UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DANIEL ALMEIDA LEITÃO

“NLFORSPEC: UMA FERRAMENTA PARA GERAÇÃO DE ESPECIFICAÇÕES FORMAIS A PARTIR DE CASOS DE TESTE EM LINGUAGEM NATURAL”

Dissertação apresentada ao Curso de Mestrado em Ciência da Computação como requisito parcial à obtenção do grau de Mestre em Ciência da Computação.

ORIENTADORA: Profa. Flávia de Almeida Barros

RECIFE, AGOSTO/2006

Agradecimentos

A minha família, fonte de carinho e segurança, por ter proporcionado todas as condições para que pudesse chegar até aqui.

A Águida, minha namorada, meu amor, por todo carinho, apoio e incentivo em todos os momentos.

A orientadora e amiga Flávia Barros, um exemplo de dedicação e paciência, por ter me guiado e incentivado durante todo o trabalho.

A Dante "*Champion*" Torres, pela amizade e parceria no desenvolvimento desse trabalho.

Aos pesquisadores do CIn/Motorola *Test Research Project*, pelas valiosas sugestões e por terem proporcionado um ambiente acadêmico altamente qualificado para desenvolvimento deste trabalho.

Aos cc99únicos, membros da melhor turma de todos os tempos, pela amizade e momentos de alegria compartilhados durante todo o mestrado.

Resumo

Este trabalho propõe NLForSpec, uma ferramenta para geração de especificações formais a partir de casos de teste em Linguagem Natural. NLForSpec é parte de um projeto maior desenvolvido em parceria entre o CIn-UFPE e a Motorola, que tem como objetivo automatizar a seleção, geração e avaliação de casos de teste para aplicações de telefones móveis. Uma das principais tarefas desse projeto é atualizar automaticamente os requisitos a partir de casos de teste mais atuais.

Nesse cenário, a ferramenta NLForSpec é responsável por gerar especificações na linguagem formal CSP (*Communicating Sequential Processes*) a partir de descrições de casos de teste. Essas especificações serão utilizadas como entrada no processo de atualização ou geração de documentos de requisitos a partir de casos de teste mais atuais.

NLForSpec foi construída com base na arquitetura simbólica tradicional para interpretação de LN, e contém quatro bases de conhecimento (Léxico, Gramática de Casos, Ontologia e Base de Especificações CSP) e três módulos de processamento (*POS-Tagging*, Processamento Semântico e Geração de Casos de Teste Formais). NLForSpec apresentou um desempenho satisfatório em um estudo de caso realizado para o domínio de descrições de casos de teste para aplicações de *Messaging* da Motorola.

Palavras-chave: Interpretação de Linguagem Natural, Geração de especificações de casos de teste, Inteligência artificial simbólica.

Abstract

This work describes the NLForSpec, a Natural Language Processing tool to translate software test cases descriptions in English into a formal representation. NLForSpec is part of a larger project (CIn-Motorola BTC Test Research Project), which aims to automate test case generation, selection and evaluation for mobile phone applications. One of the projects main goals is to provide for automatic update of requirements documents from more up-to-date test cases (since test cases change more often than requirements).

In this scenario, the NLForSpec tool is the first step in the translation process from test case descriptions into formal representations used to update or generate requirements documents from more up-to-date test cases. NLForSpec generates specifications in CSP (Communicating Sequential Processes) from English test cases descriptions.

NLForSpec was built based on the traditional pipeline NL interpretation architecture. The input sentence is parsed and then mapped into case grammar structures (based on thematic roles). These structures are then mapped into representation in CSP formal language. The CSP representations are used as input by another tool of the major project. NLForSpec reached satisfactory results in a case study performed in Motorola Messaging domain.

Keywords: Natural Language Interpretation, Test Cases Specifications, Symbolic Artificial Intelligence

ÍNDICE

1. Introdução 1
1.1. Trabalho Realizado 2
1.2. Organização da Dissertação 3

2. Interpretação de Linguagem Natural 5
2.1. Abordagens para PLN 6
2.1.1. Abordagem Simbólica 6
2.1.2. Abordagem Empírica 7
2.1.3. Abordagem Neural 8
2.2. Interpretação de LN – Abordagem Simbólica 9
2.2.1. Arquitetura Genérica 9
2.2.2. Bases de Conhecimento 10
2.2.3. Módulos de Processamento 13
2.3. Considerações Finais 27

3. Geração de Modelos a partir de Linguagem Natural 28
3.1. Sistema para Geração de Especificações Formais a partir de Documentos de Requisitos 28
3.2. Sistema para Geração de Especificações Algébricas 34
3.3. O sistema NL-OOPS 35
3.4. Geração de Modelos VHDL 38
3.5. Considerações Finais 39

***4. Test Research Project* 41**
4.1. Atividades do *Test Research Project* 42
4.1.1. Geração de Casos de Teste 43
4.1.2. Estimativas de Execução e Cobertura de Código 45
4.1.3. Atualização dos Requisitos 46
4.2. Casos de Teste 47
4.3. CSP (*Communicating Sequential Processes*) 50
4.4. Considerações Finais 53

5. A Ferramenta NLForSpec 54
5.1. Arquitetura 54
5.2. Bases de Conhecimento 55
5.2.1. Ontologia 55
5.2.2. Léxico 57
5.2.3. Gramática de Casos 60
5.2.4. Base de Especificações CSP 63
5.3. Módulos de Processamento 67
5.3.1. POS-Tagging 67
5.3.2. Processamento Semântico 70
5.3.3. Geração de Casos de Teste Formais 72

5.4. Considerações Finais 74

6. Protótipo e Experimentos 75

6.1. Protótipo Implementado 75
6.1.1. Bases de Conhecimento 76
6.1.2. Módulos de Processamento 80
6.1.3. Escolha do POS-Tagger 82
6.2. Experimentos e Resultados 83
6.2.1. Preparação das Bases de Conhecimento 84
6.2.2. Primeiro Experimento 85
6.2.3. Segundo Experimento 87
6.3. Considerações Finais 87

7. Conclusões 89

7.1. Principais Contribuições 89
7.2. Trabalhos Futuros 90

Referências 92

LISTA DE FIGURAS

Figura 2.1: Arquitetura de Sistema para Interpretação de LN 10
Figura 2.2: Exemplo de Léxico 11
Figura 2.3: Exemplo de Ontologia 13
Figura 2.4: Modelo de estados finitos para adjetivos em inglês 14
Figura 2.5: Modelo de estados finitos para adjetivos 15
Figura 2.6: Exemplo de representação na *Two-Level Morphology* 16
Figura 2.7: Exemplo de unificação 16
Figura 2.8: Exemplo de Gramática de Constituintes Imediatos 18
Figura 2.9: Exemplo de sentenças 19
Figura 2.10: Gramática no formalismo PATRII 20
Figura 2.11: Rede de Transição Recursiva 21
Figura 2.12: Exemplo de sentenças 23
Figura 2.13: Papéis Temáticos 23
Figura 2.14: Exemplo de entrada lexical do verbo “botar” 24
Figura 2.15: Exemplo de frases com o mesmo significado 24
Figura 2.16: Exemplo de Rede Semântica 25
Figura 2.17: Frases e Fórmulas Lógicas 26
Figura 3.1: Arquitetura do Sistema [Lee & Bryant, 2002] 29
Figura 3.2: Trecho de Documento de Requisito [Lee & Bryant, 2002] 29
Figura 3.3: Trecho do documento em XML 30
Figura 3.4: Resultado da Análise Sintática 30
Figura 3.5: Base de Conhecimento de Contextos 31
Figura 3.6: Especificação TLG 32
Figura 3.7: Especificação VDM++ 33
Figura 3.8: Arquitetura do Sistema LOLITA [Long & Garigliano, 1994] 36
Figura 3.9: Exemplo de Entrada e Saída do Sistema NL-OOPS 37
Figura 3.10: Exemplo de grafo conceitual [Cyre et al., 1994] 39
Figura 4.1: Fluxo de Informações do *Test Research Project* 42
Figura 4.2: Teste Baseado em Modelo [Gold Practices, 2006] 44
Figura 4.3: Exemplo de Caso de Teste 49
Figura 4.4: Exemplo de *datatype* simples 51
Figura 4.5: Exemplo de uso de tupla em um *datatype* 51
Figura 4.6: Exemplo de *datatype* composto 51
Figura 4.7: Exemplos de Processos CSP 52
Figura 5.1: Arquitetura de NLForSpec 54
Figura 5.2: Exemplo de trecho da Ontologia 56
Figura 5.3: Exemplos de *Nouns* do Léxico 58
Figura 5.4: Verbos do Léxico 58
Figura 5.5: Modificadores do Léxico 59
Figura 5.6: Modificadores parametrizados 60
Figura 5.7: Exemplo de *case frame* 61
Figura 5.8: Exemplo de Restrições de um *case frame* 62
Figura5.9: Exemplo de *datatype* CSP 65
Figura 5.10: Exemplo de canal CSP 66

Figura 5.11: Ilustração de caso de teste em CSP ____ 67
Figura 5.12: Exemplo de cabeçalho CSP ____ 73
Figura 6.1: Diagrama de classes da Ontologia ____ 77
Figura 6.2: Diagrama de classes do Léxico____ 78
Figura 6.3: Digrama de Classes da Gramática de Casos ____ 79
Figura 6.4: Diagrama de Classes da Base de Especificações CSP____ 80
Figura 6.5: Diagrama de Classes dos Módulos de Processamento____ 81
Figura 6.6: Ferramenta de análise da ontologia ____ 84

ÍNDICE DE QUADROS

Quadro 2.1: Classificação das Gramáticas Gerativas ____ 17
Quadro 2.2: Condições e Ações de uma RTA ____ 21
Quadro 2.3: Exemplo de classificação de papéis temáticos ____ 24
Quadro 5.1: Exemplo de *tags* do *Penn Trrebank Tagset* ____ 68
Quadro 5.2: Entrada e Saída após 1ª Etapa do *POS-Tagger* ____ 69
Quadro 5.3: Saída após 2ª Etapa do *POS-Tagging* ____ 70
Quadro 5.4: Saída após o processamento semântico ____ 71
Quadro 5.5: Entrada e Saída após a geração do evento CSP ____ 72
Quadro 6.1: Estudo comparativo dos POS-Taggers ____ 83
Quadro 6.2: Números da preparação inicial das bases ____ 85
Quadro 6.3: Números relacionados ao Experimento 1 ____ 86

1. Introdução

O processo de desenvolvimento de software envolve três artefatos principais: requisitos, código e casos de teste. Requisitos e casos de teste são freqüentemente descritos em alguma Linguagem Natural (LN), pois esta é a forma de comunicação natural dos *stakeholders*. Porém, descrições em LN podem ser ambíguas e/ou inconsistentes. Como conseqüência, a interpretação dos requisitos do software e dos casos de teste depende da experiência do leitor. Uma má interpretação desses artefatos pode levar a erros na fase de codificação e de execução dos testes.

Com o objetivo de diminuir esses problemas, seria de grande utilidade derivar especificações formais não-ambíguas a partir de descrições em LN (requisitos ou casos de teste). Porém, essa não é uma tarefa simples, pois o uso de notações formais requer conhecimento de um especialista e/ou suporte de ferramentas específicas. Em virtude disso, linguagens formais são mais freqüentemente utilizadas em aplicações críticas e de tempo real, que demandam requisitos extremamente bem definidos. Uma solução desejável seria gerar essas especificações formais automaticamente a partir dos documentos de requisitos ou dos casos de teste.

Já em relação às empresas de desenvolvimento de software, três cenários são comumente encontrados: (1) empresas mais organizadas mantêm requisitos, código e casos de teste definidos com base nos requisitos; (2) outras empresas mantêm código e casos de teste definidos a partir de requisitos não documentados; e, finalmente, (3) poucas empresas produzem e mantêm apenas o código, sem a utilização dos outros dois artefatos.

Neste cenário, um sistema que atualizasse automaticamente os requisitos a partir de casos de teste mais atuais[1] (engenharia reversa) seria de grande interesse para empresas na situação (1) mencionada anteriormente. Já para empresas na situação (2),

[1] Em geral, os casos de teste mudam mais rapidamente do que os requisitos, a fim de refletir as modificações no código.

tal sistema teria como objetivo contribuir com a geração parcial de documentos de requisitos a partir dos casos de teste.

O processo de atualização automática de requisitos de software teria como ponto de partida os casos de teste. A partir deles, seria possível gerar um modelo de uso formal do software. Este processo segue a abordagem *Anti-Model Based Testing* [Bertolino *et al.*, 2004]. O modelo de uso gerado seria comparado a outro modelo de uso formal obtido a partir dos requisitos, a fim de, quando necessário, atualizar os requisitos. Caso não existam documentos de requisitos, o modelo de uso gerado a partir dos casos de teste poderia ser utilizado para gerar parcialmente os requisitos.

Dessa forma, uma ferramenta que recebesse como entrada descrições de casos de teste em Linguagem Natural e gerasse uma especificação formal como saída seria de grande ajuda. A tarefa realizada por esta ferramenta seria o primeiro passo para a atualização/geração de requisitos a partir de casos de teste.

1.1. Trabalho Realizado

Este trabalho apresenta NLForSpec, uma ferramenta para geração de especificações formais a partir de casos de teste em linguagem natural. NLForSpec é parte de um projeto maior desenvolvido em parceria entre o CIn-UFPE e a Motorola, que tem como objetivo automatizar a seleção, geração e avaliação de casos de teste para aplicações de telefones móveis. Uma das principais tarefas desse projeto é atualizar automaticamente os requisitos a partir de casos de teste mais atuais.

A ferramenta NLForSpec é responsável por gerar especificações na linguagem formal CSP (*Communicating Sequential Processes*) [Hoare, 1985] [Roscoe *et al.*, 1997] a partir de descrições de casos de teste. Essas especificações serão utilizadas como entrada no processo de atualização ou geração de documentos de requisitos a partir de casos de teste mais atuais.

NLForSpec foi construída com base na arquitetura simbólica tradicional para interpretação de LN, e contém quatro bases de conhecimento (Léxico, Gramática de Casos, Ontologia e Base de Especificações CSP) e três módulos de processamento (POS-Tagging, Processamento Semântico e Geração de Casos de Teste Formais).

Com o objetivo de validar a ferramenta, foi implementado um protótipo em Java, seguindo a metodologia *eXtreme Programming* (XP) [Beck, 1999] de desenvolvimento de software. As bases de conhecimento do protótipo foram representadas em XML [W3C, 2006], favorecendo a manutenibilidade e a portabilidade. Foi realizado um estudo de caso para o domínio de descrições de casos de teste para aplicações de *Messaging* da Motorola, que revelou um desempenho satisfatório do protótipo.

1.2. Organização da Dissertação

Além deste capítulo de introdução, esta dissertação é composta por outros seis capítulos, descritos brevemente a seguir.

Capítulo 2: Interpretação de Linguagem Natural

Apresenta uma introdução à área de Interpretação de Linguagem Natural. São descritas as três abordagens para essa área, com ênfase na abordagem simbólica para Interpretação de LN. A arquitetura genérica em *pipeline* para essa abordagem é apresentada, bem como as Bases de Conhecimento e Módulos de Processamento que a compõem.

Capítulo 3: Geração de Modelos a partir de Linguagem Natural

Esse capítulo apresenta os trabalhos relacionados. São descritos alguns sistemas que geram modelos a partir de descrições em Linguagem Natural. Para cada um dos sistemas, são mostrados a arquitetura básica, técnicas utilizadas e resultados obtidos em experimentos.

Capítulo 4: *Test Research Project*

Detalha o *Test Research Project*, projeto no qual o trabalho apresentado nesta dissertação está inserido. As atividades desse projeto são apresentadas, mostrando onde a ferramenta NLForSpec se encaixa. Esse capítulo contém também uma breve discussão sobre Testes de Software e uma introdução à notação formal CSP.

Capítulo 5: A Ferramenta NLForSpec

A ferramenta NLForSpec é descrita nesse capítulo. A arquitetura de NLForSpec é apresentada e comparada à arquitetura simbólica tradicional para Interpretação de LN.

Em seguida, as Bases de Conhecimento e Módulos de Processamento que compõem a ferramenta NLForSpec são descritos em detalhes.

Capítulo 6: Protótipo e Experimentos

Apresenta o estudo de caso desenvolvido para validar a proposta do NLForSpec. Um protótipo foi implementado em Java e dois experimentos foram realizados tendo como entrada descrições de casos de teste para aplicações do domínio de *Messaging* da Motorola.

Capítulo 7: Conclusões

Apresenta as considerações finais sobre o trabalho desenvolvido, suas principais contribuições e algumas propostas de trabalhos futuros.

2. Interpretação de Linguagem Natural

O Processamento de Linguagem Natural (PLN) [Allen, 1995], uma das áreas de pesquisa da Inteligência Artificial, tem por objetivo produzir sistemas computacionais que se comuniquem com os seres humanos através de uma língua natural (*e.g.*, Português, Inglês, Espanhol, Alemão etc). PLN pode ser dividido em duas sub-áreas de trabalho: interpretação e geração de linguagem natural. A primeira delas, foco do presente trabalho, baseia-se em mecanismos que buscam "compreender" sentenças ou discurso em alguma LN, e tem por objetivo traduzi-los para uma representação que possa ser compreendida e utilizada pelo computador. Já na segunda, ocorre o oposto: o computador traduz uma representação computacional de um "conteúdo semântico" para texto em alguma língua natural.

A entrada de um sistema para interpretação de LN é um texto, e a saída é, geralmente, sua representação em alguma linguagem formal ou de representação do conhecimento. A escolha da linguagem interna para representar o texto depende da aplicação – por exemplo, em um sistema de interface em LN para Banco de Dados, a saída para uma pergunta do usuário deve ser uma consulta em alguma *query language*.

Estudos na área de PLN surgiram na década de 1950, e podem ser divididos em três abordagens: a primeira a surgir, e mais utilizada até hoje, é a abordagem simbólica; a segunda abordagem a ganhar destaque foi a abordagem empírica (estatística); por fim, a mais recente das três abordagens para Processamento de Linguagem Natural é a abordagem neural.

Este capítulo apresenta alguns detalhes sobre essas três abordagens, dando mais ênfase à abordagem simbólica, adotada no desenvolvimento do nosso trabalho.

2.1. Abordagens para PLN

Esta seção traz uma breve apresentação sobre as três abordagens para PLN descritas acima. São indicadas fontes bibliográficas onde é possível encontrar maiores detalhes sobre cada uma dessas abordagens.

2.1.1. Abordagem Simbólica

A abordagem simbólica para PLN baseia-se no uso de regras ou outros formalismos para representação explícita do conhecimento lingüístico. Desde os primeiros sistemas para tradução automática, na década de 1950, os métodos simbólicos vêm sendo usados no PLN. Nos primeiros 30 anos, grande parte do trabalho de PLN simbólico foi influenciado pela lingüística gerativa de Chomsky (1956).

Os primeiros sistemas para PLN, desenvolvidos até a década de 1980, podem ser caracterizados por possuir um único módulo de processamento e uma única base de conhecimento, que continha informação sintática e semântica específica para o domínio da aplicação. Esses sistemas, em geral, eram dedicados a um único domínio. Em conseqüência, o sistema tinha que ser praticamente refeito caso houvesse a necessidade de mudança de domínio [Barros & Robin, 1996].

Na década de 1980, diante da necessidade de se criar sistemas comercialmente viáveis, houve uma tendência de se buscar portabilidade em sistemas computacionais, de maneira geral. Seguindo essa tendência, os sistemas de PLN passaram então a se tornar modulares, tratando de domínios reais. Dessa maneira, chegou-se à arquitetura *pipeline* atual dos sistemas de PLN, que será descrita em detalhes na seção 2.2.

A principal vantagem da abordagem simbólica é que ela é capaz de tratar todos os níveis da análise lingüística de uma LN: morfológico, sintático (*parsers*), semântico, discurso e pragmática. Além disso, a abordagem simbólica favorece o uso de arquiteturas modulares, mais extensíveis e customizáveis.

Uma das limitações da abordagem simbólica é que o uso de regras pré-definidas e fixas para representação do conhecimento lingüístico, em geral, não permite a cobertura de todos os casos que podem ocorrer em um LN. Além disso, a robustez dos *parsers* depende das regras da gramática em uso e do léxico. Quando se aumenta a quantidade de regras da gramática, a fim de aumentar a sua abrangência, tem-se, como efeito

colateral, um aumento substancial de opções a seguir durante o *parsing* da sentença. Caso não exista um controle efetivo, o *parser* poderá gerar muitos caminhos que não levam à estrutura sintática da frase, o que aumenta o tempo de processamento, diminuindo a eficiência do *parser*.

2.1.2. Abordagem Empírica

A abordagem empírica para o Processamento de Linguagem Natural é baseada no uso de grandes quantidades de dados (corpora extensos), juntamente com procedimentos estatísticos para manipulação desses dados. Em comparação à abordagem simbólica, a abordagem empírica é direcionada aos dados, em oposição a regras, e direcionada à estatística, em oposição à lingüística.

A abordagem empírica também surgiu desde o início dos estudos em PLN, na década de 1950, juntamente com o aumento da capacidade de processamento dos computadores da época para corpora de texto razoavelmente grandes. Porém, na época, essa abordagem não teve destaque, em virtude da dominância dos estudos mais consolidados de Chomsky, que eram fortemente contrários ao uso de métodos empíricos na Lingüística. Apesar disso, podemos destacar a publicação do corpus de Brown [Francis & Kucera, 1964] e do Lancaster-Oslo-Bergen (LOB) corpus [Johansson et al., 1978]. O corpus de Brown é considerado o primeiro corpus moderno, de propósito geral e processável por computador. Esse corpus contém um milhão de palavras retiradas de textos em Inglês Americano das mais variadas fontes no ano de 1961. O corpus de Brown foi amplamente utilizado na época e influenciou o surgimento de outros corpora, como o LOB corpus, que utilizou o mesmo layout de Brown, só que para Inglês Britânico.

Apenas na década de 1990 é que a abordagem empírica voltou a ser vista realmente como uma alternativa à abordagem simbólica. Isso se deve primordialmente ao gigantesco aumento da capacidade de armazenamento de dados e da rapidez de processamento dos computadores modernos, juntamente com a facilidade para construção de grandes corpora através de processadores de textos e de leitores ópticos de caracteres.

Dentre as técnicas atuais mais utilizadas pela abordagem empírica, destacam-se os etiquetadores de textos *Part-of-Speech Tagging* (POS-Tagging). A etiquetagem

realizada por um POS-Tagging consiste na indicação da categoria gramatical de cada palavra de uma sentença, ou seja, consiste em "anotar" cada palavra com uma *tag* que indica a sua classe gramatical. Usualmente, a técnica empírica de POS-Tagging utiliza a probabilidade estatística de ocorrência de uma seqüência de *tags* para determinar a *tag* mais provável para uma determinada palavra, baseada no contexto das palavras vizinhas a ela na sentença (especialmente as palavras que a precedem). Um exemplo de algoritmo de *tagging* baseado nesse princípio é o desenvolvido por Brill (1995).

Uma vantagem da abordagem empírica é poder apresentar uma precisão melhor do que sistemas simbólicos em tarefas de *parsing*, uma vez que trabalha com exemplos, ao invés de regras. Contudo, para ter bom desempenho, é necessário um trabalho vasto e caro de marcação (etiquetagem) de grandes corpora. Já em relação às desvantagens, os sistemas empíricos para PLN, em geral, trabalham apenas nos níveis morfológico e sintático, não havendo processamento semântico, do discurso e nem pragmático.

2.1.3. Abordagem Neural

A abordagem conexionista para PLN (baseada em Redes Neurais Artificiais (RNA)) é mais recente, tendo surgido timidamente no início da década de 1980 com a publicação de artigos que utilizam RNAs para implementação de redes semânticas [Hinton, 1981], resolução anafórica [Reilly, 1984] e *parser* sintático [Fanty,1985] [Selman, 1985].

Apesar de estarem se tornando mais poderosos e sofisticados, os sistemas de PLN baseados na abordagem conexionista ainda não são capazes de prover funcionalidades semelhantes às exibidas pelos sistemas de PLN baseados na abordagem simbólica [Dyer, 1995].

Em relação às vantagens oferecidas pela abordagem neural, destaca-se a tolerância a ruído nos dados. Em sistemas de PLN desenvolvidos para aplicações em domínios reais, é comum a presença de erros nos dados, como erros sintáticos, por exemplo. Uma das críticas freqüentes aos sistemas de PLN simbólicos é a baixa tolerância a erros. A presença de ruídos nos dados de entrada que não foram previamente levados em conta pelo *designer* da aplicação irá causar um grau de mau funcionamento ao sistema maior do que o grau de severidade apresentado pela entrada corrompida [McKenna, 1994]. Já os sistemas baseados em RNAs são considerados menos frágeis nesse sentido, uma vez

que essas redes aproximam uma função a partir dos dados em uma curva de regressão que generaliza melhor a "forma" dos dados. Nesse sentido, as entradas corrompidas estarão a uma determinada distância desta curva; se o grau do ruído da entrada não for tão severo, a RNA irá obter a saída correspondente aos dados de treinamento distribuídos na curva, não acarretando uma saída inesperada.

Dentre as desvantagens apresentadas por sistemas de PLN conexionistas está a dificuldade de representação hierárquica das estruturas sintática e semântica das sentenças. Na abordagem simbólica, essas estruturas são simples de serem representadas, enquanto que na abordagem neural a representação é problemática. Uma outra desvantagem deve-se à dificuldade de analisar o comportamento das Redes Neurais. Como se sabe, em uma rede neural que foi treinada e possui boa capacidade de generalização e um comportamento desejado para uma determinada tarefa, os pesos das conexões determinam esse comportamento. Porém, uma análise direta desses pesos não mostra de forma clara como a rede neural resolve a tarefa.

2.2. Interpretação de LN – Abordagem Simbólica

Esta seção apresenta a Interpretação de Linguagem Natural Simbólica, a abordagem adotada no nosso trabalho. Veremos inicialmente a arquitetura genérica em *pipeline*, seguida das Bases de Conhecimento e dos Módulos de Processamento. Alguns exemplos usados nessa seção foram retirados de [Barros & Robin, 1996].

2.2.1. Arquitetura Genérica

Conforme dito na seção 2.1, os sistemas atuais de Interpretação de LN baseados na abordagem simbólica possuem uma arquitetura composta por módulos de processamento e bases de conhecimento.

Esses sistemas são, em geral, compostos por várias etapas de processamento (Morfológico, Sintático, Semântico, do Discurso e Pragmático) [Barros & Robin, 1996]. Assim, as arquiteturas desses sistemas são modulares, com cada nível de processamento sendo executado em um módulo distinto (figura 2.1). Esses módulos se comunicam

através da passagem de representações intermediárias do texto sob análise e estão representados por retângulos na figura 2.1.

As Bases de Conhecimento (BCs) são arquivos externos onde informações necessárias à interpretação dos textos são codificadas de forma declarativa. Na figura 2.1, as bases de conhecimento estão representadas por figuras elípticas.

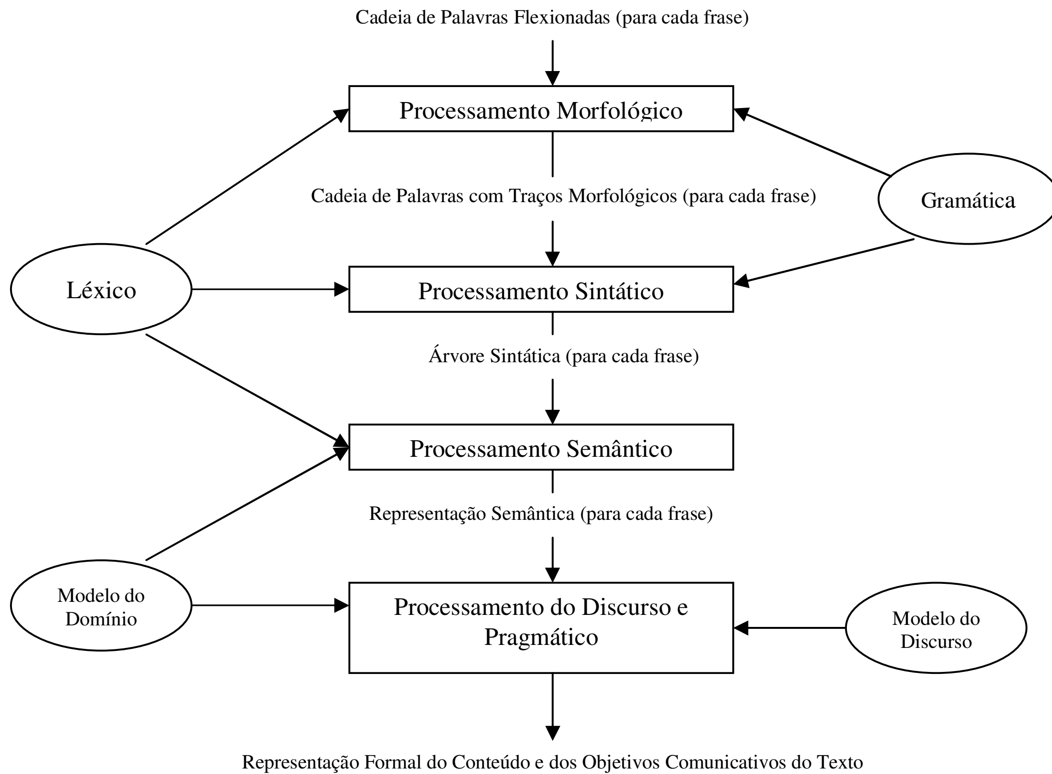


Figura 2.1: Arquitetura de Sistema para Interpretação de LN

2.2.2. Bases de Conhecimento

Léxico

O Léxico consiste em um dicionário que agrupa os termos utilizados pelo sistema para o processamento dos textos. Esses termos podem ser gerais ou pertencentes a um domínio específico. Cada termo do léxico pode estar associado às seguintes características:

- Morfológicas – conjugação dos verbos, inflexão dos substantivos e adjetivos etc;
- Sintáticas – categoria gramatical, transitividade, regência verbal etc;
- Semânticas – conceitos do domínio da aplicação que o termo pode expressar.

A figura 2.2 a seguir traz dois exemplos de termos de um léxico.

Tela	**exibiu**
<categoria>substantivo</categoria>	<categoria>verbo</categoria>
<genero>feminino</genero>	<tempo>pretérito-perfeito</tempo>
<numero>singular</numero>	<numero>singular</numero>
<dominio>itemVisualizavel</dominio>	<pessoa>3</pessoa>

Figura 2.2: Exemplo de Léxico

A figura 2.2 traz dois termos do léxico, um substantivo (tela) e um verbo (exibiu). Para "tela", além da informação de sua classe gramatical (substantivo), têm-se o gênero (feminino), o número (singular) e a sua classificação no domínio da aplicação (itemVisualizavel). Esta classificação "itemVisualizavel" indica que, para esse domínio, uma tela é um item que pode ser visualizado pelo usuário. Já para o verbo "exibiu", são encontradas informações referentes ao tempo verbal (pretérito perfeito), número (singular) e pessoa (terceira pessoa).

O Léxico pode ser representado de diferentes formas. Para o exemplo anterior, escolhemos uma representação em XML. A escolha dessa representação deve ser adequada ao formalismo utilizado na gramática e à Linguagem de Representação de Conhecimento utilizada nas outras BCs do sistema (*e.g.*, modelo do domínio, modelo do discurso etc). Além disso, alguns sistemas de PLN podem contar com mais de um léxico, sendo um mais geral, dado a priori, e os outros dedicados, montados pelo usuário com base no domínio específico de cada aplicação (*e.g.*, aplicações móveis, sistemas web etc).

Gramática

A gramática define, através de regras, quais são as cadeias de palavras válidas em uma língua. Essa verificação é feita através da aplicação dessas regras, e não através de uma lista exaustiva de frases, o que seria inviável, pois uma LN possui infinitas sentenças gramaticalmente corretas.

De acordo com Allen (1995), uma boa gramática deve apresentar as três seguintes características:

- **Generalidade** - conjunto das sentenças que a gramática analisa corretamente;
- **Seletividade -** conjunto de sentenças incorretas que a gramática analisa como problemáticas;
- **Simplicidade -** consiste na simplicidade das regras da gramática.

Uma gramática pode ser representada por diferentes tipos de formalismo. A escolha do tipo de formalismo a utilizar deve levar em conta a aplicação para a qual o sistema está sendo construído. Dentre os formalismos mais utilizados, destacam-se: Gramática de Constituintes Imediatos (*Phrase Structure Grammar - PSG*) [Gazdar *et al.*, 1985], Redes de Transição (RT) [Woods, 1970], PATR-II [Shieber, 1984] e Gramáticas de Unificação Funcional [Kay, 1984]. A seção 2.2.4 trará maiores detalhes sobre os tipos de gramáticas.

Modelo do Domínio

Esta BC pode ser vista como um modelo conceitual que armazena conhecimento a respeito das entidades, relações, eventos, lugares e datas do domínio em algum formalismo de representação do conhecimento. A figura 2.3 traz um exemplo de modelo do domínio na forma de ontologia [Chandrasekaran *et al.,* 1999]. Para os sistemas de PLN, uma ontologia pode ser vista como uma conceitualização dos termos do vocabulário, ou seja, a ontologia contém o conceito que cada termo do vocabulário representa em um determinado domínio.

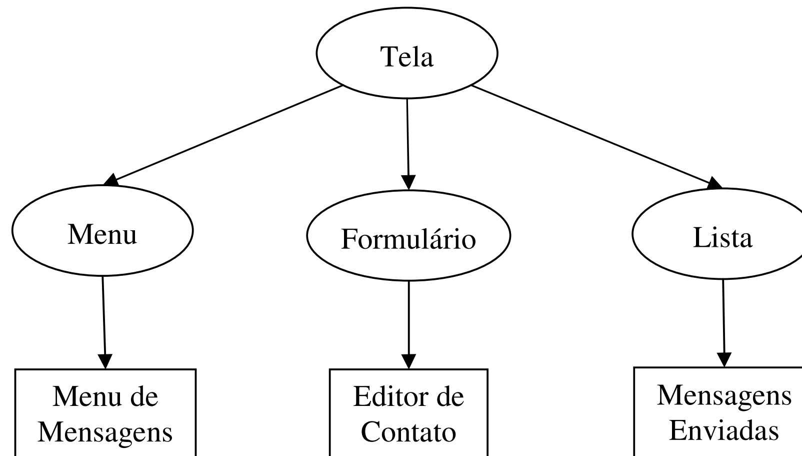


Figura 2.3: Exemplo de Ontologia

A figura 2.3 ilustra uma ontologia para o domínio de aplicações móveis, mais especificamente de aplicações para telefones celulares. Nesta ontologia, todas as relações são de especialização, ou seja, a relação entre “Tela” e “Menu” indica que “Menu” é um tipo de “Tela”, assim como “Formulário” e “Lista” também. Os termos representados por uma elipse indicam classes na ontologia, enquanto os termos representados por retângulos são as instâncias dessas classes. Dessa forma, “Mensagens Enviadas” pertence à classe Lista, que por sua vez é uma sub-classe de Tela.

Modelo do Discurso

O Modelo do Discurso fornece o contexto discursivo. Esse modelo é construído dinamicamente durante o processamento do discurso e tem como objetivo armazenar informações sobre as frases previamente processadas. O Modelo do Discurso tem papel central na interpretação de pronomes e dêiticos. Em geral, esse modelo consiste em uma pilha contendo as características sintáticas e semânticas das entidades já introduzidas no discurso. Essa pilha deve espelhar a estrutura do discurso, indicando onde começa e termina cada segmento.

2.2.3. Módulos de Processamento

Nesta seção, serão detalhados os quatro módulos de processamento que compõem a arquitetura básica de um sistema de PLN simbólico: Processamento Morfológico,

Processamento Sintático, Processamento Semântico, Processamento do Discurso e Pragmático.

Processamento Morfológico

O Processamento Morfológico é responsável por identificar palavras ou expressões das sentenças sob análise, sendo este processo auxiliado por delimitadores (pontuação e espaços em branco). As palavras identificadas são classificadas de acordo com suas características léxicas. Esse processamento é auxiliado pela base de conhecimento léxico (seção 2.2.2).

A técnica mais simples para realização do processamento morfológico consiste em uma lista de todas as possíveis palavras com suas características léxicas, o que torna a tarefa uma simples consulta a uma tabela. Porém, essa técnica só é eficiente para sistemas de PLN com um domínio restrito, pois as línguas naturais, em geral, apresentam processo de derivação de palavras muito complexo. Por exemplo, em espanhol, um verbo regular pode ocorrer em aproximadamente trinta e cinco formas. Assim, para aplicações livres de domínio, é mais eficiente manter regras que decomponham as palavras e, a partir dessa decomposição, obtenham suas características léxicas do que manter uma lista com todas as possíveis palavras.

Várias técnicas para processamento morfológico já foram propostas, porém, de acordo com Sproat (2000), apenas duas são amplamente utilizadas: técnicas baseadas em estados finitos [Beesley & Karttunen, 2003] e técnicas baseadas em unificação [Shieber, 1986].

Devido ao fato de a maioria dos fenômenos morfológicos poder ser descrita através de expressões regulares, é comum o uso de técnicas baseadas em estados finitos. Em particular, essas técnicas são úteis quando a estrutura das palavras é formada pela simples concatenação de morfemas. A figura 2.4 a seguir ilustra um modelo de estados finitos para a formação de adjetivos na língua inglesa.

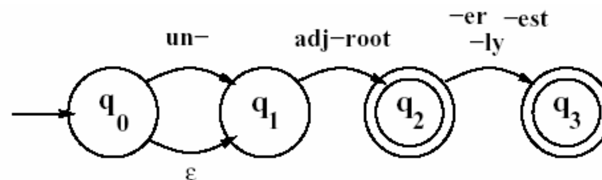


Figura 2.4: Modelo de estados finitos para adjetivos em inglês

Conforme pode ser observado, o modelo da figura 2.4 aceita corretamente adjetivos como, por exemplo: *clear, clearer, clearest, clearly, unclear* e *unclearly*. Porém, o modelo também aceita formas incorretas como *unbig, redly,* e *realest*. Para resolver esse tipo de problema, teria que ser criado um outro modelo de estados finitos, mais complexo, ilustrado na figura 2.5.

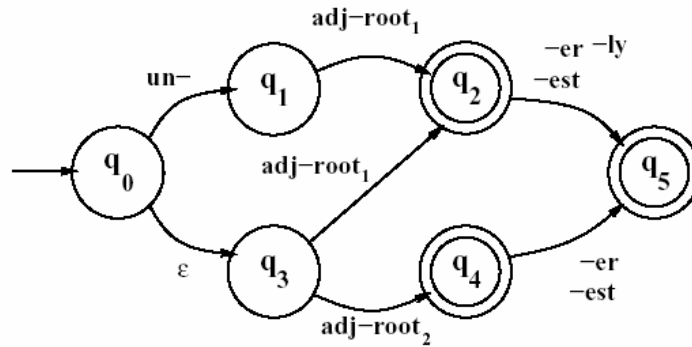


Figura 2.5: Modelo de estados finitos para adjetivos

Na figura 2.5, *adj-root*$_1$ representa os adjetivos que podem ocorrer com *un-* and *ly-* (*e.g., clear, happy, real*). Já *adj-root*$_2$ representa os adjetivos que não podem (*e.g, big, cool, red*). A informação sobre quais adjetivos seriam do tipo *adj-root*$_1$ e quais seriam do tipo *adj-root*$_2$ estaria armazenada no Léxico. A diferença entre o modelo da figura 2.4 e o modelo da figura 2.5 ilustra a principal limitação dos modelos baseados em estados finitos, a eficiência, pois o modelo da figura 2.5 se tornou mais complexo para ser capaz de representar corretamente mais adjetivos. Ou seja, à medida que os modelos de estados finitos tentam cobrir os variados casos de formações das palavras de uma determinada língua, vão ficando mais complexos, tornando-se, não raras vezes, muito grandes para serem armazenados e percorridos com eficiência.

Outra técnica baseada em estados finitos é a *Two-Level Morphology* [Koskenniemi, 1983], que consiste em representar a estrutura das palavras como uma correspondência entre dois níveis: léxico e de superfície. O nível léxico corresponde a morfemas abstratos e símbolos especiais que representam características morfológicas da palavra. O nível de superfície contém a forma atual de como a palavra é escrita. A figura 2.6 ilustra um exemplo de *Two-Level Morphology* para a palavra "*cats*".

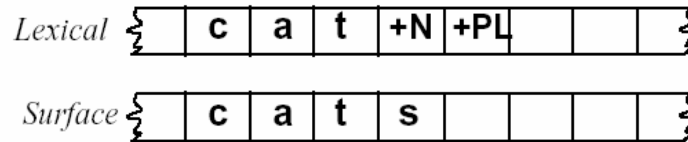
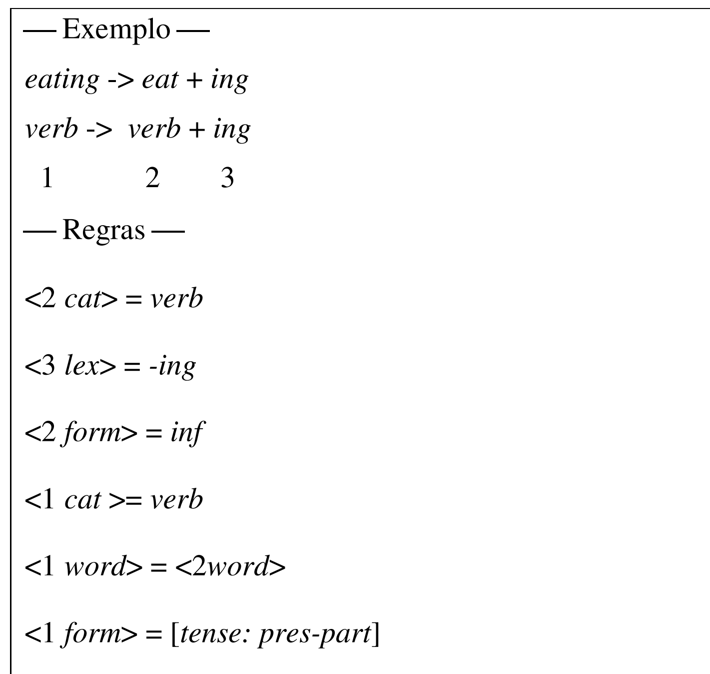


Figura 2.6: Exemplo de representação na *Two-Level Morphology*

Essa técnica pode ser implementada utilizando-se um *transducer* de estados finitos, *i.e.*, um autômato de estados finitos que reconhece pares de *strings*. A eficiência também é uma limitação para os *transducers* de estados finitos quando os modelos ficam muito grandes. Porém, existem algoritmos eficientes que tentam minimizar esse problema [Mohri, 1997].

As limitações encontradas nas técnicas baseadas em modelos de estados finitos levaram os pesquisadores a procurar novas alternativas. A mais popular das alternativas encontradas é a técnica baseada em unificação [Shieber, 1986], que consiste em utilizar *Two-Level Morphology* para modelar as alternativas fonéticas na formação das palavras, enquanto o processo de unificação é utilizado exclusivamente para tratar as possibilidades de combinações dos morfemas. Bear (1986) utiliza essa técnica no exemplo da figura 2.7, que é representado em PATR-II [Shieber, 1984], um formalismo bastante utilizado em sistemas de PLN.


— Exemplo —

eating -> *eat* + *ing*

verb -> *verb* + *ing*

1 2 3

— Regras —

<2 *cat*> = *verb*

<3 *lex*> = *-ing*

<2 *form*> = *inf*

<1 *cat* >= *verb*

<1 *word*> = <2*word*>

<1 *form*> = [*tense: pres-part*]

Figura 2.7: Exemplo de unificação

Para a figura 2.7, suponha que a palavra recebida como entrada é *eating* e que o analisador já identificou o infinitivo do verbo (*eat*) e está analisando agora a seqüência –*ing*. A primeira e a terceira regra casam com as características léxicas já identificadas (verbo e infinitivo, respectivamente) e a segunda regra casa com o –*ing*, também já identificado. Dessa forma, a quarta regra indicará que a nova palavra a ser reconhecida trata-se de um verbo, e a quinta regra, que esse verbo deriva do verbo já identificado. Por fim, a sexta regra indicará o tempo verbal como *pres-part* .

Processamento Sintático

O módulo de processamento sintático tem como objetivo obter uma representação da estrutura sintática da frase sob análise. Para isso, esse módulo utiliza o léxico e uma gramática, que determina quais são as estruturas válidas da LN em questão, associados a uma técnica de *parsing* (*i.e.*, um algoritmo que mapeia uma frase em sua estrutura sintática utilizando para isso a base de conhecimento léxico e a própria gramática). Nesta seção, serão abordados os formalismos gramaticais mais utilizados pelos sistemas de PLN.

Antes de iniciar a discussão sobre os formalismos gramaticais, é importante definir o conceito de Gramáticas Gerativas, fundamental para o entendimento destes formalismos. Uma gramática gerativa pode ser definida como um conjunto de regras que geram todas as cadeias válidas da linguagem descrita por estas regras, assinalando a cada cadeia a sua estrutura sintática [Barros & Robin, 1996].

O poder gerativo de uma gramática é determinado pelos tipos de cadeias que tal gramática pode gerar. Essas gramáticas foram classificadas em quatro tipos por Chomsky (1956), conforme ilustrado no quadro 2.1 a seguir.

Quadro 2.1: Classificação das Gramáticas Gerativas

Tipo	Nome
3	Gramáticas regulares
2	Gramáticas livres-de-contexto
1	Gramáticas sensíveis ao contexto
0	Sistemas de reescrita geral

As gramáticas do tipo 3 são subconjuntos das gramáticas do tipo 2, que por sua vez são subconjuntos das do tipo 1, que são subconjuntos das do tipo 0. As gramáticas

do tipo 3 são mais restritas, apresentando menor capacidade gerativa que as do tipo 2, 1 e 0, em ordem crescente. Ou seja, o poder gerativo de uma gramática é tão maior quanto menos restritivas forem suas regras.

As gramáticas do tipo 3, com poder gerativo fraco, não são capazes de definir linguagens simples como cadeias do tipo X^nY^n, o que pode ser facilmente obtido pelas gramáticas do tipo 2, com apenas duas regras recursivas: $F \rightarrow XFY$ e $F \rightarrow XY$. Uma característica das gramáticas do tipo 3 é que elas são capazes apenas de reconhecer se determinada cadeia pertence ou não à linguagem descrita. Já as gramáticas do tipo 2, com poder gerativo maior, também associam uma estrutura a cada cadeia da linguagem.

As gramáticas do tipo 1 e 0 apresentam dificuldades na determinação da estrutura de determinadas cadeias, uma vez que admitem que o lado esquerdo das regras de produção seja composto por mais de um símbolo, como nas cadeias abaixo:

- Tipo 1: AuB → ACB
- Tipo 0: ABCD → EF

Veremos agora alguns dos formalismos gramaticais mais utilizados pelos sistemas de PLN. O primeiro a ser abordado é a Gramática de Constituintes Imediatos (*Phrase Structure Grammar - PSG*) [Gazdar *et al.*, 1985]. Essas gramáticas são livres-de-contexto cujos símbolos das regras de produção são categorias sintáticas (não-terminais) ou palavras (terminais). A categoria da esquerda pode ser expandida (reescrita) pela(s) categoria(s) do lado direito da regra durante a derivação da estrutura da frase sob análise. Este tipo de gramática provê a estrutura sintática das frases em termos dos seus constituintes, especificando a hierarquia entre eles. A figura 2.8 a seguir traz exemplo de uma pequena gramática capaz de gerar e reconhecer a frase "A menina quebrou o jarro azul".

F → SN SV	Det → a
SN → Det Subs	Subs → menina
SN → Det Subs Adj	Subs → jarro
SN → Subs Adj	Adj → azul
SV → V SN	V → quebrou

Figura 2.8: Exemplo de Gramática de Constituintes Imediatos

No exemplo da figura 2.8, pode ser observada a presença de dez regras de produção. A primeira regra define que uma frase (F) é formada por um Sintagma Nominal (SN) e por um Sintagma Verbal (SV), enquanto a quinta regra define que um Sintagma Verbal é formado por um Verbo (V) e por um Sintagma Nominal. As cinco últimas regras de produção contêm símbolos terminais do lado direito (a, menina, jarro, azul, quebrou).

O grande interesse do PLN pelas Gramáticas de Constituintes Imediatos deve-se ao fato de essas gramáticas, além de apresentarem um bom poder gerativo, terem sido tradicionalmente utilizadas com sucesso na Ciência da Computação, havendo inúmeros algoritmos eficientes para reconhecer linguagens livres-de-contexto.

Porém, apesar de apresentarem um bom poder gerativo e de serem capazes de determinar a estrutura sintática das sentenças, as Gramáticas de Constituintes Imediatos não consideram os traços sintáticos dos constituintes que formam a sentença. Por exemplo, as sentenças da figura 2.9 serão todas consideradas sintaticamente corretas, apesar de as duas últimas apresentarem erros de concordância.

O menino anda Os meninos andam O menino falam Os menino anda

Figura 2.9: Exemplo de sentenças

Uma das maneiras de se verificar a concordância de gênero e número entre os constituintes de uma frase é através do formalismo PATR-II [Shieber, 1984]. Neste caso, as regras da gramática trazem traços sintáticos associados, que são espécies de variáveis que serão instanciadas de acordo com as características das palavras. A figura 2.10 traz exemplo da mesma gramática da figura 2.8, no formalismo PATRII.

```
Gramática                                    Léxico

F → SN SV                                    a
    <SN número> = <SV número>                    <categoria> = Determinante
    <SN pessoa> = <SV pessoa>                    <gênero> = fem
    <SN caso> = nominativo                       <número> = sing
    <SV forma> = flexionado                  menina
SN → Det Subs                                    <categoria> = Substantivo
    <Det gênero> = <Subs gênero>                 <gênero> = fem
    <Det número> = <Subs número>                 <número> = sing
SN → Det Subs Adj                            jarro
    <Det gênero> = <Subs gênero>                 <categoria> = Substantivo
    <Det número> = <Subs número>                 <gênero> = fem
    <Subs gênero> = <Adj gênero>                 <número> = sing
    <Subs número> = <Adj número>             azul
SV → V SN (verbo transitivo direto)              <categoria> = Adjetivo
    <SN caso> = objeto direto                    <número> = sing
                                             quebrou
                                                 <cat> = Verbo
                                                 <tempo> = pass
                                                 <número> = sing
                                                 <pessoa> = 3
                                                 <arg1> = SN
```

Figura 2.10: Gramática no formalismo PATRII

Durante o *parsing* da frase, os valores dos traços sintáticos das palavras, fornecidos pelo léxico, são utilizados para fixar os valores das variáveis associadas às regras da gramática. Assim, é possível verificar a corretude gramatical da frase.

Outro formalismo para representação gramatical bastante utilizado pelos sistemas de PLN são as Redes de Transição (RT) [Woods, 1970]. As RT consistem em nós, que representam os estados, e arcos, que representam as categorias gramaticais. Essas redes tanto reconhecem se uma determinada cadeia pertence a uma linguagem, como também provêem sua estrutura sintática.

As Redes de Transição estão divididas em três tipos, em que cada novo tipo é um refinamento do anterior no que diz respeito ao seu poder gerativo. Redes de Transição Simples (RTS) são o primeiro desses tipos. RTS são autômatos finitos com estados iniciais e finais. Essas redes não aceitam arcos circulares que levem a um estado anterior ao ponto de partida do arco, podendo conter, no máximo, arcos circulares em que o nó de origem é o mesmo de destino. As RTS podem ser representadas por regras de produção como as usadas nas gramáticas PSG. A principal limitação das RTS é a sua capacidade gerativa, que é igual à das gramáticas regulares (tipo 3).

Já o segundo tipo de RT, as Redes de Transição Recursivas (RTR), apresenta a mesma capacidade gerativa das gramáticas livres-de-contexto (tipo 2). As RTR são

Redes de Transição Simples cujos rótulos podem conter outra RT da gramática. O efeito resultante de se percorrerem arcos que "transferem" o processamento para uma sub-rede é obtido nas gramáticas livres-de-contexto através das chamadas recursivas. A figura 2.11 traz exemplo de uma RTR.

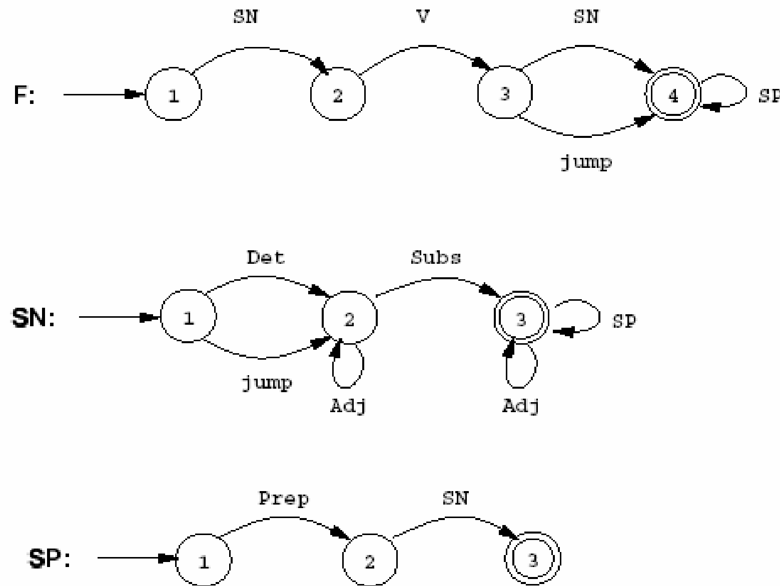


Figura 2.11: Rede de Transição Recursiva

A principal limitação das RTR é a incapacidade de verificação de concordância entre os constituintes da frase. Por exemplo, frases gramaticalmente incorretas como "A menino quebraram o jarro preta" seriam aceitas pela rede da figura 2.11.

O terceiro tipo de RT é a Rede de Transição Aumentada (RTA), que pode ser classificada como RTR acrescida de condições e ações associadas aos arcos. Para que um arco seja atravessado, a condição associada a ele deve ser verdadeira. E a ação associada ao arco é responsável pela construção de uma estrutura descritiva do constituinte sob análise. Por exemplo, podem ser associadas condições e ações às RTRs da figura 2.11, transformando-as numa RTA. O quadro 2.2 ilustra um exemplo de condições e ações associadas ao sintagma nominal (SN) da figura 2.11.

Quadro 2.2: Condições e Ações de uma RTA

Arco	Condição	Ação
1-2	-	Det := determinante Conc := conc.det

2-3	Conc ∩ conc.det	Nucleo := substantivo Conc := conc.det ∩ conc.subs
1-3	-	Subs := substantivo Conc := conc.subs

No quadro 2.2, temos os seguintes registradores utilizados pela RTA:

- **Det** – registra a existência de um determinante na cadeia sob análise;
- **Conc** – guarda a concordância do sintagma: gênero, número e pessoa;
- **Núcleo** – guarda a categoria do constituinte mais importante do sintagma, aquele que determina o tipo do sintagma;
- **Subs** – registra a existência de um substantivo no sintagma.

Para se atravessar o arco 1-2 da cadeia do Sintagma Nominal da RTR da figura 2.11, a condição é que a cadeia se inicie com um determinante. No quadro 2.2 observa-se que as ações associadas a esse arco são o registro da existência de um determinante (*Det := determinante*) e o armazenamento de sua concordância (*Conc := conc.det*). Já para o arco 2-3, a condição *default* do arco, que o constituinte seja um substantivo, é aumentada pela condição extra de que a concordância de tal substantivo deve ser a mesma do determinante já encontrado (*Conc ∩ conc.det*). A ação associada a este arco é registrar o núcleo (*Nucleo := substantivo*) e armazenar a concordância (*Conc := conc.det ∩ conc.subs*). E, finalmente, a condição para que o arco 1-3 seja atravessado é que a cadeia se inicie com um substantivo, sem a presença de um determinante. As ações associadas são o registro da existência de um substantivo (*Subs := substantivo*) e o armazenamento de sua concordância (*Conc := conc.subs*).

Dentre os três tipos de RTs vistos, as RTAs são o mais utilizado, por oferecerem, com suas aumentações, mais flexibilidade no tratamento das exceções encontradas nas LNs. Isto se deve ao fato de as condições associadas aos arcos poderem ser vistas como um contexto associado, tornando essas redes equivalentes às gramáticas sensíveis ao contexto. Porém, apesar de flexíveis, as RTAs perdem em padronização, pois as condições e ações, em muitos casos, dependem do domínio para o qual o sistema foi desenvolvido, comprometendo sua portabilidade para outro domínio. Além disso, as RTAs são procedimentais, ou seja, as ações são procedimentos que serão executados

caso determinado arco seja atravessado, o que prejudica em muito a clareza descritiva dessas redes.

Além dos formalismos explicados nessa seção, podemos destacar também as gramáticas *Definite Clause Grammar* [Pereira & Warren, 1980], *Lexical-Functional Grammar* [Bresnan, 1982] e *Categorial Unification Grammar* [Uszkoreit, 1986].

Processamento Semântico

O processamento semântico é responsável por interpretar o significado da sentença sob análise, pela interpretação de seus constituintes. Existem vários formalismos utilizados na Interpretação Semântica. Nesta seção, três dos mais usados serão discutidos: Gramáticas de Casos, Redes Semânticas e Lógica de Primeira Ordem.

As Gramáticas de Casos [Fillmore, 1968], diferentemente das gramáticas usadas no processamento sintático, não utilizam a noção de sujeito, objeto etc. Aqui, papéis temáticos, ou "casos", são atribuídos aos constituintes da frase.

Por exemplo, na primeira das sentenças da figura 2.12, intuitivamente, pode-se dizer que "João" é o ator da ação, "a janela" é o objeto e "o martelo" é o instrumento utilizado no ato de quebrar a janela. Na segunda sentença, "o martelo" também é o instrumento utilizado e "a janela" é o objeto. Ou seja, é possível estabelecer relações (papéis temáticos) tais como AGENTE, TEMA e INSTRUMENTO para capturar essa intuição. A figura 2.13 traz alguns exemplos de papéis temáticos definidos por Allen (1995).

João quebrou a janela com um martelo.

O martelo quebrou a janela.

Figura 2.12: Exemplo de sentenças

Agente – o ser animado que causa a ação

Tema – a coisa afetada pela ação ou sobre cuja existência se discute

Beneficiário – a pessoa para quem a ação é realizada

Locação – o lugar onde a ação ocorre

Fonte – locação de origem

Destinação – locação final

Possuidor – possuidor do tema

Recipiente – possuidor final

Figura 2.13: Papéis Temáticos

Na Gramática de Casos, o verbo, o constituinte central da frase, determina os papéis temáticos que servem de argumento para ele. Por exemplo, o verbo "botar" teria a seguinte entrada lexical (figura 2.14):

botar, Verbo – SN, SP (*argumentos: agente, tema, locação*)

Figura 2.14: Exemplo de entrada lexical do verbo "botar"

Essa entrada indica que o verbo "botar" contém os papéis temáticos "agente", "tema" e "locação". O quadro 2.3 a seguir traz um exemplo de um resultado dessa classificação.

Quadro 2.3: Exemplo de classificação de papéis temáticos

Papel Temático	agente		tema	locação
Papel Sintático	sujeito	núcleo do predicado	obj. direto	adjunto adv. lugar
Categoria Sintática	pronome	verbo	SN	SP
Frase	Eu	botei	o livro	sobre a mesa

É importante salientar que os papéis temáticos devem ser independentes da estrutura de superfície da frase, por serem vistos como relações entre os constituintes da estrutura profunda da frase. Ou seja, frases com significados diferentes devem ter atribuições dos papéis temáticos diferentes, e frases com o mesmo significado devem resultar na mesma atribuição de papéis. As frases da figura 2.15 exemplificam isso.

1. Maria entregou o livro novo a João.
2. Maria entregou a João o livro novo.
3. O livro novo foi entregue a João por Maria.

Figura 2.15: Exemplo de frases com o mesmo significado

As três frases da figura 2.15 têm o mesmo significado, apesar de possuírem estruturas de superfícies diferentes. Nessas frases, o papel temático de "Maria" é agente, o de "livro" é tema e o de "João" é beneficiário, apesar de, por exemplo, na frase 1 "Maria" ser o sujeito e na frase 3 ser "O livro novo".

Segundo Efe & Ng (1987), Gramática de Casos funciona muito bem para especificar sentenças isoladas, identificando as relações semânticas entre os componentes da sentença. Além disso, a gramática de casos é ideal para domínios

restritos, i.e., ações, vocabulário e papéis temáticos restritos. Porém, devido ao fato de conter informação do domínio para o qual foi desenvolvida, o seu reuso para outros domínios tende a ser difícil.

Outro formalismo utilizado para representação semântica são as Redes Semânticas [Sowa, 1991], que consistem em uma estrutura de nós e arcos em que os nós representam entidades do domínio e os arcos representam relações semânticas entre essas entidades. Os nós em uma Rede Semântica podem representar tipos, também chamados de conceitos genéricos, e instâncias, também conhecidas como conceitos individuais. A figura 2.16 traz exemplo de uma rede semântica.

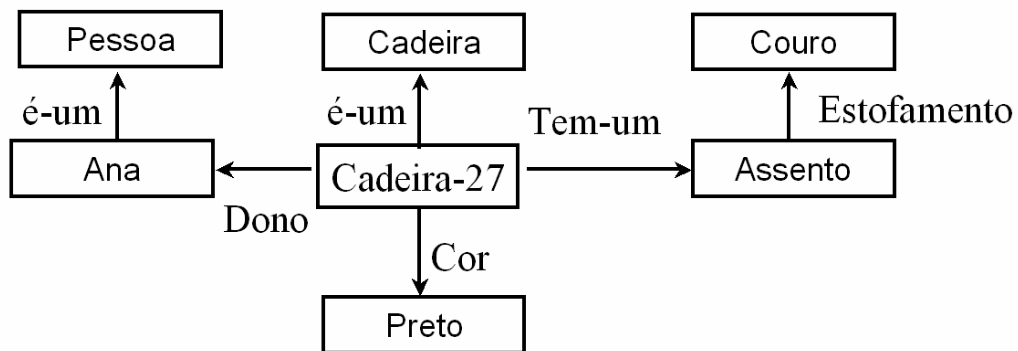


Figura 2.16: Exemplo de Rede Semântica

Na rede semântica da figura 2.16 pode-se observar a presença de vários relacionamentos entre os nós. Por exemplo, o relacionamento "Estofamento" liga "Assento" ao seu tipo de estofamento, "Couro". Há também a presença de uma relação especial, "é-um" (IS-A), que indica que uma entidade pertence a uma classe mais alta na hierarquia, e, por conseguinte, herda as características dessa classe.

A principal limitação das redes semânticas é a dificuldade de implementação de algoritmos para percorrer essas redes de forma eficiente. Outro aspecto que pode ser identificado é a falta de padronização entre as redes semânticas, o que dificulta a portabilidade desta forma de representação de conhecimento entre sistemas.

A Lógica de Primeira Ordem (LPO) é outro formalismo para representação semântica bastante utilizado. Quando esse formalismo é utilizado no PLN, as palavras do léxico são as constantes, que podem ser de dois tipos. As constantes que descrevem objetos (concretos ou abstratos) são chamadas de termos. Já as constantes que descrevem relações e propriedades são chamadas de predicados. Uma proposição é

formada por um predicado seguido de um número de termos, que são seus argumentos. A figura 2.17 traz exemplos de frases e suas respectivas fórmulas lógicas.

1. Frase: Maria caiu.
   Fórmula Lógica: caiu(maria)
2. Frase: Ela entregou o livro ao editor.
   Fórmula Lógica: entregar(ela, livro, editor)
3. Frase: André comeu e bebeu
   Fórmula Lógica: comeu(andre) ∧ bebeu(andre)

Figura 2.17: Frases e Fórmulas Lógicas

Na fórmula lógica da primeira frase da figura 2.17, podem ser observados o predicado "caiu" e o termo "maria". Já na terceira, temos os predicados "comeu" e "bebeu", o termo "andre" e o operador lógico de conjunção (∧). É possível estabelecer um paralelo entre a representação semântica em LPO e as redes semânticas: os nós das redes semânticas são representados por termos e os arcos são representados pelos predicados.

Uma das razões que tornou a LPO popular para representação semântica deve-se à capacidade de deduzir se uma frase é verdadeira ou falsa dentro do domínio, ou seja, de deduzir se a fórmula lógica de uma frase é ou não conseqüência das fórmulas lógicas armazenadas anteriormente na base de conhecimento. Porém, esse processo de inferência pode ser computacionalmente caro. Na verdade, não há nem mesmo garantia de que um processo desse tipo irá terminar. Além disso, a representação semântica em LPO não deixa clara a definição de classes de objetos e hereditariedade, que são fundamentais em algumas aplicações.

Processamento do Discurso e Pragmático

A Análise do Discurso estuda os princípios que governam a produção de seqüências estruturadas de frases (discurso escrito ou falado). O discurso é formado por segmentos, que são unidades lingüísticas que contêm uma ou mais frases consecutivas e que tratam do mesmo assunto (o foco daquele trecho do discurso).

O Processamento do Discurso é especialmente importante para sistemas de PLN que têm como objetivo identificar qual a influência de uma ou mais frases na interpretação das frases subseqüentes. Esse processamento é fundamental para

identificação de pronomes (*e.g.*, eu, você, ela, este, aquela) e dêiticos (*e.g.*, hoje, aqui, agora).

Existe uma grande variedade de trabalhos para o tratamento do discurso. Porém, em virtude de essa etapa de processamento não ser foco do nosso trabalho, apenas citaremos algumas contribuições mais relevantes, como [Hobbs, 1979] [Hobbs, 1996] [Mann & Thompson, 1987] [Groz & Sidner, 1986] [Moore & Pollack, 1992].

Já a Pragmática estuda os enunciados (frases com seu significado) no contexto do discurso sob o ponto de vista dos interlocutores, e tem como preocupação central a análise dos objetivos da comunicação (interação social). O processamento pragmático é necessário em qualquer tarefa de PLN que requeira a análise das intenções dos participantes no discurso. Como esse também não é o foco do nosso trabalho, as teorias existentes para o processamento pragmático não serão detalhadas. Porém, dentre as teorias, destacamos os Atos da Fala [Searle, 1971], que definem as atividades desenvolvidas pelos falantes de uma língua enquanto fazem uso dela.

2.3. Considerações Finais

Neste capítulo foram apresentadas as três principais abordagens para Processamento de Linguagem Natural: simbólica, empírica e neural. Em seguida, a abordagem utilizada em nosso trabalho, a simbólica, foi discutida em detalhes. As quatro bases de conhecimento (Léxico, Gramática, Modelo do Domínio e Modelo do Discurso) e os quatro módulos de processamento (Morfológico, Sintático, Semântico e do Discurso e Pragmático) que compõem a arquitetura simbólica *pipeline* tradicional de um sistema de interpretação de linguagem natural foram detalhados.

Os conceitos apresentados neste capítulo foram utilizados como base para a definição da ferramenta proposta neste trabalho e serão retomados durante toda a dissertação. No capítulo 3 a seguir, serão apresentados os trabalhos relacionados: sistemas de geração de modelos a partir de descrições em Linguagem Natural.

3. Geração de Modelos a partir de Linguagem Natural

Neste capítulo, serão apresentados alguns sistemas de geração de modelos a partir de descrições em Linguagem Natural. Veremos inicialmente sistemas que geram modelos formais a partir de LN, que é o foco do nosso trabalho. A seguir, veremos também alguns sistemas que geram outros tipos de modelos, como os Orientados a Objetos. Isto se deve ao fato de termos encontrado na literatura disponível apenas dois sistemas que geram modelos formais a partir de LN.

Este capítulo contém a descrição de quatro sistemas, dos quais dois geram modelos formais (seções 3.1 e 3.2) e dois geram modelos não-formais (seções 3.3 e 3.4). Para cada um dos sistemas apresentados, serão mostrados a arquitetura básica, técnicas utilizadas para o PLN e resultados obtidos em experimentos. Por fim, serão apresentadas as considerações finais a respeito destes sistemas.

3.1. Sistema para Geração de Especificações Formais a partir de Documentos de Requisitos

O sistema proposto por Lee & Bryant (2002) tem como objetivo gerar, a partir de documentos de requisitos em Linguagem Natural, especificações em uma linguagem formal orientada a objetos. A partir dessa especificação formal, podem ser gerados códigos na linguagem Java ou diagramas UML. A figura 3.1, a seguir, ilustra a arquitetura do sistema.

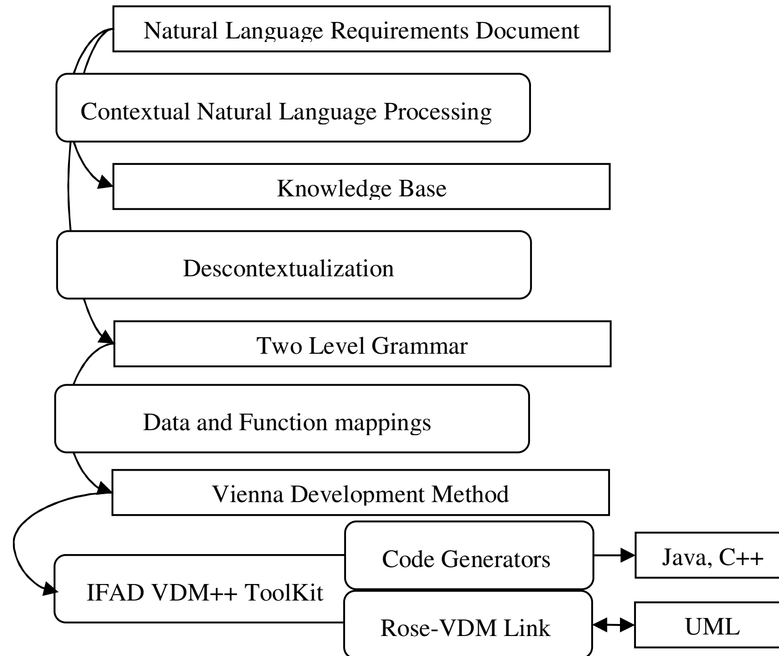


Figura 3.1: Arquitetura do Sistema [Lee & Bryant, 2002]

Conforme pode ser observado na figura 3.1, o primeiro módulo de processamento, o "*Contextual Natural Language Processing*", recebe como entrada documentos de requisitos em LN. A figura 3.2 a seguir ilustra um trecho de um documento de requisito que será utilizado como exemplo de entrada.

Bank keeps list of accounts. It verifies ID and PIN giving the balance and updates the balance with ID.
An account has three data fields ; ID, PIN, and balance. ID and PIN are integers and balance is a real number.
ATM has 3 service types; withdraw, deposit, and balance check. For each service first it verifies ID and PIN from the bank giving the balance.
ATM withdraws an amount with ID and PIN giving the balance in the following sequence. If the amount is less than or equal to the balance then it decreases the balance by the amount. And then it updates the balance in the bank with ID. ATM deposits an amount with ID and PIN giving the balance in the relieving order. It increases the balance by amount and then updates the balance in the bank with ID. ATN checks the balance with ID and PIN giving the balance.

Figura 3.2: Trecho de Documento de Requisito [Lee & Bryant, 2002]

Esses documentos de requisitos são então convertidos para um arquivo XML (*Extensible Markup Language*), sendo estruturados em seções, que podem conter subseções, que, por sua vez, agrupam parágrafos, que são formados por sentenças. Essa estruturação em arquivo XML é necessária para padronizar os documentos de requisitos recebidos como entrada, pois esses documentos podem ter formatos variados. A figura 3.3 ilustra um exemplo de um trecho do texto da figura 3.2 em formato XML.

```
<p>
    <s>ATM has 3 services types; withdraw, deposit,
       and balance check.</s>
    <s>For each service first it verifies ID and
       PIN from the bank giving the balance.</s>
</p>
<p>
    <s>ATM withdraws an amount with ID and PIN giving
       the balance in the following sequence.</s>
    <s>If the amount is less than or equal to the balance
       then it decreases the balance by the amount.</s>
    <s>And then it updates the balance in the bank
       with ID.</s>
</p>
```

Figura 3.3: Trecho do documento em XML

Depois de formatado em XML, cada sentença do documento é processada por um etiquetador (*Part-Of-Speech Tagger*), que identifica a classe gramatical de cada palavra (e.g. substantivo, verbo, adjetivo). Em seguida, é realizado um *parser bottom-up* para identificar o papel sintático das palavras que constituem a sentença (*e.g.*, sujeito, objeto, complemento). O resultado dessa análise sintática em dois passos é ilustrado na figura 3.4 a seguir.

Bank keeps list of accounts
Part of speech: bank(noun) keeps(verb) accounts_list(noun)
Part of sentence: (subject verb object)

It verifies ID and PIN giving the balance and updates the balance with ID
Part of speech: it(pronoun) verifies(verb) ID and PIN(noun) giving(verb) the(article) balance(noun)
Part of sentence: (subject verb object helping: (verb adjective object))
Part of speech: it(pronoun) updates(verb) the(article) balance(noun) with(preposition) ID(noun)
Part of sentence: (subject verb object) …

Figura 3.4: Resultado da Análise Sintática

Após a análise sintática, são realizadas etapas de análise semântica e resolução anafórica. Em seguida, é realizada a construção de uma base de conhecimento de

"contextos". Essa base de conhecimento consiste em uma estrutura no formato de árvore em que cada sentença é armazenada de acordo com o contexto ao qual pertence. Para criação dessa base, são utilizadas a estrutura XML do documento (seção, subseção e parágrafo) e a informação semântica de cada sentença. A figura 3.5 traz um trecho dessa base de conhecimento, em formato XML (a *tag* 'c' indica "contexto" e a *tag* 's' indica "sentença").

```
<c name="ATM">
 <c>
    <s>ATM has 3 services types; withdraw, deposit,
       and balance check</s>
    <c>
      <s>first ATM verifies ID and PIN from the bank
         for each service giving balance</s>
    </c>
    <c>
      <s>ATM withdraws amount with ID and PIN giving
         balance in following sequence.</s>
      <c>
        <s>if amount is less than or equal to balance
           then ATM decreases balance by amount.</s>
        <s>then ATM updates balance in the bank
           with ID.</s>
      </c>
      ...
    </c>
  </c>
</c>
```

Figura 3.5: Base de Conhecimento de Contextos

A BC de contexto é então traduzida para uma representação que usa a linguagem *Two-Level Grammar* (TLG) [Bryant & Lee, 2002]. TLG é uma linguagem de especificação de softwares orientada a objetos. A tradução da BC de contextos para TLG ocorre da seguinte forma: a raiz de cada árvore de contexto se transforma em uma classe, e o corpo de cada classe é formado pelas sentenças contidas nos sub-contextos da raiz. A Figura 3.6 traz a especificação TLG para a base de conhecimento de contextos ilustrada na figura 3.5.

```
class Bank.
    Accounts_List :: AccountList.
    ID :: Integer.
    PIN :: Integer.
    Balance :: Float.
    verify ID and PIN giving  Balance.
    Update Balance with ID
end class.
class Account.
    ID :: Integer.
    PIN :: Integer.
    Balance :: Float.
end class.
class ATM.
    Balance :: Float.
    Amount :: Float.
    ID :: Integer.
    PIN :: Integer.
   withdraw Amount with ID and PIN giving Balance
      verify ID and PIN from Bank giving Balance:
      if Amount <= Balance then
        Balance := (Balance - Amount),
        update Balance in Bank with ID
     endif
….
end class.
```

Figura 3.6: Especificação TLG

Por fim, a especificação TLG é traduzida para a linguagem formal *Vienna Development Method meta-language*, VDM++ [Dürr & Katwijk, 1992]. A linguagem VDM++ foi selecionada por sua semelhança estrutural com TLG, e também pelas ferramentas disponíveis para geração de código e de diagramas de análise a partir de especificações VDM++. Uma descrição detalhada da tradução de especificações TLG para VDM++ pode ser encontrada em [Bryant & Lee, 2002]. A figura 3.7 traz um exemplo da especificação VDM++ obtida a partir da especificação TLG da figura 3.6.

```
class Bank
instance variables
    private Accounts_List : seq of Account := []
operations
    public verify : int * int ==> real
    verify (ID, PIN) =
      (dcl Balance : real := 0;
           return Balance);
     public update : real * int ==> {}
         update (Balance, ID) ==
             return
end Bank
class Account
instance variables
   private ID : int;
   private PIN : int;
   private Balance : real
end Account
class ATM
instance variables
   private CBank : Bank : new Bank()
operations
     public withdraw : real * int * int ==> real
     withdraw(Amount, ID, PIN) ==
      dcl Balance : real;
      Balance := CBank.verify(ID,PIN);
      if Amount <= Balance then
         (Balance := (Balance - Amount);
          CBank.update(Balance, ID));
      return Balance);
...
end ATM
```

Figura 3.7: Especificação VDM++

A partir da especificação VDM++, podem ser gerados código Java ou C++, utilizando o gerador de código presente na ferramenta *VDM Toolkit* [IFAD, 200]. A geração de modelos UML (Unified Modeling Language) também é provida por essa ferramenta.

Uma questão em aberto nesse trabalho é a avaliação da qualidade das traduções obtidas. Os autores afirmam que a ferramenta funciona corretamente para o exemplo mostrado, porém não fornecem detalhes a respeito de experimentos com outros documentos de requisitos. De fato, uma análise mais detalhada do desempenho do sistema se faz necessária. Algumas dúvidas que podem surgir são:

(1) O documento de requisitos pode ser escrito livremente, ou deve seguir algum padrão de estrutura lingüística (gramatical e/ou textual)?

(2) Esse sistema é capaz de gerar código para qualquer tipo de sistema descrito por requisitos em LN?

(3) O sistema é livre de domínio e vocabulário?

3.2. Sistema para Geração de Especificações Algébricas

Ishihara *et al.* (1992) propõem um sistema para geração de especificações algébricas a partir de especificações de protocolos de comunicação descritos em Linguagem Natural. As especificações em LN definem seqüências de ações que podem ser executadas por um programa.

Uma sentença de uma especificação de protocolo tipicamente descreve uma ação que o programa deve executar e o estado do programa. Como exemplo, temos a sentença "*If Vsc is false, V(A) is set equal to V(M)*" [Ishihara *et al.*, 1992], onde "*Vsc is false*" descreve o estado e "*V(A) is set equal to V(M)*" descreve a ação. Essas sentenças são traduzidas para uma linguagem de especificação algébrica desenvolvida pelos próprios autores, chamada de *Algebraic Specification Language* (ASL) [Kasami *et al*, 1986].

O processo de geração de especificações proposto assume que uma especificação é formada por um conjunto de parágrafos, e que não há dependência contextual entre parágrafos, ou seja, no sistema em questão, cada parágrafo é processado isoladamente. Esse processo de geração possui três etapas, descritas a seguir:

(1) Análise da Sentença: É realizado um *parsing* sintático de cada sentença do parágrafo, resultando em uma árvore sintática para cada sentença.

Cada nó da árvore é associado a uma estrutura que contém informações necessárias para a etapa (3), Geração do Axioma.

(2) Análise do Contexto: É realizada uma resolução das anáforas e do escopo dos quantificadores existentes. Posteriormente, parâmetros definidos implicitamente também são analisados. O resultado dessas análises é representado em um grafo.

(3) Geração do Axioma: Para cada parágrafo, é gerado um axioma a partir do grafo gerado e das estruturas associadas aos nós da árvore de cada sentença que compõe o parágrafo.

De acordo com os autores, o sistema em questão apresentou um bom desempenho. Os experimentos relatados foram realizados em 29 parágrafos do protocolo de especificação OSI, num total de 98 sentenças. Dessas, 95 foram traduzidas corretamente.

Apesar do bom resultado apresentado, é importante salientar que o foco do sistema proposto é em descrições de protocolos de comunicação, ao passo que o foco do NLForSpec é em descrições de casos de teste.

3.3. O sistema NL-OOPS

NL-OOPS (*Natural Language – Object Oriented Production System*) [Mich, 1996] é um exemplo de ferramenta para geração de modelos orientados a objeto (OO) a partir de descrições em Linguagem Natural. Também se pode classificar NL-OOPS como uma ferramenta CASE (*Computer Aided Software Engineering*) que auxilia na análise de requisitos através de modelos obtidos de requisitos descritos em LN.

O núcleo do NL-OOPS é o sistema para PLN LOLITA [Long & Garigliano, 1994], que utiliza uma rede semântica, denominada SemNet, para representação do conhecimento. SemNet contém mais de 100.000 nós conectados, que podem ser consultados, modificados ou expandidos. LOLITA apresenta uma arquitetura modular, tradicional para PLN, como ilustrado na figura 3.8 a seguir.

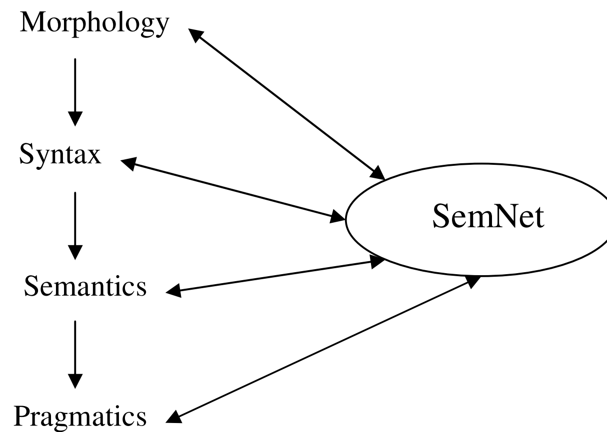


Figura 3.8: Arquitetura do Sistema LOLITA [Long & Garigliano, 1994]

Como pode ser observado na figura 3.8, LOLITA apresenta os módulos de Processamento Morfológico, Processamento Sintático, Processamento Semântico e Processamento Pragmático, característicos de um sistema de Interpretação de Linguagem Natural, conforme descrito na seção 2.2. De fato, LOLITA é um sistema de PLN de propósito geral, já tendo sido usado em sistemas com diferentes objetivos (*e.g.*, Sistemas de Extração de Informação e Tradutores Automáticos).

Para o caso específico do NL-OOPS, LOLITA recebe como entrada os requisitos em Linguagem Natural e tem como saída uma rede semântica. Para que essa representação se torne mais adequada para a análise OO, os nós da SemNet utilizados para interpretação do texto são rearranjados em tabelas. A partir daí, NL-OOPS utiliza um algoritmo próprio para obtenção das classes, associações, atributos e operações que estarão no modelo OO gerado. A figura 3.9 traz um exemplo de modelo OO gerado a partir da descrição de uma companhia de hotéis.

Sogno is a company which owns and manages hotels. It needs to modify its information system. The new system should improve the reservation services in order to get a better level of integration, which would improve the guest service. There are three hotels in the chain. All the hotels are located in the same area. The First hotel, Bolzano, is near the cathedral, in the centre of the town. It is a four-star hotel. It has 15 double rooms and 5 single rooms. Each room contains a bath and a small balcony. The hotel Bolzano has a restaurant, a private car park and a garden. The hotel Koenig is the second hotel owned by Sogno. It is not far from the railway station. It is a big new three-star hotel with 55 rooms. The third hotel, Sirena, is by the sea. It has 30 rooms. All rooms have a balcony. 10 rooms look out onto the sea. The hotel Sirena has a garage, a disco and two restaurants. Guests can make a reservation by calling the hotels or the central place of the company, Sogno. Guests are asked about the arrival-departure time and the type of room. They should leave the name, the telephone number and the way of payment. If there isn't a vacant room in the hotel called by guests, the receptionist should propose a room in another hotel of the chain. If they cancel the reservation, they lose the deposit.

NL-OOPS

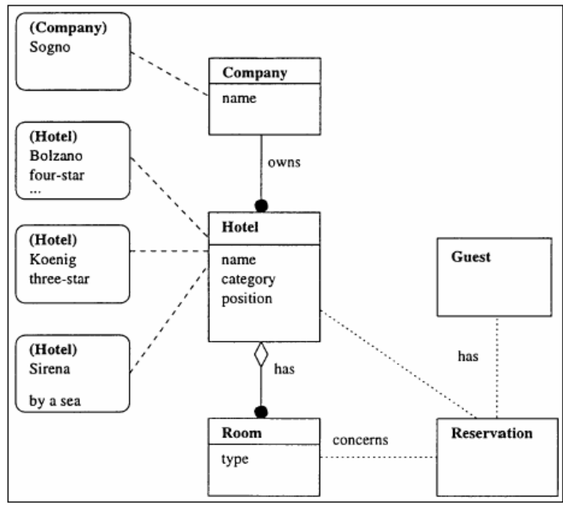


Figura 3.9: Exemplo de Entrada e Saída do Sistema NL-OOPS

O NL-OOPS apresenta duas limitações principais. A primeira delas é a necessidade de um pré-processamento manual dos documentos de requisitos em busca de ambigüidades, inconsistências e omissões antes de serem processados pelo sistema. A segunda limitação ocorre quando o tamanho dos documentos de requisitos aumenta muito. Documentos longos geram uma rede semântica muito complexa, tornando a obtenção de objetos para o modelo OO uma tarefa complicada e lenta, o que ocasiona a geração de modelos OO inadequados.

A escolha de uma rede semântica como forma de representação de conhecimento traz alguns aspectos negativos para o sistema. A dificuldade de implementação de algoritmos para percorrer essas redes de forma eficiente é um problema antigo e ainda não superado. Outro aspecto que pode ser identificado é a falta de padronização entre as redes semânticas, o que dificulta a portabilidade dessas bases de conhecimento entre sistemas.

3.4. Geração de Modelos VHDL

Outro exemplo de geração automática de modelos a partir de Linguagem Natural é o sistema para Geração de Modelos VHDL[2] (VHSIC *Hardware Description Language*) [Cyre *et al.*, 1994]. Esse sistema gera modelos VHDL com múltiplos processos a partir de descrições em Linguagem Natural.

Quatro etapas principais de processamento podem ser identificadas nesse sistema. A primeira realiza um processamento sintático utilizando um *parser bottom-up* [Winograd, 1983] e uma gramática com aproximadamente 120 regras. Na segunda etapa, ocorre a análise semântica de cada sentença. Para isso, o sistema gera um grafo conceitual [Sowa, 1984], uma espécie de rede semântica de cada sentença. A figura 3.10, a seguir, traz exemplo de um grafo conceitual para a sentença "*The 8-bit data is loaded into the ACC register when STRB rises*".

A terceira etapa é responsável por integrar os grafos conceituais de cada sentença em um grafo único, através da detecção de conceitos que se referem a um mesmo objeto ou ação. A quarta e última etapa do sistema é responsável por gerar o modelo VHDL a partir do grafo conceitual único gerado na etapa anterior. Para realizar isso, o sistema utiliza um programa denominado Linker [Honcharik, 1993]. O Linker funciona basicamente da seguinte forma: nós no grafo conceitual que representam verbos tipicamente geram processos ou condições que os governam; e nós que representam objetos tipicamente geram sinais no modelo VHDL.

[2] IEEE Standard VHDL Language Reference Manual. Disponível em: http://standards.ieee.org/reading/ieee/std_public/description/dasc/1076-1993_desc.html

```
[ 1 : load ]
      ->( object) -> [ 2 ]
      ->( destination : into ) -> [ 3 ]
      ->( condition : when ) -> [ 4 ]
[ 2 : data ]
      ->( size : adj ) -> [ 5 ]
      ->( det : the )
[ 5 : 8-bit ]
[ 3 :  register  ]
      ->( name ) -> [ 6 ]
      ->( det : the )
[ 6 : "ACC" ]
[ 4 : rise ]
      ->( agnt ) -> [ 7 ]
[ 7 : "STRB" ]
```

Figura 3.10: Exemplo de grafo conceitual [Cyre et al., 1994]

Apesar de os autores afirmarem que todos os modelos VDHL foram gerados corretamente, um aspecto negativo detectado nos experimentos realizados com o sistema foi o pequeno número de sentenças nos arquivos de descrição (no máximo 10), o que não possibilitou uma avaliação do sistema para descrições mais complexas, com maior número de informações.

3.5. Considerações Finais

Este capítulo apresentou exemplos de sistemas de PLN que geram modelos (formais ou não) como saída. Foram apresentados sistemas que possuem a arquitetura *pipeline* baseada na abordagem simbólica vista na seção 2.2.

Podem ser observadas diferenças na abordagem utilizada pelos sistemas apresentados. Por exemplo, o sistema NL-OOPS e o sistema proposto por Lee & Bryant (2002) possuem tratamento do discurso, ou seja, analisam o texto como um todo, e não em sentenças ou parágrafos isolados. Já o sistema proposto por Ishihara *et al.* (1992) trata cada parágrafo isoladamente, enquanto o sistema para geração de modelos VHDL [Cyre *et al.*, 1994] trata de cada sentença de forma isolada. A diferença entre as abordagens utilizadas pode ser explicada pela diferença entre o domínio de cada uma das aplicações. Alguns sistemas, como o NL-OOPS e o de Lee & Bryant (2002), recebem como entrada documentos de requisitos, e por isso necessitam processar o

texto como um todo; outros sistemas, como o de Cyre *et al*. (1994), podem tratar isoladamente cada sentença da descrição em LN.

Um outro ponto importante a ser discutido a respeito dos sistemas apresentados concerne à dificuldade de acesso a maiores detalhes sobre técnicas utilizadas nos diversos módulos de cada sistema. Em virtude de a maioria dos trabalhos na área de geração de modelos a partir de LN ser publicada em fóruns voltados para a Engenharia de Software, as informações referentes ao processamento de LN dos sistemas são muitas vezes simplificadas ou até mesmo omitidas. O enfoque maior é dado à contribuição dos sistemas para o processo de desenvolvimento de software, e não à área de PLN.

No capítulo 4 a seguir será apresentado o *Test Research Project*, projeto no qual o trabalho apresentado nesta dissertação está inserido. As atividades desse projeto serão apresentadas, seguidas por uma breve discussão sobre Testes de Software e uma introdução à notação formal CSP.

4. *Test Research Project*

Antes de abordar o trabalho realizado, é necessário apresentar o contexto no qual ele foi desenvolvido. Este trabalho foi realizado dentro do *Test Research Project*, um projeto de pesquisa fruto de parceria entre o Centro de Informática da UFPE e o *Motorola Brazil Test Center* (BTC). O objetivo geral desse projeto é contribuir com todo o processo de testes da Motorola, automatizando a geração, seleção e avaliação de casos de teste para aplicações móveis.

O projeto surgiu da necessidade de se identificarem oportunidades de melhorias no processo de desenvolvimento e execução de testes. Ou seja, os pesquisadores membros do *Test Research Project* têm como tarefa conhecer e estudar todo o processo de testes do BTC para identificar possíveis falhas ou oportunidades de melhorias. A partir daí, os pesquisadores devem procurar, dentro do meio acadêmico ou do meio industrial, alternativas que possam ser utilizadas dentro do processo de testes. Após realizar a análise das alternativas, os pesquisadores devem propor e implementar soluções para o processo de testes do BTC.

Conforme citado, o objetivo mais amplo do projeto é contribuir com o processo de testes da Motorola. Isso inclui os seguintes objetivos específicos:

- Documentação de Requisitos – desenvolver uma Linguagem Natural Controlada (LNC) para ser utilizada nos documentos de requisitos, com o intuito de sistematizar a geração e a seleção efetiva de casos de teste;
- Seleção de Casos de Teste – definir um algoritmo para a seleção de casos de teste, ou seja, tornar possível a identificação efetiva de testes com potencial para revelar erros importantes na aplicação, e com cobertura adequada;

- Requisitos Documentados a partir dos Testes – muitas vezes, os casos de teste contêm informações mais atualizadas do que os documentos de requisitos. A geração/atualização de requisitos a partir dos casos de testes é um outro importante objetivo desta iniciativa;
- Avaliação da Suíte de Testes e Resultados – o escopo do projeto inclui ainda o desenvolvimento de técnicas e de ferramentas que permitam analisar parâmetros como cobertura, confiabilidade e tempo de execução dos testes gerados.

Na seção a seguir, as atividades que estão sendo desenvolvidas no *Test Research Project* serão explicadas em maiores detalhes.

4.1. Atividades do *Test Research Project*

A figura 4.1, a seguir, exibe o fluxo de informação do *Test Research Project*. As atividades são representadas pelas setas numeradas, enquanto os quadros representam os artefatos de entrada/saída das atividades que estão sendo desenvolvidas no projeto.

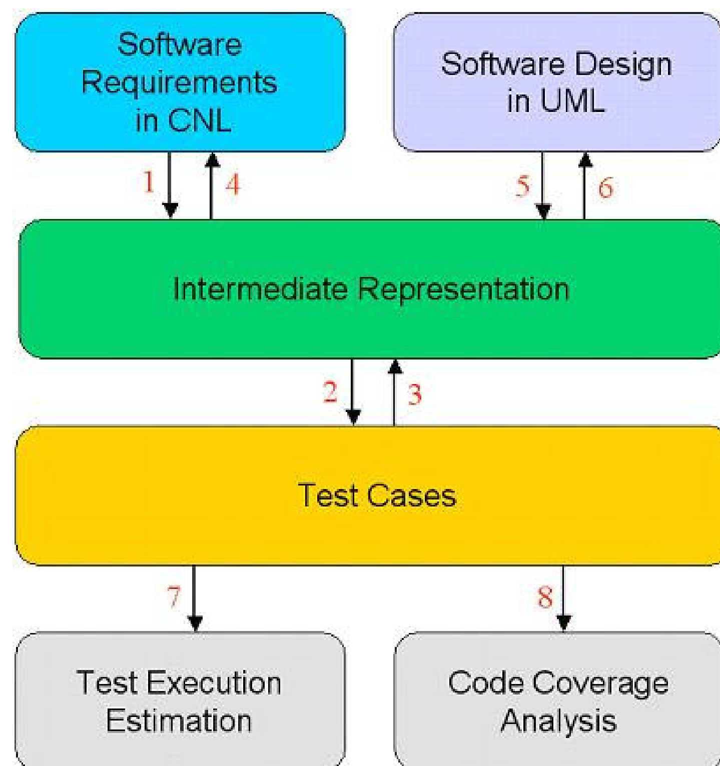


Figura 4.1: Fluxo de Informações do *Test Research Project*

As atividades do *Test Research Project* podem ser divididas em três grandes grupos, como a seguir.

- **Geração de Casos de Teste**: agrupas as atividades representadas pelas setas 1, 2 e 5, e tem como objetivo gerar casos de teste a partir de documentos de requisitos e de diagramas. As atividades e os artefatos que compõem este grupo serão detalhados na seção 4.1.1.
- **Estimativas de Execução e Cobertura de Código**: agrupa as atividades 7 e 8. Recebe como entrada casos de teste e tem como objetivo estimar a execução desses testes e analisar a sua cobertura de código. Maiores detalhes dessas duas atividades serão discutidos na seção 4.1.2.
- **Atualização dos Requisitos**: agrupa as atividades representadas pelas setas 3, 4 e 6, e tem como objetivo atualizar os requisitos (documentos e diagramas) a partir de informações mais atuais contidas nos casos de teste. As atividades e os artefatos que compõem este grupo serão detalhados na seção 4.1.3.

A seguir, será descrito cada um desses três grandes grupos de atividades. A numeração indicada pelas setas da figura 4.1 continuará sendo utilizada nas próximas seções.

4.1.1. Geração de Casos de Teste

A atividade de Geração de Casos de Teste dentro do *Test Research Project* utiliza a abordagem conhecida como Teste Baseado em Modelo (*Model Based Testing* - *MBT*) [El-Far & Whittaker, 2001]. MBT é uma técnica para geração de teste de software que consiste em especificar um modelo formal ou semi-formal dos requisitos da aplicação a ser testada de modo a caracterizar com exatidão o seu comportamento e, a partir desse modelo especificado, gerar os testes [Dalal *et al*., 1999]. A figura 4.2 ilustra a estratégia do MBT.

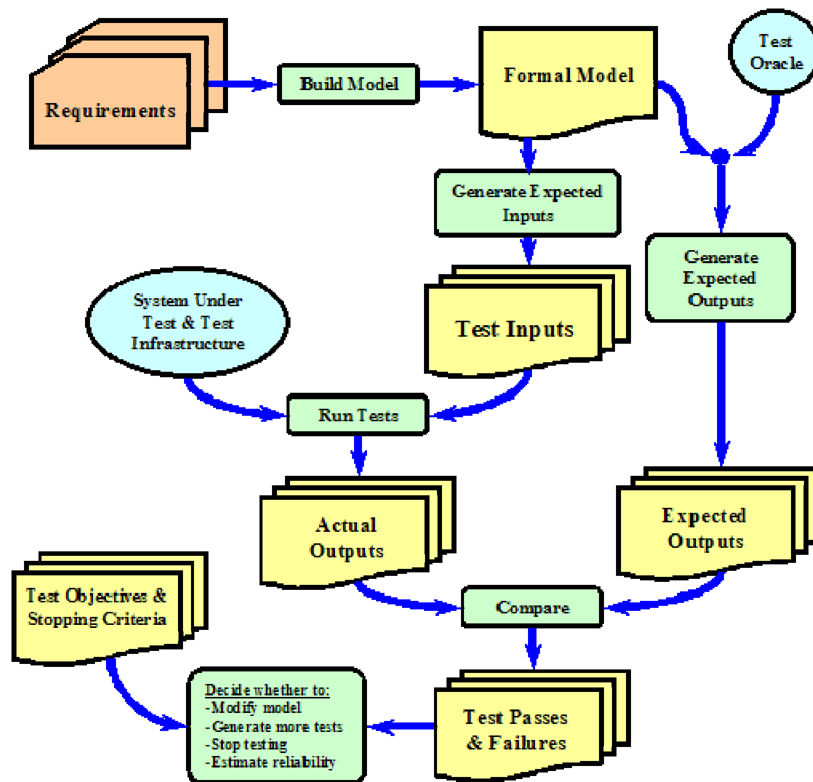


Figura 4.2: Teste Baseado em Modelo [Gold Practices, 2006]

Conforme pode ser observado na figura 4.2, a entrada para um sistema de geração de testes baseado em modelo são os requisitos da aplicação. A partir daí, é gerado o modelo formal da aplicação, que serve como base para geração dos testes. No *Test Research Project*, a estratégia para geração de testes é semelhante a essa. As atividades 1 e 5 (figura 4.1) geram uma representação intermediária (modelo formal) a partir dos requisitos e de diagramas, respectivamente. A partir daí, a atividade 2 gera os casos de teste com base nessa representação intermediária. A seguir, essas três atividades que compõem a estratégia serão detalhadas.

Geração de Modelos de Uso a partir de Documentos de requisitos (atividade 1) – É considerada a primeira no processo de geração de testes. Aqui, as funcionalidades do sistema são especificadas em casos de uso escritos em uma linguagem natural controlada (LNC). Essa LNC consiste em um subconjunto de uma língua natural (Inglês) que pode ser processado diretamente por uma máquina, e que é expressivo o suficiente para ser entendido por usuários não especialistas. Um *template* de caso de uso também foi definido no formato Microsoft Word. Essa mesma LNC será usada na edição dos casos de teste. A padronização na escrita de casos de uso e de casos de teste tem como objetivo uniformizar a documentação e reduzir a ambigüidade do texto,

aumentando a qualidade do documento e reduzindo o tempo de inspeção. Além disso, os requisitos e casos de teste escritos em LNC facilitam o processamento automático.

Com as funcionalidades do sistema (casos de uso) documentadas na LNC, é possível realizar o processamento automático da documentação com a finalidade de gerar o modelo de uso do sistema. Este modelo de uso é uma representação formal das possíveis ações do usuário do sistema. O modelo é escrito na notação formal CSP, sendo representado por *Intermediate Representation* na figura 4.1.

Geração de Modelos de Uso a partir de Diagramas em UML (atividade 5) – Esta é uma atividade complementar à Geração de Modelos de Uso a partir de Documentos de Requisitos, e consiste em traduzir diagramas UML[3] (Unified Modeling Language) de seqüência para modelos de uso na notação CSP.

Geração de Casos de Teste (atividade 2) – Esta atividade recebe como entrada o modelo de uso do sistema escrito em CSP, e gera automaticamente os casos de testes também escritos em CSP. A geração é guiada por propósitos de teste, que direcionam a geração de acordo com critérios de cobertura de requisitos. Os benefícios diretos dessa geração automática são o aumento da eficiência e da produtividade, bem como da qualidade dos testes gerados.

Os casos de testes gerados, representados na *Intermediate Representation*, podem ser então mapeados automaticamente para uma LNC, para serem manualmente executados por engenheiros de testes. Isso aumenta a eficiência da execução dos testes gerados, pois os engenheiros não precisam entender a especificação formal.

4.1.2. Estimativas de Execução e Cobertura de Código

As atividades 7 e 8 da figura 4.1 recebem como entrada os casos de testes gerados automaticamente e têm os seguintes objetivos:

Estimativas de Execução de Testes (atividade 7) – uma das linhas de pesquisa do *Test Research Project* consiste em desenvolver modelos de estimativa para a execução dos testes gerados automaticamente. Modelos existentes estão sendo analisados e adaptados para o ambiente do projeto. A padronização dos casos de teste

[3] http://www.omg.org/technology/documents/formal/uml.htm

gerados em uma LNC permite a automatização das estimativas. Isso aumenta a precisão das estimativas e a capacidade de planejamento e de decisão.

Cobertura de Código (atividade 8) – é de fundamental importância medir a qualidade dos casos de teste gerados. Dessa forma, o porcentual de redundância de um conjunto de casos de teste, bem como a quantidade de código que é coberta pelos testes são medidas importantes para a avaliação da qualidade da suíte gerada. Com essa informação em mãos, é possível guiar o processo de geração, de forma a otimizar a cobertura dos casos de teste gerados.

4.1.3. Atualização dos Requisitos

Durante o processo de desenvolvimento de software, é comum apenas os casos de teste serem atualizados de forma a refletir eventuais mudanças no código, enquanto os documentos de requisitos permanecem desatualizados. Dessa forma, o projeto tem como uma de suas metas desenvolver um procedimento sistematizado para atualização dos requisitos a partir de casos de teste mais atuais.

Esse procedimento para atualização de requisitos é baseado na abordagem *Anti-Model Based Testing* (AMBT) [Bertolino *et al.*, 2004], que é justamente o oposto da abordagem de Teste Baseado em Modelo apresentada na seção anterior. Ou seja, enquanto o MBT constrói um modelo de uso com base nos requisitos e gera os casos de testes, o AMBT recebe como entrada os casos de teste, e tem como objetivo gerar um modelo de uso a partir desses casos de teste. Esse procedimento pode ser dividido nos seguintes passos:

- selecionar os casos de teste baseado em alguma especificação de mais alto nível (por exemplo, selecionar os casos de teste associados a um determinado requisito);
- executar os casos de teste e capturar os *traces* de execução; e
- construir um modelo de uso através de engenharia reversa dos *traces* de execução.

No caso do *Test Research Project*, o processo ocorre de forma semelhante. As setas 3, 4 e 6 da figura 4.1 (que têm a direção de baixo para cima) correspondem às atividades que compõem esse processo, detalhadas a seguir.

Geração de Especificações Formais a partir de Casos de Teste (atividade 3) - O primeiro passo para a atualização dos documentos de requisitos é a tradução dos casos de teste descritos em linguagem natural para uma representação formal intermediária (*Intermediate Representation*). É justamente nessa etapa que se encaixa o trabalho apresentado nesta dissertação. NLForSpec é responsável por traduzir os casos de teste em Linguagem Natural para uma representação formal em CSP. A partir daí, ocorre a elaboração do modelo de uso com base nas especificações geradas pelo NLForSpec. Esse modelo de uso serve como entrada para as atividades descritas a seguir.

Atualização dos Requisitos (atividade 4) – Esta atividade é responsável por comparar o modelo de uso obtido a partir dos requisitos com o modelo de uso obtido a partir dos casos de testes e, quando necessário, atualizar os requisitos. Essa atualização será refletida nos documentos de requisitos por uma ferramenta que mapeia a notação formal em textos em uma Linguagem Natural Controlada.

Geração de Diagramas UML (atividade 6) – Além da atualização dos requisitos, também é possível gerar diagramas UML com as informações que vieram tanto do modelo de uso dos casos de teste, quanto do modelo de uso dos requisitos.

Depois dessa apresentação geral do *Test Research Project*, veremos agora uma discussão mais detalhada sobre as entradas e saídas do NLForSpec. Inicialmente, serão discutidos testes de software, mais especificamente, descrições de casos de teste. Em seguida, será dada uma breve introdução à notação formal CSP, com foco nas construções de CSP utilizadas neste trabalho.

4.2. Casos de Teste

Teste de Software é uma etapa de central importância no processo de desenvolvimento de software. O processo de teste de um software envolve qualquer atividade que tenha como objetivo avaliar um atributo ou uma funcionalidade de um programa ou sistema, e determinar se eles estão de acordo com os resultados esperados [Hetzel, 1988]. A concepção tradicional do processo de teste, como sendo uma fase final e independente do processo de desenvolvimento propriamente dito, tem-se mostrado bastante ineficiente, devido aos altos custos associados à correção de erros

encontrados e à manutenção do software. Tal fato contribuiu para a definição de métodos e técnicas sistemáticas de teste que fizeram do processo de teste um conjunto de tarefas à parte que pode ser aplicado ao longo do processo de desenvolvimento [McGregor & Sykes, 2001]. Testar um software é um processo concorrente no ciclo de vida da engenharia de software, e tem como objetivo medir e melhorar a qualidade da aplicação que está sendo testada [Craig & Jaskiel, 2002].

Os Testes de Software podem ser classificados da seguinte forma [Watkins, 2001]:

- *Black-box* ou funcional – são testes planejados a partir da especificação abstrata, ou seja, não há conhecimento do código;
- *White-box* ou estrutural – são testes definidos a partir do conhecimento de detalhes da implementação;
- *Gray-box* – é um teste *black-box* baseado em um conhecimento limitado de detalhes da implementação.

A essência do teste de software é determinar o conjunto de casos de teste para o item a ser testado. Um caso de teste é um conjunto de condições e variáveis que são utilizadas pelo engenheiro de testes para determinar se o item a ser testado está inteiramente ou parcialmente de acordo com o que foi especificado. Um caso de teste é composto por entradas e saídas, especificadas como a seguir [Jorgensen, 1995]:

- Entradas
    (1) Pré-condição: assegura a condição inicial para que o caso de teste possa ser executado;
    (2) Passos atuais (ações): passos a serem executados, identificados pelos métodos de teste.
- Saídas
    (1) Saída Esperada (Resultado Esperado): respostas esperadas do sistema, para o passo atual correspondente;
    (2) Pós-condição: representa o estado final do sistema, *i.e.*, condições que devem ser verdadeiras após a execução dos passos do caso de teste.

Antes de executar os passos (ações) do caso de teste, o engenheiro deve verificar se todas as pré-condições estão satisfeitas. Caso não estejam, o caso de teste não pode ser executado. Se as pré-condições forem satisfeitas, cada passo do caso de teste é executado seqüencialmente, e o resultado esperado associado a cada um dos passos é comparado à resposta do sistema. Se para cada ação executada no sistema o resultado esperado for verificado, e se, ao final da execução as Pós-Condições forem verdadeiras, pode-se dizer que o caso de teste foi satisfeito e que o sistema comportou-se da maneira esperada.

A forma mais comum de se definir um caso de teste é através de descrições em Linguagem Natural. Cada etapa que compõe um caso de teste é definida por sentenças em alguma LN. Para executar manualmente um caso de teste, um engenheiro de testes deve ler cada etapa do caso de teste e executar a ação indicada pela descrição correspondente. A figura 4.3 ilustra um exemplo de uma descrição de caso de teste em português.

Descrição:	Testa o envio de mensagem a um destinatário presente na agenda	
Condições inicais:		
1.	Existe um contato de nome "João" na agenda.	
Passo:	**Ação:**	**Resultado Esperado**
1.	Vá para o formulário de composição de nova mensagem.	O formulário é exibido.
2.	Digite o texto "Teste".	O texto é exibido na tela.
3.	Selecione a opção "Selecionar contato da Agenda".	A agenda é exibida.
4.	Selecione o contato de nome "João".	O contato é selecionado com sucesso.
5.	Selecione a opção "Enviar Mensagem".	A mensagem é enviada.
Pós-condições		
1.	A mensagem enviada foi movida para a pasta "Itens enviados".	

Figura 4.3: Exemplo de Caso de Teste

4.3. CSP (*Communicating Sequential Processes*)

Veremos agora uma breve descrição de CSP (*Communicating Sequential Processes*), a linguagem adotada pelo *Test Research Project*.

CSP [Hoare, 1985] [Roscoe *et al.*, 1997] é uma linguagem formal para modelar padrões de interação em sistemas concorrentes e distribuídos. CSP também pode ser vista como uma teoria matemática para especificar agentes independentes (ou sub-sistemas) que se comunicam entre si e com o seu ambiente em comum. O foco desta seção é discutir os aspectos de CSP que são relevantes para nosso trabalho, pois a teoria que envolve CSP é muito ampla. A versão de CSP utilizada neste trabalho é CSP_M [Scattergood, 1998] (acrônimo para *Machine-readable CSP*), que adiciona a CSP aspectos relacionados a dados do sistema.

Em CSP, os sistemas são especificados através de três elementos básicos: *eventos*, *operadores* e *processos*. Eventos são abstrações de ações do mundo real. Por exemplo, o evento *enviar.Mensagem* pode ser usado para representar a ação de envio de uma mensagem em uma aplicação móvel. Em CSP, eventos são instantâneos (atômicos), isto é, o intervalo de tempo entre um evento e o próximo não é considerado. Conceitualmente, eventos ocorrem imediatamente após suas condições de execução serem satisfeitas.

Além de eventos, que representam uma única ação, CSP também permite a definição de *canais*, que representam coleções de eventos com características em comum. Por exemplo, a declaração *channel c: Int* introduz um canal *c* que pode transmitir qualquer valor inteiro. O evento *c.1* é um dos que podem ocorrer através do uso do canal *c*. Canais em CSP podem ser tipados ou não-tipados. Canais tipados, como o canal *c*, podem transmitir dados de um determinado tipo (no caso do canal *c*, do tipo *Int*), enquanto canais não tipados são utilizados apenas como pontos de sincronização. Por exemplo, a declaração *channel nt* define um canal não-tipado de nome *nt*.

CSP usa *datatypes* para determinar classes de eventos que podem ocorrer através de um canal tipado. Por exemplo, e especificação da figura 4.4, determina que o canal

enviar pode transmitir qualquer valor do tipo *Mensagem*, ou seja, qualquer dos eventos *enviar. SMS*, *enviar. MMS* e *enviar. EMS* pode ocorrer.

```
datatype Mensagem = SMS | MMS | EMS
channel enviar : Mensagem
```

Figura 4.4: Exemplo de *datatype* simples

CSP permite o desenvolvimento de *datatypes* mais complexos, que envolvem tuplas, conjuntos, seqüências etc. Por exemplo, o *datatype* da figura 4.5 é formado por uma tupla que contém uma mensagem e um conjunto de destinatários.

```
datatype MensagemBroadcast = (Mensagem, Set(Destinatario))
```

Figura 4.5: Exemplo de uso de tupla em um *datatype*

Também é possível definir *datatypes* a partir de outros *datatypes*. O *datatype MensagemGenerica* da figura 4.6 representa uma mensagem genérica, que pode ser tanto uma mensagem simples como uma mensagem para múltiplos destinatários. A sintaxe do CSP_M exige o uso de construtores (*MSG_SIMPLES* e *MSG_COMPLEXA*) para especificar cada possível uso do *datatype MensagemGenerica*.

```
datatype MensagemGenerica = MSG_SIMPLES.Mensagem
                          | MSG_COMPLEXA.MensagemBroadcast
```

Figura 4.6: Exemplo de *datatype* composto

CSP também permite modelar a entrada e a saída de dados em canais. A expressão $e?x : \{x < 10\}$ determina que o canal *e* pode receber qualquer valor inteiro menor que 10, e este será atribuído à variável *x*. Já a expressão $e!y : \{1,2,3\}$ determina que o canal *e* pode enviar qualquer um dos valores do conjunto {1,2,3}.

Uma seqüência pré-definida de eventos determina um padrão comportamental. *Processos* são abstrações para padrões comportamentais, e são construídos através de eventos e operadores. O conjunto de eventos que podem ocorrer em um processo é

denominado de alfabeto, e é representado pelo símbolo $\sum$. A ocorrência de um evento em um processo caracteriza uma comunicação deste com pelo menos um participante. Geralmente, o participante é um outro processo, caso contrário será o próprio ambiente em que o processo está envolvido. A composição de eventos para formar um processo, bem como o relacionamento entre diferentes processos são descritos através dos *operadores algébricos* de CSP. Existe um grande número de operadores algébricos de CSP. A seguir serão descritos apenas os utilizados neste trabalho.

- **Processos Primitivos**: CSP possui dois processos especiais, denominados primitivos: *STOP* e *SKIP*. *STOP* representa um processo em *deadlock*, que não pode fazer mais nada (*e.g.*, uma máquina quebrada). Já *SKIP* representa um processo que concluiu sua execução com sucesso.
- **Prefixo:** Para construir um processo através de seus eventos é utilizado o operador $\rightarrow$, chamado operador de prefixo. O operador de prefixo expressa um processo que representa a ocorrência de um evento e depois se comporta como outro processo. Por exemplo, no comportamento do processo $e \rightarrow P$, e representa um evento e P representa um processo que ocorre após o evento.
- **Recursão:** O operador de prefixo é usado para descrever a seqüência de eventos de qualquer processo finito, mas é inviável para processos que apresentam ciclos repetitivos. Surge então a necessidade de um mecanismo para representar comportamentos infinitos. Tal mecanismo é a recursão. No exemplo *Relogio= tick→ tack→ Relógio*, o processo *Relógio* executa os eventos *tick* e *tack*, nesta ordem, e volta a comportar-se como *Relogio*, indefinidamente.

Os exemplos de processos a seguir ilustram o uso dos operadores algébricos de CSP.

IntroducaoCSP = operadoresAlgebricos → descricao → exemplo → SKIP
RelogioQuebrado = tick → tack → STOP

Figura 4.7: Exemplos de Processos CSP

O primeiro dos processos da figura 4.7 é o *IntroducaoCSP*, que começa executando o evento *operadoresAlgebricos*, passando pelo evento *descricao*, seguido

por *exemplo*. O processo *IntroducaoCSP* é finalizado por *SKIP*, indicando terminação com sucesso. Já o processo *RelogioQuebrado* representa um relógio com algum tipo de problema. Ele executa apenas um ciclo *tick* e *tack* e depois pára indefinidamente (conforme representado por *STOP*).

4.4. Considerações Finais

Neste capítulo foi apresentado o projeto no qual o trabalho descrito nesta dissertação está inserido, o *Test Research Project*. Um dos objetivos deste projeto é a atualização de requisitos a partir de descrições de casos de teste com informações mais recentes. É justamente nesse contexto que o NLForSpec surge, sendo responsável por executar o primeiro passo desse objetivo, a geração de especificações formais em CSP a partir de descrições de casos de teste em Linguagem Natural.

Em seguida, foi apresentada uma breve discussão sobre a entrada do sistema, os casos de teste. O tipo de caso de teste utilizado como entrada, *black-box*, foi apresentado, bem como as partes que o compõem. Por fim, foram discutidos os aspectos da linguagem formal CSP que serão utilizados neste trabalho.

O capítulo 5 apresenta a ferramenta NLForSpec em detalhes, sempre retomando os conceitos apresentados neste capítulo.

5. A Ferramenta NLForSpec

Neste capítulo será apresentada a principal contribuição deste trabalho, o sistema NLForSpec. Conforme explicado no capítulo anterior, NLForSpec é parte de um projeto maior e tem como objetivo gerar especificações formais em CSP a partir de descrições de casos de teste em Linguagem Natural.

A seção 5.1 apresenta a arquitetura de NLForSpec, seguida da seção 5.2, que traz detalhes sobre as bases de conhecimento do sistema, desenvolvidas para aplicações em telefonia móvel. A seção 5.3 descreve os módulos de processamento. Por fim, a seção 5.4 traz as considerações finais deste capítulo.

5.1. Arquitetura

A arquitetura do NLForSpec é modular e foi baseada na abordagem simbólica para Interpretação de Linguagem Natural, conforme ilustrado na figura 5.1.

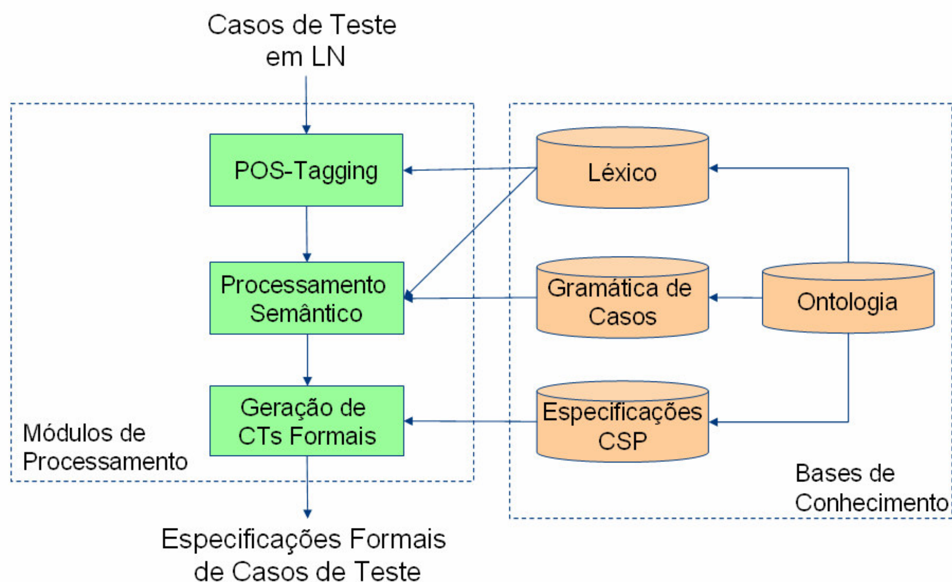


Figura 5.1: Arquitetura de NLForSpec

Como pode ser observado, NLForSpec conta com três módulos de processamento com objetivos bem definidos: POS-Tagging, Processamento Semântico e Geração de Casos de Teste (CTs) Formais. NLForSpec apresenta duas principais diferenças em comparação à arquitetura simbólica tradicional para PLN (discutida na seção 2.2.1). A primeira diferença é que NLForSpec não apresenta o módulo de Processamento Léxico, pois essa tarefa é feita em conjunto com a análise sintática pelo módulo de POS-Tagging, conforme será descrito na seção 5.3.1. A segunda principal diferença é a ausência dos módulos de Processamento do Discurso e Processamento Pragmático. Essas etapas de processamento não são necessárias aqui, uma vez que cada sentença que compõe um caso de teste representa uma ação isolada a ser tomada. Desta forma, cada sentença é interpretada de forma isolada, sem haver a necessidade de processamento do discurso e pragmático.

NLForSpec faz uso de quatro bases de conhecimento: Léxico, Gramática de Casos, Ontologia e Base de Especificações CSP. Em comparação à arquitetura simbólica tradicional de PLN, a principal diferença é a presença da base de especificação formal, responsável por armazenar conhecimento necessário para geração das especificações formais de saída.

Na seção 5.2, a seguir, serão detalhadas as quatro bases de conhecimento que compõem o sistema NLForSpec. Em seguida, na seção 5.3, os três módulos de processamento serão apresentados.

5.2. Bases de Conhecimento

Veremos a seguir detalhes sobre as quatro bases de conhecimento da ferramenta NLForSpec: Ontologia, Léxico, Gramática de Casos e Base de Especificações CSP. Todas as bases foram descritas no formato XML [W3C, 2006].

5.2.1. Ontologia

Ontologias são bases de conhecimento que contêm definições sobre as entidades que compõem o domínio da aplicação. O desenvolvimento de uma ontologia tem como objetivo classificar as entidades e suas relações, facilitando o reuso de conhecimento sobre o domínio e, principalmente, separando esse tipo de conhecimento do

conhecimento operacional da aplicação. Isso, além de facilitar a manutenção, aumenta o grau de portabilidade da aplicação.

A figura 5.2 mostra exemplo de ontologia. À esquerda da figura, está uma ilustração gráfica das classes da ontologia; à direita, a ontologia no formato XML. É importante salientar que todos os exemplos que serão mostrados, tanto para a ontologia como para as outras bases de conhecimento, foram retirados de um estudo de caso realizado para o domínio de aplicações para telefones celulares (capítulo 6).

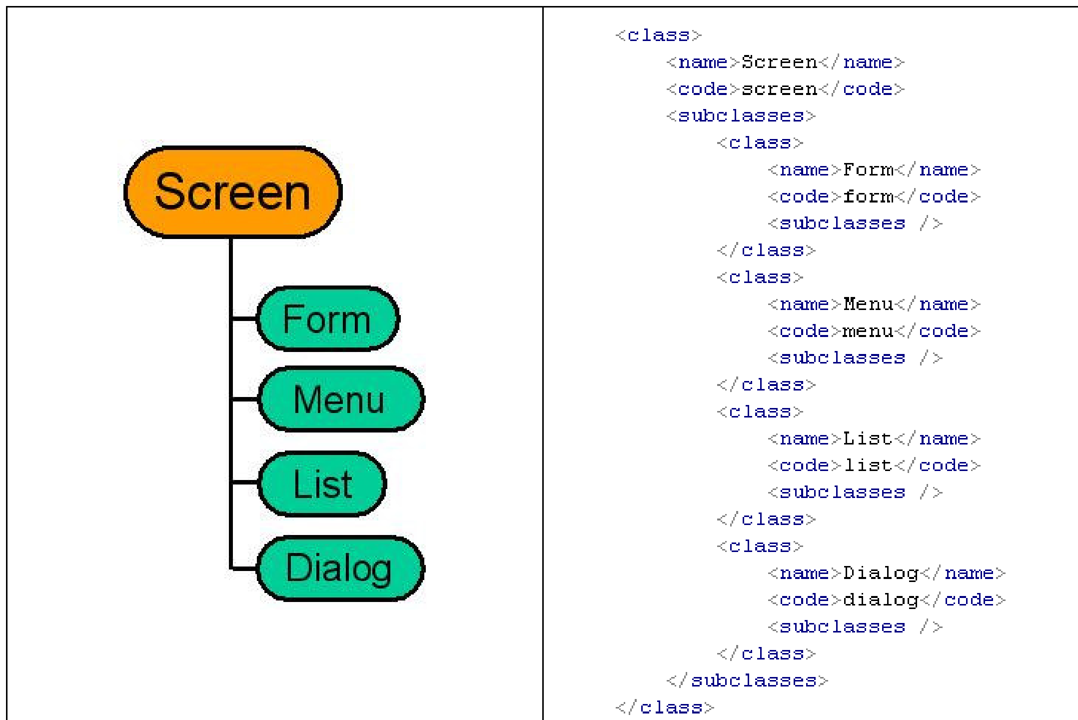


Figura 5.2: Exemplo de trecho da Ontologia

Na figura 5.2 podem ser observadas várias classes que estão presentes na ontologia de NLForSpec. A única relação que existe entre estas classes é a relação de especialização. Por exemplo, as classes *Form* (formulário), *Menu, List* (lista) *e Dialog* (tela de diálogo) são sub-classes da classe *Screen* (tela).

Na representação em XML da ontologia, a *tag* <class> indica o início de uma nova classe, no caso a classe de nome *Screen*. A *tag* <subclasses> indica que a partir dali serão listadas todas as sub-classes de *Screen*, ou seja, na figura 5.2, todas as classes que aparecem dentro da *tag* <subclasses> são sub-classes de *Screen.*

Na representação XML da ontologia estão armazenadas apenas as classes do domínio. As instâncias dessas classes (as entidades do domínio) estão representadas na

base de conhecimento léxico (seção 5.2.2) e possuem um *link* com sua respectiva classe na ontologia.

5.2.2. Léxico

A Base de Conhecimento Léxico armazena os termos que podem aparecer nas frases de entrada do sistema. Isto é, no caso do NLForSpec, o léxico contém os termos do domínio da aplicação utilizados nas descrições de casos de teste. A abordagem utilizada no NLForspec para construção do léxico foi baseada no *phrasal lexicon* [Becker, 1975], em que os termos do léxico podem ser compostos por uma ou mais palavras.

Segundo Becker, o uso da linguagem pelos seres humanos corresponde a juntar "pedaços de texto" que foram conhecidos previamente, existindo apenas um processo secundário de planejamento para adaptar esses pedaços de texto à nova situação. A complexidade dos termos de um *phrasal lexicon* pode variar desde uma agregação simples de palavras que possuem significado específico (e.g. "*White House*", "*in order to*") até termos léxicos que codificam informações gramaticais da linguagem [Zernik & Dyer, 1997].

Em NLForSpec, os termos do léxico são divididos em três categorias:

- *Nouns* (substantivos) – representam as entidades do domínio. Cada entidade possui pelo menos um nome associado. Por exemplo, para representar uma mensagem curta de texto, os seguintes textos podem ser usados: *sms*, *sms message* ou *short message*.
- *Verbs* (verbos) – representam as ações. Por exemplo, *select*, *create*, *scroll to*, *open* etc.
- *Modifiers* (modificadores) – acompanham um nome, podendo aparecer antes ou depois dele, alterando o seu significado. Por exemplo, na frase "*Delete a protected message*", o termo *protected* aparece como modificador de *message*.

Cada categoria do léxico possui um conjunto de atributos associados, que descrevem suas características. A figura 5.3 dá exemplo da representação em XML de dois substantivos do léxico.

```xml
<noun>
    <term>inbox folder</term>
    <plural/>
    <class>list</class>
    <model>INBOX_FOLDER</model>
</noun>
<noun>
    <term>contact</term>
    <plural/>
    <class>list_item</class>
    <model>CONTACT_ITEM</model>
</noun>
```

Figura 5.3: Exemplos de *Nouns* do Léxico

Na figura 5.3, observa-se a presença de dois substantivos (*nouns*): um composto por duas palavras, "*inbox folder*", e outro por uma única palavra, "*contact*". As *tags* XML que definem as características de cada *noun* são listadas a seguir.

- *term*: termo que representa o substantivo propriamente dito.
- *plural:* representa o plural do substantivo. Se estiver vazio, o plural é obtido automaticamente pelo módulo de processamento através das regras de obtenção do plural na língua inglesa.
- *class:* representa a classe da ontologia à qual o substantivo pertence.
- *model:* funciona como uma espécie de código de identificação do substantivo e será utilizado para representar o substantivo na especificação a ser gerada nas últimas etapas de processamento.

O segundo tipo de termo do léxico é o verbo. A figura 5.4 a seguir traz um exemplo da representação em XML de dois verbos.

```xml
<verb>
    <term>accept</term>
    <past/>
    <participle/>
    <gerund/>
    <thirdperson/>
</verb>
<verb>
    <term>set</term>
    <past>set</past>
    <participle>set</participle>
    <gerund>setting</gerund>
    <thirdperson/>
</verb>
```

Figura 5.4: Verbos do Léxico

Na figura 5.4 observa-se a presença de dois verbos: um regular (*accept*) e outro irregular (*set*). A diferença na representação desses dois verbos está no fato de para o verbo irregular ser necessário informar todos os seus tempos verbais (*past, participle* e *gerund*), enquanto que para o verbo regular esses tempos verbais são obtidos automaticamente (em geral com a adição do sufixo *-ed*). Já para a terceira pessoa, em ambos os casos do exemplo é necessário apenas o acréscimo do sufixo *–s*.

O terceiro tipo de termo do léxico são os modificadores (*modifiers*). Conforme dito anteriormente, esses termos aparecem antes ou depois dos substantivos, modificando o seu significado. Por exemplo, na sentença "*Select an unprotected message*" o termo "*unprotected*" modifica "*message*", indicando que a mensagem a ser selecionada não é qualquer mensagem, e sim uma mensagem não-protegida. A figura 5.5 traz mais dois exemplos de modificadores.

```
<modifier>
    <term>blank</term>
    <position>before</position>
    <model>BLANK</model>
</modifier>
<modifier>
    <term>with a predefined animation</term>
    <position>after</position>
    <model>WITH_A_PREDEFINED_ANIMATION</model>
</modifier>
```

Figura 5.5: Modificadores do Léxico

Na figura 5.5 observa-se a presença dos modificadores "*blank*" e "*with a predefined animation*". Como características dos modificadores, temos a sua posição em relação ao substantivo que ele modifica (*tag position*) e a sua identificação que será utilizada posteriormente para construção do modelo da especificação (*tag model*).

Além dos tipos de modificadores exibidos na figura 5.5, existem também os chamados modificadores parametrizados, que são modificadores cujos termos possuem em suas definições *tags* que indicam que naquele local do termo deve ocorrer a substituição da *tag* por um outro termo. A Figura 5.6 a seguir traz exemplos de modificadores com essa característica.

```
<modifier>
    <term>with <int/> bytes</term>
    <position>after</position>
    <model>WITH_N_BYTES.Int</model>
</modifier>
<modifier>
    <term>store in <noun/></term>
    <position>after</position>
    <model>STORED_IN.Hardware</model>
</modifier>
```

Figura 5.6: Modificadores parametrizados

Nos primeiro dos modificadores da figura 5.6, observa-se o termo "*with <int/> bytes*". A presença da *tag* *<int/>* indica que naquele local deve ocorrer a presença de um termo correspondente a um número inteiro. Por exemplo, na sentença "*Select a picture with two bytes"*, o termo *two* corresponderia à *tag* *<int/>*. Já no segundo *modifier*, observa-se o termo "*stored in <noun/>*", que indica que o *modifier* é composto por "*stored in*", seguido da presença de uma *noun.* Por exemplo, na sentença "*Select a picture stored in memory card*", essa *noun* seria "*memory card*".

5.2.3. Gramática de Casos

A terceira base de conhecimento utilizada em NLForSpec é a Gramática de Casos. Conforme descrito na seção 2.2.5, o objetivo principal da gramática de casos é atribuir papéis temáticos aos constituintes das frases. Para isso, o verbo, o constituinte central da frase, determina os papéis temáticos que servem de argumento para ele.

A Gramática de Casos é bastante adequada à aplicação em foco: descrições de casos de teste. As sentenças que descrevem os procedimentos do caso de teste são isoladas e possuem composição regular e previsível (ver seção 4.2), tornando apropriada a aplicação da Gramática de Casos [Efe & Ng, 1987]. Outra vantagem da Gramática de Casos é a facilidade de extensão, através da adição de novos *case frames* à base da gramática de casos. Com isso, pode-se ter um conjunto inicial de *case frames*, e, quando necessário, novos *case frames* são adicionados para especificar as novas construções frasais de casos de teste.

No NLForSpec, a Gramática de Casos consiste em um conjunto de *case frames* que contêm informações sobre os verbos do léxico e seus papéis temáticos. Esses *case frames* representam o conhecimento semântico lingüístico, enquanto que a ontologia representa o conhecimento semântico do domínio. Cada *case frame* agrupa verbos que

possuem em comum o mesmo significado e os mesmos papéis temáticos. A figura 5.7 traz exemplo de um *case frame*.

Um ponto importante a salientar é que a abordagem utilizada para definição dos *cases frames* difere da abordagem tradicional utilizada pelo Projeto *FrameNet* [Baker *et al.*, 1998], em que cada *case frame* contém um conjunto de verbos com os mesmos papéis temáticos, mas não necessariamente com o mesmo significado (*e.g.*, os verbos "*rent*" e "*buy*" do *frame "Commerce_buy"*).

```
<frame>
   <name>SelectItem</name>
   <verblist>
      <verb>select</verb>
      <verb>choose</verb>
   </verblist>
   <roles>
      <role mandatory="true">agent</role>
      <role mandatory="true">theme</role>
      <role mandatory="false">from-loc</role>
   </roles>
</frame>
```

Figura 5.7: Exemplo de *case frame*

No *case frame* da figura 5.7, observa-se a presença de dois verbos na lista de verbos (*tag* <*verblist*>): "*select*" e "*choose*". Esses dois verbos estão agrupados no mesmo *case frame,* pois, para o domínio da aplicação, eles possuem o mesmo significado e também compartilham os mesmos papéis temáticos (*tag* <*role*>), definidos com base em Allen (1995). Para cada papel temático, têm-se: o nome do papel temático, que, no caso da figura 5.7, são *agent*, *theme* e *from-loc*; e um atributo (*mandatory*), que indica se para aquele *case frame* é obrigatória ou não a presença do papel temático. Por exemplo, na sentença de um caso de teste "*Select a message from inbox folder*", os três papéis temáticos estão presentes:

- o agente *(agent),* definido de forma implícita, pois o executor do caso de teste seria o agente da ação;
- o tema (*theme*), que seria *"message"*;
- e o local (*from-loc*), de onde o tema foi selecionado, no caso "*inbox folder*".

Já para a sentença "*Choose a contact*", observa-se a presença de apenas dois papéis temáticos:

- o agente, mais uma vez implicitamente definido;
- e o tema, que seria "*contact*".

Isto pode ocorrer, pois na definição do *case frame* o papel temático "*from-loc*" não é obrigatório.

Restrições da Gramática

Além dos *case frames,* a gramática de casos ainda contém um conjunto de restrições que limita as possíveis entidades que podem preencher os papéis temáticos de cada *case frame*. Para isso, uma classe da ontologia é associada a cada papel temático. Cada possível combinação entre papéis temáticos e classes da ontologia deve ser especificada como uma restrição do *case frame*. O uso dessas restrições amarra o texto de entrada, não permitindo que sentenças semanticamente incorretas sejam interpretadas. A figura 5.8 exibe um conjunto de restrições relacionadas ao *case frame* da figura 5.7.

```
<frame>
   <name>SelectItem</name>
   <restrictions>
      <restriction name="DTSEL_MENUITEM_MENU">
         <class role="theme">menu_item</class>
         <class role="from-loc">menu</class>
      </restriction>
      <restriction name="DTSEL_SENDABLEITEM">
         <class role="theme">sendable_item</class>
      </restriction>
      <restriction name="DTSEL_ITEM_LIST">
         <class role="theme">item</class>
         <class role="from-loc">list</class>
      </restriction>
   </restrictions>
</frame>
```

Figura 5.8: Exemplo de Restrições de um *case frame*

Como pode ser observado, três restrições (*tag <restriction>*) foram especificadas para o *case frame* de nome *SelectItem*. Cada restrição é identificada unicamente para o *case frame* pelo atributo *name* (nome). Em cada restrição, os papéis temáticos são relacionados às classes da ontologia (*tag <class>*). Por exemplo, na figura 5.8, a restrição de nome "*DTSEL_MENUITEM_MENU*" restringe o uso do papel temático *theme* à classe *menu_item* da ontologia e do papel temático *from-loc* à classe *menu*. Já na restrição "*DT_SEL_SENDABLEITEM*" apenas o papel temático *theme* é relacionado

à classe *sendable_item.* Ou seja, apenas os papéis temáticos obrigatórios no *case frame* devem ser obrigatoriamente relacionados na restrição a uma classe da ontologia.

Conforme já foi dito, as restrições de um *case frame* não permitem que sentenças semanticamente incorretas para o domínio sejam interpretadas. Por exemplo, a seguir têm-se duas sentenças com os papéis temáticos e classes da ontologia associadas:

- *"Select 'Send a Message Option' in Message Center"*
  - *theme*: *Send a Message Option*, que possui como classe na ontologia *menu_item*
  - *from-loc: Message Center*, que possui como classe na ontologia *menu*
- *"Select a memory card from Message Inbox"*
  - *theme*: *memory card*, que possui como classe na ontologia *hardware*
  - *from-loc: Message Inbox,* que possui como classe na ontologia *list.*

Ambas as sentenças estão gramaticalmente corretas, porém apenas a primeira sentença seria considerada semanticamente válida. Isto se deve ao fato de, para a primeira sentença, existir uma restrição (figura 5.8) para o *case frame* em que o papel temático *theme* é associado à classe da ontologia *menu_item* e o papel temático *from-loc* é associado à classe *menu* e de não existir restrição que contemple as ligações entre papéis temáticos e classes da ontologia da segunda sentença (*theme* com *hardware* e *from-loc* com *list*). Dessa forma, a segunda sentença é considerada semanticamente inválida para o domínio, pois não se pode selecionar um *memory card* de um *Message Inbox.*

5.2.4. Base de Especificações CSP

Nesta seção, detalharemos a base de dados que armazena informações necessárias para a geração das especificações formais de saída do sistema. Para facilitar o entendimento, é necessário antes abordar como as ações dos casos de teste são especificadas em CSP.

Como dito na seção 4.3, eventos CSP são abstrações de ações do mundo real. No nosso contexto, eventos CSP são considerados abstrações das condições iniciais, dos passos, dos resultados esperados e das condições finais de um caso de teste. Por exemplo, o evento *select.Option* pode ser visto como a representação em CSP do passo "*Select an option*" de um caso de teste. Ou seja, o passo de caso de teste foi especificado pelo canal tipado *select*, representando a ação selecionar, que transmitiu o dado *Option* (opção). Essa é a forma na qual as ações de um caso de teste são representadas em CSP. Até mesmo as condições iniciais e finais de um caso de teste são consideradas como ações. Por exemplo, a sentença de uma condição inicial "*Phone is in Idle Screen*" seria considerada uma ação "*Go to Idle Screen*" (*go.IdleScreen*, em CSP). O mesmo ocorre para os resultados esperados.

A seguir, detalharemos como os tipos de dados CSP (*datatypes*) são especificados a partir da ontologia. Os *datatypes* são utilizados para especificar os tipos de dados que podem ser transmitidos pelos canais. Em seguida, apresentaremos a relação entre os *case frames* de NLForSpec e os canais CSP.

Classes da Ontologia X *Datatypes*

No NLForSpec, os *datatypes* CSP são definidos a partir das classes da ontologia, existindo um *datatype* para cada classe. A figura 5.9 (a) mostra um conjunto de *datatypes* representados em XML. No lado direito, a figura 5.9 (b) exibe a definição em CSP dos *datatypes*. Note que a definição dos *datatypes* segue a estrutura em árvore da ontologia (ver figura 5.2).

(a)

```
<datatype class="screen">
     <label>SCREEN</label>
     <name>Screen</name>
</datatype>
<datatype class="form">
     <label>FORM</label>
     <name>Form</name>
</datatype>
<datatype class="menu">
     <label>MENU</label>
     <name>Menu</name>
</datatype>
<datatype class="list">
     <label>LIST</label>
     <name>List</name>
</datatype>
<datatype class="dialog">
     <label>DIALOG</label>
     <name>Dialog</name>
</datatype>
```

(b)

```
datatype Screen =
          FORM.Form
          | MENU.Menu
          | LIST.List
          | DIALOG.Dialog
          | screen1 | screen2 …
datatype Form = form1 | form2 | …
datatype Menu = menu1 | menu2 | …
datatype List = list1 | list2 | …
datatype Dialog = dialog1 | dialog2 | …
```

Figura5.9: Exemplo de *datatype* CSP

Para especificar os modificadores do Léxico, foi criado um *datatype* especial, que é independente dos *datatypes* definidos a partir da ontologia.

***Case Frames* X Canais CSP**

Cada *case frame* da gramática corresponde a um canal tipado na especificação CSP. Os papéis temáticos do *case frame*, por sua vez, correspondem aos parâmetros do canal CSP. Cada papel e suas restrições (definidas via Ontologia) definem os tipos de dados (*datatypes*) do parâmetro associado. Por exemplo, o canal CSP que representa o "*case frame*" da figura 5.7 pode receber até dois parâmetros: um especificando o papel temático "*theme*" e outro o papel temático "*from-loc*".

Entretanto, um papel temático é composto por um nome e um conjunto de modificadores. Por exemplo, o *case frame* (ver figura 5.7) que especifica a ação "*select*" da sentença "*Select some unprotected messages*" tem o papel temático "*theme*" formado pelo nome "*message*" e pelos modificadores "*some*" e "*unprotected*". Com isso, para refletir essa característica na especificação, os canais CSP recebem tuplas de dois argumentos: o primeiro é o nome e o segundo é o conjunto de modificadores. Para o exemplo de sentença anterior, a especificação em CSP seria *select.(MESSAGE, {SOME, UNPROTECTED})*.

A figura 5.10 (a) mostra como um canal é representado na base de conhecimento (formato XML). O canal representa em CSP o *case frame* da figura 5.7. Na figura 5.10 (b), é exibida a definição do canal em CSP. O *datatype* auxiliar *DTSelect* é criado automaticamente a partir das restrições do *case frame*. Note que os nomes das restrições (figura 5.8) são usados na definição do canal.

(a)

```
<channel>
    <name>select</name>
    <casegrammar>SelectItem</casegrammar>
    <datatype>DTSelect</datatype>
</channel>
```

(b)

```
channel select : DTSelect
datatype DTSelect =
        DTSEL_ITEM.(Item, Set(Modifier))
        | DTSEL_LISTITEM_LIST.(ListItem, Set(Modifier)).(List, Set(Modifier))
        | DTSEL_MENUITEM_MENU.(MenuItem, Set(Modifier)).(Menu, Set(Modifier))
```

Figura 5.10: Exemplo de canal CSP

Representando os Casos de Teste em CSP

Já definido como as ações de um caso de teste são modeladas em eventos CSP, é necessário mostrar como eles são reunidos de modo a formar um caso de teste. Cada caso de teste é representado por um processo composto pela seqüência de eventos que representam as ações do caso de teste. Por exemplo, "*TestCase_1 = evento1 → evento2 → evento3 ... → SKIP*". Porém, é necessário especificar no CSP cada seção do caso de teste. Para isso, os eventos de controle *initialConditions*, *steps*, *expectedResults* e *finalConditions* foram definidos, representando, respectivamente, pré-condições, passos, resultados esperados e pós-condições (ver seção 4.2). Os eventos que, no processo CSP, seguem os eventos de controle ficam fazendo parte da seção que o evento de controle especifica. Por exemplo, no caso de teste da figura 5.11, os eventos *evento1* e *evento2* fazem parte das pré-condições, o evento *evento3* é um passo, enquanto o *evento4* representa o resultado esperado para o passo do *evento3*. Por fim, o *evento5* faz parte das pós-condições.

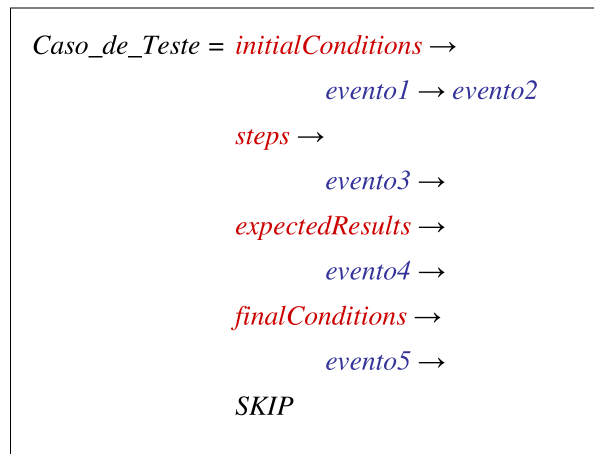


Figura 5.11: Ilustração de caso de teste em CSP

5.3. Módulos de Processamento

Nesta seção, serão detalhados os três módulos de processamento que compõem a ferramenta NLForSpec. Para ilustrar o processamento e as representações de entrada/saída de cada módulo, a sentença "*Select at least 3 messages from Inbox Folder*" servirá como exemplo de entrada do sistema. O "passo a passo" do processamento dessa sentença será detalhado em cada um dos módulos.

5.3.1. POS-Tagging

O primeiro módulo de processamento de NLForSpec é responsável por identificar a categoria gramatical de cada palavra da sentença de entrada. Essa tarefa é realizada com base na técnica de *POS-Tagging* [Brill, 2000].

POS-Tagging pode ser visto como uma alternativa à análise sintática tradicional da abordagem simbólica de PLN, que é realizada por *parsers* baseados em gramáticas (seção 2.2.3). A principal diferença entre *parsing* e *tagging* é que, enquanto o *parsing* tem como objetivo determinar a estrutura sintática da sentença completa, *tagging* tem o objetivo menos ambicioso de identificar a categoria gramatical de cada palavra da sentença [Somers, 2000]. Porém, em virtude da ambigüidade léxica, a determinação correta da categoria gramatical de uma palavra pode depender também das categorias das palavras vizinhas. Assim, a determinação da categoria gramatical de cada palavra não é feita de forma isolada, tendo que levar em consideração a categoria das palavras vizinhas a ela.

Neste módulo de NLForSpec, a determinação da categoria gramatical das palavras da sentença de entrada é feita em duas etapas: (1) POS-Tagging geral e (2) POS-Tagging específico para o domínio da aplicação.

A primeira etapa de processamento é realizada por um POS-Tagger de propósito geral, o *Stanford POS-Tagger* [Toutanova *et al.,* 2003]. Essa ferramenta foi selecionada com base em uma análise de diversos POS-Taggers para a língua inglesa disponíveis na Web para uso gratuito. O processo de avaliação do POS-Tagger mais adequado para as necessidades de NLForSpec está descrito no capítulo 6 (Protótipo e Experimentos).

As categorias gramaticais utilizadas pelo *Stanford Pos-Tagger* são baseadas no *Penn Treebank Tagset* [Marcus *et al.*, 1993], um conjunto de *tags* que é bastante utilizado pelos sistemas de PLN. O quadro 5.1 traz alguns exemplos das *tags* definidas no *Penn Treebank Tagset*.

Quadro 5.1: Exemplo de *tags* do *Penn Trrebank Tagset*

POS Tag	**Description**	**Example**
CC	*coordinating conjunction*	*And*
CD	*cardinal number*	*1, third*
DT	*determiner*	*the*
EX	*existential there*	*there is*
FW	*foreign word*	*d'hoevre*
IN	*preposition/subordinating conjunction*	*in, of, like*
JJ	*adjective*	*green*
JJR	*adjective, comparative*	*greener*
JJS	*adjective, superlative*	*greenest*
NN	*noun, singular or mass*	*table*
NNS	*noun plural*	*tables*
NNP	*proper noun, singular*	*John*
NNPS	*proper noun, plural*	*Vikings*
PDT	*predeterminer*	*both the boys*
RP	*particle*	*Give up*
TO	*to*	*to go, to him*
UH	*interjection*	*uhhuhhuhh*
VB	*verb, base form*	*take*
VBD	*verb, past tense*	*took*
VBG	*verb, gerund/present participle*	*taking*
VBN	*verb, past participle*	*taken*
VBP	*verb, sing. present, non-3d*	*take*
VBZ	*verb, 3rd person sing. present*	*takes*

A seguir, temos um exemplo do processamento dessa primeira etapa de POS-tagging para a sentença de entrada "*Select at least 3 messages from Inbox Folder*". Essa

etapa recebe a sentença de entrada e retorna a sentença "anotada" com as *tags*, conforme mostrado no Quadro 5.2 a seguir.

Quadro 5.2: Entrada e Saída após 1ª Etapa do *POS-Tagger*

<table>
<tr><td>Sentença de entrada</td><td colspan="2">Select at least 3 messages from Inbox Folder</td></tr>
<tr><td rowspan="2">POS-Tagging</td><td>1ª Etapa</td><td>Select/VB at/IN least/JJS 3/CD messages/NNS from/IN Inbox/NN Folder/NN</td></tr>
<tr><td>2ª Etapa</td><td></td></tr>
</table>

A segunda etapa de processamento deste módulo consiste em um POS-Tagging específico para o domínio da aplicação, e é realizada utilizando-se a base de conhecimento léxico (seção 5.2.2). Essa segunda etapa recebe a sentença de entrada com as *tags* geradas pelo POS-Tagger de propósito geral, e tem como objetivo principal identificar nessa representação os três tipos de termos do léxico (*Verbs, Nouns* e *Modifiers*).

Para o caso dos verbos (*Verbs*), em geral, essa tarefa é simples, consistindo em identificar na representação recebida da primeira etapa as palavras (ou termos) marcadas com as *tags* do *Penn Treebank* que indicam formas verbais (*tags* VB, VBD, VBG, VBN, VBP e VBZ). Após identificar esses termos, basta verificar se existe um verbo correspondente no léxico de NLForSpec. No caso da sentença exemplo, o termo "*Select*" (marcado com a *tag* VB) seria identificado como verbo, dado que existe uma entrada correspondente a ele no léxico da nossa ferramenta.

No caso dos substantivos (*Nouns*), pode ser necessário um processamento mais sofisticado. Como estamos trabalhando com um Léxico Frasal (*phrasal lexicon),* as entradas correspondentes a substantivos podem ser compostas por mais de uma palavra, sendo termos compostos. Assim, a busca não consiste simplesmente em verificar, uma a uma, as palavras marcadas com *tags* que indicam substantivos (*tags* NN, NNS, NNP e NNPS), e sim em buscar possíveis termos compostos. Na sentença exemplo, os substantivos identificados no léxico seriam "*message*" e "*Inbox Folder*".

Por fim, é necessário identificar possíveis ocorrências de modificadores (*modifiers*) na sentença. Para isso, é feita uma busca na sentença pelos modificadores presentes no léxico, levando-se em consideração a posição dos modificadores em

relação aos substantivos que eles modificam. É necessário também verificar os modificadores parametrizados (seção 5.2.2). Para a sentença exemplo, o modificador identificado seria "*at least 3*", que corresponde ao modificador parametrizado "*at least <int/>*" da base de conhecimento léxico. O quadro 5.3 mostra como ficaria a representação da sentença após a segunda etapa do POS-Tagging.

Quadro 5.3: Saída após 2ª Etapa do *POS-Tagging*

Sentença de entrada	*Select at least 3 messages from Inbox Folder*	
POS-Tagging	1ª Etapa	Select/VB at/IN least/JJS 3/CD messages/NNS from/IN Inbox/NN Folder/NN
	2ª Etapa	Select/VB "at least 3"/Modifier messages/Noun from/IN "Inbox Folder"/Noun

Um ponto importante a ser discutido é que nem sempre as *tags* indicadas pelo POS-Tagger de propósito geral estão corretas. Dessa forma, a segunda etapa de processamento também é responsável por corrigir essas falhas. Por exemplo, em alguns casos nenhum verbo é identificado pelo POS-Tagger na primeira etapa, cabendo à segunda etapa tentar identificar esse verbo. Caso o POS-Tagging específico também não consiga identificar nenhum verbo, a sentença não é processada. Lembramos que as sentenças do nosso domínio representam casos de teste e, portanto, sempre devem ter um verbo, que indica a ação a realizar.

5.3.2. Processamento Semântico

O módulo de processamento semântico tem como objetivo identificar, na gramática de casos, o *case frame* correspondente à sentença, bem como os papéis temáticos dos constituintes dessa sentença. A representação recebida por esse módulo é uma sentença com as *tags* indicadas pela segunda etapa do módulo POS-Tagging.

Ao receber a representação gerada pelo módulo anterior, o módulo de processamento semântico inicialmente identifica na gramática de casos os *case frames* que contêm o verbo da sentença. Vale salientar que um verbo pode estar associado a mais de um *case frame*, pois um mesmo verbo pode assumir significados diferentes, dependendo do seu contexto de uso. Por exemplo, o verbo "*enter*" tem o significado de

acessar na sentença “*Enter in Message Center Menu*”, devendo aparecer no mesmo *case frame* dos verbos “*acess”* e “*go to”*. Já na sentença “*Enter some data in the message field*”, o verbo “*enter”* tem o significado de *digitar*, *preencher*, devendo aparecer no mesmo *case frame* que os verbos “*type”* e “*fill”*.

Uma vez selecionados todos os *case frames* que contêm o verbo da sentença de entrada, esse módulo tenta identificar qual deles é o mais adequado para descrever essa sentença. Para isso, ele tenta relacionar os substantivos (*nouns*) identificados na sentença com os papéis temáticos do *case frame*, verificando também se existe uma restrição (associada ao *case frame*) que satisfaça os relacionamentos entre os papéis temáticos e as classes da ontologia.

Seguindo com nosso exemplo, considere a representação da sentença recebida do módulo anterior: “*Select/VB ‘at least 3’/Modifier messages/Noun from/IN ‘Inbox Folder’/Noun*”. O primeiro passo desse módulo seria selecionar o *case frame* que contém o verbo *select*. Em seguida, é feito o mapeamento entre os substantivos identificados na sentença e os papéis temáticos. Na sentença exemplo, “*message”* é associado ao papel temático *theme*, e “*Inbox Folder*” é associado a *from-loc*. Feito isso, o módulo de processamento semântico obtém as classes da ontologia de cada um dos substantivos associados aos papéis temáticos, e verifica se existe uma restrição que os satisfaça. Em seguida, os modificadores são associados aos substantivos que eles modificam, e é montada uma estrutura semelhante à mostrada no quadro 5.4, que servirá de entrada para o próximo módulo de NLForSpec.

Quadro 5.4: Saída após o processamento semântico

Sentença de entrada	*Select at least 3 messages from Inbox Folder*	
POS-Tagging	1ª Etapa	Select/VB at/IN least/JJS 3/CD messages/NNS from/IN Inbox/NN Folder/NN
	2ª Etapa	Select/VB “at least 3”/Modifier messages/Noun from/IN “Inbox Folder”/Noun
Processamento Semântico	Case Frame: • Verb – select • Theme – message o Modifiers - at least 3	

	• From-loc : Inbox Folder o Modifiers – { }

5.3.3. Geração de Casos de Teste Formais

O último módulo de processamento de NLForSpec é o de Geração de Casos de Teste (CTs) Formais. Esse módulo é responsável por mapear cada estrutura de *case frame* identificada no módulo anterior em um evento CSP. Para realizar esse mapeamento, é utilizada a base de especificações CSP.

Conforme dito anteriormente, um evento CSP é composto por um canal e pelos *datatypes* que serão transmitidos por esse canal. Dessa forma, o primeiro passo do módulo de geração de casos de teste formais é mapear o *case frame* selecionado no canal correspondente da base de especificações CSP. Em seguida, a restrição associada ao *case frame* é identificada e os *datatypes* associados aos substantivos que compõem os papéis temáticos são localizados na base de especificações CSP, assim como os modificadores de cada substantivo.

Seguindo com nosso exemplo, nesse módulo, o canal CSP associado ao *case frame* selecionado (quadro 5.3) é localizado na base de especificações CSP (*channel select*) e em seguida é localizada a restrição corresponde à associação entre papéis temáticos e classes da ontologia (no caso, é a restrição DTSEL_ITEM_LIST). Feito isso, os substantivos são mapeados nos *datatypes* correspondentes ("*message*" é mapeado em MESSAGE e "*Inbox Folder*" é mapeado em INBOX_FOLDER). O mesmo ocorre com os modificadores ("*at least 3*" é mapeado em AT_LEAST.3). Por fim, é formado o evento exibido no quadro 5.5.

Quadro 5.5: Entrada e Saída após a geração do evento CSP

Sentença de entrada	*Select at least 3 messages from Inbox Folder*	
POS-Tagging	1ª Etapa	Select/VB at/IN least/JJS 3/CD messages/NNS from/IN Inbox/NN Folder/NN
	2ª Etapa	Select/VB "at least 3"/Modifier messages/Noun from/IN "Inbox Folder"/Noun
Processamento	Case Frame:	

Semântico	• Verb – select
	• Theme – message
	  o Modifiers - at least 3
	• From-loc : Inbox Folder
	  o Modifiers – { }

Geração de CTs Formais	select.DTSEL_ITEM_LIST.(MESSAGE,{AT_LEAST.3}).(INBOX_FOLDER, { })

Outra importante tarefa realizada pelo módulo de geração de CTs formais é a construção do cabeçalho CSP, que contém as declarações dos canais e *datatypes* utilizados na composição dos eventos. Por exemplo, para que o evento mostrado no Quadro 5.4 seja válido, é necessário colocar no cabeçalho CSP a declaração do canal *select* e dos *datataypes* MESSAGE e INBOX_FOLDER (que são substantivos), e AT_LEAST.3 (que é modificador). A figura 5.12 a seguir traz essas declarações.

```
datatype Screen =
    LIST.List
datatype List =
    INBOX_FOLDER
datatype Item =
    LISTITEM.ListItem
datatype ListItem =
    SENDABLEITEM.SendableItem
datatype SendableItem =
    MESSAGE
datatype Modifier =
    AT_LEAST.Int
channel select : DTSelect
datatype DTSelect =

    DTSEL_ITEM_LIST.(Item, Set(Modifier)).(List, Set(Modifier))
```

Figura 5.12: Exemplo de cabeçalho CSP

Um ponto importante a ser destacado no cabeçalho CSP da figura 5.12 é que, ao se utilizar um *datatype* associado a uma classe da ontologia que é sub-classe de outra, é necessário declarar no cabeçalho também a super-classe. Por exemplo, para declarar o

datatype INBOX_FOLDER, que é da classe *List*, foi necessário antes declarar a classe *Screen*, que é super-classe de *List*.

Caso todas as sentenças que compõem um caso de teste sejam recebidas como entrada de uma só vez, o módulo de geração de CTs formais também é responsável por formar um cabeçalho único e otimizado (sem eventuais repetições de declarações nem declarações desnecessárias) que contemple todos os eventos integrantes de um caso de teste CSP.

5.4. Considerações Finais

Neste capítulo 5, foi detalhada a ferramenta NLForSpec, voltada para a geração de especificações formais a partir de descrições de casos de teste em linguagem natural. A arquitetura do sistema foi apresentada, bem como seus três módulos de processamento e as suas quatro bases de conhecimento.

Como mostrado, NLForSpec foi desenvolvido com base na abordagem simbólica tradicional de Interpretação de LN. Porém, ele não apresenta os módulos de processamento do discurso e pragmático, por não ser necessário realizar esses tipos de processamento para interpretar descrições de casos de tese.

O grande desafio no desenvolvimento do NLForSpec foi torná-lo o mais independente possível de domínio. Assim, as bases de conhecimento foram especialmente desenvolvidas para garantir essa independência e para permitir sua atualização pelos usuários da ferramenta. O desenvolvimento de uma interface gráfica para a manutenção das bases de conhecimento aparece como trabalho fundamental no intuito de facilitar a manutenção das bases.

O próximo capítulo traz o estudo de caso (Protótipo e Testes) desenvolvido para validar a proposta do NLForSpec. Um protótipo da ferramenta para o domínio de aplicações móveis será apresentado. A implementação dos módulos processadores será detalhada, bem como as bases de conhecimento utilizadas. Também serão apresentados os experimentos realizados para validar e os resultados obtidos.

6. Protótipo e Experimentos

Neste capítulo, serão apresentados o protótipo implementado (seção 6.1) e o estudo de caso realizado para validação da ferramenta NLForSpec (seção 6.2). O domínio escolhido para os experimentos foi o de aplicações para telefones móveis, em particular aplicações de *Messaging* (envio, recebimento e manipulação de mensagens).

6.1. Protótipo Implementado

O protótipo de NLForSpec foi implementado na linguagem Java[4], para garantir a independência de plataforma e para facilitar a integração com outras ferramentas do *Test Research Project*, também implementadas em Java. A metodologia de desenvolvimento de software utilizada foi baseada em *eXtreme Programming* (XP) [Beck, 1999]. Um esforço relativamente pequeno foi empregado para a criação de uma primeira versão do projeto da ferramenta, e então iniciou-se o processo de implementação com *refactoring* [Fowler, 2000]. Dessa forma, evitou-se o risco de algum erro no projeto inicial causar grandes problemas no decorrer da implementação da ferramenta, pois esse projeto inicial estava em processo constante de adaptação.

A implementação do protótipo pode ser dividida na implementação das bases de conhecimento e dos módulos de processamento. Além disso, a implementação do protótipo de NLForSpec também incluiu a escolha de um POS-Tagger de propósito geral para realizar a primeira etapa de processamento do módulo POS-Tagger (seção 5.3.1). Nessa seção, serão discutidas primeiramente a implementação das bases de conhecimento e dos módulos de processamento. Em seguida serão detalhados os experimentos realizados para escolha do POS-Tagger.

[4] http://java.sun.com/

6.1.1. Bases de Conhecimento

Esta seção apresenta as bases de conhecimento, detalhando as classes *Java* que implementam as entidades dessas bases (temos léxicos, *case frames*, classes da ontologia etc.). Também será apresentado o mecanismo utilizado no protótipo para a manipulação das bases.

Na primeira versão do protótipo de NLForSpec, optou-se por utilizar o formato XML para representação das bases de conhecimento. Um dos motivos que influenciou na escolha de XML foram as APIs (*Application Programming Interface*) Java existentes para manipulação de documentos XML. A API escolhida para a utilização na implementação do protótipo foi a *Document Object Model* (DOM) [5]. O uso desse tipo de API facilitou bastante as tarefas de inserção, consulta e atualização de entidades nas bases de conhecimento.

Para a implementação das classes de manipulação das bases de conhecimento foi seguido o padrão PDC (*Persistent Data Collections*) [Massoni et al., 2001]. Este padrão define duas camadas de software para manipulação dos dados:

(1) Camada de Negócios – compreende as funcionalidades ligadas à aplicação. Nesta camada, métodos de alto nível são definidos para a inserção, remoção e atualização das entidades da base. Aqui são realizados testes de pré-condições e verificações de regras de negócio da aplicação.

(2) Camada de Persistência (repositório) – responsável por acessar diretamente a base de dados, sendo totalmente dependente da forma de armazenamento dos dados. Esta camada deve implementar uma interface pré-definida de métodos de acesso à base.

O uso do padrão PDC facilita a portabilidade do NLForSpec para outra forma de armazenamento de dados, visto que é apenas necessário reimplementar a camada de persistência, seguindo a interface de métodos pré-definida. O restante da implementação do NLForSpec permanece o mesmo, uma vez que ela depende apenas na interface de métodos. Por exemplo, caso seja necessário mudar a forma de armazenamento de XML para um banco de dados relacional, é preciso apenas criar uma nova classe na camada

[5] http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html

de persistência que acesse o banco de dados e que implemente os métodos da interface pré-definida.

A seguir, a implementação de cada base de conhecimento será detalhada. Serão apresentadas as classes Java que representam as informações das bases, bem como detalhes do seu desenvolvimento.

Ontologia

A figura 6.1 a seguir exibe o diagrama de classes da Ontologia.

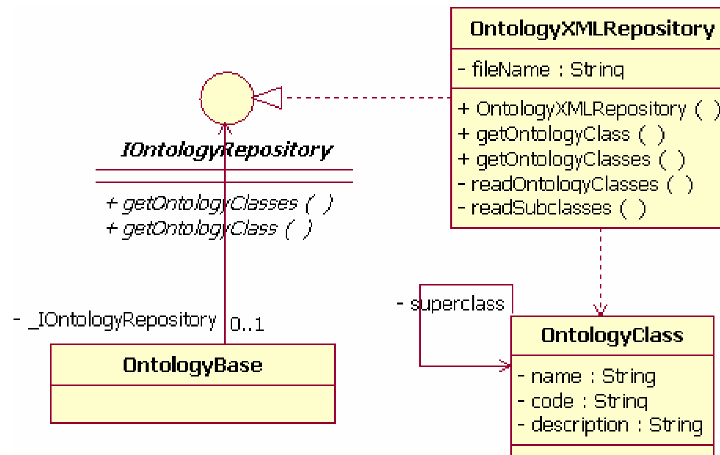


Figura 6.1: Diagrama de classes da Ontologia

Na figura 6.1, observa-se a presença de quatro classes. *OntologyClass* é a classe básica da Ontologia e representa cada entidade dessa base de conhecimento. O relacionamento *superclass* indica se existe uma outra classe da Ontologia que é superclasse desta. As outras três classes do digrama estão implementadas conforme o padrão PDC descrito anteriormente. A interface *IOntologyRepository* contém os métodos que serão implementados pela classe de acesso à camada de persistência *OntologyXMLRepository*. Por fim, a classe *OntologyBase* representa a classe da camada de negócios do padrão PDC.

Alguns métodos e atributos das classes foram ocultados do diagrama para não torná-lo muito complexo. Para as bases de conhecimento que serão explicadas a seguir, os diagramas conterão apenas as classes básicas de cada base de conhecimento. As classes das camadas de negócio e de persistência que seguem o padrão PDC não serão exibidas, pois são bastante semelhantes às da Ontologia, tornando-se assim desnecessário abordá-las novamente.

Léxico

A figura 6.2 a seguir traz o diagrama de classes da base de conhecimento léxico.

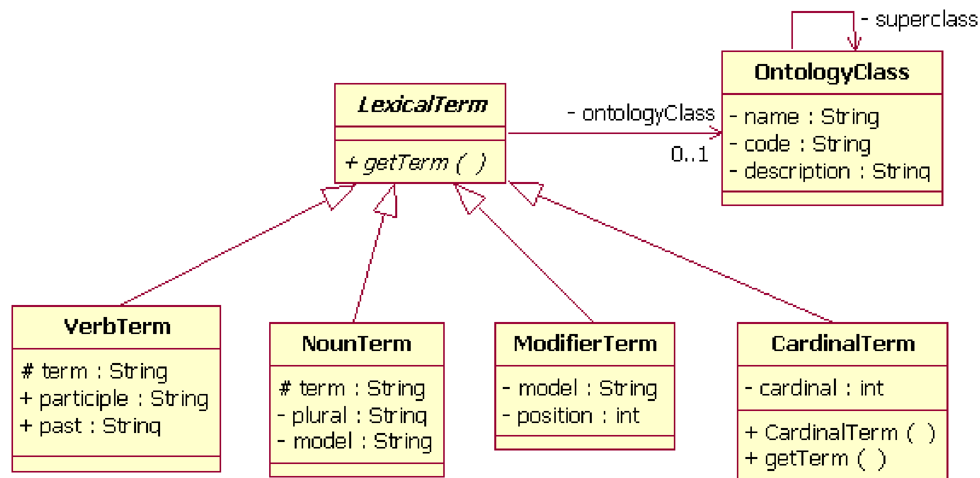


Figura 6.2: Diagrama de classes do Léxico

Na figura 6.2 observa-se a presença da classe abstrata *LexicalTerm*, que representa os termos do léxico. As quatros classes que herdam de *LexicalTerm* representam os 3 tipos de termos encontrados no léxico (*Verb*, *Noun* e *Modifier*), conforme descrito na seção 5.2.2, mais o termo cardinal. Este termo, apesar de não estar armazenado na base de conhecimento, foi acrescentado como um tipo de termo do léxico, pois eles são utilizados pelos modificadores parametrizados (seção 5.2.2). O termo cardinal é identificado na sentença pelo POS-Tagger com a *tag* "CD" e então é instanciado em um objeto da classe *CardinalTerm*. A classe *OntologyClass*, oriunda da ontologia, indica a qual classe da ontologia o termo léxico é relacionado.

Um problema encontrado durante o desenvolvimento do Léxico foi a representação do verbo *to be*, de conjugação irregular. Como mostrado na seção 5.2.2, as informações lingüísticas do verbo que são armazenadas são o infinitivo (*term*), passado (*past*), particípio (*participle*), gerúndio (*gerund*) e terceira pessoa do singular (*thirdperson*). Entretanto, o verbo *to be* é uma exceção e possui mais de uma conjugação no passado (*was* e *were*) e no presente (*am*, *is* e *are*), o que não é suportado pela base. Com isso, o verbo *to be* recebeu tratamento especial na implementação do protótipo de NLForSpec, de forma a suportar todas as possíveis conjugações.

Gramática de Casos

Conforme explicado na seção 5.2.3, a gramática de casos é composta por *case frames* e suas restrições. A figura 6.3 traz o diagrama de classes para essa base de conhecimento.

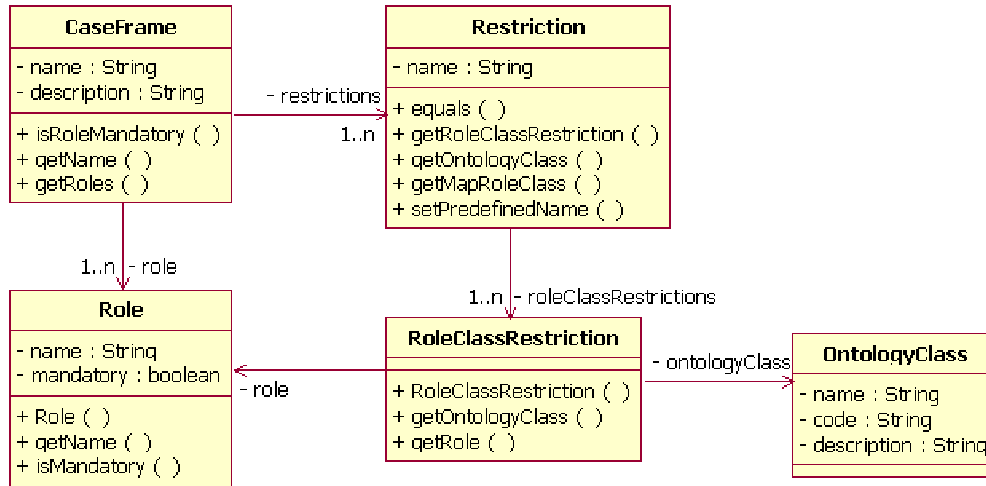


Figura 6.3: Digrama de Classes da Gramática de Casos

Na figura 6.3, observa-se a presença da classe *CaseFrame*, que, conforme o próprio nome diz, representa um *case frame* da gramática de casos. Os papéis temáticos associados a cada *case frame* são representados pela classe *Role.* As restrições de um *case frame* são representadas pela classe *Restriction*. Por fim, a classe *RoleClassRestriction* relaciona, dentro de uma restrição, um papel temático a uma classe da ontologia (*OntologyClass*).

Durante a montagem dessa base de conhecimento, o verbo *to be* novamente teve que ter um tratamento especial. No domínio do NLForSpec, dois possíveis usos do verbo *to be* foram encontrados: (1) o verbo pode ser usado para especificar o estado de alguma entidade, e.g., "*The message is available*"; (2) ou pode ser utilizado para especificar o local de alguma entidade, e.g., "*The message is in inbox folder*". Com isso, foram criados dois *case frames* diferentes para representar cada um desses tipos de sentenças. Para o caso (1), o *case frame* de nome *IsState* foi criado, com os papéis temáticos *theme* (a entidade) e *at-value* (o estado). Para o segundo caso, foi criado o *case frame IsLocation*, com os papéis temáticos *theme* (a entidade) e *at-loc* (a localização).

Além do caso especial para o verbo *to be*, é importante também ressaltar outro termo que aparece bastante nas sentenças de casos de teste, o *there is*, que descreve a existência de alguma entidade (e.g., "*There is a contact in the contact list*"). O termo

there is foi modelado no protótipo de NLForSpec como um verbo pertencente ao mesmo *case frame* do verbo *to exist* (existir), que possui os papéis temáticos *theme* (a entidade) e *at-loc* (o local onde ela existe).

Base de Especificações CSP

A última das quatro bases de conhecimento, a base de especificações CSP, agrupa as informações referentes aos canais e *datatypes* CSP. A figura 6.4 a seguir traz o diagrama de classes dessa base.

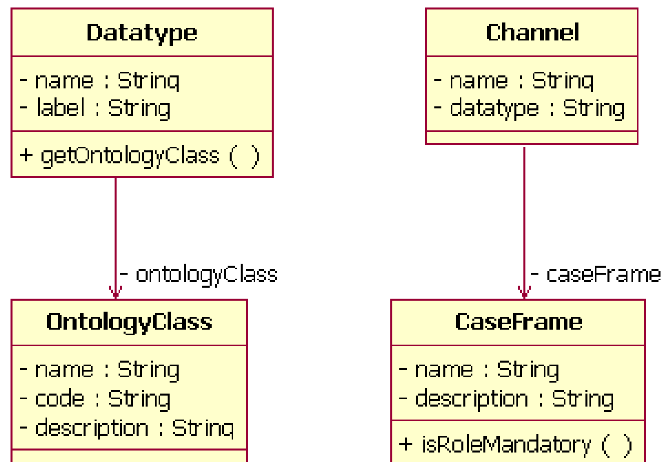


Figura 6.4: Diagrama de Classes da Base de Especificações CSP

As únicas duas classes básicas da base de especificações CSP são as classes *Datatype* e *Channel*. A primeira representa os *datatypes* utilizados nas especificações dos eventos CSP e são classificadas de acordo com as classes da ontologia (*OntologyClass*). Já a segunda representa os canais CSP e está ligada a um *case frame* (classe *CaseFrame*).

6.1.2. Módulos de Processamento

Nesta seção será discutida brevemente a implementação dos três módulos de processamento do protótipo de NLForSpec (POS-Tagging, Processamento Semântico e Geração de Casos de Teste Formais). Na implementação desses módulos, buscou-se manter o máximo de desacoplamento possível entre eles, para que, caso fosse necessária alguma modificação, não houvesse um grande impacto em todo o protótipo. A figura 6.5 a seguir traz o diagrama das classes que representam os módulos de processamento.

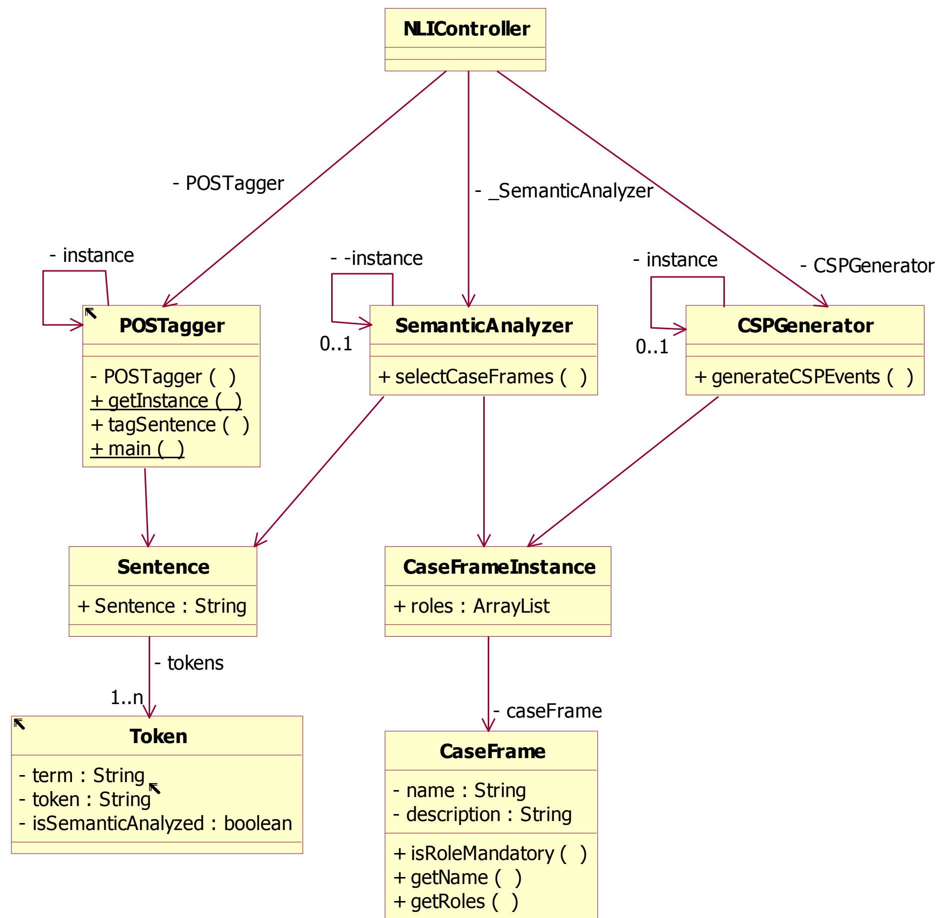


Figura 6.5: Diagrama de Classes dos Módulos de Processamento

No diagrama da figura 6.5, a classe *NLIController* representa um módulo de controle, que contém uma instância de cada um dos três módulos de processamento e é responsável por controlar as representações intermediárias trocadas entre esses módulos.

A classe *POSTagger* representa o módulo de processamento *POS-Tagging*. Essa classe recebe como entrada uma *String* com a sentença e fornece para o módulo de processamento semântico (classe *SemanticAnalyzer*) um objeto da classe *Sentence*, que contém um conjunto dos *tokens* que formam a sentença associados a sua categoria gramatical.

O módulo de processamento semântico recebe a sentença analisada pelo POS-Tagging e procura o *case frame* adequado para representação semântica da sentença. Após identificar esse *case frame,* um objeto da classe *CaseFrameInstance*, que liga os papéis temáticos aos constituintes da sentença (verbo, substantivos e seus modificadores), é instanciado e passado para o módulo de geração de casos de testes formais. Esse último módulo de processamento (classe *CSPGenerator*) é responsável

por gerar o evento CSP correspondente à instância de *case frame* recebida como entrada.

6.1.3. Escolha do POS-Tagger

Conforme explicado na seção 5.2.2, o módulo de processamento POS-Tagger é realizado em duas etapas: a primeira é realizada por um POS-Tagger de propósito geral; e a segunda por um POS-Tagger específico para o domínio da aplicação. Para o caso do POS-Tagger de propósito geral, optou-se por utilizar um POS-Tagger já existente, ao invés de implementar um do zero. Essa opção deveu-se ao fato de existirem diversos POS-Taggers de boa qualidade disponíveis para uso gratuito. Dessa forma, foram realizados experimentos para identificar qual seria o melhor POS-Tagger para o protótipo de NLForSpec.

O primeiro passo para a escolha do POS-Tagger foi a definição de critérios que guiaram a busca pelos POS-Taggers disponíveis para *download* na Web. O primeiro desses critérios trata da licença do software. O POS-Tagger deveria ser um software livre para uso acadêmico. Além disso, o segundo critério definiu que o software deveria ser implementado em Java (preferencialmente) ou ser compatível para fácil integração com ferramentas implementadas nessa linguagem. Por fim, o terceiro critério, com prioridade menor que os dois anteriores, definiu que deveriam existir relatos de experiências que comprovassem o desempenho do POS-Tagger. Após realizar a busca seguindo os três critérios, apenas três POS-Taggers foram selecionados para realização dos experimentos: Montylingua [Liu, 2004], Stanford POS-Tagger [Toutanova *et al.*, 2003] e OpenNLP [OpenNLP, 2006].

Após a escolha dos três POS-Taggers, foram selecionadas para os experimentos 450 sentenças de descrições de casos de teste dentro do domínio de *Messaging*. As sentenças foram retiradas de forma aleatória de casos de teste de diversas *features* dentro desse domínio. Feito isso, as 450 sentenças serviram como entrada para um estudo comparativo entre os três POS-Taggers. As *tags* de saída indicadas por cada um dos POS-Taggers para cada sentença foram manualmente analisadas e classificadas como corretas ou incorretas. O quadro 6.1 a seguir traz o percentual de acerto de cada um deles.

Quadro 6.1: Estudo comparativo dos POS-Taggers

POS-Tagger	Percentual de acerto
Stanford	60,47%
MontyLingua	45,99%
OPENNLP	40,57%

É importante salientar que o desempenho dos POS-Taggers nesse domínio foi inferior ao apresentado nos relatos de experiência dos autores. Isto se deve ao fato de o domínio escolhido para este estudo de caso ser bastante específico. Por isso, alguns termos utilizados (*e.g.*, *OK Key*, *Main Menu* e *MMS Message*) nas sentenças não são comumente vistos em outros domínios, o que acarretou sentenças analisadas incorretamente. Apesar disso, o desempenho do Stanford POS-Tagger foi considerado satisfatório, levando à sua escolha como POS-Tagger de propósito geral para o protótipo de NLForSpec.

Além de a questão do domínio ter influenciado no desempenho abaixo do esperado dos POS-Taggers, alguns outros motivos mais simples também exerceram influência. Por exemplo, em algumas sentenças, como "*The phone is in the Browse menu of the To: field editor*", a presença de um ":" depois da palavra "*To*" acarretava uma análise errada do POS-Tagger. Ou seja, o resultado obtido pelo Stanford POS-Tagger (60,47%) pôde ser melhorado com alguns ajustes realizados pelo módulo de processamento (seção 5.2.1). Ao ser integrado ao protótipo, com esses pequenos ajustes, o resultado do Stanford POS-Tagger foi bem superior.

6.2. Experimentos e Resultados

Nesta seção, detalharemos os dois experimentos realizados com o protótipo do NLForSpec, no *Motorola Brazil Test Center* (BTC). As descrições de casos de teste que serviram como entrada para os experimentos foram todas retiradas de casos de teste que são utilizados na prática e diariamente pelos engenheiros de testes.

O primeiro passo antes da realização dos experimentos propriamente ditos consistiu na criação e preenchimento das bases de conhecimento (seção 6.2.1). A seção 6.2.2 traz o primeiro experimento realizado com o protótipo do NLForSpec. Após a

realização desse primeiro experimento, alguns ajustes foram feitos no protótipo e um segundo experimento foi realizado (seção 6.2.3).

6.2.1. Preparação das Bases de Conhecimento

Antes de se realizar qualquer experimento com o protótipo, era necessário preencher as bases de conhecimento com um conjunto inicial de informações de forma a possibilitar a execução de experimentos.

A preparação das bases de conhecimento partiu da documentação de testes para o domínio de *Messaging* da Motorola, de onde foram selecionados aleatoriamente 428 passos de casos de teste. Esses passos de casos de teste foram analisados um a um, em uma tentativa de "simular" manualmente os objetivos de cada etapa de processamento de NLForSpec para preencher as bases de conhecimento com os termos do léxico (*nouns, verbs* e *modifiers*), classes da ontologia, *case frames* e restrições, e canais e eventos CSP. Essa tarefa contou com algumas ferramentas desenvolvidas para facilitar o preenchimento das bases e análise dos elementos identificados para cada base. A figura 6.6 a seguir ilustra a ferramenta utilizada para inserção e visualização dos termos da ontologia.

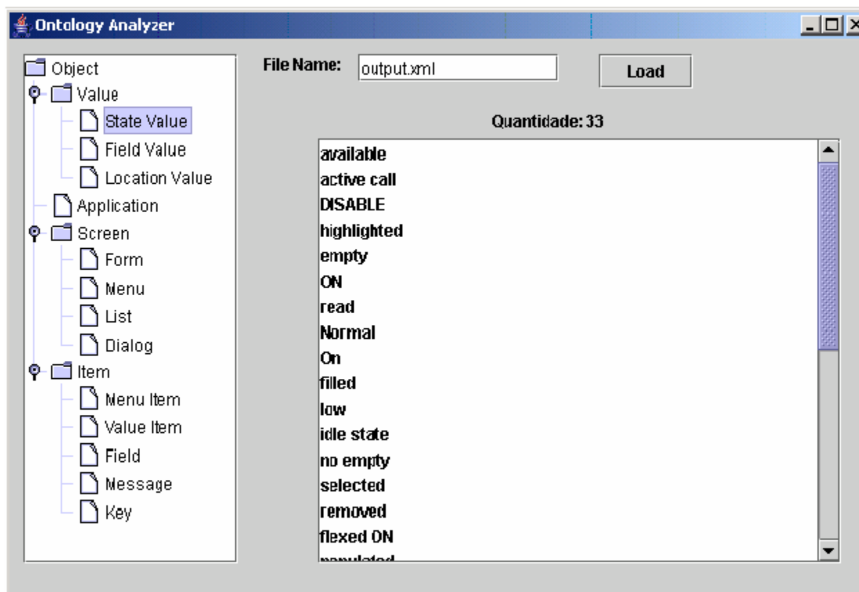


Figura 6.6: Ferramenta de análise da ontologia

A primeira etapa do processo de preparação das bases consistiu em analisar os passos de casos de teste com o objetivo de identificar e inserir na base os termos do

léxico (*nouns, verbs* e *modifiers*). A gramática de casos foi então preenchida com os *cases frames* associados aos verbos identificados na etapa anterior. Após isso, foi feita uma análise para criação das classes da ontologia que corresponderiam aos termos do léxico identificados. Em seguida, as restrições associadas a cada *case frame* foram identificadas e inseridas na base. Por fim, a base de especificações CSP foi preenchida com os canais associados aos *case frames* identificados e com os *datatypes* correspondentes aos *nouns* e *modifiers*. O quadro 6.2 a seguir traz o número de entidades inseridas nas bases.

Quadro 6.2: Números da preparação inicial das bases

Nome da Entidade		Quantidade
Léxico	*Verbs*	59
	Nouns	259
	Modifiers	39
Case Frames		35
Classes da Ontologia		18

A seguir, será detalhado o primeiro experimento realizado com o protótipo de NLForSpec e que utilizou as bases de conhecimento detalhadas nesta seção.

6.2.2. Primeiro Experimento

Após a montagem das bases de conhecimento, foi realizado o primeiro experimento com o protótipo de NLForSpec. Esse primeiro experimento ocorreu inserido em um experimento mais amplo, denominado *Motorola Big Test Brazil*, em que todos os pesquisadores envolvidos no *Test Research Project* avaliaram suas ferramentas.

Para o experimento com NLForSpec, foram selecionadas como entrada 50 descrições de casos de teste de uma *feature* do domínio de *Messaging*. Cada especificação formal de saída que NLForSpec forneceu foi manualmente analisada para verificar se estava de acordo com a descrição do caso de teste. Em seguida, todas as especificações consideradas corretas foram integradas e verificadas na ferramenta FDR (*Failures and Divergences Refinament*) [FDR, 2006]. Essa verificação no FDR serviu

para tentar identificar erros sintáticos nas especificações dos casos de testes, bem como no cabeçalho gerado. Por fim, as especificações foram fornecidas como entrada para outra ferramenta do *Test Research Project,* que gerou um modelo de uso a partir delas.

Nesse primeiro experimento, a taxa de acerto de NLForSpec foi de 62%, considerada apenas razoável. Porém, o motivo que causou esse desempenho foi a falta de informações nas bases de conhecimento. Como estas foram montadas com base em 420 passos de casos de teste de *Messaging,* e esta é apenas uma pequena parcela do total existente, vários elementos presentes nas descrições de casos de teste não foram colocados nas bases. Em virtude disso, optou-se por atualizar as bases de conhecimento com as informações presentes nas descrições dos 50 casos de teste desse primeiro experimento, e realizar um novo experimento, desta vez com o acréscimo de mais 50 casos de teste. O quadro 6.3 a seguir traz o número de elementos adicionados às bases.

Quadro 6.3: Números relacionados ao Experimento 1

<table>
<tr><th colspan="2">Nome da Entidade</th><th>Quantidade Adicionada à Base</th></tr>
<tr><td rowspan="3">Léxico</td><td>Verbs</td><td>4</td></tr>
<tr><td>Nouns</td><td>25</td></tr>
<tr><td>Modifiers</td><td>6</td></tr>
<tr><td colspan="2">Case Frames</td><td>4</td></tr>
<tr><td colspan="2">Classes da Ontologia</td><td>5</td></tr>
</table>

Pode-se observar no quadro 6.3 que a maior parte das entidades adicionadas foram referentes a *nouns*, enquanto que para as outras entidades observou-se apenas um pequeno acréscimo. Essa tendência é explicada pelo de fato de, para novas descrições de casos de teste de uma *feature*, ser comum aparecerem novos substantivos específicos para essa *feature*, ao passo que os verbos e modificadores utilizados são praticamente os mesmos.

6.2.3. Segundo Experimento

Para o segundo experimento, foram selecionadas 50 novas descrições de casos de teste, que foram somadas às 50 utilizadas no primeiro experimento. O procedimento foi o mesmo seguido no experimento anterior. As 100 descrições foram fornecidas como entrada para NLForSpec. As especificações de saída foram analisadas manualmente e em seguida analisadas na ferramenta FDR.

No segundo experimento, o protótipo de NLForSpec apresentou taxa de acerto de 87%, considerada bastante satisfatória. O aumento da taxa de acerto em relação ao primeiro experimento deveu-se ao acréscimo de entidades nas bases de conhecimento. Como os 50 casos de teste acrescidos para esse experimento foram retirados da mesma *feature* que os 50 do experimento anterior, a grande maioria das entidades já estavam armazenadas nas bases de conhecimento, pois a variação de termos utilizados dentro de descrições de casos de testes de uma mesma *feature* é relativamente pequena. Mesmo assim, ainda houve a ocorrência de especificações não corretamente geradas por falta de informação na base (por exemplo, um caso de teste referenciava um bit de nome "MMSMESSAGE_ENABLED" que ainda não tinha sido citado).

Em relação às sentenças não processadas corretamente por NLForSpec, as seguintes situações foram comumente encontradas:

1. Sentenças sem verbos, *e.g.*, a sentença "*At least 2 messages in Inbox*" nas pré-condições dos casos de teste, ao invés de "*There are at least 2 messages in Inbox*".
2. Sentenças ambíguas (que podem ser interpretadas de mais de uma maneira), *e.g.*, a sentença "*There is one read, unread, and protected message in Message Inbox*". Quais mensagens estão no *Message Inbox*, uma de cada tipo ou apenas uma com todas as características?

6.3. Considerações Finais

Neste capítulo, a implementação do protótipo e os experimentos realizados foram detalhados. Como mostrado, o desenvolvimento do protótipo foi baseado na metodologia *eXtreme Programming* e foi implementado em Java, procurando manter critérios de qualidade de software. As bases de conhecimento foram armazenadas em

arquivos no formato XML. Nessa fase inicial, o uso de XML facilitou a inserção manual de novas entidades às bases.

Foram apresentados também os dois experimentos realizados com o protótipo. Os experimentos geraram especificações em CSP a partir de descrições de casos de teste para o domínio de *Messaging* da Motorola. A taxa de acerto de 87% obtida pelo protótipo de NLForSpec no segundo experimento pode ser considerada satisfatória.

No próximo capítulo, as conclusões deste trabalho de mestrado serão apresentadas, bem como algumas possíveis extensões e melhorias.

7. Conclusões

Nesta dissertação, foi apresentado NLForSpec, uma ferramenta para geração de especificações formais em CSP a partir de casos de teste em Linguagem Natural.

Inicialmente, foi realizado um estudo das principais técnicas utilizadas pelos sistemas de interpretação de LN, e foram analisados alguns sistemas que geram modelos a partir de descrições em LN. Esse estudo possibilitou a criação de NLForSpec, que foi construída com base na arquitetura simbólica tradicional para interpretação de LN e contém quatro bases de conhecimento (Léxico, Gramática de Casos, Ontologia e Base de Especificações CSP) e três módulos de processamento (POS-Tagging, Processamento Semântico e Geração de Casos de Teste Formais).

Este trabalho foi desenvolvido como parte do projeto *Test Research Project* do CIn/BTC, em parceria entre o CIn-UFPE e a Motorola. O propósito geral desse projeto é automatizar a geração, seleção e avaliação de casos de teste para aplicações de telefonia móvel.

7.1. Principais Contribuições

Existem vários estudos na área de Interpretação de LN. Porém, na literatura disponível, pouquíssimos sistemas que geram especificações formais a partir de LN foram encontrados. E dentre esses sistemas, nenhum recebe como entrada descrições de casos de teste nem fornece como saída especificações na linguagem CSP. Dessa forma, o trabalho desenvolvido descrito nesta dissertação pode ser considerado original.

A principal contribuição do trabalho apresentado neste documento foi o desenvolvimento de uma ferramenta para geração de especificações formais em CSP a partir de casos de teste em Linguagem Natural. O trabalho aqui detalhado se baseou em técnicas de Interpretação de LN e utiliza bases de conhecimento para armazenar as informações lingüísticas e sobre o domínio.

Outra contribuição deste trabalho foi a definição de uma representação de caso de teste na linguagem formal CSP a partir das informações contidas nas BCs. Essa representação padronizada, simples e de fácil compreensão (legibilidade), contém a informação necessária para processo de atualização de requisitos que recebe como entrada as especificações geradas por NLForSpec.

Também podemos citar como contribuição as bases de conhecimento do protótipo de NLForSpec, que podem ser utilizadas em estudo posterior. As BCs possuem informações relevantes sobre os termos léxicos utilizados nas descrições de casos de teste para aplicações móveis da Motorola. Podemos ainda citar a ontologia, que traz a classificação das entidades do domínio de aplicações móveis.

Por fim, pode-se citar o estudo das principais técnicas utilizadas pelos sistemas de interpretação de LN e dos trabalhos relacionados, a partir do qual foram escritos os capítulos 2 e 3 deste documento.

7.2. Trabalhos Futuros

O trabalho desenvolvido obteve resultados satisfatórios para a tarefa de geração de especificações em CSP a partir de casos de teste em LN. Contudo, podem ser realizadas melhorias em NLForSpec. A seguir, são listados alguns possíveis trabalhos futuros.

Desenvolvimento de uma interface inteligente para atualização das BCs

O desenvolvimento de uma interface gráfica inteligente para adição, alteração e remoção de entidades nas BCs de NLForSpec tornaria mais fácil a manutenção dessas BCs por usuários não especialistas em PLN ou na linguagem CSP. Essa interface deve ser inteligente, de forma a verificar os relacionamentos entre as entidades das bases de conhecimento (*e.g.*, verbos na Gramática de Casos e suas respectivas entradas no Léxico). O uso dessa interface facilitará a integração de NLForSpec nas atividades do processo de teste de aplicações móveis e possibilitará que experimentos com novas *features* sejam realizados.

Adaptação de NLForSpec para tradução de Casos de Uso

NLForSpec pode ser adaptada para traduzir Casos de Uso (extraídos dos documentos de requisitos) em especificações CSP. Essa adaptação já está sendo

realizada no *Test Research Project* e irá representar o primeiro passo na geração automática de casos de teste a partir de requisitos (um dos objetivos do projeto). A representação de caso de uso em CSP é bastante similar à representação de caso de teste definida neste trabalho.

Definição de uma Linguagem Natural Controlada

As bases de conhecimento de NLForSpec podem ser utilizadas para definir uma Linguagem Natural Controlada (LNC). Para isso, seria criada uma nova base de conhecimento com regras que utilizariam as informações contidas na gramática de casos e no léxico para restringir a formação das sentenças. Essa LNC seria utilizada por um sistema de edição de documentos de casos de teste e de casos de usos para uniformizar as sentenças escritas, o que diminuiria o tempo de inspeção desses documentos.

Construção de um POS-Tagger específico para NLForSpec

Conforme foi mostrado, NLForSpec utiliza um POS-Tagger de propósito geral, o Stanford POS-Tagger. Porém, em algumas ocasiões, o Stanford POS-Tagger interpreta erradamente as sentenças, por não conter informações específicas do domínio. Apesar do módulo de processamento POS-Tagger de NLForSpec tentar se recuperar desses erros, nem sempre ele obtém sucesso. Dessa forma, o desenvolvimento de um POS-Tagger específico para NLForSpec diminuiria os erros de interpretação que ocorrem nessa etapa.

Referências

[Allen, 1995] ALLEN, J. F. *Natural Language Understanding*. 2nd edition Benjamin/Cummings, 1995. 654 p. ISBN: 0805303340.

[Barros & Robin, 1996] BARROS, F.; ROBIN, J. *Processamento de Linguagem Natural*. Tutorial apresentado na Jornada de Atualização em Informática do 16º Congresso da Sociedade Brasileira de Computação, Recife-PE, 1996.

[Baker *et al.*, 1998] BAKER, C. F.; FILLMORE, C. J.; LOWE, J. B. The Berkeley FrameNet project. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LINGUISTICS, 36., 1998, Montreal, Canada. *Proceedings …* ACL / Morgan Kaufmann Publishers, 1998.

[Bear, 1986] BEAR, J. *A morphological recognizer with syntactic and phonological rules*. In: COLING-86. Association for Computational Linguistics, Morriston, NJ, p. 272-276, 1986.

[Beck, 1999] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999. 224 p. ISBN: 0201616416.

[Becker, 1975] BECKER, J. D. 1975. The phrasal lexicon. In: WORKSHOP ON THEORETICAL ISSUES IN NATURAL LANGUAGE PROCESSING, June 1996, Cambridge, Massachusetts, USA. *Proceedings…*, Association for Computational Linguistics, Morristown, NJ, p. 60-63.

[Beesley & Karttunen, 2003] BEESLEY, K. R.; KARTTUNEN, L. *Finite State Morphology*. CSLI Publications, Palo Alto, CA, 2003. ISBN: 1575864347

[Bertolino *et al.*, 2004] BERTOLINO, A.; POLINI, A.; INVERARDI, P.; MUCCINI, H. Towards Anti-Model-Based Testing. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, June 2004, Florence, Italy. *Proceedings…*, p. 124–125, 2004.

[Bresnan, 1982] BRESNAN, J. *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, MA, 1982.

[Brill, 1995] BRILL, E. Transformation-based error-driven learning and natural language processing: A case study in part of speech tagging. *Computational Linguistics*, vol. 21, n. 4, p. 543-565, 1995.

[Brill, 2000] BRILL, E. Part-of-Speech Tagging. In: DALE, R.; MOISL, H.; SOMERS, H. *Handbook of Natural Language Processing*. Marcel Dekker, New York, ch. 17, p. 403-414, 2000.

[Bryant & Lee, 2002] BRYANT, B. R.; LEE, B. Two-Level Grammar as an Object-Oriented Requirements Specification Language. In: HAWAII

INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 35., January 2002, Big Island, Hawaii. *Proceedings..*, IEEE, 2002.

[Chandrasekaran *et al,* 1999] CHANDRASEKARAN, B.; JOSEPHSON, J.; BENJAMINS, V. What are ontologies and why do we need them? *IEEE Intelligent Systems*, vol. 14, n. 1, p. 20-26, 1999.

[Chomsky, 1956] CHOMSKY, N. Three Models for the description of language. *IRE Transactions on Information Theory*. Vol. 2, p. 113-124, 1956

[Craig & Jaskiel, 2002] CRAIG, R. D.; JASKIEL, S. P. *Systematic Software Testing*. Artech House Publishers, 2002. 536 p. ISBN: 1580535089.

[Cyre *et al.*, 1994] CYRE, W.R.; ARMSTRONG, J.R.; MANEK-HONCHARIK, M.; HONCHARIK, M. Generating VHDL models from natural language descriptions. In: CONFERENCE ON EUROPEAN DESIGN AUTOMATION, September 1994, Grenoble, France. *Proceedings…*, IEEE Computer Society Press, Los Alamitos, CA, p. 474-479, 1994.

[Dalal *et al.*, 1999] DALAL, S. R.; JAIN, A.; KARUNANITHI, N.; LEATON, J. M.; LOTT, C. M.; PATTON, G. C.; HOROWITZ, B. M. Model-Based Testing in Practice. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 21., May 1999, Los Angeles, California, United States. *Proceedings…*, IEEE Computer Society Press, Los Alamitos, CA, p. 285-294, 1999.

[Dürr & Katwijk, 1992] DÜRR, E.; van KATWIJK, J. VDM++: a formal specification language for object-oriented designs. In: THE SEVENTH INTERNATIONAL CONFERENCE ON TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, Dortmund, Germany. *Proceedings...*, HEEG, G.; MAGNUSSON, B.; MEYER, B. Prentice Hall International (UK) Ltd., Hertfordshire, UK, p. 63-77, 1992

[Dyer, 1995] DYER, M. Connectionist natural language processing: A status report. In: SUN, R.; BOOKMAN, L. *Computational Architectures Integrating Neural and Symbolic Processes: A Perspective on the State of the Art*. Norwell, MA. Kluwer, ch. 12, p. 389-429, 1995.

[Efe & Ng, 1987] EFE, K.; NG, P. A. A conceptual model for case grammar analysis. In: FALL JOINT COMPUTER CONFERENCE ON EXPLORING TECHNOLOGY: TODAY AND TOMORROW, 1987, Dallas, Texas, USA. *Proceedings …* IEEE Computer Society Press, Los Alamitos, California, p. 664-664.

[El-Far & Whittaker, 2001] EL-FAR, I. K.; WHITTAKER, J. A. Model-Based Software Testing. In: MARCINIAK, J. J. *Encyclopedia of Software Engineering*. 2nd ed. Wiley-Interscience, 2001. ISBN: 0471377376.

[Fanty, 1985] FANTY, M. *Context free parsing with connectionist networks*. Technical Report 174, Computer Science Department, University of Rochester, 1985.

[FDR, 2006] Formal Systems (Europe) Ltd. FDR2 Manual. Disponível em: http://www.fsel.com/documentation/fdr2/html/index.html

Último acesso em: 27 de junho de 2006.

[Francis & Kucera, 1964] FRANCIS, W.N.; KUCERA, H. *Manual of information to accompany a Standard Sample of Present-Day Edited American English, for use with digital computers.* Providence, RI: Department of Linguistics, Brown University. 1964.

[Fillmore, 1968] FILLMORE, C. J. The case for case. In: BACH, Emmon W.; HARMS, Robert T. *Universals In Linguistic Theory*. New York: Holt, Rinehart & Winston. 1968 p. 1-88. ASIN: B000BMZE7G.

[Fowler, 2000] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000. 464 p. ISBN: 0201485672.

[Gazdar *et al.*, 1985] GAZDAR, G.; KLEIN, E.; PULLUM, G.; SAG, I. *Generalised Phrase Structure Grammar*. Blackwell Publishing, Oxford, 1985.

[Gold Practices, 2006] Model Based Testing. Gold Practices website. 2006. http://www.goldpractices.com/practices/mbt/index.php Último acesso em: 06 de Maio de 2006.

[Groz & Sidner, 1986] GROZS, B. J.; SIDNER, C. L. Attention, intention, and the structure of discourse. *Computational Linguistics*, vol. 12, n. 3, p. 175-204, 1986.

[Hinton, 1981] HINTON, G. Implementing semantic networks in parallel hardware. In: HINTON, G. E.; ANDERSON, J. A. *Parallel models of associative memory*. Hillsdale, NJ. Erlbaum, p. 161-187, 1981.

[Hetzel, 1988] HETZEL, W. C. *The Complete Guide to Software Testing*. 2nd ed. John Wiley and Sons, Inc., Set. 1993. 296 p. ISBN: 0471565679.

[Hoare, 1985] HOARE, C. A. R. *Communicating Sequential Process*. Prentice Hall, 1985. ISBN: 0131532715.

[Hobbs, 1979] HOBBS, J. R. Coherence and coreference. *Cognitive Science*, vol. 3, p. 67-90, 1979.

[Hobbs, 1996] HOBBS, J. R. On the relation between the informational and intentional perspectives on Discourse. In: HOVY, E.; SCOTT, D. *Interdisciplinary Perspectives on Discourse*. Springer-Verlag, Heidelberg, 1996.

[Honcharik, 1993] HONCHARIK, A. J. *Generation of VHDL from Conceptual Graphs of Informal Specifications*. Master's Thesis, Bradley, Department of Electrical Engineering, Virginia Tech, Blacksburg, VA, July 1993.

[IFAD, 2000] IFAD. *The VDM++ Toolbox User Manual*. Technical Report, IFAD (http://www.ifad.dk/), 2000.

[Ishihara *et al.*, 1992] ISHIHARA, Y., SEKI, H., KASAMI, T. A Translation Method from Natural Language Specifications into Formal Specifications Using Contextual Dependencies. In: IEEE INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING, January 1993, San Diego, CA. *Proceedings...*, IEEE Computer Society Press, p. 232-239, 1993

[Johansson *et al.*, 1978] JOHANSSON, S.; LEECH, G.; GOODLUCK, H. *Manual of Information to Accompany the Lancaster-Oslo-Bergen Corpus*

of British English. Oslo: Department of English, Oslo University. 1978.

[Jorgensen, 1995] JORGENSEN, P. C. *Software Testing: a Craftsman's Approach*. CRC Press, 1995. ISBN: 084937345X

[Kasami *et al.*, 1986] KASAMI, T.; TANIGUCHI K.; SUGIYAMA Y.; SEKI, H. Principles of Algebraic Language. ASL/*. *Trans. of IECE Japan*, vo1. 69-D, n.7. p.1066-1074, 1986.

[Kay, 1984] KAY, M. Functional unification grammar: a formalism for machine translation. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LINGUISTICS, 10., 1984, Stanford, CA. *Proceedings …* pp. 75-78.

[Koskenniemi, 1983] KOSKENNIEMI, K. *Two-level Morphology: a general computational model for word-form recognition and production*. Ph.D. thesis, University of Helsinki, Helsinki, 1983.

[Liu, 2004] LIU, H. *MontyLingua: An end-to-end natural language processor with common sense*. (2004) Disponível em: web.media.mit.edu/~hugo/montylingua

[Lee & Bryant, 2002] LEE, B.; BRYANT, B. R. Automated conversion from requirements documentation to an object-oriented formal specification language. In: ACM SYMPOSIUM ON APPLIED COMPUTING, March 2002, Madrid, Spain. *Proceedings*..., ACM Press, New York, NY, p. 932-936, 2002.

[Long & Garigliano, 1994] LONG, D.; GARIGLIANO, R. *Reasoning by Analogy and Causality: Model and Applications*. Chichester, UK: Ellis Horwood, 1994.

[Mann & Thompson, 1987] MANN, W.; THOMPSON, S. Rhetorical structure theory: description and construction of text structures. In: Kempen, G. *Natural Language Generation: Recent Advances in Artificial Intelligence, Psychology, and Linguistics.* Kluwer Academic, 1987.

[Marcus *et al.*, 1993] MARCUS, M.; SANTORINI, B.; MARCINKIEWICZ, M.A. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics,* vol. 12, n. 2, p. 313-330, 1993.

[McGregor & Sykes, 2001] MCGREGOR, J. D.; SYKES, D. A. *A Practical Guide to Testing Object-Oriented Software*, 1st ed. Addison-Wesley, Março 2001. 393 p. ISBN: 0201325640.

[McKenna, 1994] MCKENNA, T. The Role of Interdisciplinary Research Involving Neuroscience in the Development of Intelligent System. In: HONAVAR, Vasant; UHR, Leonard. *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration.* New York, Academic Press, p. 75-92, 1994.

[Mich, 1996] MICH, L. NL-OOPS: From Natural Language to Object Oriented Requirements using the Natural Language Processing System LOLITA. *Journal of Natural Language Engineering*, Cambridge University Press, vol. 2, n.2, p. 161-187, 1996.

[Mohri, 1997] MOHRI, M. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, vol. 23, n. 2, p. 269-312,

1997.

[Moore & Pollack, 1992] MOORE, J. D.; POLLACK, M. E. A problem for RST: the need for multi-level discourse analysis. *Computational Linguistics,* vol. 18, n. 4, p. 537-544, 1992.

[OpenNLP, 2006] *The OpenNLP Homepage*. Disponível em: http://opennlp.sourceforge.net/ Último acesso: 01 de abril de 2006

[Pereira & Warren, 1980] PEREIRA, F. C. N.; WARREN, D. H. D. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Journal of Artificial Intelligence*, vol. 13, n. 3, p.231-278, 1980.

[Roscoe *et al.*, 1997] ROSCOE, A. W; HOARE, C. A. R.; BIRD, R. *Theory and Practice of Concurrency*. 1st ed. Prentice Hall, Nov. 1997. 565 p. ISBN: 0136744095.

[Reilly, 1984] REILLY, R. A connectionist model of some aspects of anaphor resolution. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LINGUISTICS, 10., July 1984, Stanford, California. *Proceedings…*, Association for Computational Linguistics, Morristown, NJ, p. 144-149.

[Scattergood, 1998] SCATTERGOOD, B. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis, University of Oxford, The Queen's College, 1998.

[Searle, 1971] SEARLE, J. R. What is a speech acts. In: SEARLE, J. R. *Oxford Readings in Philosophy*, p. 39-53. Oxford University Press, London, 1971.

[Selman, 1985] SELMAN, B. *Rule-based processing in a connectionist system for natural language understanding*. Technical Report CSRI-168, University of Toronto, Computer Science Department, 1985.

[Shieber, 1984] SHIEBER, S. M. The design of a computer language for linguistic information. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LINGUISTICS, 10., July 1984, Stanford, California. *Proceedings…*, Association for Computational Linguistics, Morristown, NJ, p. 362-366.

[Shieber, 1986] SHIEBER, S. *An Introduction to Unification-Based Approaches to Grammars*. University of Chicago Press, Chicago, 1986.

[Somers, 2000] SOMERS, H. Empirical Approaches to NLP. In: DALE, R.; MOISL, H.; SOMERS, H. *Handbook of Natural Language Processing*. Marcel Dekker, New York, ch. 15, p. 377-384, 2000.

[Sowa, 1984] SOWA, F. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading, MA, 1984.

[Sowa, 1991] SOWA, J. F., *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Sproat, 2000] SPROAT, R. Lexical Analysis. In: DALE, R.; MOISL, H.; SOMERS, H. *Handbook of Natural Language Processing*. Marcel Dekker, New York, ch. 3, p. 37-57, 2000.

[Toutanova *et al*., 2003] TOUTANOVA, K., KLEIN, D., MANNING, C., SINGER, Y. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In Proceedings of HLT-NAACL, p. 252-259, 2003.

[Uszkoreit, 1986] USZKOREIT, H. Categorial unification grammars. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LINGUISTICS, 11.., August 1986, Bonn, Germany. *Proceedings…*Association for Computational Linguistics, Morristown, NJ, p. 187-194, 1986.

[Watkins, 2001] WATKINS, J. *Testing IT:* an Off-the-Shelf Software Testing Process, Cambridge University Press, Maio 2001, 315 p. ISBN: 052179546X.

[Winograd, 1983] WINOGRAD, T. *Language as a Cognitive Process*. Vol. 1: Syntax, Addison-Wesley, Reading, MA, 1983.

[Woods, 1970] WOODS, W. A. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, vol. 13, n. 10, p. 591-606, 1970.

[W3C, 2006] World Wide Web Consortium (W3C). eXtensible Markup Language (XML). 2006. Disponível em: http://www.w3.org/XML/ . Último acesso: 02 de Junho de 2006.

[Zernik & Dyer, 1997] ZERNIK, U.; DYER, M. G. The self-extending phrasal lexicon. *Computational Linguistics: Special Issue of the Lexicon*, vol. 13, n. 3-4, p. 208-327, 1987. ISSN: 0891-2017.