UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

IMPLEMENTAÇÃO DO SUPORTE À PROGRAMAÇÃO E INTERPRETAÇÃO DA FALA NO AMBIENTE LTD

WENDEL DAVID PRZYGODA

BLUMENAU
2010

2010/1-25

WENDEL DAVID PRZYGODA

IMPLEMENTAÇÃO DO SUPORTE À PROGRAMAÇÃO E INTERPRETAÇÃO DA FALA NO AMBIENTE LTD

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. José Roque Voltolini da Silva - Orientador

BLUMENAU
2010

2010/1-25

IMPLEMENTAÇÃO DO SUPORTE À PROGRAMAÇÃO E INTERPRETAÇÃO DA FALA NO AMBIENTE LTD

Por

WENDEL DAVID PRZYGODA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. José Roque Voltolini da Silva – Orientador, FURB
Membro:	Profa. Joyce Martins, Mestre – FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB

Blumenau, 06 de Julho de 2010

Dedico este trabalho a todos os malucos que vivem na realidade o sonho dos caretas.

AGRADECIMENTOS

Primeiramente a toda minha família pela compreensão das minhas ausências nas festas e reuniões ocasionadas pelo desenvolvimento deste trabalho.

Agradeço principalmente a minha noiva Taynara, que foi quem mais me apoiou, incentivou e me deu forças para chegar até onde cheguei em minha vida, além da paciência e compreensão por não ficar ao seu lado por muitas noites e fins de semana por conta deste projeto.

Minha cadelinha Evey por não me morder nas vezes que não joguei bola quando ela pedia para brincar e também por me fazer lembrar o pouco que é necessário para ser feliz.

Aos colegas de trabalho e à chefia por compreenderem algumas ausências (espero não ter deixado ninguém na mão).

Ao meu orientador José Roque Voltolini da Silva por aceitar o desafio e acreditar que sempre é possível melhorar. Também ao professor Dalton Solano dos Reis que foi o primeiro professor que compartilhei da idéia deste projeto e deu o empurrão inicial.

Aos companheiros Denis R. Costa, Germano Fronza, Israel Damasio e todos da *Faculté Polytechnique de Mons* (Bélgica), os quais não conheci, mas que sem ter como base o trabalho deles no que tange ao MBROLA, este trabalho não seria possível.

A vingança nunca é plena, mata a alma e a envenena.

Seu Madruga

RESUMO

Este trabalho apresenta uma extensão do ambiente LTD (ambiente visual de programação voltado a aprendizagem de programação de computadores), a qual é a introdução do comando de fala na linguagem LTD. O comando fala realiza a síntese da voz a partir de um texto informado. Ainda a voz (fala) é sincronizada com a animação. Apresenta também a especificação JSML da *Sun Mycrosystems* destinada à síntese de voz, a implementação de um sintetizador para o idioma português e a utilização do MBROLA para o processamento acústico.

Palavras-chave: Síntese de voz. Tangram. LTD. Texto-fala. Processamento prosódico.

ABSTRACT

This project presents the LTD (visual environment towards the learning computer programming) environment's extension, which the introduction of the command fala in LTD language. The command performs a speech synthesis from a text informed. The voice (speech) is synchronized with the animation. It also presents a JSML specification from Sun Microsystems intended text-to-speech synthesis, a synthesizer implementation in portuguese language using MBROLA for acustic process.

Key-words: Speech synthesis. Tangram. LTD. Text-to-speech. Prosodic process.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ACSS – *Audio Cascade Style Sheet*

API – *Application Programming Interface*

OpenGL – *Open Graphics Library*

BNF – Backus-Naur *Form*

CDATA – *Charactere DATA*

CSS – *Cascade Style Sheet*

Flite – *Festival-lite*

GALS – Gerador de Analisador Léxico e Sintático

JOGL – *Java OpenGL*

JSGF – *Java Speech Grammar Form*

JSML – *Java Speech Markup Language*

HMM – *Hidden* Markov *Models*

HTML – *HyperText Markup Language*

LTD – *Language Tangram Draw*

MBROLA – *Multi-Band Re-sinthesys pitch synchronous OverLap Add*

PSOLA – *Pitch Synchronous OverLap Add*

TAD – Tipo Abstrato de Dados

URL – *Uniform Resource Locator*

VoiceXML – *Voice eXtensible Markup Language*

XHTML – *eXtensible HyperText Markup Language*

XML –*eXtensible Markup Language*

WAVE – WAVEform audio file format

SUMÁRIO

1 INTRODUÇÃO

Segundo Knihs (2008, p.26), o ambiente *Language Tangram Draw* (LTD) tem por objetivo facilitar o ensino de programação de computadores através de uma linguagem visual.

Junto com a visão, a fala é o meio de comunicação muito utilizado entre as pessoas. Em função disso, softwares de síntese de voz são largamente utilizados em aplicações de acessibilidade, como leitores de textos, principalmente para deficientes visuais.

Para realizar a síntese de voz a partir de um texto é necessário executar um processamento. Segundo Paula (2000, p. 267), existe os seguintes tipos de processamento de voz:

a) operações de puro processamento de áudio, tais como transformações de duração e altura, filtragens e principalmente as técnicas de compressão de voz;
b) síntese da voz, que compreende a transformação de texto em sequências de fonemas e desses em sinal de áudio;
c) reconhecimento de voz, que, a partir de um sinal de áudio, procura identificar fonemas, a fim de reconhecer palavras e finalmente frases, tais como comandos para o computador.

Visto o acima, este trabalho apresenta o desenvolvimento de um protótipo para implementar a síntese da voz a partir de textos descritos através do ambiente LTD, fazendo com que, além de possuir recurso visual da animação durante a execução de um programa feito em LTD, seja possível também ouvir um texto, através da síntese do mesmo em fala.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é disponibilizar a programação da fala aos modelos criados no LTD.

Os objetivos específicos do trabalho são:

a) disponibilizar comandos na linguagem do LTD para atribuir um texto que será sintetizado durante a execução do programa;
b) interpretar o programa, sintetizando o texto através da fala, conforme especificado;

c) sincronizar a fala com a animação feita no LTD.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 apresenta os assuntos relacionados ao trabalho, tais como: o ambiente LTD, síntese de voz e trabalhos correlatos. No capítulo 3 é descrito o desenvolvimento da nova versão da ferramenta. Por fim, o capítulo 4 traz as conclusões do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

A seguir são apresentados os assuntos: o ambiente LTD, síntese de voz e trabalhos correlatos.

2.1 O AMBIENTE LTD

O LTD foi inicialmente implementado por Alcântara Jr (2003), reimplementado por Theiss (2006) e continuado por Knihs (2008).

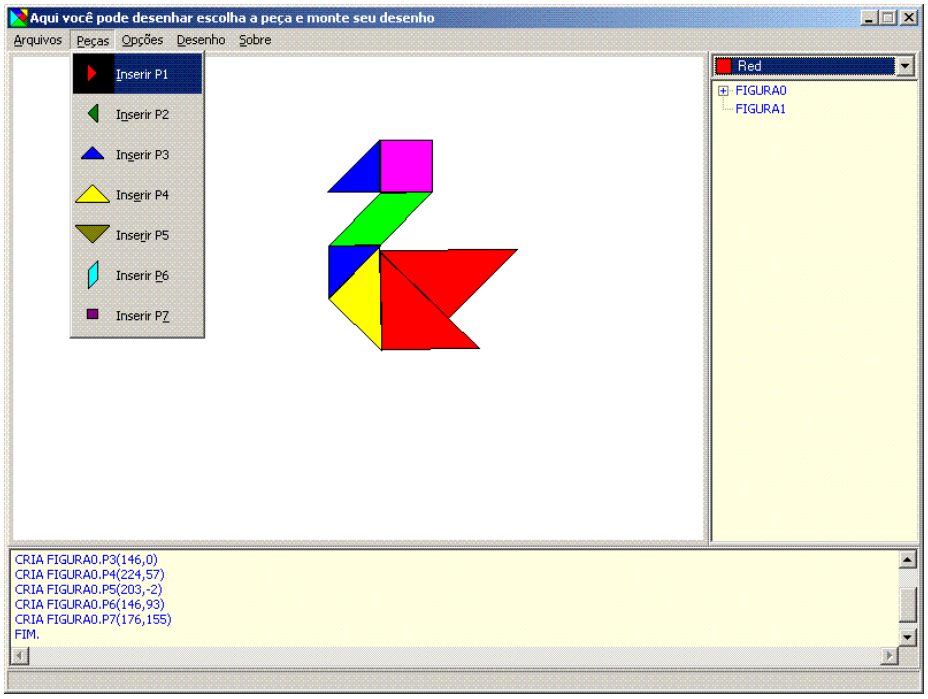
Segundo Knihs (2008, p. 6), o objetivo do LTD é o de facilitar a programação baseada em Tipos Abstratos de Dados (TAD) através de um ambiente visual. O LTD é uma ferramenta que possui um ambiente de programação visual, a qual usa formas geométricas que compõem o jogo matemático chamado Tangram, tendo como objetivo o ensino de programação de computadores para crianças.

> As linguagens de programação visuais partem do princípio de que gráficos são mais fáceis de serem entendidos do que textos. Quando se especifica um programa por meio de diagramas e outros recursos gráficos, mesmo os usuários sem muita habilidade em programação podem gerar programas pelas facilidades que os recursos gráficos oferecem. (GUDWIN, 1997, p. 13 apud KNIHS, 2008, p. 15).

O LTD também possui uma linguagem própria que pode ser classificada como sequencial. Com o intuito de ampliar o LTD, Knihs (2008) criou uma linguagem baseada em TAD.

Para mostrar a evolução do LTD ao longo de seu desenvolvimento, cita-se características marcantes em suas versões, as quais são:

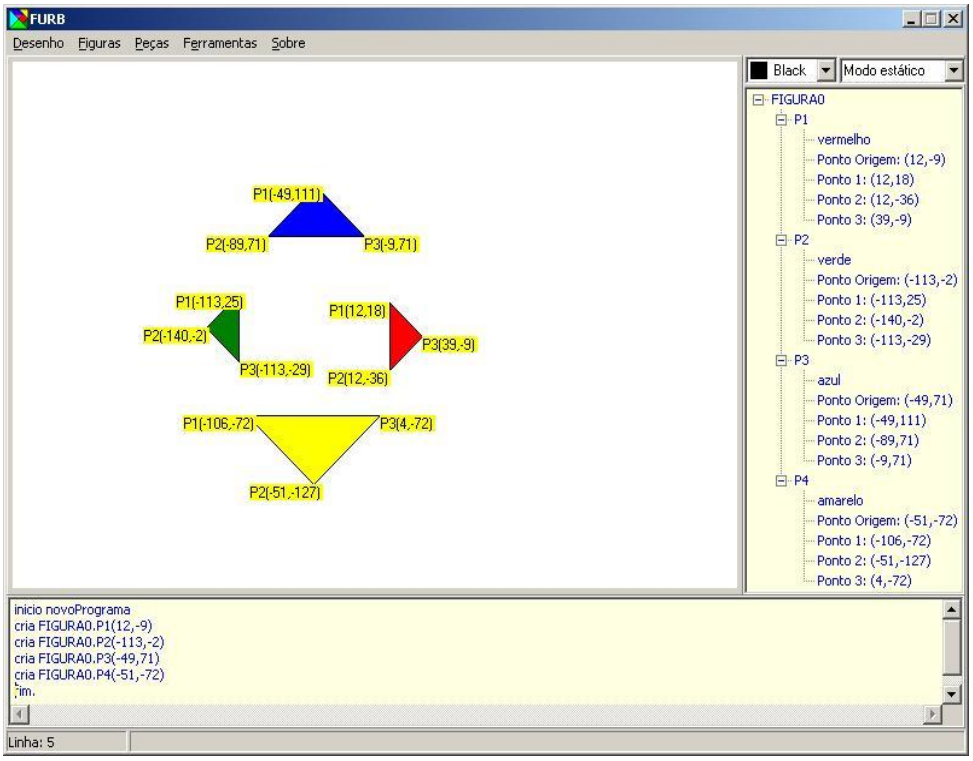
a) o ambiente desenvolvido por Alcântara Jr (2003) (Figura 1) disponibiliza uma linguagem textual de ligação com a visual, onde possui 2 editores, um visual com os modelos e outro com o texto, representando os comandos;



Fonte: Alcântara (2003, p. 53).

Figura 1 – Ambiente LTD implementado por Alcântara (2003)

b) Theiss (2006) re-implementou o LTD, corrigindo algumas funções e adicionando novas funcionalidades como a visualização em terceira dimensão e a seleção de peças e figuras, utilizando a *Application Programming Interface Open Graphics Library* (API OpenGL) (Figura 2);



Fonte: Theiss (2006, p. 75).

Figura 2 – Tela do editor de figuras do ambiente LTD

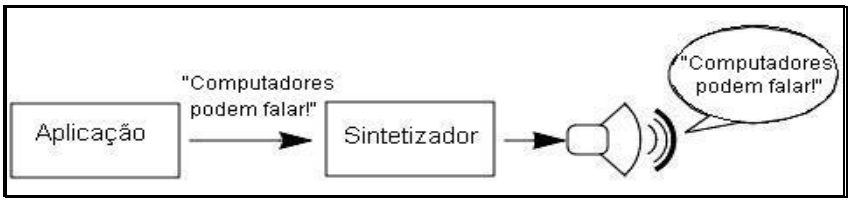
c) Knihs (2008) reimplementou o LTD codificando na linguagem Java, anteriormente implementado em *Object Pascal*, juntamente com a biblioteca *Java OpenGL* (JOGL). Ainda, implementou uma linguagem baseada em Tipo Abstrato de Dados (TAD) e disponibilizou o suporte a multiprocessamento dos modelos e criando para isso novos comandos na linguagem. A Figura 3 mostra o editor textual de comandos do ambiente LTD.

Figura 3 – Tela do editor de comandos do ambiente LTD implementado por Knihs (2008)

2.2 SÍNTESE DE VOZ

Segundo Passos (2002), a síntese de sinais de voz é um problema que pode ser resolvido por meio de métodos baseados na síntese subtrativa digital. Um destes métodos nada mais é do que um software que aplica um filtro de ruídos numa representação digital de uma onda sonora analógica, onde cada fonema é descrito por um conjunto de parâmetros de filtro que determinam a formação do som produzido. Paula (2000, p. 267) explica a síntese subtrativa como o ato de produzir sons de timbre variável por meio de cordas vocais, cujo som produzido é filtrado pela boca e nariz. Ainda, segundo Paula (2000, p. 268), estes diferentes timbres são percebidos como fonemas, ou unidades de som, das linguagens faladas. Uma sequência de fonemas forma uma palavra.

A especificação da Sun Microsystems (2009), do *Java Speech Markup Language* (JSML), provê suporte a anotações textuais que fornecem informações adicionais para o sintetizador de voz aumentar a qualidade do som de saída. A Figura 4 exemplifica a estrutura de funcionamento de um sintetizador de voz.



Fonte: Sun Microsystems (2009, tradução nossa).

Figura 4 – Transformação de um texto de uma aplicação em áudio pelo sintetizador

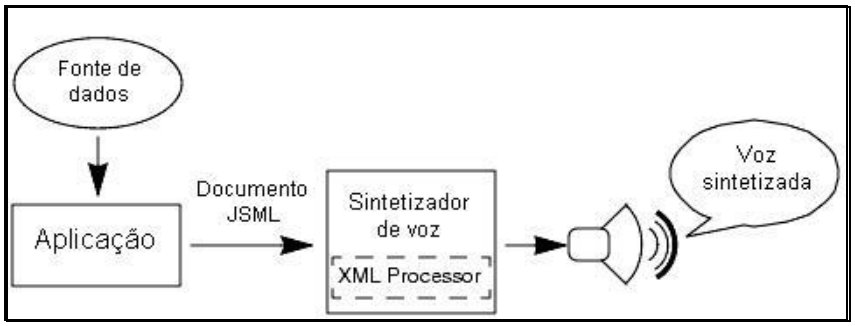
Na JSML é possível indicar na sentença “Computadores podem falar!”, que a palavra “podem”, por exemplo, tenha uma ênfase na fala, ou que alterando o ponto de exclamação por um ponto de interrogação, seja alterada a entonação da frase para representar uma pergunta.

As características da JSML, segundo Sun Microsystems (2009), são:

a) provê um controle consistente da saída da voz do sintetizador;
b) é possível produzir som de saída com voz de textos e também uma vasta gama de aplicativos;
c) é internacionalizável, o que permite que seja implantados tantos idiomas quantos forem necessários;
d) é fácil processar e escrever documentos no formato JSML;
e) para consistência e compatibilidade, todas as características podem ser implementadas de acordo com a tecnologia utilizada;
f) documentos JSML são humanamente legíveis, pois contém apenas o texto a ser sintetizado e informações adicionais a estes textos;
g) concisão do documento tem pouca importância, ou seja, as informações não precisam necessariamente fazer sentido.

Documentos JSML podem prover ao sintetizador várias fontes de dados, como livros, documentos técnicos, páginas *web*, *e-mails*, autoatendimento e informações em terminais aéreos para que se produza a voz sintetizada como saída de áudio.

Porém, com fontes de dados apenas passando textos planos ou textos simples de um livro ou um artigo para um sintetizador de voz, o sintetizador é capaz de criar variações na entonação da voz criada como interrogações e ênfases nas frases. Para isso, a aplicação deve processar este texto plano colocando informações para o sintetizador poder produzir uma voz mais clara e similar à do ser humano. Na Figura 5 é exemplificado o comportamento descrito acima.



Fonte: Sun Microsystems (2009, tradução nossa).

Figura 5 – Processamento JSML

Considerando um exemplo da leitura de uma página *web*, a fonte de dados é uma página *HyperText Markup Language* (HTML), que eventualmente utiliza *Cascade Style Sheet* (CSS) ou *Áudio Cascade Style Sheet* (ACSS), que provêem dados adicionais de como renderizar a página visualmente e auditivamente. A aplicação que processa a página *web* é o *browser*. Para renderizar um documento HTML visualmente, um *browser* controla a parte gráfica para escrever caracteres e imagens como saída na tela. Para controlar uma página HTML com conteúdo de fala, o *browser* deve prover ao sintetizador um documento JSML (SUN MICROSYSTEMS, 2009).

O mesmo aplica-se para um leitor de *e-mails*. Quando um leitor de *e-mails* converte uma mensagem em um texto falado, pode além de falar apenas o conteúdo com entonações na voz, falar corretamente datas, horários, valores monetários, siglas, etc.

Um documento JSML é composto por elementos das seguintes categorias: estrutura, produção e diversos. O Quadro 1 fornece a descrição dos elementos.

Função do Elemento	**Nome do Elemento**	**Tipo do Elemento**	**Descrição do Elemento**
Estrutura	`jsml`	*Container*	Define o elemento raiz do documento
	`div`	*Container*	Demarca estruturas de texto como parágrafos e sentenças
Produção	`voice`	*Container*	Especifica a voz utilizada para falar o texto contido
	`sayas`	*Container*	Especifica como o texto será lido
	`phoneme`	*Container*	Especifica que o texto contido é um fonema
	`emphasis`	*Container*	Especifica a ênfase que o texto contido será lido
	`break`	vazio	Efetua uma pausa na leitura
	`prosody`	*Container*	Especifica propriedades prosódicas para o texto que será falado como frequência, volume e velocidade
Diversos	`marker`	vazio	Notifica um evento ao sintetizador quando a fala chegar a este elemento
	`engine`	*Container*	Configura ou define um sintetizador

Fonte: adaptado de Sun Microsystems (2009).

Quadro 1 – Elementos de um documento JSML

O elemento `jsml` é o elemento raiz do documento. Ele contém todos os demais elementos, entidades, seções *Charactere DATA* (CDATA[1]) e textos sem marcação. O Quadro 2 mostra este elemento e sua configuração.

Atributo	Obrigatoriedade	Descrição
`lang`	opcional	Determina o idioma que o sintetizador irá utilizar, seguindo o formato padrão de internet RFC 1766
`mark`	opcional	Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 2 – Atributos do elemento `jsml`

O elemento `div` é um *container* utilizado para identificar que o texto contido nele é uma sentença ou um parágrafo, servindo assim como um divisor de estruturas de texto. O Quadro 3 mostra a configuração do elemento `div`.

Atributo	Obrigatoriedade	Valor padrão	Descrição
`type`	obrigatório	`sentence / sent`	Indica o tipo da estrutura no texto contido no elemento
		`paragraph / para`	
`mark`	opcional		Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 3 – Atributos do elemento `div`

O elemento `voice` descrito no Quadro 4 é responsável por definir a voz utilizada no processo de síntese.

O elemento `sayas` é o elemento que fornece uma informação semântica sobre o texto contido para o sintetizador, como por exemplo, ler o número "2010" e falar como "dois zero um zero" ou "dois mil e dez", ou este texto "2/5" pode ser falado como "dois barra cinco" ou "dois quintos". O mesmo para uma hora como "18:20", que é falado "dezoito horas e vinte minutos" ao invés de ler literalmente "dezoito dois pontos vinte". No Quadro 5 é descrito o elemento `sayas` e no Quadro 6 são descritos os possíveis valores para o atributo `class`.

[1] Uma seção CDATA de um eXtensible Markup Language (XML) indica que o texto contido é apenas de caracteres e não de marcação, sendo assim possível ter caracteres como "<", ">" e "&".

Atributo	Obrigatoriedade	Valor padrão	Descrição
gender	opcional	male	Gênero da voz que será sintetizada. Define-se male para masculino, female para feminino e neutral para uma voz sem gênero, como a de um robô por exemplo
		female	
		neutral	
age	opcional	child	A idade que a voz irá ter. Além destes valores padrões pode ser um número inteiro simbolizando a idade onde, child é de 0 a 12, teenager de 13 a 19, younger_adult de 20 a 39, middle_adult de 40 a 59 e older_adult de 60 em diante. O valor adult apenas, representa qualquer idade acima de 20
		teenager	
		younger_adult	
		middle_adult	
		older_adult	
		adult	
variant	opcional		Representa uma variante da voz, algo que o sintetizador consiga diferenciar uma voz de um adulto para outra voz de adulto
name	opcional		Nome da voz que será utilizada
mark	opcional		Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 4 – Atributos do elemento voice

Atributo	Obrigatoriedade	Valor padrão	Descrição
class	obrigatório	literal	Lê o texto contido segundo a semântica passada. Este atributo está sujeito a diferenças de países e idiomas
		date	
		time	
		name	
		phone	
		net	
		address	
		currency	
		measure	
		number	
mark	opcional		Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 5 – Atributos do elemento sayas

Classe	Descrição
`literal`	Soletra um texto
`date`	Especifica que o texto será lido como uma data
`time`	Especifica que o texto será lido como um horário
`name`	Especifica que o texto lido é um nome próprio
`phone`	Especifica que o texto lido é um número telefônico
`net`	Especifica que o texto lido é um *e-mail* ou uma *Uniform Resource Locator* (URL)
`address`	Especifica que o texto lido é um endereço
`currency`	Especifica que o texto lido é um valor monetário
`measure`	Especifica que o texto lido é uma medida
`number`	Especifica que o texto lido é um número inteiro, fracionário ou de ponto flutuante

Fonte: adaptado de Sun Microsystems (2009).

Quadro 6 – Valores do atributo `class`

O elemento `phoneme`, descrito no Quadro 7, informa que seu texto deve ser lido pelo sintetizador como um fonema.

Atributo	Obrigatoriedade	Descrição
`original`	opcional	Indica o texto descrito pelo fonema. Normalmente ignorado pelo sintetizador, mas utilizado para fins de *loggin* e depuração
`mark`	opcional	Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 7 – Atributos do elemento `phoneme`

O elemento `emphasis` informa ao sintetizador que seu texto deve ser lido com uma entonação diferente, sendo mais ou menos enfático, conforme descrito no Quadro 8.

Atributo	Obrigatoriedade	Valor padrão	Descrição
`level`	opcional	`strong`	Indica o nível de ênfase, onde `moderate` é o valor padrão caso não seja informado
		`moderate`	
		`none`	
`mark`	opcional		Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 8 – Atributos do elemento `emphasis`

O elemento `break` é apenas para informar ao sintetizador que uma pausa deve ser feita, ficando em silêncio conforme o tempo estipulado. O Quadro 9 demonstra a configuração deste elemento.

Atributo	Obrigatoriedade	Valor padrão	Descrição
size	opcional	none	Especifica a duração da pausa baseada no contexto da fala
		small	
		medium	
		large	
time	opcional		Tempo em milissegundos que a pausa irá durar
mark	opcional		Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 9 – Atributos do elemento `break`

O elemento `prosody` é utilizado para informar ao sintetizador sobre volume, freqüência e velocidade que o texto deve ser falado. Sua configuração está descrita no Quadro 10.

Atributo	Obrigatoriedade	Valor padrão	Descrição
rate	opcional	fast	Velocidade com que o texto contido será falado. Além dos valores descritivos padrões, este atributo é do tipo inteiro, e seu valor representa o número de palavras faladas por minuto
		medium	
		slow	
		default	
volume	opcional	loud	Volume utilizado ao falar o texto contido. Além dos valores descritivos padrões, este atributo é do tipo numérico que vai de 0.0 a 1.0
		medium	
		quiet	
		default	
pitch	opcional	high	Comprimento de onda da voz em Hertz (Hz)
		medium	
		low	
		default	
range	opcional	high	Altura de onda da voz em Hertz (Hz)
		medium	
		low	
		default	
mark	opcional		Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 10 – Atributos do elemento `prosody`

O elemento `marker` possui o mesmo funcionamento que o atributo `mark` pertencente a todos os elementos, apenas o de notificar o sintetizador sobre uma marcação, sem que seja necessário utilizar um elemento com uma fala atribuída. O Quadro 11 descreve sua configuração.

Atributo	Obrigatoriedade	Valor padrão	Descrição
`mark`	Opcional		Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 11 – Atributo do elemento `marker`

O elemento `engine` permite a utilização de outra implementação de sintetizador, ou de uma variação do mesmo, como por exemplo, num documento JSML que possua duas *engines* (PSOLA e MBROLA), cada um irá sintetizar as falas atribuídas a si. Sua configuração está descrita Quadro 12.

Atributo	Obrigatoriedade	Descrição
`name`	opcional	Identifica uma ou mais vozes, separadas por vírgula, utilizadas pelo sintetizador
`data`	obrigatório	Texto que será falado especificamente por esta voz durante a síntese
`mark`	opcional	Notifica o sintetizador que a fala alcançou esta marcação

Fonte: adaptado de Sun Microsystems (2009).

Quadro 12 – Atributos do elemento `engine`

Um exemplo de como pode ser criado um documento JSML combinando a utilização de vários elementos pode ser visto no Quadro 13.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<jsml lang="pt-br" mark="mark_jsml">
      <div type="paragraph" mark="mark_div">Este texto é um
parágrafo.</div>
      <voice gender="female" age="adult" variant="6" name="voice_name"
            mark="mark_voice"></voice>
      <div type="sentence">Diário de bordo, dia</div>
      <sayas class="number" mark="mark_sayas_number">365.</sayas>
      <div type="sentence">Data estelar</div>
      <sayas class="date" mark="mark_sayas_date">11/05/2010</sayas>
      <emphasis level="strong" mark="mark_emphasis">às</emphasis>
      <sayas class="time" mark="mark_sayas_time">01:26.</sayas>
      <sayas class="measure" mark="mark_sayas_measure">753KM.</sayas>
      <sayas class="literal">FURB.</sayas>

      <break size="large" time="3s" mark="mark=break">pausa
dramática</break>

      <prosody rate="medium" volume="loud" pitch="default"
range="default" mark="mark_prosody">Este é um exemplo verborrágico de
síntese de voz.</prosody>

      <engine name="SKYNET synth" data="SKYNET"
mark="mark_engine">Porquê?</engine>
</jsml>
```

Quadro 13 – Exemplo de um documento JSML

Para a geração digital da voz existe um *software* chamado *Multi-Band Re-sinthesys Pitch Synchronous OverLap Add* (MBR-PSOLA, ou apenas MBROLA) onde, segundo o MBROLA Team (2000), utiliza um algoritmo de mesmo nome, que foi desenvolvido pela *Faculté Polythechnique de Mons*, situada na Bélgica. O algoritmo MBROLA é derivado de outro algoritmo chamado *Pitch Synchronous OverLap Add* (PSOLA), que foi originalmente desenvolvido pela *France Telecom*.

Segundo MBROLA Team (2000), o PSOLA é basicamente dividido em 3 passos:

a) análise da fonte de som, que neste caso já é digital;
b) separação de sinais através da filtragem subtrativa;
c) reconstrução dos sinais e sintetização da voz.

O MBROLA suporta várias bases de dados de idiomas de voz. Estas bases de dados são arquivos com as definições de voz para cada fonema conforme seu idioma.

2.3 TRABALHOS CORRELATOS

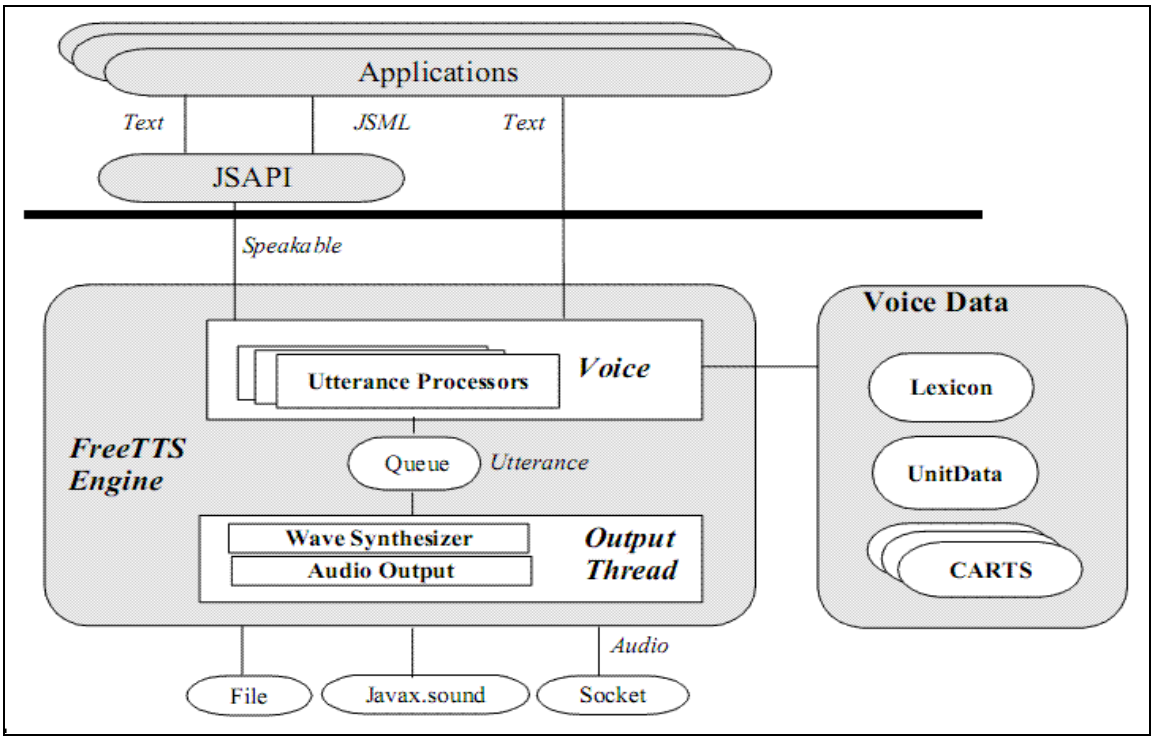
A seguir são apresentadas ferramentas que utilizam sintetizadores de voz ou reconhecimento de voz, as quais são: FreeTTS (SOURCEFORGE, 2001), *Sphinx4* (SOURCEFORGE, 2008), *Opera Voice* (OPERA DEVELOPER, 2009) e o protótipo de Oeschsler (2009).

2.3.1 Free Text-To-Speech (FreeTTS)

Segundo SourceForge (2001), o FreeTTS é uma ferramenta *open source* desenvolvida em Java, baseada no *Festival-lite* (Flite) que é uma versão compacta do *Festival*. O *Festival* é uma ferramenta que auxilia no desenvolvimento de *softwares* de síntese de voz. O FreeTTS possui uma *engine* de síntese de voz que suporta formatos de diferentes qualidade de voz no idioma inglês e suporte parcial à JSML da JSAPI. Os algoritmos são os mesmos do Flite, apenas com algumas modificações na estrutura de dados para melhoria de performance.

Outro recurso que o FreeTTS disponibiliza é a possibilidade de se trabalhar usando o modelo cliente-servidor. Neste modelo o cliente tem o papel de enviar o texto ao servidor, que por sua vez processa o texto e retorna ao cliente os dados que representam a voz, cabendo ao

cliente apenas reproduzi-la (SOURCEFORGE, 2001). A Figura 6 mostra a arquitetura do FreeTTS.



Fonte: Sourceforge (2001).

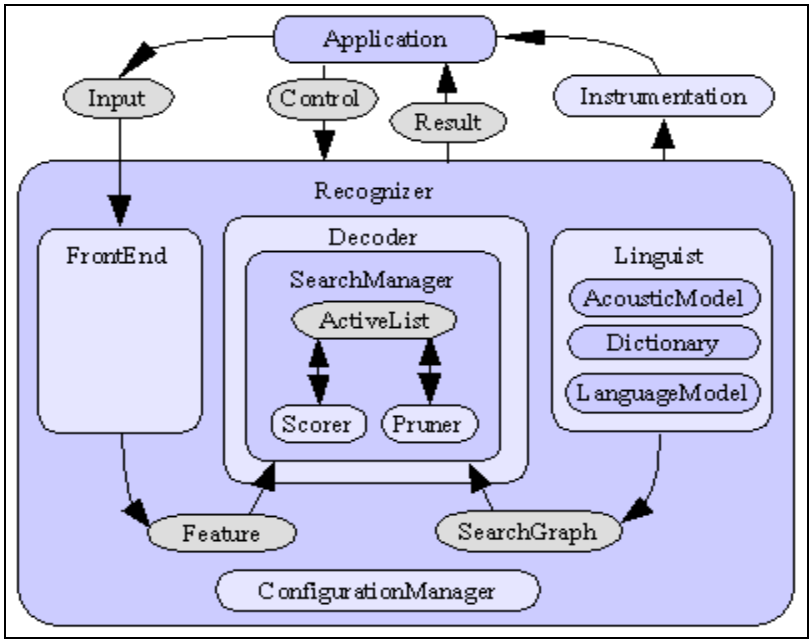
Figura 6 – Arquitetura do FreeTTS

O mapa arquitetônico do FreeTTS (Figura 6) mostra que um aplicativo pode enviar um documento JSML ou um texto simples à *engine* do sintetizador. Esse por sua vez irá realizar uma análise do texto de entrada utilizando dados da voz utilizada na fala, como gênero, frequência, etc. Esta voz contém informações de como são as características dos fonemas do idioma do texto que está sendo sintetizado. Por fim, gera a fala do texto. As possibilidades de obter esta fala resultante são: através de arquivo de áudio, direto para a placa de som ou para um *socket*.

2.3.2 Sphinx4

Segundo SourceForge (2008), o Sphinx4 é uma ferramenta escrita em Java que suporta JSAPI, neste caso a *Java Speech Grammar Form (*JSGF). Sua função é a de reconhecimento de comandos por voz. É derivada da sua versão 3 que foi desenvolvida em C++. A versão 4, desenvolvida em Java, foi criada de forma colaborativa entre várias universidade norte-americanas e empresas privadas. Ela também fornece uma API para que se possa acoplar um reconhecedor de comando de voz a um *software*.

Uma de suas características é a de que é necessário um cadastro de vocabulário com as palavras que o *Sphinx4* irá suportar na aplicação em que for acoplado. Isso se deve a questões de performance, já que seu objetivo não é o de realizar um reconhecimento de voz, como por exemplo, aplicações que identificam a voz de uma pessoa utilizando inteligência artificial (SOURCEFORGE, 2008). A Figura 7 mostra a arquitetura do Sphinx4 e seus componentes.

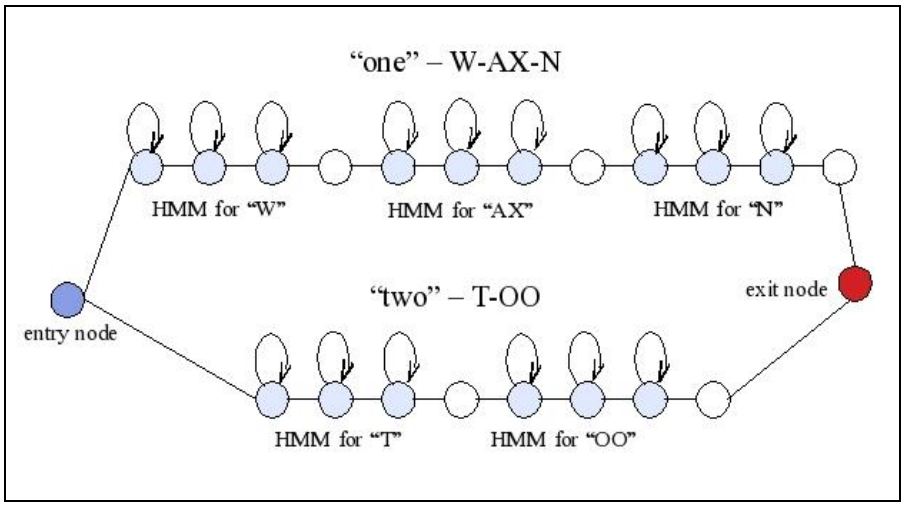


Fonte: Sourceforge (2008).

Figura 7 – Arquitetura e principais componentes

O mapa arquitetônico do Sphinx4 (Figura 7) mostra principalmente seu ciclo de utilização. A aplicação fornece informações sobre como o reconhecimento irá ocorrer, enviando como entrada uma fala que será analisada pelo decodificador. Neste momento o analisador linguístico, que possui um dicionário de palavras conhecidas, gera um grafo para resolver a fala e informar à aplicação sobre o sucesso ou falha no reconhecimento.

Segundo Sourceforge (2008), o reconhecedor de fala do Sphinx4 é baseado no modelo estatístico *Hidden* Markov *Models* (HMM). A característica de modelos baseados em HMM é que os fonemas são representados por modelos estatísticos que representam os dados deste fonema, que nada mais são que informações que diferenciam um fonema de outro. Como cada pessoa possui uma faixa de frequência de voz ligeiramente diferente, além de características próprias em sua fala, faz-se necessário utilizar um método estatístico para tratar estas diferenças. A Figura 8 exemplifica um grafo utilizado para reconhecer uma fala.



Fonte: Sourceforge (2008).

Figura 8 – Grafo de reconhecimento

Conforme o grafo (Figura 8), são analisadas as palavras "*one*" e "*two*" cuja representação fonética que as compõe são "WAXN" e "TOO" respectivamente. Como pré-condição para que qualquer palavra seja reconhecida no Sphinx4, estas palavras devem estar em um dicionário para que possam ser analisadas.

Tomando como exemplo apenas a análise da palavra "*one*", onde o primeiro fonema reconhecido é o "W", em seguida o "AX" e por último o "N", caso esta sequência de reconhecimento de fonemas for compatível com uma previamente cadastrada no dicionário utilizado, o Sphinx4 irá reconhecer como a palavra "*one*".

2.3.3 Opera Voice

Segundo Opera Developer (2009), o *Opera Voice* é uma extensão da versão *Windows* do *Opera Browser*. Suporta leitura de textos selecionados em páginas e navegação por comando de voz. Sua leitura de texto de página funciona de maneira bem simples. Basta selecionar um trecho de um texto e selecionar a opção de leitura, que o *browser* irá falar o texto selecionado.

É possível também utilizar um documento *eXtensible HyperText Markup Language* (XHTML) combinado com o *Voice eXtensible Markup Language* (VoiceXML) para fazer com que a leitura de uma página seja configurável por quem a fornece.

Já a navegação por comando de voz funciona de maneira semelhante ao funcionamento do *Sphinx4*. Existe um dicionário cadastrado com os comandos que o *browser* suporta, onde estes comandos deverão ser falados. Caso o comando seja reconhecido, o

browser executa a operação desejada, caso contrário ele informa que não entendeu o comando (OPERA DEVELOPER, 2009). O Quadro 14 fornece um exemplo de um documento XHTML que o *Opera Voice* utiliza para realizar uma síntese de seu conteúdo.

```
<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML+Voice 1.0/EN" "xhtml+voice.dtd">
<html xmlns=http://www.w3.org/1999/xhtml xmlns:ev="http://www.w3.org/2001/xml-
events">
 <head>
    <title>Example 2: Make it talk</title>
    <form xmlns="http://www.w3.org/2001/vxml" id="talker">
      <block>
         I always believed that
         <value expr="document.getElementById('phrase').value"/>
      </block>
    </form>
  </head>
  <body>
     <h1>Make Opera say what you want</h1>
     <label>Text: <input type="text" id="phrase" value="you are the greatest."
size="30"/>
     </label>
     <button ev:event="click" ev:handler="#talker">Say it now!</button>
  </body>
</html>
```

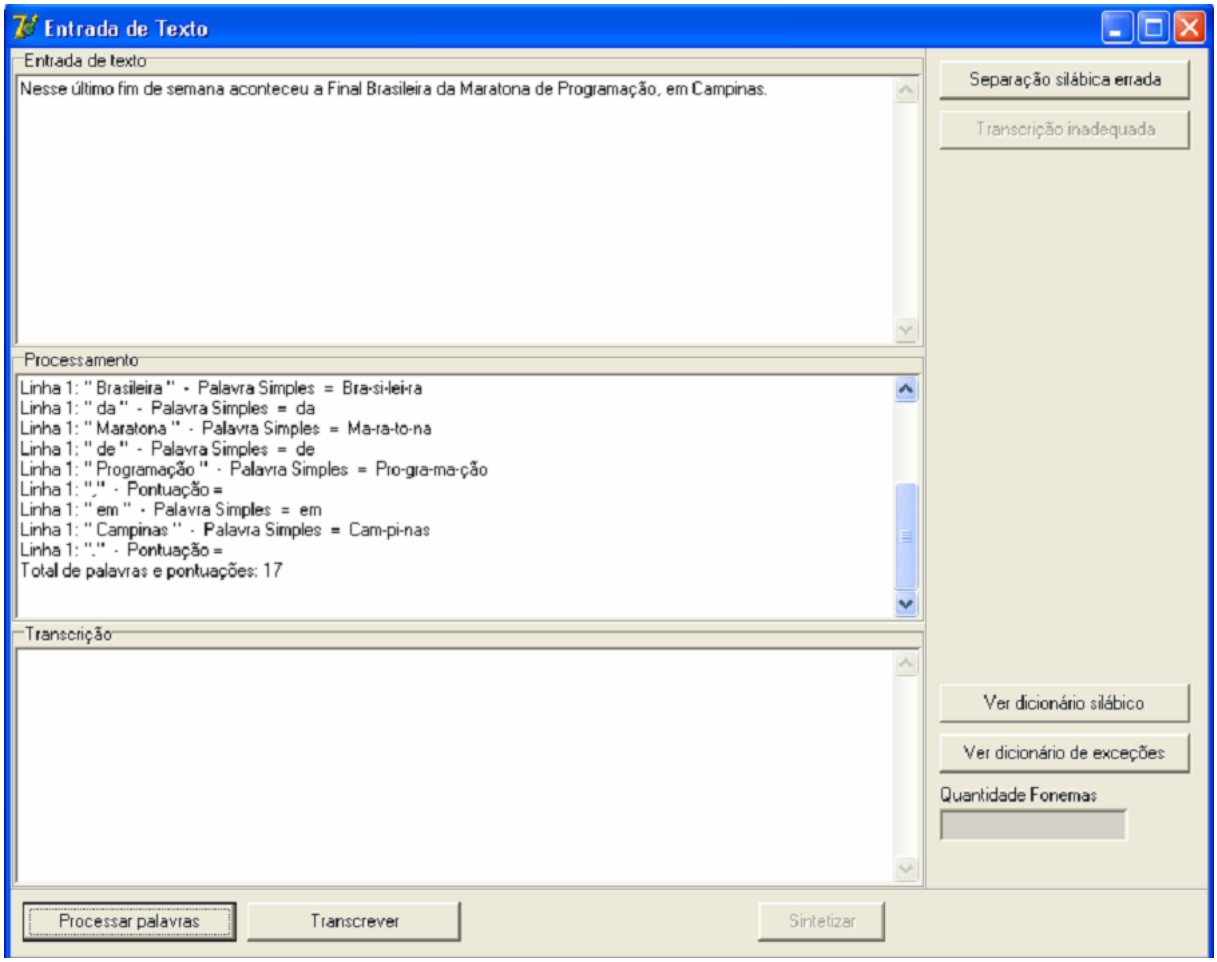
Fonte: Opera Deveolper (2009).

Quadro 14 – Exemplo de um XHML suportado pelo VoiceXML do Opera

2.3.4 Processamento escrito em linguagem natural para um sistema conversor texto-fala

O protótipo desenvolvido por Oechsler (2009) teve como objetivo converter uma entrada em linguagem natural para uma linguagem com gramática formal. Neste trabalho foi desenvolvido um analisador lingüístico que faz uma análise das palavras do texto de entrada, definindo a pronúncia de cada palavra. Oechsler (2009) também desenvolveu um analisador prosódico que determina características como entonação, duração e intensidade sonora de uma sentença.

Como sintetizador acústico, é utilizado o MBROLA em conjunto com um banco de dados de fonemas em português. A Figura 9 mostra a tela do protótipo de Oechsler.



Fonte: Oechsler (2009, p. 56).

Figura 9 – Tela do protótipo de Oechsler (2009)

Na caixa de texto `Entrada de texto` (Figura 9) é definido o texto que será sintetizado. Pressionando o botão `Processar palavras`, o texto original é processado e na caixa de texto `Processamento` são mostradas as informações do processamento do texto original, separando as palavras silabicamente. Após esta etapa, ao clicar no botão `Transcrever`, a caixa de texto `Transcrição` apresenta os fonemas com seu tempo de informação prosódica. Por fim, ao clicar no botão `Sintetizar`, o texto é sintetizado.

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as seguintes seções: especificação dos requisitos do software desenvolvido; especificação da linguagem com a inclusão do comando `fala;` lista de fonemas e formato de entrada do sintetizador; diagramas de casos de uso; classes, atividades e sequência; implementação do sintetizador e integração do Tangram com o sintetizador.

Ainda, os resultados obtidos pela síntese de voz e as dificuldades encontradas também são relatados.

3.1 REQUISITOS DO SOFTWARE A SER DESENVOLVIDO

Os requisitos funcionais do sistema são:

a) a Backus-Naur *Form* (BNF) do LTD deve suportar um comando para especificar a fala;
b) o sistema deve gerar uma definição textual da frase que será passada ao sintetizador;
c) o sistema deve permitir a um usuário ou a um sistema externo atribuir uma entrada no formato JSML ao sintetizador.

Os requisitos não-funcionais do sistema são:

a) utilizar a linguagem *Java* versão 1.6.x;
b) executar no sistema operacional *Windows XP*;
c) utilizar uma base de dados de fonemas em português compatível com o MBROLA;
d) suportar a especificação JSML versão 0.6 como formato de entrada para o sintetizador de voz.

3.2 ESPECIFICAÇÃO

Neste capítulo são apresentadas a especificação da linguagem, a lista de fonemas suportados e o formato de entrada que servirá de insumo pelo sintetizador MBROLA para gerar a voz, os diagramas de casos de uso, classes, atividades e de sequência da *Unified Modeling Language* (UML).

3.2.1 Extensão da linguagem LTD

A seguir é apresentada a definição da linguagem LTD utilizada por Knihs (2008) e sua extensão para o suporte a fala.

3.2.1.1 Definição da linguagem

O Quadro 15 apresenta um exemplo de um programa na linguagem LTD onde contém um método para criar um modelo e outro método para executar uma ação.

```
metodo CRIA
      cria p1(-613,1165,-7000) gira(90.0) cor(prata)
      cria p2(382,686,-7000) gira(180.0) cor(prata)
      cria p3(-602,-270,-7000) gira(270.0) cor(marrom)
      cria p4(-341,64,-7000) gira(270.0) cor(vermelho)
      cria p5(-229,182,-7000) gira(270.0) cor(branco)
      cria p6(-271,-248,-7000) gira(45.0) cor(marrom)
      cria p7(-239,566,-7000) gira(90.0) cor(branco)
fim;
metodo vai
repita 45 vezes inicio
    p1.gira(-1) no ponto(9)
    p2.gira(-1) no ponto(12)
    pisca(10)
fim
fim;
```

Quadro 15 – Exemplo de um programa em LTD

O método `CRIA` (Quadro 15) é responsável por criar as peças que compõem o modelo e posicioná-las na área de visualização. O método `vai` executa um bloco de comandos que estão em um laço de execução que faz com que as peças `p1` e `p2` girem causando uma animação no modelo.

A Figura 10 demonstra a execução deste programa.

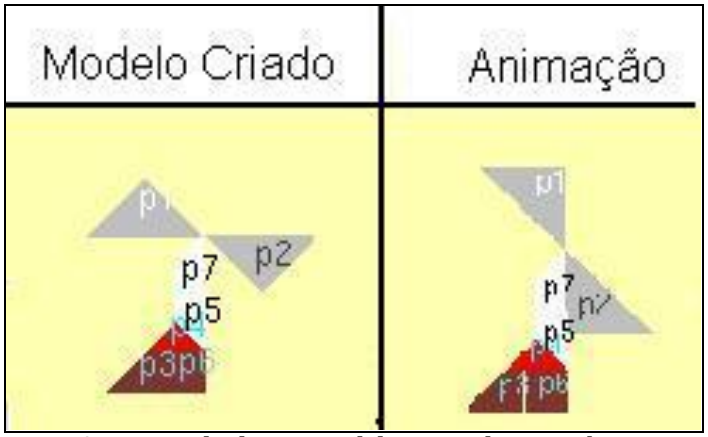


Figura 10 - Exemplo de um modelo animado no ambiente LTD

Ainda, a linguagem LTD possui outros comandos definido por Knihs (2008), como `cria`, `gira`, `pisca`, `move`, `faça`, `espelha` e `repita`. Nesta nova versão do LTD, foram incluídos os comandos `fala`, `enquanto fala` e `espera fala`.

O comando `fala` possui duas variantes que são: fala exclusiva e fala sobreposta. A fala exclusiva é quando um comando `fala` começa a ser executado e nenhum outro comando `fala` será executado simultaneamente. A fala sobreposta permite que mais de um comando `fala` seja executado simultaneamente. O comando `fala` possui dois parâmetros. O primeiro parâmetro, que é obrigatório, é o documento JSML que contém o texto da fala que será sintetizada. O outro parâmetro é opcional e indica se a fala será sobreposta e caso seja omitido, a fala será exclusiva. O Quadro 16 descreve três métodos na linguagem LTD que exemplificam a utilização do comando `fala`.

```
metodo conversa
      fala('Jose.jsml')
      fala('Artur.jsml')
fim;
metodo tagarela
      fala('Chaves.jsml'  sobreposto)
      fala('Madruga.jsml'  sobreposto)
fim;
metodo discussao
      fala('Zeca.jsml' )
      fala('Joselito.jsml'  sobreposto)
      fala('Adao.jsml'  sobreposto)
fim;
```

Quadro 16 – Exemplo de utilização do comando `fala`

O método `conversa` possui dois comandos de fala. Como ambos são comandos de fala exclusiva, isso implica que primeiro “Jose” irá falar e somente quando “Jose” terminar sua fala é que “Artur” terá a vez de falar.

O método `tagarela` possui dois comandos de fala, porém ambos são comandos de fala

sobreposta. Isso significa que assim que “Chaves” começar a falar, “Madruga” irá falar ao mesmo tempo que “Chaves”, criando assim uma sobreposição de falas.

O método `discussao` possui três comandos de fala. O primeiro de fala exclusiva e os seguintes de fala sobreposta. A fala de “Zeca” é exclusiva, então somente quando “Zeca” terminar sua fala é que “Joselito” e “Adao”, que possuem falas sobrepostas, poderão começar a falar simultaneamente.

Para sincronizar a fala com a animação, foi criado o comando `enquanto fala` que possui um bloco de comandos que são executados repetidamente enquanto a fala não chega ao fim. O Quadro 17 mostra um exemplo da utilização do comando `enquanto fala`.

```
fala('Wally.jsml' )
enquanto fala inicio
      p1.gira(45)
      p1.gira(45)
      p1.move(-190, -165, 0)
      p2.move(340, -355, 0)
fim
```

Quadro 17 – Exemplo de utilização do comando `enquanto fala`

O comando `fala` (Quadro 17) inicia a fala de “Wally” e o comando `enquanto fala` abre o bloco de comando para animações de um modelo. Os comandos deste bloco são executados repetidamente até que a fala de “Wally” termine.

Para que a execução de comandos de animação necessite do término de uma fala, foi disponibilizado o comando `espera fala`. O Quadro 18 exemplifica a utilização do comando `espera fala`.

```
fala('Uilame.jsml' )
espera fala
p1.gira(45)
p1.gira(45)
p1.move(-190, -165, 0)
p2.move(340, -355, 0)
```

Quadro 18 – Exemplo de utilização do comando `espera fala`

O comando `fala` (Quadro 18) inicia a fala de “Uilame” e o comando `espera fala` irá parar a execução do programa aguardando a fala de “Uilame” terminar sua execução para só então prosseguir na execução dos comandos seguintes.

3.2.1.2 Definições regulares

As definições regulares especificadas por Knihs (2008) na ferramenta Gerador de Analisadores Léxicos e Sintáticos (GALS) estão representadas no Quadro 19.

```
id: [a-zA-Z]+[a-zA-Z0-9_ç]*
ws:[\ \t\s\r\n]+
comentario: /[/][^\n]+
multilinha:(/\*)([^\*]|\*[^/])*(\*/)
```

Fonte: Knihs (2008, p. 34).

Quadro 19 – Definições regulares

Segundo Knihs (2008, p. 34), a palavra `id` é um identificador para uma definição regular. Após o operador dois pontos (“:”) é especificada a definição regular. A seqüência de caracteres `[a-zA-Z]` define que o `id` deve começar com uma letra minúscula ou maiúscula. O operador mais (“+“) significa que após a primeira letra podem ser concatenados, ao `id`, outros caracteres definidos por `[a-zA-Z0-9_ç]`. Esta definição permite, além de letras minúsculas e maiúsculas, números e os caracteres *underline* (“_“) e cê-cedilha (“ç”). O operador asterisco (“*“) significa nenhum, um ou vários caracteres.

3.2.1.3 *Tokens*

Os *tokens* definidos por Knihs (2008) estão relacionados no Quadro 20.

```
identificador:{id}+
 : {ws}+
numero: [\+\-]?[0-9]+[\.0-9]*
modelo = identificador: "modelo"
metodo = identificador: "metodo"
mundo = identificador: "mundo"
CRIA = identificador: "CRIA"
VIVA = identificador: "VIVA"
APAGA = identificador: "APAGA"
TERMINA = identificador: "TERMINA"
gira = identificador: "gira"
cor = identificador: "cor"
espelha = identificador: "espelha"
repita = identificador: "repita"
em = identificador: "em"
paralelo = identificador: "paralelo"
faca = identificador: "faça"
depois = identificador: "depois"
de = identificador: "de"
no = identificador: "no"
ponto = identificador: "ponto"
inicio = identificador : "inicio"
fim = identificador: "fim"
cria = identificador: "cria"
como = identificador: "como"
move = identificador: "move"
piscar = identificador: "pisca"
vezes = identificador: "vezes"
Peca1 = identificador: "p1"
Peca2 = identificador: "p2"
Peca3 = identificador: "p3"
Peca4 = identificador: "p4"
Peca5 = identificador: "p5"
Peca6 = identificador: "p6"
Peca7 = identificador: "p7"
amarelo = identificador: "amarelo"
azul = identificador: "azul"
azulMarinho = identificador : "azulMarinho"
azulPiscina = identificador: "azulPiscina"
branco = identificador: "branco"
cinza = identificador: "cinza"
marrom = identificador: "marrom"
oliva = identificador: "oliva"
prata = identificador: "prata"
preto = identificador: "preto"
rosa = identificador: "rosa"
verde = identificador: "verde"
verdePiscina = identificador: "verdePiscina"
verdeLima = identificador: "verdeLima"
vermelho = identificador: "vermelho"
violeta = identificador: "violeta"
";" : ;
"(" : \(
")" : \)
"," : ,
"." : .
:{comentario}
:{multilinha}
```

Fonte: Knihs (2008, p. 35).

Quadro 20 – *Tokens*

A inclusão dos *tokens* que compõem o comando `fala` estão descritos no Quadro 21.

```
jsml: '(.+)'
sobreposto = identificador: "sobreposto"
fala = identificador : "fala"
enquanto = identificador : "enquanto"
espera = identificador : "espera"
```

Quadro 21 – Novos *tokens*

O *token* `jsml` (Quadro 21) é uma sequência de caracteres entre apóstrofos que representa o documento JSML que será utilizado pelo sintetizador. O *token* `sobreposto` é uma palavra reservada utilizada como parâmetro opcional do comando fala. O *token* `fala` que também é uma palavra reservada, define o comando de fala. O *token* `enquanto` define o comando `enquanto fala`. Por fim, o *token* `espera` define o comando `espera fala`.

3.2.1.4 Gramática

As produções da gramática seguem a notação utilizada pela ferramenta GALS que, segundo Gesser (2003), são representadas entre os símbolos “<” e “>” seguindo o formato da notação BNF. As ações semânticas são especificadas pelo caractere “#” seguido de um número inteiro não negativo. O caractere “`î`”, um símbolo terminal, representa a palavra vazia (símbolo *épsilon*).

A gramática especificada por Knihs (2008) encontra-se no Quadro 22.

```
<codigo> ::= <modelo> | <mundo> | <metodo_cria> #2 ;
<modelo> ::= modelo  #0 <figura> <Lista_de_metodos> fim "." #27 ;
<Lista_de_metodos> ::= <metodo_cria> <metodos_outros> ;
<metodos_outros> ::= <metodo> <metodos_outros> | î ;
<metodo_cria> ::= metodo CRIA #4 #12 <bloco> fim ";" #26 #36 ;
<metodo> ::= metodo <nome_Metodo> #12 <bloco> fim ";" #26 ;
<bloco> ::= <comando> <bloco_2> ;
<bloco_2> ::= <bloco> | î ;
<comando> ::= <comando_cria> | <comandos_em_ids> |  <comando_piscar> #22
#14 | <comando_repete> | <comando_faca> ;
<comandos_em_ids> ::= <id> "." <comando_de_id> ;
<comando_de_id>  ::=  <comando_move>  #14  |  <comando_gira>  #14  |
<comando_cor> #14 | <comando_espelha> #14 ;
<comando_cria> ::= cria <peca> "(" <X> "," <Y> "," <Z>  ")" #13 #14
<cria_extra> ;
<cria_extra> ::= <extra> | î ;
<extra>  ::=  <comando_cor>  #14  <cria_extra>  |  <comando_gira>  #14
<cria_extra> | <comando_espelha> #14 <cria_extra> ;
<id> ::= <figura> #15 | <peca> #16 ;
<comando_move> ::= move  "(" <X>  "," <Y>  "," <Z>  ")" #17 ;
<comando_gira> ::= gira "(" <X> ")" <gira_extra> #18 ;
<gira_extra> ::= no ponto "(" <Y> ")" | î ;
<comando_cor> ::= cor "(" <cor> ")" #20 ;
<comando_espelha> ::= espelha #21 ;
<comando_piscar> ::= piscar "(" <X> ")" ;
<comando_repete> ::= repita <X> vezes #23 inicio <bloco> fim #24 ;
<comando_faca> ::= faca identificador #25 #14 ;
<figura> ::= identificador #3 ;
<nome_Metodo> ::= identificador #4 ;
<modelo_id> ::= identificador #5 ;
<nome_do_mundo> ::= identificador #6 ;
<peca> ::= Peca1 #7 |Peca2 #7 |Peca3 #7 |Peca4 #7 |Peca5 #7 |Peca6 #7
|Peca7 #7 ;
<cor> ::= amarelo #8 | azul #8 | azulMarinho #8 | azulPiscina #8 | branco
#8  | cinza #8  | marrom #8  | oliva #8 | prata #8 | preto #8 | rosa #8 |
verde #8 | verdePiscina #8 | verdeLima #8 |vermelho #8 | violeta #8 ;
<X> ::= numero #9 ;
<Y> ::= numero #10 ;
<Z> ::= numero #11 ;
<mundo> ::= mundo  #1 <nome_do_mundo> <bloco_do_mundo> fim"." ;
<bloco_do_mundo> ::= <comando_mundo> <bloco_do_mundo_2> ;
<bloco_do_mundo_2> ::= <bloco_do_mundo> | î ;
<comando_mundo>  ::=  <comando_cria_como>  |  <comando_repete_mundo>  |  #33
<comando_faca_mundo> #14 | <comando_piscar> #22 #14 ;
<comando_cria_como> ::= cria <modelo_id> como <figura> #28 #14 "(" <X>
"," <Y>  "," <Z>  ")" #29 #14 ;
<comando_repete_mundo> ::=  repita <X> vezes #23 inicio <bloco_do_mundo>
fim #24 ;
<comando_faca_mundo> ::=  faca <modelo_id> "." <metodo_do_id> ;
<metodo_do_id>  ::=  <comando_viva>  #30  |  TERMINA  #31  |  APAGA  #32  |
<nome_Metodo> <em_paralelo> #34 ;
<comando_viva> ::= VIVA "(" <nome_Metodo> ")" ;
<em_paralelo> ::= em paralelo #35;
```

Fonte: Knihs (2008, p. 36).

Quadro 22 – Gramática especificada por Knihs (2008)

O Quadro 23 descreve quais operadores foram envolvidos na inclusão do comando fala na linguagem LTD.

```
<comando> ::= <comando_cria> | <comandos_em_ids> |  <comando_piscar> #22
#14 | <comando_repete> | <comando_faca>
|<comando_fala>|<comando_enquanto_fala> | <comando_espera_fala>;
<comando_fala> ::= fala "(" jsml #37 <comando_fala_sobreposto> ")" #39
#14;
<comando_fala_sobreposto> ::= sobreposto #38 | î;
<comando_enquanto_fala> ::= enquanto  fala #40 inicio <bloco> fim #41 #14;
<comando_espera_fala> ::= espera fala #42 #14;
```

Quadro 23 - Alteração da gramática para inclusão do comando de fala

O não-terminal `<comando_fala>` especifica a sintaxe do comando fala. Começa com a palavra reservada `fala` seguida de parênteses (“(“ e “)”). Entre os parênteses vão os parâmetros. O primeiro parâmetro é o documento JSML representado pelo *token* `jsml`, sendo este obrigatório. Em seguida vem o parâmetro opcional descrito pelo não-terminal `<comando_fala_sobreposto>` que especifica duas possibilidades: trazer o *token* `sobreposto` ou o terminal “`î`“ significando que não possui nenhum parâmetro no comando. O não-terminal `<comando_enquanto_fala>` especifica a sintaxe do comando `enquanto fala` que possui um bloco de comandos que são executados repetidamente enquanto existir alguma fala sendo executada. O não-terminal `<comando_espera_fala>` especifica a sintaxe do comando `espera fala` que interrompe a execução do programa até que não haja nenhuma fala esteja sendo executada. As ações semânticas do Quadro 22 e Quadro 23 estão descritas no Apêndice A.

3.2.2 Lista de fonemas e formato de entrada do sintetizador

O MBROLA recebe como entrada um banco de voz em um idioma específico e um arquivo textual com os fonemas que descrevem uma fala. O banco de voz possui os fonemas relativos ao idioma e sua representação sonora. O arquivo textual fornece os fonemas com suas respectivas configurações de duração e informações prosódicas que o sintetizador acústico em conjunto com o banco de voz irá utilizar para gerar o som resultante.

O Quadro 24 lista todos os fonemas suportados pela língua portuguesa, exemplo de uma palavra com o fonema e a transcrição para o MBROLA.

Fonema	Exemplo	Transcrição
b	barco	“ba r2 ku
k	com	“k om
d	doce	“do se
g	grande	“gr am de
p	pai	“pai
t	taco	“ta ko
f	fácil	“fa si u
v	vinho	“vi nh o
j	jato	“já to
s	sala	“sa la
s2	casca	“ka s2 ka
x	chave	“xá ve
z	zebra	“ze br a
m	mesmo	“me s2 mo
n	nunca	“n um ka
nh	galinha	ga” li nh a
l	lanche	“l am xe
lh	alho	“a lh o
r	puro	“pu ro
r2	arpa	“a r2 pa
rr	torre	“to rr e
a	vale	“va le
@	não	n @ o
am	campanha	k am pa nh a
e	pêra	“pe ra
ee	quero	“k ee ro
em	quente	“k em te
i	pico	“pi ko
im	brinco	“br im ko
o	tolo	“to lo
oo	bola	“b oo la
om	ombro	“om bro
u	duro	“du ro
um	algum	aw”g um
y	mais	“may s2
w	mau	“maw
_	silêncio	

Quadro 24 – Fonemas

O formato de entrada para o MBROLA sintetizar um texto possui os fonemas seguidos do tempo de duração em milissegundos e uma informação prosódica. O Quadro 25 mostra um exemplo da transcrição de fonemas para o formato de entrada do MBROLA.

```
t 105 100 100.0
o 105 100 100.49480791850904
r2 105 100 100.95885107720841
n 120 100 101.36327752004667
a 120 100 101.6829419696158
d 105 100 101.89796923871117
o 105 100 101.9949899732081
```

Quadro 25 – Transcrição da palavra “tornado”

Analisando a primeira linha do Quadro 25 tem-se o fonema “t” que vem seguido de um espaço em branco e o número 105 significando que o fonema irá ter 105 milissegundos de duração durante a síntese. Os números a seguir na mesma linha são informações prosódicas do fonema. O 100 indica que 100% destes 105 milissegundos a freqüência será de 100Hz. Diferentemente, o fonema “a” tem sua duração de 120 milissegundos e 100% deste tempo com a frequência a 101.6829419696158Hz.

Segundo Oechsler (2009, p38), é recomendado que o tempo de pronúncia de um fonema não ultrapasse 150 milissegundos assim resultando em uma fala natural. Da mesma forma, Oechsler utiliza uma variação da frequência de 80Hz a 100Hz.

Para representar a breve pausa entre a pronúncia das palavras, o MBROLA fornece um fonema que representa silêncio, o “_” (*underscore*), seguido do tempo que este silêncio deve permanecer, assim como nos demais fonemas, com a diferença de não possuir a informação prosódica (Quadro 26).

```
_ 300
```

Quadro 26 – Representação de silêncio

3.2.3 Diagramas de casos de uso do protótipo

O diagrama de casos de uso da Figura 11 representa uma extensão do diagrama de casos de uso de Knihs (2008, p. 39), onde o caso de uso `Comandos LTD` abrange todos os comandos especificados por Knihs (2008) e o caso Cria `comando cria` o comando no editor textual. A descrição dos casos de uso `fala`, `enquanto fala` e `espera fala` são:

a) `fala`: permite informar um documento JSML com o conteúdo que será sintetizado. Esta fala pode ser do tipo Sobreposta, significando que se a fala for do tipo `sobreposta`, ela poderá permitir que outro comando `fala` seja executado simultaneamente;

b) `enquanto fala`: possui um bloco de comandos que são executados repetidamente

enquanto existe uma fala sendo executada;

c) `espera fala`: interrompe a execução do programa enquanto houver alguma fala sendo executada.

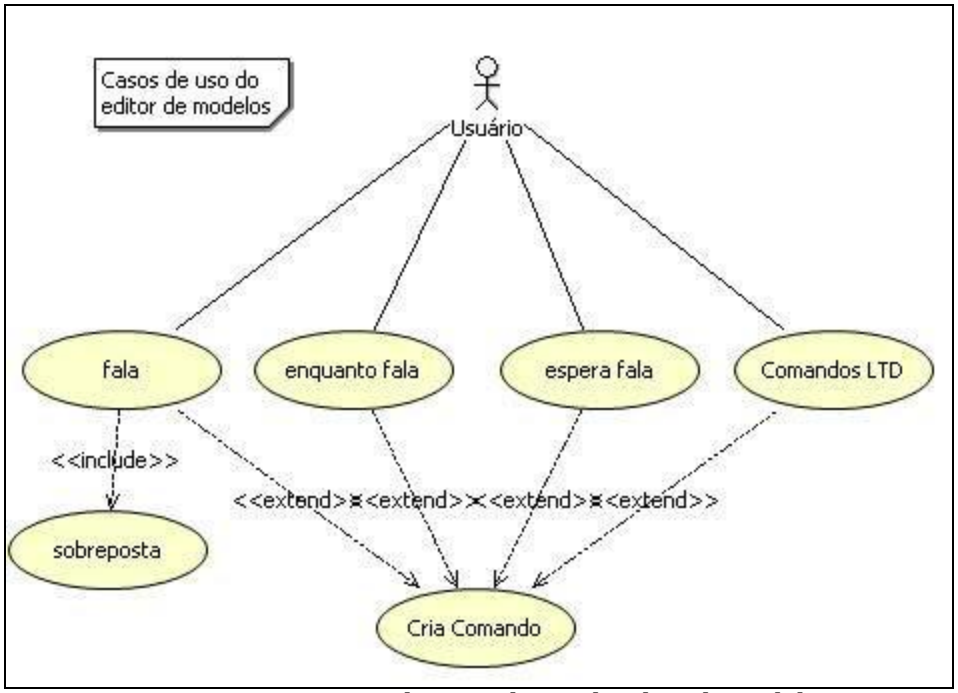


Figura 11 – Diagrama de casos de uso do editor de modelos

Os casos de usos definidos para o sintetizador são: `Informa documento JSML` e `Sintetiza texto` (Figura 12).

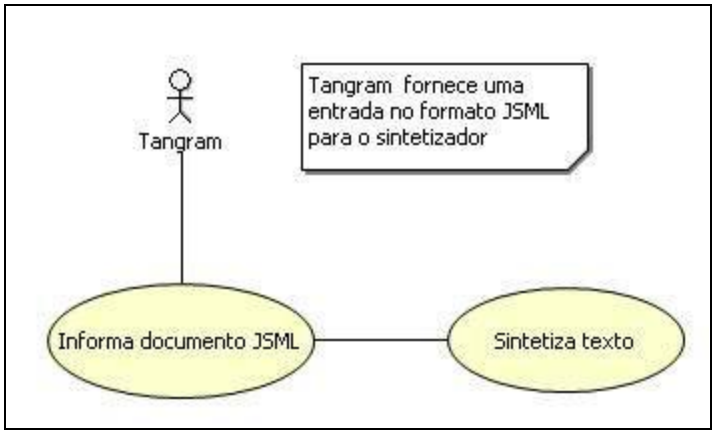


Figura 12 – Diagrama de casos de uso do sintetizador

O diagrama de casos de uso da Figura 12 mostra como um aplicativo externo, neste caso o Tangram, utiliza uma API onde informa-se um documento JSML com a especificação do texto a ser sintetizado, e em seguida a síntese deste texto.

Os casos apresentados na Figura 12 são:

a) `Informa documento JSML`: disponibiliza uma API que permite informar um documento JSML com o texto a ser sintetizado;

b) `Sintetiza texto`: processa os dados extraídos do documento JSML para gerar

um arquivo de áudio com a fala a partir do texto contido neste documento.

3.2.4 Diagrama de classes do protótipo

O diagrama da Figura 13 mostra como são construídos os elementos JSML e seus atributos. Cada elemento é uma implementação da *interface* `ISynthElement`, porém o *parser* não conhece a implementação de cada elemento, apenas fornece o seu tipo. Para deixar transparente este processo, foi utilizado o *design pattern Factory Method*, que tem como objetivo fornecer uma implementação de uma *interface* sem fornecer sua implementação concreta.

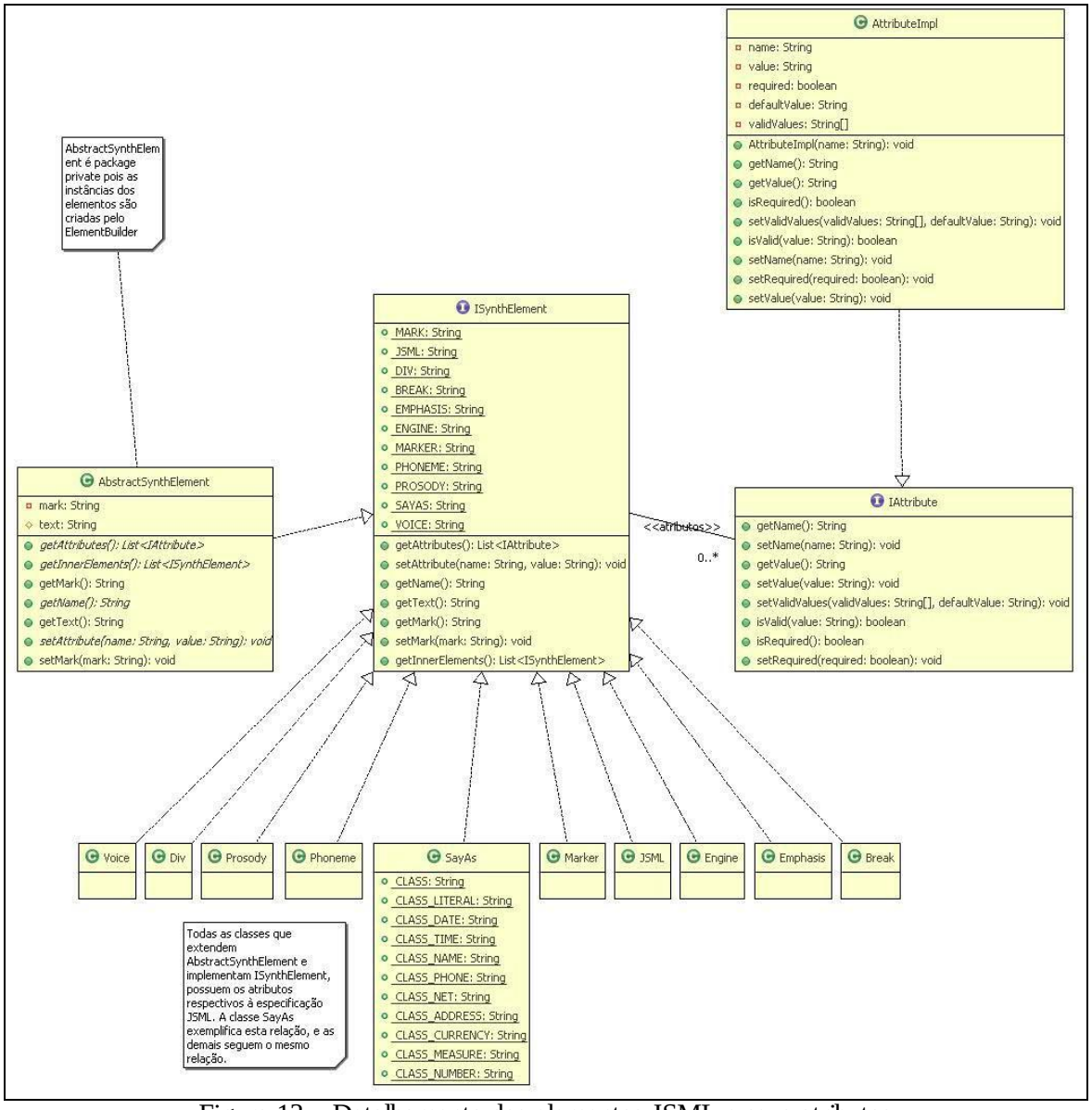


Figura 13 – Detalhamento dos elementos JSML e seus atributos

As classes envolvidas no processo de conversão dos elementos JSML são apresentadas na Figura 14.

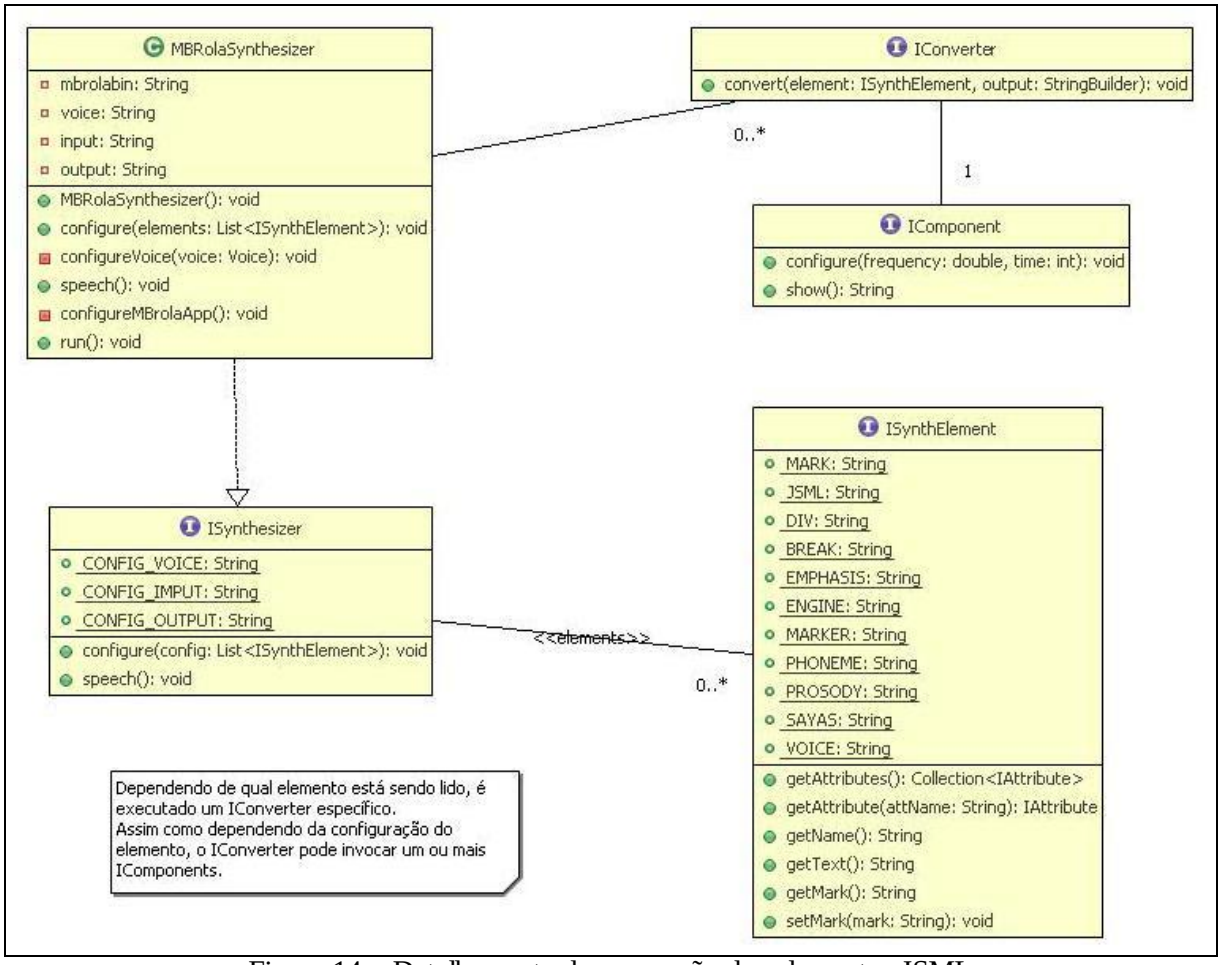


Figura 14 – Detalhamento da conversão dos elementos JSML

Um elemento JSML como o `sayas`, por possuir várias classes como configuração, pode assumir diferentes formas para tratar o texto, além de poder passar por mais de uma etapa de transformação do texto. Por exemplo, um elemento `sayas` de classe `time`, primeiramente precisa identificar o que é hora, minuto e segundo. Em seguida transformar de formato numérico para extenso. Por fim, gerar o formato intermediário do sintetizador a partir do texto por extenso desta hora. A cada etapa desta transformação a informação é passada para uma classe diferente tratar.

Para a implementação deste mecanismo é utilizado o *design pattern Chain of Responsibility*. Este padrão tem como fazer com que cada objeto de uma cadeia de objetos receptores receba uma mensagem ou um objeto para ser tratado. Caso não seja possível realizar este tratamento ou que ainda falte algum tratamento adicional, esta mensagem ou objeto será enviado para o próximo objeto receptor tratar, e assim por diante até que não tenha mais nada pendente.

A Figura 15 especifica o mecanismo para fazer com que o Tangram execute o sintetizador respeitando a condição da execução do sintetizador ser assíncrona ou síncrona.

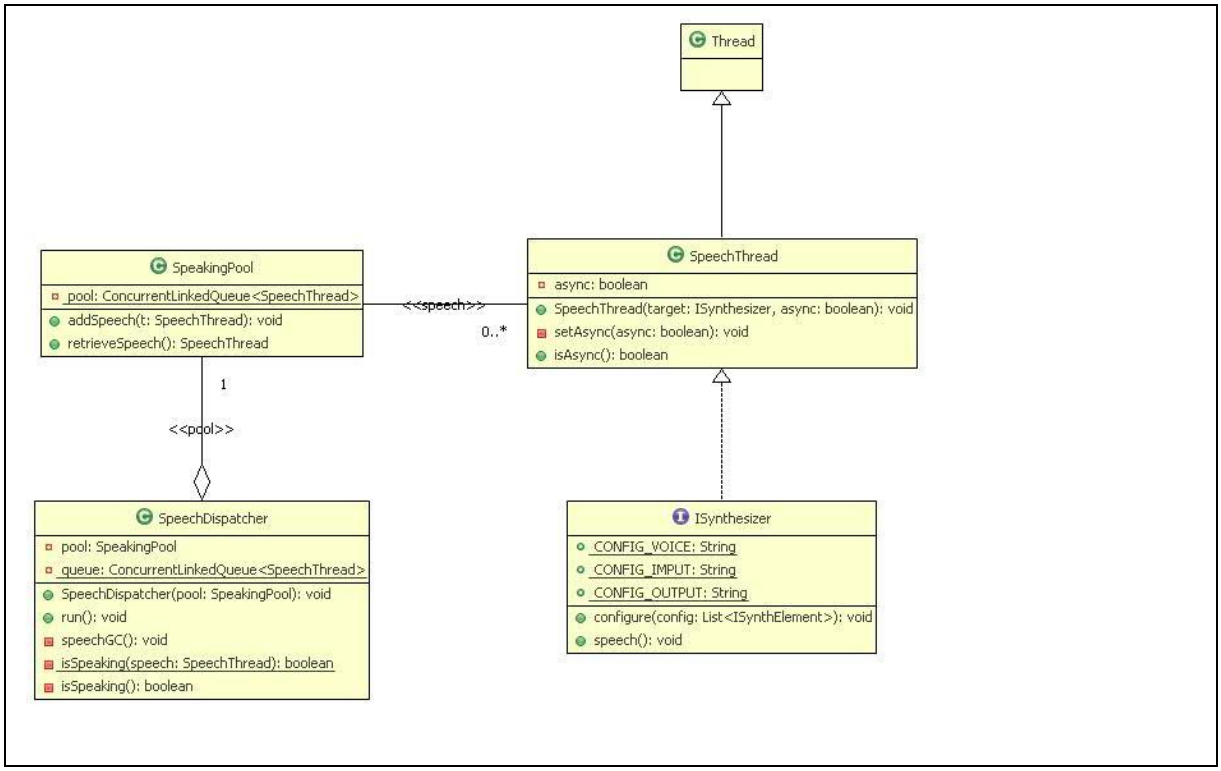


Figura 15 – *Pool* de falas no Tangram

A classe `SpeechThread` é uma classe `Thread` e recebe um objeto `ISynthesizer`, que representa o sintetizador, para ser executado. A classe `SpeakingPool` recebe várias `SpeakingThreads` e as armazena para que possam ser executadas conforme a classe `SpeechDispatcher` solicitar. Já a classe `SpeechDispatcher` é a responsável por executar as *threads* que ficam armazenadas no *pool*.

3.2.5 Diagrama de atividades do protótipo

A visão macro do processo de síntese está detalhado na Figura 16. Nela é possível visualizar a responsabilidade do *parser* do documento JSML, do sintetizador e como o Tangram comunica-se com o *parser*.

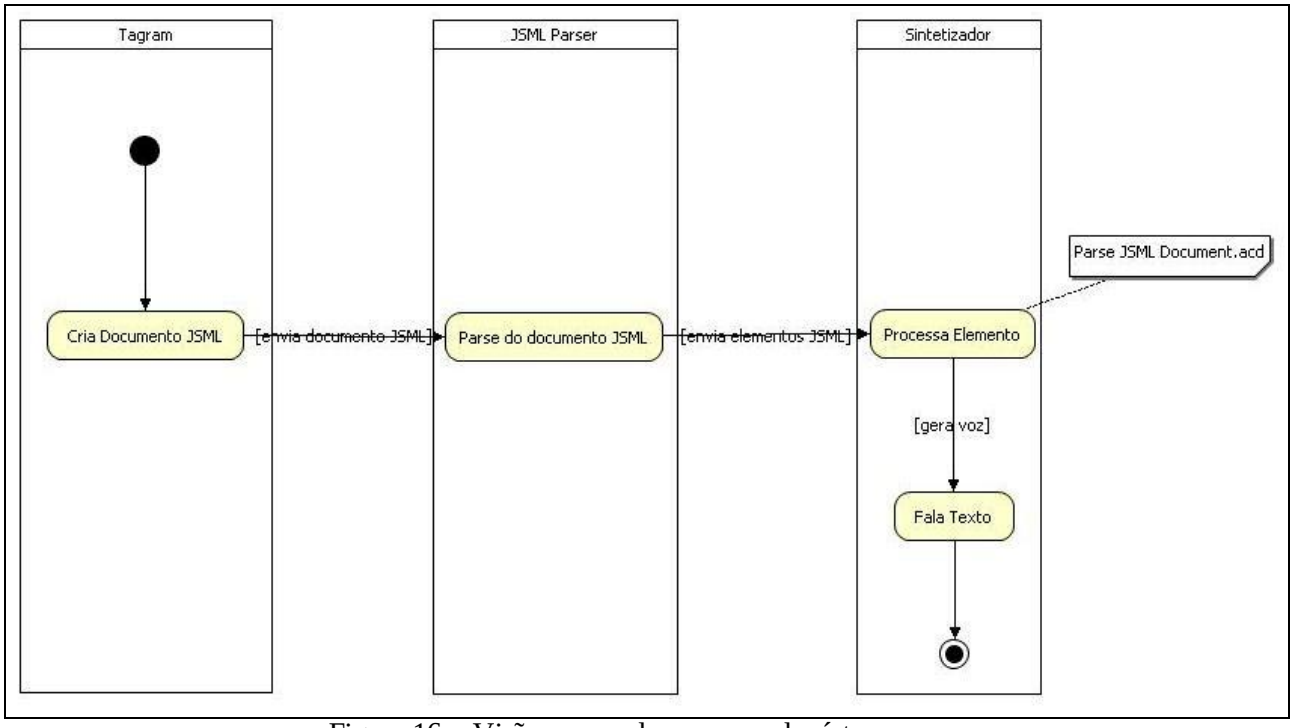


Figura 16 – Visão macro do processo de síntese

A atividade de *parsing* do documento consiste em ler o documento JSML e popular uma estrutura de objetos que represente este documento.

No sintetizador, as atividades são de receber estes objetos para realizar sua configuração e em seguida gerar a fala deste texto.

Conforme a Figura 17, a atividade `Configure Element` é responsável por criar as instâncias dos elementos JSML. Estes elementos são fornecidos pelo *parser* que faz a análise sintática do documento JSML que está no formato XML.

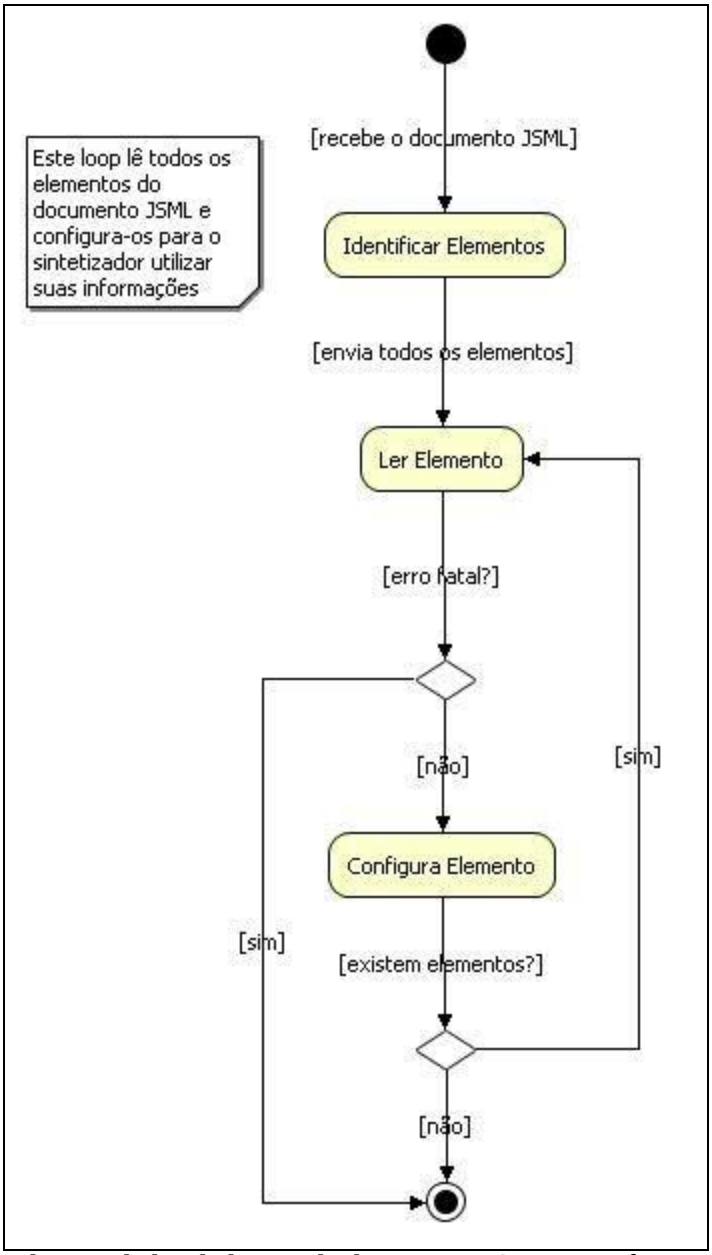


Figura 17 – Descrição das atividades de leitura do documento JSML e configuração de seus elementos

Apesar de que possam haver elementos ou valores de atributos inválidos, ou seja, não suportados pela JSML, não acarretam na falha da leitura ou configuração do sintetizador. Se o elemento for inválido, ele será apenas suprimido, ou se o elemento possuir alguma inconsistência em algum atributo, este atributo será suprimido caso não seja obrigatório, ou senão o atributo irá receber o valor padrão. Entretanto, se o XML possuir um erro de sintaxe detectado pelo *parser*, este erro é caracterizado como um erro fatal e todo o processo é abortado.

3.2.6 Diagrama de sequência do protótipo

O diagrama de sequência representado pela Figura 18 exemplifica o fluxo de utilização da API exposta do sintetizador pelo Tangram.

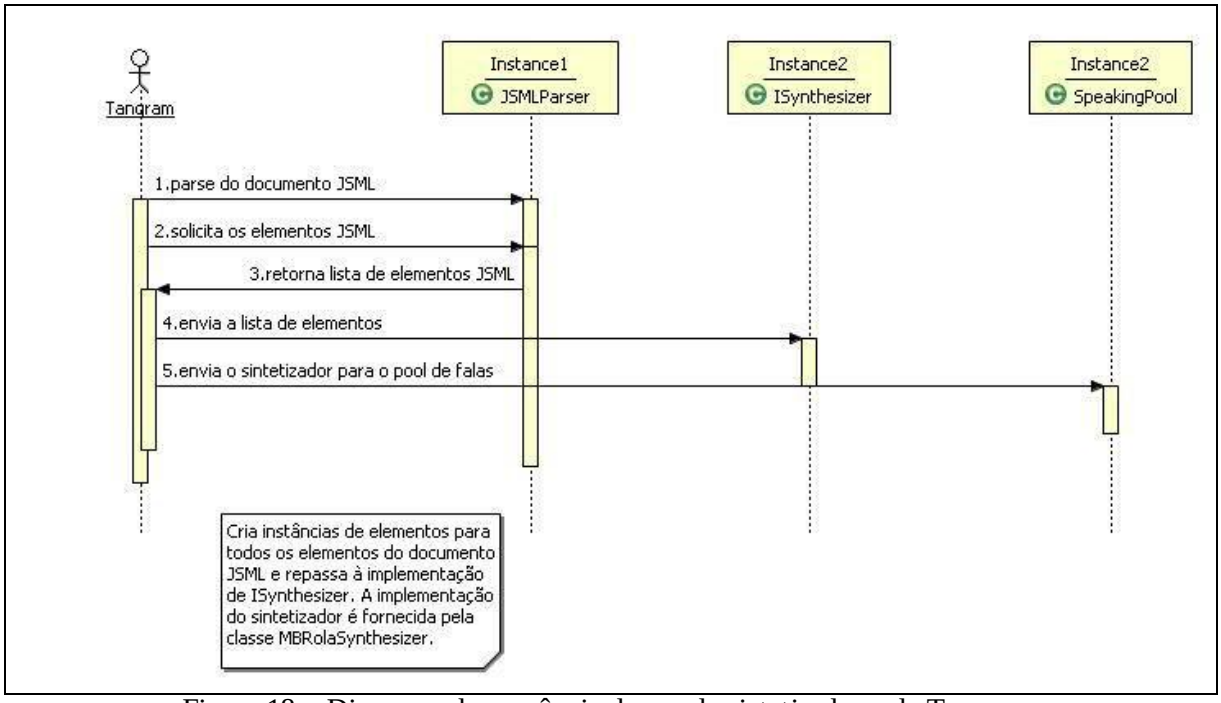


Figura 18 – Diagrama de sequência de uso do sintetizador pelo Tangram

O diagrama de sequência (Figura 18) mostra como o Tangram utiliza a classe `JSMLParser` para realizar o *parse* do documento JSML e como o *parser* devolve uma lista de objetos que representam os elementos JSML e em seguida o Tangram envia esta lista de elementos ao sintetizador. Após o sintetizador estar configurado com a lista de elementos, o sintetizador é enviado para o *pool* de falas onde será realizada a síntese do texto contido no documento JSML.

3.3 IMPLEMENTAÇÃO

A seguir é mostrado como foi realizada a implementação do sintetizador e a integração do sintetizador no Tangram.

3.3.1 Implementação do sintetizador

Conforme a especificação da Sun Microsystems (2009) para a JSML, que provê estruturas para fornecer informações ao sintetizador sobre como o texto deve ser falado, a primeira etapa consiste em realizar um *parse* deste documento e em seguida extrair as informações pertinentes para o sintetizador. O Quadro 27 mostra a implementação da classe que armazena as configurações do elemento `sayas` lido pelo *parser*.

```
public final class SayAs extends AbstractSynthElement implements ISynthElement {
  public static final String CLASS = "class";
  public static final String CLASS_LITERAL = "literal";
  public static final String CLASS_DATE = "date";
  public static final String CLASS_TIME = "time";
  public static final String CLASS_NAME = "name";
  public static final String CLASS_PHONE = "phone";
  public static final String CLASS_NET = "net";
  public static final String CLASS_ADDRESS = "address";
  public static final String CLASS_CURRENCY = "currency";
  public static final String CLASS_MEASURE = "measure";
  public static final String CLASS_NUMBER = "number";
  private Map<String,IAttribute> attributes = new HashMap<String,IAttribute>(1);
      SayAs(Element e) {
            final String[] validValues = { null, CLASS_LITERAL, CLASS_DATE,
                        CLASS_TIME, CLASS_NAME,
                        CLASS_PHONE, CLASS_NET,
                        CLASS_ADDRESS, CLASS_CURRENCY,
                        CLASS_MEASURE, CLASS_NUMBER };
            IAttribute clazz = new AttributeImpl(CLASS);
            clazz.setRequired(false);
            clazz.setValidValues(validValues, "");
            clazz.setValue(e.attributeValue(CLASS));
            attributes.put(CLASS, clazz);
            setMark(e.attributeValue(MARK));
            text = e.getTextTrim();
      }
      public Collection<IAttribute> getAttributes() {
            return Collections.unmodifiableCollection(attributes.values());
      }
      public String getName() {
            return "sayas";
      }
      public IAttribute getAttribute(String attName) {
            return attributes.get(attName);
      }
}
```

Quadro 27 – Classe `SayAs`

Todas as implementações de `ISynthElement` possuem apenas *getters* fornecidos pela *interface*, constantes que representam os atributos do respectivo elemento e um construtor que recebe um objeto que representa um elemento do documento JSML fornecido pelo *parser*.

Em seguida, o sintetizador deve converter o texto juntamente com esta informação fornecida e, de acordo com as regras gramaticais do idioma, criar o documento intermediário que servirá para o MBROLA gerar a voz, neste caso um formato de som *WAVEform audio*

file format (WAVE). O Quadro 28 mostra como o objeto da classe `SayAs` é convertido conforme sua classificação.

```
class SayAsConverter implements IConverter{
  public void convert(ISynthElement element, StringBuilder output) {
       IComponent text = null;
       final String elementText = element.getText();
       final String elementClass = element.getAttribute(SayAs.CLASS).getValue();
/*identifica a classe do element sayas e trata o texto conforme necessário e
instancia a classe que irá tratar esta classe*/
       if (SayAs.CLASS_NUMBER.equals(elementClass)) {
              String numToWord = NumToWord.convert(new BigInteger(elementText));
              text = new Text(numToWord);
       }else if(SayAs.CLASS_LITERAL.equals(elementClass)) {
              text = new Speller(elementText);
       }else if(SayAs.CLASS_DATE.equals(elementClass)) {
              String date = Calendar.formatDate(elementText);
              text = new Text(date);
       }else if(SayAs.CLASS_TIME.equals(elementClass)) {
              String time = Calendar.formatTime(elementText);
              text = new Text(time);
       }else if(SayAs.CLASS_MEASURE.equals(elementClass)) {
              String measure = Measures.processMeasure(elementText, true);
              text = new Text(measure);
       }else if(SayAs.CLASS_NET.equals(elementClass)) {
              String net = Net.processNet(elementText);
              text = new Text(net);
       }else if(SayAs.CLASS_ADDRESS.equals(elementClass)) {
              String address = Address.processAddress(elementText);
              text = new Text(address);
       }else if(SayAs.CLASS_CURRENCY.equals(elementClass)) {
              String currency = Currency.processCurrency(elementText);
              text = new Text(currency);
       }else if(SayAs.CLASS_PHONE.equals(elementClass)) {
              String phone = Phone.processPhone(elementText);
              text = new Text(phone);
       }else {
              text = new Text(element.getText());
       }
       if (text != null) {
       text.configure(ComponentGlobals.BASE_FREQUENCY, ComponentGlobals.BASE_TIME);
       output.append(text.show());
       }
   }
}
```

Quadro 28 – Classe *SayAsConverter*

Para cada classe do elemento `sayas` é invocado um componente do sintetizador para processar adequadamente seu texto. Tomando como exemplo que um elemento seja da classe `date`, o conversor do elemento `sayas` irá invocar o componente `Calendar` para processar um texto no formato "11/05/1985" e retornar na forma "onze de maio de mil novecentos e oitenta e cinco".

Conforme visto no Quadro 28, onde são descritos os fonemas, o Quadro 29 mostra como é feita a conversão da letra "c" para o formato intermediário do sintetizador.

```
switch (letterChar) {
case 'c': {
  sound = "k";
/*se a proxima letra for uma vogal "a", "o" e "u", a letra "c" tem som de "k" */
  if(inArray(convert(nextSyllChar,getVogalsInBasic()),new String[]{"a","o","u"})) {
    sound = "k";
//se a proxima letra for uma vogal "e" e "i", a letra "c" tem som de "s"
 }else if(inArray(convert(nextSyllChar,getVogalsInBasic()),new String[]{"e","i"})){
    sound = "s";
/*se a proxima letra for uma consoante "h", a junção das letras "c" e "h" tem som
de "x" */
  }else if (nextSyllChar.equals("h")) {
    sound = "x";
  i++;
/*se a proxima letra for qualquer outra consoante, a letra "c" tem som de "k" */
  }else if (inArray(nextSyllChar, ComponentGlobals.CONSONANTS)) {
    sound = "k";
  }
break;
}
```

Quadro 29 – Conversão para o formato intermediário

A letra "c" possui dois fonemas, dependendo da letra seguinte. Caso a letra seguinte seja uma das vogais "a", "o" ou "u", seu som será o de uma letra "k". Por exemplo, cita-se as palavras casa, costura e curral. Se a letra seguinte for uma das vogais "e" ou "i", seu som será igual ao de um "s". Como exemplo cita-se as palavras cego e cimento. Já se a letra seguinte for a consoante "h", então a união das letras "c" e "h", "ch", formam o som de um "x". Como exemplo cita-se a palavra chá. Para as demais consoantes, seu som será o de um "k". Como exemplo cita-se a palavra compacto.

Por último, para efetivamente falar, o sintetizador utiliza um *player* de áudio para reproduzir o WAVE. O Quadro 28 mostra uma implementação simples de um *player* de áudio para o formato WAVE.

```
private void playAudioStream(AudioInputStream audioInputStream) {
//provê informação do formato de áudio como canais,tamanho e sample rate
       AudioFormat audioFormat = audioInputStream.getFormat();
//abre uma linha de dados a serem lidos
       DataLine.Info info = new DataLine.Info(SourceDataLine.class, audioFormat);
       if (!AudioSystem.isLineSupported(info)) {
             return;
       }
       try {
//cria um SourceDataLine para reprodução do áudio
       SourceDataLine dataLine = (SourceDataLine) AudioSystem.getLine(info);
//abre a linha de dados e aloca recursos
       dataLine.open(audioFormat);
       if (dataLine.isControlSupported(FloatControl.Type.MASTER_GAIN)) {
//ajusta o volume da reprodução
         FloatControl volume = dataLine.getControl(FloatControl.Type.MASTER_GAIN);
         volume.setValue(4);
       }
       dataLine.start();
//reproduz o áudio
       int bufferSize = (int) audioFormat.getSampleRate() *
audioFormat.getFrameSize();
       byte[] buffer = new byte[bufferSize];
       try {
       int bytesRead = 0;
       while (bytesRead >= 0) {
         bytesRead = audioInputStream.read(buffer, 0, buffer.length);
         if (bytesRead >= 0) {
           dataLine.write(buffer, 0, bytesRead);
         }
       }
       } catch (IOException e) {
         e.printStackTrace();
       }
//encerra a reprodução e libera os recursos
         dataLine.drain();
         dataLine.close();
       } catch (LineUnavailableException e) {
         e.printStackTrace();
       }
}
```

Quadro 30 – *Player* de áudio

O implementação do *player* de áudio simplesmente abre um fluxo de dados para leitura do WAVE através da classe `AudioFormat`. Para a reprodução do áudio é utilizada a classe `SourceDataLine`. Este fluxo de dados lidos do WAVE é lido por demanda e armazenado num *buffer*. Assim que este *buffer* é carregado, ele é passado para o `SourceDataLine` escrevê-lo na saída de som, assim reproduzindo a fala contida no WAVE. Ao final, são liberados os recursos utilizados na reprodução.

3.3.2 Integração do Sintetizador no Tangram

Por meio da API exposta pelo sintetizador, o Tangram pode, através da implementação do comando `fala`, repassar um documento JSML e fazer com que o sintetizador gere a fala

deste documento. O fluxo de execução do sintetizador segue apenas em três partes: leitura do documento JSML, sintetização da voz a partir das informações fornecidas pelo *parser* e reprodução da fala. O Quadro 31 apresenta como o Tangram faz uso desta API.

```
/**Singleton do comando fala.*/
public class ComandoFala implements Comando{
  private static ComandoFala instance = new ComandoFala();
  private static SpeakingPool speakingPool;
  private static SpeechDispatcher speechDispatcher;
/**
 * Retorna um singleton do comendo fala.
 * @return instancia única do comando fala.
 */
public static ComandoFala getInstance() {
  return instance;
}
private ComandoFala() {
  speakingPool = new SpeakingPool();
  speechDispatcher = new SpeechDispatcher(speakingPool);
}
/**Invoca o sintetizador para falar o texto passado no documento JSML. Informa
também se esta síntese deve ser assíncrona.
 * @param filePath documento JSML.
 * @param async se a síntese deste documento deverá ser assíncrona*/
public void speech(String filePath, boolean async) {
  //realiza o parse do documento JSML
  File jsmlFile = new File(filePath);
  JSMLParser parser = new JSMLParser(jsmlFile);
  parser.parse();
  // instancia o sintetizador
  ISynthesizer synth = new MBRolaSynthesizer();
  // configura o sintetizador com os elementos JSML
  synth.configure(parser.getSynthElements());
  // criar uma thread para a fala e adiciona no pool de falas
  SpeechThread t = new SpeechThread(synth, async);
  speakingPool.addSpeech(t);
  //verifica se o dispatcher não foi iniciado, senão não inicia
  if (!speechDispatcher.isAlive()) {
    speechDispatcher.start();
  }
}
```

Quadro 31 – Utilização da API exposta

O *Javadoc* das classes `JSMLParser`, `ISynthesizer`, `MBrolaSynthesizer` estão disponíveis no Apêndice B.

A classe `ComandoFala` segue o padrão *singleton*, permitindo apenas uma instância em todo o Tangram. Sempre que um comando de fala for executado na linguagem LTD, o semântico irá executar o método `speech` da classe `ComandoFala`, passando o documento JSML que será sintetizado e se a fala é sobreposta ou exclusiva.

Para representar se a fala é sobreposta ou exclusiva é utilizado o parâmetro booleano `async`, abreviado de *asynchronous*. Uma fala assíncrona indica que ela independe de outras, podendo haver várias falas sendo executadas simultaneamente, ou seja, a fala é sobreposta. Uma fala síncrona significa que é necessário que ela termine para que outra seja executada, assim sendo uma fala exclusiva. O valor `true` para o parâmetro `async` indica que a *thread* da

fala será assíncrona e `false` para indicar que a *thread* será síncrona.

Primeiramente a classe `JSMLParser` irá realizar o *parse* do documento JSML e fornecer seus elementos para a implementação do sintetizador, a classe `MBrolaSynthesizer`. Em seguida é criada uma *thread* para a fala, fornecendo o sintetizador e configurando se ela será assíncrona ou síncrona e adicionada num *pool* de falas. O *pool* é implementado na classe `SpeakingPool` (Quadro 32). Por último, a classe `SpeechDispatcher` (Quadro 33) é a responsável por recuperar as falas contidas no *pool* e executá-las obedecendo o critério de serem assíncronas ou síncronas.

```
public final class SpeakingPool {
  private static ConcurrentLinkedQueue<SpeechThread> pool = new
ConcurrentLinkedQueue<SpeechThread>();//falas que estão aguardando sua vez
  /**Adiciona uma fala no final da fila de falas.
   * @param t thread de fala. */
  public synchronized void addSpeech(SpeechThread t) {
       pool.offer(t);
       notify();
  }
  /**Retorna uma {@link SpeechThread} do pool removendo-a. Caso o pool esteja
vazio, aguarda o método addSpeech(SpeechThread)
 ser chamado para então devolver a fala solicidata.
 * @return retorna a primeira fala da fila.
 * @throws InterruptedException */
  public synchronized SpeechThread retrieveSpeech() throws InterruptedException {
       notify();
       if(pool.isEmpty()) {
              wait();
       }
       return pool.poll();
  }
}
```

Quadro 32 – Implementação do *pool* de falas

O método `addSpeech` (Quadro 32) adiciona uma *thread* do sintetizador no final da fila de falas do tipo `ConcurrentLinkedQueue<SpeechThread>` e notifica que existem falas a serem executadas. O método `retrieveSpeech` (Quadro 32) retorna uma *thread* do sintetizador para ser executada e notifica que uma fala foi removida da fila. Se a fila estiver vazia quando este método for invocado, o *pool* irá aguardar até que alguma fala seja incluída na fila novamente.

```
public final class SpeechDispatcher extends Thread {
  private SpeakingPool pool;
  private static ConcurrentLinkedQueue<SpeechThread> queue = new
ConcurrentLinkedQueue<SpeechThread>();//falas que estão aguardando sua vez
  /** Construtor do dispathcher de falas.
   * @param pool é o pool de threads pendentes de serem faladas. */
  public SpeechDispatcher(SpeakingPool pool) {
       this.pool = pool;
  }
  public void run() {
    while (true) {
       try {
       SpeechThread t = pool.retrieveSpeech();
       if (t.isAsync()) {// é assincrona (sobreposta)
              queue.offer(t);
              t.start();
       } else {// é sincrona (não sobreposta)
              if (!isSpeaking()) {// tem alguem falando?
                     t.start();
                     t.join();
              } else {
              // tem alguem falando, então devolve a fala pro pool
                     pool.addSpeech(t);
              }
       }
       speechGC();// limpa as falas que terminaram
       } catch (InterruptedException e) {
              e.printStackTrace();
       }
    }
  }
  /**Varre todas as falas e verifica se tem alguma terminada. se encontrar alguma
remove da fila. */
  private synchronized void speechGC() {
       for (SpeechThread t : queue) {
              if (t.getState() == State.TERMINATED) {
                     queue.remove(t);
              }
       }
  }
  /** Retorna se a thread do parâmetro está falando. */
  private static boolean isSpeaking(SpeechThread speech) {
       return speech != null && (speech.getState() == State.RUNNABLE &&
speech.getState() == State.TIMED_WAITING);
  }
  /**Retorna se alguem da lista está falando. */
  private synchronized boolean isSpeaking() {
  // retorna se alguem da lista de falas está executando
       for (SpeechThread t : queue) {
              if (isSpeaking(t)) {
                     return true;
              }
       }
       return false;
  }
}
```

Quadro 33 – Implementação da classe `SpeechDispatcher`

A classe `SpeechDispatcher` (Quadro 33) é responsável por executar as falas que estão no *pool* de falas. Esta classe armazena uma fila de falas que estão sendo executadas simultaneamente e o *pool* de onde são retiradas as falas.

Uma premissa para execução das falas é de que se existe uma *thread* síncrona sendo

executada, nenhuma outra *thread* de fala é executada até que esta *thread* síncrona termine. A outra premissa é a de que se existe alguma *thread* assíncrona sendo executada e uma *thread* síncrona seja eleita para execução, esta *thread* síncrona voltará para o *pool* até que todas as demais *threads* terminem de executar.

3.3.3 Operacionalidade de um modelo com fala

Para a confecção de um documento JSML é utilizado um editor de texto externo ao Tangram. Nele será informado o texto que será sintetizado juntamente com informações de como este texto deverá ser sintetizado. O Quadro 34 descreve um documento JSML com variações do elemento `sayas`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<jsml lang="pt-br">
       <voice gender="male" age="adult" variant="6" name="br1"></voice>
       <sayas>Atenção Proprietário do veículo placa</sayas>
       <break time="500" />
       <sayas class="literal">MDK1234</sayas>
       <break time="500" />
       <sayas>favor comparecer ao seu veículo.</sayas>
       <break time="1000" />
       <sayas>O carro novo</sayas>
       <sayas class="measure">0Km</sayas>
       <sayas>custa</sayas>
       <sayas class="currency">R$3560,00</sayas>
       <sayas>em até</sayas>
       <sayas class="number">15</sayas>
       <sayas>vezes.</sayas>
       <break time="500" />
       <sayas>Mas isto até</sayas>
       <sayas class="date">09/07/2010</sayas>
       <sayas>no endeço</sayas>
       <sayas class="address">R. Antônio da Veiga n°</sayas>
       <sayas class="number">140</sayas>
       <sayas>telefone</sayas>
       <sayas class="phone">3323-6544</sayas>
       <sayas>ou no endereço eletrônico</sayas>
       <sayas class="net">http://www.inf.furb.br</sayas>.
</jsml>
```

Quadro 34 – Exemplo de um documento JSML

Com posse de um documento JSML, cria-se um modelo de moinho no LTD conforme a Figura 19.

Figura 19 – Menu de criação de modelo

Após o modelo ter sido criado, utiliza-se o editor visual (Figura 20(1)) para modelar as peças formando um moinho. Ao movimentar as peças de um modelo, esta configuração é refletida no editor textual (Figura 20(2)).

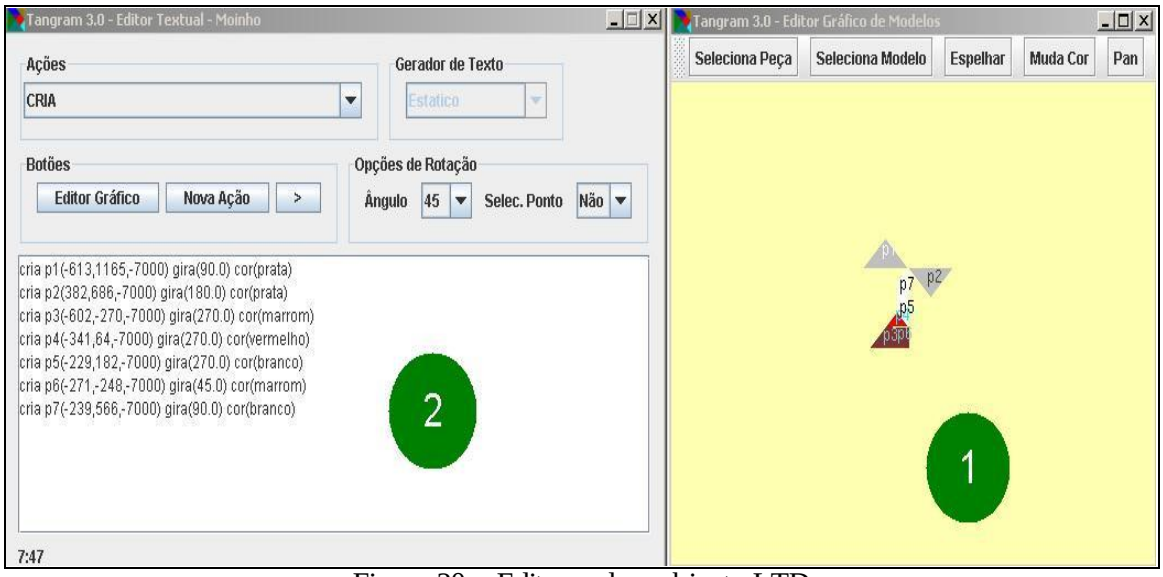


Figura 20 – Editores do ambiente LTD

O LTD suporta criação e execução de métodos para um modelo. Para criar um método clica-se no botão "Nova Ação" do editor textual. Em seguida atribui-se um nome a este método. Por fim, utiliza-se o editor textual para implementar o método com o comando `fala` e comandos de animação do modelo. O Quadro 35 descreve o método `tagarela`.

```
fala('fala_moinho.jsml')
enquanto fala inicio
      p1.gira(-1) no ponto(9)
      p2.gira(-1) no ponto(12)
      pisca(10)
fim
```

Quadro 35 – Implementação do comando `tagarela`

Quando da execução do método `tagarela` (Quadro 35), o comando fala ao ser executado irá sintetizar o conteúdo do documento `fala_moinho.jsml`. Em seguida os comandos de animação do modelo contidos no bloco do comando `enquanto fala` será executado continuamente enquanto o documento `fala_moinho.jsml` estiver sendo sintetizado pelo comando `fala`. Assim que a síntese terminar, a execução irá sair do comando `enquanto fala` e irá seguir normalmente sua execução a patir do comando seguinte ao fim do bloco.

3.4 RESULTADOS E DISCUSSÃO

Este trabalho é semelhante ao FreeTTS por possuir suporte à síntese de voz, que, diferentemente do *Sphinx4,* implementa suporte ao reconhecimento de voz. Tanto o *Sphinx4* quanto o FreeTTS implementam uma parte da JSAPI. Apesar de o *Opera Voice* realizar a síntese e o reconhecimento de voz, utiliza outra especificação, o *VoiceXML*. Todos estes citados suportam apenas o idioma inglês.

Em comparação com o protótipo de Oechsler (FurbTTS) evidenciou-se uma maior semelhança principalmente por suportarem o idioma português e utilizarem o mesmo processador acústico, o MBROLA.

No Quadro 36 é apresentado um comparativo das principais características de cada trabalho correlato com este projeto desenvolvido (LTD TTS).

Características	**LTD TTS**	**FreeTTS**	**FurbTTS**	**Sphynx4**	**OperaVoice**
Vocabulário irrestrito	sim	sim	sim	não	sim
Síntese de texto	sim	sim	sim	não	sim
Reconhecimento de voz	não	não	não	sim	sim
Multiplataforma	sim	sim	não	sim	não
Passível de extensão	sim	sim	sim	sim	não
Suporte idioma português	sim	não	sim	não	não
Suporte multi-idioma	não	não	não	não	não

Quadro 36 – Comparativo com trabalhos correlatos

4 CONCLUSÕES

O LTD é um ambiente visual de programação voltado à aprendizagem. Nele é possível criar mundos e modelos animados através de um *script* com comandos que definem um comportamento. Um modelo programado no LTD possui uma forma e uma animação, sendo que neste trabalho foi desenvolvido o suporte a síntese de voz.

Este suporte a fala no LTD, além de proporcionar a experiência mais rica com o modelo, também demonstra entonações na voz, dando maior autenticidade em frases interrogativas e exclamativas, palavras enfáticas e distinguindo quando um número representa uma data, um valor monetário ou um horário, por exemplo, assim como soletração de siglas.

Por não utilizar um dicionário de palavras para diferenciar conjugação de verbos com substantivos, foram constatadas algumas deficiências no processo de síntese, como a palavra "piloto" que pode ser aplicada nas seguintes sentenças "Eu piloto um avião azul" e "O piloto de moto ganhou a corrida". Na primeira a palavra piloto é uma conjugação do verbo pilotar. Já na segunda é o substantivo piloto. Nestes exemplos a palavra piloto possui entonações diferentes apesar da grafia ser a mesma. Outro exemplo é a sílaba "que" utilizada nas palavras "queijo" e "cinquenta". Na implementação deste trabalho assumiu-se que sempre que a letra "q" for seguida do encontro vogal "ue" e "ui" o som será o fonema "ke" e "ki" respectivamente, como na palavra "queijo". Esta deficiência poderia ser resolvida utilizando um dicionário de palavras de exceção. Neste caso o sintetizador utilizaria este dicionário para analisar o emprego da palavra na frase e realizaria um tratamento diferenciado para suportar mais de uma pronúncia para a mesma palavra.

Por fim, conclui-se que a implementação do suporte à fala no LTD mostrou-se funcional e com um resultado final satisfatório, sendo possível entender o que está sendo falado pelo sintetizador, apesar das limitações citadas.

4.1 EXTENSÕES

O fato de o LTD ter como foco o ensino de programação a crianças, uma de suas características deve ser a usabilidade. Para tanto são sugeridas as seguintes extensões, além de incrementos no próprio sintetizador:

a) editor de documento JSML integrado ao Tangram;
b) suporte a síntese de voz de outros idiomas além do português;
c) suporte a dicionário de palavras de exceção, onde será possível diferenciar a pronúncia de palavras que possuem a mesma grafia;
d) integração do LTD com outros sintetizadores, como por exemplo, o FreeTTS ou o protótipo de Oechsler (2009).

REFERÊNCIAS BIBLIOGRÁFICAS

ALCÂNTARA Júnior, Oscar. **Protótipo de uma linguagem de programação de computadores orientada por formas geométricas, voltada ao ensino de programação**. 2003. 58 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GESSER. Carlos. E. **GALS**: gerador de analisador léxico e sintático. [S.1.], 2003. Disponível em: <http://gals.sourceforge.net/>. Acesso em: 01 jun. 2010.

KNIHS, Glauco. **Linguagem de programação visual baseada em tipos abstratos de dados**. 2008. 109 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MBROLA TEAM. **MBROLA**. [S.1.], 2000. Disponível em: <http://tcts.fpms.ac.be/synthesis/mbrola.html>. Acesso em: 28 mar. 2010.

OECHSLER, Thiago M. **Processamento de texto escrito em linguagem natural para um sistema conversor texto-fala**. 2009. 74 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

OPERA DEVELOPER. **Opera Voice**. [S.1.], 2009. Disponível em: <http://dev.opera.com/articles/voice/>. Acesso em: 10 mar. 2010.

PASSOS, Luis O. T. Análise das possibilidades e qualidades musicais dos métodos de síntese digital do som usados na obra “sinos, tambores e órgãos”. In: SEMINÁRIO DE ENGENHARIA DE ÁUDIO, 1., 2002, Belo Horizonte. **Anais...** Belo Horizonte: DELT/UFMG, 2002. Não paginado. Disponível em: <http://www.cpdee.ufmg.br/~semea/anais/artigos/LuisPassos.pdf>. Acesso em: 28 out. 2009.

PAULA, Wilson P. F. **Multimídia:** conceitos básicos. Rio de Janeiro: LTC, 2000.

SOURCEFORGE. **Freetts 1.2**. [S.1.], 2001. Disponível em: <http://freetts.sourceforge.net/>. Acesso em: 03 mar. 2010.

_____. **Sphinx4**. [S.1.], [2008?]. Disponível em: <http://cmusphinx.sourceforge.net/sphinx4/>. Acesso em: 10 mar. 2010.

SUN MICROSYSTEMS. **API markup language specification**. [S.1.], [2009?]. Disponível em: <http://java.sun.com/products/java-media/speech/forDevelopers/JSML/Specification.html>. Acesso em: 14 mar. 2010.

THEISS, Fabricio J. **Linguagem visual orientada por formas geométricas, voltada ao ensino de programação**. 2006. 82 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

APÊNDICE A – Ações semânticas

Os significados das ações semânticas apresentadas na gramática do Quadro 22 são explicados no Quadro 37 e as novas ações relacionadas ao comando `fala` no Quadro 38.

Ação	Significado
#0	Inicia a compilação de um modelo.
#1	Inicia a compilação de um mundo.
#2	Retira o comando laço da pilha de comandos laço e cria um modelo com apenas o método cria.
#3	Guarda o nome do modelo na variável nomeDoModelo. Se o nome antigo não for nulo e o novo nome for diferente dispara uma exceção.
#4	Guarda nome do método na variável nomeDoMetodo.
#5	Guarda nome dado ao modelo no mundo na variável idDoModelo.
#6	Guarda nome do mundo na variável nomeDoMundo.
#7	Guarda peça na variável idDaPeca.
#8	Guarda a cor na variável nomeDaCor.
#9	Guarda o valor na variável x.
#10	Guarda o valor na variável y.
#11	Guarda o valor na variável z.
#12	Cria comando laço e o coloca no topo da pilha.
#13	As seguintes restrições devem ser atendidas: o método em que o comando irá ser inserido deve ser o método CRIA e a peça não deve estar criada. Se atender as restrições, então cria-se o comando cria (para a peça), marcando a peça como criada. Marca a variável comandoEmModelo igual a falso.
#14	Adiciona comando criado ao laço do topo da pilha de comandos laço e zera variáveis x, y e z.
#15	Marca a variável comandoEmModelo igual a verdadeiro.
#16	Marca a variável comandoEmModelo igual a falso e verifica se a peça foi criada.
#17	Cria comando move.
#18	Cria comando gira.
#20	Cria comando cor.
#21	Cria comando espelha.
#22	Cria comando pisca.
#23	Cria comando laço e coloca-o no topo da pilha de comandos laço.
#24	Retira o comando laço do topo da pilha de comandos laço e adiciona-o no novo topo da

	pilha.
#25	Cria o comando faça e coloca o nome do método na lista de métodos usados.
#26	Retira o comando laço do topo da pilha e coloca-o na lista de métodos do modelo.
#27	Verifica se os identificadores da lista de métodos usados foram todos definidos como um método do modelo. Se algum método foi usado sem ser definido, retorna um erro. Cria o método VIVA para o modelo e o finaliza criando um objeto da classe ModeloExecutavel.
#28	Verifica se o id dado ao modelo já foi usado. Verifica na lista de modelos compilados se o modelo já foi compilado. Se não foi, compila-o e o coloca na lista de modelos compilados. Cria novo ModeloExecutavel com uma nova Figura para o id dado ao modelo e adiciona na lista de id do mundo. Cria o comando faça-no-mundo, o qual chama o método CRIA do id dado ao modelo.
#29	Cria o comando move-para, o qual move o modelo para o ponto inicial.
#30	Verifica se o id dado ao modelo foi criado. Caso foi criado, verifica se o modelo não está executando o comando VIVA e não foi executado o comando APAGA. Caso atenda as verificações, então cria o comando VIVA.
#31	Verifica se o id dado ao modelo foi criado. Caso foi criado, verifica se o modelo está executando o comando VIVA. Caso não esteja executando, cria comando faça-no-mundo chamando o método TERMINA do id dado ao modelo.
#32	Verifica se o id dado ao modelo foi criado. Cria o comando APAGA.
#33	Limpa as variáveis x, y, z, idDoModelo e emParalelo.
#34	Verifica se o id dado ao modelo foi criado. Cria-se o comando faça-no-mundo para o modelo do id, chamando o método da variável nomeDoMetodo.
#35	Marca variável emParalelo como verdadeiro.

Fonte: Knihs (2008, p. 107).

Quadro 37 – Ações semânticas criadas por Knihs (2008)

#37	Guarda o valor da variável jsml do comando fala.
#38	Informa que a fala é assíncrona (sobreposta).
#39	Cria o comando fala.
#40	Cria o comando enquanto fala e coloca no topo da pilha.
#41	Retira o comando enquanto fala do topo da pilha de comandos e adiciona de novo na pilha.
#42	Cria o comando espera fala

Quadro 38 – Ações semânticas do comando `fala`

APÊNDICE B – Javadoc da API exposta

Neste apêndice é listado o *Javadoc* (Quadro 39, Quadro 40 e Quadro 41) das classes `JSMLParser` e `MBrolaSynthesizer` e da *interface* `ISynthesizer`.

inf.furb.xml

Class JSMLParser

```
java.lang.Object
  └ inf.furb.xml.JSMLParser
```

```
public final class JSMLParser
```

extends java.lang.Object

Parser XML do documento JSML.

Constructor Summary

JSMLParser(java.io.File jsmlFile) Construtor do parser do documento JSML que recebe um `File` que representa o local físico do documento.	

Method Summary

java.util.List<inf.furb.synthesis.jsml.ISynthElement>	**getSynthElements**() Retorna a lista de `ISynthElement` lidos do documento JSML passado construtor e lidos no método parse();
void	**parse**() Realiza o parse do documento JSML passado no construtor.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

JSMLParser

```
public JSMLParser(java.io.File jsmlFile)
```

Construtor do parser do documento JSML que recebe um `File` que representa o local físico do documento.

Parameters:
`jsmlFile` - arquivo do documento JSML

Method Detail

getSynthElements

```
public java.util.List<inf.furb.synthesis.jsml.ISynthElement> getSynthElements()
```

Retorna a lista de `ISynthElement` lidos do documento JSML passado construtor e lidos no método parse();

Returns:
lista de `ISynthElement`

---

parse

```
public void parse()
```

Realiza o parse do documento JSML passado no construtor.

Quadro 39 – *Javadoc* da classe `JSMLParser`

inf.furb.synthesis.mbrola

Class MBRolaSynthesizer

```
java.lang.Object
  └ inf.furb.synthesis.mbrola.MBRolaSynthesizer
```

All Implemented Interfaces:
inf.furb.synthesis.ISynthesizer, java.lang.Runnable

```
public final class MBRolaSynthesizer
```

extends java.lang.Object
implements inf.furb.synthesis.ISynthesizer

Implementação do sintetizador MBRola.

Field Summary

Fields inherited from interface inf.furb.synthesis.ISynthesizer
CONFIG_IMPUT, CONFIG_OUTPUT, CONFIG_VOICE

Constructor Summary

MBRolaSynthesizer() Construtor padrão do sintetizador MBRola.

Method Summary

void	**configure**(java.util.List<inf.furb.synthesis.jsml.ISynthElement> elements)
void	**run**()
void	**speech**()

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MBRolaSynthesizer

```
public MBRolaSynthesizer()
```

Construtor padrão do sintetizador MBRola.

Method Detail

configure

```
public void configure(java.util.List<inf.furb.synthesis.jsml.ISynthElement> elements)
```

Specified by:
configure in interface inf.furb.synthesis.ISynthesizer

---

speech

```
public void speech()
```

Specified by:
speech in interface inf.furb.synthesis.ISynthesizer

---

run

```
public void run()
```

Specified by:
run in interface java.lang.Runnable

Quadro 40 – *Javadoc* da classe `MBrolaSynthesizer`

inf.furb.synthesis

Interface ISynthesizer

All Superinterfaces:
java.lang.Runnable

```
public interface ISynthesizer
```

extends java.lang.Runnable

Iterface que define um sintetizador.

Field Summary	
static java.lang.String	**CONFIG_IMPUT**
static java.lang.String	**CONFIG_OUTPUT**
static java.lang.String	**CONFIG_VOICE**

Method Summary	
void	**configure**(java.util.List<inf.furb.synthesis.jsml.ISynthElement> config) Configura o sintetizador.
void	**speech**() Faz com que o sintetizador fale o texto atribuído a ele.

Methods inherited from interface java.lang.Runnable
run

Field Detail

CONFIG_VOICE

```
static final java.lang.String CONFIG_VOICE
```

See Also:
Constant Field Values

CONFIG_IMPUT

```
static final java.lang.String CONFIG_IMPUT
```

See Also:
Constant Field Values

CONFIG_OUTPUT

```
static final java.lang.String CONFIG_OUTPUT
```

See Also:
Constant Field Values

Method Detail

configure

```
void configure(java.util.List<inf.furb.synthesis.jsml.ISynthElement> config)
```

Configura o sintetizador.
Nesta etapa o sintetizador inicia suas configurações necessárias como vozes disponíveis, SO utilizado, I/O, etc.

Parameters:
config -

---

speech

```
void speech()
```

Faz com que o sintetizador fale o texto atribuído a ele.

Quadro 41 – *Javadoc* da *interface* `ISynthesizer`