa porque tem todos os seus atributos obrigatórios.

Predicate/1: Predicate(PredicateName)

Declara a existência de um predicado. Os tipos dos argumentos (domínio do predicado) podem ser especificados através do predicado Argument/3.

Exemplo: Predicate(melhor_preco). Define o predicado *melhor_preco*.

Function/2: Function(FunctionName, FunctionType)

Declara a existência de uma função e o seu tipo de dados, isto é, o tipo de dados dos valores retornados pela função (contradomínio). Os tipos dos argumentos (domínio da função) são especificados através do predicado Argument/3.

Utilizações: Function(FuncName, DataType). Em vez de *DataType*, pode ser *ClassName*, *AssociationName*, *PredicateName*, ou *FunctionName*. É possível distinguir *DataType* de *ClassName*, de *AssociationName*, de *PredicateName* e de *FunctionName* pelo nome já que estas entidades não podem ter nomes idênticos.

Exemplo: Function(Idade, Integer). Define a função *Idade*, a qual retorna um valor inteiro.

Se for desejado, pode especificar-se o papel desempenhado pelo valor retornado, através da faceta *return_role*. Se a faceta *return_role* não for especificada, assume-se que o nome da função descreve o papel desempenhado pelo valor retornado.

Action/1: Action(ActionName)

Declara a existência de uma acção. Se a acção devolver um valor, é necessário especificar o seu tipo de dados e o papel desempenhado por esse valor, através das facetas *return_type* e *return_role*. Os tipos e papéis dos argumentos (domínio da acção) podem ser especificados através do predicado Argument/3.

Exemplo

Action(DarPassoEmFrente).

EntityFacet(DarPassoEmFrente, return_type, boolean).

EntityFacet(DarPassoEmFrente, return_role, execution_status).

As três proposições do exemplo definem a acção *DarPassoEmFrente*, a qual devolve um booleano que representa o estado de sucesso da execução da acção (*true* significa sucesso, e *false* significa insucesso, por exemplo).

Facet/1: Facet(FacetName)

Declara a existência de uma faceta nova.

Os predicados ValidFacetElement/2 e ValidFacetType/2 são usados para especificar os elementos da ontologia a que a nova faceta se pode aplicar e os valores válidos que a faceta pode tomar. A faceta *default_value* pode ainda ser usada para especificar o valor que a faceta pode tomar por omissão.

Exemplo:

Facet(cor).

EntityFacet(cor, default_value, branco).

Declara a faceta Cor. Através de default_value, foi estabelecido que o valor da faceta cor, por omissão, é *branco*.

Para além disto, o significado da faceta, para um sistema de gestão de ontologias baseado no modelo O3F, será totalmente nulo a menos que a nova faceta seja caracterizada, por exemplo, usando os predicados *ValidFacetElement/2* e *ValidFacetType/2*, e axiomas.

ValidFacetElement/2: ValidFacetElement(FacetName, ValidEntity)

Especifica o tipo de entidades a que a faceta se pode aplicar. Os tipos de entidades válidos podem ser especificados através de uma meta-entidade (e.g., *class*, *individual*, *attribute*, *predicate*) ou através de um conjunto de entidades concretas (e.g., Set(Pessoa.Nome, Pessoa.Morada, Empresa.Nome, Empresa.Morada)).

Para cada faceta, pode existir mais do que uma asserção do predicado ValidFacetElement/2.

Utilizações

ValidFacetElement(FacetName, Meta-Entity), em que Meta-Entity é um dos tipos de entidades da ontologia, por exemplo *class*, o que significaria que a faceta poderia ser aplicada a todas as classes.

ValidFacetElement(FacetName, ConcreteEntitySet), em que ConcreteEntitySet é um conjunto de entidades concretas, caso em que a faceta se poderia aplicar às entidades concretas contidas nesse conjunto.

Exemplo

Neste exemplo, cria-se a faceta *controlled_access* que serve para especificar as entidades da ontologia cujo acesso é controlado e quem é que exerce esse controlo. A faceta *controlled_access* pode ser aplicada a atributos e a (valores de retorno de) funções e métodos funcionais. Esta especificação pode ser feita através das seguintes asserções.

Facet(controlled_acess).

ValidFacetElement(controlled_access, attribute).

ValidFacetElement(controlled_access, function).

ValidFacetElement(controlled_access, functional_method).

ValidFacetType/2: ValidFacetType(FacetName, Type)

Especifica o tipo de dados da faceta, isto é, os valores que ela pode tomar. Os tipos de dados podem ser quaisquer dos pré-definidos no modelo ou que venham a ser definidos na ontologia. Os valores possíveis de uma faceta podem também ser especificados através de um conjunto.

Utilizações

ValidFacetType(FacetName, Type) significa que os valores válidos da faceta são do tipo Type. Type pode ser um tipo pré-definido no O3F ou um tipo definido numa ontologia.

ValidFacetType(FacetName, ConcreteValueSet) significa que a faceta apenas pode tomar os valores enumerados no conjunto ConcreteValueSet.

Exemplo

Podemos especificar que o acesso aos valores aos elementos da ontologia pode ser controlado apenas pela Ana Respício e pelo Ricardo Fonseca através da definição dos valores válidos da faceta *controlled_acess*, criada no exemplo relativo ao predicado ValidFacetElement/2:

ValidFacetType(controlled_access, Set("Ana Respício", "Ricardo Fonseca")).

Axiom/2: Axiom(AxiomName, AxiomDefinition)

Define um axioma. Se for desejado arrumá-lo num dado tipo de dados, usa-se a faceta *Owner*.

Exemplo

O seguinte axioma, chamado defIdade, define a função Idade, a qual se aplica a objectos da classe Pessoa. A faceta *Owner* associa o axioma à classe Pessoa.

Nota: a linguagem usada para definir o axioma é meramente hipotética; a linguagem CO3L ainda não foi apetrechada com a linguagem adequada de axiomas.

```
Axiom(
  defIdade,
  forall(?x,
    implies(
      instance(?x,  Pessoa),
      ?x.Idade() = currentDate().Minus(?x.DataNascimento)))
)

EntityFacet(defIdade, Owner, Pessoa).
```

Attribute/3: Attribute(Host, Name, Type)

Declara a existência de um atributo de uma classe ou associação e seu tipo de dados.

Utilizações: Attribute(ClassName, AttributeName, DataTypeName). Em vez de *ClassName* pode ser *AssociationName*, e em vez de *DataTypeName* pode ser *ClassName, AssociationName*, FunctionName, ou *PredicateName*. A distinção das várias utilizações baseia-se no nome das entidades envolvidas.

Exemplo: *Attribute(Pessoa, dataNascimento, Date)*. Define o atributo chamado *dataNascimento* da classe *Pessoa*, cujo valor é do tipo *Date*.

Key/3: Key(Classifier, KeyName, Attribute)

Especifica um atributo ou argumento de uma classe, associação, operador ou método que constitui um identificador único das instâncias dessa classe/associação/operador/método ou que pertence a um conjunto de atributos / argumentos que constituem globalmente um mecanismo de identificação. Cada classe / associação / operador / método pode ter mecanismos de identificação alternativos. O predicado Key/3 corresponde à classe *UnambigousKey* do modelo O3F.

Utilização: Key(ClassName, KeyName, AttributeName). Em vez de *ClassName*, pode usar-se *AssociationName, Operator, MethodIdentification*.

Exemplo: O exemplo abstracto que se segue define dois mecanismos de identificação alternativos, um com um único atributo, e outro com dois atributos.

```
Key(C, key1, a1).     // Classe C com uma chave formada apenas pelo atributo a1

Key(C, key2, a2).     // Classe C com uma chave formada por dois atributos: a2 e a3

Key(C, key2, a3).
```

Isto significa que os objectos da classe C podem ser identificados ou pelo atributo a1, ou pelo conjunto de atributos a2 + a3.

RelationalMethod/2: RelationalMethod(Host, Method)

Declara a existência de um método relacional e a classe ou associação (Classifier) a que pertence. Os argumentos do método são declarados através do predicado Argument.

Utilizações:

RelationalMethod(ClassName, RelationalMethod). Em vez de *ClassName*, pode usar-se *AssociationName*. Distingue-se uma classe de uma associação, apenas pelo seu nome, porque as classes e as associações não podem ter nomes iguais.

Exemplo:

O seguinte exemplo declara o método relacional descendente da classe Pessoa que relaciona a pessoa a que é aplicado com um dos seus descendentes, que também é uma pessoa.

RelationalMethod(Pessoa, descendente). Declara o método relacional *descendente* da classe *Pessoa*. O tipo de dados do argumento de *descendente* é definido através do predicado *Argument*:

Argument(Pessoa.descendente, descendente, Pessoa). O método relacional *descendente* da classe *Pessoa* tem um argumento cujo papel é *descendente* e cujo tipo é a classe *Pessoa*.

FunctionalMethod/3: FunctionalMethod(Host, Method, Type)

Declara a existência de um método funcional, o tipo de dados que retorna e a classe ou associação a que pertence. Se for desejado, pode especificar-se o papel desempenhado pelo valor retornado, através da faceta *return_role*. Se a faceta *return_role* não for especificada, assume-se que o nome do método funcional descreve o papel desempenhado pelo valor retornado. Os argumentos do método são declarados através do predicado Argument/3.

Utilizações:

FunctionalMethod(ClassName, FunctionalMethod, DataTypeName). Em vez de ClassName, pode usar-se *AssociationName* (para métodos de associação). Em vez de *DataTypeName*, pode usar-se *ClassName*, *AssociationName*, *PredicateName*, ou *FunctionName*. As várias utilizações diferentes distinguem-se entre si com base nos nomes dos argumentos, pois as entidades que podem ser passadas como argumentos de FunctionalMethod não podem ter nomes coincidentes.

Exemplo: *FunctionalMethod(Pessoa, Idade,* Integer*)*. Declara o método funcional *Idade* da classe *Pessoa*, o qual devolve um inteiro que representa a idade da pessoa expresso em número de anos. Para se saber que o valor retornado seria o número de anos da pessoa, teria de se usar a faceta *return_role*:

EntityFacet(Pessoa.Idade, return_role, numero_de_anos).

ActionMethod/2: ActionMethod(Host, Method)

Declara a existência de um método de acção e indica a classe ou associação a que pertence. Se o método devolver um valor, é necessário especificar o seu tipo de dados e o papel desempenhado por esse valor, através das facetas *return_type* e *return_role*. Os argumentos do método são declarados através do predicado Argument/3.

Exemplo:

ActionMethod(Restaurante, reservarMesa).

EntityFacet(Restaurante.reservarMesa, return_type, Natural).

EntityFacet(Restaurante.reservarMesa, return_role, codigoReserva).

As três asserções do exemplo declaram a existência do método de acção *reservarMesa* da classe *Restaurante*, o qual retorna um natural que desempenha o papel de código da reserva. O método tem vários argumentos – BI da pessoa em nome da qual a reserva é feita, data e hora pretendida para a reserva, e número de pessoas. Estes argumentos definem-se através do predicado Argument/3, por exemplo:

Argument(Restaurante.reservarMesa, nPessoas, Natural). O método de acção *reservarMesa* da classe *Restaurante* tem um argumento cujo papel é *nPessoas* e cujo tipo é *Natural*.

AttributeFunction/3: AttributeFunction(Host, Atribute, Function)

Especifica o nome do atributo de uma classe ou associação, e o nome da função que faz o papel de atributo, algo como "F é a função que implementa o atributo A da classe C". É uma maneira alternativa, em relação à utilização do predicado Attribute/3, de declarar um atributo de uma classe ou associação.

Utilizações:

AttributeFunction(ClassName, Attribute, FunctionName). Em vez de *ClassName*, pode ser *AssociationName*.

Exemplo: Imaginando que existe a função *DataNascimento/1* que recebe uma pessoa como argumento e devolve a sua data de nascimento, a seguinte proposição especifica que o atributo chamado *dataNascimento* da classe *Pessoa* se define à custa da função *DataNascimento/1*:

AttribueFunction(Pessoa, dataNascimento, DataNascimento).

AMethodAction/4: AMethodAction(Host, ActionMethod, Action, Self)

Especifica o nome de um método de acção de uma classe ou associação, o nome da acção que faz o papel do método especificado, e o argumento da acção que faz o papel da instância a que o método é aplicado. Esta é uma forma alternativa de especificar métodos relativamente à utilização do predicado ActionMethod/2.

Utilizações: AMethodAction(ClassName, ActionMethod, ActionName, SelfArgument). Em vez de *ClassName*, pode usar-se *AssociationName*.

O argumento *SelfArgument* especifica o argumento da acção que faz o papel da instância a que o método é aplicado, numa invocação. Esta especificação faz-se através do papel (*role*) desse argumento.

Exemplo:

Neste exemplo, assume-se que existe a acção *ReservarBilhete* para reservar bilhetes para um dado filme, a qual recebe a identificação de uma pessoa (*bi*), o número de bilhetes a reservar (*nbilhetes*), o cinema em que a reserva é feita (*cinema*), a sala (*sala*) onde passa o filme que se pretende ver e o horário da sessão (*horario*). O método *fazerReserva* da classe *Cinema* pode ser definido através da acção *ReservarBilhete*, através da seguinte proposição:

AMethodAction(Cinema, fazerReserva, ReservarBilhete, cinema).

SelfArgument : cinema significa que o argumento com papel (role) *cinema* da acção ReservarBilhete corresponde à instância da classe Cinema a que o método será aplicado numa invocação particular.

Fazendo esta definição do método *fazerReserva* e assumindo que x é uma instancia da classe Cinema, as duas expressões seguintes seriam equivalentes:

reserva = ReservarBilhete(bi : 5469226, nbilhetes : 3, cinema : x, sala : 1, horário : 15:30)

reserva = x.fazerReserva (bi : 5469226, nbilhetes : 3, sala : 1, horário : 15:30)

Ambas reservam três bilhetes, em nome da pessoa cujo BI é 5469226 na sala 1 do cinema especificado por x, para a sessão das 15:30.

FMethodFunction/4: FMethodFunction(Host, FunctionalMethod, Function, Self)

Especifica o nome de um método funcional de uma classe ou associação, o nome da função que faz o papel do método funcional especificado, e o argumento da função que faz o papel da instância a que o método é aplicado. Esta é uma forma alternativa de especificar métodos relativamente à utilização do predicado FunctionalMethod/3.

Utilizações: FMethodFunction(ClassName, FunctionalMethod, FunctionName, SelfArgument). Em vez de *ClassName*, pode usar-se *AssociationName*.

O argumento *SelfArgument* especifica o argumento da função que faz o papel da instância a que o método funcional é aplicado, numa invocação. Esta especificação faz-se através do role desse argumento.

O exemplo apresentado a propósito do predicado AMethodAction/4 pode ser consultado para uma melhor compreensão do papel argumento *SelfArgument*.

RMethodPredicate/4: RMethodPredicate(Host, RelationalMethod, Predicate, Self)

Especifica o nome de um método relacional de uma classe ou associação, o nome do predicado que faz o papel do método relacional especificado, e o argumento do predicado que faz o papel da instância a que o método é aplicado. Esta é uma forma alternativa de especificar métodos relativamente à utilização do predicado RelationalMethod/2.

Utilizações: RMethodPredicate(ClassName, RelationalMethod, PredicateName, SelfArgument). Em vez de *ClassName*, pode usar-se *AssociationName*.

O argumento *SelfArgument* especifica o argumento do predicado que faz o papel da instância a que o método é aplicado, numa invocação. Esta especificação faz-se através do role desse argumento.

O exemplo apresentado a propósito do predicado AMethodAction/4 pode ser consultado para uma melhor compreensão do papel argumento *SelfArgument*.

Argument/3: Argument(Operator, ArgumentRole, ArgumentType)

Especifica o argumento de um operador (predicado, função ou acção), de um método (relacional, funcional ou de acção) ou de uma associação. No caso de um método, o método tem de ser univocamente identificado pela classe ou associação a que pertence, por exemplo *Restaurante.reservarMesa*.

Se for necessário especificar a multiplicidade de um argumento (o que pode acontecer em geral para argumentos de predicados, métodos relacionais e associações), usa-se a faceta *Multiplicity*.

Utilizações:

Argument(Action, Role, DataTypeName). Em vez de *Action*, poderá ser *ActionMethod*, *Function*, *FunctionalMethod*, *Predicate*, *RelationalMethod* ou *Association*; e em vez de *DataTypeName*, poderá ser *ClassName*, *AssociationName*, *FunctionName*, ou *PredicateName*.

Exemplos

Argument(Restaurant.researvarMesa, nPessoas, natural) define o argumento *nPessoas* de tipo *natural* do método de acção *reservarMesa* da classe *Restaurante*.

Argument(CaoPessoa, cao, Cao)

Argument(CaoPessoa, dono, Pessoa)

EntityFacet(CaoPessoa.cao, Multiplicity, 0..*)

EntityFacet(CaoPessoa.dono, Multiplicity, 0..*)

Subtype/3: Subtype(Hierarchy, ParentType, SubType)

Especifica um dos subtipos de um determinado supertipo, na hierarquia especificada. Quaisquer tipos de dados podem ser particularizados/generalizados numa hierarquia, por exemplo classes, associações e predicados. No entanto, o meta-tipo de um nó de uma hierarqui deve ser igual ou um subtipo do meta-tipo do nó imediatamente acima, na mesma hierarquia.

Utilizações

Subtype(HierarchyName, ClassName, ClassName). Em vez de *ClassName*, pode usar-se *AssociationName*, *PedicateName*, *FunctionName*, ou *DatatypeName*.

Exemplos

Hierarchy(Tipo_de_documento).

Subtype(Tipo_de_documento, Documento, DocumentoIdentificação).

Mais utilizações

Além de poderem ser usados para especificar a relação de generalização / particularização entre um conjunto e os seus subconjuntos, os predicados *Hierarchy/1* e *Subtype/3* podem também ser usados para especificar uma hierarquia completa de conceitos, com vários níveis, como por exemplo, a taxonomia dos mamíferos.

A Figura 1 mostra um Diagrama de Classes UML que representa uma hierarquia com vários níveis.

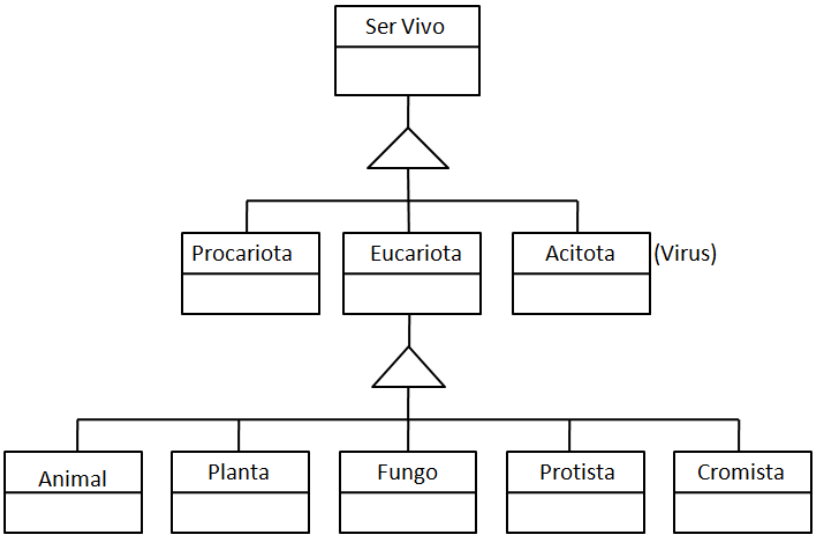

Figura 1 – Classificação de Seres Vivos

O A hierarquia do Diagrama de Classes da Figura 1 pode ser representado através das seguintes proposições CO3L:

Hierarchy(ClassificaçãoSeresVivos)
Subtype(ClassificaçãoSeresVivos, SerVivo, Eucariota)
Subtype(ClassificaçãoSeresVivos, SerVivo, Procariota)
Subtype(ClassificaçãoSeresVivos, SerVivo, Acitota)
Subtype(ClassificaçãoSeresVivos, Eucariota, Animal)
Subtype(ClassificaçãoSeresVivos, Eucariota, Planta)
Subtype(ClassificaçãoSeresVivos, Eucariota, Fungo)
Subtype(ClassificaçãoSeresVivos, Eucariota, Protista)
Subtype(ClassificaçãoSeresVivos, Eucariota, Cromista)

Há casos em que é necessário definir mais do que uma hierarquia. Por exemplo, na Figura 2, a divisão da classe Animal nas subclasses Radiata, Mixozoa, Mesozoa e Bilateria pertence à mesma hierarquia que a divisão da classe Eucariota nas suas subclasses. No entanto, a divisão de Animal nas subclasses Doméstico e Selvagem não tem nada que ver com a hierarquia dos seres vivos.

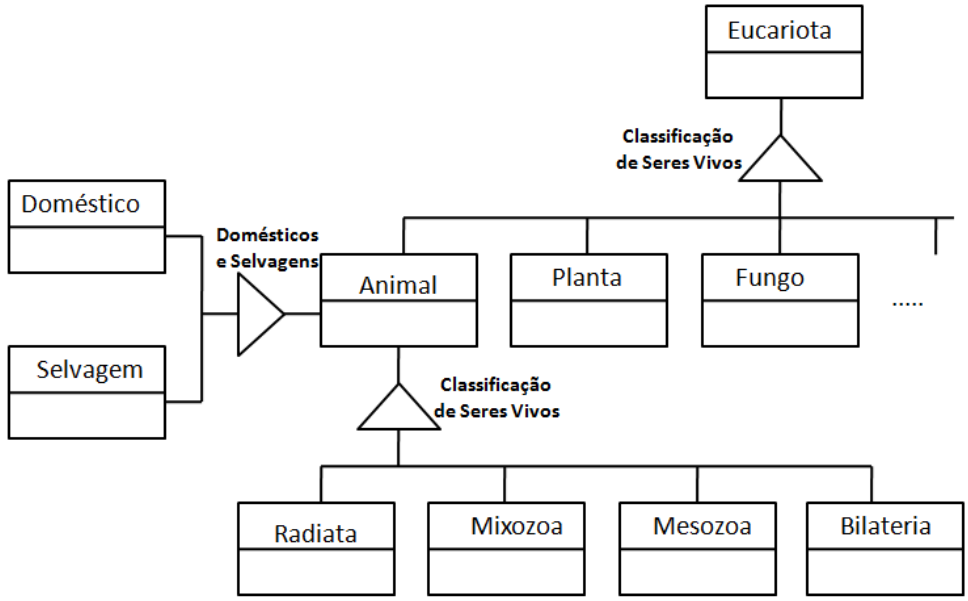

Figura 2 – Duas hierarquias

Em CO3L, as hierarquias da Figura 2 representam-se através das seguintes proposições:

Hierarchy(ClassificaçãoSeresVivos)
Subtype(ClassificaçãoSeresVivos, Eucariota, Animal)
Subtype(ClassificaçãoSeresVivos, Eucariotas, Planta)
Subtype(ClassificaçãoSeresVivos, Eucariotas, Fungo)
....
Subtype(ClassificaçãoSeresVivos, Animal, Radiata)
Subtype(ClassificaçãoSeresVivos, Animal, Mixozoa)
Subtype(ClassificaçãoSeresVivos, Animal, Mesozoa)
Subtype(ClassificaçãoSeresVivos, Animal, Bilateria)

Hierarchy(Domésticos_Selvagens)
Subtype(Domésticos_Selvagens, Animal, Doméstico)
Subtype(Domésticos_Selvagens, Animal, Selvagem)

Instance/2: Instance(InstanceSpecification, TypeSpecification)

Declara a existência de uma instância de um dado tipo, em particular de um *Classifier* (classe ou association). As instâncias podem ser escritas de duas formas, uma por extenso e outra abreviada. Uma instância de um *classifier* escrita por extenso tem o seguinte formato:

Set(<list of attribute = value pairs>)

As instâncias de predicados e de funções, de métodos relacionais e de métodos funcionais têm o seguinte formato.

Set(<list of argument = value pairs>)

Os pares attribute = value e argument = value separam-se uns dos outros por vírgulas.

Quando se usa a especificação, por extenso, de uma instância, todos os atributos ou argumentos obrigatórios, se os houver, têm de ter valor. Além dos atributos obrigatórios, a especificação de uma instância pode incluir qualquer número de atributos ou argumentos adicionais, incluindo atributos ou argumentos que não fazem parte da definição do classifier, do operador ou do método a que a instância pertence.

A escrita abreviada de uma instância faz-se recorrendo ao operador # e à identificação única da instância, por exemplo #p001. O operador # aplicado ao identificador de uma instância significa a sua definição, por exemplo Set(bi = 5110227, nome = "Luís Botelho", tlm = +351999402240).

Os identificadores únicos das instâncias, quando usados, têm de ser definidos no predicado ObjectDef/2.

Exemplos:

Instance(Set(bi = 5110227, nome = "Luís Botelho", tlm = +351999402240), Pessoa).

Instance(#p001, Pessoa).

```
Instance(
  Set(
    matricula = "31-EH-20",
    marca = "Honda",
    dono = Set(bi = 5110227, nome = "Luís Botelho", tlm = +351999402240)
  ),
  Carro
).
```

Instance(Set(matricula = "31-EH-20", marca = "Honda", dono = #p001), Carro).

Instance(#c0012, Carro).

Datatype(Par, Integer).

Instance(2, Par). 2 é uma instância do tipo de dados Par, o qual é definido através de uma asserção do predicado Datatype/2.

Predicate(Dobro).

Argument(Dobro, numero, Natural).

Argument(Dobro, dobro, Natural).

Instance(Set(numero = 3, dobro = 6), Dobro). O conjunto Set(numero = 3, dobro = 6) é uma instância do conjunto denotado pelo predicado Dobro.

Os exemplos e explicações apresentados mostram claramente a unificação, existente no O3F, do paradigma de centrado em objectos e do paradigma relacional e funcional.

DependencyArgument/3: DependencyArgument(DependencyRelation, Role, Entity)

Especifica a entidade que desempenha um dado papel numa relação de dependência. *Entity* é a entidade relacionada pela relação de dependência *DependencyRelation*. *Role* é o papel desempenhado pela entidade *Entity* na relação de dependência.

Exemplo:

Vamos supor que uma classe C1, mais geral, depende da uma das suas subclasses C2 porque, em determinadas circunstâncias, o método M1 da classe C1 invoca o método M2 da classe C2.

Dependency(UtilizaMetodo)

DependencyArgument(UtilizaMetodo, classe_geral, C1)

DependencyArgument(UtilizaMetodo, subclasse, C2)

EntityFacet(UtilizaMetodo.classe_geral, dependency, dependent)

EntityFacet(UtilizaMetodo.subclasse, dependency, independent)

A primeira asserção de *DependencyArgument/3* especifica que *C1* é um argumento da relação de dependência *UtilizaMetodo*, o qual desempenha o papel *classe_geral*.

A segunda asserção de *DependencyArgument/3* especifica que *C2* é um argumento da relação de dependência *UtilizaMetodo*, o qual desempenha o papel *subclasse*.

A natureza concreta desta dependência pode ser captada por um axioma que especifique as condições em que o método M1 da classe C1 usa o método M2 da classe C2.

As duas asserções do predicado *EntityFacet/3* especificam que o argumento UtilizaMetodo.classe_geral é dependente, e que o argumentos UtilizaMetodo.subclasse é independente.

EntityFacet/3: EntityFacet(Entity, Facet, Value)

Associa uma faceta com um dado valor a qualquer entidade da ontologia. O valor da faceta (argumento *Value*) pode ser especificado através de uma constante (e.g., 5, Set(a, b, c, d), concrete) ou através de uma expressão cuja avaliação resulta num valor específico (e.g., current_date(), current_date().year()).

Cada tipo de entidade pode ser associada apenas a um conjunto específico de facetas, e cada faceta pode ser associada a um ou mais tipos de entidade.

Muitas entidades podem ser identificadas apenas pelo seu nome (e.g., uma classe ou um tipo de dados (*datatype*)). No entanto, existem também entidades que têm de ser identificadas por dois ou mesmo três componentes (e.g., atributo de uma classe, argumento de um método de uma classe ou associação). Uma entidade cuja identificação é constituída por mais do que um componente é especificada através de uma expressão única que agrega os vários componentes constitutivos da identificação.

Os atributos e métodos de uma classe ou associação são identificados através de uma expressão com o formato geral *Host.Entity*, em que *Host* é o nome da classe ou da associação, e *Entity* é o nome do método ou do atributo.

Os argumentos de uma associação são especificados por uma expressão com o formato *Association.Role*, em que *Association* representa o nome da associação e *Role* é o papel desempenhado pelo argumento identificado.

Os argumentos de uma relação de dependência são igualmente especificados através do operador ponto (.).

Os argumentos de um operador são identificados por uma expressão com o formato geral *Operator.Role*, em que *Operator* é o nome do operador a que o argumento pertence, e *Role* é o nome que identifica um argumento do operador especificado.

Os argumentos de um método de uma classe ou associação são identificados por uma expressão com o formato geral *Host.Method.Role*, em que *Host* é o nome da classe ou associação a que o método pertence, *Method* é o nome do método a que o argumento pertence, e *Role* é o nome que identifica um argumento do método especificado.

Uma generalização também pode ser caracterizada por facetas. A faceta *complete* aplica-se a um par formado por uma hierarquia e um super-tipo. Este par é especificado por uma expressão com o formato Hierarchy!SuperType.

Dada a grande variedade e quantidade de entidades de uma ontologia O3F, não será possível apresentar todas as utilizações possíveis e exemplos concretos do predicado *EntityFacet*. Será mais importante apresentar um conjunto de utilizações e de exemplos concretos suficiente para se perceber o estilo da especificação de modo a poder aplicá-la noutras situações.

Utilizações:

EntityFacet(Class.ActionMethod.ArgumentRole, FacetName, FacetValue). Especifica uma faceta de um argumento de um método de uma dada classe ou associação. Em vez de *Class*, poderia ser *Association*. Em vez de *ActionMethod*, poderia ser *FunctionalMethod*, ou *RelationalMethod*.

EntityFacet(Action.ArgumentRole, FacetName, FacetValue). Especifica uma faceta de um argumento de um operador. Em vez de *Action*, poderia ser *Function*, ou *Predicate*.

EntityFacet(AssociationName.ArgumentRole, FacetName, FacetValue). Especifica uma faceta de um argumento de uma associação.

EntityFacet(Class.Attribute, FacetName, FacetValue). Especifica uma faceta de um atributo de uma dada classe ou associação. Em vez *Class*, também poderia ser *Association*.

EntityFacet(Class.FunctionalMethod, FacetName, FacetValue). Especifica uma faceta de um método de uma classe ou associação. Em vez de *Class*, poderia ser *Association*. Em vez de *FunctionalMethod*, poderia usar-se *RelationalMethod* ou *ActionMethod*.

EntityFacet(Hierarchy!SuperType, FacetName, FacetValue). Especifica uma faceta de uma generalização.

EntityFacet(Function, FacetName, FacetValue). Especifica a faceta de um operador. Em vez de *Function*, poderia usar-se *Predicate*, ou *Action*.

EntityFacet(ClassName, FacetName, FacetValue). Especifica uma faceta de um *classifier*. Em vez de *ClassName*, poderia usar-se *AssociationName*.

EntityFacet(DataTypeName, FacetName, FacetValue). Especifica uma faceta de um tipo de dados básico (Datatype).

Para certas facetas, os atributos, métodos e operadores são interpretados como se fossem conjuntos; para outras facetas, é o seu carácter de operador que é usado.

Exemplos:

EntityFacet(Restaurante.reservarMesa.nPessoas, minimum_value,1). A expressão associa a faceta *minimum_value* com valor 1 ao argumento *nPessoas* do método *reservarMesa* da classe Restaurante.

EntityFacet(Restaurante.reservarMesa.nPessoas, Arg_number, 2). A expressão associa a faceta *arg_number* com valor 2 ao argumento *nPessoas* do método *reservarMesa* da classe Restaurante. Isto significa que o argumento que faz o papel nPessoas é o segundo argumento do método reservarMesa. Esta faceta é útil quando não há possibilidade de especificar explicitamente o papel do argumento na invocação de um método ou operador.

EntityFacet(Pessoa.bi, mandatory, true). A expressão associa a faceta *mandatory* com valor *true*, ao atributo *bi* (bilhete de identidade) da classe *Pessoa*. Ou seja, bi é um atributo obrigatório da classe Pessoa.

EntityFacet(Pessoa.nomeDeDescendent, scope, instance). Associa a faceta *Scope* com valor *Instance* ao método relacional *nomeDeDescendente* da classe *Pessoa*. Isto é, nomeDeDescendente é um método das instancias de Pessoa.

EntityFacet(Tipo_de_documento!Documento, complete, true). A generalização identificada pela hierarquia Tipo_de_documento e pela super-classe Documento é completa. Isto significa que qualquer instância da classe Documento tem de pertencer a uma das suas subclasses, definidas pela hierarquia Tipo_de_documento.

EntityFacet(Animal, materialization, abstract). Indica que a classe *Animal* é uma classe abstracta, isto é, não é possível especificar instâncias directamente na classe *Animal*. Apenas as subclasses de *Animal* têm instâncias. As instâncias de *Animal* são herdadas das suas subclasses.

EntityFacet(Tipo_de_documento, leaf, BI).

EntityFacet(Tipo_de_documento, leaf, NF). A hierarquia *Tipo_de–documento* tem duas classes folha: *BI* (Bilhete de Identidade) e *NF* (Número Fiscal). Sabe-se que *BI* e *NF* são classes apenas pelo seu nome. Assume-se que existem as duas asserções Class(BI) e Class(NIF).

O próximo exemplo é mais completo que os anteriores. Nele, define-se uma associação de tipo composição entre a classe Hotel e a classe Quarto. Os atributos das classes não são especificados porque não são necessários para o propósito do exemplo, o qual se centra na utilização das facetas *AssociationType*, *Whole* e *Part*.

Class(Hotel)

Class(Quarto)

Association(QuartoHotel)

EntityFacte(QuartoHotel, association_type, Composition)

Argument(QuartoHotel, hotel, Hotel)

Argument(QuartoHotel, quarto, Quarto)

EntityFacet(QuartoHotel.hotel, Multiplicity, 1)

EntityFacet(QuartoHotel.quarto, Multiplicity, 1..*)

EntityFacte(QuartoHotel, Whole, hotel)

EntityFacte(QuartoHotel, Part, quarto)

As asserções Class(Hotel), Class(Quarto) e Association(QuartoHotel) declaram a existência das classes *Hotel* e *Quarto*, e da associação *QuartoHotel* (i.e., quarto do hotel).

EntityFacet(QuartoHotel, association_type, Composition) especifica que a associação *QuartoHotel* é uma composição.

As duas asserções *Argument/3* especificam que a associação têm dois argumentos, um cujo papel é *hotel* e cujo tipo é *Hotel*, e outro cujo papel é *quarto* e cujo tipo é *Quarto*.

EntityFacet(QuartoHotel.hotel, Multiplicity, 1) significa que cada quarto tem exactamente um hotel.

EntityFacet(QuartoHotel.quarto, Multiplicity, 1..*) significa que um hotel tem um ou mais quartos.

Finalmente, EntityFacte(QuartoHotel, whole, hotel) e EntityFacte(QuartoHotel, part, quarto) especificam que o argumento cujo papel é *hotel* é o todo (i.e., o resultado da composição), e que o argumento cujo papel é *quarto* é a parte (i.e., a parte da composição).

Das duas facetas *whole* e *part* do exemplo, apenas uma delas seria necessária pois a associação é binária e, consequentemente, se um dos argumentos é a parte (por exemplo), o outro tem de ser o todo.

No próximo exemplo define-se o tipo de dados TIdade cujo valor mínimo é 0. Como um tipo de dados é um conjunto, o seu valor mínimo é especificado pela faceta smallest_instance.

Datatype(TIdade, Integer).

EntityFacet(TIdade, smallest_instance, 0).

Function(Idade, TIdade).

Ao declarar a função Idade com sendo do tipo TIdade, diz-se implicitamente que o menor valor retornado pela função é 0.

Um efeito semelhante poderia ter sido obtido, restringindo directamente o valor retornado pela função Idade, a qual recebe uma pessoa e devolve a sua idade em anos.

Function(Idade, Integer).

Argument(Idade, pessoa, Word).

EntityFacet(Idade, minimum_value, 0).

Como uma função de N argumentos denota um conjunto de túpulos de N+1 elementos, a função Idade denota o conjunto de pares Pessoa/Idade. A função smallest_instance aplicada a um conjunto de pares é obrigatoriamente um par. Consequentemente, em vez da faceta smallest_instance, deve ser usada a faceta minimum_value, a qual se aplica apenas a atributos, métodos e operadores e cuja semântica é definida para condicionar o valor do atributo, do método ou do operador.

Os exemplos e explicações apresentados não esgotam todas as possibilidades da linguagem. A próxima secção introduz novas possibilidades que serão exemplificadas ao longo do texto.

2 Notação, operadores e expressões da linguagem

Para além dos predicados usados nas asserções que descrevem uma ontologia, a linguagem CO3L dispõe de um conjunto de operadores adicionais com os quais se podem escrever expressões. As expressões podem também envolver operadores definidos em ontologias. Existem três tipos de expressões CO3L:

expressões relacionais, expressões funcionais e expressões de acção (estas últimas dizem respeito exclusivamente aos comandos da linguagem, hoje em dia, apenas o comando *import*).

As expressões funcionais descritas nesta secção podem ser usadas no lugar dos argumentos dos predicados das asserções de topo que descrevem a ontologia. Embora sem interesse prático, nas duas seguintes asserções, usa-se uma expressão funcional para indicar que o menor valor do tipo de dados T é 10.

Datatype(T, Natural).

EntityFacet(T, Smallest_instance, 8+2).

Em futuras versões da linguagem, quando esta for estendida com a potencialidade de representação de axiomas, as expressões relacionais e funcionais poderão ser usadas também na escrita de axiomas.

A linguagem inclui um conjunto de operadores muito gerais, incluindo os operadores aritméticos habituais (e.g., +, -, ×), os operadores relacionais habituais (e.g., =, ≠, >, ≥) e os operadores funcionais e relacionais definidos sobre colecções (e.g., ∩, ∪, ⊃, ⊇, ⊄, ∈, /). Todos estes operadores têm uma versão de tipo *"text book"* e uma versão computacional. As versões *text book* e computacionais de alguns operadores são iguais, as versões *text book* e computacionais de outros operadores são diferentes.

Alem destes, existem ainda outros operadores e notações:

Ponto (.) – Separa uma classe, uma associação ou uma instância de um atributo ou de um método. Separa igualmente uma relação de dependência, uma associação, um operador ou um método de um argumento. Finalmente, o operador ponto (.) separa também uma ontologia de uma das suas entidades (recursos).

Duplo dois pontos (::) – Separa um *namespace* mais geral de um dos seus sub espaços.

Cardinal (#) – Aplica-se ao identificador de um objecto composto para obter a sua definição. No caso de um objecto composto, a sua definição é o conjunto dos elementos que o constituem.

Ponto de exclamação (!) – Associa uma hierarquia e um tipo dessa hierarquia (e.g., uma classe) para especificar uma generalização

Parêntesis rectos ([...],]...], [...[,]...[) – Especifica intervalos, por exemplo, [7, 15]

Range – Contradomínio de uma função

Domain – Domínio de uma função, de um predicado, ou de uma associação

Evaluate (Eval) – Operador que se aplica a uma expressão e que serve para forçar a sua avaliação. Por exemplo, Eval(range(f)) é uma expressão funcional que devolve o contradomínio da função f, e range(f) usa-se para descrever / referir o contradomínio de f, mas não devolve esse contradomínio

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

Sequence, Set e Bag – Operadores n-ários construtores de colecções: sequências, conjuntos e sacos. Uma expressão formada com um operador de colecção é considerada uma constante, embora seja composta.

A linguagem será futuramente estendida com outros operadores ligados à escrita de axiomas.

As expressões da linguagem são expressões simples ou compostas. As expressões simples são constantes simples, por exemplo 5, Ana e "Vila Nova de Milfontes". As expressões compostas são formadas, usando parêntesis, ou a partir da combinação de outras expressões através da utilização de operadores da linguagem ou de operadores descritos numa ontologia, respeitando a sua sintaxe e significado.

Por defeito, as expressões funcionais da linguagem CO3L não são avaliadas automaticamente. Para serem avaliadas, é necessário usar o operador de avaliação (Eval/1). No entanto, há expressões cuja avaliação é igual à própria expressão: constantes simples (e.g., números, palavras), compostas (colecções especificadas por enumeração explícita dos seus elementos ou por intervalos), e nomes de entidades da própria linguagem (e.g., facetas, tipos de dados básicos, e nomes de operadores da linguagem) e definidas na ontologia (e.g., nomes de classes, nomes de tipos de dados básicos definidos através do predicado Datatype/2, e nomes de operadores e métodos). Tal como os atributos das classes ou das associações, por exemplo, as generalizações são identificadas por expressões compostas por duas partes. As generalizações são identificadas por expressões com o formato Hierarquia!Tipo. Enquanto que, sendo C uma classe e A um atributo de C, a expressão C.A pode ter um valor (se A for um atributo da classe e não das suas instâncias), o valor de uma generalização é a própria generalização, isto é Eval(Hierarquia!Tipo)

= Hierarquia!Tipo. Consequentemente, a caracterização de generalizações não envolve a aplicação do operador Eval/1.

Será que se pode especificar algumas circunstâncias onde as expressões são automaticamente avaliadas, sem usar o operador de avaliação, por exemplo em sítios em que se espera obrigatoriamente um valor, como na especificação do valor de uma faceta, ou do valor por defeito de um atributo?

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

Exemplos

Sendo E1 e E2 duas expressões funcionais, então E1 + E2 e Eval(E1 + E2) são expressões.

Se, na ontologia tiver sido definido a peseudo função current_date() que devolve a data actual, e a função year() que recebe uma data e devolve o ano dessa data, então a expressão year(current_date()) é uma expressão cujo valor é o valor do ano da data actual na altura em que a expressão é avaliada.

A especificação da linguagem tem de ser bem clara quanto à altura em que se dá a avaliação de expressões que têm de ser avaliadas. As expressões podem ser armazenadas na ontologia e serem avaliadas sempre que forem usadas. Nesse caso, é possível que a avaliação da mesma expressão em circunstâncias diferentes possa originar resultados diferentes, por exemplo a expressões cujo valor dependem do tempo (e.g., current_date()), e expressões cujo valor depende do valor de outras entidades que vão mudando (e.g., o valor de um atributo de um objecto). As expressões podem ser avaliadas apenas uma vez, numa circunstância bem definida, por exemplo, quando a ontologia é armazenada num servidor de ontologias. Nesse caso, torna-se impossível especificar valores que dependem das circunstâncias.

Por outro lado, não se pode passar para o servidor de ontologias, a responsabilidade de avaliar expressões que envolvam operadores definidos na ontologia (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.

2.1 Descrição de operadores e de notação

Operadores aritméticos binários infixos

As versões *text book* e computacionais são iguais excepto a potência e a multiplicação.

Se E1 e E2 forem expressões numéricas,

E1+E2 (soma), E1-E2 (subtracção), E1/E2 (divisão) são expressões numéricas

Se E1 e E2 forem expressões numéricas inteiras

E1//E2 (divisão inteira) e E1 mod E2 (resto da divisão inteira) são expressões numéricas inteiras

Potência

Se E1 e E2 forem expressões numéricas,

Versão *text book*: $E1^{E2}$ é uma expressão numérica

Versão computacional: E1^E2 é uma expressão numérica.

Multiplicação

Se E1 e E2 forem expressões numéricas,

Versão *text book*: E1×E2 é uma expressão numérica

Versão computacional: E1*E2 é uma expressão numérica.

Operadores aritméticos unários prefixos

Se E for uma expressão numérica, -E e +E são expressões numéricas. Os operadores unários prefixos + e - podem também aplicar-se ao símbolo especial da linguagem __oo__ (símbolo de infinito).

Operadores relacionais de ordem binários infixos

Se E1 e E2 forem expressões entre as quais seja possível estabelecer uma relação de ordem,

Versão *text book*: E1>E2, E1≥E2, E1<E2, E1≤E2 são expressões relacionais

Versão computacional: E1>E2, E1>=E2, E1<E2, E1=<E2 são expressões relacionais

Operadores de igualdade e de desigualdade binários infixos

Se E1 e E2 forem expressões,

Versão *text book*: E1=E2 e E1≠E2 são expressões

Versão computacional: E1=E2 e E1<>E2 são expressões

Operadores funcionais binários de colecções

Se C1 e C2 forem colecções, ou expressões que descrevem ou resultam em colecções,

Versão *text book*: C1∩C2 (intersecção), C1∪C2 (reunião), C1/C2 (diferença) são expressões que descrevem ou resultam em colecções

Versão computacional: cIntersect(C1, C2) (intersecção), cUnion(C1, C2) (reunião), cDiff(C1, C2) (diferença) são expressões que descrevem ou resultam em colecções

Exemplos

cIntersect(Set(a, b, c, d), Set(c, d, e, f)) = Set(c, d).

cUnion(Sequence(a, b, c, d), Sequence(c, d, e, f)) = Sequence(a, b, c, d, c, d, e, f)

cUnion(Set(a, b, c, d), Set(c, d, e, f)) = Set(a, b, c, d, e, f)

cDiff(Set(a, b, c), Set(a, b)) = Set(c)

Operadores relacionais binários de colecções

Se C1 e C2 forem colecções, ou expressões que descrevem ou resultam em colecções,

Versão *text book*: C1⊂C2 (C1 está contido em C2), C1⊆C2 (C1 está contido ou é igual a C2), C1⊃C2 (C1 contém C2), C1⊇C2 (C1 contém ou é igual a C2) são expressões relacionais

Versão computacional: contains(C1, C2) (C1 contém C2) é uma expressão relacional

Se C for uma colecção ou uma expressão que descreve ou resulta numa colecção, e E for uma constante ou expressão funcional,

Versão *text book*: E ∈ C (E pertence a C) e E ∉ C (E não pertence a C) são expressões relacionais

Versão computacional: member(E, C) (E pertence a C) é uma expressão relacional

Exemplos

Member(c, Set(a, b, c)) é verdade

Member(c, Bag(a, b, c, c, d)) é verdade

Ponto (.)

O ponto (.) separa uma classe, uma associação, uma relação de dependência, um indivíduo composto estruturado, um método ou um operador, de um atributo, de um argumento ou de um método. O operador ponto (.) separa também uma ontologia de uma das suas entidades (recursos).

Se E for o nome de um *classifier* (classe ou associação), de um objecto composto (por atributos ou argumentos e valores), de uma relação de dependência, de um método ou operador, ou descrever ou resultar num *classifier*, num objecto composto, numa relação de dependência, num método ou operador, e A for um atributo ou argumento ou descrever ou resultar num atributo ou argumento, então E.A é uma expressão que descreve um atributo ou argumento de E.

Se O for o nome de uma ontologia, ou descrever ou resultar no nome de uma ontologia, e E for uma entidade ou descrever ou resultar numa entidade (recurso) da ontologia O, então O.E é uma expressão que descreve a entidade (recurso) E da ontologia O.

Exemplos:

Pessoa.data_nascimento descreve o atributo data_nascimento da classe Pessoa (assumindo que Pessoa é uma classe)

PessoaCarro.proprietario descreve o argumento proprietário da associação PessoaCarro (assumindo que PessoaCarro é uma associação com um argumento com papel *proprietario*).

Se #x for uma instância da classe Pessoa, #x.Idade(2005/10/27) descreve a aplicação do método Idade com argumento 2005/10/27 à Pessoa #x.

Set(bi = 9987669, nome = "Ávila Duque").bi descreve o bi do objecto Set(bi = 9987669, nome = "Ávila Duque")

#p001.bi descreve o bi do objecto #p001

Se C for uma classe da ontologia O, O.C identifica a classe C especificando que pertence a O.

Duplo dois pontos ou quatro pontos (::)

O operador duplo dois pontos ou quatro pontos (::) serve para separar *namespaces* no nome da ontologia, no entanto para separar o nome da ontologia dos recursos que ela contém usa-se o operador ponto (.).

Se O for o nome de uma ontologia ou descrever ou resultar no nome de uma ontologia e R for uma das entidades (recursos) da ontologia, ou descrever ou resultar numa entidade da ontologia, então O.R é uma expressão que identifica R na ontologia O. Por outro lado, se O1 e O2 forem, descreverem ou resultarem em nomes de *Namespaces*, e se O2 estiver contido em O1, então O1::O2 é uma expressão que descreve o *namespace* O2 como subconjunto de O1. Assume-se que O2 só pode ser univocamente identificado recorrendo a O1.

Adicionalmente, se R for uma das entidades da ontologia O2 e se uma outra ontologia O3 também possuir uma entidade R, então R só pode univocamente identificado recorrendo à ontologia a que pertence (e.g. O1::O2.R ou O3.R). Esta identificação sem ambiguidade é deixada à responsabilidade do editor da ontologia.

Cardinal (#)

O operador cardinal (#) aplica-se ao identificador de um objecto composto para obter a sua definição, isto é, os seus atributos ou argumentos e valores correspondentes. Por exemplo, se tivermos o objecto Set(bi = 4537987, nome = "Ana Rita Alçada") cujo identificador é p001, então #p001 é uma abreviatura de Set(bi = 4537987, nome = "Ana Rita Alçada").

Assim, se Obj for o nome de um objecto ou resultar no nome de um objecto, #Obj é uma abreviatura de definição do objecto identificado por Obj.

Ponto de exclamação (!)

O ponto de exclamação (!) associa uma hierarquia e um tipo dessa hierarquia (e.g., uma classe) para especificar uma generalização.

Se H for o nome de uma hierarquia, ou descrever ou resultar no nome de uma hierarquia e T for um dos tipos dessa hierarquia, ou descrever ou resultar no nome de um tipo dessa hierarquia, então H!T é uma expressão que descreve uma generalização.

Exemplo

Consideram-se neste exemplo as seguintes duas hierarquias, assumindo que os tipos T, T1, T2, T3, T4, T1.1, T1.2, T1.3, T2.1 e T2.2 existem na ontologia:

Hierarchy(H1).

Subtype(H1, T, T1).
Subtype(H1, T, T2).
Subtype(H1, T1, T1.1).
Subtype(H1, T1, T1.2).
Subtype(H1, T1, T1.3).
Subtype(H1, T2, T2.1).
Subtype(H1, T2, T2.2).

Hierarchy(H2).

Subtype(H2, T1, T3).
Subtype(H2, T1, T4).

A Figura 3 mostra uma representação gráfica das duas hierarquias e quatro generalizações definidas pelas asserções do exemplo.

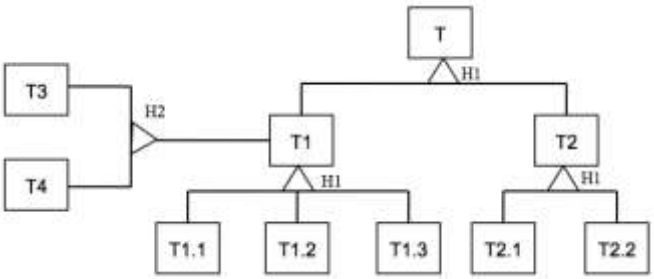

Figura 3 – Duas hierarquias, quatro generalizações

No exemplo, existem quatro generalizações, em que o termo generalização tem um significado semelhante ao conceito com o mesmo nome do UML. Uma generalização relaciona T com T1 e T2; outra generalização relaciona T1 com T1.1, T1.2 e T1.3; outra generalização relaciona T2 com T2.1 e com T2.2; finalmente, a quarta generalização relaciona T1 com T3 e T4.

Como a linguagem CO3L não tem generalizações com um nome único que as identifique, é necessário arranjar um mecanismo de identificação de generalizações recorrendo aos outros conceitos da linguagem. Se houvesse apenas uma hierarquia, cada generalização poderia ser identificada pelo nome do tipo mais geral. Por exemplo, a generalização que relaciona T1 com T1.1, T1.2 e T1.3 poderia ser identificada por T1. No entanto, T1 é o tipo mais geral de duas generalizações, por isso não basta o nome da classe mais geral para identificar generalizações. Em CO3L, uma generalização pode ser univocamente identificada pelo nome da hierarquia a que pertence e pelo nome da sua classe mais geral. Por exemplo, a generalização que relaciona T1 com T1.1, T1.2 e T1.3 é univocamente identificada por H1 e T1; e a generalização que relaciona T1 com T3 e T4 é univocamente identificada por H2 e T1.

A linguagem usa o operador ponto de exclamação (!) para associar uma hierarquia e um tipo para identificar uma generalização. No exemplo, a generalização que relaciona T com T1 e T2 é identificada pela expressão H1!T; a generalização que relaciona T1 com T1.1, T1.2 e T1.3 é identificada por H1!T1; a generalização que relaciona T2 com T2.1 e com T2.2 é identificada por H1!T2; e a generalização que relaciona T1 com T3 e T4 é identificada pela expressão H2!T1.

A avaliação de uma expressão de identificação de uma generalização é a própria expressão: Eval(H!T) = H!T.

Se pretendermos dizer que uma generalização é completa, por exemplo, podemos usar a faceta *Complete*:

EntityFacet(H1!T1, complete, true).

Se pretendermos dizer que uma hierarquia tem uma folha, podemos usar a faceta *Leaf*:

EntityFacet(H1, leaf, T1.1).
EntityFacet(H1, leaf, T1.2).
EntityFacet(H1, leaf, T1.3).
EntityFacet(H1, leaf, T2.1).
EntityFacet(H1, leaf, T2.2).

Analogamente ao que se passa no diagrama de classes do UML, o modelo O3F permite fazer inferências baseadas nas generalizações.

Range/1

Range/1 é um operador funcional que descreve o contradomínio de uma função ou de um método funcional.

Se F for o nome de uma função, ou descrever ou resultar no nome de uma função, então Range(F) é uma expressão que descreve o contradomínio de F.

Se C for o nome de um *classifier* ou uma instância de um *classifier*, ou descrever ou resultar no nome de um *classifier* ou numa instância de um *classifier*, e M for o nome de um método funcional, ou descrever ou resultar no nome de um método funcional, então Range(C.M) é uma expressão que descreve o contradomínio de M.

Domain/1

Domain/1 é um operador funcional que descreve o domínio de uma função, de um predicado, ou de uma associação.

Se O for o nome de uma função, de um predicado ou de uma associação, ou descrever ou resultar no nome de uma função, de um predicado ou de uma associação, então Domain(O) é uma expressão que descreve o domínio de O.

Se M for o nome de um método funcional, de um método relacional, de um método de acção ou de uma associação, ou descrever ou resultar no nome de um método, e C for uma classe ou associação ou uma expressão que descrever ou resultar no nome de uma classe ou associação, então Domain(C.M) é uma expressão que descreve o domínio de C.M.

Evaluate (Eval/1)

Eval/1 é um operador funcional que se aplica a uma expressão e que serve para forçar a sua avaliação.

Se E for uma expressão, então Eval(E) é uma expressão que devolve o valor da expressão E.

Exemplos:

Sendo f o nome de uma função, Range(f) descreve o contradomínio de f, e Eval(range(f)) é o contradomínio de f.

Sendo C uma classe e A um atributo dessa classe, C.A descreve o atributo A da classe C, Eval(C.A) é o valor do atributo A de C.

Sendo O uma instância de uma classe e A um atributo dessa classe, O.A descreve o atributo A de O, e Eval(O.A) é o valor do atributo A de O.

Construtores de colecções

Sequence, Set e Bag são operadores n-ários construtores de sequências, conjuntos e sacos. As colecções construídas podem ter elementos de qualquer tipo ou de vários tipos. Os argumentos dos operadores construtores de colecções são os elementos dessas colecções.

Sendo Ei uma constante ou expressão funcional, então

Sequence(E1, E2, ..., En) é a sequência de E1, seguido de E2, seguido de ..., seguido de En;

Set(E1, E2, ..., En) é o conjunto formado por E1, E2, ..., En; e

Bag(E1, E2, ..., En) é o saco formado por E1, E2, ..., En.

Sequence(), Set() e Bag() são colecções vazias.

Uma sequência e um saco podem ter elementos repetidos.

Um conjunto não tem elementos repetidos.

Numa sequência, a ordem é importante. Os conjuntos e os sacos não têm ordem.

Na versão actual da linguagem CO3L, os operadores construtores de colecções são avaliados, mas futuramente, tem de se decidir se eles devem ou não ser automaticamente avaliados sem a utilização do operador de avaliação, especialmente quando são definidos por compreensão.

Na versão actual da linguagem CO3L, as colecções apenas podem ser construídas por enumeração ou resultar de operações funcionais de outras colecções. Futuramente, tem de se decidir se a linguagem deve permitir a construção de colecções por compreensão e como.

Intervalos

Intervalos são sequências potencialmente infinitas. Podem definir-se intervalos sobre qualquer tipo de dados para o qual se possa estabelecer uma relação de ordem, por exemplo números, caracteres, e palavras.

Parêntesis rectos ([...],]...], [...[,]...[) especificam intervalos, por exemplo, [7, 15]

[Limite inferior, Limite superior] – intervalo fechado limitado inferiormente por *Limite inferior* e superiormente por *Limite superior*.

]Limite inferior, Limite superior] – intervalo aberto à esquerda limitado inferiormente por *Limite inferior* (exclusive) e superiormente por *Limite superior* (inclusive).

[Limite inferior, Limite superior[– intervalo aberto à direita limitado inferiormente por *Limite inferior* (inclusive) e superiormente por *Limite superior* (exclusive).

]Limite inferior, Limite superior[– intervalo aberto limitado inferiormente por *Limite inferior* (exclusive) e superiormente por *Limite superior* (exclusive).

Num intervalo, tem de se verificar a restrição *Limite inferior* < *Limite superior* em que o sinal < representa a relação de ordem estabelecida para o tipo de dados em que o intervalo foi definido.

Sintacticamente, Limite inferior = Limite superior = Limite

O Limite é uma constante (do tipo de dados em que o intervalo se define) ou uma expressão cuja avaliação resulta numa constante desse tipo. Existe, na linguagem, a constante especial __oo__ que significa infinito e pode ser usada para especificar intervalos não limitados.

Exemplos:

[a, f] – sequencia a, b, c, d, e, f

]7, 10.7] – Intervalo que vai do 7, mas não o incluí, até ao 10.7 inclusive

[-8, __oo__[– Intervalo desde o -8 (inclusive) sem limite superior.

]-__oo__, -10.7[– Intervalo sem limite inferior e que termina em -10.7 exclusive.

Sempre que o extremo de um intervalo é infinito, o intervalo tem obrigatoriamente de ser aberto nesse extremo.

Notas:

- *É natural que me tenha esquecido de operadores existentes.*
- *Nesta versão da linguagem, enquanto não houver uma linguagem para representar axiomas e para especificar conjuntos por compreensão, não serão necessários operadores relacionais.*

2.2 Sintaxe geral das expressões

Esta secção apresenta uma versão simplificada da sintaxe das expressões da linguagem CO3L. A simplificação feita nesta gramática livre de contexto tem por consequência a impossibilidade de efectuar todas as verificações sintácticas possíveis. Por exemplo, um argumento não pode ser uma expressão relacional, mas esta gramática diz apenas que um argumento pode ser uma expressão; não há distinção entre expressões funcionais e expressões relacionais.

Além de simples, a sintaxe apenas diz respeito à versão computacional da linguagem.

Uma expressão pode ser simples ou composta:

```
Expression → SimpleExpression | CompoundExpression.

SimpleExpression → Constant.

CompoundExpression →
   "(" Expression ")" |
   SignOperator Expression |
   PrefixOperator "(" Arguments ")" |
   Expression InfixOperator Expression |
   IntervalExpression.
```

```
SignOperator → "-" | "+".

PrefixOperator → "Eval" | "cUnion" | "cIntersect" | "cDiff" | "#"...

InfixOperator → "+" | "-" | "+" | "*" | "/" | "//" | "mod" | "^".
InfixOperator → "=" | "<>" | ">" | ">=" | "<" | "=<".
InfixOperator → ".." | "!" | "::" | "+" .....

Arguments → Expression.
Arguments → Expression "," Arguments.

IntervalExpression →
   "[" LowerLimit, UpperLimit "]" |
   "[" LowerLimit, UpperLimit "[" |
   "]" LowerLimit, UpperLimit "]" |
   "]" LowerLimit, UpperLimit "[".

LowerLimit → Expression.

UpperLimit → Expression.

Constant → "__oo__" | "*" | Number | String | Word | Char | Date ....
```

Notas:

Deveria distinguir-se diversos tipos de expressão, pelo menos, as expressões relacionais e as expressões funcionais, e as expressões de acção (por enquanto, existe apenas a acção import). Além desta divisão formal, talvez as expressões também tenham de ser divididas de acordo com o seu conteúdo (tipo dos seus argumentos e eventualmente dos valores retornados).

Nesta versão da linguagem, enquanto não houver uma linguagem para representar axiomas e para especificar conjuntos por compreensão, não serão necessárias expressões relacionais.

3 Sumário da linguagem CO3L

3.1 Secções da descrição da ontologia

Uma descrição de uma ontologia em CO3L tem três secções, para além da indicação do nome da ontologia: A secção de parâmetros, a secção de comandos, e a secção de asserções de definições. A primeira secção poderá incluir os seguintes parâmetros: *Owner*, *Initial_date*, *Last_modification_date* e *Location*.

A segunda secção poderá apenas incluir um ou mais comandos de importação: import(OntologyName, OntologyLocationSequence). *OntologyName* é o nome lógico da ontologia, o qual a identifica univocamente. *OntologyLocationSequence* é a localização ou a sequência de localizações onde a ontologia pode ser acedida. Uma localização é especificada por uma URL.

A terceira secção é constituída por um conjunto de asserções que declaram a existência de entidades básicas da ontologia e de relações estruturais entre elas. As entidades básicas incluem as seguintes: Datatype, Class, Association, Hierarchy, relações de dependência (Dependency) símbolo proposicional (PropSymbol) Predicate, Function, Action, Type, Facet, ValidFacetElement, ValidFacetType, Axiom, e definição de um objecto (ObjectDef). Apesar de serem entidades básicas, algumas destas têm de se relacionar obrigatoriamente com outras. Por exemplo, um tipo de dados básico novo tem de ser definido à custa de outro tipo de dados básico. As relações entre e as propriedades das entidades da ontologia incluem os, mas não se limitam a atributos e métodos de classes e de associações, os argumentos de funções, predicados, acções, métodos funcionais, métodos relacionais, métodos de acção, os argumentos de associações, argumentos de relações de dependência, as particularizações de um dado tipo de dados, e instâncias de tipos dados. As facetas e os axiomas estabelecem outras relações, propriedades e restrições relativas às entidades da ontologia.

Nome da ontologia

O nome da ontologia deve ser um nome único que possa ser usado na sua identificação unívoca. Usa-se a notação dos namespaces, por exemplo PT::ISCTE-IUL::Ontologias::Alunos, e EDU::MIT::Ontologies::Software.

Parâmetros da definição de Ontologia

Parâmetro	Explicação	Exemplo
Owner	Responsável pela criação e manutenção da ontologia. Se houver mais que um responsável, o valor é um conjunto	Owner : Set("ADETTI", "ISCTE", "IT")
Initial_date	Data em que a ontologia começou a ser feita	Initial_date : 2004/05/02
Last_modification_date	Data em que a ontologia foi modificada pela última vez	Last_modification_date : 2004/05/22
Location	Localização de uma (cópia) da ontologia. Se a ontologia puder ser consultada em localizações alternativas, o valor é uma sequência	Location : "iscte.pt/~luis/Ontologias/Bibliotecas.txt"

Tabela 1 – Parâmetros de definição de ontologia

Nenhum dos parâmetros da ontologia (Tabela 1) é obrigatório, e podem ser escritos em qualquer ordem desde que apareçam antes das asserções de definição. O nome dos parâmetros pode escrever-se com letras maiúsculas ou minúsculas.

Asserções de Definição da Ontologia

As asserções que definem a ontologia podem ser agrupadas em dois tipos: declarações e composições. As declarações limitam-se a especificar a existência de uma dada entidade, por exemplo, uma classe, uma associação, ou um tipo de dados. As composições especificam relações estruturais dos elementos da ontologia.

Tem que existir pelo menos uma asserção de definição da ontologia. Havendo mais que uma, as asserções que definem a ontologia podem ser especificadas por qualquer ordem. Os predicados podem ser escritos com letras maiúsculas ou minúsculas.

A linguagem CO3L pode definir atributos de uma classe ou de uma associação (i.e., de um classificador – *classifier* em inglês) através da definição directa do atributo nesse classificador, ou através de uma relação entre o classificador e uma função. Igualmente, a definição de um método pode ser feita directamente na classe ou associação a que o método pertence ou através de uma relação entre o classificador e um operador (i.e., funcção, predicado, ou acção). As seguintes asserções definem directamente o atributo *preço* da associação *DVD_Loja*:

Association(DVD_Loja).

Attribute(DVD_Loja, preço, Float).

As asserções que se seguem criam o atributo *marca* da classe *Carro* através de uma relação entre *Carro* e a função *marca_de_carro*, a qual recebe um carro e devolve a sua marca:

Class(Carro).

Function(marca_de_carro, String).

Argument(marca_de_carro, carro, Carro).

AttributeFunction(Carro, marca, marca_de_carro).

Embora não seja obrigatório, a classe e a função envolvidas na definição de um dado atributo costumam ser declaradas antes do atributo, para facilitar a leitura.

3.2 Asserções da linguagem

A apresentação das asserções da linguagem organiza-se nas seguintes secções:

- Definição de tipos de dados
- Classifiers: classes e associações
- Operadores e símbolos proposicionais

- Relação entre classes e operadores
- Hierarquias
- Indivíduos
- Relações de dependência
- Facetas e axiomas

Definição de tipos de dados

Predicado	**Papel**	**Exemplo / Descrição**
Datatype/2	Tipo de dados	*Datatype(numero_de_bi, Natural).* Declara a existência de um tipo para representar números de bilhete de identidade, o qual é ou define-se à custa do tipo básico *Natural*.

Classifiers: classes e associações

Predicado	**Papel**	**Exemplo / Descrição**
Class/1	Classe	*Class(Pessoa).* Declara a classe chamada *Pessoa*
Association/1	Associação	*Association(CarroPessoa).* Declara a associação chamada *CarroPessoa*
Attribute/3	Atributo de um classifier (classe ou associação)	*Attribute(Pessoa, dataNascimento, Date). Define o atributo chamado dataNascimento que associa a classe Pessoa com o tipo de dados (datatype) Date*
Argument/3	Argumento de um operador, de um método ou de uma associação	*Argument(CaoPessoa, dono, Pessoa)* *Argument(CaoPessoa, cao, Cao)* *Define os dois argumentos da associação CaoPessoa*
Key/3	Mecanismo de identificação	*Key(Classifier, KeyName, Attribute). O atributo Attribute faz parte do mecanismo de identificação KeyName do classifier Classifier*
FunctionalMethod/3	Método funcional de um Classifier	*FunctionalMethod(Pessoa, idade, Natural). Define o método funcional idade da classe Pessoa, o qual devolve um natural que representa a idade da pessoa*
RelationalMethod/2	Método relacional de um Classifier	*RelationalMethod(ParPontos, percurso). Define o método relacional percurso da associação ParPontos. Seria usado para representar um ou mais percursos entre dois pontos num grafo. Sabe-se que ParPontos é uma associação pela asserção que a declara.*
ActionMethod/2	Método de acção de um Classifier	*ActionMethod(Hotel, reservarQuarto). Declara o método de acção reservarQuarto da classe Hotel.*

Operadores e símbolos proposicionais

Predicado	**Papel**	***Exemplo / Descrição***
PropSymbol/1	Símbolo proposicional	*PropSymbol(server_not_available).* Declara o símbolo proposicional *server_not_available*, o qual pode ter o valor verdade ou falso.
Predicate/1	Predicado	*Predicate(melhor_preco).* Declara o predicado *melhor_preco*.
Function/2	Função	*Function(idade_da_pessa, Natural).* Declara a função *idade_da_pessoa*, a qual retorna um valor natural.
Action/1	Acção	*Action(reset_printer).* Declara a acção *reset_printer*.
Argument/3	Argumento de um operador, de um método ou de uma associação	*Argument(reservar_mesa, numero_de_pessoas, integer).Define o argumento numero_de_pessoas de tipo inteiro do operador de acção reservar_mesa*

Relação entre classes e operadores

Predicado	**Papel**	**Exemplo / Descrição**
AttributeFunction/3	Definição de um atributo de um *Classifier* através de uma função	*AttributeFunction(Pessoa, dataNascimento, data_nascimento).* A classe Pessoa tem o atributo dataNascimento, o qual se define à custa da função data_nascimento.
AMethodAction/4	Definição de um método de acção de um Classifier à custa de uma acção.	*AMethodAction(Restaurante, reservarMesa, ResevarMesa, restaurante).* A classe *Restaurante* tem o método de acção *reservarMesa* que se define à custa da acção *ReservarMesa*. O objecto da classe *Restaurante* a que o método *reservarMesa* se aplica desempenha o papel *restaurante* da acção *ResevarMesa*.
FMethodFunction/4	Definição de um método de funcional de um Classifier à custa de uma função.	*FMethodFunction(Pessoa, idade, Idade, pessoa).* A classe *Pessoa* tem o método funcional *idade* que se define à custa da função *Idade*. O objecto da classe *Pessoa* a que o método *idade* se aplica desempenha o papel *pessoa* da função *Idade*.
RMethodPredicate/4	Definição de um método relacional de um Classifier à custa de um predicado.	*RMethodPredicate(Pessoa, descendente, Descendent, pessoa).* A classe *Pessoa* tem o método relacional *descendente* que se define à custa do predicado *Descendente*. O objecto da classe *Pessoa* a que o método *descendente* se aplica desempenha o papel *pessoa* do predicado *Descendente*.

Hierarquias

Predicado	**Papel**	**Exemplo / Descrição**
Hierarchy/1	Hierarquia	*Hierarchy(ClassesDeAnimais)*
Subtype/3	Particularização do tipo especificado, de acordo com a hierarquia especificada	*Subtype(DocumentType, Document, WordDocument).* De acordo com a hierarquia chamada *DocumentType*, a classe *WordDocument* é uma subclasse de *Document*.

Indivíduos

Predicado	**Papel**	**Exemplo / Descrição**
ObjectDef/2	Indivíduo	ObjectDef(p001, Set(bi=8265101, nome = “Che Gutierres”)) ObjectDef(Dobro4, Set(numero=4, dobro=8))
Instance/2	Instância de um tipo de dados qualquer, em particular de um classifier.	Instance(#p001, Pessoa). p001 tem de ser definido por uma asserção do predicado ObjectDef/2. Supondo que p001 é o identificador do objecto composto estruturado Set(bi = 6798268, nome = “Katya Vanessa”, banda_preferida = “Abba”), a asserção de Instance/2 significa que Set(bi = 6798268, nome = “Katya Vanessa”, banda_preferida = “Abba”) é uma Pessoa.

Relações de dependência

Predicado	**Papel**	**Exemplo / Descrição**
Dependency/1	Relação de dependência	Dependncy(DependencyRelation). Declara a existência de uma relação de dependência com o no e especificado
DependencyArgument/3	Argumento de uma relação de dependência	DependencyArgument(InvocaMétodo, classe_geral, C2). C2 faz o papel de classe_geral da relação de dependência InvocaMétodo.

Facetas e axiomas

Predicado	**Papel**	**Exemplo / Descrição**
Facet/1	Faceta	*Facet(cor).* Declara a nova faceta *Cor*.
ValidFacetType/2	Valor da faceta	*ValidFacetType(Cor, Word)*. Os valores válidos da faceta Cor são palavras.
ValidFacetElement/2	Elementos a que a faceta se aplica	*ValidFacetElement(Cor, Attribute).* A faceta Cor pode aplicar-se a atributos.
EntityFacet/3	Faceta de uma entidade	*EntityFacet(Vogal, Instances_Sequence, Sequence(a, e, i, o, u)).* A sequência de valores do tipo *Vogal* é *a*, *e*, *i*, *o*, *u*. Isso significa também que *a* é menor que *e*, o qual é menor que *i*, ... A seguinte expressão associa a faceta *minimum_value* com valor 1 ao argumento *numero_de_pessoas* do operador *reservar_mesa*. *EntityFacet(reservar_mesa.numero_de_pessoas, minimum_value, 1).* A seguinte expressão associa a faceta *minimum_value* com valor 1 ao argumento *data* do método *idade* da classe *Pessoa*. *EntityFacet(Pessoa.idade.data, minimum_value, 1).*
Axiom/2	Axioma (A linguagem de definição de axiomas ainda não foi criada; o exemplo apresentado serve apenas para dar uma ideia)	*O seguinte axioma, chamado defIdade, define a função idade, a qual se aplica a objectos da classe Pessoa. A definição do axioma não o associa a uma classe ou associação.* *Axiom(* *defIdade,* *forall(?x,* *implies(* *instance(?x, Pessoa)* *idade_da_pessoa(?x) =* *year_diff(data_actual(), ?x.DataNascimento)))* *)*

3.3 Operadores adicionais

Alem das acções usadas nos comandos que permitem manipular ontologias ou partes delas (actualmente, apenas a acção *import*) e dos predicados usados nas asserções de topo que descrevem uma ontologia, a linguagem CO3L dispõe de um conjunto de operadores adicionais com os quais se podem escrever expressões relacionais e funcionais. As expressões funcionais podem ser usadas no lugar de argumentos dos predicados usados nas asserções de topo que descrevem a ontologia. Futuramente, os operadores relacionais e funcionais serão também usados na escrita de axiomas.

Esta secção descreve sumariamente todos os operadores adicionais da linguagem. Embora, os operadores tenham uma sintaxe tipo *text book* e uma sintaxe computacional, as tabelas seguintes recorrem apenas à sintaxe computacional.

Operadores Funcionais

E1+E2, E1–E2, E1*E2, E1/E2, E1//E2, E1 mod E2, E1^E2 (E1 e E2 são expressões numéricas)	Operadores aritméticos habituais binários, infixos. E1//E2 (divisão inteira) E1 mod E2 (E1 módulo E2, ou resto da divisão inteira de E1 por E2) E1^E2 (potência: E1 elevado a E2)
-E, +E (E é uma expressão numérica)	Operadores numéricos prefixos unários. Os operadores unários prefixos + e – podem aplicar-se também ao símbolo especial __oo__ que representa o conceito matemático abstracto infinito.
cIntersect(C1, C2) (intersecção) cUnion(C1, C2) (reunião) cDiff(C1, C2) (diferença) (C1 e C2 são colecções)	Operadores funcionais binários, prefixos de colecções
Range(F) Range(C.M)	Contradomínio da função F ou do método funcional M do *classifier* C
Domain(O) Domain(C.M)	Domínio da função, predicado, ou associação O Domínio do método funcional ou relacional M do *classifier* C
Eval(Expression)	Evaluate. Operador que força a avaliação da expressão *Expression*. Por exemplo, Eval(Range(F)).
#ObjectName	Forma abreviada da definição do indivíduo designado por ObjectName
Sequence(E1, E2, ..., En) Set(E1, E2, ..., En) Bag(E1, E2, ..., En) (Os Ei são os elementos das colecções)	Operadores prefixos, n–ários, construtores de colecções: sequências, conjuntos e sacos. Sequence(), Set() e Bag() são colecções vazias.
[Li, Ls],]Li, Ls], [Li, Ls[,]Li, Ls[	Intervalos são sequências. Li e Ls são os limites inferior e superior do intervalo, respectivamente.
C.A, C.M, O.R	Ponto (.). Operador que separa o nome de um *classifier* ou de um indivíduo de um atributo ou de um método. Também separa o nome de uma associação ou uma sua instância do seus argumentos. Finalmente, separa o nome de uma ontologia de uma das suas entidades (ou recursos)
O::R	O1::O2 refere o *Namespace* O2 (subconjunto de O1) relativamente a O1. Assume-se que O2 só pode ser univocamente identificado com relação a O1.
H!T	Ponto de exclamação (!) Operador que especifica a generalização que se define por uma hierarquia e um tipo

Tabela 2 – Operadores Funcionais

Operadores Relacionais

E1>E2, E1>=E2, E1<E2, E1=<E2 (E1 e E2 são expressões entre as quais é possível estabelecer uma relação de ordem)	Operadores relacionais de ordem binários infixos
E1=E2 e E1<>E2 (E1 e E2 são expressões)	Operadores de igualdade e de desigualdade binários infixos
contains(C1, C2) (C1 contém C2) member(E, C) (E pertence a C)	Operadores relacionais binários de colecções

Tabela 3 – Operadores Relacionais

3.4 Facetas da linguagem CO3L

Existem várias facetas na linguagem CO3L, as quais não são dependentes do domínio. De seguida, apresentam-se as facetas existentes na linguagem, organizadas pela categoria de entidades a que se podem aplicar. Do ponto de vista da aplicação de facetas, existem as seguintes categorias de elementos:

- Facetas de todas as entidades
- Facetas de conjuntos
- Facetas de *Classifiers*
- Facetas de hierarquias e generalizações
- Facetas de atributos, argumentos, operadores e métodos

Facetas de todas as entidades

Faceta	**Descrição**	**Valores Possíveis**
Visibility	Indica a visibilidade de entidades da ontologia. Entidades públicas são globalmente visíveis. Quando a visibilidade tem o valor *Ontology*, a entidade é visível apenas na ontologia a que pertence. Uma entidade privada só é visível no *namespace** mais restritivo a que pertence. O valor *Protected* aplica-se apenas a atributos e argumentos. Uma entidade é visível no *namespace* mais restritivo a que pertence, nos subtipos do tipo definidor do *namespace*, e na ontologia a que pertence.	Private/Protected/Ontology/Public Por omissão assume-se o valor public
Access	Indica o tipo de acesso ao valor de uma entidade.	Read_only, final, read_write....
Dependence	Indica se uma entidade é obtida a partir de outras entidades, se depende de outras entidades, mesmo que não se obtenha a partir delas, ou se é independente. Entidades (por exemplo, atributos) *Derived* são obtidos a partir de outras entidades. Entidades *Derived_union* são conjuntos obtidos pela união de outros conjuntos. Entidades *Dependent*, ainda que não se possam calcular a partir de outras, são dependentes delas. Entidades *Independent* não são determinados a partir de outras entidades.	Derived / Derived_union / Dependent /Independent. Por omissão, assume-se o valor Independent. A faceta Dependence nada diz sobre a natureza exacta da relação de dependência. Por exemplo EntityFacet(Class.a, Dependence, Derived) significa apenas que Class.a se obtém a partir de outros elementos da ontologia. Mas não se sabe se é o conjunto denotado por Class.a que deriva de outros elementos ou se é cada um dos seus elementos que é derivado, ou se é o valor do atributo Class.a que é derivado de outras entidades.
Owner	Especifica a classe, a associação, ou qualquer outro tipo de dados a que uma dada entidade da ontologia pertence (por exemplo, um axioma ou uma classe)	Class(C1) Class(C2) EntityFacet(C2, Owner, C1) A classe C2 pertence (está embebida) na classe C1. Se *Owner* não for especificado, a entidade pertence apenas à ontologia.

* *Namespace*. Um *Namespace* permite desambiguar as designações de entidades. Uma ontologia define um namespace. As classes, associações, operadores, métodos e relações de dependência definem *namespaces*. Por exemplo, um argumento de um operador só pode ser univocamente designado pela composição do operador a que pertence (o seu namespace) com a sua designação no namespace onde foi definido.

Facetas de conjuntos

Faceta	**Descrição**	**Valores Possíveis**
Largest_instance	Aplica-se a conjuntos, incluindo atributos e métodos. A entidade a que a faceta se aplica é interpretada como conjunto. *Largest_instance* estabelece a maior instância do conjunto, de acordo com a relação de ordem vigente.	O tipo de dados da faceta depende do conjunto a que a faceta está associada. Não tem valor por defeito.
Smallest_instance	Estabelece a menor instância do conjunto, de acordo com a relação de ordem vigente.	O tipo de dados da faceta depende do conjunto a que a faceta está associada. Não tem valor por defeito
Instance_maximum_size	Aplica-se a conjuntos, incluindo atributos e métodos. A entidade a que a faceta se aplica é interpretada como conjunto. Comprimento máximo de uma instância do conjunto a que é aplicada. O significado de "comprimento" depende do tipo de dados considerado. O comprimento de colecções (e.g., bag) é o número de elementos da colecção. O comprimento de uma cadeia de caracteres é o seu número de caracteres. O comprimento de um número é o número de algarismos do número. O comprimento de uma instância de uma classe é o número de pares atributo/valor da instância.	Número inteiro não negativo. Não tem valor por defeito.
Instance_minimum_size	Comprimento mínimo de uma instância do conjunto a que é aplicada.	Número inteiro não negativo. Não tem valor por defeito.
Default_instance	Aplica-se a conjuntos, incluindo atributos e métodos. A entidade a que a faceta se aplica é interpretada como conjunto. Indica a instância mais representativa do conjunto a que se aplica.	O tipo de dados da faceta depende do tipo de dados do conjunto a que ela está associada
Instances_set	Aplica-se a conjuntos, incluindo atributos e métodos. A entidade a que a faceta se aplica é interpretada como conjunto. Especifica o conjunto de instâncias de um conjunto.	Conjunto enumerado de valores possíveis (e.g., set(on, off)).
Instances_sequence	Especifica o conjunto de instâncias de um conjunto, e estabelece uma relação de ordem entre elas.	Sequência de valores possíveis. Por exemplo, sequence(baixo, médio, alto) significa que as instâncias do conjunto são *baixo*, *médio*, *alto* e que, baixo < médio < alto, de acordo com a relação de ordem definida.

Facetas de Classifiers

Faceta	**Descrição**	**Valores Possíveis**
Materialization	Especifica se uma classe é abstracta, concreta, ou interface. No âmbito da interacção entre agentes, não faz sentido falar de interfaces	Abstract/Concrete/Interface Por omissão, assume-se o valor *concrete*
Realization	Especifica que uma classe implementa uma dada interface. No âmbito da interacção entre agentes, não faz sentido falar de interfaces	Nome da interface implementada
Process_control	Um *classifier* "active" detém o controlo da sua actividade, podendo iniciá-la. Um classifier "passive" armazena dados e serve outros classifiers.	Passive / Active. Por defeito, assume-se o valor passive.
Association_type	Especifica o tipo da associação	Association, Aggregation, Composition
Whole	Especifica o argumento de uma composição ou de uma agregação que resulta da composição das suas partes	Nome do papel (role) de um dos argumentos da associação
Part	Especifica o argumento de uma composição ou de uma agregação que faz o papel de parte do todo.	Nome do papel (role) de um dos argumentos da associação

Facetas de hierarquias e generalizações

Faceta	**Descrição**	**Valores Possíveis**
Root	Aplica-se a uma hierarquia e especifica a sua raiz (o nó da hierarquia do qual todos os outros descendem)	Nome de um tipo de dados
Leaf	Serve para especificar os tipos folha (não decomponíveis) de uma dada hierarquia.	Nome de um tipo de dados Por omissão, assume-se que os nós de uma hierarquia não são folhas
Complete	Faceta das generalizações, indica se qualquer instância do tipo pai tem que ser forçosamente uma instância de um tipo particular.	Booleano. *True*, significa que as instâncias do supertipo são forçosamente instâncias dos seus subtipos; e *False* significa que o supertipo pode ter instâncias que não pertencem a nenhum dos seus subtipos. Por omissão, assume-se o valor *False*

Facetas de atributos, argumentos, operadores e métodos

Faceta	**Descrição**	**Valores Possíveis**
Maximum_value	Indica o valor máximo de atributos, argumentos e valores de retorno de operadores e métodos. Não tem valor por defeito.	O tipo de dados do valor máximo depende do tipo de dados da entidade a que a faceta está associada
Minimum_value	Indica o valor mínimo de atributos, argumentos e valores retornados de operadores e métodos. Não tem valor por defeito.	O tipo de dados do valor mínimo depende do tipo de dados da entidade a que a faceta está associada
Maximum_size	Comprimento máximo de um atributo, argumento, ou valor de retorno de operadores e métodos. O significado de "comprimento" depende do tipo de dados considerado. O comprimento de colecções (e.g., bag) é o número de elementos da colecção. O comprimento de uma cadeia de caracteres é o seu número de caracteres. O comprimento de um número é o número de algarismos do número. O comprimento de uma instância de uma classe é a soma de comprimentos dos atributos da classe a que a instância pertence.	Número inteiro não negativo
Minimum_size	Comprimento mínimo de um atributo, argumento, ou valor de retorno de operadores e métodos.	Número inteiro não negativo
Default_value	Indica o valor assumido em caso de omissão. Associa-se com atributos, argumentos e valores retornados de operadores e métodos.	O tipo de dados do valor assumido por defeito depende do tipo de dados da entidade a que a faceta está associada
Values_set	Especifica o conjunto de valores possíveis de um atributo, de um argumento ou de um valor retornado	Conjunto enumerado de valores possíveis (e.g., set(on, off))
Values_sequence	Especifica o conjunto de valores possíveis de um atributo, de um argumento, ou de um valor retornado e a sua relação de ordem.	Sequência de valores possíveis, e respectiva relação de ordem. Por exemplo, sequence(baixo, médio, alto) significa que os valores podem ser um dos elementos *baixo*, *médio*, *alto* e que, baixo < médio < alto, de acordo com a relação de ordem definida.

Facetas de atributos, argumentos, operadores e métodos

Faceta	**Descrição**	**Valores Possíveis**
Mandatory	Indica que um atributo de um *classifier* ou um argumento de operador é obrigatório.	Booleano. *True*, significa obrigatoriedade; e *false* significa não obrigatoriedade *False* é o valor por omissão para atributos. *True* é o valor por omissão para argumentos
Distinct	Indica que um determinado atributo ou argumento de uma determinada instância, se existir, tem um valor que é diferente do valor do mesmo atributo ou argumento de outras instâncias. Se o atributo ou argumento for obrigatório, funciona de identificador.	Booleano. *True*, significa que o atributo tem um valor sempre distinto; e *false* significa que o atributo não tem valor distinto. Por omissão, assume-se o valor *False*.
Scope	Indica se um atributo é de um *classifier* (classe ou associação) ou de uma instância	Classifier / Instance. Por omissão, assume-se *Instance*.
Navigability	Aplica-se a argumentos de associações e de relações de dependência. Especifica se um argumento de um relacionamento pode ser acedido a partir dos outros argumentos do mesmo relacionamento. *[O significado desta faceta não está estabilizado]*	Navigable / NonNavigable / NonSpecified. Se um argumento é navegável, ele pode ser acedido a partir dos outros argumentos do relacionamento. Por omissão, a faceta tem o valor NonSpecified.
Return_type	Faceta que se aplica a acções e métodos de acção, se estes devolverem um valor	*Word*. O valor da faceta return_type é um tipo de dados qualquer
Return_role	Faceta que se aplica a acções, métodos de acção se estes devolverem um valor, e a funções e métodos funcionais se for desejado especificar o papel desempenhado pelo valor retornado. Se o *return role* não for especificado para uma função ou método funcional, assume-se que o papel desempenhado pelo valor devolvido é o nome da acção ou do método funcional.	*Word.* Palavra que descreve o papel desempenhado por um valor retornado.
Arg_direction	Especifica a direccionalidade de um argumento de um método ou de um operador. Só é aplicável a métodos e operadores relacionais e os métodos e operadores de acção com interface relacional.	*In* para especificar que o argumento serve para passar valores a um método, *Out* para especificar que o argumento serve para devolver valores calculados num método, *Inout* para especificar que o argumento serve ambos os propósitos. Por omissão assume-se o valor *In*
Arg_number	Especifica o número de ordem de um argumento de um operador ou de um método, através do seu papel	Natural
Multiplicity	Especifica a multiplicidade de um argumento	As mesmas possibilidades que nos diagramas de classes do UML

3.5 Tipos de dados incluídos no CO3L

A linguagem CO3L dispõe de um conjunto razoável de tipos de dados básicos (Tabela 4). Esses tipos não se declaram na ontologia. Outros tipos necessários no domínio da aplicação têm que ser declarados através do predicado *Datatype/2*.

Tipo	**Descrição**
Thing (Type, no O3F)	Qualquer termo seja ele uma classe, um datatype, uma associação, etc. É a raiz da hierarquia de tipos de dados
Scalar	Qualquer tipo de dados simples (existente ou que venha a ser criado), por exemplo Number e Word. Opõem-se aos tipos de dados compostos, os quais são as colecções e todos os seus sub-tipos.
Classifier	Qualquer tipo de dados definido por uma classe ou uma associação.
Number	Qualquer número incluindo inteiros e números de virgule flutuante.
Integer	Números inteiros, incluindo os naturais
Natural	Números naturais, os quais não incluem o 0
Float	Números de vírgula flutuante
String	Cadeia de caracteres entre aspas
Word	Cadeia de caracteres com uma única palavra. Pode conter qualquer letra maiúscula ou minúscula e ainda o *underline*.
Char	Um único carácter
Boolean	*True* / *False*
CalendarDate	Representa uma data relativamente a um dado calendário. Formato: *Calendário* : *Sinal YYYY*/*MM*/*DD* *Calendário* : Existem vários calendários, entre os quais o calendário cristão ou gregoriano (gc), o calendário islâmico (ic), e o calendário chinês, o qual não tem início (cc). Os calendários são identificados por palavras. *Sinal* : O sinal positivo indica datas iguais ou mais recentes que a data de início do calendário (e.g., depois de Cristo, no calendário cristão ou gregoriano); o sinal negativo usa-se para datas mais antigas que a data de início (e.g., antes de Cristo, no calendário cristão ou gregoriano)
Date	Representa uma data positiva no calendário gregoriano ou cristão: YYYY/MM/DD. A data simples YYYY/MM/DD é uma abreviatura de gc:+YYYY/MM/DD em que gc é o calendário cristão ou gregoriano.
Time	Representa um instante de tempo: HH:MM:SS:mmm
CalendarDate_and_time	Calendar:SYYYY/MM/DD-HH:MM:SS:mmm
Date_and_time	YYYY/MM/DD-HH:MM:SS:mmm

Tipo	Descrição
Collection	Tipo de dados para representar qualquer colecção de objectos do mesmo ou de tipos diferentes; inclui sacos (*Bag*), conjuntos, e sequências.
Collection_of	O mesmo que *collection* mas para objectos do mesmo tipo, o qual poderá ser qualquer tipo básico (pré-definido ou não) e qualquer classe da ontologia. Exemplos: *collection_of number*, collection_of Person
Bag	Caso particular de colecção em que a ordem dos elementos não interessa mas em que podem existir elementos repetidos. Os elementos de um saco (*Bag*) podem ser do mesmo ou de tipos diferentes.
Bag_of	O mesmo que *Bag*, mas os seus elementos são todos do mesmo tipo. Exemplo: *Bag_of Char*
Set	Conjunto de elementos do mesmo ou de tipos diferentes. Num conjunto, a ordem dos elementos é irrelevante e não há repetições.
Set_of	O mesmo que *Set* mas em que os elementos são todos do mesmo tipo. Exemplo *Set_of Person*
Ordered_set	Conjunto ordenado de elementos, sem repetições
Ordered_set_of	O mesmo que Ordered_set mas os elementos são todos do mesmo tipo
Sequence	Colecção de elementos do mesmo ou de tipos diferentes em que a ordem é importante. Numa sequência pode haver elementos repetidos.
Sequence_of	O mesmo que *Sequence* mas para elementos do mesmo tipo. Exemplo *Sequence_of identity_number*

Tabela 4 – Tipos "built-in" do CO3L

Os tipos de dados CO3L organizam-se de acordo com a hierarquia apresentada na Figura 4.

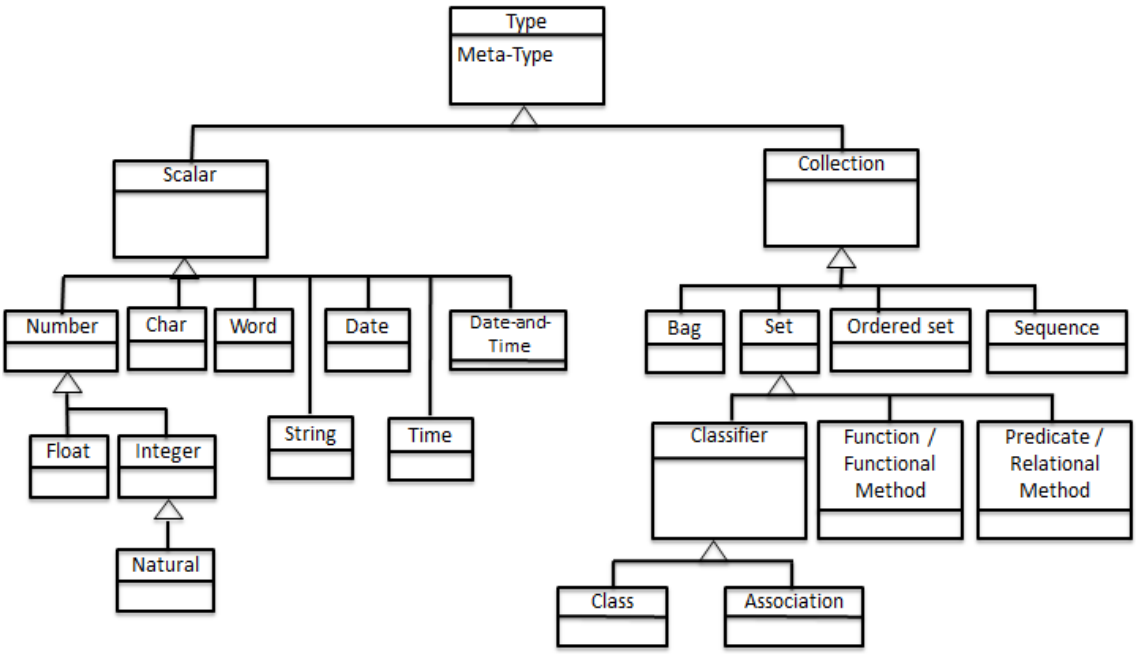

Figura 4 – Hierarquia de Tipos de Dados CO3L

4 Exemplos de Ontologias

As subsecções desta secção apresentam ontologias relativas a cenários de aplicações multiagente descritos mais detalhadamente na documentação sobre a linguagem de conteúdo. Os exemplos descritos têm o propósito de ilustrar vários dos aspectos mais usados numa especificação CO3L. No entanto, há muitas características da linguagem que não são cobertas nos exemplos apresentados.

4.1 Cenário do agente de stocks

Nesta secção descreve-se um cenário em que existe um agente gestor de stocks (B) e um agente da direcção comercial de uma unidade fabril (A) que interage com B para obter informação relevante acerca dos stocks existentes.

A ontologia *ont-materia-prima* descreve o domínio do problema. Existe a classe *Matéria-Prima* com os atributos *nome* que designa a matéria-prima, *existência* que especifica a quantidade existente em stock dessa matéria e *stock-ruptura* que específica o stock de ruptura dessa matéria prima.

Definem-se os tipos de dados (*Datatype*) *TNome* do tipo String que especifica um nome, e *TQuantidade* do tipo float, a qual serve para representar quantidades. Estas definições de novos tipos de dados não são estritamente necessários. Foram feitas para ilustrar a definição de novos tipos simples, mas também para usar tipos com designações menos genéricas do que *String* e *Integer*, e ainda para se poder caracterizar de forma mais económica os tipos que vamos usar.

Os atributos *nome*, *existência* e *stock-ruptura* da classe *Matéria-Prima* são dos tipos *TNome* e *TQuantidade*.

A definição desta ontologia inclui também a indicação do seu proprietário e das datas em que a criação da ontologia foi iniciada e em que a ontologia foi modificada pela última vez.

```
Ontology ont-materia-prima {
  Owner : "luis"
  Initial_date : 2004/05/08
  Last_modification_date : 2009/03/11

  Datatype(TNome, String)
  EntityFacet(TNome, Instance_maximum_size, 60)
  Datatype(TQuantidade, Float)
  EntityFacet(TQuantidade, Smallest_instance, 0)

  Class(Materia-Prima)
  Attribute(Materia-Prima, nome, TNome)
  Attribute(Materia-Prima, existencia, TQuantidade)
  Attribute(Materia-Prima, stock-ruptura, TQuantidade)
}
```

Figura 5 – Ontologia *ont-materia-prima*

Sendo *TNome* um tipo de dados, ele representa um conjunto – um conjunto de nomes. A faceta *Instance_maximum_size* (a qual se aplica a conjuntos) caracteriza os elementos do conjunto a que se aplica. Neste caso concreto diz que os nomes do conjunto *TNome* têm um comprimento máximo de 60 caracteres. A faceta *Smallest_instance* aplicada a *TQuantidade* diz que o elemento mais pequeno deste conjunto é 0.

Na ontologia *ont-materia-prima*, definimos uma classe com três atributos, todos eles com tipos de dados simples. No próximo exemplo, surgirá um atributo cujo tipo de dados é uma classe.

4.2 Agente de uma livraria

No exemplo que se segue, o cenário é o de uma livraria com um sistema de informação sobre os livros existentes e sobre os seus autores. Este sistema de informação tem as classes *Livro* e *Pessoa* com os seus atributos, os quais estão definidos na ontologia *ontologia-livros*. O referido sistema está associado a um agente que dá informação sobre os livros existentes na livraria.

```
Ontology ontologia-livros {

   Datatype(TNome, String)
   EntityFacet(TNome, Instance_maximum_size, 60)
   Datatype(TBi, Natural)
   EntityFacet(TBi, Smallest_instance, 1000000).
   EntityFacet(TBi, Largest_instance, 100000000).

   Class(Livro).
   Class(Pessoa).
   Attribute(Livro, ISBN, Word).
   EntityFacet(Livro.ISBN, Mandatory, True).
   EntityFacet(Livro.ISBN, Distinct, True).
   Attribute(Livro, titulo, TNome).
   EntityFacet(Livro.titulo, Mandatory, True).
   Attribute(Livro, autor, Pessoa).
   EntityFacet(Livro.autor, Mandatory, True).
   Attribute(Pessoa, BI, TBi).
   EntityFacet(Pessoa.BI, Mandatory, True).
   EntityFacet(Pessoa.BI, Distinct, True).
   Attribute(Pessoa, nome, TNome).
   EntityFacet(Pessoa.nome, Mandatory, True).
   Attribute(Pessoa, nacionalidade, TNome).
}
```

Figura 6 – Ontologia *ontologia-livros*

Um livro é caracterizado por um ISBN, um título e um autor. Neste exemplo, considera-se que cada livro tem apenas um autor. Uma pessoa tem um BI, um nome e uma nacionalidade. Ao contrário do outros atributos, cujos tipos estão definidos através de *Datatype/2*, o tipo de dados do atributo *autor* da classe *Livro* é a classe *Pessoa* também definida na mesma ontologia.

Além da utilização de um atributo cujo tipo de dados é uma classe, esta ontologia apresenta uma inovação face à anterior. O atributo *ISBN* da classe *Livro* e o atributo *BI* da classe *Pessoa* são ambos obrigatórios (faceta *Mandatory*) e os seus valores são únicos (faceta *Distinct*), consequentemente, estes atributos podem identificar as instâncias da classes respectivas. Finalmente, alguns atributos (*nome* da classe *Pessoa*, e *título* e *autor* da classe *Livro*) são obrigatórios mas não têm de ter valor único.

As ontologias definidas nos exemplos das secções 4.1 e 4.2 têm classes apenas com atributos. Nos próximos exemplos, surgem classes com métodos, os quais têm argumentos.

4.3 Gestão de informação num servidor

No cenário que serve de base ao exemplo, existe um agente que efectua operações de acesso a informação disponível em ficheiros num dado servidor. Uma das operações do agente é o envio de ficheiros por ftp anónimo para directorias de computadores especificados. Um agente pode usar a ontologia *ont-servidor*, a qual contém a classe *Ficheiro* com o atributo *filename* (entre outros que não são relevantes para o exemplo), e o método de acção *enviar-ftp-anonimo*. O método *enviar-ftp-anonimo* recebe uma sequência com dois argumentos – o *hostname* e a *directoria* de destino do ficheiro a ser enviado.

```
Ontology ont-servidor {

   Datatype(TNome, String).
   EntityFacet(TNome, Instance_maximum_size, 60).

   Class(Ficheiro).
   Attribute(Ficheiro, filename, TNome).
   EntityFacet(Ficheiro.filename, Mandatory, True).
   EntityFacet(Ficheiro.filename, Distinct, True).
   ActionMethod(Ficheiro, enviar-ftp-anonimo).
   Argument(Ficheiro.enviar-ftp-anonimo, hostname, String).
   Argument(Ficheiro.enviar-ftp-anonimo, directoria, String).
}
```

Figura 7 – Ontologia *ont-servidor*

Com uma ontologia que contenha as definições apresentadas em *ont-servidor*, é possível ter várias aplicações de sistemas multi-agente, por exemplo um agente pede ao agente do servidor para este lhe enviar um ficheiro desejado, por ftp anónimo.

4.4 Domínio "video on-demand"

O cenário que se descreve em seguida serve como exemplo de uma aplicação com negociação entre agentes. Num cenário ***"video-on-demand"***, assume-se a existência de diversas empresas de televisão que prestam o serviço *video on-demand*. O assinante tem a possibilidade de seleccionar a exibição de um determinado vídeo. Neste cenário, cada empresa de televisão terá um agente representante capaz de negociar a prestação de serviços. Os assinantes de televisão também dispõem de agentes representantes. Se o assinante pretender receber um determinado vídeo, poderá especificar o vídeo desejado ao agente que o representa. Este negociará com os agentes das empresas de televisão por cabo, a exibição do vídeo nas condições mais favoráveis.

A ontologia *ont-video-on-demand* descreve o domínio da aplicação. Existe a classe *Filme* com atributos *realizador*, *titulo* e *preco-transmissao* e método de acção *transmitir-video*. O método *transmitir-video* recebe três argumentos – o número do assinante para quem o vídeo é transmitido, a data de transmissão do vídeo, e a hora de transmissão.

A ontologia define também os tipos de dados (*Datatype*) *TNome* (para nomes de entidades), e *TPreço* (para preços de filmes) que são designações mais específicas do que *String* e *Float*. Além disso, o valor mínimo do preço é definido.

De acordo com a ontologia, os atributos *titulo*, *realizador* e *preço-de-transmissão* do filme são obrigatórios (faceta *Mandatory*). O *titulo* e o *realizador* formam um mecanismo composto de identificação dos filmes (*Key*).

```
Ontology ont-video-on-demand {
  Datatype(TNome, String)
  EntityFacet(TNome, Instance_maximum_size, 60)
  Datatype(TPreco, Float)
  EntityFacet(TPreco, Smallest_instance, 0)

  Class(Filme)
  Attribute(Filme, titulo, TNome)
  EntityFacte(Filme.titulo, Mandatory, True)
  Attribute(Filme, realizador, TNome)
  EntityFacte(Filme.realizador, Mandatory, True)
  Key(Filme, id1, titulo)
  Key(Filme, id1, realizador)
  Attribute(Filme, preco-de-transmissao, TPreco)
  EntityFacet(Filme.preco-de-transmissao, Mandatory, True)
  ActionMethod(Filme, transmitir-video)
  Argument(Filme.transmitir-video, assinante, Word)
  Argument(Filme.transmitir-video, data, Date)
  Argument(Filme.transmitir-video, hora, Time)
}
```

Figura 8 – Ontologia *ont-video-on-demand*

Na ontologia *ont-video-on-demand* (como em todas as outras, até ao momento), omitiu-se a faceta *Scope* dos atributos e do método pois, em caso de omissão, todos os atributos e métodos são aplicados às instâncias. Se pretendêssemos tornar esse facto mais explícito com o objectivo de melhorar a legibilidade da ontologia, teríamos de acrescentar a faceta *Scope* aos atributos e ao método:

```
  EntityFacet(Filme.titulo, Scope, Instance)
  EntityFacet(Filme.realizador, Scope, Instance)
  EntityFacet(Filme.transmitir-video, Scope, Instance)
```

Também não foi especificado que os argumentos do método *transmitir-video* são todos argumentos de input (i.e., são argumentos usados para passar dados para o método). Essa especificação não é necessária porque, por omissão, todos os argumentos são considerados argumentos de input. No entanto, se pretendêssemos explicitar essa caracaterística para melhorar a clareza da ontologia, poderíamos tê-lo feito, usando a faceta Arg_direction.

```
   EntityFacet(Filme.transmitir-video.assinante, Arg_direction, In)
   EntityFacet(Filme.transmitir-video.data, Arg_direction, In)
   EntityFacet(Filme.transmitir-video.hora, Arg_direction, In)
```

Esta especificação mostra ainda que o operador ponto (.) é também usado para separar os nomes de métodos e operadores dos seus argumentos.

Os tipos de dados das ontologias apresentadas até agora foram tipos de dados básicos (Datatype) ou classes definidas na própria ontologia. No entanto, a linguagem CO3L disponibiliza outros tipos de dados, entre as quais as colecções. No exemplo que se segue, a faceta *Values_set* usa uma colecção para especificar o conjunto de valores possíveis de um dado tipo de dados.

4.5 Dispositivo de segurança numa organização

No cenário que serve de base a este exemplo, o edifício de uma organização tem um dispositivo de segurança com câmaras vídeo e diversos sensores que disparam alarmes em determinadas circunstâncias. Há um sistema multi-agente composto pelos agentes associados às câmaras e aos sensores, e por um sistema de informação que descreve toda a organização. Este sistema de informação detém, entre outras coisas, a relação entre cada câmara ou sensor e a sala onde está instalada. Este sistema também pode saber sempre que um alarme foi activado (através de comunicação com os agentes dos sensores).

A ontologia *ontologia-edificio* descreve o domínio do problema. Existem as classes *Sala* e *Pessoa*. *Sala* tem os atributo *designacao* e *alarme* o qual pode tomar apenas os valores *activado* ou *desactivado*. A classe *Pessoa* tem os atributos *nome* do tipo *TNome*, e *localização* do tipo *Sala*. Para usar tipos de dados com designações (e por vezes, definições) mais específicas do domíno, definem-se os tipos de dados (Datatype) *TNome* (nomes de coisas ou de pessoas) e *TEstadoAlarme* (activado / desactivado).

```
Ontology ontologia-edificio {
  Datatype(TNome, String)
  EntityFacet(TNome, Instance_maximum_size, 60)
  Datatype(TEstadoAlarme, Word)
  EntityFacet(
    TEstadoAlarme,
    Instances_set,
    Set(activado, desactivado))

  Class(Sala)
  Attribute(Sala, designacao, Word)
  EntityFacet(Sala.designacao, Mandatory, True)
  EntityFacet(Sala.designacao, Distinct, True)
  Attribute(Sala, alarme, TEstadoAlarme)
  EntityFacet(Sala.alarme, Mandatory, True)

  Class(Pessoa)
  Attribute(Pessoa, nome, TNome)
  EntityFacet(Pessoa.nome, Mandatory, True)
  EntityFacet(Pessoa.nome, Distinct, True)
  Attribute(Pessoa, localizacao, Sala)
}
```

Figura 9 – Ontologia *ontologia-edificio*

As definições incluídas em *ontologia-edifício* poderiam suportar diversas interacções entre agentes, por exemplo em que um agente (A) envia uma mensagem ao agente associado ao sistema de informação do dispositivo de segurança na organização (B) requerendo que este o informe das identificações (nomes) das pessoas presentes numa sala sempre que for activado um alarme relativo a essa sala.

Nas ontologias exemplificadas até ao momento apenas existem classes com atributos e métodos (os quais têm argumentos). Ainda não surgiu nenhum exemplo com associações entre classes. O próximo exemplo ilustra a utilização de associações.

4.6 Cursos e disciplinas de uma escola universitária

Neste exemplo considera-se uma ontologia que se centra nos vários cursos e disciplinas de uma escola universitária concreta, o ISCTE.

```
ontology ont-cursos-disciplinas {
  Datatype(TNome, String)
  EntityFacet(TNome, Instance_maximum_size, 60)

  Datatype(TBi, Natural)
  EntityFacet(TBi, Smallest_instance, 100000)
  EntityFacet(TBi, Largest_instance, 10000000)

  Class(Curso)
  EntityFacet(Curso, Materialization, Abstract)
  Attribute(Curso, sigla, Word)
  Attribute(Curso, nome, TNome)
  Attribute(Curso, nvagas, Integer)
  EntityFacet(Curso.nvagas, Minimum_value, 0)
  Attribute(Curso, responsavel, TNome)
  ActionMethod(Curso, inscrever)
  EntityFacet(Curso.inscrever, Return_type, Natural)
  EntityFacet(Curso.inscrever, Return_role, numero-inscricao)
  Argument(Curso.inscrever, BI, TBi)

  Class(Licenciatura)

  Class(Mestrado)

  Class(Doutoramento)

  Hierarchy(TiposDeCurso)

  Subtype(TiposDeCurso, Curso, Licenciatura)

  Subtype(TiposDeCurso, Curso, Mestrado)

  Subtype(TiposDeCurso, Curso, Doutoramento)

  Class(Disciplina)
  Attribute(Disciplina, id, Word)
  Attribute(Disciplina, nome, TNome)
  Attribute(Disciplina, responsavel, TNome)

  Association(Curso-Disciplina)
  Argument(Curso-Disciplina, curso, Curso)
  EntityFacet(Curso-Disciplina.curso, Multiplicity, 1..*)
  Argument(Curso-Disciplina, disciplina, Disciplina)
  EntityFacet(Curso-Disciplina.disciplina, Multiplicity, 1..*)
  Attribute(Curso-Disciplina, ano, Natural)
  Attribute(Curso-Disciplina, semestre, Natural)
  EntityFacet(
    Curso-Disciplina.semestre,
    Values_sequence,
    Sequence(1, 2))
  Attribute(Curso-Disciplina, obrigatoriedade, Word)
  EntityFacet(
    Curso-Disciplina.obrigatoriedade,
    Values_set,
    Set(obrigatoria, opcional))

}
```

Figura 10 – Ontologia *ont-cursos-disciplinas*

A ontologia *ont-cursos-disciplinas* ilustra algumas características da linguagem CO3L que ainda não tinham sido apresentadas nos exemplos anteriores. A mais simples destas novidades é a utilização das facetas *Largest_instance* e *Smallest_instance*, as quais especificam o maior e o menor elemento de um conjunto. As facetas *Maximum_value* e *Minimum_value* servem um propósito semelhante, mas são usadas para restringir os valores tomados por atributos e argumentos, e os valores retornados por operadores e métodos. A diferença entre as facetas Instance_maximum_size e Instance_minimum_size, e as facetas Maximum_Size e Minimum_size é que as primeiras duas aplicam-se a conjuntos e caracterizam o tamanho das suas instâncias, enquanto que as segundas duas aplicam-se aos valores de

atributos, métodos e valores retornados. As facetas *Return_type* e *Return_role* também surgiram pela primeira vez. A faceta *Return_type* foi usada para especificar o tipo de dados do valor devolvido pelo método *inscrever* da classe *Curso*. Sempre que uma acção ou um método de acção retorna um valor, além do tipo dessse valor, é necessário especificar o papel desempenhado pelo valor devolvido. A faceta *Return_role* foi usada para dizer que o papel desempenhado pelo método *inscrever* da classe *Curso* é *numero-inscrição*. Isto é, o método de inscrição num curso do ISCTE devolve o número de inscrição. A factea *Materialization* com valor *Abstract* foi aplicada à classe *Curso* para indicar que esta é uma classe abstracta.

Como, por omissão, o valor de *Materialization* é *Concrete*, não foi necessário dizer que as classes *Licenciatura*, *Mestrado* e *Doutoramento* são concretas. As facetas *Values_set* e *Values_sequence* foram aplicadas directamente aos atributos *obrigatoriedade* e *semestre* da associação *Curso-Disciplina*. Nos exemplos anteriores, foram definidos tipos de dados enumerados que foram usados na definição dos atributos. Finalmente, apenas para não sobrecarregar muito a descrição, não foi apresentada na figura a especificação dos atributos obrigatórios e dos atributos com valor único. Mas, numa ontologia completa, essa especificação deve ser feita.

A ontologia *ont-cursos-disciplinas* ilustra outra potencialidade da linguagem CO3L: a definição de hierarquias. A asserção hierarchy(TiposDeCurso) limita-se a dizer que há uma hierarquia com esse nome. As asserções Subtype(TiposDeCurso, Curso, Licenciatura), Subtype(TiposDeCurso, Curso, Mestrado) e Subtype(TiposDeCurso, Curso, Doutoramento) significam que as classes Licenciatura, Mestrado e Doutoramento são subtipos de Curso, na hierarquia TiposDeCurso. Assim sendo, Licenciatura, Mestrado e Doutoramento herdam todos os atributos e métodos de Curso.

Finalmente, *ont-cursos-disciplinas* define a associação chamada Curso-Disciplina, através da asserção Association (Curso-Disciplina). Os dois argumentos da associação Curso-Disciplina são caracterizados por duas asserções *Argument/3*. Um dos argumentos é da classe *Curso* e desempenha o papel de *curso*. O outro argumento é da classe *Disciplina* e desempenha o papel *disciplina*. A multiplicidade dos argumentos da associação é definida pela faceta Multiplicity. Uma disciplina pode estar associada a um ou mais cursos, e um curso pode estar associado a uma ou mais disciplinas.

Até ao momento, todos os exemplos de ontologias apresentadas descrevem os respectivos domínios, em termos de objetos. Todas elas eram constituídas essenciamente por classes, as quais tinham atributos e métodos. Apesar de permitir a descrição de ontologias centradas no conceito de objecto, a linguagem CO3L e o respectivo modelo O3F também possibilitam a definição de ontologias que não recorrem ao conceito de objectos. A ontologia do próximo exemplo ilustra essa possibilidade.

4.7 Índices de valores e cotação de acções na bolsa

A ontologia que se descreve seguidamente diz respeito ao mundo da bolsa de valores. O principal objectivo da sua apresentação é a ilustração da possibilidade de se descrever um dado domínio sem recorrer ao conceito de objecto. Na ontologia que se apresenta, declara-se duas funções e uma acção. As funções retornam o valor de um dado índice de valores e a cotação de uma dada acção, respectivamente. A acção serve para dar uma ordem de venda de um determinado produto.

```
Ontology ontologia-bolsa {
   Datatype(TCotacao, Float).
   EntityFacet(TCotacao, Smallest_instance, 0).

   Datatype(TValor, Float).
   EntityFacet(TValor, Smallest_instance, 0).

   Datatype(TIndice, Word).
   EntityFacet(TIndice, Instances_set, Set(psi-20, bvlp, …)).

   Datatype(TQuantidade, Integer).
   EntityFacet(TQuantidade, Smallest_instance, 0).

   Function(ValorIndice, TValor).
   Argument(ValorIndice, indice, TIndice).

   Function(Cotacao, TCotacao).
   Argument(Cotacao, accao, String).

   Action(Vender).
   EntityFacet(Vender, Return_type, TQuantidade).
   EntityFacet(Vender, Return_role, numero-vendas).
   Argument(Vender, produto, String).
   Argument(Vender, numero-accoes, Natural).
}
```

Figura 11 – Ontologia *ontologia-bolsa*

A primeira parte da ontologia não apresenta qualquer novidade, constando apenas da definição de alguns tipos de dados. As duas asserções *Function/2* declaram duas funções, uma chamada *ValorIndice* que devolve um valor do tipo *TValor*, e outra chamada *Cotacao* que devolve um valor do tipo *TCotacao*. A função *ValorIndice* tem um argumento do tipo *TIndice*, o qual desempenha o papel *indice*. *Cotacao* tem um argumento do tipo *String* que desempenha o papel de *accao*. Finalmente, Action(Vender) significa que existe a acção chamada *Vender*. As duas facetas de *Vender* caracterizam o valor retornado pela acção: trata-se de uma quantidade (*TQuantidade*) que desempenha o papel *numero-vendas*. Finalmente, as duas asserções *Argument/3* significam que *Vender* tem dois argumentos: uma *String* que desempenha o papel *produto*; e um *Natural* que desempenha o papel *numero-accoes*.

Embora as ontologias apresentadas anteriormente sejam exclusivamente ontologias orientadas por objectos, e esta última ontologia não tenha qualquer conceito da orientação por objectos, a linguagem CO3L permite a definição de ontologias mistas, contendo conceitos OO e conceitos não OO.