# Tecnologia para Sistemas Inteligentes
# Apontamentos para as aulas sobre

# Interação de agentes: linguagem de conteúdo SL (Semantic Language)

**Luís Miguel Botelho**

**Departamento de Ciências e Tecnologias da Informação**
**Instituto Superior de Ciências do Trabalho e da Empresa**

**Abril de 2012**

# Tecnologias para Sistemas Inteligentes

## Apontamentos para as aulas

# Índice

# Interação de agentes: linguagem de conteúdo SL (Semantic Language)

Embora o conteúdo das mensagens possa ser escrito em várias linguagens, neste documento adoptaremos uma linguagem de conteúdo que se baseia na linguagem SL (*"Semantic Language"*), a qual tem sido usada em muitos projectos de investigação, em particular em projectos participados pelo grupo de Agentes e Inteligência Artificial do ISCTE - projecto Agentcities [Willmott et al 2001], e projecto CASCOM [Schumacher, Helin and Schuldt 2008].

A FIPA definiu a linguagem SL ("Semantic Language") [FIPA 2002-08] para os conteúdos das mensagens ACL. De acordo com as especificação FIPA, é obrigatório usar um subconjunto de SL nas mensagens trocadas com os agentes da plataforma (e.g., na comunicação com o AMS e com o DF). A esse subconjunto chama-se SL0. As outras mensagens enviadas e recebidas pelos agentes podem usar SL ou outra linguagem de conteúdo qualquer, tal como o Prolog.

A linguagem SL baseia-se na lógica de predicados de predicados de primeira ordem com algumas extensões que permitem a sua utilização para efectuar interrogações e para representar crenças, objectivos, intenções e acção. Daqui até ao fim desta secção, as principais características do SL são apresentadas através de exemplos.

Esta secção descreve os aspectos mais importantes da comunicação em ACL com conteúdos em OO-SL (Object Oriented Semantic Language), uma linguagem quase igual à linguagem SL, mas com algumas extensões que lhe permitem referir classes, objectos, atributos e métodos. Esta extensão é necessária para que as mensagens possam referir as entidades descritas em ontologias orientadas por objectos, o que será o caso tanto nas ontologias OWL (um standard de facto), como nas ontologias O3F (o modelo adoptado neste texto).

# 1 Tipos de Expressões de Conteúdo

Para que uma linguagem de conteúdo possa ser usada com todas os tipos de mensagens da linguagem de comunicação FIPA ACL, é necessário que tenha a capacidade de representar afirmações (usadas em mensagens informativas e em perguntas fechadas), condições (usadas em propostas, pedidos de execução condicionadas de acções, entre outros tipos de mensagens), razões (usadas, por exemplo, nas mensagens de rejeição de pedidos e de propostas, em mensagens de falha), expressões referencias (usadas por exemplo em perguntas abertas), e acções (usadas por exemplo em propostas e em pedidos de execução de acção).

A linguagem OO-SL, bem como a linguagem FIPA-SL em que esta se isnpirou, tem capacidade de representar todos estes tipos de conteúdo. Afirmações, condições e razões são representadas por proposições lógicas, expressões de referência pertencem à linguagem, e acções são representadas por termos de acção. Esta secção descerve sumariamente as expressões de conteúdo, tanto para a linguagem OO-SL como para a linguagem FIPA-SL. Apenas a secção 4 apresenta especificidades da linguagem OO-SL que não se incluem na linguagem FIPA-SL.

Todos os tipos de conteúdo referidos relacionam-se uns com os outros através da gramática informalmente descrita na Tabela 1, Tabela 2, e Tabela 3.

| Expressão de Conteúdo |
|---|
| Proposição |
| Expressão de referência |
| Acção |

**Tabela 1 – Expressões de Conteúdo em SL**

As proposições podem ter diversos tipos, como se vê na Tabela 2.

| Proposição | Proposição Atómica | Proposição Lógica | Proposição Quantificada | Proposição de Acção | Proposição Modal |
|---|---|---|---|---|---|
| Proposição atómica | Símbolo Proposicional | Conjunção | Proposição Universal | Evento | Crenças |
| Proposição lógica | | Disjunção | | Execuibilidade | Incertezas |
| Proposição quantificada | Proposição Predicativa ou Relacional | Implicação | Proposição Existencial | | Objectivo Persistente |
| Proposição de Acção | | Equivalência | | | Intenção |
| Proposição Modal | | Negação | | | |

**Tabela 2 – Tipos de Proposições em OO-SL**

Tanto os termos funcionais como os termos de acção são casos particulares de termos, os quais aparecem como argumentos nas proposições atómicas relacionais, nos termos funcionais, e nos designadores de acção, os quais integram os termos de acção.

| Termo |
|---|
| Variáveis |
| Escalares |
| Termos Funcionais |
| Termos de Acção |
| Expressões de Referência |

**Tabela 3 - Tipos de Termos em OO-SL**

As duas próximas secções descrevem em maior detalhe as proposições e os termos da linguagem OO-SL, apresentando a sua sintaxe e exemplos fora do contexto da comunicação entre agentes, recorrendo a cenários realistas. Tudo o que é dito nas secções 2 e 3 relativamente à linguagem OO-SL é igualmente válido para a linguagem FIPA-SL. As particularidades da linguagem OO-SL que não fazem parte da linguagem FIPA-SL são descritas na secção 4.

# 2 Proposições

Embora as proposições envolvam forçosamente termos, é mais intuitivo começar por elas e passar depois a uma explicação mais detalhada dos termos.

## *2.1 Proposições Atómicas*

As proposições atómicas podem ser símbolos proposicionais e proposições relacionais ou predicativas.

**Exemplo de Símbolo Proposicional**

```
esta_bom_tempo
```

A leitura informal deste símbolo proposicional é "Está bom tempo".

**Exemplo de Proposição Predicativa**

```
(PrimeiroMinistro UK 2003 Blair)
```

A sua leitura informal é "O Tony Blair é o Primeiro Ministro do Reino Unido no ano de 2003"

*PrimeiroMinistro* é o predicado. Os termos *UK*, *2003* e *Blair* são os argumentos do predicado *PrimeiroMinistro* naquela proposição específica.

Em vez de (PrimeiroMinistro UK 2003 Blair), poderia usar-se uma escrita alternativa:

```
(PrimeiroMinistro :país UK :data 2003 :pm Blair)
```

Nesta alternativa, cada argumento do predicado *PrimeiroMinistro* é precedido pela especificação do seu papel, o qual pode ser descrito na ontologia que representa o domínio. Há essencialmente duas vantagens de usar as indicações dos papéis dos argumentos. Por um lado, a proposição torna-se mais clara porque o papel de cada argumento é indicado explicitamente. Por outro lado, a ordem pela qual os argumentos são escritos deixa de ser relevante. Em vez de (PrimeiroMinistro :país UK :data 2003 :pm Blair), poderia escrever-se

```
(PrimeiroMinistro :data 2003 :pm Blair :país UK)
```

exactamente com o mesmo significado.

Como qualquer das escritas das proposições relacionais é válida, com e sem a indicação dos papéis dos argumentos, será usada uma ou a outra escrita conforme for mais claro.

## *2.2 Proposições Lógicas*

As proposições lógicas são formadas a partir das conectivas lógicas de conjunção (*and*), disjunção (*or*), implicação (*implies*), equivalência (*equiv*), e negação (*not*).

Exemplo de uma Proposição Lógica

```
(implies (PrimeiroMinistro UK 2003 Blair)  (Ministro UK 2003 Blair))
```

A sua leitura é "Se o Tony Blair é Primeiro Ministro, então é Ministro"

## *2.3 Proposições Quantificadas*

As proposições quantificdas universalmente expressam relações gerais. As proposições quantificadaas existencialemente expressam relações aplicáveis a pelo menos um objecto.

**Exemplos de Proposições Quantificadas**

```
(forall ?x
  (implies (PrimeiroMinistro UK 2003 ?x)  (Ministro UK 2003 ?x)))
```

Esta proposição tem a seguinte leitura informal: "Se ?x é o primeiro ministro do Reino Unido em 2003, então ?x é Ministro do Reino Unido em 2003"

```
(forall ?x (forall ?p (forall ?a
  (implies (PrimeiroMinistro ?p ?a ?x)  (Ministro ?p ?a ?x)))))
```

"Os Primeiros Ministros de qualquer país num ano qualquer são ministros desse país no mesmo ano"

## *2.4 Proposição de Acção*

Existem dois tipos de proposições de acção: os eventos, isto é, proposições que representam o facto de que uma acção foi executada; e as proposições de execuibilidade, as quais exprimem o facto de que é possível executar uma dada acção. Os eventos são representados recorrendo ao operador *Done*. As proposições de execuibilidade de acção recorrem ao operador *Feasible*.

**Sintaxe e significado informal dos eventos**

```
(done <termo de acção> <proposição>)
```

A acção especificada por <termo de acção> foi executada. Imediatamente antes da sua execução, a condição especificada por <proposição> era verdadeira. Depois da execução, não se sabe nada sobre essa condição. Numa proposição *Done* não é necessário especificar todas as condições verdadeiras. O agente que envia a mensagem com a proposição *Done* é que decide o que pretende dizer ou perguntar.

A abreviatura (Done <termo de acção>) ≡ (Done <termo de acção> True) é usada quando não se pretende especificar a condição que era verdadeira imediatamente antes da execução da acção.

**Sintaxe e significado informal das proposições de execuibilidade de acções**

```
(feasible <termo de acção> <proposição>)
```

A acção especificada por <termo de acção> é exequível. Se for executada, a condição especificada por <proposição> será verdadeira após a execução.

A abreviatura (Feasible <termo de acção>) ≡ (Feasible <termo de acção> True) é usada quando não se pretende especificar as condições que serão verdadeiras imediatamente após a execução da acção.

Serão dados agora exemplos de proposições de acção recorrrendo ao cenário que se descreve.

**Cenário**

Trata-se de um cenário no mundo dos blocos. Existem blocos (a, b, c) e posições (1, 2, 3). Existe um agente robótico que manipula os blocos: agente-blocos. Existe a acção (move Bloco Origem Destino) que representa "Mover Bloco da posição Origem para a posição Destino".

Os predicados Desimpedido e NoTopo descrevem o estado do mundo.

(desimpedido X): O bloco ou a localização X está desimpedida, no sentido em que não tem nada em cima.

(noTopo Bloco X): O bloco especificado está no topo do bloco ou da posição X

**Exemplo**

Neste exemplo, existe a configuração de blocos e posições apresentada na Figura 1.

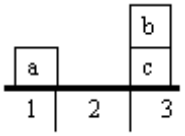


**Figura 1 – Configuração no mundo dos blocos**

O agente Agente_Blocos moveu o bloco A da posição 1 para o bloco B. Antes da execução da acção, o bloco A, a posição 2 e o bloco B estavam desimpedidos, o bloco A estava na posição 1, o bloco C estava na posição 3 e o bloco B estava no topo do bloco C.

```
(done
    (action (agent-identifier :name agente_blocos) (move a 1 b))
    (and (desempedido a) (desempedido 2) (desempedido b) (noTopo a 1) (noTopo c 3) (noTopo b c))
)
```

O agente Agente_Blocos pode mover o bloco A da posição 1 para o bloco B, em resultado do que a posição 1 ficará desimpedida e o bloco A ficará colocado sobre o bloco B.

```
(feasible
    (action (agent-identifier :name agente_blocos) (move a 1 b))
    (and (desimpedido 1) (noTopo a b))
)
```

## *2.5 Proposições Modais*

As proposições modais da linguagem OO-SL permitem expressar os estados mentais de crença (através do operador B), incerteza (através do operador U), desejo (através do operador C), objectivo persistente (através do operador PG), e intenção (através do operador I). Em qualquer das proposições modais, o primeiro argumento do operador modal é um agente, e o segundo argumento é uma proposição.

As crenças representam aquilo em que um agente acredita. As incertezas representam aquilo em que um agente está mais inclinado a acreditar mas admite a possibilidade de estar enganado. Os desejos são estados do mundo que o agente deseja que sejam verdadeiros. Os objectivos persitentes são estados do mundo que o agente pretende que sejam sempre satisfeitos. Um objectivo persistente é aquilo que o agente que envia uma subscrição de informação pretende que o receptor crie: o objectivo persistente de enviar a informação subscrita. Intenções são os estados mentais que antecedem a acção do agente na tentativa de atingir um estado do mundo em que a intenção é satisfeita. O agente forma a intenção de que uma dada condição seja verdadeira apenas se decidiu agir em prol de atingir a sua intenção.

**Cenário**

Existe uma empresa chamada COPAM com um agente gestor de stocks chamado Agente-Stocks. No domínio da aplicação, existem os predicados materia-prima/1 que serve para enumerar as matérias primas existentes, stock-ruptura/2 que relaciona uma matéria prima com o seu stock de ruptura e existencia/2 que relaciona uma matéria prima com a quantidade existente.

Vamos supor que neste cenário, o agente gestor de stocks acredita que todas as matérias primas estão acima do stock de ruptura:

$\forall x \; \forall q \; \forall r \; [\text{Materia-Prima}(x) \wedge \text{Existencia}(x, q) \wedge \text{Stock-Ruptura}(x, r) \Rightarrow q > r]$

```
(B
    (agent-identifier :name Agente-Stocks@COPAM.PT)
    (forall ?x (forall ?q (forall ?r
      (implies (and (materia-prima ?x) (existencia ?x ?q) (stock-ruptura ?x ?q)) (> ?q ?x)))))
)
```

# 3 Termos

Parte dos tipos de termos da linguagem OO-SL existe igualmente na lógica de predicados de primeira ordem e são muito simples

**Variáveis** (e.g., ?sala)

**Escalares**: **Números** (e.g., 2004, 21.5); **Palavras** (e.g., Portugal); **Strings** (e.g., "Campeonato do Mundo de Futebol" o \ funciona como caracter de escape, por exemplo para escapar ao significaado das aspas: "Isto \" é o caracter aspas").

## *3.1 Expressões Funcionais*

**Syntaxe**

Expressão Funcional =

"(" <functor> <function arguments> ")" |

"(" <functor> <function parameters> ")"

Function Arguments = term+

Function Parameters = parameter+

Parameter = ":" <parameterName> <Term>

O functor é uma palavra dependente do domínio da aplicação ou os functores da linguagem *"set"* e *"sequence"* os quais têm o sentido de construtores que recebem uma colecção de elementos e criam um conjunto ou uma sequência.

**Exemplos**

(Capital Portugal): Capital de Portugal. Descreve a constante Lisboa

(set a e i o u): conjunto com as vogais do alfabeto

(sequence a b a c): sequência com os elementos a, b, e a por esta ordem

(set): Conjunto vazio

(sequence): Sequência vazia

(Division :divisor 10 :dividendo 2)

Como qualquer das escritas de termos funcionais é válida, com e sem a indicação dos papéis dos argumentos, será usada uma ou a outra escrita conforme for mais claro.

## *3.2 Termos de Acção*

Existem dois tipos de termos de acção: os termos de acção simples (que representam uma única acção) e os termos de acção compostos (que representam sequências e alternativas de acção).

**Sintaxe**

Termos de Acção Simples: "(" "action" <identificador de agente> <designador de acção> ")"

Sequência de Ações: "(" ";" <Termo de Acção> <Termo de Acção>+ ")"

Alternativa de acções: "(" "|" <Termo de Acção> <Termo de Acção>+ ")"

**Exemplos de Termo Simples**

O Agente_Blocos move o bloco A da localização 1 para o topo do bloco B

(action (agent-identifier :name agente_blocos) (move a 1 b))

(action (agent-identifier :name agente_blocos) (move :bloco a :origem 1 :destino b))

Como qualquer das escritas de termos de acção é válida, com e sem a indicação dos papéis dos argumentos, será usada uma ou a outra escrita conforme for mais claro.

**Exemplos de Sequência de Acções**

O agente gestor de livros da editora Bertrand imprime o livro identificado pelo identificador l001 e depois distribui-o

```
(;
   (action (agent-identifier :name agent-gestor-de-livros@bretrand.pt) (imprimir-livro l001))
   (action (agent-identifier :name agent-gestor-de-livros@bretrand.pt) (distribuir-livro l001))
)
```

**Exemplos de Alternativa de Acções**

O agente gestor do processamento de livros da editora Bertrand aceita ou rejeita um livro submetido para publicação

```
(|
   (action (agent-identifier :name agent-gestor-de-livros@bretrand.pt) (aceitar-livro l001))
   (action (agent-identifier :name agent-gestor-de-livros@bretrand.pt) (rejeitar-livro l001))
)
```

## *3.3 Expressões Referenciais*

Um pouco à semelhança com o que acontece com as expressões funcionais, as expressões de referência descrevem objectos (entidades) sem especificarem o seu nome. A diferença entre uma expressão funcional e uma expressão de referência é que esta última permite especificar o objecto referido através de um conjunto de condições representadas por uma proposição.

Na linguagem OO-SL existem três tipos de expressões de referência: as que referem o único objecto que satisfaz uma dada condição; as que referem um dos objectos que satisfazem uma dada condição; e as que referem o conjunto de todos os objectos que satisfazem uma dada condição.

**Sintaxe**

Expressão Referencial = "(" <operador referencial> <termo> <proposição> ")"

Operador Referencial = iota | all | any

Os operadores de referência *iota*, *all*, e *any* têm a seguinte leitura informal:

Iota – "O único que"

All – "(O conjunto de) todos os que"

Any – "Um dos que"

**Exemplos**

O único (clube) que satisfaz a propriedade de ser campeão de futebol de Portugal de 2003

(iota ?x  (campeao-de-futebol Portugal 2003 ?x))

Este termo refere o F.C. Porto.

O conjunto de todos os objectos que satizfazem a propriedade de ser vogal

(all ?x (vogal ?x))

Este termo refere o conjunto das vogais (set a e i o u)

Um dos objectos que satisfaz a propriedade de ser uma consoante

(any ?x (consoante ?x))

Este termo refere uma das consoantes, por exemplo a letra C.

# 4 Objectos, o predicado instance/2 e o operador ponto (.)

Até aqui, todas as construções sintácticas descritas para a linguagem OO-SL são igualmente construções sintácticas da linguagem FIPA-SL e têm exactamente o mesmo significado e a mesma utilização em ambas as linguagens. Nesta secção, apresenta-se a única diferença entre as duas linguagens. Apesar de ser uma diferença sintacticamente muito simples, ela possibilita à linguagem OO-SL expressar conceitos e relações envolvendo entidades modeladas por uma abordagem orientada por objectos.

Um dos conceitos fundamentais dos modelos orientados por objectos é o conceito de classe. Uma classe é um conjunto de objectos com características semelhantes. Neste tipo de modelo, uma classe é caracterizada por um conjunto de atributos e um conjunto de métodos. Os atributos permitem representar a estrutura estática dos objectos da classe, e os métodos representam as operações que podem ser efectuadas pelos objectos da classe.

O predicado *instance/2* da linguagem OO-SL relaciona um objecto com o nome da sua classe. *instance(<objecto>, <classname>)* significa que <objecto> é um membro, isto é, uma instância da classe designada pela stringh <classname>. Por exemplo, admitindo que existe a classe CidadePortuguesa, a expressão (all ?x (instance ?x CidadePortuguesa)) representa exactamente o conjunto de todas as cidades portuguesas.

Tal como nas linguagens de programação e de especificação orientadas por objectos, o operador ponto (.) aplica-se quer a classes quer a objectos (instâncias) para aceder às suas partes. O operador ponto (.) permite especificar quer atributos (de classe ou de instâncias) quer os seus métodos. Recorrendo ainda ao exemplo baseado na classe CidadePortuguesa, podemos supor que esta classe tem alguns atributos, entre os quais a designação da cidade e o distrito a que pertence. Usando estes dois atributos, podemos especificar o conjunto dos nomes das cidades portuguesas que ficam no distrito de Lisboa:

(all ?x.designacao (and (instance ?x CidadePortuguesa) (= ?x.distrito Lisboa)))

Imaginemos agora que, no domínio das cidades portuguesas em que nos temos vindo a centrar, cada cidade tem uma operação que lhe permite desenhar o seu mapa no ecrã do computador. Essa operação é representada por um método de acção, por exemplo, o método *desenharMapa*. Se a variável ?x tiver sido previamente instanciada com o objecto que representa a Amadora, então a expressão (?x.desenharMapa) é a invocação do método *desenharMapa* na cidade Amadora. O resultado desta invocação seria o surgimento, no ecrã do computador, do mapa da Amadora.

Os objectos e as póprias classes podem ter métodos de três tipos: funcionais, relacionais e de acção. Os métodos funcionais servem apenas para devolver um valor quando aplicados ao objecto ou à classe a que pertencem. Os métodos relacionais estabelecem uma relação entre o objecto ou a classe a que pertencem e outras entidades as quais são recebidas como parametros do método. Os métodos de acção, quando são executados, efectuam alterações no mundo. Eventualmente, um método de acção pode devolver um valor. Qualquer tipo de método pode ter argumentos. Por exemplo, o método relacional *distancia* dos objectos da classe CidadePortuguesa tem dois argumentos - uma distância e uma cidade – e significa que a cidade a que é aplicado fica a uma dada distância de uma dada cidade. Usando este método relacional, a expressão de referência

```
(all
    (sequence ?x.nome ?d)
    (exists ?y (and
        (instance ?y CidadePortuguesa)
        (= ?y.designacao Porto)
        (instance ?x CidadePortuguesa)
        (?y.distancia ?x ?d))))
```

representa o conjunto de todos os pares ordenados formados por nomes de cidades portuguesas e respectivas distâncias ao Porto.

Finalmente, e para terminar o assunto da representação e utilização de classes, objectos, atributos e métodos, a linguagem OO-SL permite a representação explícita de instâncias (i.e., de objectos) de uma dada classe. Uma instância é uma expressão funcional em que o functor é o nome da classe a que a

instância pertence e pela especificação dos valores dos seus atributos. Nem todos os atributos têm que ser especificados; basta especificar os atributos obrigatórios. Recorrendo ao exemplo das cidades portuguesas, admitindo que o nome da cidade é obrigatório, a seguinte expressão funcional representa a instância correspondente a Sines:

(CidadePortuguesa :designacao Sines :distrito Setúbal)

Na instância representada, foi especificada a designação da cidade, o que corresponde a um atributo obrigatório, e foi também especificado que Setúbal é o distrito a que Sines pertence.

# 5 Proposição OO-SL como Conteúdo de Mensagens de Informação

Mesmo usando a mesma linguagem de comunicação e a mesma linguagem de conteúdo, a compreensão das mensagens trocadas entre agentes durante a comunicação depende muito da ontologia definida.

Os exemplos que se seguem dizem respeito a uma aplicação em que existe um agente representante do serviço de stockagem e aprovisionamentos de uma empresa. Nesse cenário, ambos os exemplos apresentados são mensagens enviadas pelo agente de stocks dizendo que as existências de todas as matérias primas são superiores ao stock de ruptura correspondente. Os conteúdos das mensagens são bastante diferentes uns dos outros porque a ontologia usada em cada caso é também ela diferente.

No primeiro exemplo, usa-se uma ontologia orientada por objectos, tal como definido a propósito da descrição da linguagem CO3L mas apresentada aqui novamente para facilitar.

```
Ontology ont-materia-prima-oo {
  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 2 – Ontologia orientada por objectos *ont-materia-prima-oo***

A ontologia *ont-materia-prima-oo* modela a matéria prima como uma classe com os atributos nome, existência e stock de ruptura. Usando esta ontologia, na mensagem da Figura 3, o agente A informa o agente B que todas as matérias primas têm um stock actual superior ao seu stock de ruptura.

```
(inform
  :sender A
  :receiver (set B)
  :content "(
    (forall ?x
      (implies
        (instance ?x Materia-Prima)
        (>?x.existencia ?x.stock-ruptura)))
  )"
  :language oo-sl
  :Ontology (set ont-materia-prima-oo)
)
```

**Figura 3 - Mensagem *inform*, usando uma ontologia OO**

O conteúdo da mensagem da Figura 3 corresponde à seguinte proposição do cálculo de predicados de primeira ordem, aumentada com o operador ponto (.) típico das linguagens orientadas por objectos.

$$\forall_x \text{ instance}(x, \text{Materia-Prima}) \Rightarrow x.\text{existencia} > x.\text{stock-ruptura}$$

Para todos os x, se x for uma matéria prima, então a quantidade existente de x é maior do que o stock de ruptura de x.

No exemplo seguinte, usa-se uma ontologia alternativa que, em vez de classes e atributos, usa relações e funções.

```
Ontology ont-materia-prima {

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

   Predicate(materia-prima).
   Argument(materia-prima, materiaPrima, TNome).

   Function(existencia, TQuantidade).
   Argument(existencia, materiaPrima, TNome).

   Function(stock-ruptura, TQuantidade).
   Argument(stock-ruptura, materiaPrima, TNome).
}
```

**Figura 4 – Ontologia relacional e funcional *ont-materia-prima***

Na ontologia *ont-materia-prima*, as matérias primas são mantidas na relação *matéria-prima*. Nesta ontologia, a função *existencia/1* devolve a quantidade existente de uma matéria prima; e a função *stock-ruptura* devolve o stock de ruptura de uma matéria prima.

```
(inform
   :sender A
   :receiver (set B)
   :content "(
      (forall ?x
         (implies
            (materia-prima ?x)
            (>(existencia ?x)(stock-rutura ?x))))
   )"
   :language oo-sl
   :Ontology (set ont-materia-prima)
)
```

**Figura 5 – Mensagem *inform* com uma ontologia relacional e funcional**

As mensagens *inform* da Figura 7 e da Figura 3, embora diferentes, são bastante semelhantes. A razão para essa semelhança é que o modelo de objectos é essencialmente um modelo funcional com organização e sintaxe diferentes.

No próximo exemplo, usa-se uma ontologia puramente relacional.

```
Ontology ont-materia-prima-relacional {

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

   Predicate(materia-prima).
   Argument(materia-prima, materiaPrima, TNome).
   Argument(materia-prima, existencia, TQuantidade).
   Argument(materia-prima, stock-ruptura, TQuantidade).
}
```

**Figura 6 – Ontologia relacional *ont-materia-prima-relacional***

A ontologia representa as matérias primas, as suas existências e os seus stocks de ruptura através de uma relação com três argumentos. Com esta representação, a mensagem que informa que as existências das matérias primas são superiores aos seus stocks de ruptura é a que se representa na Figura 7.

```
(inform
   :sender A
   :receiver (set B)
   :content ”(
      (forall ?x
        (forall ?r
          (forall ?q
            (implies
              (materia-prima ?x ?q ?r)
              (> ?q ?r)))))
   )”
   :language oo-sl
   :Ontology (set ont-materia-prima-relacional)
)
```

**Figura 7 – Mensagem *inform* com uma ontologia relacional**

Desta vez, a mensagem inform é bem diferente das duas anteriores pois o modelo relacional é muito diferente quer do modelo funcional quer do modelo orientado por objectos.

# 6 Expressão de Referência em Perguntas Abertas

Há três mensagens para efectuar interrogações: *query-if*, *query-ref* e *subscribe*. A mensagem *query-if* é usada para fazer perguntas fechadas, isto é, perguntas cuja resposta sópode ser sim ou não. A mensagem *query-ref* é usada para fazer perguntas abertas, isto é, perguntas cuja resposta não está limitada, por exemplo, "qual é o stock de ruptura do milho?". Finalmente, *subscribe* é semelhante a *query-ref* mas, em vez de se pretender apenas uma resposta, pretende-se que o agente receptor da mensagem lhe responda sempre que acreditar que têm resposta enviar. Isto é, pretende-se que o agente que recebe a mensagem, crie o objectivo persistente de enviar respostas ao agente que faz a subscrição.

A mensagem *query-if* tem uma proposição como argumento. Aquilo que se pretende, é saber se a proposição é verdadeira. Com as mensagens *query-ref* e *subscribe*, pretende-se receber os objectos que satisfazem uma dada condição. O seu conteúdo é uma expressão de identificação, também chamada expressão de referência, isto é, uma expressão que identifica um dado objecto.

Existem três operadores que se podem usar em expressões de identificação: *iota*, *all*, e *any*. *Iota* (nome de uma letra grega) serve para indicar o único objecto que satisfaz uma dada condição. *All* serve para indicar o conjunto de todos os objectos que satisfazem a condição especificada. Finalmente, *any* serve para indicar um dos objectos que satisfazem a condição especificada.

Se um agente recebe uma pergunta com o operador *iota* e descobre que não existe nenhuma resposta ou que existe mais do que uma resposta, deve enviar uma mensagem *failure*.

Se um agente recebe uma pergunta com o operador *any*, deve responder com uma das respostas possíveis (se existirem mais do que uma). A escolha da resposta é da responsabilidade do agente que responde, ou tem que ser previamente combinada. Se não existir nenhuma resposta, então há uma falha.

Finalmente, se um agente recebe uma pergunta com o operador *all* e não existir nenhum objecto que satisfaz a condição especificada, a resposta é uma mensagem *inform* com um conjunto vazio.

Uma expressão de identificação é formada por um operador de identificação, por um termo que especifica aquilo que pretendemos obter na resposta, e por uma proposição que especifica a condição que tem que ser satisfeita pelos objectos a incluir na resposta:

(<operador de referência> <termo> <proposição>)

Neste exemplo, vamos efectuar a pergunta "Qual é o stock de ruptura de milho?". Como se trata de uma pergunta aberta, usa-se a mensagem *query-ref*. Como só pode existir um único stock de ruptura para o milho, deve usar-se o operador *iota*. Se o agente que recebe a pergunta falhar ao tentar responder, fica a saber-se que há uma deficiência no conhecimento do agente que responde.

Usando a ontologia *ont-materia-prima-oo*, apresentada na Figura 2, a pergunta usada para saber o valor do stock de ruptura do milho é a que se representa na Figura 8.

```
(query-ref
   :sender B
   :receiver (set A)
   :content "(
      (iota ?x.stock-ruptura
         (and (instance ?x Materia-Prima) (= ?x.nome \"milho\")))
   )"
   :language oo-sl
   :Ontology (set ont-materia-prima-oo)
   :protocol fipa-query
   :conversation-id c0072
   :reply-with query1
)
```

**Figura 8 – Pergunta aberta: qual é o stock de ruptura do milho**

O conteúdo da mensagem da Figura 8 é uma expressão de identificação (ou expressão de referência) com a seguinte leitura informal *"o stock de ruptura do único objecto ?x que é uma instância da classe*

*Matéria-Prima e que se chama milho"*. No conteúdo da menssagem surge o caracter especial *backslash* (\) antes das aspas da designação "milho". O *backslash* (\) tem de se usar para dizer que as aspas da String "milho" fazem parte do conteúdo da mensagem.

O parâmetros *:language OO-SL e :Ontology (set ont-materia-prima-oo)* especificam a linguagem de conteúdo e a ontologia usada nesse conteúdo.

O parâmetro *:protocol fipa-query* indica que a conversação iniciada pela pergunta deve seguir o protocolo de interacção *fipa-query*. Todas as mensagens de uma conversação têm o mesmo identificador de conversação, o qual deve ser um identificador único. Cabe ao agente que inicia a conversação criar o seu identificador único. O parâmetro conversation-id c0072 foi usado para estabelecer o identificador desta conversação.

O parâmetro *:reply-with query1* serve para dizer ao receptor da mensagem que a resposta deve referir a mensagem *query1*.

De acordo com a convenção usual, a reposta a uma interrogação deve ser uma mensagem *inform* com o formato "*o objecto que satisfaz a propriedade… é…"* (Figura 9).

```
(inform
   :sender A
   :receiver (set B)
   :content "(
      (= (iota ?x.stock-ruptura
           (and (instance ?x Materia-Prima) (= ?x.nome \"milho\")))
         250)
   )"
   :language oo-sl
   :Ontology (set ont-materia-prima-oo)
   :protocol fipa-query
   :conversation-id c0072
   :in-reply-to query1
)
```

**Figura 9 – Resposta a uma interrogação**

Em resposta à mensagem *query1*, o agente A informa o agente B que o stock de ruptura do único objecto que satisfaz a propriedade de ser uma matéria prima cujo nome é milho é 250. Dito de outra forma, o stock de rutura do milho é 250.

Em vez de pretender saber o stock de ruptura do milho, o agente B poderia pretender saber os valores do stock de ruptura de todas as matérias primas. Neste caso não se pode usar o operador *iota* dado que se esperam várias respostas, uma para cada matéria prima. Neste caso usa-se o operador *all* porque se pretende obter o conjunto de todas as respostas.

Note-se que não pretendemos obter apenas um conjunto de valores de stocks de rutura. Pretendemos obter um conjunto de pares ordenados nome da matéria prima – stock de rutura. Qualquer conjunto ordenado pode formar-se recorrendo ao operador *sequence*.

```
(query-ref
   :sender B
   :receiver (set A)
   :content "(
      (all (sequence ?x.nome ?x.stock-ruptura)
         (instance ?x Materia-Prima)
   )"
   :language oo-sl
   :Ontology (set ont-materia-prima-oo)
   :protocol fipa-query
   :conversation-id c0073
   :reply-whith query2
)
```

**Figura 10 – Interrogação com respostas múltiplas**

A mensagem da Figura 10 tem a seguinte leitura informal "*Qual é o conjunto de todos os pares ordenados de nome e stock de ruptura dos objectos da classe materia-prima?*". A resposta é um *inform* com um conjunto de respostas (Figura 11).

```
(inform
   :sender A
   :receiver (set B)
   :content "(
      (=
         (all (sequence ?x.nome ?x.stock-ruptura)
            (instance ?x Materia-Prima)
         (set (sequence \"milho\" 250) (sequence \"batata\" 100)...)
      )
   )"
   :language oo-sl
   :Ontology (set ont-materia-prima-oo)
   :protocol fipa-query
   :conversation-id c0073
   :in-reply-to query2
)
```

**Figura 11 – Resposta a uma interrogação com várias respostas**

A leitura informal desta mensagem é a seguinte "Em resposta à interrogação query2, o conjunto dos pares ordenados nome stock de ruptura de todos os objectos da classe Materia-Prima é {<milho, 250>, <batata, 100>}".

Nas interrogações apresentadas até aqui, o emissor pretende que o receptor lhe envie uma mensagem com a resposta a uma dada pergunta. Para isso, usa a mensagem *query-ref*. No entanto, em domínios dinâmicos, faz sentido pedir ao receptor para responder de tempos a tempos a uma dada pergunta. Para isso, usa-se a mensagem *subscribe*, a qual tem uma sintaxe idêntica à da mensagem *query-ref*.

A utilização de subscribe recorrerá ao exemplo do agente de uma livraria, já descrito na apresentação da linguagem CO3L. Trata-se de uma plicação em que um agente representante de uma livraria tem acesso a um sistema de informação com os livros existentes e seus autores.

```
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).
}
```

Neste exemplo, o agente A pretende que o agente B lhe diga os títulos de livros de autores portugueses existentes na livraria representada por B.

```
(subscribe
   :sender A
   :receiver (set B)
   :content "(
      (all ?x.titulo
         (and
            (instance ?x Livro)
            (= ?x.autor.nacionalidade \"portuguesa\")
   )"
   :language oo-sl
   :Ontology (set ontologia-livros)
   :protocol fipa-subscribe
   :conversation-id c0203
   :reply-with query3
)
```

**Figura 12 – Subscrição de informação**

Para melhor compreender a expressão ?x.autor.nacionalidade, basta que nos recordemos que ?x.autor é uma pessoa, a qual é caracterizada por vários atributos, entre os quais o atributo nacionalidade. ?x.autor.nacionalidade significa a nacionalidade doautor de ?x.

Ao receber esta mensagem, o agente B, informará o agente A do conjunto de livros de autores portugueses já existentes e daí em diante, voltará a enviar os títulos de autores portugueses existentes sempre que achar conveniente, com uma certa frequência.

A frequência com que o agente B tem que voltar a responder à pergunta não é especificada na mensagem. Se o agente A pretender especificar uma dada frequência deverá fazê-lo através de outra mensagem. Neste caso específico, o agente B sabe que basta responder quando recebe ou quando vende livros.

# 7 Termos de Acção em Mensagens com Pedidos

O exemplo de interacção apresentado baseia-se num cenário em que se supõe a existência de um agente de gestão de um servidor, o qual é capaz de efectuar diversas acções no servidor e de prestar informação sobre os recursos existentes. O exemplo que se descreve incide apenas sobre uma pequena parte deste cenário, a qual é captada pela ontologia ont-servidor.

```
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)
}
```

Neste exemplo, o agente A pede ao agente do servidor (agente B) para este lhe enviar o ficheiro com o mapa da sua instituição por ftp anónimo. O envio de um ficheiro por ftp anónimo faz-se recorrendo à acção *enviar-ftp-anonimo* da classe *Ficheiro* descrita na ontologia *ont-servidor*.

```
(request
   :sender A
   :receiver (set B)
   :content "( (action B ((iota ?fich (and
        (instance ?fich Ficheiro)
        (= ?fich.filename \"/organizacao/mapa.jpg\"))
     ).enviar-ftp-anonimo
        :hostname \"ftp.iscte.pt\" :directoria \"/publico\"))
   )"
   :language oo-sl
   :Ontology (set ont-servidor)
   :protocol fipa-request
   :conversation-id c1203
)
```

**Figura 13 – Mensagem com um pedido para execução de uma acção**

As mensagens da família *request* são usadas por um agente para pedir a outro que execute uma dada acção. O conteúdo da mensagem *request* é a acção que deve ser executada. Para especificar uma acção, usa-se o operador *action*, o qual tem dois argumentos – o agente que efectua a acção e um designador de acção, o qual especifica a acção que será executada. No exemplo que se descreve, a acção a ser executada é um método da classe ficheiro. A primeirra coisa a fazer é localizar o ficheiro a que o método *enviar-ftp-anonimo* se aplica. A expressão de referência (iota ?fich (and (instance ?fich ficheiro) (= ?fich.filename \"/organizacao/mapa.jpg\") identifica o ficheiro pretendido: "O único objecto que satisfaz a condição de ser uma instância da classe ficheiro e de ter um *filename* igual a "/organizacao/mapa.jpg". Como a expressão de referência é o objecto desejado, pode aplicar-se o método desejado. É esse o significado do operador ponto (.) a seguir à expressão de referência seguido do nome do método. :hostname "ftp.iscte.pt" e :directoria "/publico" são os dois parametros do método *enviar-ftp-anonimo*. De acordo com a ontologia, um dos parâmetros desempenha o papel "hostname" e o outro desempenha o papel "directoria". Em OO-SL, os papéis desempenhados pelos parâmetros dos operadores ou dos métodos são indicados através do operador dois pontos (:) seguido pelo papel.

O protocolo *fipa-request* especifica que, após um agente ter recebido uma mensagem *request*, deve enviar uma mensagem *agree* ao remetente dizendo que se compromete a executar a acção, ou enviar uma mensagem *refuse* dizendo que recusa o pedido. Se o agente concorda com o pedido e, ao tentar executar a acção especificada, ocorre uma falha, o agente deve enviar uma mensagem *failure* ao

remetente, informando-o de que houve uma falha e da razão para essa falha. Finalmente, após a acção ter sido executada com sucesso, o agente que recebe o pedido (*request*) envia uma mensagem *inform* dizendo que a acção foi executada ou informando o agente que envia o *request* do resultado da acção. De acordo com o mesmo protocolo, se o destinatário não perceber a mensagem recebida, deve enviar uma mensagem *not-understood* dizendo a razão pela qual não percebeu.

Até ao fim desta secção, apresemtam-se as possíveis mensagens que seriam enviadas pelo agente do servidor, depois de ter recebido a mensagem *request*, nas várias fases do protocolo *fipa-request*.

```
(agree
   :sender B
   :receiver (set A)
   :content "(
     (action B ((iota ?fich (and
        (instance ?fich Ficheiro)
        (= ?fich.filename \"/organizacao/mapa.jpg\"))
     ).enviar-ftp-anonimo
        :hostname \"ftp.iscte.pt\" :directoria \"/publico\"))
      True
   )"
   :language oo-sl
   :Ontology (set ont-servidor)
   :protocol fipa-request
   :conversation-id c1203
)
```

**Figura 14 – O agente B aceita executar a acção**

Através da mensagem agree da Figura 14, o agente A aceita efectuar a acção pedida no *request* recebido na mesma conversação. A mensagem *agree* tem um conteúdo formado por duas partes: a acção que se aceita efectuar e as condições em que a acção será efectuada. Por vezes, essa segunda parte do conteúdo – a condição em que a acção será efectuada – é usada para especificar, por exemplo, o instante da execução. No exemplo da Figura 14, a proposição True significa que não se especifica qualquer condição para a execução da acção. Isto significa em rigor que a acção será executada quando *True* for verdade. Como *True* é sempre verdade, isto é o mesmo que não especificar condições.

Em vez de aceitar executar a acção pedida, o agente do servidor poderia recusar-se a executá-la. Nesse caso, enviaria a mensagem *refuse* com uma razão para rejeitar o pedido recebido. A Figura 15 mostra um exemplo do que seria a mesnagem *refuse*.

```
(refuse
   :sender B
   :receiver (set A)
   :content "(
     (action B ((iota ?fich (and
        (instance ?fich Ficheiro)
        (= ?fich.filename \"/organizacao/mapa.jpg\"))
     ).enviar-ftp-anonimo
        :hostname \"ftp.iscte.pt\" :directoria \"/publico\"))
     (acesso-negado :filename \"/organizacao/mapa.jpg\" :requester A)
   )"
   :language oo-sl
   :ontology (set ont-servidor)
   :protocol fipa-request
   :conversation-id c1203
)
```

**Figura 15 – O agente B rejeita executar a acção pedida**

Nesta mensagem, a expressão (acesso-negado :filename \"/organizacao/mapa.jpg\" :requester A) é a razão da rejeição. O agente de gestão do servidor recusa-se a enviar um ficheiro por ftp porque o agente A, que o pediu, não tem permissão para aceder ao ficheiro requerido. Uma razão para uma recusa, é

uma proposição que, por ser verdadeira constitui a razão para o agente se recusar a realizar a acção requerida. Sendo uma proposição, isso significa que *acesso-negado* é um predicado. Para que esta mensagem possa ser interpretada pelos agentes envolvidos na conversação, é necessário que o predicado *acesso-negado* e os seus argumentos sejam descritos na ontologia:

```
Predicate(acesso-negado)
Argument(acesso-negado, filename, TNome)
Argument(acesso-negado, requestor, agent-identifier)
```

*agent-identifier* é a classe que representa a identificação dos agentes. Quando estas declarações forem incluídas na ontologia *ont-servidor*, ou se importa a ontologia onde se define a classe *agent-identifier*, ou se define essa classe na própria ontologia.

Vamos supor que o agente do servidor (B) aceitou executar a acção de transferência do ficheiro. Nesse caso, quando a for executar, ela poderá ter sucesso ou insucesso. Se tiver insucesso, de acordo com o protocolo *fipa-request*, o agente B tem de enviar uma mensagem *failure* ao agente A dizendo que houve uma falha na execução da acção e informando-o da razão dessa falha. A mensagem apresentada na Figura 16 exemplifica esta situação.

```
(failure
   :sender B
   :receiver (set A)
   :content "(
     (action B ((iota ?fich (and
        (instance ?fich Ficheiro)
        (= ?fich.filename \"/organizacao/mapa.jpg\"))
     ).enviar-ftp-anonimo
        :hostname \"ftp.iscte.pt\" :directoria \"/publico\"))
     (ficheiro-desconhecido :filename \"/organizacao/mapa.jpg\")
   )"
   :language oo-sl
   :ontology (set ont-servidor)
   :protocol fipa-request
   :conversation-id c1203
)
```

**Figura 16 – O agente B informa A que houve uma falha**

A proposição (ficheiro-desconhecido :filename \"/organizacao/mapa.jpg\") significa que o ficheiro cujo filename é */organizacao/mapa.jpg* não é conhecido do agente do servidor. Mais uma vez, para que esta proposição possa ser compreendida, o predicado *ficheiro-desconhecido* e o seu argumento têm de estar declarados na ontologia *ont-servidor*.

```
Predicate(ficheiro-desconhecido)
Argument(ficheiro-desconhecido, filename, TNome)
```

Supondo que a execução da acção de transferência do ficheiro por ftp anónimo teve sucesso, o agente do servidor tem de informar o agente que fez o pedido que a acção terminou com sucesso. Há duas possibilidades. Se a acção produzir um valor de retorno, esse valor de retorno deverá ser enviado ao agente que fez o pedido. Se a acção não produzir qualquer informação, basta que o agente B diga ao agente A que a acação já foi executada. Consultando a ontologia *ont-servidor*, podemos verificar que o método de acção *enviar-ftp-anonimo* não tem qualquer valor de retorno. Consequentemente, o agente B (agente do servidor) envia uma mesnagem inform ao agente B (que fez o pedido) dizendo que a acção foi executada.

```
(inform
   :sender B
   :receiver (set A)
   :content "(
     (Done
        (action B
          ((iota ?fich (and
             (instance ?fich Ficheiro)
             (= ?fich.filename \"/organizacao/mapa.jpg\"))
          ).enviar-ftp-anonimo
              :hostname \"ftp.iscte.pt\" :directoria \"/publico\"))
     )
   )"
   :language oo-sl
   :ontology (set ont-servidor)
   :protocol fipa-request
   :conversation-id c1203
)
```

**Figura 17 – O agente B informa A que a acção pedida foi executada**

As mensagens que podem ser trocadas nesta conversação, regida pelo protocolo de interacção *fipa-request*, são suportadas na ontologia da Figura 18.

```
Ontology ont-servidor {

   import(
     fipa-agent-management,
http://www.fipa.org/ontologies/ fipa-agent-management.co3l)

   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)

   Predicate(acesso-negado)
   Argument(acesso-negado, filename, TNome)
   Argument(acesso-negado, requestor, agent-identifier)

   Predicate(ficheiro-desconhecido)
   Argument(ficheiro-desconhecido, filename, TNome)
}
```

**Figura 18 – Versão actualizada da ontologia *ont-servidor***

A nova versão da ontologia *ont-servidor* começa por importar a ontologia *fipa-agent-management*, a qual define a classe *agent-identifier* usada no predicado *acesso-negado*. O comando *import* especifica que a ontologia *fipa-agent-management* pode ser obtida em http://www.fipa.org/ontologies/fipa-agent-management.co3l. De facto a ontologia *fipa-agent-management* existe e foi especificada pela FIPA mas não está disponível num servidor publicamente acessível, muito menos na linguagem CO3L.

Na nova versão de *ont-servidor*, foram definidos os predicados *acesso-negado* (usado para dizer que um dado agente não tem permissão para aceder a um dado ficheiro) e *ficheiro-desconhecido* (usado para dizer que o filename especificado não é conhecido do agente do servidor. Estes predicados foram usados na interacção para especificar as razões para o agente do servidor ter recusado a execução da acção de transferência do ficheiro, e para especificar a razão pela qual a acção de transferência do ficheiro por ftp falhou. Num caso real, haverá muito mais razões para que o agente se recuse a executar as suas acções e para que a execução das acções possa falhar. Todas os predicados usados nessas razões e seus argumentos teriam de ser incluídos na ontologia.

As razões para recusar executar uma acção e para que a execução de uma acção falhe foram modeladas através de predicados fora da classe ficheiro. No entanto, os predicados usados poderiam ter sido definidos como métodos relacionais. A ontologia *ont-servidor-oo* mostra essa alternativa.

```
Ontology ont-servidor {

   import(
      fipa-agent-management,
http://www.fipa.org/ontologies/fipa-agent-management.co3l)

   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)
   RelationalMethod(Ficheiro, acesso-negado)
   Argument(Ficheiro.acesso-negado, requestor, agent-identifier)
   RelationalMethod(Ficheiro, ficheiro-desconhecido)
   EntityFacet(Ficheiro.ficheiro-desconhecido, Scope, Classifier)
   Argument(Ficheiro.ficheiro-desconhecido, filename, TNome)
}
```

**Figura 19 – Versão actualizada da ontologia *ont-servidor-oo***

Quando o ficheiro que se pretende receber não é conhecido pelo agente gestor do servidor de ficheiros, esse ficheiro não pode ser encontrado. Portanto, o método relacional *ficheiro-desconhecido* não é aplicável às instâncias da classe, pois teria de ser aplicado a uma instância inexistente. Assim sendo, a ontologia especifica que esse método aplica-se apenas à classe. Isso faz-se com a faceta *Scope* com valor *Classifier*. Adicionalmente, o método relacional tem, como argumento, o nome do ficheiro desconhecido.

Usando, a ontologia *ont-servidor-oo*, as mensagens refuse e failure apresentadas seriam diferentes. A Figura 20 mostra a nova versão da mensagem refuse, usando agora a ontologia *ont-servidor-oo*.

```
(refuse
   :sender B
   :receiver (set A)
   :content "(
     (action B ((iota ?fich (and
        (instance ?fich Ficheiro)
        (= ?fich.filename \"/organizacao/mapa.jpg\"))
     ).enviar-ftp-anonimo
        :hostname \"ftp.iscte.pt\" :directoria \"/publico\"))
     ((iota ?fich (and
        (instance ?fich ficheiro)
        (= ?fich.filename \"/organizacao/mapa.jpg\"))
     ).acesso-negado :requester A)
   )"
   :language oo-sl
   :ontology (set ont-servidor-oo)
   :protocol fipa-request
   :conversation-id c1203
)
```

**Figura 20 – Versão da mensagem refuse, usando a ontologia *ont-servidor-oo***

Como, nesta versão da mensagem, *acesso-negado* é um método relacional das isntâncias da classe *ficheiro*, a primeira coisa que é necessário para o usar, é identificar o objecto a que o método se aplica. A expressão de referência (iota ?fich (and (instance ?fich ficheiro) (= ?fich.filename

\“/organizacao/mapa.jpg\”))) identifica esse objecto, isto é, o ficheiro com filename /organizacao/mapa.jpg.

**Exercício**

Escreve uma troca de mensagens entre o agente A e o agente B em que o primeiro pergunta ao segundo qual é o nome do ficheiro que contém o mapa da organização; B informa A sobre o nome do ficheiro. Para isso, é natural que tenha de se alterar a ontologia.

# 8 Par Acção / Condição num Pedido Persistente

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). Finalmente, mediante a utilização de sistemas de localização dos empregados, o sistema detém sempre informação actualizada sobre as salas em que se encontra cada empregado.

```
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)
}
```

No exemplo, o agente A envia uma mensagem ao agente B requerendo que este o informe dos nomes das pessoas presentes numa sala cujo alarme tenha sido activado, sempre que um alarmes seja activado. A mensagem *request-whenever* permite requerer a execução de uma acção sempre que a condição especificada se verifique. Neste caso, a acção a realizar é o envio de uma mensagem informativa.

```
(request-whenever
   :sender A :receiver (set B)
   :content "(
      (action B
         (inform-ref
            :sender B :receiver (set A)
            :content \"(
               (all (sequence ?x.nome ?x.localizacao.designacao)
                  (and
                     (instance ?x Pessoa)
                     (= ?x.localizacao.alarme activado)))
            )\"
            :language oo-sl
            :ontology (set ontologia-edificio))
      )
      (exists ?sala
         (and (instance ?sala Sala)(= ?sala.alarme activado)))
   )"
   :language oo-sl
   :ontology (set ontologia-edificio))
   :conversation-id rw00010
)
```

**Figura 21 – Mensagem para requerer a execução condicional de uma acção**

Sempre que existir uma sala qualquer com o alarme activado - (exists ?sala (and (instance ?sala sala)(= ?sala.alarme activado))) - enviar uma mensagem *inform* com as identificações das pessoas presentes em salas onde o alarme foi activado – (inform-ref :sender B :receiver (set A) :content \"(all ....)...).

A acção *inform-ref* especifica o envio de determinada informação. Não pode ser usada a mensagem *inform* porque a proposição que serviria de conteúdo à mensagem *inform* não é conhecida na altura em que o *request-whenever* é enviado. Se o agente A (sender) conhecesse de antemão o conteúdo do *inform* (i.e., os nomes das pessoas presentes em salas cujo alarme tivesse disparado), não teria necessidade de pedir ao agente B (receiver) para este lhe enviar essa informação. A acção *inform-ref* permite ultrapassar esta dificuldade. *inform-ref* especifica o envio de uma mensagem *inform* sem necessidade de se conhecer o conteúdo dessa mensagem *inform*. O conteúdo da acção *inform-ref* é uma expressão de referência que especifica justamente a informação que deve ser enviada na mensagem *inform*. Para além de conteúdo (:content), a acção inform-ref tem exactamente os mesmos parâmetros que uma mensagem *inform*. Os parâmetros :sender e :reciever especificam quem envia e quem recebe a mensagem *inform*. Os parâmetros :language e :ontology são exactamente os mesmos que serão usados na mesnagem *inform* que será enviada. A mensagem *inform* que será enviada sempre que um alarme for activado não pertence à conversação iniciada pela mensagem *request-whenever*.

Quando o agente B acredita que um alarme foi disparado numa dada sala, ele envia uma mensagem *inform* com os nomes das pessoas presentes na sala (Figura 22).

```
(inform
   :sender B :receiver (set A)
   :content "(
      (=
         (all (sequence ?x.nome ?x.localizacao.designacao)
            (and
               (instance ?x Pessoa)
               (= ?x.localizacao.alarme activado)))
         (set \"Joao Santos\" \"Ana Silva\" \"Paula Reis\")
   )"
   :language oo-sl
   :ontology (set ontologia-edificio))
)
```

**Figura 22 – Resposta a uma mensagem request-whenever**

*"O conjunto dos objectos que satisfazem a propriedade de serem pessoas localizadas numa sala onde foi disparado um alarme é {"Joao Santos", "Ana Silva", "Paula Reis"}"*.

Além da acção *inform-ref*, existe a acção *inform-if* que é usada para especificar o envio de uma mensagem *inform* dizendo se uma dada proposição é verdadeira ou falsa. Tal como *inform-ref*, a acção *inform-if* tem os mesmos parâmetros que a mensagem *inform* especificada: sender, receiver, content, etc. No entanto, enquanto que o conteúdo de *inform-ref* é uma expressão de referência, o conteúdo de *inform-if* é uma proposição. Com uma acção *inform-if* cujo conteúdo é a proposição P, especifica-se o envio de uma mensagem inform com conteúdo P (se P for verdade) ou de uma mensagem inform com conteúdo not(P) se P for falso.

Às acções *inform-ref* e *inform-if* chama-se mensagens macro por aparecem em vez das verdadeiras mesnagens. Salienta-se que as mensagens-macro não podem ser enviadas. Elas servem apenas para especificar a mensagem que será realmente enviada, mas cujo conteúdo é desconhecido nessa altura.

# 9 Representação de Propostas na Negociação

Existem diversas mensagens ACL que suportam a negociação entre agentes: ***cfp*** (call for proposals), ***propose***, ***accept-proposal*** e ***reject-proposal***. A mensagem ***cfp*** é usada por um agente para solicitar a submissão de propostas para a execução de uma dada tarefa. A mensagem ***propose*** é usada por um agente para efectuar uma proposta de execução de uma dada acção e das condições em que essa acção será executada. ***Accept-proposal*** e ***reject-proposal*** são usadas respectivamente para aceitar e para rejeitar uma proposta. No caso da aceitação, pode ser especificada uma condição de aceitação. No caso da rejeição, é indicada uma razão, isto é, uma proposição que especifica a condição que, por se ter verificado, conduziu à rejeição.

**Cenário de aplicação: "*video-on-demand*"**

O cenário usado nesta secção, o qual serve como exemplo de uma aplicação com negociação entre agentes, já foi abordado a propósito da definição de ontologias. Num cenário ***"video-on-demand"***, assume-se a existência de diversas empresas de televisão. 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 (exibição de programas a pedido). 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, a exibição do vídeo nas condições mais favoráveis.

Neste exemplo imaginado supõe-se a existência de três empresas de distribuição de televisão: TVCabo, CaboVisão e RTPCabo. Cada uma destas empresas terá um agente que a representa. Será considerado apenas um agente de utilizador, o qual será chamado AssistenteVideo. Assume-se que todos os agentes considerados estão registados numa plataforma FIPA específica, na qual existe pelo menos um DF ("Directory Facilitator") chamado DF Audiovisual. Os agentes representantes das empresas distribuidoras de televisão estão registados no DF Audiovisual. Eventualmente, existirão outros DFs, por exemplo um DF para agentes de assinante, mas isso não é importante.

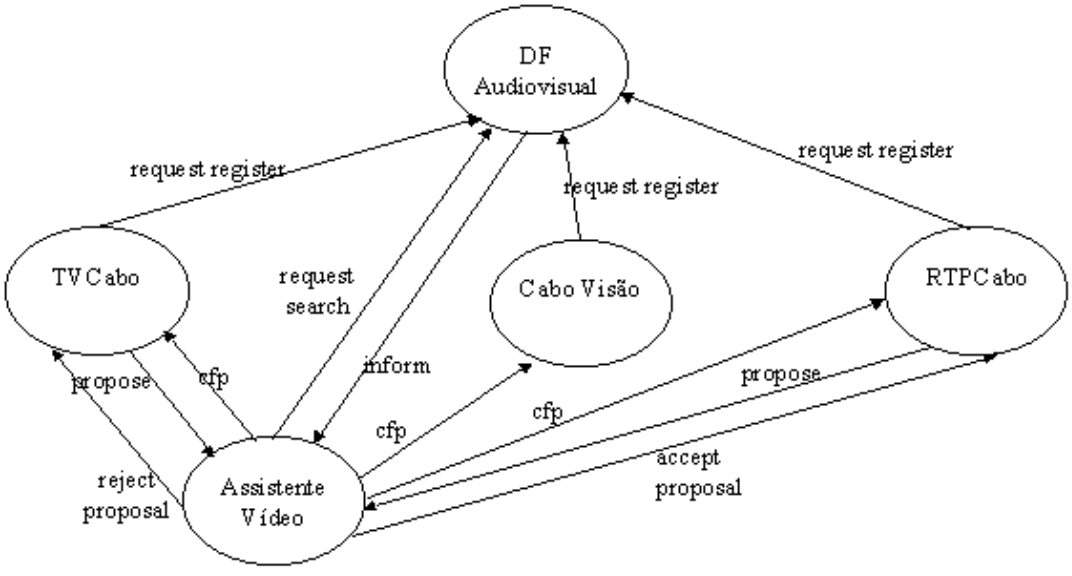


**Figura 23 - Agentes e mensagens no cenário "*video-on-demand*"**

A Figura 23 mostra os agentes envolvidos neste cenário e uma possível troca de mensagens entre eles. No início do seu ciclo de vida, cada agente representante de uma empresa de distribuição de televisão por cabo regista o serviço "*video-on-demand*" no DF Audiovisual. O agente Assistente Vídeo pede ao DF Audiovisual para procurar os agentes que disponibilizam o serviço "*video-on-demand*". Este pedido faz-se através de uma mensagem ***request*** cujo conteúdo é formado pela acção ***search***. O DF Audiovisual informa o Assistente de Vídeo que os agentes RTPCabo, TVCabo e CaboVisão prestam o

serviço "*video-on-demand*". A mensagem usada é um ***inform*** com um conteúdo ***result***. No restante desta secção descrevem-se as mensagens de negociação trocadas entre os agentes depois de o agente Assistente Vídeo ter sido informado pelo DF Audiovisual.

```
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)
}
```

O Assistente Vídeo envia mensagens ***cfp*** aos agentes RTPCabo, TVCabo e CaboVisão pedindo para que estes apresentem as suas propostas para a transmissão do vídeo desejado.

```
(cfp
   :sender AssistenteVideo
   :receiver (set TVCabo)
   :content "(
      (action TVCabo
           ((iota ?filme (and
              (instance ?filme Filme)
              (= ?filme.titulo \"Outubro\")
              (= ?filme.realizador \"Eisenstein\"))).transmitir-video
              :assinante a0001 :data 2009/04/23 :hora 22:00)
      )
      (iota ?filme.preco-transmissao (and
         (instance ?filme Filme)
         (= ?filme.titulo \"Outubro\")
         (= ?filme.realizador \"Eisenstein\")
         (=< ?filme.preco-transmissao 1.5)))
   )"
   :language oo-sl
   :ontology (set ont-video-on-demand)
   :protocol fipa-contract-net
   :conversation-id cN00222
)
```

**Figura 24 - "Call for proposals" pedindo uma proposta para a transmissão de um filme**

Na mensagem da Figura 24, o Assistente Vídeo pede ao agente da TVCabo para apresentar uma proposta para a transmissão do filme Outubro do realizador Eisenstein (no dia 23 de Abril de 2009, pelas 22:00). Como *transmitir-video* é um método dos objectos da classe Filme, é necessário identificar o filme ao qual aplicar o método. A expressão de referência (iota ?filme (and (instance ?filme Filme) (= ?filme.titulo "Outubro") (= ?filme.realizador "Eisenstein")) identifica o filme que se pretende transmitir, isto é, o Outubro de Eisenstein. Como a composição do título com o realizador identificam os filmes univocamente, a expressão de referência usa o operador *iota*. Uma vez identificado o filme, usa-se o operador ponto (.) para invocar o método *transmitir-video*, o qual recebe três parâmetros: o assinante, a data e a hora de transmissão.

A mensagem *cfp* (*call for proposals*) especifica ainda que a proposta a ser apresentada pelo agente TVCabo tem de indicar o preço de transmissão, o qual não pode exceder 1,5 € - (iota ?filme.preco transmissao (and (instance ?filme Filme) (= ?filme.titulo “Outubro”) (= ?filme.realizador “Eisenstein”) (=< ?filme.preco transmissao 1.5))).

Os agentes RTPCabo e TVCabo enviam mensagens ***propose*** com as propostas de transmissão do filme desejado ao Assistente Vídeo.

```
(propose
   :sender TVCabo
   :receiver (set AssistenteVideo)
   :content "(
      (action TVCabo
           ((iota ?filme (and
              (instance ?filme Filme)
              (= ?filme.titulo \"Outubro\")
              (= ?filme.realizador \"Eisenstein\")).transmitir-video
              :assinante a0001 :data 2009/04/23 :hora 22:00)
      )
      (=
         (iota ?filme.preco-transmissao (and
           (instance ?filme Filme)
           (= ?filme.titulo \"Outubro\")
           (= ?filme.realizador \"Eisenstein\")
           (=< ?filme.preco-transmissao 1.5)))
        1.25)
   )"
   :language oo-sl
   :ontology (set ont-video-on-demand)
   :protocol fipa-contract-net
   :conversation-id cN00222
)
```

**Figura 25 – Proposta de transmissão de um filme por 1.25 euros**

A mensagem enviadas pelo agente RTPCabo é análoga à envida pelo TVCabo mas o preço de transmissão é 1 Euro em vez dos 1.25 propostos pelo TVCabo. O agente CaboVisão recusa-se a efectuar a trnamissão porque o cliente não é válido.

```
(refuse
   :sender CaboVisao
   :receiver (set AssistenteVideo)
   :content "(
      (action CaboVisao
           ((iota ?filme (and
              (instance ?filme Filme)
              (= ?filme.titulo \"Outubro\")
              (= ?filme.realizador \"Eisenstein\")).transmitir-video
              :assinante a0001 :data 2009/04/23 :hora 22:00)
      )
      (not (cliente-valido :assinante a0001))
   )"
   :language oo-sl
   :ontology (set ont-video-on-demand)
   :protocol fipa-contract-net
   :conversation-id cN00222
)
```

**Figura 26 – Recusa de transmistir um filme**

A segunda parte do conteúdo da mensagem refuse é a razão pela qual o agente se recusa a executar a acção especificada. A proposição (not (cliente-valido a0001)), que constitui a razão para a recusa,

refere o predicado cliente-valido, o qual não foi descrito na versão inicial da ontologia *ont-video-on-demand*. É necessário incluir o predicado e o seu argumento na nova versão da ontologia:

```
Predicate(cliente-valido)
Argument(cliente-valido, assinante, TNome)
```

Usando um processo de decisão interno, o Assistente Vídeo aceita a proposta do agente da RTPCabo e rejeita a proposta do agente da TVCabo porque o preço não é o mais baixo.

```
(reject-proposal
   :sender AssistenteVideo
   :receiver (set TVCabo)
   :content "(
      (action TVCabo
           ((iota ?filme (and
              (instance ?filme Filme)
              (= ?filme.titulo \"Outubro\")
              (= ?filme.realizador \"Eisenstein\")).transmitir-video
              :assinante a0001 :data 2009/04/23 :hora 22:00)
      )
      (=
         (iota ?filme.preco-transmissao (and
            (instance ?filme Filme)
            (= ?filme.titulo \"Outubro\")
            (= ?filme.realizador \"Eisenstein\")
            (=< ?filme.preco-transmissao 1.5)))
         1.25)
      (not (menor-preco :preco 1.25))
   )"
   :language oo-sl
   :ontology (set ont-video-on-demand)
   :protocol fipa-contract-net
   :conversation-id cN00222
)
```

**Figura 27 – Rejeição da proposta de transmissão do filme**

O conteúdo da mensagem *reject-proposal* é formado por três partes: a acção proposta (transmitir o filme), as condições em que a acção é executada (preço = 1,25 €) e a razão pela qual a proposta é rejeitada ((not (menor-preco 1.25))). As duas primeiras partes do conteúdo são formadas apenas por entidades descritas na ontologia *on-video-on-demand*, mas a terceira parte do conteúdo é uma proposição que utiliza o predicado menor-preco, o qual não foi declarado na ontologia. A nova versão da ontologia terá de incluir o predicado menor-preco e o seu argumento:

```
Predicate(menor-preco)
Argument(menor-preco, preco, TPreco)
```

Na mensagem de aceitação, o AssistenteVideo estabelece que o filme deve ser transmitido no canal 75.

```
(accept-proposal
   :sender AssistenteVideo
   :receiver (set RTPCabo)
   :content "(
      (action RTPCabo
         ((iota ?filme (and
            (instance ?filme Filme)
            (= ?filme.titulo \"Outubro\")
            (= ?filme.realizador \"Eisenstein\")).transmitir-video
              :assinante a0001 :data 2009/04/23 :hora 22:00)
      )
      (= (canal-transmissao) 75)
   )"
   :language oo-sl
   :ontology (set ont-video-on-demand)
   :protocol fipa-contractNet
   :conversation-id cN00222
)
```

**Figura 28 – Aceitação de uma proposta de transmissão do filme**

A partir do momento em que o agente da RTPCabo recebe a mensagem de aceitação da Figura 28, fica obrigado a transmitir o Outubro de Eseinstein para o assinante representado pelo AssistenteVideo.

O conteúdo da mensagem *accept-proposal* tem duas partes: a acção proposta (transmitir o filme) e uma condição adicional de aceitação (canal de transmissão = 75). A primeira parte do conteúdo refere apenas entidades descritas na ontologia *ont-video-on-demand*, mas a proposição (= (canal-trasmissao) 75) contém a função *canal-transmissao* que não pertence à versão original da ontologia. Para que esta mensagem possa ser compreendida a nova versão da ontologia tem de incluir também a função *canal-transmissao*:

```
Function(canal-transmissao, Natural).
```

A apresenta a nova versão da da ontologia ont-video-on-demand, de acordo com as exisgências da interacção descrita nesta secção.

```
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)

   Predicate(cliente-valido)
   Argument(cliente-valido, assinante, TNome)

   Predicate(menor-preco)
   Argument(menor-preco, preco, TPreco)

   Function(canal-transmissao, Natural)
}
```

**Figura 29 – Nova versão da ontologia *ont-video-on-demand***

A nova versão da ontologia contém apenas os ingrediantes estritamente necessários ao exemplos de mensagens apresentados nesta secção. No entanto, há muito mais razões pelas quais um agente representante de uma companhia de televisão pode recusar a transmissão de um vídeo para um dado cliente, há muito mais razões para rejeitar propostas, e há muitas outras condições adiccionais de aceitação de propostas. A ontologia tem de contemplar tudo isso.

O domínio da aplicação descrita poderia ter sido modelado de uma forma mais "pura" fazendo uso exclusivo da abordagem orientada por objectos. Nessa versão alternativa deixariam de existir os predicados *cliente-valido*, *menor-preco* e a função *canal- transmissão* desgarrados das classes.

Em vez disso, supõe-se que o domínio inclui a classe *cliente* que representa os clientes de uma televisão, e a classe *resultado-negociado* com os atributos *preco* e *canal-transmissão* entre outros.

```
Ontology ont-video-on-demand-oo {

   /* Definição de tipos de dados mais específicos */
   Datatype(TNome, String)
   EntityFacet(TNome, Instance_maximum_size, 60)
   Datatype(TPreco, Float)
   EntityFacet(TPreco, Smallest_instance, 0)

   /* Definição de Classes, Atributos e Métodos */

   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)

   Class(Cliente)
   Attribute(Cliente, num-assinante, Word)
   EntityFacet(Cliente.num-assinante, Mandatory, True)
   Attribute(Cliente, nome, TNome)
   EntityFacet(Cliente.nome, Mandatory, True)
   Attribute(Cliente, ultima-quota-paga, Date)
   EntityFacet(Cliente.ultima-quota-paga, Mandatory, True)
   RelationalMethod(Cliente, quotas-pagas)

   Class(Resultado-Negociado)
   Attribute(Resultado-Negociado, assinante, Cliente)
   EntityFacet(Resultado-Negociado.assinante, Mandatory, True)
   Attribute(Resultado-Negociado, video, Filme)
   EntityFacet(Resultado-Negociado.video, Mandatory, True)
   Attribute(Resultado-Negociado, preco, TPreco)
   Attribute(Resultado-Negociado, canal-transmissao, Natural)
   Attribute(Resultado-Negociado, data-transmissao, Date)
   Attribute(Resultado-Negociado, hora-transmissao, Time)
}
```

**Figura 30 – Versão OO pura da ontologia *ont-video-on-demand-oo***

A versão completamente orientada por objectos da ontologia é a que se representa na Figura 30.

A título de exemplo da utilização da versão orientada por objectos da ontologia (*ont-video-on-demand-oo)*, apresentamos a nova versão da mensagem *reject-proposal* (Figura 31). As mensages *refuse* e *accept-proposal* teriam igualmente novas versões.

```
(reject-proposal
   :sender AssistenteVideo
   :receiver (set TVCabo)
   :content "(
      (action TVCabo
        ((iota ?f (and
             (instance ?f Filme)
             (= ?f.realizador Eisenstein)
             (= ?f.titulo Outubro))).transmitir-video
               :assinante a0001 :data 2009/04/23 :hora 22:00)
      )
      (= (iota (?f.preco-transmissao)(and
           (instance ?f Filme)
           (= ?f.realizador Eisenstein)
           (= ?f.titulo Outubro))
           (<= ?f.preco-transmissão 1.5)))
          1.25
      )
      (exists ?r (and
        (instance ?r Resultado-Negociado)
        (= ?r.assinante.num-assinante a0001)
        (= ?r.video.titulo \"Outubro\")
        (= ?r.video.realizador \"Eisenstein\")
        (= ?r.data-transmissao 2009/04/23)
        (= ?r.hora-transmissao 22:00)
        (< ?r.preco 1.25)
   )"
   :language oo-sl
   :ontology (set ont-video-on-demand-oo)
   :protocol fipa-contractNet
   :conversation-id cN00222
)
```

**Figura 31 – Mensagem *reject-proposal* com a versão OO pura da ontologia**

O conteúdo da mensagem *reject-proposal* da Figura 31 tem uma razão (a razão pela qual a proposta é rejeitada) mais clara do que a razão da mensagem correspondente, usando a versão híbrida da ontologia (Figura 29). Na versão OO, torna-se mais claro que o preço que resulta da negociação (atributo *preco* da classe *Resultado-Negociado*), seja ele qual for, é mais baixo do que 1.25 que é o valor da proposta apresentada pelo agente representante da TVCabo.

As ontolologias que serviram de suporte às interacções descritas até esta altura incluem classes, predicados e funções. Nenhuma delas contém uma associação entre classes. A próxima interacção é suportada por uma ontologia com uma associação entre classes.

# 10 Pergunta aberta usando uma associação entre classes

O cenário relativo aos cursos e disciplinas de uma escola universitáriapode ser modelado pela ontologia *ont-cursos-disciplinas.*

Além de outras entidades, a ontologia descreve as classes *Curso*, *Licenciatura*, *Mestrado* e *Doutoramento*, e a classe *Disciplina*. *Licenciatura*, *Mestrado* e *Doutoramento* são subclasses de *Curso*, pelo que herdam todos os seus atributos e métodos. Além destas classes e da hierarquia de cursos, a ontologia inclui também uma associação entre cursos e disciplinas - *Curso-Disciplina* – a qual relaciona um curso com todas as suas disciplinas, e uma disciplina com todos os cursos a que pertence. Para facilitar a explicação da interacção deste exemplo, apresenta-se seguidamente os aspectos mais significativos da ontologia *ont-cursos-disciplinas*.

```
Ontology ont-cursos-disciplinas {

   ...

   Class(Curso)
   EntityFacet(Curso, Materialization, Abstract)
   Attribute(Curso, sigla, Word)
   Attribute(Curso, nome, nome)
   Attribute(Curso, nvagas, Natural)
   Attribute(Curso, responsavel, nome)
   ...

   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))
   ...

}
```

O principal propósito da interacção analisada nesta secção é o de mostrar como se usa uma associação entre classes e a herança estabelecida numa hierarquia entre classes.

Na interacção que nos serve de exemplo, o agente representante de um candidato ao ensino superior (A) pede ao agente gestor dos cursos e disciplinas de uma escola universitária particular (B) para este

lhe dizer quais as disciplinas do primeiro semestre do primeiro ano da Licentiatura em Engenharia Informática (LEI).

```
(query-ref
   :sender A  :receiver (set B)
   :content "(
      (all ?cd.disciplina.id (and
         (instance ?cd Curso-Disciplina)
         (= ?cd.curso.sigla LEI)
         (= ?cd.semestre 1)
         (= ?cd.ano 1)))
   )"
   :language oo-sl
   :ontology (set ont-cursos-disciplinas)
   :protocol fipa-query
   :conversation-id c03307
)
```

**Figura 32 – Pergunta aberta usando uma associação entre classes**

A primeira coisa a dizer sobre o conteúdo da mensagem da Figura 32, é que para todos os efitos, uma associação entre classes pode ser tratada exactamente como se fosse uma classe. Em particular, o predicado instance pode ser usado para aceder às instâncias da associação exactamente da mesma forma como acede aos objectos de uma classe. A expressão (instance ?cd Curso-Disciplina) relaciona a associação *Curso-Disciplina* com as suas instâncias.

Olhando para o conteúdo da mensagem *query-ref*, vemos também que o operador ponto (.) serve quer para associar um um argumento a uma instância de uma associação (e.g., ?cd.curso, ?cd.disciplina), quer para associar um atributo (ou um método) a uma instância (e.g., ?cd.semestre, ?cd.ano, ?cd.disciplina.id).

Finalmente, os efeitos do mecanismo de herança, neste exemplo, é que embora *curso* seja uma classe abstracta (faceta Materialization com valor Abstract), isto é, é uma classe sem instâncias explicitas, a mensagem faz referência a uma das suas instâncias (mais propriamente, uma instância de *licenciatura*), a qual herda todos os atributos de *curso* e a associação entre *curso* e *disciplina*.

# 11 Pergunta fechada num modelo sem objectos

Esta secção ilustra a realização de uma pergunta fechada suportada numa ontologia sem classes, nem objectos, nem atributos. Serve-nos de exemplo é um cenário em que uma dada Bolsa de Valores dispõe de um sistema de informação controlado por um agente que pode interactuar com outros agentes, por exemplo agentes representantes de investidores. Este cenário foi modelado pela ontologia *ontologia-bolsa*. Ao contrário das ontologias usadas até ao momento, esta não contém classes nem atributos nem métodos. *A*presentar-se seguidamente um excerto da ontologia *ontologia-bolsa*.

```
Ontology ontologia-bolsa {

   ...

   Function(Cotacao, TCotacao)
   Argument(Cotacao, Accao, String)

   ...
}
```

Na interacção que se segue, um agente representante de um investidor (A) pergunta ao agente da bolsa (B) se é verdade que a cotação das acções da PT é de 20€. Dado que se trata de uma pergunta fechada, usa-se a mensagem query-if, cujo conteúdo é uma proposição.

```
(query-if
   :sender A :receiver (set B)
   :content "(
      (= (Cotacao \"pt\") 20)
   )"
   :language oo-sl
   :ontology (set ontologia-bolsa)
   :protocol fipa-query
   :conversation-id c0101
)
```

**Figura 33 – Pergunta aberta usando uma função**

A função *Cotacao* devolve a cotação instantânea de uma dada acção. (Cotacao \"pt\") é uma expressão funcional que representa a cotação instantânea das acções da PT. Como o conteúdo da mensagem *query-if* é uma proposição, usa-se o predicado de igualdade que relaciona (Cotacao \"pt\") com 20€.

Dado que a cotação das acções da PT não é20€, a resposta do agente da bolsa (B) é a negação da proposição da pergunta.

```
(inform
   :sender B :receiver (set A)
   :content "(
      (not (= (Cotacao \"pt\") 20))
   )"
   :language oo-sl
   :ontology (set ontologia-bolsa)
   :protocol fipa-query
   :conversation-id c0101
)
```

**Figura 34 – Resposta a uma pergunta aberta**

Se a cotação das acções da PT fosse 20€, o conteúdo da resposta seria apenas (= (Cotacao \"pt\") 20).

# 12 Referências Bibliográficas

[FIPA 2002-08] Foundation for Intelligent Physical Agents. 2002. “FIPA SL Content Language Specification”. Report 00008. http://www.fipa.org/specs/fipa00008/