PMR2550 - Projeto de Conclusão do Curso II

Monografia Final

Sistema de Gerenciamento de Diálogo com Memória Semântica

Edson Rosa Nakada
Lucas Moraes Pinheiro

Orientador: Prof. Dr. Marcos Ribeiro Pereira-Barretto

19 de novembro de 2015

Monografia apresentada no Departamento de Engenharia Mecatrônica e Sistemas Mecânicos da Escola Politécnica da Universidade de São Paulo para obtenção do título de Engenheiro.

Área de Concentração: Engenharia Mecatrônica.

Declaração de Originalidade

Este relatório é apresentado como requisito parcial para obtenção do grau de Engenheiro Mecatrônico na Escola Politécnica da Universidade de São Paulo. É o produto do meu próprio trabalho, exceto onde indicado no texto. O relatório pode ser livremente copiado e distribuído desde que a fonte seja citada.

Resumo

Avanços tecnológicos na área de inteligência artificial permitem interações mais humanas com as máquinas. Entretanto, sistemas que podem interagir com pessoas por meio de conversas ainda se mostram limitados. É por esta motivação que este trabalho propôs o desenvolvimento de um *framework* genérico, capaz de manter uma conversa natural com seus usuários. O foco principal do projeto foi o módulo do Gerenciador de Diálogo, responsável por controlar o andamento da conversa. Foram implementados três tipos de memória: semântica, procedural e episódica, sendo que o foco maior foi no desenvolvimento da estrutura das memórias semântica e procedural.

Ao final do trabalho, apresentou-se um exemplo de aplicação com o contexto de pedidos de pizza. O sistema foi capaz de satisfazer os requisitos propostos e obteve bons resultados durante os testes. Ele foi capaz de manter um baixo tempo de resposta e recebeu boas avaliações dos voluntários que o testaram.

Abstract

Technological advaances in the field of artificial intelligence allow for a more humanized interaction with machines. However, systems that can interact with people through normal conversations are still very limited. This is the reason why this paper proposes the development of a generic framework capable of keeping a natural conversation with its users. This project's main focus is the Dialog Manager module, which is responsible for controlling the conversation's flow. Three kinds of memory were implemented to support this module: episodic, semantic and procedural memory, with the last two having a higher priority during the development.

At the end of this paper, an example application for ordering pizzas is shown. The system was able to satisfy the proposed requirements and got good results during the tests. The application was able to keep a short response time and received positive feedback from the volunteers who tested it.

Agradecimentos

Ao Prof. Dr. Marcos Ribeiro Pereira-Barretto, pela orientação e o incentivo durante todo o projeto.

Aos integrantes do Academic Working Capital, em especial ao Eduardo Bonilha, pelo suporte financeiro e teórico, que nos ajudaram a guiar nossa visão de negócios.

Aos nossos pais, por nos darem todo o suporte para alcançarmos nossos sonhos.

À Marina Salles, pelo constante apoio e paciência, e por sempre acreditar que era possível irmos longe.

Aos Politrônicos, por nos ajudarem durante toda a graduação.

Siglas

AIML *Artificial Intelligence Markup Language*.

ASR *Automatic Speech Recognition*.

GD Gerenciador de Diálogo.

GL Gerador de Linguagem.

IRI *International Resource Identifier*.

JADE *JAVA Agent DEvelopment Framework*.

OWL *Web Ontology Language*.

RDF *Resource Description Framework*.

TTS *Text-To-Speech*.

URI *Uniform Resource Identifier*.

XML *Extensible Markup Language*.

Sumário

1 Introdução **3**

1.1 Motivação . . . 3
1.2 Objetivo . . . 3
1.3 Estrutura do Trabalho . . . 4

2 Revisão e Estado da Arte **4**

2.1 Jindigo . . . 6
2.2 RavenClaw . . . 6
2.3 Atos Dialogais . . . 7

3 Tecnologias Usadas **8**

3.1 RDF . . . 8
3.2 OWL . . . 9
3.3 AIML . . . 10
3.4 JADE . . . 12
3.5 ASR . . . 13
3.6 TTS . . . 14

4 Testes com ASR e TTS **14**

4.1 Coruja . . . 14
4.2 Web Speech API . . . 15
4.3 Microsoft Speech Platform . . . 15
4.4 Sistema de Fala Escolhido . . . 16

5 Análise de Requisitos **17**

6 Solução Proposta **17**

6.1 Arquitetura . . . 18
6.2 Normas e Especificações . . . 18
6.2.1 Comunicação entre agentes . . . 18
6.2.2 Atos dialogais . . . 19
6.3 Semantizador . . . 20
6.4 Gerador de Linguagem . . . 22
6.5 Gerenciador de Diálogo . . . 22
6.5.1 Controlador de Novas Mensagens . . . 24
6.5.2 Unidade de Obrigações Sociais . . . 24
6.5.3 Unidade Padrão de Erros . . . 25
6.5.4 Controlador da Pilha de Requisições . . . 25
6.5.5 Controlador do Fluxo de Diálogo . . . 26
6.6 Memória Semântica . . . 26

7 Exemplo de Aplicação **27**

7.1 Análise de Requisitos da Aplicação . . . 27
7.2 Ontologias Desenvolvidas . . . 28
7.2.1 Ontologia de Alimentos (“Consumíveis”) . . . 28
7.2.2 Ontologia do Menu do Restaurante . . . 29
7.2.3 Ontologia de Atendimento de Pedidos . . . 29

7.3 Implementações Necessárias . . . 30
7.3.1 Unidade de Pedidos . . . 31
7.3.2 Unidade de Cancelamento de Itens de Pedido . . . 31
7.3.3 *Helper* de Confirmação do Pedido. . . . 32
7.4 Avaliação da Solução . . . 32
7.4.1 Tempo de Resposta do Sistema . . . 32
7.4.2 Grau de Entendimento . . . 34
7.4.3 Percepção do Usuário . . . 34
7.5 Exemplos de Transcrição . . . 35
7.5.1 Exemplo 1 . . . 35
7.5.2 Exemplo 2 . . . 35
7.5.3 Exemplo 3 . . . 36

8 Conclusões **36**

9 Referências **38**

Apêndice A Diagrama de Componentes **40**

Apêndice B Protocolo de Comunicação de Agentes **41**
B.1 Communication Performative . . . 41
B.1.1 Communication: ASR Module - Language Understanding . . . 41
B.1.2 Communication: Language Understanding - Memory . . . 41
B.1.3 Communication: Language Understanding - Dialog Manager . . . 42
B.1.4 Communication: Dialog Manager - Language Understanding . . . 42
B.1.5 Communication: Dialog Manager - Memory . . . 43
B.1.6 Communication: Dialog Manager - Language Generation . . . 44

Apêndice C Atos Dialogais **45**

Apêndice Dialog Act Standards **45**
Iteration 1 . . . 45
Dialog Dimensions . . . 45
Dialog Functions . . . 45
Dimension: Procedural . . . 46
Dimension: Allo-Feedback . . . 48
Dimension: Social Obligations Management . . . 48
Dimension: Partner Communication Management . . . 48
Examples . . . 48
Basic dialog test . . . 49

1 Introdução

1.1 Motivação

Avanços tecnológicos na área de inteligência artificial permitiram interações mais humanas com as máquinas. Entretanto, sistemas que podem se relacionar com pessoas por meio de conversas ainda se mostram limitados. Em sua maioria, restringem-se a comandos (como no caso da Siri), sem que haja um fluxo de diálogo. Por outro lado, sistemas que permitem esse fluxo não são capazes de lidar com situações mais gerais, em que variam as possibilidades de mensagens trocadas. Este projeto visa solucionar esse problema com a criação um sistema que possa manter uma conversa natural com o usuário, similar a diálogos entre seres humanos.

Dentre as pesquisas na área de robôs sociáveis, podemos citar o Jibo [1], lançado este ano. que trata-se de uma nova plataforma de desenvolvimento, na qual o projeto poderia participar como um aplicativo disponibilizado na Jibo Store.

1.2 Objetivo

O objetivo deste trabalho é desenvolver um sistema capaz de interagir com um usuário por meio de diálogos falados. O foco principal encontra-se no desenvolvimento do Gerenciador de Diálogo, responsável por controlar o andamento da conversa e os demais módulos do sistema. O projeto final poderá servir como base para outros sistemas que envolvam o gerenciamento de diálogos, como a reserva automatizada de salas em um prédio comercial (ou mesmo em uma universidade).

O módulo de gerenciamento de diálogo (bem como os demais módulos que o dão suporte) é um tópico recorrente, como visto em [1], [2], [3]. Dentre os *frameworks* já desenvolvidos para esta tarefa, foram analisadas duas soluções, o Ravenclaw [4] e o Jindigo [5]. A leitura da documentação de ambas possibilitou uma maior visão de como o problema poderia ser abordado, conforme mostrado na seção 2.

Para guiar o desenvolvimento do projeto, decidiu-se como tema para a solução proposta o atendimento para pedidos de pizza. Tal escolha representa uma abordagem concentrada em um problema específico, de forma a permitir maior foco nos casos de uso relevantes. Dessa forma, o sistema poderá posteriormente ser ampliado para incluir atendimentos em outras áreas. E inicia-se tendo como mercado total, só na cidade de São Paulo, cerca de 4500 pizzarias [6].

Foram feitas também pesquisas com clientes de pizzarias, com resultados promissores. Todos os entrevistados afirmaram que utilizariam sistemas como o proposto, desde que ele funcionasse corretamente. Além disso, foi citado que um assistente virtual poderia ter a vantagem de não sofrer com ruídos do ambiente, como costuma acontecer no atendimento por funcionários.

[1] `http://www.jibo.com`

1.3 Estrutura do Trabalho

Este trabalho é organizado como descrito a seguir: na Seção 2 faz-se uma revisão sobre o estado da arte; em 3 são descritas as ferramentas e tecnologias utilizadas para o desenvolvimento do sistema; em 4 são detalhados os teste realizados com as ferramentas de reconhecimento e síntese de voz; em 5 explora-se os requisitos do projeto; em 6 é descrita a arquitetura da solução proposta; em 7 descreve-se a aplicação específica para pedidos de pizza, com sua própria análise de requisitos e testes; finalmente, em 8 discute-se os resultados obtidos com o projeto e analisa-se possíveis melhorias para trabalhos futuros.

2 Revisão e Estado da Arte

Para possibilitar o diálogo entre um ser humano e um atendente virtual de forma mais natural, é necessário um sistema composto por uma série de módulos, conectados em uma estrutura encadeada [2], [4]. A arquitetura básica de sistemas deste tipo (ilustrada na Fig. 1) é dada da seguinte forma:

- **Reconhecimento da fala:** o módulo de *Automatic Speech Recognition* (ASR) - ou Reconhecimento Automático da Fala, em português - transforma o áudio (com ruídos) em uma ou mais palavras. Estas palavras são então enviadas para o módulo do semantizador;
- **Semantizador:** módulo de processamento natural de linguagem que analisa o texto obtido pelo ASR e obtém o significado semântico da frase;
- **Gerenciador de Diálogo (GD):** também chamado de contextualizador, é nele que o significado semântico é interpretado de acordo com o contexto atual da conversa. Este componente é o responsável pelo controle e a coordenação dos demais;
- **Gerador de Linguagem (GL):** o contexto processado é passado para o Gerador de Linguagem (ou Gerenciador de Ações), o qual cria e envia para o *Text-To-Speech* (TTS) o plano de fala a ser sintetizado;
- **Síntese da fala:** o TTS executa a fala de saída de acordo com os resultados das etapas anteriores.

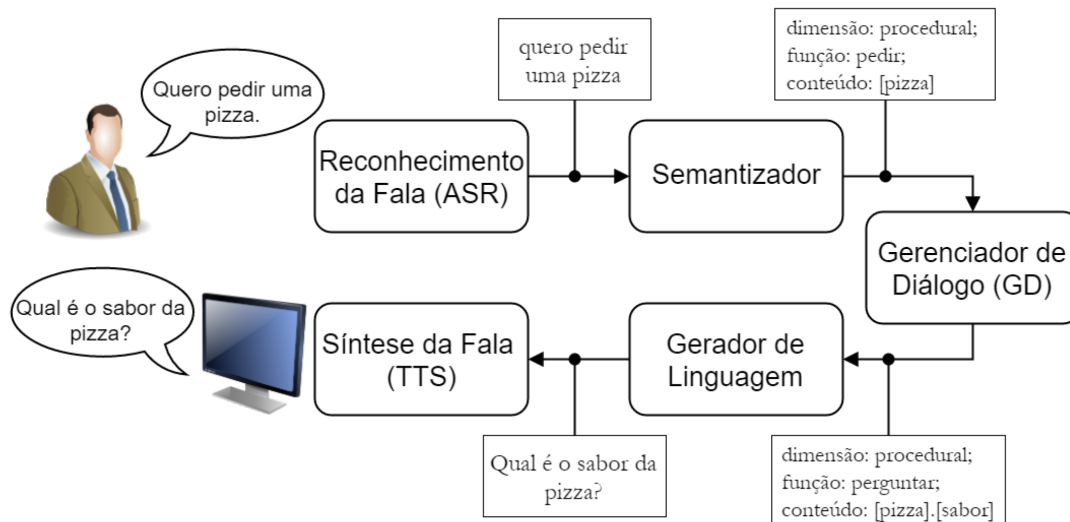


Figura 1: Esquema representando a arquitetura básica de um sistema de diálogo. Autoria própria.

Outros aspectos devem ser considerados ao se definir qual será a abordagem para a construção de um sistema de diálogo. A seguir será feita uma breve descrição de dois deles, conforme visto em mais detalhes em [2]. Uma primeira característica importante é o grau de iniciativa, que descreve o funcionamento do fluxo de informações entre o usuário e o sistema. Há três categorias básicas possíveis:

- Iniciativa do sistema: cada etapa é guiada pelo sistema;
- Iniciativa do usuário: o sistema apenas responde às informações fornecidas pelo usuário;
- Iniciativa mista: embora o sistema tenha um controle geral do diálogo, seu andamento pode ser alterado de acordo com o que é dito pelo usuário.

A escolha de qual método utilizar depende da aplicação desejada. Como o objetivo é uma conversa que deve ser a mais natural possível, será adotado o terceiro método, de forma a permitir que informações sejam inseridas a qualquer momento e em qualquer ordem.

O segundo aspecto importante refere-se ao tratamento de erros. Embora atualmente muito avançados, sistemas de ASR ainda estão sujeitos a erros na detecção das palavras ditas (causados por ruídos, sons ambientes, etc). Pode-se estabelecer, portanto, um grau de confiança para o texto obtido. Dependendo desse grau, diferentes estratégias de recuperação de erro podem ser utilizadas: confirmação explícita, confirmação implícita, refraseamento estático e refraseamento dinâmico.

A modelagem do diálogo pode ser feita de três formas: sistemas baseados em grafos, em formulários e em agendas [1]. O primeiro tem limitações quanto ao seu tamanho, uma vez que possui estrutura em árvore, a qual limita as possibilidades de diálogo e demanda a criação de vários estados que preveem a ocorrência de todos os eventos. Já sistemas baseados em formulários proporcionam uma alternativa mais flexível, pois não demandam uma ordem específica para o fornecimento das informações. Todavia, as soluções para tais sistemas tornam-se muito específicas, e não podem ser utilizadas em casos nos quais há

grande variação no conteúdo de cada conversa. Sistemas baseados em agendas resolvem esses e outros problemas ao adotarem produtos dinâmicos em estruturas de árvores, e uma agenda que determina um plano geral para o andamento da conversa. As árvores são compostas por nós, os quais possuem frases que podem ser ditas para a obtenção de uma informação. No caso de informações complexas, os nós filhos representam partes mais simples da informação do nó pai. Uma explicação detalhada está disponível no trabalho de Rudnicky & Xu [1].

2.1 Jindigo

Foram analisados trabalhos que tratam de *frameworks* para a criação do GD (bem como dos demais módulos necessários). Dentre eles, podemos destacar o trabalho de Skantze & Hjalmarsson [5], o qual descreve o Jindigo, um framework em Java baseado no modelo de processamento incremental proposto por Schlangen & Skantze [3]. A vantagem de se utilizar um sistema incremental ao invés de um sistema não incremental está no fato de que, em um sistema não incremental, deve-se esperar que o falante termine seu turno, para então começar o processamento daquilo que foi dito e poder-se chegar em uma resposta. Se esse processamento for muito demorado, haverá problemas com o tempo de resposta do sistema.

O modelo de processamento incremental utilizado pelo Jindigo consiste em uma rede de módulos de processamento. Cada módulo possui um *buffer* esquerdo, um processador e um *buffer* direito que se comunicará com o *buffer* esquerdo do próximo módulo. Estes módulos trocam entre si unidades incrementais (UIs), correspondentes aos menores pedaços de informação que podem ativar os módulos (como palavras, frases, etc).

Como no processamento incremental nem sempre trabalhamos com frases completas, o módulo de entendimento da linguagem torna-se extremamente importante. É neste módulo que se faz uma previsão sobre o que está sendo falado pelo usuário. Quanto mais precisa a previsão, menos processamento será gasto com reparos mais a diante.

Neste *framework*, o GD, além de interpretar a frase obtida de acordo com o contexto, também monitora a saída do sistema para saber se o usuário está concordando com o que está sendo dito. Caso isto não seja verdade, o sistema deve revisar o plano de fala (*Speech Plan*). O maior desafio dos gerenciadores de diálogo incrementais está nas revisões (ou reparos). Como a fala do usuário é processada em pequenos incrementos, todos os módulos trabalham com previsões do que está sendo dito e muitas vezes estas hipóteses mostram-se erradas e o sistema deve rever suas informações. Para isso, cada módulo deve trabalhar com três situações: a inclusão de uma UI que gera o processamento da mesma, a revogação de uma UI que induz a revisão da saída desse módulo e o compromisso (*commit*) com uma UI responsável por indicar que aquela hipótese não será mais revisada.

2.2 RavenClaw

Outro *framework* que será analisado é o RavenClaw, descrito no trabalho de Bohus & Rudnicky [4]. Ele consiste em um modelo de gerenciamento baseado em planos e independente da tarefa almejada. Isso é possível pois o RavenClaw possui uma clara separação entre os aspectos específicos do campo de aplicação adotado e os aspectos independentes

(gerais). Ou seja, tarefas como tratamento de erros, o tempo de espera e a troca de turnos entre o usuário e o sistema não demandam um grande esforço dos desenvolvedores, os quais podem se focar em criar a lógica de gerenciamento de diálogo.

O RavenClaw utiliza uma modelagem baseada em agendas. Elas representam metas, informações que devem ser obtidas, as quais podem ser compostas por outras informações. Desta forma, possui a característica de ser capaz de adaptar-se mais facilmente a informações de conteúdo variado, como é o caso deste projeto. A estrutura de sistemas deste tipo assemelha-se a árvores, cujos nós representam os dados requisitados.

2.3 Atos Dialogais

Para analisar semanticamente o diálogo, ser capaz de extrair significado coerente e entendê-lo, é preciso que adote-se um padrão que permita a todos os componentes do sistema se comunicarem. Com este intuito, baseou-se o desenvolvimento do projeto na norma ISO 24671-2, a qual foi criada tendo em vista a necessidade de uma anotação com bom embasamento teórico e empírico, que possa lidar tanto com linguagem falada como escrita ou multimodal e que possa ser usada tanto por humanos como por métodos automáticos [7]. Ela também define uma representação a ser utilizada como referência, a DiAML (*Dialogue Act Markup Language*, ou Linguagem de Marcação de Ato Dialogal, em português).

Dentro da norma ISO 24671-2, atos dialogais são considerados como sendo operadores para atualização dos estados de informação dos participantes do diálogo [8], ou seja, a forma como o falante deseja influenciar a situação de interação [7]. Assim, eles possuem dois componentes: o conteúdo semântico, o qual descreve eventos, objetos e propriedades, e a função comunicativa, que define como o conteúdo deve ser utilizado. Este segundo pode ser dividido em dimensão e função. As dimensões são definidas como aspectos ortogonais da comunicação com os quais o ato dialogal pode estar envolvido. Logo, uma fala pode ter uma função em mais de uma dimensão, mas não mais que uma função em uma dada dimensão [8].

A seguir tem-se uma breve descrição das nove dimensões dialogais definidas na ISO 24671-2, conforme pode ser visto em [9].

- Tarefa/Procedural (*Task*): está relacionada com a tarefa principal do sistema. No caso deste projeto, envolve o pedido de pizzas.
- Auto-Feedback: fornece informações sobre como o falante processou suas próprias falas anteriores.
- Allo-Feedback: fornece informações sobre como o falante processou falas anteriores de outrem.
- Gestão de Turnos (*Turn Management*): relaciona-se com as mudanças no papel de falante.
- Gestão de Tempo (*Time Management*): atos para lidar com o uso de tempo na interação.
- Estruturação do Discurso (*Discourse Structuring*): atos que envolvam a estrutura do diálogo, como por exemplo a mudança de tópicos.

- Gestão da Própria Comunicação (*Own Communication Management*): indica que o falante está editando sua própria fala anterior.
- Gestão da Comunicação do Parceiro (*Partner Communication Management*): indica que o falante tem intenção de editar (completar, auxiliar) a fala de outrem.
- Gestão de Obrigações Sociais (*Social Obligations Management*): atos que lidam com convenções sociais, como cumprimentos, agradecimentos, pedidos de desculpa etc.

3 Tecnologias Usadas

3.1 RDF

O *Resource Description Framework* (RDF) é o modelo padrão de troca de informação na Web. Ele foi criado para ser usado em situações em que a informação precisa ser processada por aplicativos ao invés de ser apresentada para pessoas [10]. Muitas outras linguagens são escritas em RDF, como é o caso do SKOS (*Simple Knowledge Organization System*) e do *Web Ontology Language* (OWL).

O RDF utiliza *International Resource Identifiers* (IRIs) para identificar as coisas. IRI é uma generalização de *Uniform Resource Identifier* (URI), permitindo que caracteres não pertencentes ao conjunto de caracteres ASCII sejam usados [11]. A estrutura básica de uma afirmação em RDF é dividida em três partes: sujeito, predicado e objeto. Estas afirmações podem ser ainda representadas por um grafo orientado, no qual sujeito e objeto correspondem a nós e o predicado é aparece como um arco orientado que liga o sujeito ao predicado.

Um exemplo do uso do RDF é dado na Figura 2. Este exemplo representa os metadados de uma página na *internet*. Do grafo pode-se obter as seguintes informações: a página foi criada pelo funcionário referenciado por *http://www.example.org/staffid/85740*, no dia 16 de Agosto de 1999, e está escrita em inglês. Os nós quadrados indicam que o conteúdo não é um IRI, mas sim um valor fixo.

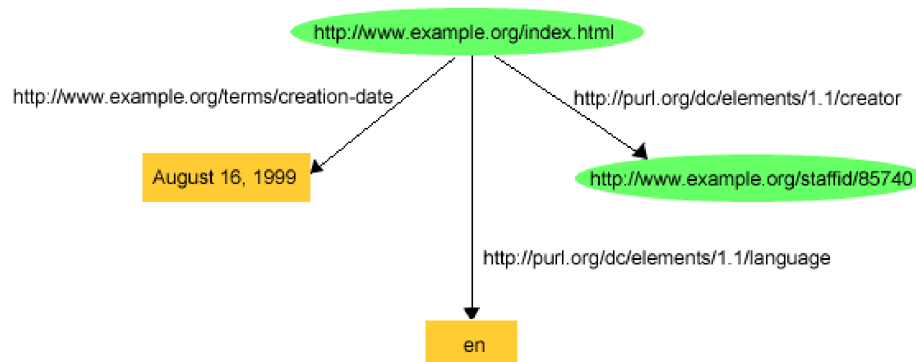


Figura 2: Exemplo de representação por grafo de uma afirmação em RDF. Reproduzida de [10].

Uma outra característica do RDF é que ele utiliza o *Extensible Markup Language* (XML) para escrever suas afirmações de maneira que o computador possa processar mais facilmente. Esta sintaxe leva o nome de RDF/XML. Um exemplo da sintaxe que descreve a afirmação "http://www.exemplo.com/index.html foi criada por Joaquim." seria:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
    <rdf:Description rdf:about="http://www.examplo.com/index.html">
        <dc:creator>Joaquim</dc:creator>
    </rdf:Description>
</rdf:RDF>
```

No exemplo podemos observar a declaração de dois *namespaces* em XML representados pelo atributo *xmlns*. Isto permite compactar o documento. A estrutura <rdf:Description rdf:about="IRI» indica que estamos descrevendo determinada IRI. E <dc:creator> é referência para o elemento descrito por *http://purl.org/dc/elements/1.1/creator*. Caso o criador da página não fosse descrito por um valor, mas sim por uma IRI, a linha se tornaria <dc:creator rdf:resource="IRI"/>.

3.2 OWL

Para que o sistema seja capaz de lidar com conceitos abstratos (como pedidos, comidas e tamanhos, entre outros) torna-se imprescindível a utilização de uma forma de representar o conhecimento. Isso inclui não só as características dos elementos, mas também as relações possíveis entre cada par deles. Uma solução para este problema é a *Web Ontology Language* (OWL), que consiste em uma linguagem computacional baseada em lógica, com o objetivo de expressar ontologias. Explicaremos a seguir o funcionamento básico dessa linguagem, com base na cartilha oficial da OWL 2 [12], criada pelo W3C (*World Wide Web Consortium*).

Uma ontologia pode ser descrita como um conjunto de afirmações precisas para a descrição de uma certa parte do mundo (domínio de interesse). Ontologias podem referenciar umas às outras, permitindo o compartilhamento e a extensão de áreas diversas do conhecimento.

As afirmações básicas expressadas por uma ontologia são denominadas axiomas. Assim, ontologias em OWL são essencialmente uma série de axiomas relacionando diversas entidades, de forma a possibilitar a dedução de consequências e relações mais complexas. Entidades podem ser tanto instâncias (uma pessoa em específico) como uma classe ("pessoa", "homem", "mulher" etc). Além disso, pode-se estabelecer relações de hierarquia entre as classes (por exemplo, a classe "homem" consiste em uma subclasse de "pessoa"), igualdade ("pessoa" e "humano" são essencialmente a mesma classe) e disjunção (uma pessoa que pertença à classe "adulto" não pode pertencer à classe "criança").

Além das relações de hierarquia entre as classes, podem ser estabelecidas propriedades para os objetos (instâncias). As propriedades representam relações entre objetos. Por exemplo, podemos usar a propriedade "temFilho" para relacionar os objetos "João"

e "Antônio". Ou seja, "João temFilho Antônio" estabelece o parentesco entre ambos. Por meio de funções mais avançadas de propriedades, podemos definir também relações entre elas ("temPai" é a propriedade inversa de "temFilho") e outras características ("temCônjuge" é uma propriedade reflexiva).

Por fim, outro tipo de propriedade importante é a que relaciona um objeto a um valor de determinado tipo (*datatype*). Um uso possível para uma propriedade de dado é com o atributo "temIdade". Para caracterizar melhor esta informação, pode-se utilizar domínios e alcances, de forma a garantir que o dado associado a "temIdade" seja sempre um inteiro (domínios e alcances também podem ser determinados para propriedades entre objetos). Logo, pode-se criar a relação "João temIdade 23", sendo "23" um número inteiro que representa uma quantidade de anos.

Há quatro sintaxes para a escrita de ontologias em OWL (estilo funcional, RDF/XML, *Turtle*, *Manchester* e OWL/XML). Como faremos uso de RDF, nossas ontologias serão escritas na sintaxe RDF/XML. A seguir temos um trecho de ontologia (descrevendo o exemplo anterior de idade para João) escrito com esta sintaxe.

```
<owl:DatatypeProperty rdf:about="temIdade">
    <rdfs:domain rdf:resource="Pessoa"/>
    <rdfs:range rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger"/>
</owl:DatatypeProperty>

<Pessoa rdf:about="Joao">
    <temIdade rdf:datatype=
        "http://www.w3.org/2001/XMLSchema#integer">23</temIdade>
</Person>
```

3.3 AIML

O *Artificial Intelligence Markup Language* (AIML) - ou Linguagem de Marcação para Inteligência Artificial, em português - é uma linguagem de marcação derivada do XML, desenvolvida para ser fácil de implementar, de ser usada por iniciantes e para interagir com XML e seus derivados [13]. Seu objetivo principal é funcionar como um meio para que conhecimento seja transferido para e de sistemas com inteligência artificial. No nosso caso, ele será usado para converter o texto obtido da fala em informação para o GD, bem como realizar o caminho inverso (transformar as informações básicas em texto a ser lido pelo TTS).

Por ser derivado do XML, o AIML é formado por diversas marcações, chamadas de *tags*. Cada um desses *tags* define o começo e o fim de determinada parte do código (similar à marcação usada em HTML). A seguir serão apresentados os conceitos que servem como base para o AIML, conforme pode ser visto em [13], [14].

Um documento escrito em AIML consiste em categorias, marcadas por `<category>`. Elas definem respostas que devem ser dadas para cada possível entrada que o sistema

receba. As entradas são denominadas *patterns*, e as respostas são chamadas de *templates*. Assim, se for necessário criar uma categoria que responda à pergunta "Olá, como vai você?" com "Tudo bem, e você?", sua escrita será como a seguir.

```
<category>
    <pattern> OLA COMO VAI VOCE </pattern>
    <template> Tudo bem, e você? </template>
</category>
```

Com esse exemplo, vê-se que não há pontuação nem acentuação na entrada, a qual encontra-se toda em letras maiúsculas. O processo que faz o texto ficar neste formato é chamado de normalização, e é útil para evitar ambiguidades, além de ser necessário devido à falta de pontuação do texto obtido pelo ASR. Uma observação importante é que, como toda linguagem de marcação, o AIML depende de um programa que seja capaz de decodificá-lo (um interpretador), o qual irá passar a operar de acordo com as categorias presentes.

De maneira geral, há três tipos de categorias definidas em AIML: *atomics* (atômicos), *defaults* (padrões) e *recursives* (recursivos). A primeira delas, *atomic*, representa a categoria mais simples existente, pois consiste apenas de definições sem o uso de caracteres coringa (que substituem trechos desconhecidos), como o "*" e o "_". Já o segundo tipo de categoria consiste na ampliação da primeira com o uso dos tais coringas, daí a origem da denominação *default*. Se, por exemplo, uma categoria contiver `<pattern>OLÁ *</pattern>`, significa que ela aceitará qualquer entrada que começar com "olá", como "olá, tudo bem?" ou "olá, como vai você?". Por fim, categorias recursivas são aquelas que mapeiam uma entrada a outra. Ou seja, geram uma outra entrada que será então processada. Isso permite que situações similares sejam mapeadas da mesma forma, simplificando o sistema. Para determinar este comportamento, utiliza-se a marcação `<srai>`, como no exemplo a seguir, em que uma entrada "adeus" deve ser processada da mesma forma que uma entrada "tchau".

```
<category>
    <pattern> ADEUS </pattern>
    <template><srai> TCHAU </srai></template>
</category>
```

Tendo em vista estas estruturas básicas, há ainda a possibilidade de haver conflitos entre diferentes categorias. Por exemplo, se houver um *pattern* `OI` em alguma categoria, e `OI *` em outra, o programa deve ser capaz de escolher qual delas será usada. Para tanto, tem-se uma ordem de prioridade: primeiramente são buscados atômicos com a marcação `<that>`, depois os sem essa marcação. A função do `<that>` é a de indicar qual *template* foi usado anteriormente, de forma a referenciar o contexto da mensagem.

Depois dos *atomics*, a busca é feita nos *defaults* de maneira análoga: primeiro *defaults* com marcação *that*, depois os sem a marcação. Se nem nesta última busca for encontrado um *pattern* correspondente, não há resultado disponível. Para possibilitar melhor definição do contexto de terminado trecho da conversa, as categorias podem também ter tópicos (*topics*). Logo, as buscas são realizadas primeiramente em categorias com o

tópico em questão (tanto em *atomics* quanto *defaults*), e somente depois em categorias mais gerais, sem tópico.

3.4 JADE

A fim de permitir a comunicação entre os vários módulos, bem como processamento paralelo, será utilizado o *JAVA Agent DEvelopment Framework* (JADE) [2]. Trata-se de um ambiente para a criação de sistemas distribuídos em agentes, totalmente implementado em Java. A partir desta estrutura, será possível abstrair detalhes de comunicação e gerenciamento de informações trocadas entre os módulos. A seguinte explicação dos componentes do JADE tem como base os tutoriais disponíveis em [15], [16]. A aplicação dos agentes descritos nesta seção pode ser melhor vista na seção 6.

Toda a lógica de sistemas feitos em JADE é baseada em entidades denominadas **Agentes**. Cada um deles tem um nome único e é responsável por executar determinada tarefa. Eles podem interagir entre si por meio de troca de mensagens. No caso do nosso projeto, um agente pode, por exemplo, ter a tarefa de obter do usuário a data de uma reunião, e enviar uma mensagem contendo esta informação para outro agente.

Cada instância do ambiente do JADE recebe o nome de **Contêiner**, unidade que contém zero ou mais agentes. O conjunto de todos os contêineres ativos corresponde a uma **Plataforma**. Em cada plataforma está sempre presente um **Contêiner Principal**, responsável por gerenciar os demais. Por esse motivo, todos os outros contêineres devem se registrar com o principal quando são criados. Se um novo contêiner principal for iniciado em algum lugar da rede é criada, portanto, uma nova plataforma.

[2] `http://jade.tilab.com/`

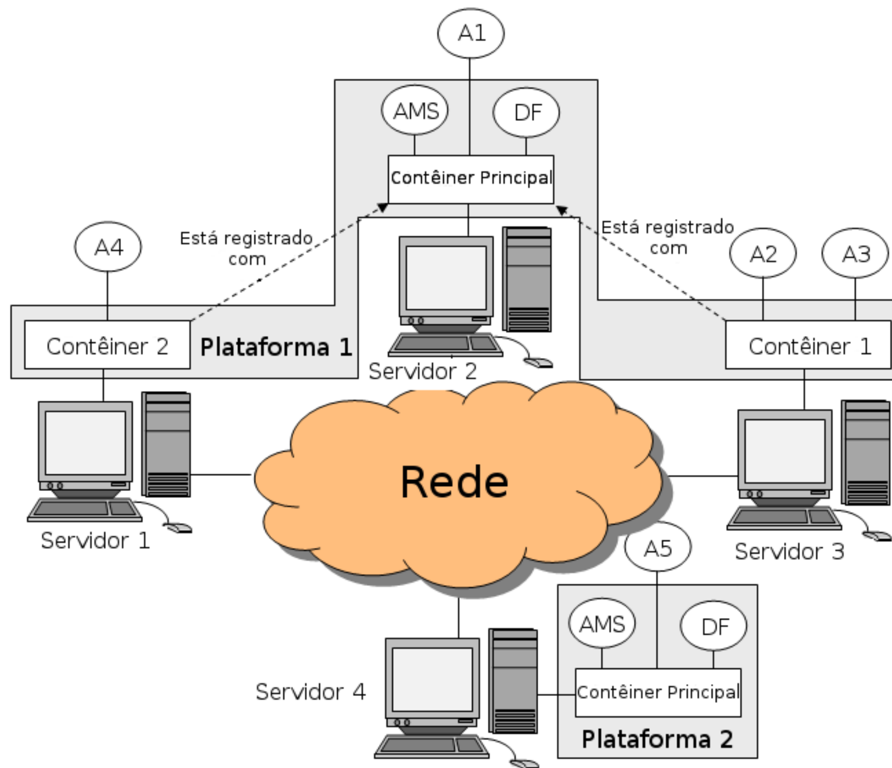


Figura 3: Estrutura de um sistema construído na plataforma JADE. Imagem adaptada de [15].

A estrutura descrita anteriormente é exemplificada na Figura 3. Pode-se ver nela duas plataformas, a primeira composta por três contêineres em três diferentes servidores e a segunda com apenas um contêiner. Nota-se, portanto, que uma plataforma não está limitada a apenas um servidor. Contudo, apesar de não ser mostrado na imagem, um único servidor pode ter mais de um contêiner, ou até mesmo mais de uma plataforma. A criação de novos contêineres e plataformas depende, então, das necessidades de cada sistema.

Diferentemente de contêineres normais, um contêiner principal possui dois agentes logo que é criado: o AMS (*Agent Management System*, ou Sistema de Gerenciamento de Agentes) e o DF (*Directory Facilitator*, ou Moderador de Diretórios). Como o próprio nome indica, o primeiro tem a tarefa de gerenciar os demais agentes da plataforma. Logo, pode criar e destruir agentes, além de controlar a unicidade de nomes entre eles. O DF, por sua vez, fornece um serviço de “páginas amarelas”, ou seja, permite que agentes encontrem outros com base nas tarefas que cada um é capaz de executar.

3.5 ASR

O termo *Automatic Speech Recognition* (ASR) é usado para denominar *softwares* de reconhecimento de voz. Este componente recebe a entrada de áudio do usuário e a transforma em texto. Seu funcionamento detalhado foge ao escopo deste trabalho, restando apenas entender as configurações necessárias para que ele funcione.

A maioria dos ASR's estudados necessitam que o desenvolvedor carregue um arquivo de gramática. Esta gramática corresponde a todas as possíveis entradas de áudio que o sistema deve reconhecer (o que não se restringe apenas a palavras, mas também a frases e expressões). Nota-se que o sistema não irá gerar uma saída de texto caso a entrada não esteja definida na gramática. Existem diferentes linguagens para gramáticas, o que é o caso do VoiceXML e do SRGS (*Speech Recognition Grammar Specification*). Como não há um padrão, cabe à empresa que desenvolveu o ASR definir o tipo de arquivo aceito.

Outra característica configurável em pelo menos um dos programas de reconhecimento de voz testados é o método de aquisição da voz, o qual pode ser absoluto ou incremental. No modo absoluto o sistema espera o usuário parar de falar para começar a processar o áudio. No modo incremental, o processamento das informações é feito conforme a fala ainda está ocorrendo.

São empresas de destaque no ramo de reconhecimento de voz: Microsoft, Google, IBM, Apple, Nuance e Verbio.

3.6 TTS

O *Text-To-Speech* (TTS) é um sistema que transforma uma entrada de texto em áudio. A sintetização da voz ocorre concatenando pedaços gravados de voz guardados em um banco de dados. Um dos desafios do TTS está na entonação da voz gerada, além de outros fatores que contribuem para que a fala assemelhe-se à humana, como a fluidez. Atualmente há diversas soluções de TTS disponíveis no mercado, com resultados que variam muito em qualidade. Na maioria dos casos, programas com módulos de ASR possuem também um módulo de TTS. Isso facilita o emprego destas tecnologias neste projeto, pois só é necessária a integração com uma plataforma, ao invés de duas.

4 Testes com ASR e TTS

Com o intuito de escolher um software de ASR e TTS para ser utilizado no trabalho, três API's foram testadas.

4.1 Coruja

Desenvolvido pelo grupo FalaBrasil do LaPS (Laboratório de Processamento de Sinais) da Universidade Federal do Pará, o Coruja [3] é um *software* para reconhecimento de voz totalmente concebido para o português brasileiro. Pode ser utilizado tanto com o modelo acústico LaPSAM (desenvolvido pelo mesmo grupo) como com outros que tenham sido criados com o uso do *toolkit* HTK [4].

Mesmo com seu foco na língua portuguesa, os testes realizados mostraram que ele possui duas grandes desvantagens. Primeiramente, o reconhecimento das palavras não

[3] `http://www.laps.ufpa.br/falabrasil/`

[4] `http://htk.eng.cam.ac.uk/`

é preciso, mesmo em testes simples com poucas palavras. Um programa que deveria reconhecer quatro comandos de direção (frente, trás, direita e esquerda) não era capaz de realizar tal tarefa satisfatoriamente, e este erro só tende a aumentar conforme mais palavras são necessárias. O segundo problema notado foi que o programa aparentava comportamento instável, por vezes parando de funcionar logo ao iniciar, mesmo sendo utilizado o código de teste que acompanha o Coruja.

Apesar dos problemas constatados, o Coruja encontra-se em contínuo desenvolvimento. Logo, seus erros e *bugs* podem ser minimizados futuramente. Pretende-se implementar também recursos adicionais, como o suporte a ditado (o que evitaria o uso de gramáticas) e a compatibilidade com a *Speech API* da Microsoft.

4.2 Web Speech API

A primeira API testada foi a *Web Speech API* fornecida pelo Google. Previamente, para acessá-la, o desenvolvedor precisava apenas criar código em qualquer linguagem que possuísse uma biblioteca capaz de enviar um comando de *request* pelo protocolo de transferência de hipertexto (*http*). Hoje, o acesso ao TTS continua funcionando de maneira similar. O ASR, por sua vez, foi incluído no Google Chrome a partir da versão 25 do navegador, de acordo com novo protocolo especificado pelo W3C [17], e o acesso por *http* foi descontinuado. O TTS também pode ser usado por meio do novo protocolo, mas possui maior suporte dentre os demais navegadores do que o módulo de ASR.

A necessidade da utilização do navegador representa uma vantagem do ponto de vista gramatical, uma vez que todas a palavras e termos que podem ser reconhecidos estão definidos no servidor da empresa responsável por ele. Ou seja, não é necessário o uso de uma gramática. Entretanto, isso também representa um problema, uma vez que uma conexão estável e com boa velocidade torna-se necessária para que a conversa possa fluir naturalmente. Outro problema é ser requerido o uso de um navegador que suporte a API (o que, atualmente, é feito apenas pelo Google Chrome), o qual pode não estar configurado corretamente ou nem mesmo instalado na máquina do usuário.

4.3 Microsoft Speech Platform

Outra API analisada é a Microsoft Speech Platform [18], que funciona no Windows Vista ou superior e necessita que a biblioteca do Microsoft .NET Framework 4.0 esteja instalada. Esta API, diferentemente da anterior, não necessita de acesso à internet. A plataforma da Microsoft requer uma gramática definida, com possíveis padrões de entrada, para que o ASR reconheça o áudio. Ela pode ser definida no padrão VoiceXML [19] ou SRGS [20].

Foram testados diferentes arquivos de gramática. Pode-se observar que o sistema funciona muito bem para gramáticas do tipo A, ou seja, bem estruturadas gramaticalmente e com uma variedade reduzida de palavras possíveis. Os testes com gramática do tipo B (sem estruturação) mostraram que a precisão do sistema diminui conforme aumentamos a lista de palavras possíveis. Ambos os testes foram realizados repetindo uma frase reconhecível pelo sistema 5 vezes e analisando o texto gerado pelo ASR. O gráfico da figura 4 mostra os resultados do teste.

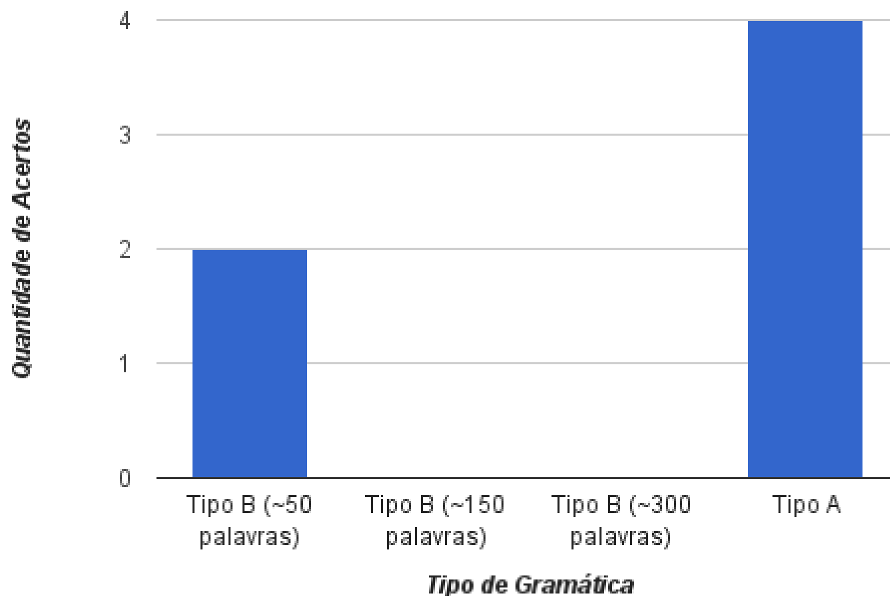


Figura 4: Gráfico de taxa de acerto dos testes do sistema para um total de 5 testes realizados com cada tipo de gramática. Autoria própria.

Tipo de Gramática	Saída do Sistema
Tipo A	marque uma festa
Tipo B (aprox. 50 palavras)	marque uma festa capitulo
Tipo B (aprox. 150 palavras)	marque nos as passaros do
Tipo B (aprox. 300 palavras)	Martinho festa casa Lucas

Tabela 1: Demonstrativo de saída errada do sistema para os diferentes testes.

A tabela 1 mostra uma das saídas errada do sistema quando ele foi incapaz de reconhecer a frase teste ("Marque uma festa na casa do Lucas") para cada uma das gramáticas. O TTS da API funciona sem problemas observados, apesar de não ser capaz de pronunciar palavras com entonações específicas, como ao fazer perguntas. Esta dificuldade é comum entre as tecnologias de geração de fala.

4.4 Sistema de Fala Escolhido

Diante dos resultados obtidos com as três soluções, escolheu-se para o reconhecimento e a síntese da fala a *Web Speech API*. Embora ela requira acesso constante e estável à *internet*, isso não se mostrou ser um empecilho durante o desenvolvimento do sistema. Em sua maioria, as redes móveis foram as mais problemáticas nesse sentido. Entretanto, como pizzarias podem ter acesso a conexões mais velozes do que as de aparelhos móveis, essa preocupação é minimizada.

A grande vantagem da *Web Speech API* em relação às demais foi a possibilidade de não se usar uma gramática. Como pôde-se ver com os testes da Microsoft Speech Platform,

ela precisa de uma gramática bem estruturada e com milhares de palavras para funcionar corretamente. Assim, provou-se muito comum que palavras parecidas fossem trocadas, o que prejudicaria o funcionamento do sistema, em alguns casos até mesmo inviabilizando-o. Além disso, seria preciso alocar tempo para a construção da gramática, impedindo que algumas funcionalidades fundamentais fossem implementadas.

Tendo escolhida a API, foi programada a interface de ASR/TTS. Para tanto, foi usado como base o Node.js[5], um sistema de *runtime* para o desenvolvimento de programas em Javascript. Ele foi escolhido pois o *Web Speech API* está definido em Javascript, e também porque é capaz de criar um servidor local para gerenciar o recebimento e o envio de mensagens a partir de uma janela de navegador. Como (até o momento) apenas o Google Chrome possui suporte para as funcionalidades necessárias para este projeto, a página de acesso ao sistema deve ser aberta nele.

5 Análise de Requisitos

Pode-se elencar três requisitos subjetivos, os quais indicam se a conversa flui de maneira "natural" (parecida com uma conversa com outro ser humano). Eles estão descritos a seguir, e foram usados nos testes com voluntários (Seção 7.4).

- **Tempo de resposta do sistema:** embora este fator seja mensurável, seu efeito varia de acordo com cada indivíduo. O foco para este parâmetro está justamente na reação dos usuários com a demora ou não do sistema em responder.
- **Grau de entendimento:** este requisito está relacionado à capacidade do sistema entender o que é dito pelo usuário e, caso ocorra um erro de entendimento, repará-lo e manter a conversa fluindo.
- **Percepção do usuário:** representa a impressão que o usuário teve ao utilizar o sistema. Para que possa ser considerada uma interação com sucesso, não pode haver estranheza no modo como a conversa é conduzida, e o usuário deve ter uma impressão próxima à de um diálogo natural.

Dependendo da aplicação específica adotada para o sistema desenvolvido, outros requisitos devem ser analisados. Eles serão tratados na Seção 7.1.

6 Solução Proposta

Nesta seção encontra-se uma descrição da solução proposta para o desenvolvimento do sistema almejado. A arquitetura do projeto (melhor descrita na próxima seção) segue aquela apresentada na Figura 1 da Seção 2. O diagrama de componentes proposto encontra-se no Apêndice A.

Para o Semantizador e o Gerador de Linguagem será usado como base o AIML. Ambos os módulos possuem essencialmente a mesma funcionalidade, porém invertida: o primeiro recebe frases quase genéricas e fornece como saída qual é o ato dialogal que a representa,

[5] `https://nodejs.org/`

enquanto o segundo transforma uma requisição de informação em uma frase completa. Com as propriedades do AIML, é possível executar ambas.

Uma funcionalidade importante para o sistema é ser capaz de entender conceitos complexos, como o que deve fazer parte de um pedido e quais atributos uma pizza pode ter. Para isso, serão usadas ontologias escritas em OWL.

A seguir tem-se uma explicação da arquitetura geral do sistema proposto e das normas e especificações adotadas na comunicação dos módulos. Depois, explica-se mais detalhadamente o funcionamento dos três principais módulos para o gerenciamento do diálogo, o Semantizador, o Gerador de Linguagem e o Gerenciador de Diálogo. Por fim, descreve-se o desenvolvimento da Memória Semântica e as ontologias utilizadas.

6.1 Arquitetura

Com o intuito de permitir a utilização de soluções de ASR escritas em qualquer linguagem de programação, um *buffer* intermediário entre ASR e Semantizador se faz necessário. O *buffer* é um controlador de filas, implementado por meio do programa RabbitMQ [6]. O módulo de reconhecimento adiciona as frases de saída ao fim da fila e cabe ao agente de comunicação com o ASR requisitá-las, enviando cada uma ao Semantizador. Desta forma, a linguagem em que o ASR foi escrito não se torna restritiva ao projeto. O mesmo é feito para a comunicação entre o GL e o TTS, permitindo também que este último seja programado em qualquer linguagem, inclusive diferente daquela usada no ASR.

Toda a comunicação entre Semantizador, GD, GL e memória é feita por meio da interface de agente do JADE. O JADE implementa em seu *framework* uma fila de mensagens para cada agente e todos os métodos necessários para acessá-la. Ao longo do restante deste trabalho, toda a menção a agentes refere-se aos módulos principais do sistema, os quais estendem a interface de agente do JADE e estão indicados no diagrama de componentes.

O interpretador de AIML é um componente importante para o funcionamento do Semantizador e do GL e será tratado nos capítulos destes respectivos módulos. O mesmo serve para o corretor de mensagens e a unidade de memória procedural, que serão melhor explicados no capítulo que referente ao GD.

6.2 Normas e Especificações

6.2.1 Comunicação entre agentes

A comunicação de agentes do jade utiliza a especificação da FIPA de mensagens ACL [21]. As mensagens ACL possuem os parâmetros apresentados na tabela 2. De todos os parâmetros, oficialmente, apenas o campo de *performative* é obrigatório.

[6] `https://www.rabbitmq.com/`

Parameter	Category of Parameters
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of content
encoding	Description of content
ontology	Description of content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

Tabela 2: Tabela de parâmetros da mensagem ACL. Retirado de [21].

Neste projeto, nem todos os campos disponíveis se mostraram úteis, portanto apenas os campos de remetente, ID da conversa, ontologia, conteúdo e performativa foram utilizados. De maneira geral, os campos de remetente e ID da conversa servem para filtrar as mensagens que chegam, o parâmetro de ontologia indica a ação que deve ser realizada pelo destinatário, e a performativa indica o tipo da mensagem em atos comunicativos, conforme descrito pela FIPA [22].

Maiores detalhes sobre a comunicação entre agentes e o valor que os parâmetros da mensagem ACL devem conter em cada caso são encontrados no Apêndice B.

6.2.2 Atos dialogais

Conforme visto na Seção 2.3, foram utilizados atos dialogais tal qual definido na norma ISO 24671-2. Todavia, embora ela seja usada como embasamento teórico, a representação adotada não será a mesma. Essa decisão tem dois motivos. Primeiramente, trata-se de uma notação em XML, o que complicaria o seu uso nos arquivos AIML (necessários para o funcionamento do Semantizador e do GL), pois as marcações de ambos se confundiriam. Além disso, será usada uma notação mais simples, suficiente para o problema abordado, facilitando o desenvolvimento não apenas dos arquivos AIML como dos módulos do sistema.

Dada a base teórica descrita, adotou-se uma notação na qual cada ato dialogal possui três atributos (dimensão, função e conteúdo) e é separado dos demais por estar contido dentro de chaves. Portanto, é marcado da seguinte maneira: `{dimension: *; function: *; content: *}`. Os asteriscos representam os valores para os atributos, que variam de acordo com o ato dialogal.

Mais informações e exemplos podem ser vistos na documentação dos atos dialogais, a qual encontra-se no Apêndice C.

6.3 Semantizador

O semantizador desenvolvido tem como base arquivos de AIML, responsáveis pela conversão das frases genéricas em atos dialogais. Como interpretador de AIML, foi utilizada a biblioteca do Program AB [7]. Com ela pode-se criar "seções de *chat*", para as quais é fornecido o texto obtido do ASR. Desta forma, pode-se focar na criação dos arquivos de AIML, fundamentais para que os atos dialogais obtidos estejam corretos.

O trecho seguinte mostra como uma frase na forma "pizza *" é processada de forma a se obter um ato dialogal de dimensão procedural, função de pedido e conteúdo especificando que se trata de um pedido de pizza. As informações adicionais, representadas pelo asterisco, são colocadas ao final do conteúdo, para serem analisadas pelo GD. Por exemplo, no caso da frase de entrada ser "pizza de calabresa", o resultado é `{dimension: procedural; function: order; content: [pizza], de calabresa}`.

```
<category>
    <pattern>PIZZA *</pattern>
    <template>
        {dimension: procedural; function: order; content: [pizza
          ], <star />}
    </template>
</category>
```

Outra categoria, com *pattern* "PIZZA DE *", tem *template* `<srai>PIZZA <star /><srai>`, de forma a evitar a presença da preposição "de" no conteúdo. Ou seja, a frase anterior seria processada duas vezes, gerando resultado `{dimension: procedural; function: order; content: [pizza], calabresa}`.

Durante o desenvolvimento do sistema, percebeu-se uma limitação problemática na estrutura da linguagem AIML: cada categoria, embora possa ter muitos *templates*, pode ter apenas um *pattern*. Assim, para que frases equivalentes possam representar o mesmo ato dialogal, é necessário criar uma categoria para cada uma delas, de forma a que elas redirecionem para a mesma construção. Por exemplo, deseja-se que ambas as frases "Eu gostaria de uma pizza *" quanto "Quero uma pizza *" tenham *template* `<srai>PIZZA <star /></srai>`, o que faria o resultado de ambas ser o mesmo do trecho usado como exemplo anteriormente. Usando AIML, precisa-se criar duas categorias para cumprir este objetivo. Entretanto, conforme mais possibilidades são adicionadas ("Queria uma pizza *", "Eu quero uma pizza *" etc) torna-se complicada a manutenção do arquivo (saber quais possibilidades não foram analisadas e evitar repetições).

Para resolver este problema, foi criado um tipo de arquivo adicional, apelidado de "AIML skelleton" (ou "esqueleto de AIML"), com extensão do tipo **.aiml.skel*. Sua estrutura é análoga à da linguagem original, porém com algumas importantes diferenças:

- Cada abertura ou fechamento de *tag* deve estar contido em uma linha separada;
- A marcação *pattern* foi substituída por *patterns*, e pode conter em cada linha um padrão diferente.

[7] `https://code.google.com/p/program-ab/`

Com isto, tem-se arquivos mais simples e de fácil edição. Desenvolveu-se também um interpretador para converter esses novos arquivos em AIML padrão, de forma a possibilitar a leitura pelo Program AB. Para ilustrar a vantagem em usar esse novo método, pode-se ver o exemplo a seguir.

```
<category>
    <patterns>
        EU GOSTARIA DE UMA PIZZA *
        EU QUERO UMA PIZZA *
        EU QUERIA UMA PIZZA *
    </patterns>
    <template>
        <srai>PIZZA <star /></srai>
    </template>
</category>
```

Embora haja no exemplo apenas três das várias possibilidades para este caso, ele já se mostra muito mais simples e intuitivo do que o resultado dele após compilação, conforme mostrado abaixo.

```
<category>
    <pattern>EU GOSTARIA DE UMA PIZZA *</pattern>
    <template>
        <srai>PIZZA <star /></srai>
    </template>
</category>

<category>
    <pattern>EU QUERO UMA PIZZA *</pattern>
    <template>
        <srai>PIZZA <star /></srai>
    </template>
</category>

<category>
    <pattern>EU QUERIA UMA PIZZA *</pattern>
    <template>
        <srai>PIZZA <star /></srai>
    </template>
</category>
```

A conversão dos "esqueletos" em arquivos de AIML ocorre quando o sistema é iniciado, para que possíveis alterações possam ter efeito. Cria-se então a sessão de *chat* do Program AB. A partir disso, o agente do semantizador aguarda o recebimento de mensagens, processando-as por meio da sessão de *chat* e enviando os resultados para o GD.

6.4 Gerador de Linguagem

Como mencionado em seções anteriores, o funcionamento do Gerador de Linguagem (GL) é análogo ao do semantizador. Contudo, nesta etapa ocorre o inverso: os atos dialogais são convertidos em linguagem natural. Devido ao fato da mensagem de entrada do GL ser uma saída do GD, sua formatação é padronizada, o que torna mais simples a criação dos arquivos AIML. Portanto, não é necessário fazer arquivos de “esqueleto”, como é feito no Semantizador. A seguir tem-se um exemplo de categoria para este módulo.

```
<category>
    <pattern>{dimension: procedural; function: inform; content:
        [pizza], [topping]=*, [price]=*}</pattern>
    <template>
        <random>
            <li>A pizza de <star index="1" /> custa <star index=
                "2" />. </li>
            <li>Ela custa <star index="2" />. </li>
        </random>
    </template>
</category>
```

Neste caso, o ato dialogal prevê que seja informado para o ouvinte o preço de um determinado sabor de pizza. Na marcação `<star>` é usado o parâmetro “index”, o qual serve para indicar qual dos “*” é o referenciado por ela. Também são utilizadas outras duas *tags* importantes: `<random>` e `<li>`. Elas permitem que um mesmo *pattern* possa ter como resposta mais de uma possibilidade, escolhida aleatoriamente. Cada trecho entre marcações `<li>` representa uma dessas possibilidades, e todas têm igual probabilidade.

Assim, caso o GD envie como ato dialogal `{dimension: procedural; function: inform; content: [pizza], [topping]=calabresa, [price]=23 reais}`, há duas respostas possíveis: “A pizza de calabresa custa 23 reais.” ou “Ela custa 23 reais.”. O resultado obtido pelo GL é enviado para o agente de TTS.

6.5 Gerenciador de Diálogo

O modelo de Gerenciador de Diálogo proposto neste trabalho é uma adaptação do descrito no trabalho de Bohus & Rudnicky [4]. As figuras 5 e 6 ilustram o modelo proposto.

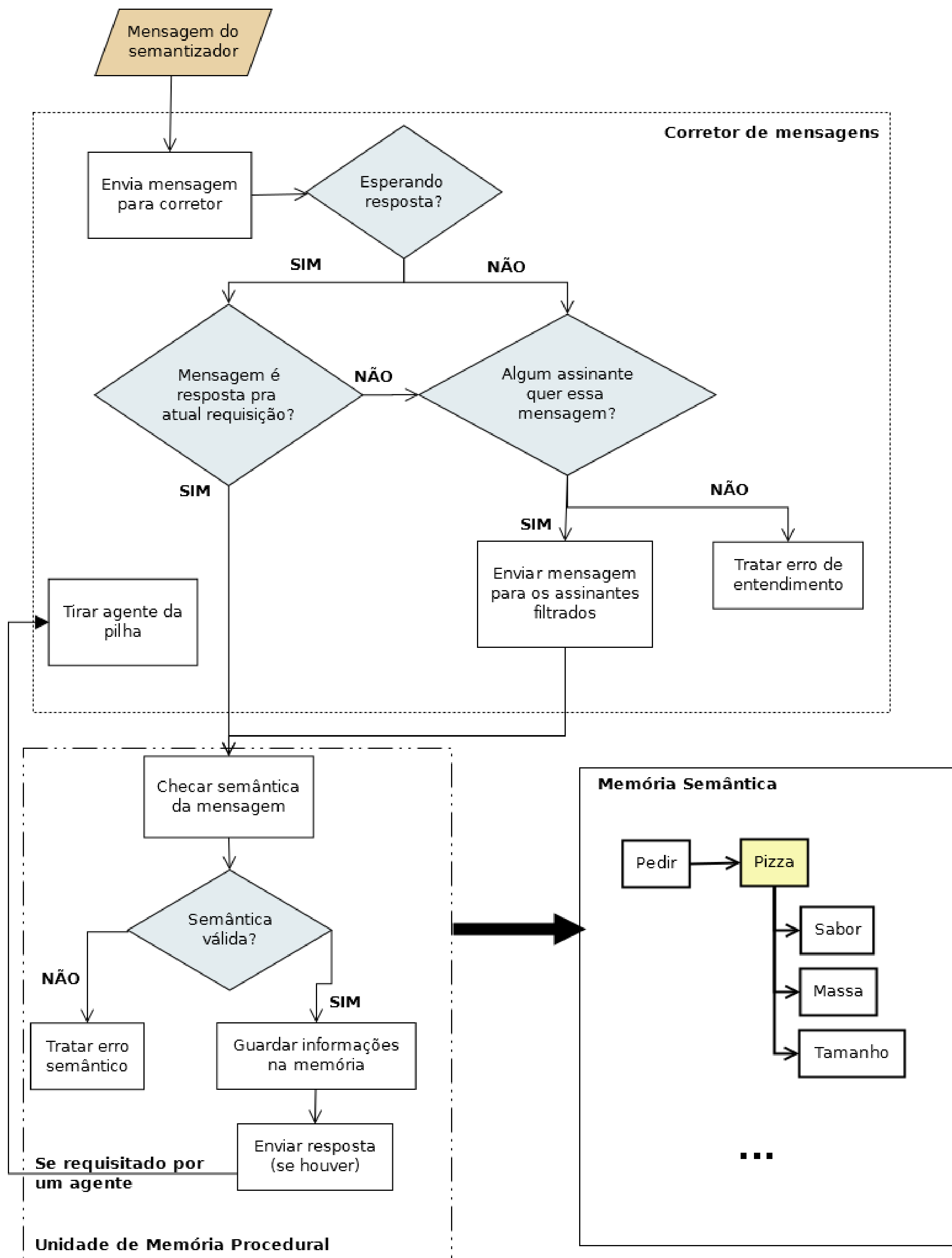


Figura 5: Modelo de Gerenciador de Diálogo proposto. Autoria própria.

O GD se utiliza de duas estruturas importantes: uma pilha de requisições e uma agenda de expectativas.

A pilha de requisições localiza-se na memória episódica e guarda as ações do diálogo.

Cada requisição possui uma pergunta a ser realizada e sua respectiva agenda de expectativas. Elas não são carregadas todas ao mesmo tempo na pilha. O controlador do fluxo do diálogo é encarregado de evitar que a pilha esvazie, adicionando agentes a ela uma vez que o diálogo se iniciou.

A agenda de expectativas guarda as possíveis respostas esperadas pelo sistema após determinada pergunta feita por ele ao usuário. Por exemplo, se a próxima pergunta for "A pizza é de tamanho grande?", a agenda de expectativas irá conter "[SIM]" ou "[NÃO]" como possíveis respostas do usuário.

O GD possui três subsistemas importantes para seu funcionamento:

- Controlador de novas mensagens;
- Controlador da pilha de requisições;
- Controlador do fluxo de diálogo.

6.5.1 Controlador de Novas Mensagens

O controlador de novas mensagens foi projetado com uma topologia de Editor-Assinante. Toda nova mensagem que chega no GD é enviada ao corretor de mensagens. Este corretor possui alguns assinantes cadastrados, e sua função é filtrar quais mensagens irão para quais assinantes, baseando-se no tipo de mensagem de interesse de cada assinante. Na hora de se cadastrar no corretor, cada assinante indica que tipo de mensagens quer receber. Como as mensagens recebidas são atos dialogais, os assinantes indicam o tipo de dimensão e função para os quais desejam se cadastrar.

Antes de enviar as mensagens aos respectivos assinantes, o corretor checa se a mensagem é uma resposta a uma pergunta previamente feita pelo sistema. Neste caso, a mensagem não é enviada aos assinantes.

Cada assinante é uma unidade de memória procedural. Memória procedural é um tipo de memória não-declarativa que está relacionada a lembrar procedimentos e não requer participação consciente. As unidades acessam diretamente a memória semântica carregada pelo GD. Além disso, elas podem se carregar na pilha do controlador de fluxo do diálogo a fim de completar alguma informação semântica. Por exemplo, caso seja pedido uma pizza, uma unidade que cuide de pedidos pode se carregar no controlador para perguntar a respeito do sabor e massa da pizza, completando assim a semântica necessária de uma pizza.

Fazem parte do *framework* as seguintes unidades procedurais, que serão explicadas adiante:

- Unidade de obrigações sociais;
- Unidade padrão de erros.

6.5.2 Unidade de Obrigações Sociais

A unidade de obrigações sociais cuida do tratamento das convenções sociais, como cumprimentos e agradecimentos. Este assinante se cadastra para receber mensagens que pos-

suam ato dialogal {dimension: socialOblig; function: *; content: *}. Como o tratamento de obrigações sociais é algo que não chega a acessar a memória semântica, esta unidade apenas encaminha o ato dialogal correto para o GL. Pelo fato do sistema ser ativo, a obrigação social de cumprimento foi desativada. Ou seja, ao chegar um ato dialogal {dimension: socialOblig; function: greet; content: hi}, o sistema não encaminha uma mensagem ao GL, uma vez que o primeiro cumprimento da conversa será sempre do sistema.

6.5.3 Unidade Padrão de Erros

Esta unidade trata de atos dialogais do tipo {dimension: unknown; function: *; content: *}. Ela é o *fallback* do corretor de mensagens. Caso nenhuma unidade de memória procedural esteja cadastrada em determinada mensagem, ela será enviada para este assinante, o qual trata de erros de entendimento do sistema. Ele apenas encaminha um erro de entendimento ao GL, por meio do ato dialogal {dimension: alloFeedback; function: handle_error; content: *}.

6.5.4 Controlador da Pilha de Requisições

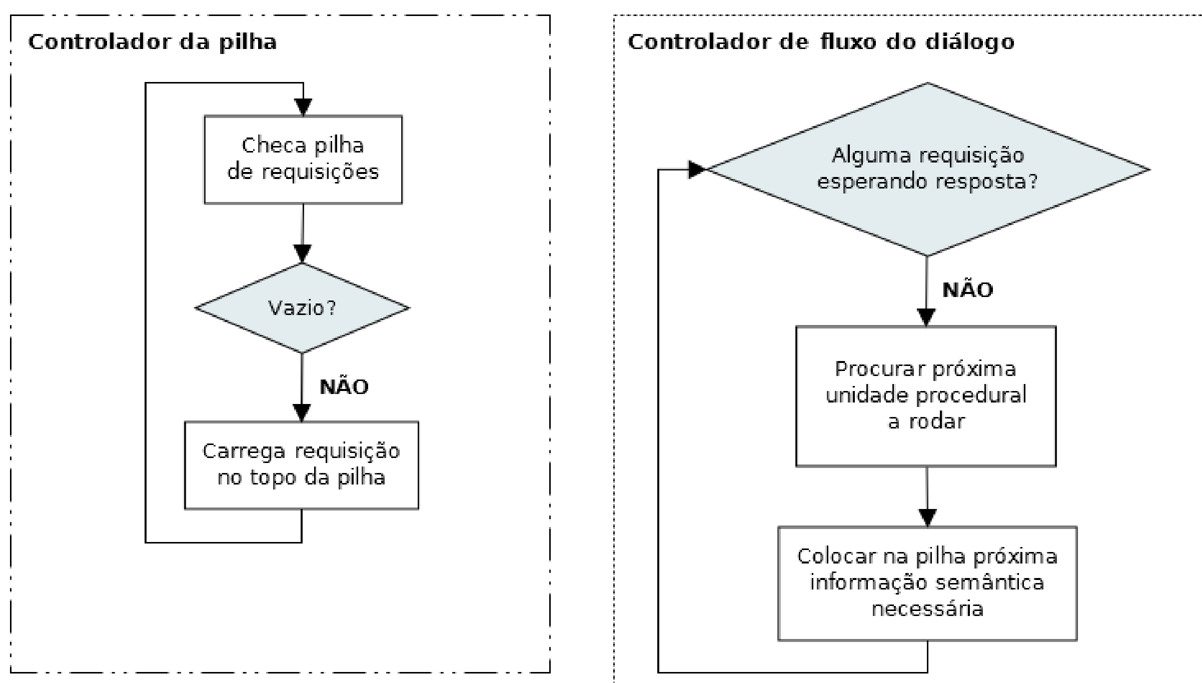


Figura 6: Modelo do controlador da pilha de agentes e do controlador do fluxo do diálogo. Autoria própria.

O controlador da pilha de requisições checa em intervalos regulares o topo da pilha que se localiza na memória episódica. Caso exista uma nova requisição no topo da pilha, esta é carregada e qualquer outra requisição previamente carregada que ainda não se completou é terminada. Apesar de ser uma estrutura do GD, o controlador é dividido em duas partes, sendo uma localizada na própria memória episódica e outra no GD. A metade

localizada na memória é responsável por checar o topo da pilha em intervalos regulares. Ao identificar uma nova requisição, uma mensagem é enviada ao GD com esta nova requisição. A metade localizada no gerenciador recebe esta mensagem e é responsável substituir a requisição que estava rodando anteriormente pela atual.

6.5.5 Controlador do Fluxo de Diálogo

O controlador do fluxo de diálogo é responsável por guiar o diálogo a fim de completar as informações semânticas necessárias. Ele é constituído de uma pilha de unidades de memória procedural e algumas classes auxiliares, denominadas genericamente como *"helpers"*. O controlador verifica, em intervalos regulares, se o sistema está esperando uma resposta a alguma requisição anterior. Em caso negativo, checa-se a pilha de unidades de memória procedural para ver se existe alguma informação semântica necessária faltando para o GD. Esta pilha inicia-se vazia. Conforme explicado na Seção 7.3.1, as próprias unidades de memória procedural se carregam na pilha do controlador de fluxo de diálogo quando necessitam completar alguma informação semântica.

Os *"helpers"* são classes ligadas a alguma unidade de memória procedural e servem de complemento para a mesma. Eles rodam em intervalos regulares de tempo e podem ser acessados pelo GD por meio do controlador de fluxo do diálogo.

O *framework* implementa o *"helper"* de obrigações sociais do sistema. Ele conta com diferentes estados, os quais podem ser alterados por outros *"helpers"*. Em seu estado inicial, este auxiliar envia para a pilha de requisições o cumprimento inicial do sistema e uma requisição inicial da conversa (por exemplo, "O que você deseja?"). Em seguida, entra-se em estado de espera até que algum outro auxiliar altere seu estado.

Para estender o campo de atuação do sistema, basta aumentar a memória semântica e implementar novas unidades de memória procedural e *"helpers"* associados, mantendo-se o *framework* original.

6.6 Memória Semântica

A Memória Semântica do sistema representa o seu conhecimento sobre conceitos abstratos. É ela que define, por exemplo, o que é uma pizza (suas características, possíveis valores e como ela se relaciona com o restante do mundo). Conforme visto na Seção 3.2, tais informações podem ser armazenadas em ontologias. Embora algumas ontologias possam ser compartilhadas entre sistemas com finalidades diferentes, o seu desenvolvimento está relacionado à aplicação específica na qual serão utilizadas. Portanto, as ontologias criadas para este trabalho estão descritas na Seção 7.2.

Para a leitura das informações armazenadas nas ontologias foi usada a OWL API [8]. Trata-se de uma API para Java que permite criar e manipular ontologias escritas em OWL.

[8] `http://owlapi.sourceforge.net/`

7 Exemplo de Aplicação

7.1 Análise de Requisitos da Aplicação

Como exemplo de aplicação, o sistema desenvolvido foi adaptado para o atendimento de pedidos de pizza. Para satisfazer tal situação, foram propostos os requisitos funcionais apresentados na Fig 7 em forma de casos de uso. Este diagrama representa as interações que o usuário será capaz de realizar com a máquina, tendo como ponto de vista o cliente do estabelecimento.

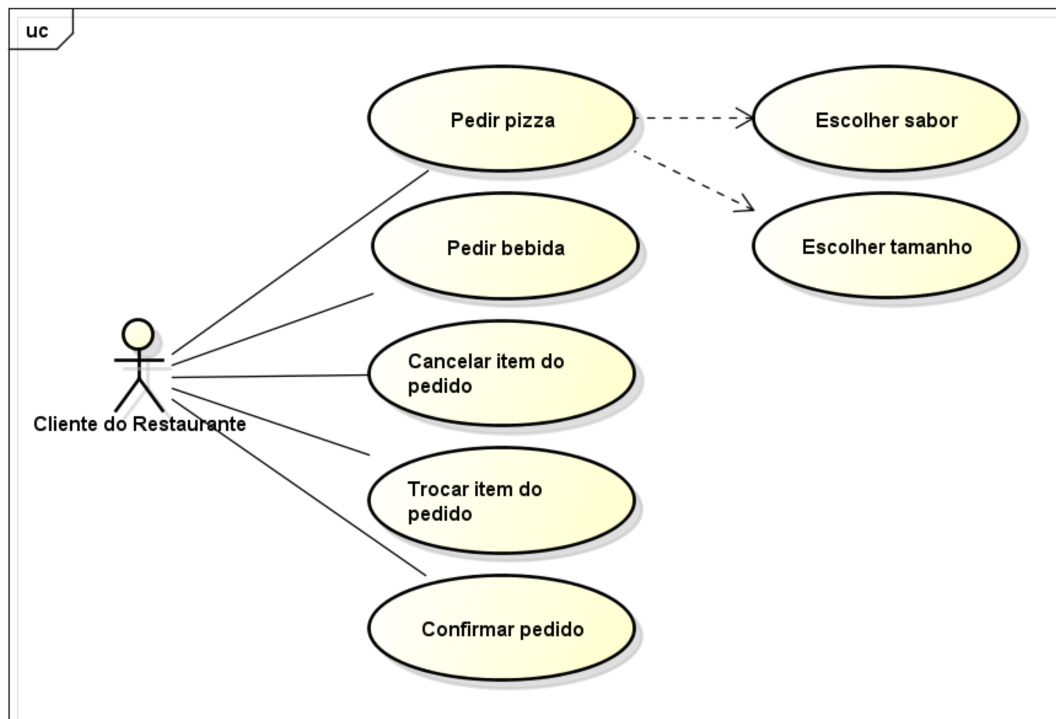


Figura 7: Diagrama de casos de uso relativo ao projeto. Autoria própria.

A seguir será explicado o funcionamento de cada caso ilustrado.

- **Pedir pizza:** esta é a função crítica do sistema como um todo, pois representa o ponto de maior desenvolvimento e com maior probabilidade de erros. Envolve a escolha de características específicas, como o sabor e o tamanho da pizza.
- **Pedir bebida:** similar à função anterior, porém mais simples, pois não será implementada a escolha de características (como o volume). A adição de bebidas aos itens que podem ser pedidos tem como finalidade comprovar se o sistema pode ser usado de maneira mais geral, e não apenas restrito a um tipo de item do pedido.
- **Cancelar item do pedido:** apesar de menos comum, é fundamental que o cliente seja capaz de cancelar seu pedido, caso tenha ocorrido algum erro ao pedir ou tenha mudado de ideia.
- **Trocar item do pedido:** esta função é similar a um cancelamento seguido de um novo pedido, mas possui uma dinâmica mais direta.

- **Confirmar pedido:** após escolhidos os itens que fazem parte do pedido, o usuário deve ser capaz de confirmá-los. Para isso, o seu pedido completo deve ser repetido pelo sistema, como garantia.

Durante o atendimento real em pizzarias, mais casos de uso são necessários, como "Escolher forma de pagamento" e "Informar endereço". Todavia, tais funcionalidades não foram abordadas neste projeto devido às restrições de tempo. Para que o sistema pudesse ser concluído no prazo, optou-se pelos casos já mostrados, considerados pelos autores como fundamentais. Pelo mesmo motivo, não foram incluídos casos de uso do ponto de vista do dono da pizzaria. Embora sejam necessários para a viabilidade comercial da solução, eles não se relacionam com o gerenciamento do diálogo.

7.2 Ontologias Desenvolvidas

Para este trabalho, foram criadas três ontologias, todas escritas em OWL com o auxílio do programa Protégé [9]. Trata-se de um editor de ontologias gratuito e de código aberto, que facilita a visualização e o entendimento dos elementos e suas relações.

As três ontologias desenvolvidas englobam diferentes conjuntos de conhecimentos necessários ao sistema:

- Alimentos (ou, como denominado no projeto, "consumíveis");
- Menu do restaurante;
- Atendimento de pedidos.

Escolheu-se criar três conjuntos em vez de apenas um pelas vantagens geradas com essa modularização:

- A visualização dos conceitos e suas relações é facilitada, pois limita-se a um grupo menor de elementos;
- A estrutura de uma das ontologias pode ser alterada sem que isso afete as demais;
- Pode-se trocar uma ou mais delas dependendo dos requisitos de um novo sistema, enquanto as outras são mantidas, com pouca ou nenhuma alteração. Ao mudar a pizzaria que utiliza o assistente virtual, por exemplo, é necessário que seja alterado apenas o "Menu do restaurante".

Serão explicadas agora as estruturas de cada uma das ontologias, bem como o relacionamento entre elas.

7.2.1 Ontologia de Alimentos ("Consumíveis")

Esta parte do conhecimento refere-se tanto aos alimentos que podem ser usados de recheios nas pizzas como a outros produtos alimentícios disponíveis para serem pedidos no estabelecimento. Para este projeto de conclusão de curso, foram estabelecidas duas

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

categorias principais: *Food* (Comida) e *Drink* (Bebida). A classe Comida possui mais algumas subclasses (como Queijo e Carne). Os alimentos de fato correspondem a instâncias das subclasses de Comida e da classe Bebida.

Embora pouco utilizada, a ontologia de alimentos gera possibilidades para desenvolvimentos futuros, como o detalhamento e personalização de cada pizza (poder pedir para retirar um ingrediente, por exemplo) e a separação delas em categorias (um cliente poderia pedir para serem listadas as pizzas sem queijo, ou com carne etc).

A Figura 8 mostra a estrutura descrita.

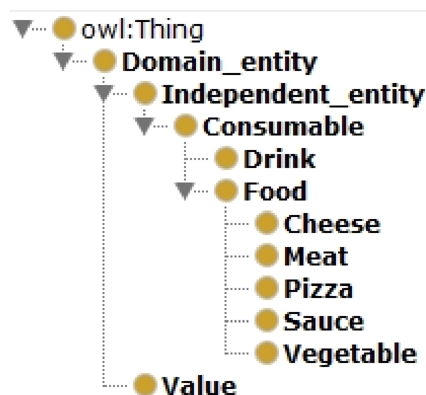


Figura 8: Estrutura da ontologia de alimentos. Autoria própria.

7.2.2 Ontologia do Menu do Restaurante

Trata-se de uma ontologia com alta variação dependendo do estabelecimento que adote o sistema. É nela que são definidos os sabores de pizza disponíveis e quais ingredientes os compõem. Também possui uma relação dos outros produtos vendidos, como por exemplo bebidas (definidas pelo seu nome - *Drink_name*). Tanto as pizzas como os demais produtos consistem em itens do menu (*Menu_item*), os quais são subclasses do menu em si. Por definição, valores específicos (como sabores de pizza e nomes de bebidas) são representados por instâncias de classes. Essa hierarquia pode ser vista na Figura 9.

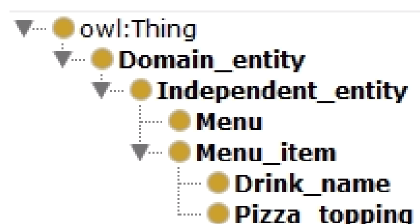


Figura 9: Estrutura da ontologia do menu do restaurante. Autoria própria.

7.2.3 Ontologia de Atendimento de Pedidos

Esta é a ontologia principal do sistema, e, portanto, é nela que as demais são importadas e referenciadas. A classe principal, Pedido (*Order*), é definida como um conjunto de itens

de pedido (*Order_Item*), que possui como subclasses Pizza e Bebida (*Drink*). Ambas são equivalentes às classes Pizza e Bebida estabelecidas na ontologia de alimentos. A classe Bebida possui apenas um único nome (*Drink_name*), enquanto Pizza tem um sabor (*Pizza_topping*), um tamanho (*Pizza_size*), um recheio da borda (*Pizza_border*) e uma espessura de massa (*Pizza_base*). Dentre os atributos de Pizza, apenas dois são obrigatórios: o sabor e o tamanho. Eles também foram os únicos a serem implementados no reconhecimento e na síntese da fala. Diferentemente dos sabores de pizza, os valores para os demais atributos são definidos por classes, e não instâncias.

A obrigatoriedade de certos atributos é definida por uma marcação (*flag*) com valor de "requerido" (*required*). Algumas propriedades adicionais (e suas respectivas propriedades inversas) foram implementadas para relacionar classes a outras classes e a instâncias: *has_base* / *is_base_of*, *has_border* / *is_border_of*, *has_size* / *is_size_of*, *has_topping* / *is_topping_of*, *has_name* / *is_name_of*, *has_item* / *is_item_of*, *has_ingredient* / *is_ingredient_of*.

A estrutura descrita é mostrada na Figura 10.

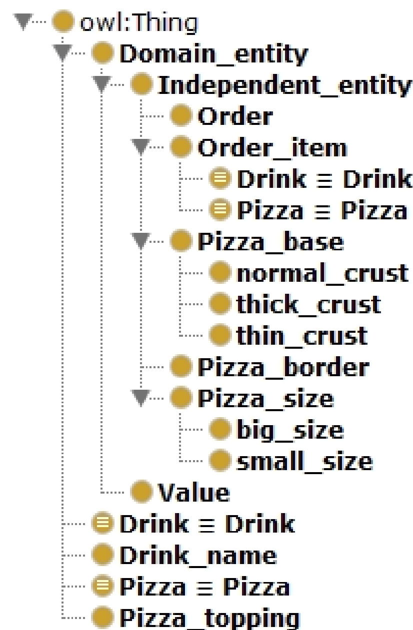


Figura 10: Estrutura da ontologia de atendimento de pedidos. Autoria própria.

7.3 Implementações Necessárias

Para incluir a funcionalidade de pedidos no sistema e satisfazer os requisitos funcionais da aplicação indicados na Fig. 7 é necessário implementar as seguintes estruturas:

- Unidade de Pedidos;
- Unidade de Cancelamento de Itens de Pedido.
- *Helper* de Confirmação do Pedido.

7.3.1 Unidade de Pedidos

A unidade de pedidos é a unidade principal do projeto, e se cadastra no corretor de mensagens para receber atos dialogais do tipo `{dimension: procedural; function: order; content: *}`. Seu funcionamento segue aquele descrito na Fig. 5. O campo *content* do ato dialogal que chega à unidade contém termos chaves separados por vírgula.

A unidade de pedidos checa então cada termo para ver se eles fazem parte da semântica de pedidos. Se algum dos termos não for válido, é enviado um erro semântico ao GL. Se todos os termos forem validados, o GD mapeia um novo item de pedido com as informações contidas no campo *content* e, caso as informações estejam incompletas semanticamente, esta unidade é carregada no controlador de fluxo de diálogo, a fim de completar a semântica do item do pedido (as informações necessárias para defini-lo). Por exemplo, no caso de uma pizza, precisamos ter como informação o sabor, tamanho e tipo de borda.

Para entender melhor como o GD mapeia os itens do pedido, tem-se um exemplo:

Ato dialogal de entrada: `{dimension: procedural; function: order; content: calabresa}`
Sequência de processos da unidade:
- checa se "calabresa" faz parte da semântica de pedidos (sim);
- checa se "calabresa" é um conceito (não);
- procura pelo nó principal relacionado a "calabresa" (pizza);
- procura pelo nó diretamente acima de "calabresa" (pizza_topping);
- cria-se uma nova ID para o item de pedido, e neste ID mapeia-se uma estrutura similar à encontrada na memória semântica;
- unidade carrega-se no controlador de fluxo do diálogo, passando também o ID do item de pedido criado para que as informações que faltam sejam completadas.

Ao final do exemplo, o GD possui mapeado uma estrutura de árvore com "pizza" > "pizza_topping" > "calabresa". Quando o controlador do fluxo de diálogo rodar a próxima requisição, será indicado primeiramente que falta informação sobre o tamanho do nó mapeado "pizza". A pergunta sobre o tamanho será encaminhada à pilha de requisições, e assim por diante. Ao completar semanticamente o item "pizza", o sistema procurará outros itens necessários no pedido (como bebidas) e enviará ao GL um ato dialogal que se traduzirá em "Acompanha bebidas?", por exemplo.

7.3.2 Unidade de Cancelamento de Itens de Pedido

A unidade de cancelamento de itens de pedido cadastra-se no corretor para receber mensagens do tipo `{dimension: procedural; function: cancel_order_item; content: *}`. As mensagens que chegam neste assinante possuem o campo *content* com a mesma estrutura que na unidade de pedidos. O processo da mensagem também se dá de maneira similar:

- checa-se semanticamente todos os termos que estão no campo *content*;
- se algum dos termos não faz parte da semântica, é enviado um erro semântico ao

GL.

Se o conteúdo da mensagem possuir uma semântica válida, esta unidade checa o mapa de respostas do GD por um item de pedido que possua os termos indicados e retira esse item do mapa. Caso nenhum item apresente a descrição, um erro de “pizza não encontrada em seu pedido” é enviado ao GL.

7.3.3 *Helper* de Confirmação do Pedido.

O *“helper”* de confirmação do pedido está relacionado à Unidade de Pedidos. Ele checa regularmente se a semântica do pedido está completa. No caso específico da pizzaria, é verificado se o GD possui uma pizza cadastrada no pedido e se já foi realizada a pergunta por bebidas. Caso a semântica tenha sido respondida, passa-se a requisitar a confirmação do pedido. Após receber uma confirmação positiva, este auxiliar altera o estado do *“helper”* de obrigações sociais para o agradecimento e posterior finalização da conversa com uma despedida.

7.4 Avaliação da Solução

Os testes da solução elaborada consistiram na utilização do sistema por voluntários, sendo avaliada a reação deles durante o processo, bem como os *feedbacks* ao final. Foram realizados testes com dez pessoas em diferentes ambientes. Os voluntários eram em sua maioria homens na faixa de 20 a 30 anos de idade. A Fig. 11 apresenta a análise da amostra.

Procurou-se avaliar os seguintes pontos de maneira qualitativa:

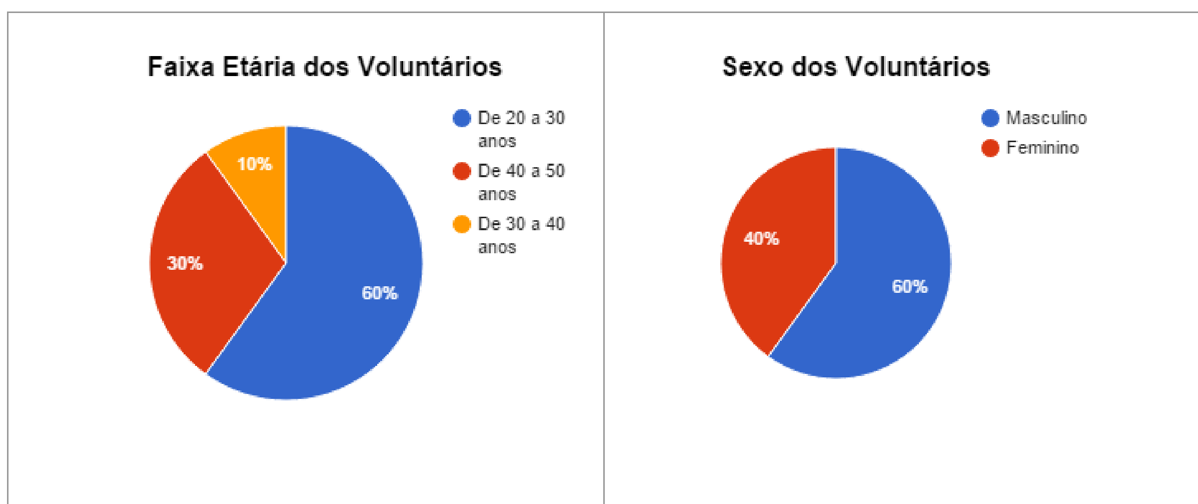
- Tempo de resposta do sistema;
- Grau de entendimento;
- Percepção do usuário.

A maneira como estes pontos foram analisados e os resultados obtidos são descritos nas seções a seguir.

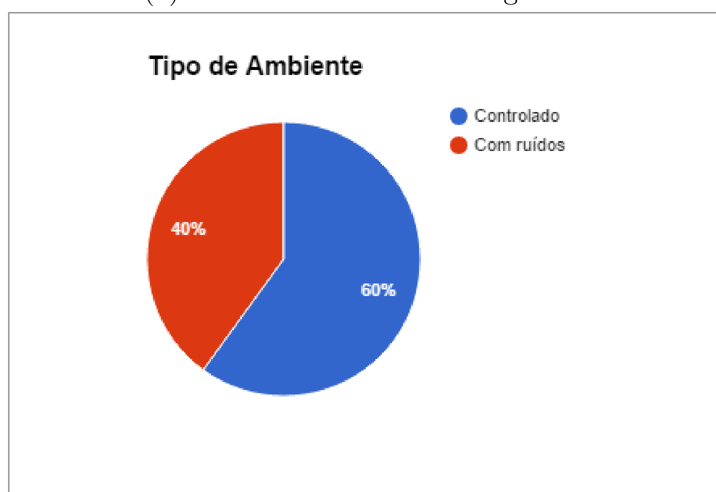
7.4.1 Tempo de Resposta do Sistema

Para o tempo de resposta, procurou-se avaliar se o sistema interagia com as pessoas de maneira que o intervalo entre as falas não deixasse de parecer natural. É necessário que o sistema não demore a responder o usuário para que a conversa flua naturalmente.

Em nenhum dos testes realizados o tempo de resposta foi um problema. Em todos eles o sistema foi capaz de interagir em intervalo de tempo compatível. Ao final do teste, perguntou-se se os voluntários notaram problemas no que dizia respeito ao tempo de resposta do sistema. Nove dos dez voluntários disseram não notar problema com o intervalo de tempo levado pelo sistema para respondê-los. Notou-se, porém, que em alguns ambientes o sistema demorou mais que em outros. Isto se deve ao uso de um ASR/TTS que depende de uma conexão com a *internet*. Nesses ambientes, a conexão não estava boa e, portanto, ocorria um atraso no reconhecimento e na síntese da voz.



(a) Gráficos de análise demográfica.



(b) Gráfico de tipos de ambiente.

Figura 11: Gráfico de análise qualitativa dos testes. Autoria própria.

7.4.2 Grau de Entendimento

Para o grau de entendimento, procurou-se avaliar se o sistema mapeou a linguagem necessária para manter uma conversa natural em um cenário de pedido de pizza. Caso isto não se mostrasse verdade, avaliou-se também quais eram os efeitos para a conversa se ocorresse um "*loop* de não-entendimento". Por *loop* de não-entendimento, entende-se casos em que o usuário tenta interagir com frases compatíveis no âmbito do pedido de pizza, porém não mapeadas no semantizador. Quando isso acontece, o sistema responde com "Desculpe, não entendi", o que pode fazer o usuário pensar que a frase não foi reconhecida corretamente, fazendo-o repeti-la muitas vezes.

Um teste destaca-se neste quesito. Durante ele, a pessoa tentou pedir sua pizza por meio da frase "uma pizza de mussarela". Porém, por um equívoco no semantizador, este modelo de frase não havia sido mapeado, o que resultou em um *loop* de não-entendimento. Após repetir cinco vezes a frase, a equipe que cuidava do teste teve que intervir e pedir para que frase fosse reestruturada. Deste teste, pôde-se perceber que as pessoas se irritam após não serem entendidas cerca de 3 vezes seguidas. Além disso, notou-se que o sistema não era capaz de lidar com problemas deste tipo.

Os outros testes correram normalmente, com apenas alguns problemas pontuais de entendimento. Nestes casos, ocorreram erros no reconhecimento da frase/palavra. Um dos erros de reconhecimento comuns encontrado em três dos testes aconteceu com a palavra "certo". Entre as saídas do ASR para esta palavra, obteve-se "Sérgio", "site" e "sério". Apesar destes problemas, os testes continuaram normalmente, não sendo necessário que a pessoa repetisse esta palavra mais do que duas vezes para ser entendida. De maneira geral, avaliou-se que o sistema foi capaz de reconhecer corretamente a entrada do usuário em 90% dos casos.

Uma das variáveis que inicialmente pensou-se ser um problema ao entendimento correto das frases foi o ruído ambiente. Metade dos testes foram realizados em ambientes com ruídos como salas de aula e restaurantes. Porém, na média, os erros obtidos nesses lugares foram equivalentes àqueles obtidos em ambientes controlados. Ou seja, os ruídos não se apresentaram como um problema para o funcionamento do sistema.

Ao final dos testes, perguntou-se aos dez voluntários sua opinião a respeito do grau de entendimento da conversa pelo sistema. Do total, sete consideraram que o sistema foi capaz de entendê-los bem, enquanto três apontaram pequenos problemas neste quesito.

7.4.3 Percepção do Usuário

Para a percepção dos usuários, perguntou-se ao final do teste o que os eles acharam a respeito do sistema. Procurou-se saber se, na percepção da pessoa, o diálogo fluiu de maneira natural ou se algo os causou estranheza no decorrer da conversa.

Nenhum dos voluntários reclamou da maneira como o sistema os guiou através do diálogo. Para eles, a conversa desenvolveu-se de forma parecida com o que aconteceria se estivessem pedindo pizza e falando com uma pessoa real. A única opinião negativa veio da pessoa que teve o diálogo comprometido pelo *loop* de não-entendimento. Para ela, o sistema deveria ser mais robusto, a fim de evitar este tipo de erro.

Quando perguntadas sua opinião geral a respeito do sistema, nove de dez pessoas o consideraram bom enquanto uma considerou o sistema regular.

7.5 Exemplos de Transcrição

Para servir como exemplo das situações encontradas e das análises obtidas por meio delas, três dos testes realizados foram transcritos nesta seção. Eles foram escolhidos por serem uma amostra representativa dos resultados encontrados. Nas transcrições, S representa uma fala dita pelo sistema, enquanto U corresponde a uma fala do usuário.

7.5.1 Exemplo 1

Teste realizado em um ambiente controlado, sem ruídos.

S: Bem vindo ao sistema de pedidos automático.
S: O que você deseja?
U: Uma pizza de mussarela.
S: Desculpe, não entendi o que você disse.
U: Uma pizza de mussarela.
S: Desculpe, não entendi o que você disse.
U: Uma pizza de mussarela.
S: Desculpe, não entendi o que você disse.
U: Uma pizza de mussarela.
S: Desculpe, não entendi o que você disse.

> Intervenção da equipe <

U: Queria uma pizza de mussarela.
S: Qual o tamanho da pizza?
U: Grande.
S: Gostaria de mais alguma coisa?
U: Não.
S: Seu pedido é uma pizza grande de mussarela, certo?
U: Isso mesmo.
S: Obrigado por pedir conosco.
S: Até logo.

7.5.2 Exemplo 2

Teste realizado em uma sala de aula (ambiente não controlado).

S: Bem vindo ao sistema de pedidos automático.
S: O que você deseja?
U: Quero uma pizza de tijolo.
S: Desculpe, não entendo tijolo.
U: Quero uma pizza.
S: Qual o sabor da pizza?
U: Calabresa.
S: Qual o tamanho da pizza?
U: Grande.
S: Gostaria de mais alguma coisa?
U: Não.
S: Seu pedido é uma pizza grande de calabresa, certo?
U: Sim.
S: Obrigado por pedir conosco.
S: Até logo.

7.5.3 Exemplo 3

Teste realizado em ambiente controlado.

S: Bem vindo ao sistema de pedidos automático.
S: O que você deseja?
U: Quero uma pizza de calabresa.
S: Qual o tamanho da pizza?
U: Grande.
S: Gostaria de mais alguma coisa?
U: Não.
S: Seu pedido é uma pizza grande de calabresa, certo?
U: Certo.
S: Desculpe, não entendi.
U: Certo.
S: Obrigado por pedir conosco.
S: Até logo.
U: Muito obrigado.
S: Desculpe, não entendi.

8 Conclusões

A escolha do *software* de ASR é crucial para obter-se a naturalidade esperada do sistema. Além do reconhecimento correto da fala do usuário, a necessidade de conexão com a *internet* é importante no momento de decidir por qual *software* utilizar. Ambas as características geram problemas na fluidez do diálogo, como analisado na Seção 7.4. Não se espera que o ASR possua uma taxa de acerto de cem por cento. Erros de entendimento são esperados até mesmo em uma conversa entre pessoas. O importante é saber lidar com estes erros de maneira satisfatória.

Outro ponto a ser destacado é o *loop* de não-entendimento, explicado na Seção 7.4.2.

Avalia-se que a topologia do Semantizador não é a ideal. O uso da linguagem AIML para tratar a saída do reconhecimento de voz mostrou-se insuficiente. Muitos recursos foram gastos no intuito de se mapear todas as possíveis estruturas de frases de entrada do diálogo. Recomenda-se para trabalhos futuros uma revisão da arquitetura e tecnologia empregadas no Semantizador, de forma a generalizá-lo. Uma possível melhoria seria o emprego de técnicas de aprendizagem de máquina para o reconhecimento de padrões de frases. A revisão do Semantizador deve ser acompanhada de modificações e crescimento da Memória Semântica. Estas ações melhorariam o entendimento do sistema.

Para aplicações reais, percebeu-se a necessidade de adicionar a comunicação com um banco de dados externo para armazenar os pedidos realizados pelo sistema. Esta interação deve ser incluída no passo final da confirmação do pedido, realizada pelo *helper* de confirmação do pedido. Também é necessária uma interface para alterar o cardápio disponível para o estabelecimento, o que refletiria em mudanças na ontologia do menu do restaurante.

Nota-se que o trabalho foi capaz de satisfazer os requisitos propostos inicialmente, conforme descrito na Seção 5. Os requisitos funcionais (estabelecidos nos casos de uso) puderam ser implementados, com exceção de "trocar item do pedido". Porém alguns, como o cancelamento do item do pedido, só puderam ser desenvolvidos após os testes. Desta forma, cabe realizar mais etapas de teste para análise destas funcionalidades e das melhorias feitas nas demais. Em relação aos requisitos subjetivos, verificou-se durante os testes que eles puderam todos ser atingidos, como explicado nas seções anteriores.

Propõe-se, para trabalhos futuros, a ampliação do módulo de memória episódica. Neste momento, o módulo apenas contém a pilha de requisições. Sua ampliação tornará possível um sistema de recomendações. Será possível utilizar informações de conversas anteriores e outros dados do usuário a fim de melhorar a fluidez do diálogo. Este desenvolvimento da memória episódica tornará possível pular passos da conversa, agilizando o processo do pedido.

O *framework* desenvolvido no projeto pode ainda ser ampliado para outros campos de atuação, conforme descrito na Seção 1.2. A utilização para pedidos de informação ou agendamentos, por exemplo, é possível a partir do desenvolvimento de novas unidades de memória procedural e da ampliação do conhecimento estruturado na memória semântica.

9 Referências

[1] A. RUDNICKY e W. XU, **An agenda-based dialog management architecture for spoken language systems**, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[2] C. LEE, S. JUNG, K. KIM, L. D. e G. G. LEE, **Recent approaches to dialog management for spoken dialog systems**, em *Journal of Computing Science and Engineering, Vol. 4, No. 1*, Department of Computer Science, Engineering, Pohang University of Science e Technology (POSTECH). Pohang, República da Coréia, 2010, pp. 1–22.

[3] D. SCHLANGEN e G. SKANTZE, **A general, abstract model of incremental dialogue processing**, em *Proceedings of the 12th Conference of the European Chapter of the ACL*, Grécia: Association for Computational Linguistics, 2009, pp. 710–718.

[4] D. BOHUS e A. I. RUDNICKY, **The ravenclaw dialog management framework: architecture and systems**, em *Computer Speech and Language 23*, Microsoft Research & Carnegie Mellon University, Estados Unidos, 2009, pp. 332–361.

[5] G. SKANTZE e A. HJALMARSSON, **Towards incremental speech generation in dialogue systems**, em *Proceedings of SIGDIAL 2010: the 11th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, Japão: Association for Computational Linguistics, 2010, pp. 1–8.

[6] V. S. Paulo. (2015). Dados da cidade de são paulo, endereço: `http://www.visitesaopaulo.com/dados-da-cidade.asp` (acesso em 02/11/2015).

[7] H. BUNT, J. ALEXANDERSSON, J.-W. CHOE, A. C. FANG, K. HASIDA, V. PETUKHOVA, A. POPESCU-BELIS e D. TRAUM, **Iso 24617-2: a semantically-based standard for dialogue annotation**, em *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, Istambul, Turquia, maio de 2012. Disponível em: <`http://people.ict.usc.edu/~traum/Papers/Buntetal-ISO24617-2.pdf`> , acesso em 14/09/2015.

[8] H. BUNT, M. KIPP e O. PETUKHOVA, **Using diaml and anvil for multimodal dialogue annotation**, em *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, Istambul, Turquia, 2012. Disponível em: <`http://semantic-annotation.uvt.nl/BuntKippPetukhovaLREC2012.pdf`> , acesso em 16/09/2015.

[9] H. BUNT, J. ALEXANDERSSON, J. CARLETTA, J.-W. CHOE, A. C. FANG, K. HASIDA, K. LEE, V. PETUKHOVA, A. POPESCU-BELIS, L. ROMARY, C. SORIA e D. TRAUM, **Towards an iso standard for dialogue act annotation**, em *Proceedings of the 7th International Conference on Language Resources and Evaluation (LREC)*, Valletta, Malta, maio de 2010. Disponível em: <`http://people.ict.usc.edu/~traum/Papers/lrec2010-iso-dacts-paper.pdf`> , acesso em 21/09/2015.

[10] F. MANOLA e E. MILLER, **Rdf primer**, W3C: W3C Working Group, 10 de fev. de 2004. Disponível em: <`http://www.w3.org/TR/2004/RECrdfprimer20040210/`> , acesso em 06/04/2015.

[11] G. SCHREIBER e Y. RAIMOND, **Rdf 1.1 primer**, VU University Amsterdam, BBC: W3C Working Group, 24 de jun. de 2014. Disponível em: <`http://www.w3.org/TR/rdf11-primer/`> , acesso em 06/04/2015.

[12] P. HITZLER, M. KRÖTZSCH, B. PARSIA, P. F. PATEL-SCHNEIDER e S. RUDOLPH, **Owl 2 web ontology language primer (second edition)**, Wright State University, University of Oxford, University of Manchester, Nuance Communications, FZI Research Center for Information Technology: W3C Recommendation, 11 de dez. de 2012. Disponível em: <http://www.w3.org/TR/owl2-primer/> , acesso em 05/04/2015.

[13] N. BUSH, **Artificial intelligence markup language (aiml)**, A.L.I.C.E. AI Foundation, 25 de out. de 2001. Disponível em: <http://www.alicebot.org/TR/2001/WD-aiml/> , acesso em 21/05/2015.

[14] T. RINGATE, R. S. WALLACE, A. TAYLOR, J. BAER e D. DANIELS, **Aiml primer**, 30 de out. de 2011. Disponível em: <http://www.alicebot.org/documentation/aiml-primer.html> , acesso em 21/05/2015.

[15] D. GRIMSHAW, **Jade administration tutorial**, Ryerson University, Toronto, Canadá, 26 de mar. de 2010. Disponível em: <http://jade.tilab.com/documentation/tutorials-guides/jade-administration-tutorial/> , acesso em 12/05/2015.

[16] G. CAIRE, **Jade tutorial and primer**, Telecom Italia Lab, Torino, Itália, 30 de jun. de 2009. Disponível em: <http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf> , acesso em 16/05/2015.

[17] G. SHIRES e H. WENNBORG, **Web speech api specification**, W3C Working Group, 19 de out. de 2012. Disponível em: <https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html> , acesso em 20/05/2015.

[18] S. McGlashan, D. C. Burnett, J. Carter, P. Danielsen, J. Ferrans, A. Hunt, B. Lucas, B. Porter, K. Rehor e S. Tryphonas, **Microsoft speech platform**. Disponível em: <https://msdn.microsoft.com/en-us/library/office/hh361572(v=office.14).aspx> , acesso em 20/05/2015.

[19] ——, **Voice extensible markup language (voicexml) version 2.0**, W3C Working Group, 16 de mar. de 2004. Disponível em: <http://www.w3.org/TR/voicexml20/> , acesso em 20/05/2015.

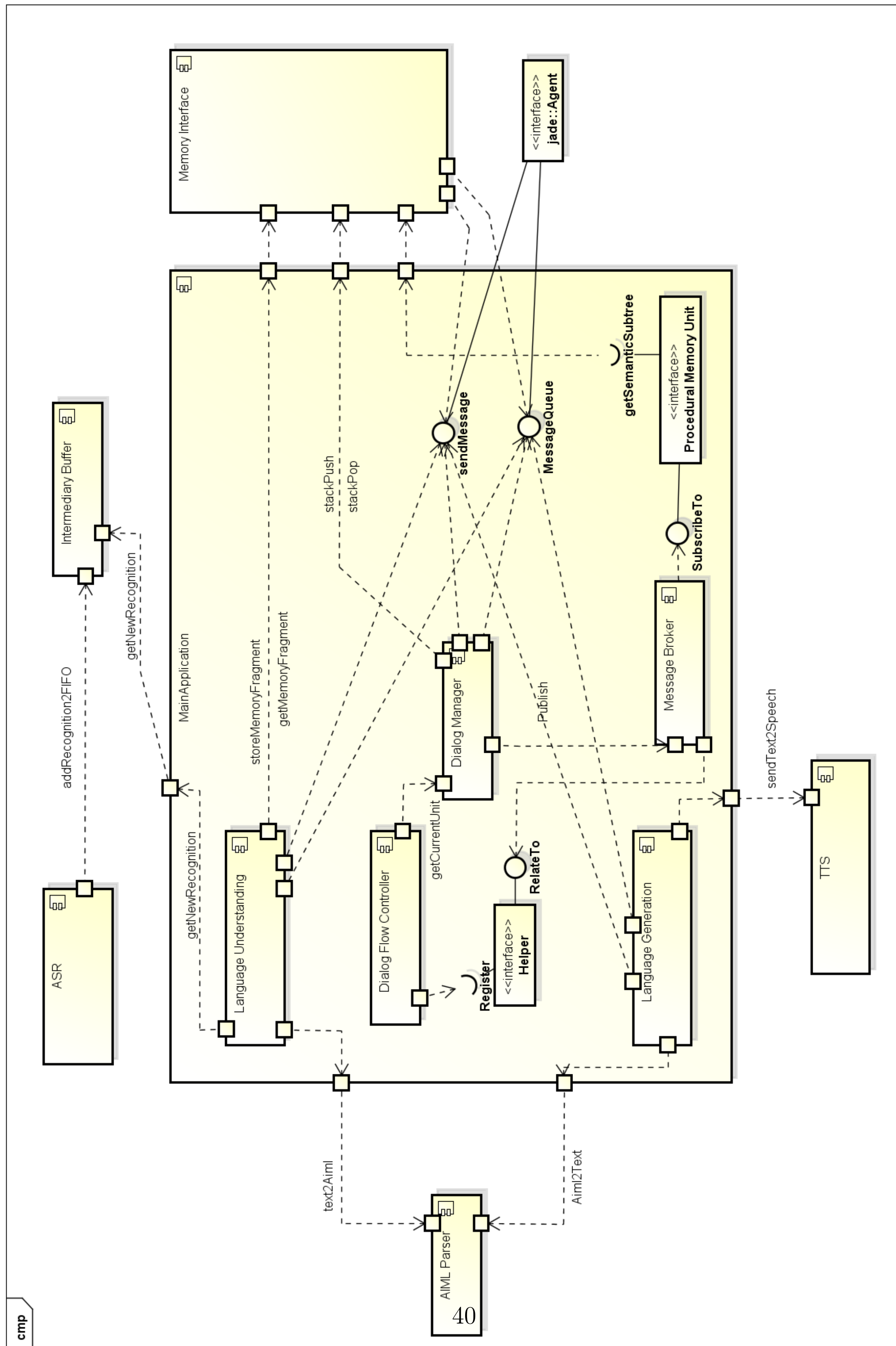
[20] A. HUNT e S. MCGLASHAN, **Speech recognition grammar specification version 1.0**, W3C Working Group, 16 de mar. de 2004. Disponível em: <http://www.w3.org/TR/speech-grammar/> , acesso em 20/05/2015.

[21] Foundation for Intelligent Physical Agents (FIPA), **Fipa acl message structure specification**, 3 de dez. de 2002. Disponível em: <http://www.fipa.org/specs/fipa00061/SC00061G.html> , acesso em 20/09/2015.

[22] ——, **Fipa communicative act library specification**, 3 de dez. de 2002. Disponível em: <http://www.fipa.org/specs/fipa00037/SC00037J.html> , acesso em 20/09/2015.

Apêndices

Apêndice A Diagrama de Componentes



Apêndice B Protocolo de Comunicação de Agentes

B.1 Communication Performative

B.1.1 Communication: ASR Module - Language Understanding

ACLMessage structure for this communication should be:

- Sender: sender's AID object.
- ConversationId: a java UUID object converted to string without "-". This field can be used to find the reply for a message when multiple behaviours of a same agent can handle the arrival of a new message.
- Ontology: the action that should be performed by the receiver. Available values are:
  - "new-recognition "
- Content: the output from the ASR module, which represents the last message received from the user (the arrival of a new recognition).
- Performative: should be INFORM when informing the arrival of a new recognition. The answer for the request will be
  - If the information was understood, the answer will be CONFIRM with the received message content in the content field;
  - If the agent could not understand the information, the answer will be FAILURE with the error message in the content field.

B.1.2 Communication: Language Understanding - Memory

ACLMessage structure for this communication should be:

- Sender: sender's AID object.
- ConversationId: a java UUID object converted to string without "-". This field can be used to find the reply for a message when multiple behaviours of a same agent can handle the arrival of a new message.
- Ontology: the action that should be performed by the receiver. Available values are:
  - "get-fragment"
  - "store-fragment"
- Content: the object (MemoryTypeRecognition) in a string format where each field is separated by the default separator " | ". This string should be reconstructed back in an object of the same type in its constructor. The object should be able to recreate a list of dialogue acts, which are represented by yet another object (DialogAct).

- Performative: should be REQUEST when requesting something. The answer for a request depends on the request
  - If the request is to store a new value in the memory, the answer will be CONFIRM with any additional information in the content field (i.e. the key where the value was stored);
  - If the request is to recover something stored in the memory, the answer will be INFORM with the requested object in the content field;
  - If the agent could not complete the request, the answer will be FAILURE with the error message in the content field.

B.1.3 Communication: Language Understanding - Dialog Manager

ACLMessage structure for this communication should be:

- Sender: sender's AID object.
- ConversationId: a java UUID object converted to string without "-". This field can be used to find the reply for a message when multiple behaviours of a same agent can handle the arrival of a new message.
- Ontology: the action that should be performed by the receiver. Available values are:
  - "new-recognition"
  - "new-response"
- Content: the object (MemoryTypeRecognition) in a string format where each field is separated by the default separator " | ". This string should be reconstructed back to an object of the same type in its constructor.
- Performative: should be INFORM when informing the arrival of a new recognition. The answer for the request will be
  - If the information was understood, the answer will be CONFIRM with the received message content in the content field;
  - If the agent could not understand the information, the answer will be FAILURE with the error message in the content field.

B.1.4 Communication: Dialog Manager - Language Understanding

ACLMessage structure for this communication should be:

- Sender: sender's AID object.
- ConversationId: a java UUID object converted to string without "-". This field can be used to find the reply for a message when multiple behaviours of a same agent can handle the arrival of a new message.

- Ontology: the action that should be performed by the receiver. Available values are:
  - "wait-response"
- Content: the object (ExpectationAgenda) in a string format where each field is separated by the default separator " | ". This string should be reconstructed back in an object of the same type in its constructor.
- Performative: should be INFORM when informing that the manager is waiting for an answer. The answer for the request will be
  - If the information was understood, the answer will be CONFIRM with the received message content in the content field;
  - If the agent could not understand the information, the answer will be FAILURE with the error message in the content field.

B.1.5 Communication: Dialog Manager - Memory

ACLMessage structure for this communication should be:

- Sender: sender's AID object.
- ConversationId: a java UUID object converted to string without "-". This field can be used to find the reply for a message when multiple behaviours of a same agent can handle the arrival of a new message.
- Ontology: the action that should be performed by the receiver. Available values are:
  - "get-fragment"
  - "store-fragment"
  - "empty-stack"
  - "stack-peek"
  - "stack-pop"
  - "stack-push"
- Content: the object (MemoryTypeRecognition or StackUnit) in a string format where each field is separated by the default separator " | ". This string should be reconstructed back to an object of the same type in its constructor.
- Performative: should be REQUEST when requesting something. The answer for a request depends on the request
  - If the request is to store a new value in the memory, the answer will be CONFIRM with any additional information in the content field (i.e. the key where the value was stored);
  - If the request is to recover something stored in the memory, the answer will be INFORM with the requested object in the content field;

- If the agent could not complete the request, the answer will be FAILURE with the error message in the content field.

B.1.6 Communication: Dialog Manager - Language Generation

ACLMessage structure for this communication should be:

- Sender: sender's AID object.
- ConversationId: a java UUID object converted to string without "-". This field can be used to find the reply for a message when multiple behaviours of a same agent can handle the arrival of a new message.
- Ontology: the action that should be performed by the receiver. Available values are:
  - "new-utterance"
- Content: the dialog act object, which represents the new output message (utterance). Such content follows the standard: {dimension: *; function: *; content: *}.
- Performative: should be INFORM when informing that a new utterance should be generated. The answer for the request will be
  - If the information was understood, the answer will be CONFIRM with the received message content in the content field;
  - If the agent could not understand the information, the answer will be FAILURE with the error message in the content field.

Apêndice C Atos Dialogais

Dialog Act Standards

Iteration 1

The following standards refer to iteration 1.

Important: the **values** for variables should be always enclosed between spaces (as in "[pizza]-size= normal ").

Dialog Dimensions

The following is a list of the dialog dimensions, which are based on ISO 24617-2. The string in parenthesis refer to the name used on the code as values for the *dialogDimension*variable:

- **Task/Procedural (procedural):** is related to the system's main task, like scheduling a meeting or ordering pizza.
- **Auto-Feedback (autoFeedback):** provides information about the speaker's processing of his/her own utterance.
- **Allo-Feedback (alloFeedback):** provides information about the speaker's processing of another's utterance.
- **Turn Management (turnManag):** is concerned with changes on the speaker role.
- **Time Management (timeManag):** acts for managing the use of time in the interaction.
- **Discourse Structuring (discourseStruct):** acts that refer to the dialog structure.
- **Own Communication Management (ownComm):** indicates that the speaker is editing his/her own current contribution to the dialog.
- **Partner Communication Management (partnerComm):** occurs when the listeners has the intention of assisting the speaker on his/her contribution to the dialog.
- **Social Obligations Management (socialOblig):** refers to acts that are required to fulfill social conventions, such as greeting or apologizing.

Dialog Functions

The following table lists the currently available functions for the dialog acts, divided according to their dimensions. All the function names are shown as implemented, or blank if there's no function created for the dimension yet. New ones should be added as soon as they are created.

Dimension: Procedural

Function	Description
inform	Used when the there's a new information relative to the main task, be it an answer or not. The content's structure is: variable_name, information.
order	Occurs on the utterance of a new order. The content's structure is:what_is_ordered,additional_information. For example: pizza, três queijos; coca-cola, 2litros.
ask	This function happens when the speaker requests information. The content's structure is: what_is_wanted, what_is_known. For example, to get a pizza's size, given its topping: pizza_size, pizza_topping=calabresa. In case there is no known information, just what is wanted is parsed on the content.
ask_to_include	This occurs when the speaker asks the listener if an item should be included on the order. The content refers to the type of item being asked, i.e. a content "drink" corresponds to a question "Do you want anything to drink?".
confirm_order	Used when the speaker wants to confirm an order. The content is a list of order item description, each item separated inside parenthesis (see the examples on the next section).
confirm_order_price	Used when the speaker wants to confirm the order's price. The content is an expression informing the price as a float with two significant digits, using a comma instead of the dot (e.g.: "price= 23,50 ").
confirm_order_item_cancel	Used for confirming that an order item was successfully canceled. The content specifies which order item is referred.
deny_order_item_cancel	Almost the same as the function above, but happens when the order item cancellation could not be completed for some reason.
cancel_order_item	Occurs when the speaker wants to cancel an item on the order he has made until that instant. The content is: order_item_name, additional_information. For example, the content for canceling a pizza item which has a calabreza topping is: pizza, calabreza.
change_order_item	This function happens when the speaker wants to change a specific item on his order. The content is: order_item_name, info_about_current_item, info_about_new_item. So, if the speaker wanted to change its pizza from calabreza to toscana, the content would be: pizza, calabreza, Toscana.

Dimension: Allo-Feedback

Function	Description
handle_error	Happens when the speaker wasn't able to understand the other's utterance, so it's trying to fix the communication error.
confirm	This function occurs when the speaker is confirming or denying the other's previous utterance. The content is "true" in case it's really a confirmation, or "false" when it's a denial.

Dimension: Social Obligations Management

Function	Description
greet	Used when there's a greeting, either on the start (as in saying "hi") or at the end of the conversation (as in saying "bye"). The content consists on the word used as greeting, or a more situational use (for example, at the end of the dialog, the content should be "final_bye").
thank	Used when the speaker needs to thank the listener. For a normal situation, the context is simply "thanks", but it is different on some more situational contexts (for example, at the end of the dialog, the content should be "final_thanks").
ask	This function happens on some complementing social obligation, such as "please" and "excuse me".

Dimension: Partner Communication Management

Function	Description
keep_alive	Happens when the speaker wasn't able to hear the other's last utterance, or the other took too much time to say anything. In this case, the speaker tries to maintain the conversation with the listener.

Examples

Basic dialog test

Dialog	LU 2 DM	DM 2 LG

Oi	{
	dimension: socialOblig;
	function: greet;
	content: hi
	}	

blhsasfsd	{
	dimension: unknown;
	function: unknown;
	content: blhsasfsd
	}	

Desculpe, não entendi		{
		dimension: alloFeedback;
		function: handle_error;
		content: did not understand
		}

Quero uma pizza de blah	{
	dimension: procedural;
	function: order;
	content: pizza, blah
	}	

Desculpe, mas não conheço pizza de blah		{
		dimension: procedural;
		function: inform;
		content: unknown pizza, blah
		}

Quero uma pizza de calabresa	{
	dimension: procedural;
	function: order;
	content: pizza, calabresa
	}	

Qual o tamanho da pizza?		{
		dimension: procedural;
		function: ask;
		content: pizza_size
		}

Broto (as an answer to “Qual o tamanho da pizza?”)	{
	dimension: procedural;
	function: inform;
	content: pizza_size= small
	}	

Seu pedido é uma pizza broto de calabresa, certo?		{
		dimension: procedural;
		function: confirm_order;
		content: (pizza, pizza_topping= calabresa , pizza_size= small)
		}

Sim (as an answer to "Seu pedido é *?")	{
	dimension: procedural;
	function: is_order_confirmed;
	content: true
	}	