UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DANTE GAMA TORRES

"SpecNL: Uma Ferramenta para Gerar Descrições em Linguagem Natural a partir de Especificações de Casos de Teste"

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

Resumo

Teste de Software é uma tarefa central, porém muito cara, no processo de desenvolvimento de software. Estudos sugerem que as tarefas de teste chegam até 50% do custo total de desenvolvimento do software. Com o objetivo de automatizar e otimizar as atividades de teste, várias ferramentas têm sido utilizadas para assistir o processo de testes, desde a geração dos testes até a sua execução. Os casos de teste gerados por essas ferramentas são scripts, que podem ser executados automaticamente ou manualmente. Os scripts executados automaticamente requerem uma plataforma de execução, não sendo necessário para o engenheiro de teste entender a linguagem que descreve o script de teste. Porém, alguns casos de teste não podem ser automatizados, necessitando da intervenção do engenheiro de teste, que deve conhecer a linguagem de script que descreve o caso de teste.

Este trabalho propõe uma ferramenta para gerar descrições em linguagem natural (LN) a partir de scripts de teste, com o intuito de ajudar os engenheiros de teste a executarem testes manuais. Para validar a ferramenta proposta, nós desenvolvemos um protótipo que recebe como entrada scripts de teste especificados na linguagem formal CSP. Como saída, o sistema devolve um texto em inglês que descreve os passos do caso de teste para aplicações móveis. O protótipo foi desenvolvido em Java, seguindo a metodologia de desenvolvimento de software *eXtreme Programming*. Este trabalho faz parte de um projeto maior desenvolvido em parceria entre o CIn-UFPE e a Motorola.

Palavras-chave: Geração de Linguagem Natural, Especificações de Casos de Teste, Inteligência Artificial Simbólica.

Abstract

Software Testing is an important task in the software development process. However, this task is very expensive. Studies suggest that Software Testing takes up to 50% of the total cost of software development. In order to automate and optimize the testing activities, several tools have been applied to aid the testing process, ranging from test generation to its execution. The test cases generated by these tools are scripts, which may be automatically or manually executed. The automatically executed test cases require an execution framework, and it is not necessary for the test engineer to understand the language that describes the test script. Nevertheless, some test cases can not be automated, demanding a mediation of the test engineer, which must be skilled in the script language that describes the test case.

This work proposes a tool to generate natural language descriptions from test case scripts, intending to help the test engineers in the manual test cases execution. In order to validate the proposed tool, we developed a prototype that receives as input test case scripts specified in CSP formal language. As output, the tool delivers a text in English, describing the test case steps. The prototype works on the mobile applications domain, and it was developed in Java, following the *eXtreme Programming* software development process. This work is part of a major project developed in a partnership between the CIn-UFPE and Motorola.

Keywords: Natural Language Generation, Test Case Specifications, Symbolic Artificial Intelligence.

ÍNDICE

LISTA DE FIGURAS

ÍNDICE DE TABELAS

1 Introdução

Sistemas computacionais tornaram-se indispensáveis tanto para a sociedade como para a indústria. Dessa forma, técnicas que auxiliem a produção de softwares confiáveis são cada vez mais necessárias. Nesse sentido, uma das atividades mais utilizadas para alcançar a qualidade e confiabilidade do software é o Teste de Software.

Teste de Software é uma atividade importante e tradicional no processo de desenvolvimento de software. Seu principal objetivo é verificar a corretude da implementação de um sistema. Para isso, entradas (ou estímulos) são fornecidas à implementação em teste, em um ambiente controlado, e baseado nas respostas às entradas, é verificado se aquela implementação comportou-se da maneira esperada.

A atividade de Teste de Software pode ser dividida em duas grandes etapas: geração de testes e execução de testes. Durante a etapa de geração de testes, um conjunto de casos de teste é criado com base em uma avaliação sobre quais requisitos devem ser cobertos pelos testes. Um caso de teste compreende um conjunto de entradas, condições de execução, e resultados esperados desenvolvidos para testar um requisito específico. Já durante a execução de teste, os casos de teste criados são executados no sistema em teste, e os erros encontrados durante a execução são reportados.

Apesar de ser uma ótima solução para garantir a confiabilidade do software, o Teste de Software é uma atividade extremamente cara, tomando em torno de 50% do custo total de desenvolvimento de software [Boehm, 1981]. Com o intuito de automatizar e otimizar essa atividade, muitas ferramentas têm sido desenvolvidas e aplicadas ao processo de teste, tanto na geração de testes, quanto na sua execução. Experimentos relatados por [Dustin *et al.*, 1999] mostraram que o uso da automação em todo o processo de teste reduziu em 75% o esforço empregado nesta atividade.

As ferramentas de geração automática de testes utilizam alguma linguagem (geralmente de script) para especificar os casos de testes gerados. Assim, alguns desses

casos de teste podem ser executados por ferramentas automáticas, dispensando a intervenção do engenheiro de teste.

Contudo, em geral, alguns dos testes gerados exigem a intervenção humana para permitir a sua execução, seja pela falta de ferramentas de execução automática, seja pela impossibilidade de executá-lo automaticamente (e.g., um caso de teste que verifica o bom funcionamento de uma aplicação móvel após a remoção e a reposição da bateria do dispositivo).

Como dito acima, os casos de teste gerados automaticamente estão descritos em linguagens de script. Claramente, isso pode constituir um obstáculo aos engenheiros, que devem ter profundo conhecimento da linguagem utilizada para representar o caso de teste, a fim de interpretá-lo corretamente. Nesse caso, seria de grande ajuda uma ferramenta que gerasse automaticamente descrições em linguagem natural (LN) a partir de scripts casos de testes. Isso eliminaria os custos relacionados ao treinamento dos engenheiros na linguagem de representação utilizada, reduzindo também o esforço relacionado ao entendimento do caso de teste.

1.1 O SpecNL

Esta dissertação apresenta o SpecNL, uma ferramenta voltada para a geração de descrições em LN a partir de especificações de teste. Seu principal objetivo é auxiliar os engenheiros a executarem os casos de teste manuais. A ferramenta recebe como entrada um conjunto de casos de teste especificados na linguagem formal CSP[1] [Hoare, 1985] [Roscoe *et al.*, 1997], e devolve, como saída, as descrições em Inglês dos casos de teste.

O SpecNL conta com cinco bases de conhecimento para armazenar informações lingüísticas e sobre o domínio da aplicação. As bases foram desenvolvidas de forma a facilitar a sua atualização por usuários sem conhecimento em geração de linguagem natural (GLN) ou no processo de geração de texto do SpecNL. O processo de geração do texto é realizado por quatro módulos de processamento, baseados nas técnicas tradicionais de GLN. Entretanto, o SpecNL é considerado um sistema híbrido, seguindo uma tendência atual na área.

[1] *Communicating Sequential Process*

Para validar a proposta do SpecNL, um protótipo foi implementado em Java, sob a metodologia *eXtreme Programming* de desenvolvimento de software. A implementação procurou manter critérios de qualidade, como manutenibilidade, portabilidade e reuso de código. As bases de conhecimento desse primeiro protótipo do SpecNL foram representadas em XML [W3C, 2006], com o intuito de facilitar a inserção manual de dados nessas bases. O protótipo foi desenvolvido para o domínio de aplicações para dispositivos móveis. Entretanto, o SpecNL permite a extensão para outros domínios, bastando apenas adaptar as bases, sem a necessidade de alteração dos módulos de processamento.

Foram realizados dois experimentos com o protótipo do SpecNL. Cada experimento consistiu em mapear, em linguagem natural, especificações de casos de teste para aplicações móveis. Os experimentos mostraram que as bases de conhecimento podem ser mantidas por pessoas sem conhecimento em GLN, tornando viável o uso do SpecNL em um ambiente real.

Poucos são os trabalhos voltados à geração de linguagem natural a partir de especificações de teste. Na literatura pesquisada, apenas o projeto ADL [ADL, 2001] se preocupou em mapear em texto casos de teste gerados automaticamente. Entretanto, o ADL utiliza técnicas muito simples de geração de LN, assemelhando-se a um sistema de mala direta. Isso atesta a originalidade do trabalho desenvolvido nesta pesquisa.

Este trabalho é parte do projeto *Test Research Project* do CIn/BTC, que está sendo desenvolvido em uma 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.

1.2 Organização da Dissertação

Além deste capítulo introdutório, esta dissertação é composta por mais 6 capítulos e um apêndice:

Capítulo 2 – neste capítulo, será apresentada uma visão geral sobre sistemas de geração de linguagem natural (GLN). Serão detalhadas as entradas e saídas desse tipo de sistema, bem como as principais tarefas realizadas no processo de geração. Por fim,

será apresentado um ensaio sobre os sistemas de geração híbridos, que misturam as técnicas mais tradicionais de geração, com técnicas mais simples e menos custosas.

Capítulo 3 – alguns trabalhos relacionados são apresentados neste capítulo. Os trabalhos apresentados geram descrições em linguagem natural a partir de especificações de software. Como dito, apenas um deles faz o mapeamento de especificações de casos de teste em LN.

Capítulo 4 – este capítulo detalha o projeto *Test Research Project*, no qual o trabalho apresentado nesta dissertação está inserido. As principais atividades do projeto são apresentadas, mostrando onde a ferramenta SpecNL se encaixa dentro do projeto. Este capítulo também traz noções sobre a linguagem CSP, bem como uma breve descrição sobre Teste de Software.

Capítulo 5 – o SpecNL é apresentado neste capítulo. As bases de conhecimento são descritas em detalhes, e as funcionalidades realizadas por cada módulo de processamento são apresentadas. O capítulo ainda mostra que é possível definir uma linguagem natural controlada a partir das bases de conhecimento do SpecNL.

Capítulo 6 – este capítulo apresenta detalhes de implementação do protótipo desenvolvido para validar a proposta do SpecNL. Dois experimentos realizados com casos de teste para aplicações do domínio de *Messaging* da Motorola são em seguida relatados.

Capítulo 7 – este capítulo conclui a dissertação, apresentando as contribuições oferecidas por este trabalho, e enumerando trabalhos que podem ser realizados futuramente para melhorar o desempenho do protótipo e a qualidade do texto gerado.

Apêndice A – neste apêndice são apresentados alguns exemplos de entradas das bases de conhecimento do SpecNL.

2 Sistemas de Geração de Linguagem Natural

A Geração de Linguagem Natural (GLN) é um sub-campo da Inteligência Artificial e do Processamento de Linguagem Natural (PLN) que objetiva o desenvolvimento de sistemas computacionais capazes de gerar textos em alguma linguagem natural. Basicamente, esses sistemas recebem como entrada uma representação não-linguística da informação que se deseja expressar e, fazendo uso de conhecimento sobre a linguagem e sobre o domínio da aplicação, produzem o texto de saída.

A GLN é uma tecnologia emergente, que envolve questões relacionadas à Inteligência Artificial, Ciências Cognitivas e Interação Homem Máquina: como o conhecimento do domínio e lingüístico devem ser representados e computados, o que é um texto bem escrito e como a informação é melhor comunicada entre o homem e a máquina [Reiter & Dale, 2000].

Pode-se dizer que a pesquisa em GLN começou nos anos 1950 e 1960 com projetos de tradução automática [Hutchins & Somers, 1992]. Porém, esses trabalhos tinham como foco o mapeamento de uma linguagem em outra. Os primeiros trabalhos da GLN moderna (geração de texto a partir de uma representação não lingüística) apareceram nos anos 1970, e as pesquisas mais significativas foram realizadas nos anos 1980. Nessa década, foram realizadas as primeiras tentativas de modularização dos sistemas de GLN. Foi nessa década também que surgiu o primeiro Congresso Internacional de Geração de Linguagem Natural[2], em 1983, solidificando muitos dos conceitos de GLN usados atualmente.

Na década de 1990, surgiram as primeiras aplicações reais de GLN, com destaque para o FoG [Goldberg *et al.*, 1994], o primeiro sistema de geração de texto a entrar em

[2] *International Workshop on Natural Language Generation*, http://ai.uwaterloo.ca/~inlg98/

uso na história. Nessa década, houve um crescente interesse na combinação de texto com gráficos [McKeown *et al.*, 1990] [Wahlster *et al.*, 1993] [André *et al.*, 1996], bem como em sistemas multilíngüe, com destaque para arquiteturas e técnicas que facilitassem a geração de texto em diversas línguas [Iordanskaja *et al.*, 1992] [Paris *et al.*, 1995] [Busemann & Horacek, 1998].

Atualmente, a grande maioria dos trabalhos em GLN está focada na produção de documentos em geral, automatizando tarefas repetitivas e enfadonhas aos humanos. Outros trabalhos têm como objetivo apresentar e explicar informações complexas para aqueles que não têm conhecimento ou tempo para entender o dado bruto. A GLN é de grande importância para tornar a comunicação entre os seres humanos e os computadores mais eficaz.

2.1 Quando Usar NLG?

Antes do desenvolvimento de qualquer sistema computacional, uma análise de requisitos é realizada para se conhecer o problema do cliente. Da mesma maneira, antes do desenvolvimento de um sistema de GLN, é muito importante conhecer as necessidades do usuário.

Um sistema de GLN nem sempre é a melhor maneira de atender a essas necessidades. Algumas vezes, é melhor apresentar ao usuário a informação em uma forma gráfica em vez de texto. Existem muitas situações em que a comunicação da informação é mais eficientemente realizada com o uso de elementos gráficos, como figuras, diagramas esquemáticos ou mapas. Antes de se construir um sistema de GLN, o engenheiro precisa considerar se gráficos, textos ou uma mistura dos dois é a melhor forma de solucionar o problema do usuário.

Segundo Reiter & Dale (2000), não há motivos contundentes para se decidir sobre o uso de gráfico ou texto. Geralmente, esta decisão é baseada no tipo de informação que se deseja comunicar. Informações sobre localizações físicas são melhores comunicadas através de gráficos anotados, enquanto conceitos abstratos, como informações sobre causa ou efeito, são mais eficientemente comunicados através de palavras. Uma boa maneira de decidir se texto ou gráfico deve ser usado é analisar os documentos existentes sobre o domínio e a forma como estes comunicam a informação.

GLN não é a única forma para se gerar texto. Muitos sistemas fazem uso da tecnologia de mala direta, que consiste simplesmente em inserir os dados de entrada em *slots* predefinidos de um *template* de documento. Novamente, é difícil afirmar quando é melhor usar mala direta ou GLN. Enquanto sistemas de mala direta são simples e de fácil desenvolvimento, as técnicas de GLN são mais adequadas para gerar textos mais variados e de melhor qualidade. Além disso, sistemas baseados em GLN são de mais fácil manutenção, visto que fazem uso de algoritmos e modelos de linguagem de propósito geral, enquanto os *templates* e algoritmos dos sistemas de mala direta são extremamente atrelados ao domínio.

Outro aspecto a considerar na escolha da tecnologia mais apropriada é o tempo de resposta do sistema. Para sistemas interativos, pode ser mais plausível o uso de uma abordagem mais simples, como mala direta, do que técnicas complexas de GLN, que possuem um elevado custo computacional.

Porém, construir um sistema de geração de texto nem sempre é a melhor solução. Desenvolver e manter um sistema pode custar caro e talvez seja mais viável economicamente contratar e treinar pessoas para escrever manualmente os documentos. Muitas vezes, a decisão entre a escrita manual e sistemas GLN se baseia na quantidade de texto a ser produzido. Um sistema de GLN que produz milhões de páginas de texto por ano possui um melhor *custo-benefício* do que um sistema que produz apenas algumas centenas de páginas por ano. Para este último caso, a contratação de alguém para manualmente escrever os textos pode ser mais viável.

No entanto, o fator econômico não pode ser sempre considerado decisivo para a automatização da tarefa de produção de texto. Outros fatores que podem ser levados em consideração são:

- Dados de entrada complexos – dados muito complexos podem ser mais facilmente processados por sistema computacionais do que por pessoas;
- Consistência – o ser humano pode inserir inconsistência no texto produzido, seja por displicência, tédio ou fadiga, o que não ocorre com um sistema de GLN;
- Conformidade com padrões – em alguns casos, os documentos devem ser produzidos seguindo padrões de escrita e conteúdo. Enquanto seres

humanos podem sentir dificuldade para seguir tais padrões, um sistema computacional pode ser programado para gerar texto seguindo um padrão especificado.

- Rapidez na geração dos documentos – muitas vezes, o tempo de geração de documentos é um fator importante na decisão do uso de sistemas GLN. Naturalmente, seres humanos levam mais tempo para produzir textos do que a máquina.

A seguir, serão detalhados a etapas que constituem o processo de geração de texto.

2.2 Entrada e Saída de um Sistema de GLN

Antes de apresentar a arquitetura dos sistemas de GLN e as etapas da geração de texto, veremos as entradas e saídas desses sistemas.

Podemos caracterizar a entrada de um sistema de GLN como uma tupla de quatro elementos <*k, c, u, d*>, em que *k* é a base de conhecimento a ser usada, *c* é o objetivo comunicativo a ser alcançado, *u* é o modelo do usuário, e *d* é o histórico do discurso [Reiter & Dale, 2000].

A base de conhecimento nada mais é que a informação sobre o domínio, que normalmente é armazenada em uma ou mais bases de conhecimento. Tanto o conteúdo quanto a representação destas bases são fortemente dependentes da aplicação. Ou seja, cada aplicação geralmente possui seu próprio formato de representação, tornando difícil o reuso dessas bases.

O objetivo comunicativo é o que se tenta comunicar no texto a ser gerado. Segundo Reiter & Dale (2000), não se pode confundir o propósito geral do sistema com o seu objetivo comunicativo. Por exemplo, o sistema FoG [Goldberg *et al.*, 1994], cujo propósito geral é a geração de textos sobre previsão de tempo, possui dois objetivos comunicativos distintos: (1) *marine forecast*, que foca mais nas velocidades dos ventos e suas direções, e o (2) *public forecast*, no qual as nuvens e precipitações são mais significantes.

O modelo do usuário especifica o perfil do leitor para o qual o texto é gerado. Em muitos sistemas de GLN, o modelo do usuário não é explicitamente especificado,

estando embutido no comportamento do sistema. Entretanto, alguns sistemas permitem a geração de texto de acordo com o perfil do usuário. O sistema IDAS [Reiter *et al*., 1995], por exemplo, faz uso de um modelo de usuário que leva em conta, entre outros, o vocabulário técnico do usuário e o seu conhecimento do domínio.

O histórico do discurso é um modelo que armazena as informações que já foram apresentadas no texto produzido. Esse histórico é utilizado para fazer referências a entidades expressadas em declarações anteriores. Sistemas de uma única interação começam com o histórico do discurso vazio e o preenchem à medida que vão produzindo o texto de saída. Em sistemas de diálogo, esse modelo é armazenado e utilizado em diferentes interações com o usuário.

A saída de um sistema de GLN consiste, basicamente, em texto. O tamanho do texto gerado varia de aplicação para aplicação, podendo ser apenas uma palavra (e.g., *okay*, *no*) ou um texto mais longo, de várias páginas. Muitos sistemas existentes não se preocupam com a formatação do texto, produzindo apenas uma seqüência de caracteres ASCII. Claramente, para aplicações reais, é mais adequado exibir o texto já formatado, estruturando-o em seções, e essas seções em parágrafos, de forma a facilitar a sua leitura. Muitas vezes é importante gerar o texto em um formato que possa processado por alguma ferramenta auxiliar (e.g., Microsoft Word, Internet Explorer, etc.), cuja função é formatar, exibir ou até mesmo sintetizar (i.e., produzir som) o texto gerado.

2.3 Etapas de um Sistema de Geração de Linguagem Natural

Como a grande maioria dos processos computacionais, é extremamente útil decompor o processo de geração de linguagem natural em etapas com funcionalidades específicas. O número de sub-etapas que um gerador de LN realiza é ainda motivo para debate. Contudo, é possível dizer que, de acordo com a comunidade de geração de linguagem natural, existem seis atividades básicas que podem ser identificadas durante o processo de geração, listadas a seguir:

1. Determinação do conteúdo – processo de decidir que informação será apresentada no texto de saída;

2. Organização do discurso – consiste em ordenar as informações selecionadas no passo anterior, a fim de estruturar o texto;
3. Agregação de Sentenças – processo que agrupa as informações já ordenadas em uma ou mais sentenças;
4. Lexicalização – tarefa que decide quais palavras ou termos devem ser usados para expressar os conceitos e relações do domínio contidos nas informações oriundas dos passos anteriores;
5. Geração de Expressões de Referência – etapa que insere no texto pronomes e dêiticos, a fim de facilitar a identificação das entidades nele citadas;
6. Realização sintática – etapa final, cuja função é gerar sentenças sintaticamente e morfologicamente corretas.

Existem diferentes maneiras de se construir um sistema de GLN. A forma mais natural (do ponto de vista computacional) seria implementar um módulo diferente para cada etapa acima listada, e conectar a saída de um módulo à entrada do outro. Porém, existem tarefas que são bastante interligadas, e que aparecem juntas em um único módulo na maioria das arquiteturas de sistemas de geração [Reiter & Dale, 1997]. A figura 2.1 mostra a arquitetura genérica de um sistema geração de linguagem natural.

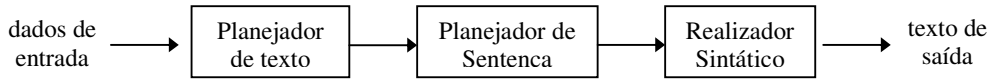


Figura 2.1: Arquitetura genérica dos sistemas de GLN.

Na arquitetura da figura 2.1, as tarefas de determinação de conteúdo e organização de discurso foram agrupadas no módulo de *Planejamento de Texto*. As tarefas de agregação de sentenças, a lexicalização e a geração de referências acabaram reunidas no módulo de *Planejamento de Sentença*. E por fim, a tarefa de realização sintática ficou separada em um único módulo.

A arquitetura mostrada na figura 2.1, apesar de ser largamente usada pelos sistemas de geração, não é a única existente na literatura de GLN. Para maiores informações sobre arquiteturas de sistemas de geração, ver [Paiva, 1998]. Vale ressaltar que o agrupamento de tarefas apresentado acima não é um consenso na área de GLN. Mathiessen (1991), por exemplo, argumenta que a etapa de lexicalização deve aparecer

junta com a realização sintática. A seguir, as etapas de um sistema GLN serão detalhadas.

2.3.1 Determinação do Conteúdo

Esta etapa consiste em decidir que informação deve ser comunicada no texto de saída. Normalmente, essa informação é dada como entrada ao sistema, ficando a cargo do sistema de geração a tarefa de selecionar o subconjunto mais adequado da informação disponível a ser comunicada.

A escolha do conteúdo a ser expresso no texto de saída é fortemente influenciada pelo objetivo comunicativo do sistema. Por exemplo, o sistema FoG possui dois objetivos comunicativos distintos: (1) *marine forecast*, que foca mais na velocidade dos ventos e suas direções, e o (2) *public forecast*, no qual as nuvens e precipitações são mais significantes. Ou seja, dependendo do objetivo comunicativo (*marine* ou *public forecast*), o sistema FoG irá gerar textos com conteúdos diferentes.

Outro fator que pode impactar na determinação do conteúdo é o nível de detalhe desejado para o texto de saída (texto resumido ou detalhado). Até questões mais simples, como o tamanho do texto de saída, podem influenciar a escolha da informação a ser comunicada.

A determinação do conteúdo depende também das características do domínio da aplicação, o que torna difícil generalizá-la. Um sistema de geração de textos de previsão de tempo tem um módulo de determinação de conteúdo bem diferente de um sistema de sumarização de dados estatísticos extraídos de um banco de dados. Porém, existem tarefas comuns aos módulos de determinação de conteúdo, como filtrar, resumir e, algumas vezes, processar os dados de entrada.

A técnica mais comum utilizada para a determinação de conteúdo é o uso de regras que procuram extrair informações mais abstratas a partir do dado bruto de entrada. Essas informações mais abstratas foram chamadas por [Reiter & Dale, 1997] de mensagens, que expressam, por exemplo, informações sobre as propriedades de uma entidade ou sobre as relações entre entidades. Por exemplo, a mensagem da figura 2.2 foi extraída por uma regra que verifica se o número de faltas em um jogo de futebol é

maior que 50. Caso seja, a mensagem é extraída com a informação de que o jogo foi violento.

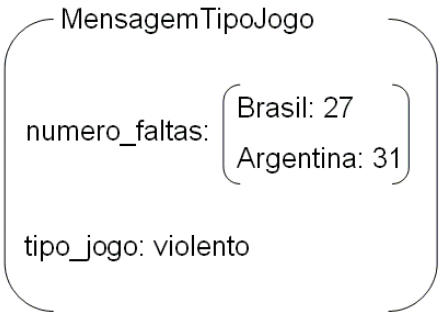


Figura 2.2: Exemplo de mensagem sobre o número de faltas em no jogo

A figura 2.3 exibe um texto que contém a informação da mensagem da figura 2.2.

> Brasil e Argentina jogaram, pelas eliminatórias da copa do mundo, na ensolarada tarde do último domingo. Foi um jogo violento, com um total de 58 faltas cometidas. O Brasil marcou 2 gols, e a Argentina apenas 1. Com a vitória, o Brasil garantiu a liderança do grupo A, com 6 pontos, enquanto a Argentina caiu para segundo, com 4 pontos.

Figura 2.3: Resenha sobre uma partida de futebol

2.3.2 Planejamento do Discurso

Esta etapa é responsável por ordenar e estruturar as informações selecionadas pela etapa de determinação de conteúdo. Um texto não é somente uma coleção aleatória de informações. A informação nele apresentada possui uma determinada ordem, que segue alguma estrutura base, como uma história que possui início, meio e fim, ou uma redação de que conta com introdução, desenvolvimento e conclusão. Porém, a maioria dos documentos possui estruturas mais complexas. Uma boa estrutura torna um texto bem mais legível.

A estrutura de um texto pode ser vista como uma árvore, existindo uma hierarquia entre as diversas partes do texto. Um exemplo claro dessa hierarquia é o índice desta dissertação, que, na sua forma de árvore, estrutura e ordena o conteúdo do documento. Os constituintes dessa hierarquia mantêm entre si relações de discurso (relações retóricas). O trabalho de relações de discurso mais influente na área de GLN é a *Rhetorical Structure Theory* (RST) [Mann & Thompson, 1988]. A RST define

aproximadamente 25 relações retóricas, entre as mais freqüentes estão *elaboration*, *exemplification*, *contrast* e *narrative sequence*. A figura 2.4 mostra a estrutura do texto exibido na figura 2.3. Observe as relações retóricas entre as partes da hierarquia.

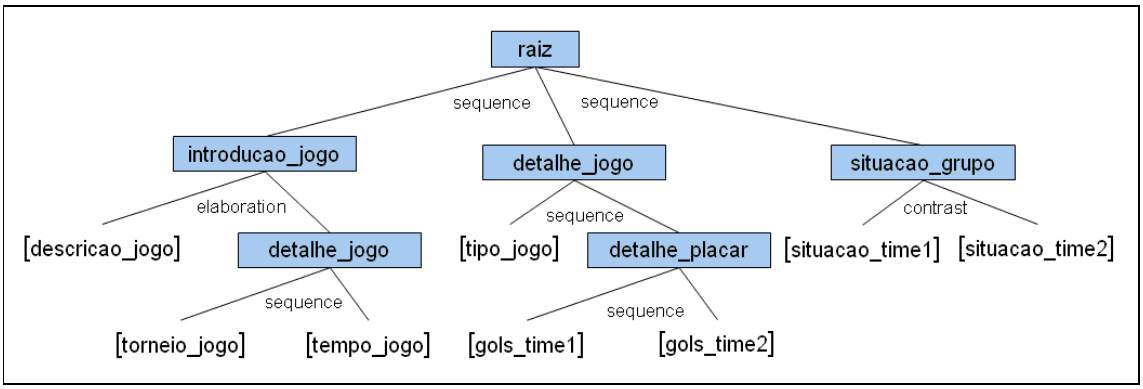


Figura 2.4: Estrutura de discurso do texto da figura 2.3

A técnica mais popular para implementar a organização de discurso é o uso de esquemas (*schemas*). Essa técnica se baseia no fato de que a maioria dos textos apresenta uma organização regular. Por exemplo, artigos científicos possuem uma introdução, um capítulo sobre estado da arte, outro sobre a pesquisa e, por fim, a conclusão. Na abordagem de *schemas*, cada esquema especifica como o texto deve ser construído usando esquemas menores, seguindo o padrão de árvore, em que as folhas são as mensagens selecionadas pela etapa de determinação de conteúdo. Um exemplo de *schema*, utilizado no sistema Joyce [Rambow & Korelsky, 1992], será apresentado no capítulo 3 (figura 3.4).

Mais detalhes sobre a organização do discurso podem ser encontrados em [Mann & Thompson, 1988], [Hovy, 1988], [McKeown, 1985] e [Kittredge, Korelsky & Rambow, 1991].

2.3.3 Lexicalização

A lexicalização consiste em escolher os itens lexicais – substantivos, adjetivos, advérbios, etc. – necessários para se expressar o conteúdo selecionado pela tarefa de Determinação de Conteúdo.

Em muitos casos, a lexicalização é realizada de maneira simples, associando um determinado item lexical (palavra ou expressão) a cada entidade ou relação do domínio.

Quando existe mais de uma alternativa, o sistema de geração busca escolher o termo mais adequado. Por exemplo, na frase "chutou a bola na trave", a palavra "chutou" pode ser substituída por outras palavras, como: "bicou", "meteu". A frase inteira pode ser refeita: “carimbou o travessão”.

A etapa de lexicalização é uma das mais complexas, devido à grande variedade de fatores que exercem influência sobre ela. A escolha entre as diversas alternativas deve considerar, entre outros fatores: (1) o estilo utilizado (formal ou informal, jornalístico ou técnico, etc.); (2) o uso regular das palavras (usa-se mais freqüentemente o verbo "chutar" quando junto com "bola"); (3) o nível de conhecimento do leitor; (4) o conteúdo e a forma das frases precedentes no texto gerado.

A funcionalidade da tarefa de lexicalização depende, obviamente, da aplicação. A lexicalização pode ser realizada de forma simples em sistemas que geram texto em uma única linguagem, sem se preocupar com variação de estilo, nível de conhecimento do leitor, melhor uso das palavras, etc. Entretanto, esta tarefa pode ser bastante complicada em sistemas multilíngüe, como o FOG, ou em sistemas como o IDAS, que leva em conta modelos de estilo de linguagem.

Da mesma forma que as tarefas de planejamento do discurso, esta tarefa é essencial para garantir uma boa legibilidade e fluência ao texto. Mais detalhes sobre a lexicalização podem ser encontrados em [Evens, 1989], [Elhadad *et al.*, 1997] e [Nogier & Zock, 1992].

2.3.4 Geração de Expressões de Referência

Um texto pode ser visto como a descrição de eventos e/ou relações envolvendo entidades de um determinado domínio. Dentro do texto, devem existir termos e expressões que possibilitem ao leitor discernir as diferentes entidades referenciadas. A tarefa de Geração de Expressões de Referência tem como função garantir que uma entidade referenciada no discurso seja distinguida das demais entidades do texto, impedindo que ele se torne ambíguo.

No exemplo da figura 2.5, temos um exemplo uso de expressões de referência. No item (a) temos um texto ambíguo (afinal quem marcou o gol: Ricardinho ou Zé

Roberto?). O texto do item (b) faz uso de uma expressão que identifica o autor do gol (o primeiro).

a. Ricardinho e Zé Roberto entraram no segundo tempo. *Ele* marcou o único gol do jogo.
b. Ricardinho e Zé Roberto entraram no segundo tempo. *O primeiro* marcou o único gol do jogo.

Figura 2.5: Exemplo de frases com expressão de referências

Dois casos de referências a entidades podem ser observados: (1) referência inicial e (2) referência subseqüente. A referencia inicial (1) objetiva introduzir uma nova entidade no discurso. A geração de referências iniciais tem sido pouco estudada na literatura, visto que é muito dependente tanto do domínio quanto da aplicação. Por exemplo, pode-se apresentar uma pessoa como um jogador de futebol, um empresário de banda de pagode ou como o filho de dona Maria. A melhor forma de apresentação vai depender do domínio em questão.

A referência subseqüente (2) referencia uma entidade já introduzida no discurso. Para isso, o sistema de geração faz uso do modelo de discurso, que armazena alguma representação das entidades já mencionadas no texto. O objetivo da referência subseqüente é evitar ambigüidade, fornecendo informação suficiente para que o leitor possa fazer a distinção entre as entidades (ver exemplo da figura 2.5). Por outro lado, deve-se tomar cuidado para não tornar o texto redundante e para não introduzir informação desnecessária.

Geração de referências é uma tarefa intimamente ligada com a lexicalização, visto que ambas produzem termos ou expressões que identificam os elementos do domínio. Porém, diferentemente da lexicalização, a geração de referências está preocupada em inserir informação no texto de forma a garantir a identificação não ambígua entre as entidades do discurso.

2.3.5 Agregação de Sentenças

Na etapa de agregação de sentenças, o conteúdo selecionado é agregado em sentenças. O módulo de agregação de sentenças deve decidir quais informações devem ser agregadas para formar uma sentença, e também deve escolher o elemento gramatical

(conjunções, pronomes relativos, etc.) a ser usado na combinação dessas informações. Para exemplificar, a figura 2.6 mostra duas formas de se agregar as sentenças "*Ricardinho entrou no segundo tempo*" e "*Ricardinho marcou o único gol do Brasil*".

a. Ricardinho entrou no segundo tempo. Ricardinho marcou o único gol do Brasil.
b. Ricardinho, que entrou no segundo tempo, marcou o único gol do Brasil.

Figura 2.6: Exemplo de frases com agregação

No exemplo (a) da figura 2.6, não foi usado nenhum mecanismo sintático para agregar as informações: cada informação ficou disposta separadamente em uma frase. Já no exemplo (b), foi usado o pronome relativo "que" para unir as duas informações em uma única sentença. Embora a agregação de sentenças não altere o conteúdo da informação a ser expressa, esta tarefa contribui para a legibilidade do texto de saída.

A grande dificuldade desta etapa é decidir como agregar as sentenças entre as diversas formas de agregação. Segundo Reiter & Dale (1997), as melhores regras de agregação levam em consideração conhecimento psicolingüístico sobre compreensão de leitura e sugestões de pessoas experientes em edição de texto. Porém, estas sugestões são muito vagas e difíceis de implementar computacionalmente.

Um maior aprofundamento sobre esta etapa de processamento de linguagem natural pode ser encontrado em [Dalianis & Hovy, 1999], [Appelt, 1985] e [Stone & Doran, 1997].

2.3.6 Realização Sintática

Uma sentença é formada por uma seqüência de palavras que seguem regras sintáticas e morfológicas da linguagem na qual está escrita. A realização sintática tem como objetivo gerar sentenças gramática e morfologicamente corretas. Para isso, o realizador sintático aplica regras de gramática a representações abstratas de sentenças. A complexidade desta etapa vai depender do nível de abstração dessas representações.

A seguir, estão alguns exemplos de tarefas que são realizadas nesta etapa do processo de geração:

- Mapear a representação abstrata da sentença em uma estrutura sintática de superfície;
- Determinar a devida conjugação do verbo de acordo com o tempo (passado, presente ou futuro), e em algumas línguas, como o Inglês, de acordo com tipo da sentença (pergunta ou afirmação) e a polaridade (sentença negativa, por exemplo). Todas estas informações são especificadas pelos estágios anteriores do processo de geração.
- Aplicar as regras de concordância entre o verbo e o sujeito.

O objetivo do realizador sintático é tornar transparentes as peculiaridades (algumas listadas acima) da língua do texto de saída. Essa transparência permite, nas etapas anteriores, o uso de modelos de linguagem mais simples e estruturados, que são independentes dos detalhes da língua de saída. Com isso, é possível desenvolver sistemas de geração multilíngües, que trabalham com representações abstratas das línguas nas etapas anteriores, e aplicam as regras de gramática da língua alvo nesta etapa final de realização sintática.

A realização sintática geralmente é implementada em um módulo à parte. É uma etapa bem conhecida pelos pesquisadores de GLN, tanto do ponto de vista computacional, quanto do ponto de vista lingüístico. Portanto, muitos projetos não incluem o realizador sintático no escopo da pesquisa, fazendo uso de realizadores de terceiros (e.g., KPML [Bateman, 1997], SURGE [Elhadad & Robin, 1996] e RealPro [Lavoie & Rambow, 1997]).

2.4 Sistemas Híbridos

Até agora, detalhamos a abordagem de GLN conhecida como *deep generation*, a mais tradicional na área. Esta abordagem consiste na geração automática de texto a partir de bases de conhecimento que descrevem o domínio, o usuário, a língua de saída e/ou o contexto. Entretanto, em determinadas situações, a criação dessas bases de conhecimento pode exigir muito esforço, inviabilizando o projeto. Ainda existem outras desvantagens relacionadas à *deep generation*, como a necessidade de alto poder

computacional para realizar a geração do texto e a falta de conhecimento técnico sobre certas tarefas do processo de geração [Reiter & Mellish, 1993].

Uma alternativa à *deep generation* é a *shallow generation*, que ao sacrificar alguns benefícios providos pela primeira abordagem de geração, procura reduzir os custos de criação das bases de conhecimento, diminuir a necessidade de poder computacional, e contornar aquelas tarefas do processo de geração ainda pouco compreendidas. Segundo Reiter (1995), existem milhares, se não milhões, de programas que geram texto automaticamente, porém poucos deles realizam as etapas detalhadas na seção 2.3. Os outros 99.9% dos sistemas simplesmente manipulam seqüências de caracteres, usando muito pouco, ou nenhum, conhecimento lingüístico.

A aplicação mais conhecida da *shallow generation* está no uso de *templates* e de *canned text*. Sistemas baseados em *canned text* mantêm uma coleção de strings pré-definidas em LN. Dada uma entrada, o sistema seleciona a string mais adequada como saída. Esta técnica não realiza nenhuma análise sintática ou semântica. Entretanto, algum tratamento ortográfico na saída pode ser requerido, por exemplo, pontuação, capitalização, etc. *Templates*, por sua vez, são mais flexíveis que *canned text*, já que as strings de saída pré-definidas podem conter *slots* a serem preenchidos com informação dinâmica. Esta técnica pode ser usada por sistemas de geração multi-sentença, cujo texto de saída possui uma estrutura regular.

Sistemas de geração de texto baseados puramente na *shallow generation* são bastante limitados, não oferecendo uma maior variação do texto de saída. Entretanto, é possível, obviamente, construir sistemas que incorporam tanto técnicas da *shallow* como da *deep generation*. Eles são chamados Sistemas Híbridos, e se caracterizam pelo uso das técnicas da *deep generation* que mais agregam valor ao sistema, combinadas com abordagens mais simples, que implementam tarefas em que a *deep generation* não é necessária, ou é muito cara. A decisão de que técnica usar deve ser baseada em uma análise de custo-benefício [Reiter & Mellish, 1993].

Segundo Reiter (1995), a forma mais comum de Sistemas Híbridos são aqueles que utilizam técnicas de *deep generation* na determinação de conteúdo e no planejamento de sentenças, e usam *templates* para a realização sintática. Segundo a autora, isso mostra que a principal contribuição da *deep generation* está no

processamento de alto nível da LN, e não no fato garantir a corretude da sintaxe do texto de saída.

Contudo, a tendência são os sistemas baseados em *deep generation* se sobressaírem em relação aos sistemas híbridos. As desvantagens relacionadas à *deep generation* serão amenizadas no futuro com melhores computadores, compartilhamento de bases de conhecimento e mais pesquisas sobre as tarefas de geração pouco conhecidas. Porém, no momento atual da pesquisa em GLN, sistemas híbridos continuam sendo uma alternativa à altura a sistemas puramente *deep generation*. Isso porque, enquanto os pesquisadores estão preocupados em melhorar a qualidade do texto gerado, as empresas estão preocupadas com os custos do projeto, preferindo assim a abordagem mais barata.

2.5 Considerações Finais

Este capítulo apresentou o estado da arte em GLN. Inicialmente, foi introduzida a área de geração de linguagem natural, mostrando um pouco de sua história. Um ensaio foi feito sobre a melhor forma de se usar sistemas GLN. Em seguida, foram apresentadas as entradas e saídas desse tipo de sistema.

Na seção 2.3, as principais tarefas realizadas por um sistema de GLN foram detalhadas. Por fim, na seção 2.4, apresentamos os Sistemas Híbridos, que utilizam técnicas complexas de GLN (abordagem *deep generation*), combinada com técnicas mais simples e de baixo poder computacional. Também mostramos que esse tipo de sistema é uma alternativa à altura aos sistemas puramente *deep generation*.

No próximo capítulo, detalharemos 4 sistemas de geração de linguagem natural a partir de especificações formais.

3 Sistemas de Geração de Texto a partir de Especificações

Neste capítulo, quatro sistemas de geração serão detalhados: Joyce (seção 3.1) e ModelExplainer (seção 3.2), ADL (seção 3.3) e Review (seção 3.4). Os dois primeiros sistemas geram descrições em linguagem natural a partir de especificações sobre a estrutura de um software. O sistema Review segue uma linha parecida, gerando texto a partir de especificações formais de sistemas. Todos os três utilizam algumas das técnicas apresentadas no capítulo 2. Já o sistema ADL gera texto a partir de especificações de sistemas e utiliza a mesma técnica para também gerar descrições dos casos de teste gerados automaticamente a partir da especificação.

Não encontramos na literatura disponível nenhum sistema voltado exclusivamente para a geração de texto a partir de especificações de casos de teste. O sistema ADL é o que mais se assemelha ao SpecNL, uma vez que se preocupa em gerar LN a partir de especificações de teste. Apesar de o Joyce, o ModelExplainer e o Review não trabalharem com casos de teste, eles possuem uma interseção com o nosso sistema: geração de descrições em linguagem natural a partir de especificações.

3.1 Joyce

O sistema Joyce [Rambow & Korelsky, 1992] [Rambow, 1990] foi desenvolvido como um sub-módulo do Ulysses [Korelsy & Ulysses-Staff, 1988], uma ferramenta de apoio ao projeto de software que possui um ambiente gráfico para auxiliar o desenvolvimento de sistemas distribuídos e seguros. Joyce gera dois tipos diferentes de texto sobre um projeto de software:

- Anotações sobre o software, que fazem parte da documentação durante a fase de projeto. Duas versões destes textos são geradas (uma curta e outra mais longa, cujo tamanho pode chegar a vários parágrafos);
- Explicações sobre o resultado de uma das ferramentas do Ulysses, o "*Flow Analyzer*", que detecta falhas de segurança no projeto de software.

A seguir, temos um exemplo de um componente de projeto de software desenvolvido com o auxílio da ferramenta Ulysses, e que serve de entrada para o gerador de texto Joyce (Figura 3.1). As figuras apresentadas nesta seção foram retiradas de [Rambow & Korelsky, 1992].

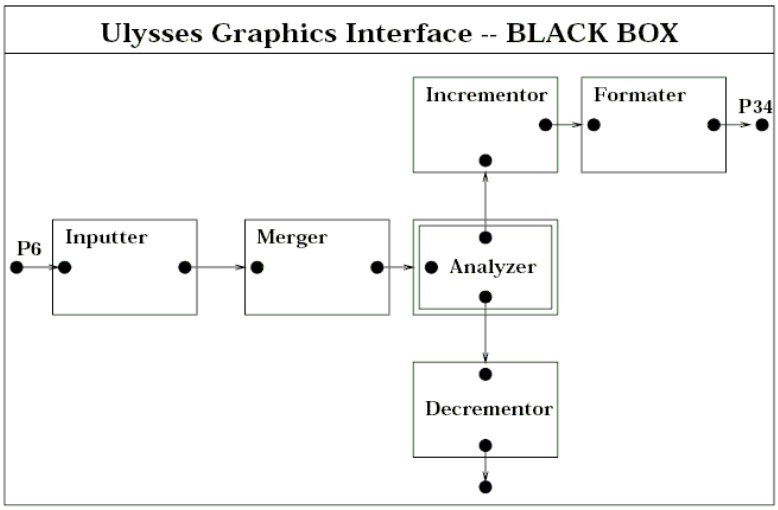


Figura 3.1: Representação gráfica dos componentes HOST e BLACK BOX

Já a figura 3.2 mostra o texto gerado sobre uma falha de segurança no componente BLACK BOX.

BLACK BOX: INSECURE FLOW

In the Black Box an insecure flow occurs. Classified information enters the Black Box through P6. It is passed through the Inputter to the Merger, which may upgrade it to top-secret. The Merger passes it to the Analyzer, which has been assumed secure. The Analyzer downgrades it to secret. It passes it through the Incrementor to the Formatter, which downgrades it when a classified corrected reading leaves through P34.

Figura 3.2: Texto gerado sobre falha de segurança no componente BLACK BOX

Joyce é baseado em uma arquitetura *pipeline* de três módulos, compreendendo o Planejador de Texto, o Planejador de Sentença e o Realizador Sintático. A figura 3.3 mostra a arquitetura do sistema.

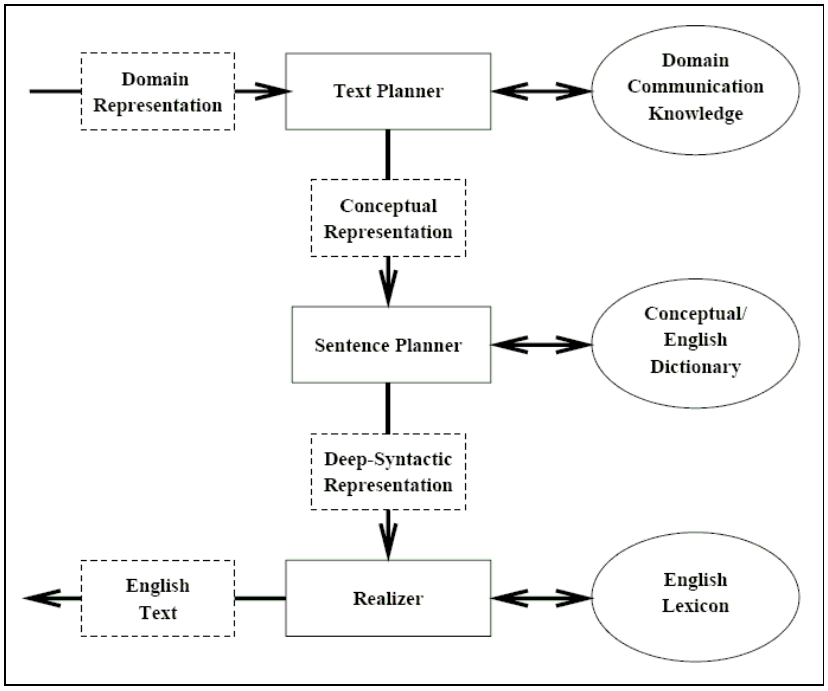


Figura 3.3: Arquitetura do sistema Joyce

O Planejador de Texto, que recebe como entrada a representação de um projeto de software criada pela ferramenta Ulysses. A partir daí, Joyce gera as proposições que representam tanto o conteúdo quanto a estrutura do texto.

Para auxiliar a execução dessa primeira etapa, foi criada uma linguagem de esquemas chamada DICKENS (*DomaIn Communication Knowledge Encoding Schemas*), usada para representar o *Domain Communication Knowledge* (DCK), conhecimento do domínio a ser comunicado. O DCK compreende o conhecimento sobre a informação que deve ser apresentada no texto final e em que ordem. Segundo Rambow (1990), esse tipo de conhecimento está presente em todos os sistemas de geração, porém ele é codificado implicitamente no módulo de planejamento de texto. Com a linguagem DICKENS, este conhecimento pode ser codificado explicitamente[3],

[3] Existem oito esquemas DICKENS para auxiliar a geração de texto sobre falhas de segurança no projeto de software. A quantidade de esquemas para a geração de textos sobre os componentes do projeto e suas relações não foi informada.

na tentativa de se aumentar a eficiência, a modularidade e a portabilidade do sistema de geração.

As proposições geradas pelo Planejador de Texto são imediatamente enviadas para o módulo de Planejamento de Sentença. Este segundo módulo realiza duas tarefas: (1) monta a *Deep Syntatic Representation* (DsyntR)[4] a partir da proposição, e (2) tenta agrupar essas representações de forma a gerar sentenças maiores. A geração das representações DSyntR faz uso do *Conceptual/English Dictionary* (ver figura 3.3), que foi implementado como um conjunto de procedimentos que operam sobre as proposições.

A representação DSyntR construída é então enviada para o último módulo do sistema, o Realizador Sintático. Este módulo, baseado na *Meaning-Text Theory* (MTT) [Melcuk, 1988], trabalha com três níveis sucessivos de representação, que são tratadas por três sub-módulos diferentes. O primeiro sub-módulo transforma as representações DSyntR, recebidas do planejador de texto, em *Surface Syntatic Representation* (SSynrR), que se diferencia da DSyntR por agregar lexemas que serão usados na sentença final. Esses lexemas são obtidos do componente *English Lexicon* (ver figura 3.3). O segundo sub-módulo recebe, então, a SSyntR e constrói a *Deep Morphological Representation* (DMorphR), que é a linearização dos nós da representação SSyntR. Por fim, o último sub-módulo do Realizador Sintático transforma a representação DMorphR em *Surface Morphological Representation*, a sentença final em Inglês.

Para exemplificar o processo de geração, suponha que o planejador de texto esteja executando o comando marcado com uma seta na figura 3.4, e que o foco do planejador seja o componente BLACK BOX (componente sendo descrito). A execução do `make-proposition` resultará na criação da proposição mostrada na figura 3.5.

[4] A DsyntR é estruturada em forma de árvore, tendo um verbo como raiz. Cada DSyntR representa uma oração.

```
(defschema flow-analysis-error
  :title "Insecure flow"
  :theme "information"
  :make-proposition (insecure-flow :location focus)
  :make-proposition (enter :agent (get-information)
                           :object focus
                           :location (entry-port focus))
  :make-proposition (id-security :agent (get-information)
                                 :value (get-level (entry-port focus)))
  conceptual-break
  :shift-focus-and-edit (next-component initial-follow-path #'merge-send-data))
```

Figura 3.4: Esquema DICKENS para a geração de texto sobre falhas de segurança

```
ENTER
  AGENT      #<information>
  OBJECT     #<COMPONENT Black Box>
  LOCATION   #<PORT P9>
```

Figura 3.5: Proposição para o verbo *Enter*

A proposição da figura 3.5 será mapeada para a DSyntR da figura 3.6 e, após passar pelo Realizador Sintático, resultará na sentença "*information enters the Black Box through P6*".

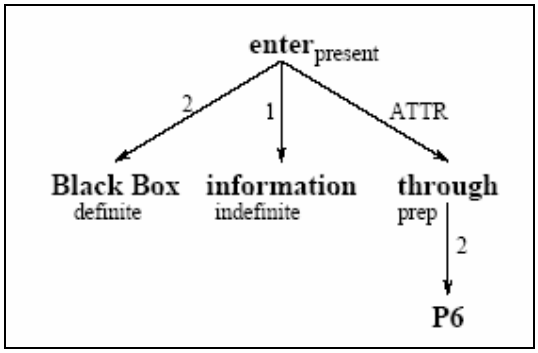


Figura 3.6: DsyntR da sentença "information enters the Black Box through P6"

Segundo Rambow & Korelsky (1992), a ferramenta Joyce se mostrou bastante útil na interface do usuário do Ulysses. Entretanto, os autores não deram maiores informações sobre a manutenção das bases de conhecimento do Joyce. Nada foi informado sobre a complexidade de se adicionar termos léxicos no *English Lexicon*, o que pode impossibilitar o uso da ferramenta Joyce por usuários comuns, sem expertise em GLN. Sobre o *Conceptual/English Dictionary*, os autores afirmaram apenas que o usuário é livre pra escrever entradas no dicionário. Isso pode acarretar inconsistências

no texto gerado. Também não foram informados detalhes que tornassem possível a avaliação da complexidade da manutenção do *Conceptual/English Dictionary*.

3.2 ModelExplainer (ModEx)

O sistema ModEx [Lavoie *et al*., 1996] [Lavoie *et al*., 1997] é um gerador de descrições em linguagem natural de modelos de software orientado a objeto (ver figura 3.7). Os textos são gerados em formato de *hypertexto*, o que permite ao usuário navegar pelo modelo. O MoDex inclui no texto final alguns exemplos, com a finalidade de auxiliar o entendimento do modelo.

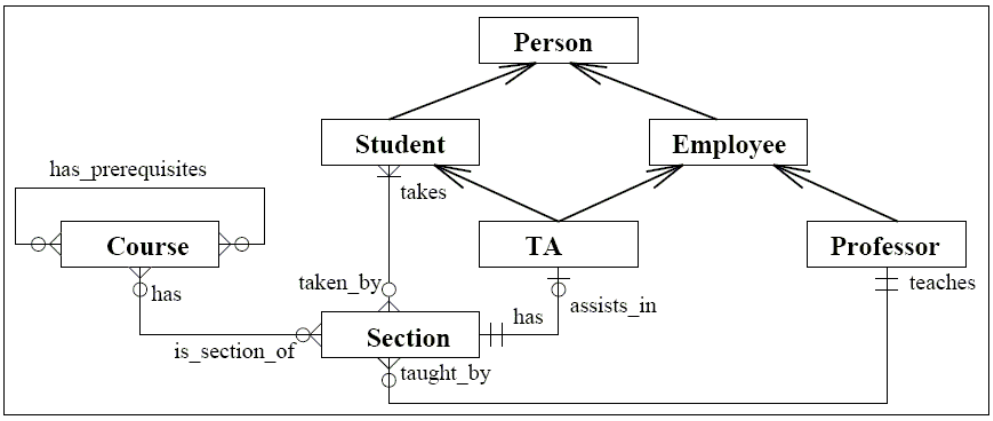


Figura 3.7: Modelagem OO de uma Universidade

Na figura 3.8, temos o texto gerado para a classe *Section* do modelo apresentado na figura 3.7. As palavras sublinhadas são *links* para outras descrições.

Description of the class `Section'

General Observations:
A Section must be taught by exactly one Professor and may belong to zero or more Courses. It must be taken by one or more Students and may have at most one TA.

Examples:
For example, Sect1 is as Section and is taught by the Professor John Brown. It belongs to two Courses, Math165 and Math201, and is taken by two Students, Frank Belford and Sue Jones. It has the TA Sally Blake.

Figura 3.8: Descrição da classe *Section*

A arquitetura do ModEx é um *pipeline* de quatro módulos: Planejador de Texto, Planejador de Sentença, Realizador Sintático e Formatador. A arquitetura do sistema pode ser vista na figura 3.9.

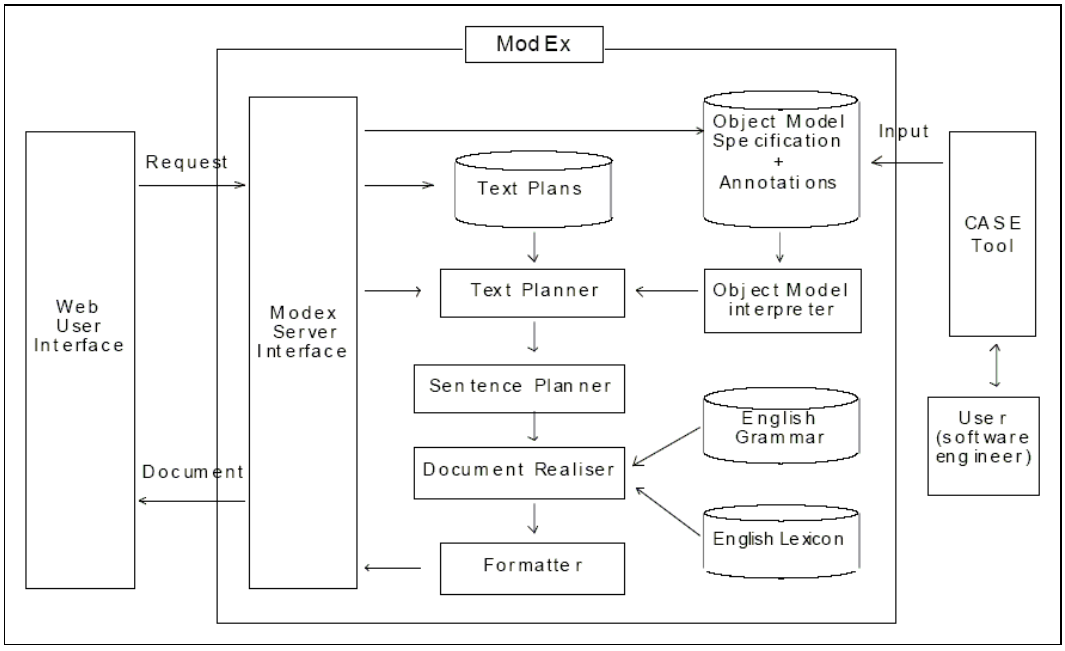


Figura 3.9: Arquitetura do Modex

O Planejador de Texto recebe como entrada a modelagem OO de um software e um plano de texto, que determina a organização do texto de saída. O MoDex possui uma interface que permite que esses planos sejam editados pelo usuário, de forma a personalizar o formato de saída de acordo com as suas preferências. A figura 3.10 exibe o plano utilizado para gerar o texto da figura 3.8. A saída deste primeiro módulo não foi exemplificada nos artigos sobre o sistema.

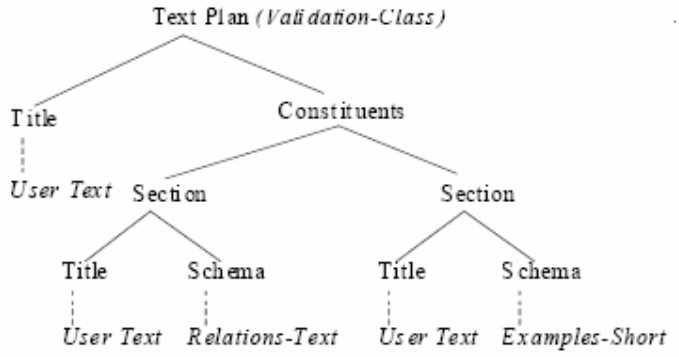


Figura 3.10: Plano de texto do MoDex

O segundo módulo do sistema é o Planejador de Sentenças. Também não foram informados maiores detalhes sobre esta etapa do processo de geração. Porém, é plausível que este módulo, da mesma forma que o sistema Joyce, transforme a saída do planejador de texto em representações DSyntR[5] e, a partir daí, realize agregação dessas representações para formar sentenças maiores.

A realização sintática do ModEx recebe como entrada as representações DSyntR. Esta etapa é realizada com o auxílio da ferramenta RealPro [Lavoie & Rambow, 1997], baseada na *Meaning-Text Theory* [Melcuk, 1988]. Finalmente, o resultado da realização sintática é passado para o módulo Formatador, que produz o documento HTML final.

O ModEx não possui conhecimento sobre o domínio que a modelagem OO descreve, ele tem apenas o conhecimento sobre a modelagem OO (tipos de entidades utilizadas na modelagem e seus relacionamentos). Com isso, para que o MoDex tenha sucesso na geração de textos, os *labels* usados no modelo OO para descrever as entidades e suas relações devem seguir certas convenções[6] (observe os nomes *has_prerequisite*, *takes*, *is_section_of*, etc. das relações do modelo da figura 3.7). Por um lado, isto é uma vantagem, pois o sistema fica independente de domínio. Contudo, isso pode acarretar a geração de textos incoerentes. Obviamente, este não é o caso do exemplo da figura 3.8.

[5] Deduzimos que o sistema MoDex utilize as representações DSyntR porque estas representações devem servir de entrada para o módulo de realização sintática. A realização sintática do ModEx faz uso da ferramenta RealPro, que, por sua vez, foi desenvolvido com base na Meaning-Text Theory e que recebe como entrada justamente representações DSyntR.

[6] "O ModEx espera que as classes sejam rotuladas com nomes no singular, e que as relações sejam rotuladas com verbos na terceira pessoa do singular da voz ativa, verbos na voz passiva seguidos do termo *by* ou nomes." (Lavoie et al. (1996), página 11).

3.3 ADL Project

O *Assertion Definition Language* (ADL) *Project* [ADL, 2001] [deRaeve *et al*., 1995] surgiu do desejo de se possuir ferramentas que acelerassem o desenvolvimento de interfaces. ADL é uma linguagem de alto nível que provê uma gramática formal para especificação de comportamento de interfaces, que permite a geração automática de testes baseada em especificações de API.

O projeto ADL possui um módulo de geração de documentação, que possui duas saídas possíveis. A primeira são manuais de referência que descrevem a interface da funcionalidade a ser testada. A segunda saída possível é a documentação dos testes gerados, que descreve tanto o comportamento das funcionalidades a serem testadas, como a seqüência de ações que compõem os testes. O conteúdo desses documentos pode vir de *templates*, pode ser gerado automaticamente ou pode ser informado diretamente pelo usuário. O módulo de geração de documentos foi desenvolvido de forma gerar textos em mais de uma língua.

Antes de abordar o módulo de geração de documentos, detalharemos as entidades que compõem a especificação ADL. Especificações escritas em ADL são asserções em lógica de primeira ordem sobre os objetos da especificação. A semântica da ADL é bem definida e livre de contexto, o que torna a tradução das asserções ADL para linguagem natural muito mais simples.

As expressões escritas em ADL são formadas por operadores, objetos de dados e funções definidas pelo usuário. A semântica dos operadores é fixa: os usuários não podem renomeá-los ou redefinir sua semântica. ADL permite que objetos de dados sejam definidos e referenciados por identificadores. ADL também permite a criação de novas funções, com semântica definida pelo usuário.

A conversão das especificações ADL em linguagem natural consiste em traduzir cada elemento descrito acima em um sintagma (nominal ou verbal). Essas orações são então combinadas sem levar em consideração o contexto. Cada elemento da especificação possui estratégias diferentes de tradução, descritas a seguir:

- Operadores – a semântica não ambígua dos operadores ADL os torna prontamente expressáveis em linguagem natural. Com isso, cada operador pode ser representado por um sintagma nominal;

- Objetos de dados – nesse caso, sintagmas nominais são usados para expressar os objetos da especificação;
- Funções – são descritas como uma combinação de sintagmas verbais, que descrevem o significado da função, e de sintagmas nominais, que descrevem como os parâmetros da função participam nas ações da função.

A figura 3.11 mostra um exemplo simples, mas sintaticamente completo, de um módulo especificado em ADL. A especificação descreve o comportamento da função *isosceles* que verifica se dois inteiros são iguais, e descreve a função *equilateral*, que faz uso da função *isosceles* e verifica, dados os tamanhos dos lados, se o triângulo é eqüilátero. As imagens e exemplos relacionados ao ADL foram retirados de [deRaeve *et al*., 1995].

```
module triangle {
        boolean isosceles(int a, int b)
        semantics {
                (a == b) <-> return == TRUE
        };
        boolean equilateral(int a, int b, int c)
        semantics {
                isosceles(a,b)&& isosceles(b,c) <->
                return == TRUE
        };
}
```

Figura 3.11: Exemplo de um módulo de sistema

A figura 3.12 exibe o resultado da tradução para linguagem natural do módulo especificado na figura 3.11. O texto da figura 3.12 é um documento NLS (*natural language specification*), que é um dos tipos de documento gerados pelo sistema.

SYNOPSIS:

boolean

isosceles(int a, int b)

GENERAL BEHAVIOR:

a is equal to b if and only if return is equal to TRUE.

SYNOPSIS:

equilateral(int a, int b, int c)

GENERAL BEHAVIOR:

The value returned by isosceles(a,b) and the value returned by isosceles(b,c) if and only if return is equal to TRUE.

Figura 3.12: Tradução em LN do modulo da figura 3.11

O mapeamento das especificações ADL em LN é realizado por um conjunto de regras escritas em Prolog. A geração do texto para outros idiomas é possível graças a diferentes conjuntos de regras. Isto é, a estrutura sintática de cada idioma alvo é garantida por regras em Prolog específicas.

Entretanto, como pode ser visto na figura 3.11, para se produzir um texto de qualidade em linguagem natural é claramente necessário expressar o significado por trás dos identificadores. Para isso, foi criado o *Natural Language Dictionary* (NLD), que agrega informações que não estão disponíveis nas especificações. O NLD relaciona cada identificador a uma descrição em LN, sem nenhuma restrição semântica ou sintática.

O NLD é fornecido pelo usuário. Como não há nenhum mecanismo de controle sobre a edição do dicionário, mapeamentos sem sentido podem ser inseridos no NLD. A figura 3.13 mostra um exemplo de NLD construído para o módulo da figura 3.11 e que foi utilizado na geração das descrições da figura 3.14.

```
equilateral(p1, p2, p3) {
        p1 =“the first leg of the triangle”,
        p2 =“the second leg of the triangle”,
        p3 =“the third leg of the triangle”,
        $DESCRIPTION = “Verify that the triangle specified by its legs is an equilateral triangle.”
}
Isosceles(p1, p2) = “test whether” p1 “is equal to” p2 {
        p1 = “leg one”,
        p2 = “leg two”,
        $DESCRIPTION = “Verify that the two legs given are equal.”
}
```

Figura 3.13: Exemplo do *Natural Language Dictionary*

SYNOPSIS:

boolean

isosceles(int a, int b)

DESCRIPTION:

Verify that the two legs given are equal.

GENERAL BEHAVIOR:

Leg one is equal to leg two if and only if return is equal to TRUE.

SYNOPSIS:

equilateral(int a, int b, int c)

DESCRIIPTION: Verify that the triangle specified by its legs is an equilateral triangle.

GENERAL BEHAVIOR:

Test whether the fist leg of the triangle is equal to the second leg of the triangle and test whether the second leg of the triangle is equal to the third leg of the triangle if and only if return is equal to TRUE.

Figura 3.14: Tradução em LN do modulo da figura X com NLD

Apesar de bastante simples, os autores garantiram que a geração de linguagem natural apresentada aqui foi satisfatória. O processo de geração não faz nenhum uso das técnicas lingüísticas apresentadas no capítulo 2. Como o próprio autor diz, é apenas um processo substituição de expressões em LN.

3.4 Review System

O sistema Review [Salek *et al.*, 1994] tem como propósito geral mapear especificações formais em linguagem natural. O Review está atrelado ao sistema Metaview [Sorenson *et al.*, 1988], um meta-sistema cujo objetivo é a geração automática de uma grande parte do software necessário para ambientes CASE [Fisher, 1991]. Com o Metaview, é possível definir o ambiente CASE através da linguagem *Enviroment Definition Language* / *Enviroment Constraint Language* (EDL/ECL) e, a partir daí, gerar o software necessário para o ambiente.

Antes de falar sobre o sistema Review, mostraremos um pouco da sua interação com o Metaview. O sistema Review não faz parte do software gerado pelo Metaview. O Review foi desenvolvido para gerar texto a partir de especificações de sistemas projetados em qualquer ambiente CASE modelado pelo Metaview. Para isso, ele recebe como entrada o modelo do ambiente CASE com o objetivo de gerar texto especificamente para o ambiente em questão. A partir daí, o gerador Review é desacoplado do Metaview, agindo como uma ferramenta independentemente e específica para o ambiente escolhido pelo analista.

O sistema Review recebe duas grandes entradas. A primeira são tabelas que foram pré-compiladas a partir do modelo EDL/ECL do ambiente CASE. A outra entrada são as especificações que serão mapeadas em linguagem natural, e que estão armazenadas em um banco de dados. Além dessas entradas, o sistema recebe uma requisição do usuário, que determina o tipo de relatório a ser gerado. O Review pode gerar relatórios completos ou simplificados.

As especificações que servem de entrada para o sistema Review são escritas na linguagem *Simple Specification Language* (SSL). Cada ambiente CASE possui uma linguagem SSL distinta, definida pelo modelo EDL/ECL do ambiente. Porém, a diferença entre essas linguagens está apenas nos nomes dos objetos da linguagem. A sintaxe e a semântica são as mesmas. A figura 3.15 mostra um exemplo (incompleto) de uma especificação de entrada escrita em uma SSL. Os exemplos e figuras sobre o Review exibidos aqui foram retirados de [Salek *et al.*, 1994].

```
BEGIN
process query_access
        (description := "The query_access process provides the means for library patrons to
pose    queries on the holdings (books, etc.) of the library");
data_store library_holdings
        (form := "disk file"; cardinality := 4; access_method := "random");
stored_data holdings_info
        (description := "Holdings_info includes all of the information required to identify each
        item (periodical, book, thesis, microfiche, etc.) in the library");
stores (library_holdings, holdings_info);
changes (database_access, holdings_info);
reads (database_access, holdings_info);
sends (query_access, parsed_query, database_access) (frequency := (100, day));
sends (database_access, reply_info, query_access) (frequency := (100, day));
…
END librarian_system;
```

Figura 3.15: Exemplo de especificação SSL

A figura 3.16 exibe a arquitetura do sistema Review, com cinco módulos principais.

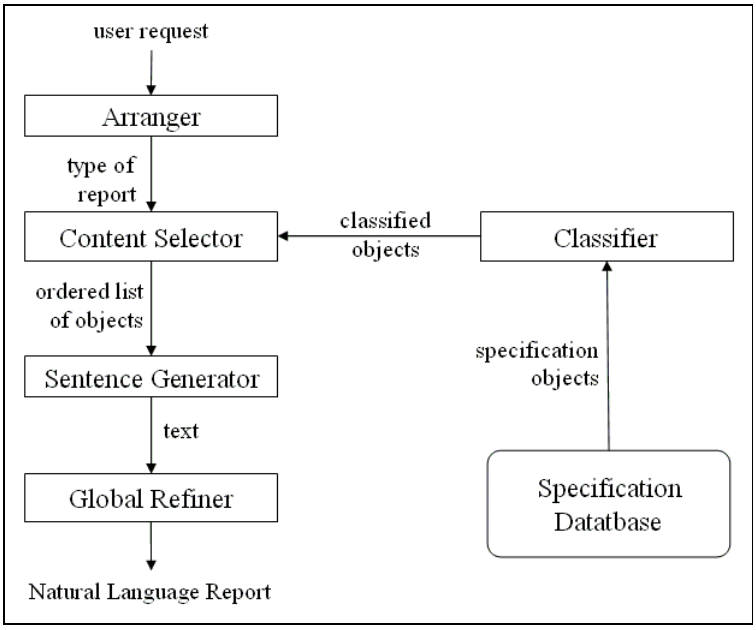


Figura 3.16: Arquitetura do sistema Review

O módulo *Input Processor* é responsável pelo processamento da requisição do usuário (tipo de relatório desejado). O *Classifier* recebe como entrada a especificação armazenada no banco de dados, e identifica cada objeto nela presente. Os objetos classificados e a requisição do usuário são passados para o *Content Selector*, cuja tarefa é selecionar os objetos da especificação mais relevantes, de acordo com a requisição do usuário. Além disso, o *Content Selector* ordena os objetos da especificação através da construção de um grafo. Esse grafo é percorrido por um algoritmo que determina o melhor ordenamento dos objetos.

Os objetos já ordenados são passados para o *Sentence Generator* (detalhado mais tarde) que tem como função gerar sentenças que descrevem os objetos do sistema. Essas sentenças são então enviadas para o *Global Refiner*, cuja função principal é organizá-las de forma que pareçam um texto coerente. Para isso, o *Global Refiner* faz uso de conectivos entre as sentenças e, em alguns casos, as combina para aumentar a legibilidade. A figura 3.17 exibe o texto gerado pelo Review para a especificação da figura 3.15.

Library holdings are data stores whose form is a disk file, whose acces method is random, and whose cardinality is 4. Library holdings store holdings info. Holdings info includes all of the information required to identify each item (periodical, book, thesis, microfiche, etc.) in the library.

Database access is a process which changes and reads holdings info.

The query access process provides the means for library patrons to pose queries on the holdings (books, etc.) of the library. This process sends a parsed query to the database access process. The frequency is measured as times per unit. The frequency has a quantity of 100 and a unit of 1 day. Database access sends a reply info to query access. The frequency has a quantity of 100 and a unit of 1 day.

Figura 3.17: Exemplo de texto gerado pelo Review

Agora detalharemos o *Sentence Generator*, principal módulo do sistema Review, e responsável por gerar texto a partir dos objetos da especificação SSL. Na figura 3.18 podemos ver os principais sub-módulos do *Sentence Generator*: o *Arranger*, o *Preliminary Sentence Generator* e o *Local Refiner*.

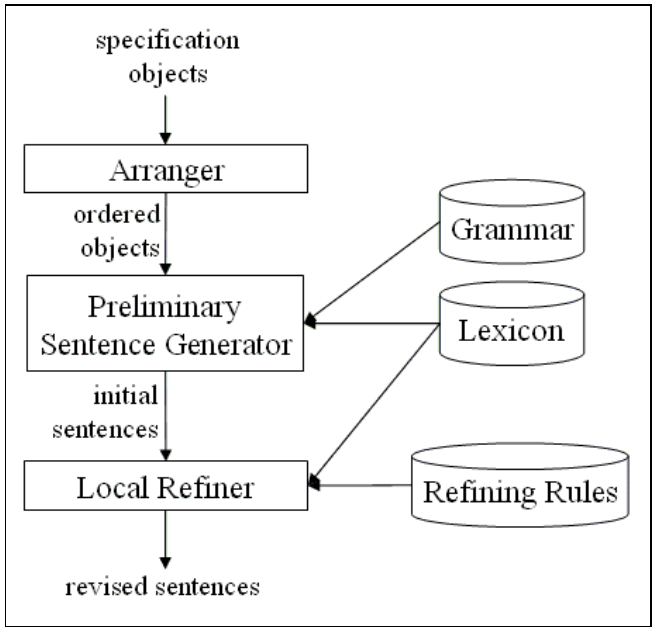


Figura 3.18: Componente de geração de sentença do Review

O *Arranger* tem como função reordenar os atributos do objeto da especificação baseado nos seus tipos, agrupando o máximo de atributos de mesmo tipo. Por exemplo, no objeto da figura 3.19, os atributos *form* e *access_method* são agrupados no ordenamento por serem do mesmo tipo *string*.

```
data_store library_holdings
    (form := "disk file"; cardinality := 4; access_method := "random";
```

Figura 3.19: Excerto de especificação SSL

O objeto com seus atributos ordenados é passado para o *Preliminary Sentence Generator* (PSG). Com a ajuda de uma gramática e de um léxico, sentenças iniciais são geradas. Para o objeto da especificação da figura 3.19, o PSG geraria a sentença inicial exibida na figura 3.20.

```
library_holdings is a data_store whose form is disk file, whose access_method
is random, and whose cardinality is 4
```

Figura 3.20: Sentença gerada a partir da especificação SSL da figura 3.18

Por fim, as sentenças geradas pelo PSG são enviadas ao *Local Refiner*, cuja tarefa é capitalizar a sentença, garantir a concordância de número e entre o sujeito e o verbo,

escolher os artigos apropriados, separar palavras compostas e realizar a pontuação. Na figura 3.21, temos a sentença do exemplo da figura 3.20 em sua versão final.

> Library holdings is a data store whose form is disk file, whose access method is random, and whose cardinality is 4.

Figura 3.21: Sentença gerada a partir da especificação SSL da figura 3.18

O sistema Review apresenta algumas técnicas complexas no seu processo de geração, como, por exemplo, o uso de uma gramática e de um léxico. Além disso, é possível identificar algumas das tarefas descritas no capítulo 2 (seção 2.3). Porém, nota-se que várias tarefas não foram realizadas ou foram mescladas com outras em um único passo.

Uma das falhas do Review é o fato de exigir que a especificação contenha alguns trechos em linguagem natural que serão usados na geração de saída. Além disso, esses trechos são escritos na própria especificação, sem nenhum controle prévio. Isso exige que o engenheiro conheça o processo de geração de LN do Review para não inserir nenhuma inconsistência no texto de saída.

3.5 Considerações Finais

Neste capítulo, foram apresentados quatro sistemas de geração de texto a partir de especificações. Os sistemas Joyce e ModelExplainer geram descrições em linguagem natural a partir de especificações sobre a estrutura de um software. O sistema Review segue uma linha parecida, gerando texto a partir de especificações formais de sistemas. Todos os três utilizam algumas das técnicas apresentadas no capítulo 2. Já o sistema ADL gera texto a partir de especificações de sistemas e utiliza a mesma técnica para também gerar descrições dos casos de teste gerados automaticamente a partir da especificação.

Dentre a literatura pesquisada, somente o sistema ADL tinha um objetivo parecido com o do SpecNL: geração de texto a partir de especificações de casos de teste. Entretanto, o sistema ADL é bastante simples e não utiliza, durante o processo de geração, as técnicas lingüísticas apresentadas no capítulo 2.

No capítulo seguinte, apresentaremos o *Test Research Project*, projeto no qual o trabalho apresentado nesta dissertação foi desenvolvido, e a entrada e a saída do SpecNL serão introduzidas.

4 Test Research Project

Antes de abordar o trabalho realizado, é importante apresentar o contexto no qual ele foi desenvolvido. Como já foi dito, este trabalho foi realizado pelo *Test Research Project*, um projeto de pesquisa fruto de uma 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 complementar os outros projetos do BTC. Durante o dia a dia, vários times de desenvolvimento do BTC identificavam oportunidades de melhoria no processo de desenvolvimento e execução de testes. Porém, devido à correria do trabalho e aos cronogramas apertados, esses times normalmente não têm tempo para explorar as possíveis soluções. O *Test Reasearch Project* aparece nesse contexto para explorar em detalhes essas oportunidades, propondo e implementando soluções para o processo de testes do BTC.

O objetivo mais amplo do projeto, como dito, é contribuir com todo o processo de testes da Motorola. Isso inclui os seguintes objetivos específicos:

- Documentação de Requisitos – desenvolvimento de uma Linguagem Natural controlada (LNC) a ser utilizada na documentação de requisitos, com o intuito de sistematizar a geração e seleção efetiva de casos de teste.
- Seleção de Casos de Teste – definir um procedimento bem definido para seleção de casos de teste, tornando 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 a documentação 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 ferramentas que permitam analisar parâmetros como cobertura e confiabilidade dos testes e estimar o tempo de execução dos casos de teste gerados.

A seguir, as atividades que estão sendo desenvolvidas no *Test Research Project* serão detalhadas.

4.1 Atividades do *Test Research Project*

A figura 4.1 exibe o fluxo de informação do *Test Research Project*. Os quadros representam os artefatos gerados pelas atividades que estão sendo desenvolvidas no projeto. As atividades são representadas pelas setas numeradas.

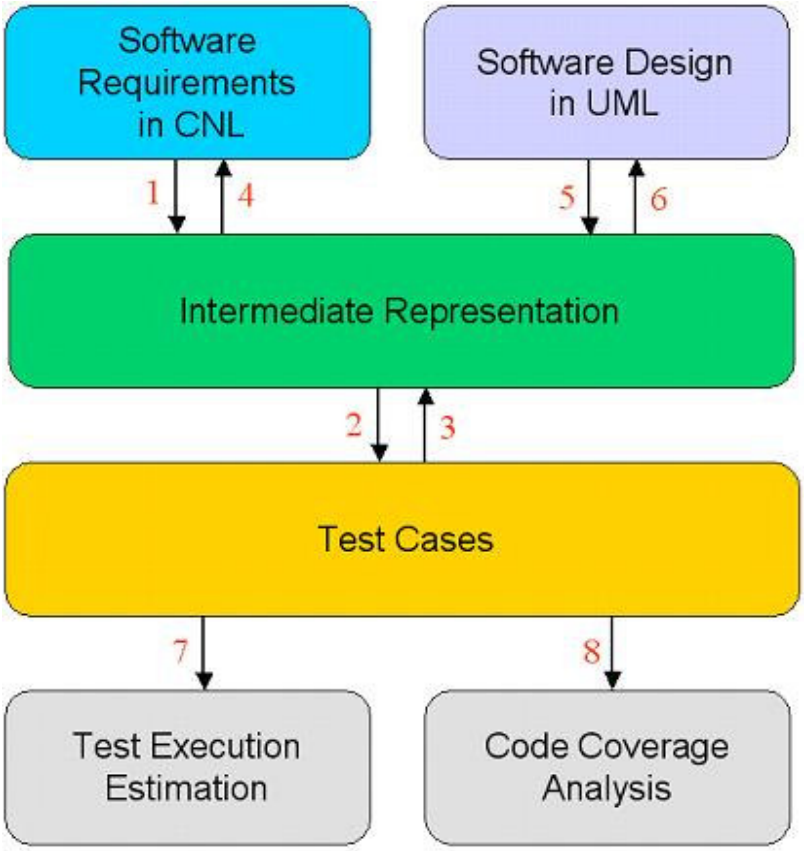


Figura 4.1: Fluxo de Informações do *Test Research Project*

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

- **Geração de Casos de Teste**: agrupa 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 UML. As atividades e artefatos que compõem este grupo serão detalhados na seção 4.1.1. Essas atividades fazem parte do processo de engenharia direta realizada pelo projeto.
- **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 4.1.2. Já esse grupo de atividades compõe o processo de engenharia reversa realizada pelo *Test Research Project*.
- **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 4.1.3.

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].

No *Test Research Project*, a estratégia para geração de testes é baseada na abordagem MBT. Os modelos formais utilizados no processo de geração são produzidos a partir da documentação do sistema em teste. Na figura 4.1, as atividades 1 e 5 geram

uma representação intermediária (modelo formal) a partir dos requisitos e de diagramas do sistema, respectivamente. A partir daí, a atividade 2 gera os casos de testes com base nessa representação intermediária. A seguir, essas três atividades que compõem a estratégia do projeto serão detalhadas.

Geração de Modelos de Uso a partir de Documentos de requisitos (atividade 1) – Esta atividade 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 é um subconjunto do Inglês (linguagem natural) que pode ser processado diretamente por uma máquina, além de ser expressiva o suficiente para ser entendida por não especialistas. Para padronizar a documentação de casos de uso, um *template* no formato Microsoft Word foi definido. A mesma LNC usada na escrita casos de uso também 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 sobre o sistema. O modelo é escrito na notação formal CSP (ver seção 4.3), sendo representado pela *Intermediate Representation* na figura 4.1.

Geração de Modelos de Uso a partir de Diagramas UML (atividade 5) – Sendo complementar à atividade anterior (atividade 1), a Geração de Modelos de Uso a Partir de Diagramas UML consiste em traduzir diagramas de seqüência UML (*Unified Modeling Language*) para modelos de uso na notação CSP. Esta é uma atividade secundária do *Test Research Project*, já que o objetivo principal do projeto é a geração de testes a partir de casos de uso.

Geração de Casos de Teste (atividade 2) – esta tarefa representa o propósito geral do projeto de pesquisa: a geração automática de teste. Esta atividade recebe como entrada o modelo de uso em CSP do sistema em teste e gera automaticamente os casos de teste, também especificados 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 é o aumento da eficiência e da produtividade, bem como da qualidade dos testes gerados.

Porém, os casos de testes gerados ainda estão sendo representados na *Intermediate Representation* (descritos na notação formal CSP). É nesse momento que entra o trabalho apresentado nesta dissertação. Com o objetivo de tornar a notação formal CSP transparente para os engenheiros de teste, os casos de teste em CSP são mapeados automaticamente para a LNC. Isso aumenta a eficiência da execução manual dos testes gerados, visto que os engenheiros não perdem tempo interpretando a especificação formal.

4.1.2 Atualização dos Requisitos

Durante o processo de desenvolvimento de software, é comum apenas os casos de testes serem atualizados de forma a refletir eventuais mudanças no código, enquanto os documentos de requisitos permanecem desatualizados. Dessa forma, o *Test Research Project* tem como meta desenvolver um procedimento sistematizado para a 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* (Anti-MBT) [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 Anti-MBT recebe como entrada os casos de testes, e tem como objetivo gerar um modelo de uso a partir desses casos de teste.

No *Test Research Project*, o processo de atualização de requisitos é representado pelas atividades 3, 4 e 6 da figura 4.1 (que têm a direção de baixo para cima). As atividades são detalhadas a seguir.

Geração de Modelos de Uso a partir de Casos de Teste (atividade 3) - O primeiro passo para a atualização dos documentos de requisitos é a tradução dos casos de testes descritos em linguagem natural ou em LNC para uma representação formal intermediária (*Intermediate Representation*). Um processador de texto desenvolvido no projeto é utilizado nessa tradução. A partir daí, ocorre a elaboração do modelo de uso

com base nas especificações formais traduzidas. Esse modelo de uso serve como entrada para a atividade descrita a seguir.

Atualização de Requisitos (atividade 4) – Esta atividade é responsável por atualizar os requisitos com informações dos casos de teste. Essa atualização é realizada através da comparação do modelo de uso obtido a partir dos requisitos, com o modelo de uso obtido a partir dos casos de teste. Dessa forma, é possível verificar pontos de discrepância entre os dois modelos e, quando necessário, atualizar o modelo de requisitos. Essa atualização é refletida nos documentos de requisitos por uma ferramenta que mapeia o modelo atualizado em texto na Linguagem Natural Controlada.

Geração de Diagramas UML (atividade 6) – Além da atualização dos requisitos, os modelos de uso, sejam oriundos dos casos de teste e/ou dos requisitos, servem de entrada para a geração automática de diagramas UML. Esta atividade, sendo automatizada, aumenta a produtividade do desenvolvimento de software.

4.1.3 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 deste projeto consiste em desenvolver modelos de estimativa para mensurar o tempo de execução dos testes gerados automaticamente. Modelos existentes estão sendo analisados e adaptados para o ambiente do projeto. A padronização dos casos de testes 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 testes gerados. Claramente, o percentual de redundância de um conjunto de casos de testes, 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.

Depois dessa apresentação geral do *Test Research Project*, veremos agora uma discussão mais detalhada sobre as entradas e saídas do SpecNL. 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]. Estudos sugerem que as atividades de teste levam aproximadamente 50% do custo total do desenvolvimento do software [Boehm, 1981]. Com isso, muitas ferramentas têm sido propostas com o intuito de automatizar e otimizar o processo de teste de software, assistindo desde a geração até a execução dos testes.

Os Testes de Software podem ser classificados da seguinte forma [Watkins, 2000]:

- *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 do código implementado;
- *Gray-box* – é um teste *black-box* baseado no 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, como especificado a seguir [Jorgensen, 1995]:

- Entradas
  (1) Pré-condição – assegura a condição inicial para que o caso de teste possa ser executado;

(2) Passos (ações) – passos a serem executados, identificados pelos métodos de teste.

- Saídas

  (1) Resultado Esperado – respostas esperadas do sistema, para o passo 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 com a 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 figura 4.2 ilustra um exemplo de caso de teste descrito 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 4.2: Exemplo de Caso de Teste

A seguir, a notação CSP utilizada para especificar os casos de teste será apresentada. A seção 4.3 enfoca apenas os operadores e funcionalidades utilizadas na especificação dos casos de teste.

4.3 CSP (*Communicating Seqüencial Process*)

CSP (*Communicating Sequential Processes*) [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 serão utilizados neste trabalho, pois a teoria que envolve CSP é bastante 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, a especificação da Figura 4.3 determina que o canal *enviar* pode transmitir qualquer valor do tipo Mensagem, ou seja, qualquer dos eventos *enviar.SMS*, *enviar.MMS* e *enviar.EMS* podem ocorrer.

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

Figura 4.3: Exemplo de *datatype* simples

CSP permite o desenvolvimento de *datatypes* mais complexos, que envolvem tuplas, conjuntos, seqüências, etc. Por exemplo, o *datatype* da figura 4.4 é formado por uma tupla que contém uma mensagem e um conjunto de destinatários.

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

Figura 4.4: Exemplo de uso de tupla em um *datatype*

Também é possível definir *datatypes* a partir de outros *datatypes*. O *datatype MensagemGenerica* da figura 4.5 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 MesnagemGenerica*.

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

Figura 4.5: Exemplo de *datatype* composto

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 e 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.

Os exemplos de processos a seguir ilustram o uso dos operadores algébricos de CSP (Figura 4.6).

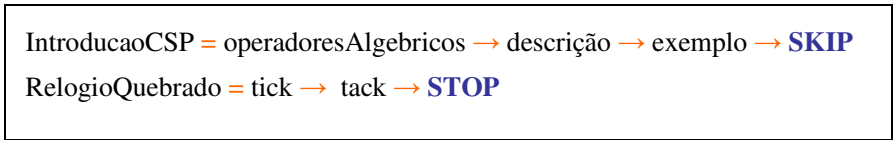


Figura 4.6: Exemplo de Processos CSP

O primeiro dos processos da figura 4.6 é o processo *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 geração de casos de teste em CSP a partir de modelos de uso formais. É justamente nesse contexto que o SpecNL é aplicado, sendo responsável pelo mapeamento das especificações CSP dos casos de testes em descrições em linguagem natural.

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

No capítulo 5, o SpecNL é apresentado. As bases de conhecimento da ferramenta serão detalhadas, bem como as funcionalidades de cada módulo de processamento.

5 SpecNL - Gerando Texto a partir de Especificações

Neste capítulo, detalharemos o SpecNL, uma ferramenta voltada para a geração de descrições em linguagem natural a partir de especificações de casos de teste. Como visto no capítulo 4, este trabalho faz parte do *Test Research Project*, que visa melhorar o processo de testes da Motorola para aplicações móveis. Uma das tarefas desse projeto consiste na geração automática de casos de teste a partir de modelos de uso. Estes casos de teste gerados são especificados na notação formal CSP, o que vem a ser uma dificuldade para os engenheiros de teste, visto que eles precisam compreender a notação CSP para executar os testes com sucesso.

O SpecNL aparece nesse contexto para tornar a notação formal CSP transparente ao engenheiro de teste, aumentando a eficiência e a produtividade da equipe de execução de testes. Além disso, documentos de casos de teste gerados automaticamente implicam na eliminação dos erros de escrita, melhorando a qualidade da documentação dos testes e reduzindo o tempo de inspeção dos documentos.

A ferramenta SpecNL foi desenvolvida dentro da abordagem tradicional de geração de linguagem natural. A ferramenta possui quatro módulos de processamento e conta com cinco bases de conhecimento. Entretanto, a ferramenta deve ser considerada um sistema híbrido (ver seção 2.4), uma vez que ela utiliza estruturas lingüísticas para tratar a especificação de entrada e, ao mesmo tempo, usa *templates* de saída no processo de realização sintática.

Aa literatura relacionada apresenta vários trabalhos de geração de linguagem natural a partir de especificações de sistemas. Contudo, pouquíssimos desses trabalhos exploraram a geração de texto a partir de especificações de casos de teste, propósito do SpecNL. Dentre a literatura pesquisada, apenas o projeto ADL [ADL 2001] (ver seção 3.3) se propõe a gerar texto a partir de especificações de teste. Entretanto, a técnica de

geração empregada no ADL é muito simples e não faz uso de bases de conhecimento, o que não permite, por exemplo, o reuso dos termos léxicos utilizados no texto de saída. O SpecNL foi desenvolvido utilizando técnicas mais complexas de GLN, objetivando portabilidade e manutenibilidade, tanto dos módulos de processamento, como das bases de conhecimento.

A seguir, a seção 5.1 apresenta a arquitetura geral da ferramenta; a seção 5.2 detalha as bases de conhecimento utilizadas pelo SpecNL; e, por último, a seção 5.3 apresenta em detalhe as funcionalidades de cada módulo de processamento da ferramenta.

5.1 Arquitetura

A figura 5.1 apresenta a arquitetural geral do SpecNL. Os retângulos representam os quatro módulos de processamento (seção 5.3): *Processador da Especificação de Entrada*, *Gerador de Case Frames*, *Realizador Sintático* e *Formatador da Saída*. Os cilindros correspondem às bases de conhecimento (seção 5.2): *Léxico*, *Gramática de Casos*, *Ontologia*, *Especificação de Entrada* e *Templates de Saída*.

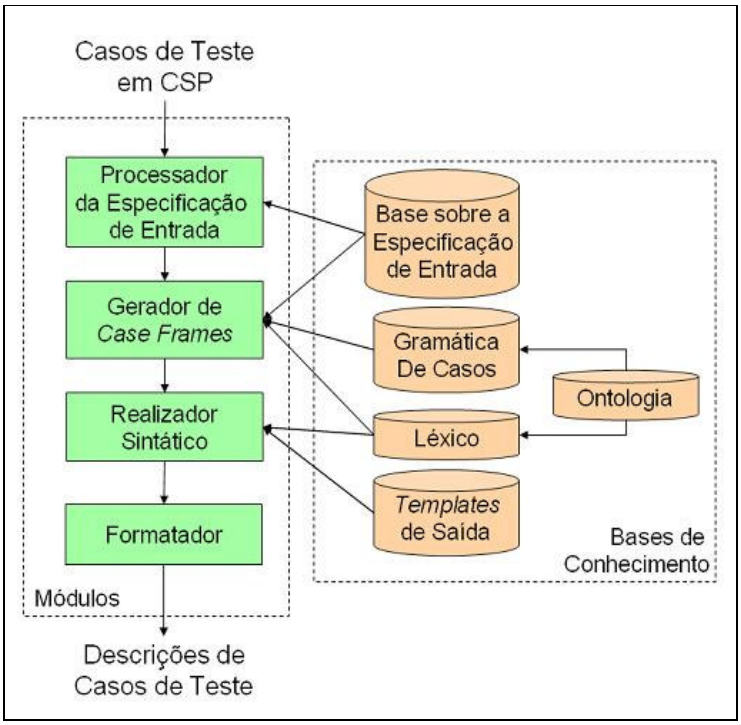


Figura 5.1: Arquitetura Geral do SpecNL

Como dito, o desenvolvimento do SpecNL foi baseado na abordagem tradicional de GLN. Entretanto, nem todas as tarefas apresentadas na seção 2.3 são realizadas no processo de geração de LN do SpecNL. A tarefa de *Planejamento de Sentença* não é realizada, pois os passos dos casos de teste podem ser tratados como sentenças individuais, não sendo necessário existir uma ligação entre as sentenças das descrições de saída.

As tarefas de *Planejamento do Discurso* e *Determinação de Conteúdo* são ambas realizadas no módulo *Processador da Especificação de Entrada*, que além de processar a especificação de entrada, identifica as partes do caso de teste. Como mostrado na seção 4.2, um caso de teste é composto por pré-condições, passos (ações), resultados esperados e pós-condições.

Apesar de o SpecNL ter sido desenvolvido para processar especificações de entrada em CSP, ele pode ser adaptado para alguma outra linguagem de entrada. Para isso, é necessário alterar o módulo *Processador da Especificação de Entrada* e a base *Especificação de Entrada*.

5.2 Bases de Conhecimento

Sistemas de inteligência artificial utilizam conhecimento sobre o domínio para solucionar problemas de maneira eficiente. Em alguns casos, esse conhecimento é embutido no código do sistema. Entretanto, por motivos de modularidade, reusabilidade e manutenibilidade, é mais adequado armazenar o conhecimento sobre o domínio em bases de conhecimento.

Nesta seção, as bases de conhecimento do SpecNL serão detalhadas. Todas as bases foram descritas no formato XML [W3C, 2006].

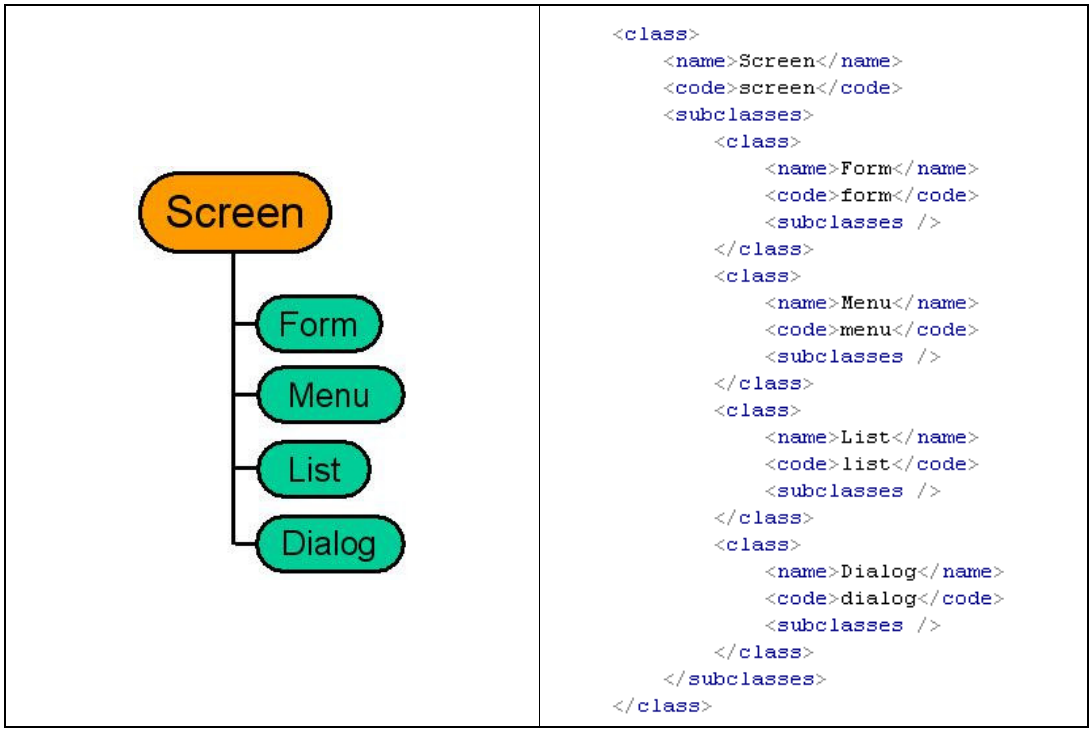
5.2.1 A Ontologia

Ontologias são bases de conhecimento que contêm definições sobre as entidades que compõem o domínio da aplicação. O desenvolvimento de uma ontologia objetiva 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.

No SpecNL, a ontologia é usada para restringir o uso dos termos nas sentenças de saída, impossibilitando que sentenças semanticamente incorretas sejam geradas. O grande desafio ao desenvolver o SpecNL foi mantê-lo independente de domínio. Para isso, foi fundamental que todas as bases de conhecimento fossem personalizáveis, inclusive a ontologia. Por exemplo, para fazer o SpecNL gerar descrições de teste para o domínio de aplicações para celular, a ontologia poderia conter classes como *menu*, *dialog*, *soft key*, etc. Entretanto, caso a intenção seja a geração descrições de teste no domínio de televisores domésticos, as classes da ontologia poderiam ser, por exemplo, *remote control*, *screen*, *panel button*, *speakers*, etc.

Com isso, é necessário permitir que o usuário defina a sua própria ontologia, realizando também a manutenção. A figura 5.2 exibe um exemplo de ontologia para o domínio de aplicações para celular. Na esquerda da figura, está uma ilustração gráfica das classes da ontologia, e na direita, a ontologia no formato XML é exibida.




```
<class>
    <name>Screen</name>
    <code>screen</code>
    <subclasses>
        <class>
            <name>Form</name>
            <code>form</code>
            <subclasses />
        </class>
        <class>
            <name>Menu</name>
            <code>menu</code>
            <subclasses />
        </class>
        <class>
            <name>List</name>
            <code>list</code>
            <subclasses />
        </class>
        <class>
            <name>Dialog</name>
            <code>dialog</code>
            <subclasses />
        </class>
    </subclasses>
</class>
```

Figura 5.2: Fragmento de uma ontologia do SpecNL

A ontologia do SpecNL contém apenas relações de especialização entre as classes (topologia de árvore). Essa decisão foi tomada para facilitar a adição de novas entidades ao Léxico (seção 5.2.2), visto que é necessário apenas atribuir uma classe à nova entidade.

5.2.2 O Léxico

O léxico armazena todos os termos que podem aparecer nas descrições de saída. Ele é baseado na abordagem *phrasal lexicon* [Becker, 1975], na qual os termos da base podem ser compostos por mais de uma palavra. 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 texto à nova situação. A complexidade dos termos de um *phrasal lexicon* pode variar desde uma agregação simples de palavras que possuem um significado específico (e.g. "*White House*", "*in order to*"), até termos léxicos que codificam informações gramaticais da linguagem [Zernik & Dyer, 1997].

As entradas do Léxico do SpecNL podem ser de três tipos:

- Verbo (*Verb*) – representam verbos das sentenças de saída e representam uma ação a ser executada pelo engenheiro de teste. Por exemplo, *select*, *create*, *scroll to*, *open*, etc.
- Nome (*Noun*) – 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*.
- Modificador (*Modifier*) – 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 tipo de entrada possui um conjunto de atributos associados, que descrevem suas características. As entradas que representam verbos possuem os seguintes atributos: (1) *term*, que representa o verbo no infinitivo; (2) *thirdperson*, especificando a conjugação do verbo na terceira pessoa do singular; (3) *past*, representa a conjugação do

verbo no passado; (4) *participle*, representa a conjugação do verbo no particípio passado; e (5) *gerund*, especifica o gerúndio do verbo. Os atributos *thirdperson*, *past*, *participle* e *gerund* não são obrigatórios. Esses atributos, caso não sejam especificados, são construídos em tempo de execução a partir do atributo *term*, que é obrigatório. A figura 5.3 exibe um exemplo em XML de um verbo.

```
<verb>
    <term>select</term>
    <thirdperson>selects</thirdperson>
    <past>selected</past>
    <participle>selected</participle>
    <gerund>selecting</gerund>
</verb>
```

Figura 5.3: Exemplo de verbo

Cada nome no Léxico possui os seguintes atributos: (1) *term*, representando o próprio termo léxico no singular; (2) *plural*, que não é obrigatório e especifica o plural do nome; (3) *class*, que relaciona o nome a uma classe da ontologia, classificando as entidades do domínio; e (4) *model*, representando um código que é utilizado nas especificações CSP. O atributo *plural*, caso não seja informado, é construído em tempo de execução. A figura 5.4 ilustra um exemplo de entrada de nome. Note que o atributo *plural* não foi informado.

```
<noun>
    <term>inbox folder</term>
    <plural/>
    <class>list</class>
    <model>INBOX_FOLDER</model>
</noun>
```

Figura 5.4: Exemplo de nome

Por fim, os atributos de um modificador são: *term*, representando o próprio termo léxico; *position*, indicando se o *modifier* deve aparecer antes ou depois do nome que ele modifica; *precedence*, utilizado para determinar a ordem dos modificadores na sentença, caso exista mais de um modificador relacionado a um mesmo nome; *numberinflection*, que determina se o nome ao qual o modificador está relacionado deve aparecer no plural; *article*, indicando se o *modifier* pode ser precedido por algum artigo; e, finalmente, o atributo *model*, que informa como aquele modificador aparece na especificação de entrada. A figura 5.5 exibe um exemplo de modificador.

```
<modifier>
    <term>unprotected</term>
    <position>before</position>
    <precedence>1</precedence>
    <numberinflection>singular</numberinflection>
    <article>yes</article>
    <model>UNPROTECTED</model>
</modifier>
```

Figura 5.5: Exemplo de modificador

Os termos do tipo modificador podem ainda conter *slots*, que são preenchidos com base nas informações da especificação de entrada. Os *slots* podem ser: <*int*/>, que aceita apenas números inteiros; ou <*noun*/>, que aceita os nomes do léxico. A figura 5.6 ilustra um modificador com um *slot* que aceita números inteiros.

```
<modifier>
    <term>at least <int/></term>
    <position>before</position>
    <precedence>0</precedence>
    <numberinflection>plural</numberinflection>
    <article>no</article>
    <model>AT_LEAST.Int</model>
</modifier>
```

Figura 5.6: Exemplo de modificador do tipo *template*

Para ilustrar o uso de modificadores, vejamos a sentença "*Select at least 3 unprotected messages*", que contém os modificadores das figuras 5.5 e 5.6. O atributo *position* dos modificadores indica que eles devem aparecer antes do nome *message*. O modificador "*at least 3*" possui o atributo *precedence* menor que o do modificador "*unprotected*", e com isso, aparece antes no ordenamento dos modificadores. Por fim, o nome "*messages*" aparece no plural por exigência do modificador "*at least 3*" (ver atributo *numberinflection*).

5.2.3 A Gramática de Casos

Conhecimento semântico é usado para se atribuir uma interpretação semântica à sentença sob análise, a partir do significado de seus termos e suas inter-relações. A atribuição de significado deve ser independente da ordem de ocorrência dos termos na sentença, i.e., independente da estrutura sintática da sentença. Existem diversos

formalismos na área de geração de linguagem natural para tratar a informação semântica, e.g., *Meaning-Text Theory* [Melcuk, 1988], *Sentence Planning Language* [Kasper, 1989] e Gramática de Casos [Fillmore, 1968]. O formalismo de representação semântica da linguagem adotada no SpecNL foi a Gramática de Casos.

As Gramáticas de Casos (*case grammar*) foram introduzidas por Charles Fillmore (1968). Esse formalismo propõe analisar cada frase como uma combinação de um verbo principal com um conjunto de "casos" (i.e., papéis temáticos), como por exemplo, "agente", "instrumento", "local", "paciente", etc. Cada combinação de verbo com papéis temáticos é chamada de *case frame*. Por exemplo, na sentença "*John broke the window with the hammer*", o verbo principal é "*to break*", enquanto os papéis temáticos são "*John*" como agente, "*window*" como paciente, e "*hammer*" como instrumento.

Os papéis temáticos relacionados ao verbo podem ser obrigatórios ou opcionais. Um papel temático é obrigatório se a sentença for gramaticalmente incorreta caso ele seja omitido. Por exemplo, a frase "*John broke*" é gramaticalmente incorreta, visto que o papel temático "paciente" é obrigatório (alguém quebra alguma coisa).

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.

Entretanto, algumas anomalias podem ser apontadas na gramática de casos, como por exemplo, em sentenças que não possuem um verbo principal. A frase "*The car is yellow*", por exemplo, não possui verbo principal, apenas o verbo auxiliar *is*. Caso *is* seja considerado o verbo principal e *car* seja o paciente, existe um problema para identificar o papel temático do adjetivo *yellow*. Para resolver esse caso, Jacobs & Rosenbaum (1968) sugerem tratar o adjetivo *yellow* como o verbo principal da sentença e *car* como seu paciente. Porém, segundo Efe & Ng (1987), isso além de ser uma possível fonte de inconsistência na gramática de casos, é uma contradição nas convenções lingüísticas (um adjetivo funcionando como um verbo).

Um importante trabalho que envolve o formalismo de Gramática de Casos é o projeto FrameNet [Baker *et al*., 1998]. O objetivo do projeto é mapear todos as possíveis combinações semânticas entre cada palavra em cada um de seus significados. Atualmente, o FrameNet conta com um total de 800 *case frames*.

A Gramática de Casos do SpecNL

Como dito, o formalismo de representação semântica adotado no SpecNL foi a Gramática de Casos. O formalismo é bastante adequado à aplicação em foco: descrições de casos de teste. As sentenças que descrevem os procedimentos do caso de teste são isoladas e possuem uma composição regular e bastante previsível (ver seção 4.2), tornando a aplicação da Gramática de Casos adequada [Efe & Ng, 1987].

A Gramática de Casos é também facilmente extensível, 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.

Um outro bom motivo para adotar a gramática de casos, é o fato de vários realizadores sintáticos receberem como entrada *case frames* preenchidos com termos léxicos, como, por exemplo, o KPML [Bateman, 1997] e o SURGE [Elhadad & Robin, 1996]. Isso acaba facilitando a adaptação do SpecNL para o uso desses realizadores sintáticos. No momento, o SpecNL, utiliza *templates* de saída na realização sintática (seção 5.2.5).

A figura 5.7 exibe um *case frame* da gramática definida para o SpecNL. O primeiro atributo, <*name*>, guarda o nome do case frame, que dever ser único na base, identificando essa entrada da gramática. O segundo atributo, <*verblist*>, traz a lista de todos os verbos que o *case frame* representa. Os verbos de um mesmo *case frame* devem ser sinônimos e podem ser substituídos na frase sem alteração da semântica. A figura também mostra a lista de papéis temáticos do *case frame*. Cada papel temático, especificado na figura pelo elemento XML <*role*>, pode ser obrigatório (atributo *mandatory* igual a *True*) ou opcional (*mandatory* igual a *False*).

```
<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*

O *case frame* do SpecNL difere daquele utilizado pelo FrameNet [Baker *et al*., 1998], no qual verbos não sinônimos podem aparecer no mesmo *case frame*. Por exemplo, o case frame do FrameNet *Cause_harm* descreve situações em que um agente machuca uma vítima (papel temático paciente). Nesse *case frame* estão listados, entre outros, os verbos "*cut*" (cortar) e "*burn*" (queimar), que não são sinônimos e não poderiam ser trocados em uma sentença sem alteração do significado.

Uma lista mais extensa de exemplos de *case frames* da nossa gramática pode ser vista no Apêndice A, seção A.3.

Restrições da Gramática

A gramática de casos possui ainda um conjunto de restrições que limitam as possíveis entidades que podem preencher os papéis temáticos do *case frame*. Essa restrição é feita associando-se uma classe da ontologia 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 de restrições restringe o texto de saída, não permitindo que sentenças semanticamente incorretas sejam geradas.

A figura 5.8 exibe um conjunto de restrições relacionadas ao case frame da figura 5.7. Como pode ser observado, três restrições <*restriction*> foram especificadas para o *case frame* de nome *SelectItem*. Cada restrição é identificada unicamente para o *case frame* pelo atributo <*name*>. Em cada restrição, os papéis temáticos são relacionados às classes da ontologia. Na figura 5.8, a restrição de nome "*DTSEL_ITEM*" restringe o uso do papel temático "*theme*" à classe "*item*" da ontologia. Vale salientar que, se o papel

temático for obrigatório no *case frame*, ele deve estar relacionado na restrição a uma classe da ontologia.

```
<frame>
    <name>SelectItem</name>
    <restrictions>
        <restriction name="DTSEL_ITEM">
            <class role="theme">item</class>
        </restriction>
        <restriction name="DTSEL_LISTITEM_LIST">
            <class role="theme">list_item</class>
            <class role="from-loc">list</class>
        </restriction>
        <restriction name="DTSEL_MENUITEM_MENU">
            <class role="theme">menu_item</class>
            <class role="from-loc">menu</class>
        </restriction>
    </restrictions>
</frame>
```

Figura 5.8: Exemplo de restrições de *case frame*

5.2.4 Base sobre a Especificação de Entrada

Nesta seção, detalharemos a base de dados que armazena conhecimento sobre a especificação de entrada. 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 dos passos (ações) 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.

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* do SpecNL e os canais CSP.

Classes da Ontologia X *Datatypes*

No SpecNL, os *datatypes* CSP são definidos a partir das classes da ontologia, existindo um *datatype* para cada classe da ontologia. 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).

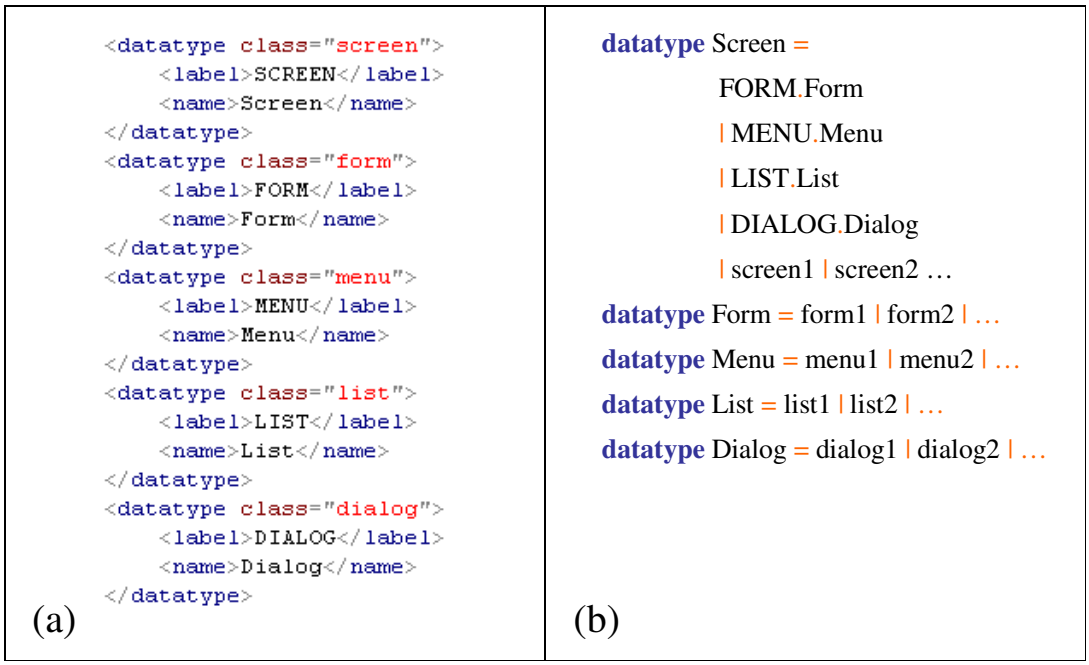


Figura 5.9: Exemplo de *datatype* CSP

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*". O papel temático *agent* não aparece na especificação, ficando subentendido que é sempre o usuário que realiza a ação.

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 importante 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.

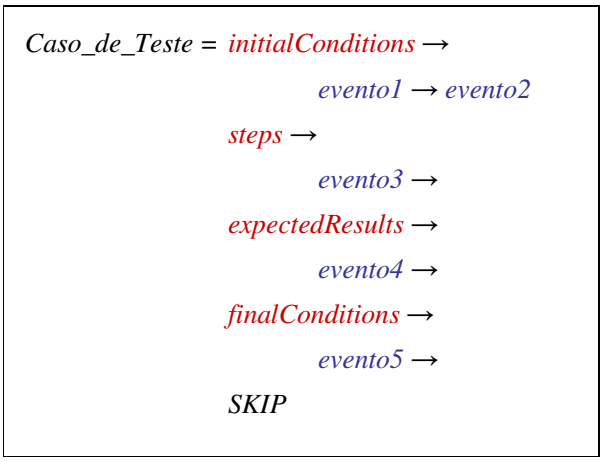


Figura 5.11: Ilustração de caso de teste em CSP

5.2.5 Templates de Saída

A gramática de casos apresentada é a forma de o SpecNL representar o conhecimento semântico do texto de saída. Para representar o conhecimento sintático, o SpecNL faz uso de *templates*, que são utilizados pelo módulo de Realização Sintática na geração das descrições dos casos de teste.

Diferentemente de realizadores sintáticos baseados em gramática, o uso de *templates* não requer tanto poder computacional, e torna o desenvolvimento da ferramenta mais eficiente [Pianta & Tovena, 1999]. Além disso, a criação dos *templates* não exige conhecimento especializado, o que permite desenvolvedores com pouco *expertise* em NLG adicionar novos *templates* à base.

No SpecNL, um *template* representa um passo do caso de teste (uma única sentença). Cada *template* é especificamente desenvolvido para representar sintaticamente um *case frame*, podendo existir mais de um *template* para um *case frame*. Dessa forma, não encontraremos, no texto de saída, mais de um *case frame* sendo mapeado em um único passo de caso de teste[7].

[7] Essa decisão de projeto visa simplificar o módulo de Realização Sintática. O mapeamento de dois ou mais *case frames* em uma única sentença não agrega valor lingüístico ao texto de saída que justifique a sua implementação.

Entretanto, é possível definir *templates* que agreguem mais de um case frame em uma sentença. Para isso, é necessário definir regras no módulo de Realização Sintática que lidem com esses *templates*. A figura 5.12 define um conjunto de *templates* que mapeiam o *case frame* ilustrado na figura 5.7.

```
<entry>
    <caseframe>SelectItem</caseframe>
    <outputs>
        <template info="default">
            <verb>
                <tense>infinitive</tense>
            </verb>
            <article agreement="1"/>
            <role id="1">theme</role>
        </template>
        <template info="default">
            <verb>
                <tense>infinitive</tense>
            </verb>
            <article agreement="1"/>
            <role id="1">theme</role>
            of <article agreement="2"/>
            <role id="2">from-loc</role>
        </template>
    </outputs>
</entry>
```

Figura 5.12: Conjunto de *templates* para o case frame *SelectItem*

Os *templates* de saída podem conter os seguintes elementos:

- *String* fixa – são pedaços de texto que não são alterados dinamicamente. Como pode ser visto no segundo *template* da figura 5.12, a única *string* fixa presente é "*of*".
- *Article* (artigo) – representa um artigo definido ou indefinido. A decisão de qual tipo de artigo utilizar fica a cargo do módulo de Realização Sintática. Inclusive, o módulo pode decidir por não utilizar nenhum artigo.
- *Verb* (verbo) – neste *slot*, o verbo já conjugado é inserido. Para especificar a conjugação a ser utilizada, dois atributos foram definidos: (1) *tense*, que representa o tempo de conjugação, podendo ser *past*, *present* ou *future* (passado, presente ou futuro, respectivamente); (2) *voice*, podendo assumir os valores *active* ou *passive* e indicando se a conjugação deve estar na voz ativa ou passiva, respectivamente.

- *Role* (papel temático) – este é o *slot* no qual o papel temático é mapeado. Nele são inseridos o nome e os modificadores a ele relacionados. O ordenamento dos termos léxicos fica a cargo do módulo de Realização Sintática.

Como pode ser observado na figura 5.12, os elementos do *template* (exceto as *strings* fixas) ainda podem possuir os atributos *agreement* e *id*. Esses atributos são usados juntos para garantir a concordância de número dos artigos e verbos com os papéis temáticos. Por exemplo, a figura 5.13 exibe um possível *template* para gerar a sentença seguinte na voz passiva: "*Two messages were created in the inbox folder*". Como pode ser observado no *template*, o verbo deve concordar em número com o seu sujeito, o papel temático *theme*.

```
<template info="default">
    <article agreement="1"/>
    <role id="1">theme</role>
    <verb agreement="1">
        <tense>past</tense>
        <voice>passive</voice>
    </verb>
    in <article agreement="2"/>
    <role id="2">from-loc</role>
</template>
```

Figura 5.13: Exemplo de uso dos atributos *agreement* e *id*

5.3 Módulos de Processamento

O processo de geração de linguagem natural do SpecNL é realizado por quatro módulos conectados em *pipeline*. Os módulos são:

- Processador da Especificação de Entrada – primeiro módulo do *pipeline* e tem a função de determinar o conteúdo que será expresso no texto de saída.
- Gerador de *Case Frames* – tem como função criar os *case frames* a serem usados na geração, e preenchê-los com os termos léxicos necessários.

- Realizador Sintático – tem como objetivo mapear, com o auxílio de *templates*, os *case frames* gerados pelo módulo anterior em estruturas sintáticas de superfície.
- Formatador da Saída – com as descrições de casos de teste já geradas, este módulo tem como objetivo formatar o texto de saída, de maneira que seja processado por algum software de visualização de texto.

A seguir, os módulos de processamento do SpecNL serão detalhados

5.3.1 Processador da Especificação de Entrada

Este módulo, como o primeiro do *pipeline*, recebe como entrada especificações de casos de teste em CSP. A especificação é composta por um conjunto de processos CSP, cada um representando um caso de teste. Entretanto, cada processo é tratado individualmente.

A primeira tarefa deste módulo é processar sintaticamente a especificação. Para isso, deve ser utilizado um *parser* para a linguagem formal CSP. A saída do *parser* é a árvore sintática da especificação, que é então vasculhada com o objetivo de identificar cada caso de teste, suas seções e as ações que as compõem. Este módulo ainda verifica se a especificação de entrada segue o formato padrão definido pelo *Test Research Project*.

Como saída, este módulo retorna uma representação intermediária do caso de teste, de forma a facilitar o processamento do próximo módulo do *pipeline*. Nessa representação, cada evento CSP é desmembrado com o intuito de identificar a ação especificada pelo evento e as entidades que participam da ação.

Fazendo um paralelo com as tarefas de GLN apresentadas na seção 2.3, este módulo é responsável por duas tarefas: (1) *Determinação de Conteúdo*, visto que o módulo processa a especificação de entrada e define as informações que serão expressas nas descrições de saída; e (2) *Planejamento do Discurso*, que apesar de simples, visa identificar cada seção do caso de teste, de maneira a organizar o texto final.

5.3.2 Gerador de Case Frames

Com cada evento CSP desmembrado, e a ação e as entidades nela envolvidas identificadas, é necessário determinar os termos léxicos que devem representar a ação e as entidades no texto de saída.

Este módulo recebe como entrada a representação intermediária gerada pelo módulo anterior, sendo responsável por gerar como saída *case frame*s lexicalizados (*case frames* com os papéis temáticos preenchidos com termos léxicos). Cada ação do caso de teste (especificada por um evento CSP) é representada por um *case frame*. A função deste módulo é procurar no Léxico (seção 5.2.2) o verbo que representa a ação e os nomes que representam as entidades envolvidas na ação, juntamente com os seus modificadores. A figura 5.14 ilustra o *case frame* lexicalizado para a sentença "*Select at least three unprotected messages from inbox folder*".

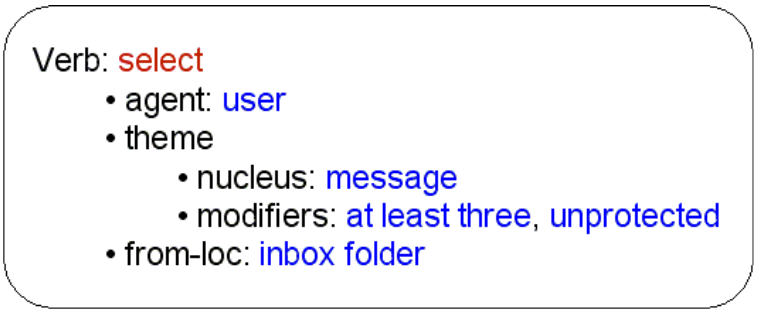


Figura 5.14: Ilustração de *case frame* lexicalizado

É possível existir mais de um verbo representando uma ação e mais de um nome representando uma entidade. Este módulo deve escolher qual termo léxico mais apropriado para gerar o texto de saída. Comparando com as tarefas de GLN detalhadas na seção 2.3, este módulo realiza a etapa de *Lexicalização*.

5.3.3 Realizador Sintático

Com as entidades identificadas, e seus termos léxicos selecionados, o módulo *Realizador Sintático* tem como função construir as descrições finais do caso de teste. Como entrada, o módulo recebe os *case frames* lexicalizados gerados pelo módulo anterior. As tarefas realizadas pelo *Realizador Sintático* são detalhadas a seguir.

Escolha do *Template* de Saída – o primeiro passo para a geração das descrições em LN consiste em determinar o *template* de saída a ser usado. Como foi dito, cada *case frame* é mapeado em uma sentença através de um *template* de saída. Porém, pode existir mais de um *template* relacionado a um *case frame*, sendo necessário um mecanismo para selecionar apenas um *template* dentre os vários existentes. O processo de escolha de *templates* leva em consideração as seguintes características:

- Os papéis temáticos que foram lexicalizados pelo módulo anterior. Alguns papéis temáticos não são obrigatórios, e podem não ter sido lexicalizados. Com isso, deve ser escolhido um *template* que desconsidere esse papel temático não lexicalizado.

- A seção de caso de teste à qual o *case frame* lexicalizado está relacionado. Cada seção de caso de teste possui uma forma diferente para expressar a informação. Por exemplo, na seção de pré-condições (ver seção 4.2), as sentenças são escritas no presente e na voz passiva, por exemplo, "*An object is highlighted in the object list*". Já na seção que descreve as ações/passos do caso de teste, as sentenças são escritas no imperativo. "*Highlight an object in the object list*". Ou seja, existem *templates* de saída específicos para cada seção do caso de teste, sendo necessário escolher aquele apropriado para a seção à qual o *case frame* lexicalizado está relacionado.

Mapeamento dos Papéis Temáticos – como pode ser visto na seção 5.2.5, os *templates* contém *slots* do tipo <*role*>, nos quais o conteúdo dos papéis temáticos do *case frame* lexicalizado deve ser inserido. Como vimos, o papel temático é composto por um nome e um conjunto de modificadores. Sendo assim, é necessário ordenar esses termos léxicos a fim de garantir que a sentença final seja sintaticamente correta. Nesse ordenamento, o nome é tomado como referência e os modificadores são colocados antes ou depois do nome, de acordo com o atributo *position* de cada modificador. Logo após, é necessário definir a ordem dos modificadores que aparecem antes e depois do nome. Para isso, o atributo *precedence* dos modificadores é levado em consideração.

Conjugação dos verbos – a seção 5.2.5 também mostrou que os *templates* contem *slots* do tipo <*verb*>, nos quais o verbo que representa a ação do caso de teste deve ser inserido. Para isso, o Realizador Sintático deve, de acordo com os atributos do *slot*

<*verb*>, conjugar o verbo na forma requerida. A conjugação do verbo deve levar em consideração o número (plural ou singular) do seu sujeito, representado por um papel temático lexicalizado. O SpecNL utiliza o atributo *agreement* para identificar com qual papel temático o verbo deve concordar.

Determinação dos artigos – os *templates* de saída ainda podem conter *slots* do tipo <*article*>, no qual um artigo (definido ou indefinido) pode ser inserido. A decisão entre utilizar um artigo definido ou indefinido (ou simplesmente não utilizar artigo) é tomada com base no atributo *agreement*, que especifica com qual papel temático este artigo deve concordar.

Por exemplo, o sintagma nominal "*three messages*", não pode ser precedido por um artigo indefinido, visto que ele está no plural, i.e., três entidades "*message"* estão sendo referenciadas. Com isso, um artigo definido deve ser utilizado, formando "*the three messages*". Entretanto, caso essas entidades estejam sendo referenciadas pela primeira vez (referência inicial), o artigo definido não pode precedê-las. Sendo assim, nenhum artigo será utilizado. A determinação do uso de artigos pode ser implementada através de regras que analisam o texto a ser gerado para definir a necessidade do uso do artigo. Aprendizagem de máquina também pode ser utilizada para implementar esta tarefa. Segundo Knight & Chander (1994), o uso adequado de artigos facilita a leitura, e melhora a fluência do texto.

Na seção 2.3.4 desta dissertação, dois tipos de referências podem ser observadas: referência inicial e subseqüente. Como descrito, um dos propósitos da tarefa de determinação de artigos é tentar descobrir o tipo de referência (inicial ou subseqüente) para, a partir daí, definir o artigo mais apropriado.

A saída do módulo Realizador Sintático é uma estrutura que relaciona as sentenças geradas com as seções do caso de teste da qual fazem parte. Essa estrutura é enviada para o módulo Formatador, que organiza o texto de acordo com a necessidade do usuário. Comparando com as etapas de GLN apresentadas na seção 2.3, este módulo realiza, obviamente, a tarefa de *Realização Sintática*. O módulo também procura tornar clara a identificação das entidades expressadas no texto através do uso de artigos, evitando assim ambigüidades. Isso é uma característica da tarefa de *Geração de Expressões de Referência*.

5.3.4 Formatador

A saída do módulo de Realização sintática é uma estrutura que armazena a seqüência de frases que descreve o caso de teste. Entretanto, estas frases não estão formatadas de maneira que facilite a visualização das descrições. O módulo Formatador tem como objetivo apresentar essa estrutura em um formato que permita o processamento do caso de teste gerado por alguma ferramenta de visualização de texto, como por exemplo, Microsoft Word, Microsoft Excel ou Netscape Navigator.

A seguir, será apresentada uma Natural Controlada que pode ser definida a partir das bases de conhecimento do SpecNL, sendo uma contribuição secundária deste trabalho.

5.4 Linguagem Natural Controlada

Uma Linguagem Natural Controlada (LNC) é um subconjunto de uma linguagem natural, como o Inglês ou Português. Segundo Hartley & Paris (2001), uma LNC prescreve as construções gramaticais, um vocabulário comum e uma terminologia especializada que os autores estão habilitados a usar. Com isso, uma LNC é uma variação restringida de uma linguagem humana, desenvolvida para atender as necessidades comunicativas de uma área particular, ou até de uma empresa particular.

Uma das demandas do *Test Research Project* era uma LNC a ser usada na edição de documentos de casos de teste e casos de uso[8] (ver capítulo 4). Essa demanda surgiu ao se observar ambigüidades e falta de uniformidade na documentação dos requisitos e dos testes. Além do mais, um texto descrito em uma LNC pode ser processado facilmente.

O objetivo principal do trabalho apresentado nesta dissertação foi o desenvolvimento de uma ferramenta capaz de gerar descrições em Inglês a partir de especificações em CSP. Entretanto, o trabalho realizado na construção das bases de conhecimento do SpecNL dá suporte ao desenvolvimento de uma LNC.

A base do Léxico (seção 5.2.2) define o conjunto de termos (vocabulário) que podem ser utilizados na edição dos casos de teste. Os *case frames* (ver seção 5.2.3)

[8] O mesmo estilo de linguagem é utilizado nos documentos tanto de casos de uso como de casos de teste.

tratam a LNC no nível semântico, impossibilitado, através das restrições, que sentenças semanticamente incorretas sejam editadas. Já no nível sintático, os *templates* de saída (ver seção 5.2.5) são utilizados para especificar as possíveis formações sintáticas da LNC.

Essa LNC, definida a partir das bases de conhecimento do SpecNL, é um resultado secundário do trabalho realizado nesta dissertação. Ela já está sendo utilizada de forma experimental na Motorola. Entretanto, há a necessidade de se desenvolver uma ferramenta para a validação e assistência na edição dos casos de teste e casos de uso com a LNC. O desenvolvimento dessa ferramenta será considerado um trabalho futuro desta dissertação.

5.5 Considerações Finais

Neste capítulo 5, a ferramenta SpecNL foi detalhada. A ferramenta tem como objetivo a geração de descrições em linguagem natural a partir de especificações de casos de teste. A arquitetura do SpecNL foi apresentada, e os quatro módulos de processamento da ferramenta foram detalhados, bem como as suas cinco bases de conhecimento.

Como mostrado, o SpecNL foi desenvolvido com base na abordagem tradicional de GLN. Porém, a ferramenta é considerada um sistema híbrido, uma vez que *templates* de saída são utilizados no processo de realização sintática, ao mesmo tempo estruturas lingüísticas são usadas para tratar a especificação de entrada.

Vimos também que a ferramenta não agrega mais de um *case frame* em uma sentença. Para tornar isso possível, um novo módulo de processamento deve ser construído. Através de regras de agregação, esse módulo descobriria, dentre os *case frames* lexicalizados gerados, aqueles que podem ser unidos em uma mesma sentença. Apesar de não ser uma característica fundamental no processo de geração do SpecNL, a agregação de *case frames* pode garantir ao texto gerado uma maior variabilidade e legibilidade.

O grande desafio no desenvolvimento do SpecNL foi torná-lo independente de domínio. Com isso, as bases de conhecimento precisaram ser especialmente

desenvolvidas para garantir essa independência e para permitir a atualização das bases pelo usuário da ferramenta. O desenvolvimento de uma interface gráfica para a manutenção das bases de conhecimento aparece como trabalho fundamental no intuito de facilitar o trabalho de manutenção das bases.

No próximo capítulo, um protótipo do SpecNL para o domínio de aplicações móveis será apresentado. A implementação dos módulos processadores será detalhada, bem como as bases de conhecimento utilizadas. Também será apresentado o experimento ao qual o protótipo foi submetido e os resultados obtidos.

6 Protótipo e Testes

Este capítulo apresenta o estudo de caso desenvolvido para validar o SpecNL, compreendendo a implementação de um protótipo e a realização de experimentos. O estudo de caso foi desenvolvido no *Motorola Brazil Test Center* (BTC), no domínio de testes para aplicações para telefones móveis. Foram realizados dois experimentos com casos de teste para aplicações no domínio de *Messaging*, que engloba as funcionalidades de manipulação de mensagens da Motorola.

A seguir, a seção 6.1 apresenta o desenvolvimento do protótipo, detalhando as classes Java das bases de conhecimento (BCs) e a implementação do protótipo. Na seção 6.2, os experimentos realizados serão apresentados.

6.1 O Protótipo

O protótipo do SpecNL foi modelado segundo os conceitos de orientação a objetos. A implementação foi feita na linguagem Java, e procurou manter critérios desejáveis de qualidade de software, como reusabilidade, extensibilidade e modularidade. A metodologia de desenvolvimento adotada foi baseada em *eXtreme Programming* (XP) e *refactoring* [Fowler, 2000]. Assim, um esforço relativamente pequeno foi necessário para a criação de uma primeira versão do projeto do sistema, contudo, sempre que uma modificação estrutural era necessária, a parte do protótipo afetada pela modificação era re-projetada. Seguindo essa metodologia, não se corre o risco de um mau projeto inicial causar grandes problemas no decorrer da implementação, pois o projeto de um sistema normalmente está em processo constante de evolução.

Veremos agora detalhes de implementação das bases de conhecimento, apresentadas na seção 6.1.1 a seguir.

6.1.1 Bases de Conhecimento

Esta seção detalha as classes *Java* que implementam a entidades das BCs (termos léxicos, *case frames*, classes da ontologia, etc.). Também será apresentado o mecanismo utilizado pelo protótipo para a manipulação das bases (o *handler*).

Como visto na seção 5.2, as bases de conhecimento são estruturas que armazenam informações lingüísticas e sobre o domínio da aplicação. A escolha da estrutura de armazenamento dessas informações (banco de dados, arquivos, etc.) e do formato dos dados deve levar em conta fatores como performance, escalabilidade e manutenibilidade. Nesta primeira versão do protótipo, as BCs são armazenadas em arquivos no formato XML, uma vez que essa linguagem apresenta as seguintes vantagens: (1) existem ferramentas e APIs que lêem e escrevem no formato XML; (2) é uma tendência na área de processamento de linguagem natural[9]; (3) facilita a inserção e atualização manual dos dados, tarefa muito freqüente na etapa de aquisição de conhecimento para popular as BCs.

Contudo, o uso de arquivos XML não é muito adequado para grandes volumes de dados, devido à enorme quantidade de metadados, e à ausência de mecanismos que garantam a consistência dos dados armazenados. Levando em consideração uma possível mudança na estrutura de armazenamento das BCs, desenvolvemos um mecanismo de leitura e escrita das bases que tornasse essa estrutura transparente.

O mecanismo utilizado foi um *handler*, responsável pelas funcionalidades de leitura, escrita, remoção das informações das bases de conhecimento. O desenvolvimento do *handler* seguiu o padrão de projeto *Persistence Data Collections* (PDC) [Massoni *et al.*, 2001] que define duas camadas de software:

(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 estrutura de

[9] *Workshop on NLP and XML*, http://www.ling.helsinki.fi/~gwilcock/NLPXML/

armazenamento dos dados. Esta camada deve implementar uma interface pré-definida de métodos de acesso à base.

O uso desse padrão de projeto facilita a portabilidade do SpecNL para outra estrutura de armazenamento dos dados, visto que é apenas necessário reimplementar a camada de persistência, seguindo a interface de métodos pré-definida. O restante da implementação do SpecNL permanece o mesmo, uma vez que depende apenas da interface de métodos.

Para cada base de conhecimento descrita na seção 5.2, um *handler* de acesso foi implementado. Cada *handler* é composto por uma classe de negócios e uma ou mais classes que fazem acesso aos repositórios das bases. A figura 6.1 exibe o diagrama de classes da implementação do *handler* para a base do Léxico. A classe *Lexicon* pertence à camada de negócios da arquitetura PDC. A interface *ILexiconRepository* contém a declaração dos métodos de acesso ao repositório de dados. Essa interface é implementada pela classe *LexiconXMLRepository*, que codifica o acesso aos arquivos XML. Os *handlers* das outras bases são implementados da mesma forma, e portanto, não serão detalhados aqui.

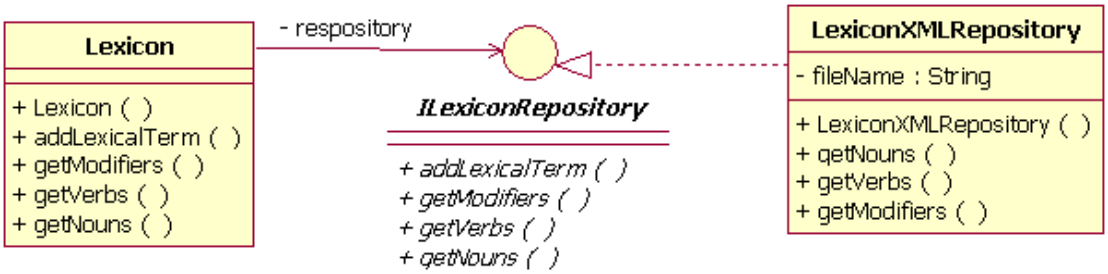


Figura 6.1: Exemplo de *handler* de acesso às bases de conhecimento

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 6.2 exibe a representação em Java da classe da ontologia do SpecNL, representada pela classe *OntologyClass*. Como pode ser visto, a classe foi simplificada, contento apenas os atributos *name* (nome da classe), *code* (identificador da classe), *superclass* (a superclasse) e *subclasses* (a lista de subclasses).

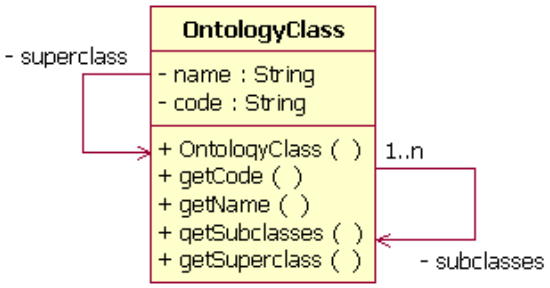


Figura 6.2: Classe Java básica da Ontologia

É possível acessar a superclasse e as subclasses através dos métodos *getSuperclass()* e *getSubclasses()*. Com isso, é possível acessar qualquer classe da ontologia.

Léxico

A figura 6.3 exibe o diagrama de classes Java do Léxico. A classe *LexicalTerm* representa um termo genérico. As seguintes classes representam os possíveis termos léxicos de uma sentença: *NounTerm* representa um nome; *VerbTerm* representa um verbo do Léxico; *ModifierTerm* representa um modificador; e por fim, a classe *CardinalTerm* representa um número cardinal.

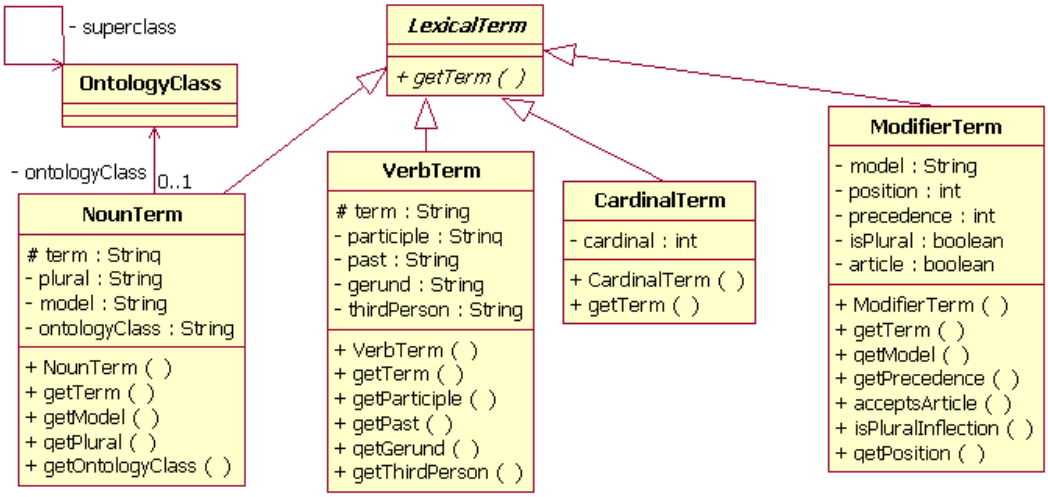


Figura 6.3: Classes Java básicas do Léxico

Como pode ser observado, a classe *VerbTerm* comporta atributos para representar as formas verbais infinitivo, terceira pessoa do singular, gerúndio, particípio e passado atendendo às nossas necessidades de flexão verbal. Contudo, essa representação não suporta todas as variações do verbo *to be*, de conjugação irregular. O verbo *to be* possui

mais de uma conjugação no passado (*was* e *were*) e no presente (*am*, *is* e *are*), o que não é suportado pela classe Java definida. Assim, o verbo *to be* recebeu um tratamento particular, embutido no código da implementação do protótipo, de forma a suportar todas as possíveis conjugações.

Gramática de Casos

Como mostrado na seção 5.2.3, a gramática de casos é composta por *case frames* e suas restrições. A fim de modularizar a implementação do *handler* de acesso, essas informações foram armazenadas em dois repositórios separados, um para armazenar os *case frames* e outro para as restrições.

A figura 6.4 exibe o diagrama de classes Java da implementação da gramática de casos. Um *case frame* é representado em Java pela classe *CaseFrame*, que contém um conjunto de instâncias da classe *Role* (papel temático). A classe *Restriction* representa uma restrição do *case frame*. Uma restrição determina, através da classe *RoleClassRestriction*, a relação entre um papel temático e uma classe da ontologia.

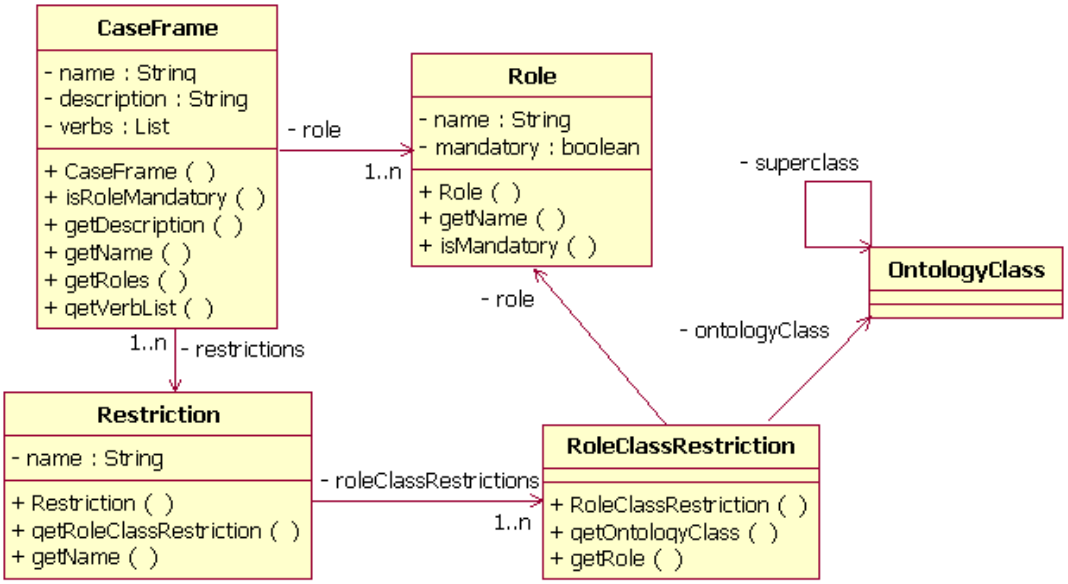


Figura 6.4: Classes Java básicas da Gramática de Casos

Como dito, cada *case frame* representa um conjunto de verbos "sinônimos" que possuem os mesmo papéis temáticos. Aqui, o verbo *to be* não requer tratamento especial, uma vez que a relação entre o verbo e o *case frame* é feita usando apenas o infinitivo.

Base sobre a Especificação de Entrada.

Da mesma forma que a gramática de casos, dois repositórios foram utilizados para armazenar as informações sobre a especificação de entrada: um para armazenar os canais (*channels*) e outro os tipos de dados (*datatypes*). As classes Java que representam os canais e tipos de dados são apresentadas na figura 6.5.

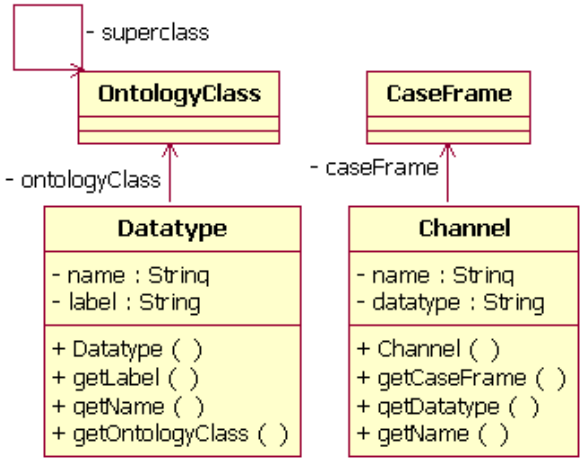


Figura 6.5: Classes Java básicas da especificação de entrada

A classe *Datatype* representa um tipo de dado CSP, e possui como atributo uma classe da ontologia. A classe *Channel* representa um canal CSP, e está relacionada a um *case frame*.

Templates de Saída.

Os *templates* de saída são representados em Java através das classes apresentadas no diagrama da figura 6.6. A classe *CaseFrameOutput* agrega todos os *templates* relacionados a um *case frame*. Um *template* de saída é representado pela classe *OutputTemplate*, e é formado por uma lista de instâncias da classe *OutputObject*, que representa um objeto genérico do *template*. Os objetos do *template* podem ser: um artigo (*OutputArticle*); um verbo (*OutputVerb*); uma string fixa (*OutputString*); ou, finalmente, um papel temático (*OutputRole*).

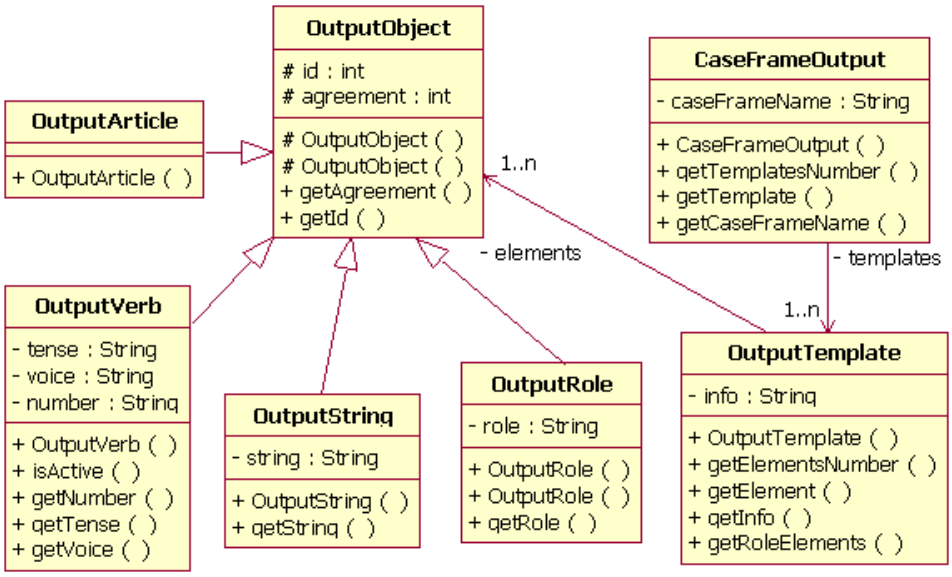


Figura 6.6: Classes Java básicas dos *templates* de saída

A seguir, será detalhada a implementação dos módulos de processamento do protótipo.

6.1.2 Módulos de Processamento

Nesta seção, detalharemos o desenvolvimento dos quatro módulos do protótipo do SpecNL. Os módulos foram desenvolvidos em Java, buscando a modularização e o desacoplamento.

Antes de abordar a implementação dos módulos, apresentaremos como um caso de teste é representado em Java pelo protótipo. Essa representação utiliza *generics* da versão 1.5 da linguagem Java. Foi definida uma classe para um tipo genérico em Java que especifica um caso de teste como contendo uma lista seções (pré-condições, passos, etc.). Dessa forma, é possível representar um caso de teste e suas seções independentemente do tipo de representação utilizada para especificar os seus passos ou condições (eventos CSP, *case frames* lexicalizados ou seqüência de caracteres). Além de tornar o código reusável, essa estrutura padroniza a representação dos casos de teste. A figura 6.7 exibe o diagrama de classes do caso de teste. A classe *TestCase* representa um caso de teste e a classe *TCSection* representa uma seção.

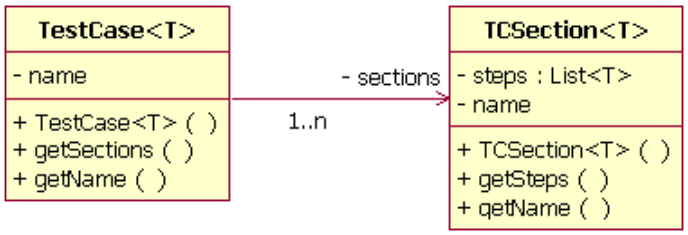


Figura 6.7: Representação do caso de teste em Java com o uso de *generics*

A seguir, será detalhado o desenvolvimento de cada módulo deste protótipo.

Processador da Especificação de Entrada

Como mostrado na seção 5.2.1, este primeiro módulo tem a função de processar a especificação CSP de entrada, encontrar as ações formais e separá-las em seções. Para facilitar o desenvolvimento e a manutenção, este módulo foi dividido em duas partes: (1) a primeira processa e encontra os eventos CSP (ações formais e eventos de controle); e a segunda separa as ações formais em suas respectivas seções.

O módulo recebe como entrada uma seqüência de caracteres contendo a especificação CSP do caso de teste. Para realizar o processamento da especificação, a primeira parte deste módulo faz uso de um *parser* CSP desenvolvido no Centro de Informática da UFPE. O resultado do *parser* é a árvore sintática da especificação CSP. Os nós da árvore são então percorridos, e a lista de eventos CSP do caso de teste é extraída.

A figura 6.8 exibe as classes Java que modelam a especificação CSP. Um processo (*CSPProcess*) é formado por uma lista de eventos, que podem ser de controle (*CSPEvent*), ou que podem representar uma ação formal (*CSPMsgEvent*). Cada ação formal é formado por uma lista de mensagens (*CSPMessage*), que , por sua vez, são formadas por uma lista de modificadores (*CSPModifier*).

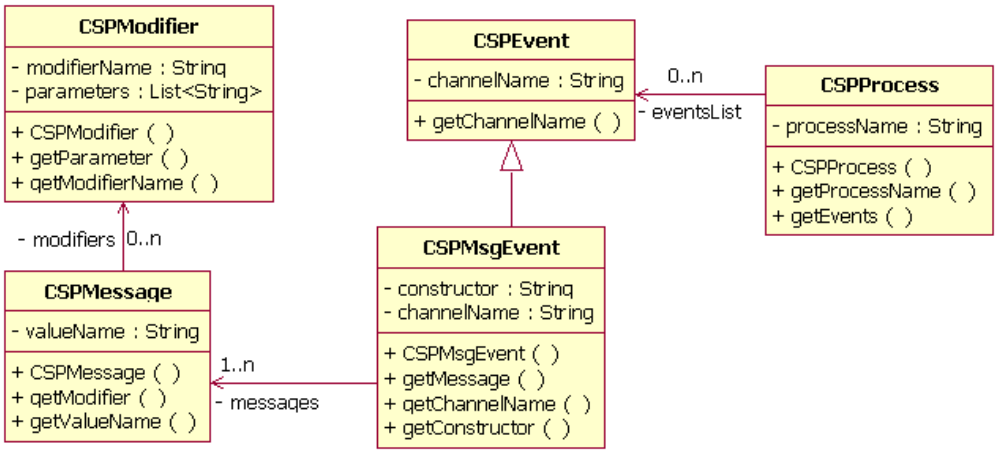


Figura 6.8: Diagrama das classes de eventos CSP.

A partir daí, a segunda parte do processamento recebe como entrada a lista de eventos CSP de um caso de teste, e gera, como saída, uma instância da classe *TestCase* para o tipo *CSPMsgEvent*[10].

Gerador de Case Frames

Este módulo é responsável por gerar *case frames* lexicalizados, a partir dos eventos CSP gerados pelo módulo anterior. A figura 6.9 mostra o diagrama das classes que representam os *case frames* lexicalizados. A classe *CaseFrameInstance* representa em Java um *case frame* lexicalizado, que é composto por uma lista de objetos da classe *CaseFrameRoleInstance*, representando um papel temático lexicalizado.

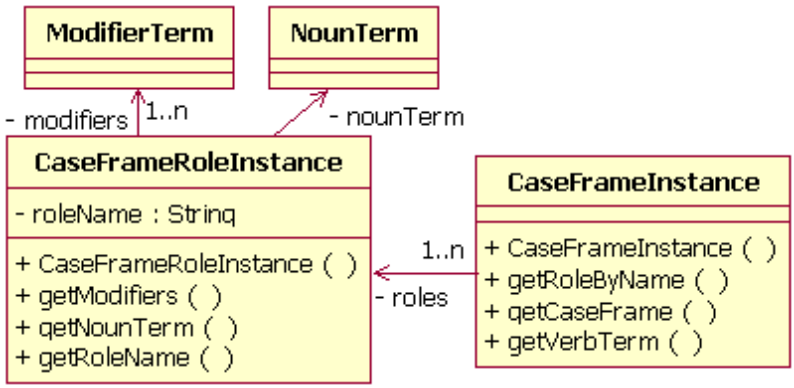


Figura 6.9: Diagrama das classes que representam um *case frame* lexicalizado

A entrada deste módulo são objetos Java que especificam os casos de teste CSP (*TestCase<CSPMsgEvent>*) gerados pelo módulo anterior. Como saída, são gerados

[10] Em Java, o tipo seria declarado como *TestCase<CSPMsgEvent>*.

objetos Java que especificam casos de teste cuja representação dos passos são *case frames* lexicalizados (*TestCase<CaseFrameInstance>*).

Para isso, o módulo deve buscar no Léxico os termos que representam as entidades das ações do caso de teste. Como o objetivo do SpecNL é gerar documentação de teste para um único tipo de leitor (engenheiros de teste da Motorola), não é necessária nenhuma variação no estilo lingüístico do texto gerado. Com isso, a escolha dos termos léxicos é a mais simples possível, escolhendo-se o primeiro termo encontrado no Léxico[11].

Realizador Sintático

Como mostrado na seção 5.3.3, uma das tarefas do módulo de realização sintática consiste em escolher o *template* de saída adequado para descrever a ação do caso de teste. No protótipo do SpecNL, o Realizador Sintático leva em consideração dois tipos de *template*: (1) um primeiro que descreve estados do sistema em teste e especifica sentenças na voz passiva; (2) e um segundo que especifica sentenças no imperativo e é usado para descrever as ações executadas pelo usuário. Caso a seção do passo do caso de teste seja pós-condições, pré-condições ou resultados esperados, o primeiro *template* (1) é escolhido. Caso contrário, o *template* que descreve ações no imperativo (2) é selecionado.

Outra tarefa do Realizador Sintático é a determinação dos artigos. Neste protótipo, regras foram usadas para escolher o artigo a ser usado. As regras são especificadas em XML, com o intuito de facilitar a definição de novas regras pelo usuário. O módulo conta com uma lista de regras que são testadas seqüencialmente. Aquela que disparar primeiro define o artigo a ser utilizado. Entretanto, caso nenhuma regra seja disparada, o Realizador Sintático verifica no histórico do discurso se a entidade referenciada pelo artigo já apareceu no texto. Caso afirmativo, será usado o artigo definido *the*. Caso contrário, o artigo indefinido *a*/*an* é utilizado.

Existem dois tipos de regras: (1) o primeiro tipo define o artigo que acompanha o papel temático de um case frame; (2) o segundo tipo é usado para determinar o artigo que acompanha uma entidade de uma classe especifica. Para exemplificar, a figura 6.10, exibe uma regra de cada tipo.

[11] O SpecNL permite mais de um termo léxico representando a mesma entidade. Entretanto, espera-se que, no ambiente Motorola, apenas um termo por entidade esteja presente na base.

(a)

```
<frame name="GotoItem">
    <role name="to-loc" article="none">
        <except>list_item</except>
        <except>field_value</except>
    </role>
</frame>
```

(b)

```
<class article="definite">menu_item</class>
```

Figura 6.10: Regras de determinação de artigos

A regra (a) da figura 6.10 é disparada sempre que se deseja conhecer o artigo que acompanha o papel temático *to-loc* do *case frame* de nome *GotoItem*. No caso, o atributo *article* igual a *none* define que nenhum artigo deve utilizado. Entretanto, se a entidade do papel temático *to-loc* for da classe *list_item* ou *field_value*, a regra não será disparada. Já a regra (b) é mais simples e define que qualquer entidade da classe *menu_item* deve estar acompanhada do artigo definido *the*. A utilização dessas regras atingiu uma taxa de acerto de 90%. Entretanto, para manter essa taxa, a base de regras precisa ser atualizada cada vez que novas classes da ontologia e novos *case frames* são criados.

Formatador

Neste momento do *pipeline*, todo trabalho de geração já foi realizado, restando, portanto, organizar as descrições do caso de teste em um formato adequado. A primeira versão do protótipo organizava as descrições em arquivos de texto ASCII. Entretanto, para adequar ao ambiente da Motorola, o módulo Formatador foi ajustado para gerar planilhas Miscrosoft Excel contento as descrições dos casos de teste. Este módulo conta com a biblioteca POI [Apache, 2006], desenvolvida em Java pelo projeto Jakarta da Apache, e usada para ler e escrever arquivos Microsoft Word e Microsoft Excel.

6.2 Experimentos e Resultados

Nesta seção, detalharemos os dois experimentos realizados com o protótipo do SpecNL. Os experimentos foram realizados no ambiente Motorola, e contaram com a ajuda de desenvolvedores do *Test Research Project*.

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 primeiro experimento realizado com o protótipo do SpecNL, e consistiu em gerar as descrições a partir de casos de teste . Esses foram gerados automaticamente a partir dos requisitos de uma aplicação para telefones móveis da Motorola. Após a realização do primeiro experimento, alguns ajustes foram feitos no protótipo, e um segundo experimento foi realizado (seção 6.2.3).

Foi desenvolvida uma interface gráfica bem simples para facilitar a manipulação dos casos de teste em CSP, e das descrições geradas durante os experimentos realizados na ferramenta. Algumas outras ferramentas foram desenvolvidas com o intuito de dar suporte ao desenvolvimento inicial e à análise das bases de conhecimento. Entretanto, o SpecNL não necessita de interface gráfica, visto que funciona como uma API (*Application Programming Interface*) a ser usada por outros sistemas do *Test Research Project* (ver capítulo 4). A seguir, os experimentos serão apresentados em detalhes. Logo depois, uma avaliação do protótipo será apresentada.

6.2.1 Preparação das Bases

Antes de se realizar qualquer experimento com o protótipo, foi necessário preencher as bases de conhecimento com um conjunto inicial de informações. Essa preparação inicial baseou-se na documentação de teste para o domínio de *Messaging* da Motorola, de onde casos de teste foram selecionados aleatoriamente. A partir desse corpus, 428 passos distintos de casos de teste foram selecionados. Esses passos foram analisados[12], e um conjunto de 544 *case frames* lexicalizados foi extraído manualmente. O número de *case frames* é maior que o número de passos de casos de teste, pois alguns deles eram formados pela união (através de pronomes relativos e conjunções) de mais de uma sentença.

A partir daí, também manualmente, foram adicionados às bases os termos léxicos, os *case frames*, restrições, classes da ontologia, etc. Após esta etapa de aquisição de dados, *templates* de saída foram criados baseando-se nos *case frames* encontrados. A tabela 6.1 exibe os números relacionados a esta preparação inicial das bases.

[12] A análise foi realizada semi-automaticamente, com o auxílio de uma ferramenta especificamente desenvolvida.

Tabela 6.1: Números da preparação inicial das bases

Nome da Entidade		Quantidade
Léxico	**Verbos**	54
	Nomes	259
	Modificadores	39
Case Frames		35
Classes da Ontologia		18
***Templates* de Saída**		55

Veremos a seguir algumas questões relacionadas à aquisição de conhecimento para preencher as BCs, destacando o número atual de entradas das bases, e entidades que receberam tratamento especial.

Ontologia

A grande maioria das entidades até agora identificadas foi classificada pela ontologia sem maiores problemas. Entretanto, algumas entidades possuíam comportamento variável, o que dificultava a classificação. Isso tem impacto direto nas restrições da gramática de casos, como será detalhado mais tarde. Até o presente momento, foi verificada a existência de 28 classes de entidades no domínio (ver Apêndice A).

Léxico

Como foi visto, o Léxico pode conter três tipos de termos: modificadores, verbos e nomes. Levando-se em consideração os modificadores e verbos, foi verificado que a tendência é o número desses termos se estabilizar, visto que há um grande reuso de modificadores e verbos[13]. Até o momento, existem 68 verbos e 60 modificadores na base. Já o número de nomes tem comportamento diferente. Cada nova aplicação testada implica na inserção de novos nomes que representam as entidades que compõem a aplicação.

[13] A quantidade de ações que podem ser executadas em uma aplicação móvel é limitada. Com isso, o número de verbos da base deve permanecer estável. O mesmo acontece com os modificadores, uma vez que as maneiras de expressar as entidades do domínio (nomes) são restritas.

Com isso, a tendência é o número de nomes da base sempre aumentar. Vale salientar que aplicações maiores e mais complexas possuem um número maior de entidades e, por conseqüência, teremos mais termos inseridos no Léxico. Levando esse fato em consideração, é inexpressivo dizer a quantidade de nomes do Léxico, visto que esse número é dependente da quantidade e da complexidade das aplicações já testadas.

Gramática de Casos

Até o momento, os 68 verbos do léxico foram distribuídos em 48 *case frames*. Foi verificado que, para o domínio do protótipo, o número atual de *case frames* deve permanecer estável, sem a necessidade de muitas atualizações. O repositório de *case frames* foi construído com base nos verbos presentes na documentação de testes da Motorola. A cada novo verbo encontrado, tentava-se encaixá-lo em algum *case frame* existente. Caso não fosse possível, um novo *case frame* era criado no repositório para suportar o verbo encontrado.

Como informado na seção 5.3.2, a gramática de casos pode apresentar anomalias para representar frases com o verbo *to be*. No domínio do SpecNL, 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 utilizada 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).

Um termo que aparece bastante nas sentenças de casos de teste é o *there is*, descrevendo a existência de alguma entidade, e.g., "*There is a contact in the contact list*". O termo *there is* foi modelado no SpecNL 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).

Como dito, as restrições sobre as gramáticas de casos são controladas por um repositório à parte. O número de restrições por *case frame* varia de acordo com a quantidade de classes da ontologia. Quanto mais classes na ontologia, mais restrições

existirão na base. Com isso, mantendo fixo o número de classes na ontologia, o número de restrições da gramática de casos permanecerá estável.

Porém, como foi citado, a Ontologia encontrou problemas para especificar certas entidades que poderiam possuir mais de um comportamento. Por exemplo, o termo *phone number* (número de telefone) pode aparecer como entidade da classe *List Item* (item de uma lista) ou *Field Value* (valor de campo de formulário). Isso tem impacto direto na construção do repositório de restrições, visto que estas fazem uso das classes da ontologia para restringir os termos léxicos que podem aparecer nos papéis temáticos dos *case frames*. Para contornar este problema, foram adicionadas, no Léxico, entidades de mesmo nome para cada possível comportamento, i.e., entidades de mesmo nome, porém de classes diferentes.

Base Sobre a Especificação de Entrada

Como já descrito na seção 5.2.4, o preenchimento da base de especificação de entrada do SpecNL é feito a partir da Gramática de Casos e da Ontologia. Assim, cada *case frame* da gramática de casos possui seu correspondente no repositório de canais. Da mesma forma, cada classe da ontologia também tem seu correspondente no repositório de tipos de dados. O conteúdo dessa base pode ser gerado automaticamente a partir dos *case frames* e classes da ontologia. Entretanto, durante o desenvolvimento do protótipo, os dados foram inseridos na base manualmente.

Templates de Saída

Neste protótipo, dois tipos de *templates* são desenvolvidos para cada *case frame*. O primeiro tipo de *template* é utilizado para descrever as ações do caso de teste executadas pelo testador. Para isso, o *template* é construído de forma a gerar como saída uma sentença no imperativo (e.g., "*Send a message to an email address*"). Esse *template* é utilizado na seção de passos/ações do caso de teste (ver seção 4.2).

Um segundo *template* é usado para descrever estados do sistema e resultados das ações executadas. Com isso, o *template* especifica a estrutura de superfície para uma sentença na voz passiva (e.g. "*A message is sent to an email address*"). Esse *template* é utilizado para gerar sentenças das seções de pré-condições, pós-condições e resultados esperados.

6.2.2 Experimento 1

Esse primeiro experimento consistiu em gerar as descrições de casos de teste para uma aplicação móvel da Motorola do domínio de *Messaging*. Para isso, foram criados 4 casos de usos em Inglês sobre a aplicação em teste, a partir dos quais um modelo de uso em CSP foi gerado automaticamente. Esse modelo serviu de entrada para o sistema de geração de testes, que foi capaz de gerar 19 casos de teste. Esse trabalho inicial de geração de casos de teste foi realizado com a ajuda de outros desenvolvedores do *Test Reseach Project.*

Durante a escrita dos casos de uso, foi necessário adicionar às bases novos termos léxicos, *case frames* identificados e classes da ontologia. A tabela 6.2 exibe os números relacionados a este experimento. É mostrada a quantidade total das entidades utilizadas para modelar a aplicação, e quantas entidades foi necessário adicionar à base.

Tabela 6.2: Números relacionados ao Experimento 1

Nome da Entidade		**Quantidade Total**	**Quantidade Adicionada à Base**
Léxico	**Verbos**	16	4
	Nomes	36	25
	Modificadores	10	6
Case Frames		16	4
Classes da Ontologia		15	5
***Templates* de Saída**		20	6

Como pode ser visto na tabela, a atualização das bases tem como gargalo a adição de nomes do léxico. Este foi o primeiro experimento realizado, com isso, a quantidade de classes da ontologia, verbos, *case frames* e *templates* de saída adicionados, mesmo que alta, foi considerada normal. A tendência é o número dessas entidades se estabilizar, como é mostrado no experimento 2.

As descrições foram geradas em um arquivo ASCII. Segundo os engenheiros de teste consultados, a qualidade geral do documento foi considerada satisfatória. Neste

primeiro experimento, foram detectados alguns problemas com a escolha dos artigos nas sentenças geradas, o que motivou o desenvolvimento de um sistema de regras mais eficaz a ser usado no segundo experimento.

6.2.3 Experimento 2

Neste segundo experimento com o protótipo do SpecNL, algumas mudanças foram realizadas de maneira a melhorar a qualidade do texto gerado. Um sistema de regras mais eficaz foi desenvolvido (ver seção 6.1.2), e as descrições passaram a ser geradas em planilhas Microsoft Excel.

Esse experimento, igualmente ao primeiro, consistiu em gerar as descrições de casos de teste para uma aplicação móvel da Motorola. O procedimento foi o mesmo: 5 casos de usos em Inglês foram criados, a partir dos quais um modelo de uso em CSP foi gerado automaticamente; em seguida, 22 casos de teste foram gerados automaticamente. A tabela 6.3 exibe os números relacionados a este experimento.

Tabela 6.3: Números relacionados ao Experimento 2

Nome da Entidade		**Quantidade Total**	**Quantidade Adicionada à Base**
Léxico	**Verbos**	12	1
	Nomes	53	42
	Modificadores	10	2
Case Frames		12	1
Classes da Ontologia		13	0
***Templates* de Saída**		20	1

Este segundo experimento demonstrou que poucos verbos, *case frames*, modificadores e classes da ontologia foram adicionados a cada nova aplicação testada. Por outro lado, o número de novos nomes adicionados ao Léxico é bem maior (ver tabela 6.3), visto que refletem as novas entidades da aplicação em teste. A seguir, diversos aspectos deste protótipo do SpecNL serão avaliados.

6.2.4 Avaliação do Protótipo

O objetivo da avaliação é determinar se o sistema alcançou de forma satisfatória os objetivos que ele se propôs a atingir. Está claro que alguma noção de avaliação é importante para a geração de linguagem natural, da mesma forma que é para o processamento de LN. Entretanto, a forma de avaliação empírica utilizada no processamento de LN, bem como suas métricas (precisão, cobertura, etc.), não são aplicáveis aos sistemas de geração de LN.

Muito pouco é conhecido sobre a melhor maneira de se avaliar sistemas de GLN [Dale & Mellish, 1998]. Uma das formas mais utilizadas é medir manualmente a qualidade do texto gerado (análise qualitativa). Entretanto, a noção de qualidade é vaga e pode variar de um humano para outro. A avaliação de sistemas de GLN também pode (e deve) considerar, além da qualidade do texto gerado, diversos outros fatores, como custo-benefício, manutenibilidade, performance, etc. A seguir, será detalhada a avaliação do SpecNL segundo os fatores mais relevantes.

Qualidade do Texto Gerado – Embora, por motivos óbvios, as descrições de casos de teste geradas pelo SpecNL não tenham a mesma flexibilidade dos escritos manualmente, os engenheiros de teste consultados consideraram o texto gerado satisfatório. Os casos de teste gerados foram, inclusive, executados com sucesso em telefones celulares, o que prova a eficácia do SpecNL em transmitir o conteúdo desejado.

Mantenabilidade das BCs – A ferramenta irá gerar descrições para novos casos de teste se tiver suas bases atualizadas. Sendo assim, as BCs estarão em constante atualização. Do ponto de vista de atualização, as entidades das BCs podem ser divididos em dois grupos: (1) entidades que não requerem do engenheiro mantenedor *expertise* em GLN e/ou no processo de geração do SpecNL; (2) entidades que requerem do engenheiro esse tipo de *expertise*.

As entidades que não requerem *expertise* (1) são os nomes do Léxico, visto que para adicionar um novo nome à base é apenas necessário saber a forma plural do termo e atribuir uma classe da ontologia. A atualização dos nomes, que é mais freqüente, ficaria sob controle dos próprios engenheiros de teste, que não possuem o *expertise* requerido.

Já as entidades que requerem *expertise* em GLN e/ou no processo de geração do SpecNL (2) são os *case frames*, classes da ontologia, *templates* de saída, verbos, modificadores. Como vimos nos experimentos, essas entidades possuem poucas atualizações[14], que podem ser realizadas por alguns poucos engenheiros mantenedores com o *expertise* necessário.

Essa descentralização da atualização das bases viabiliza o uso do SpecNL em um ambiente real, uma vez que não é exigido que todos os usuários da ferramenta tenham conhecimentos específicos de GLN e/ou do processo de geração de LN do SpecNL.

Independência de Domínio – O SpecNL foi desenvolvido de forma a facilitar a manutenção das suas bases, o que acabou garantindo à ferramenta uma independência de domínio. Para isso, as bases de conhecimento precisam ser adaptadas com informações sobre o novo domínio, sem a necessidade de alteração na implementação dos módulos.

Performance da Geração – A geração da documentação de casos de teste não exige resposta imediata do SpecNL, não sendo necessária uma alta performance. Para exemplificar, no experimento 1, o documento de casos de teste foi gerado em 3 segundos. Já no segundo experimento, o documento foi gerado em 9,3 segundos. A média é de 0,3 segundos por caso de teste, o que torna o SpecNL perfeitamente aceitável em termos de performance.

Custo-Benefício da Ferramenta – O custo-benefício do SpecNL está intimamente relacionado ao do *Test Research Project*. Para o projeto, o custo-benefício se baseia no aumento de produtividade trazido pela geração automática de casos de teste a partir de requisitos. Para ilustrar esse ganho de produtividade, experimentos relatados por Dustin *et al.* (1999) mostraram que a automação do planejamento e do desenvolvimento dos testes reduziu o esforço em 47%.

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

[14] Espera-se que com o tempo, não serão necessárias atualizações dessas entidades.

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 facilita a inserção manual de novas entidades às bases.

Foram apresentados também os dois experimentos realizados com o protótipo. Os experimentos geraram descrições de casos de teste para duas aplicações do domínio de *Messaging* da Motorola. Na seção 6.2.4, o protótipo foi avaliado segundo alguns fatores como, qualidade do texto gerado, custo benefício, performance, manutenibilidade das BCs e independência de domínio. Foi mostrado, também, que existe a possibilidade de descentralizar a atualização das bases de conhecimento, o que viabiliza o uso do SpecNL em um ambiente real.

No próximo capítulo, a conclusão deste trabalho de mestrado será apresentada, bem como algumas possíveis extensões e melhorias.

7 Conclusão

Nesta dissertação, foi apresentado o SpecNL, uma ferramenta para a geração de descrições em linguagem natural a partir de especificações CSP de casos de teste.

Inicialmente, foi realizado um estudo sobre as principais técnicas de geração de linguagem natural, e alguns trabalhos relacionados foram analisados. Esse estudo possibilitou a criação do SpecNL, que faz uso de bases de conhecimento para o armazenamento de informações lingüísticas e do domínio da aplicação, utilizadas durante o processo de geração de texto. A ferramenta foi desenvolvida de forma a facilitar a manutenção de suas bases por engenheiros sem *expertise* em GLN, e também possibilitar sua portabilidade para outros domínios.

Este trabalho foi desenvolvido como parte do projeto *Test Research Project* do CIn/BTC, em uma parceria entre o CIn-UFPE e a Motorola.

7.1 Principais Contribuições

Existem muitos estudos voltados para a geração de LN a partir de especificações de software. Entretanto, na literatura pesquisada, apenas um trabalho se preocupa em gerar LN a partir de especificações de casos de teste. Contudo, esse trabalho utiliza técnicas muito simples de geração de texto, assemelhando-se a um sistema de mala direta.

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

Também podemos citar como contribuição as bases de conhecimento (BC) do protótipo SpecNL, que podem ser utilizadas em um estudo posterior. As BCs possuem informações relevantes sobre os termos léxicos utilizados nas descrições de casos de teste para aplicações móveis da Motorola, bem como informações sobre a estrutura sintática (*templates* de saída) dessas descrições. Podemos ainda citar a Ontologia, que traz a classificação das entidades do domínio de aplicações móveis.

Uma outra grande contribuição deste trabalho foi o desenvolvimento da estrutura de uma Linguagem Natural Controlada (LNC) para a edição de documentos de casos de teste e casos de uso. Essa LNC é definida a partir das bases de conhecimento do SpecNL, e tem como objetivo padronizar o texto da documentação de teste e de requisitos.

7.2 Trabalhos Futuros

O trabalho desenvolvido obteve resultados bastante satisfatórios para a tarefa de geração de descrições em LN a partir de especificações em CSP de casos de teste. Contudo, melhorias podem ser realizadas no SpecNL, tanto nos módulos de processamento como nas suas bases de conhecimento. A seguir, são listados alguns possíveis trabalhos futuros.

Uso de propriedades para classificar as entidades do domínio.

Como mostramos, algumas entidades do domínio de aplicações móveis podem ter mais de um comportamento, o que dificulta a classificação delas pela ontologia do SpecNL. Para solucionar este problema, as entidades do domínio poderiam ter uma ou mais "propriedades" associadas. As propriedades seriam mais genéricas do que as classes da ontologia, o que facilitaria a classificação das entidades do domínio. Por exemplo, as propriedades *displayable* (que pode ser exibido) e *editable* (que pode ser escrito) poderiam ser aplicadas à entidade *phone number* (número de telefone).

Uso de aprendizagem de máquina para determinação dos artigos.

No capítulo 6, mostramos que regras foram usadas para determinar qual artigo deve acompanhar os nomes do papel temático. Entretanto, à medida que novos *case frames* e classes da ontologia são adicionados às bases de conhecimento, novas regras devem ser criadas para cobrir essas novas classes e *case frames*.

A aplicação de aprendizagem de máquina na determinação dos artigos eliminaria a necessidade de adição de novas regras. O sistema aprenderia a melhor utilização dos artigos durante a escrita de casos de uso, que estariam sendo editados na LNC. Atributos como o *case frame* da sentença, o papel temático e a classe da ontologia poderiam ser utilizados no processo de aprendizagem.

Implementação de uma GUI para manutenção das bases de conhecimento.

É muito importante a implementação de uma interface gráfica que facilite a adição, alteração e remoção de entidades das bases de conhecimento. A interface deve ser amigável o suficiente possibilitar a manutenção das bases por pessoas sem *expertise* em GLN. Além disso, ela deve ser inteligente, de forma a verificar os relacionamentos entre as entidades das bases de conhecimento (e.g., verbos na base de conhecimento e suas respectivas entradas no Léxico).

Mapeamento de mais de um case frame em uma sentença de saída.

Atualmente, cada *case frame* é mapeado em uma sentença individual. Para adicionar flexibilidade e variabilidade ao texto gerado, um dos trabalhos futuros a ser realizado é a agregação de mais de um *case frame* em uma única sentença. Para isso, é necessário adicionar um novo módulo ao *pipeline* do SpecNL que contenha regras de agregação de *case frames*. Também é necessária a criação de *templates* de saída que suportem a agregação. Além do mais, o Realizador Sintático precisa ser adaptado para manipular esses novos *templates* de agregação. Para exemplificar essa agregação, consideremos as sentenças "*The phone is in Inbox Folder*" e "*The Inbox Folder has at least 3 messages*". Elas poderiam ser agregadas em uma única sentença, formando "*The phone is in Inbox Folder with at least 3 messages*".

Desenvolvimento de um sistema para validar sentenças escritas na LNC.

Como mostrado, um resultado secundário deste trabalho foi a definição de uma linguagem natural controlada (LNC). Entretanto, apesar de toda a estrutura da LNC já estar definida, não existe nenhuma ferramenta que auxilie e/ou valide a escrita de casos de teste e casos de uso. Um trabalho futuro desta dissertação é o desenvolvimento de uma ferramenta que utilizaria os *templates* de saída do SpecNL para validar sintaticamente o texto escrito e a gramática de casos para realizar uma validação semântica.

Referências

[ADL, 2001] ADL PROJECT. Assertion Definition Language project website. 2000. Disponível em: http://adl.opengroup.org/index.html. Último acesso: 06 de Março de 2006.

[André & Rist, 1993] ANDRÉ, E.; RIST, T. The design of illustrated documents as a planning task. In: MAYBURY, M. T. *Intelligent Multimedia Interfaces*. AAAI Press, 1993. ISBN: 0262631504.

[Apache, 2006] The Apache Jakarta Project. *Jakarta POI*: Java API To Access Microsoft Format Files, Disponível em: http://jakarta.apache.org/poi/index.html. Último acesso: 10/07/2006.

[Appelt, 1985] APPELT, D. E. *Planning English Sentences*. Cambridge University Press, New York, USA, 1985. 181 p. ISBN: 0521301157.

[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.

[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.

[Bateman, 1997] BATEMAN, J. A. Enabling technology for multilingual natural language generation: the KPML development environment. *Journal of Natural Language Engineering*, vol. 3, n. 1, p. 15-55, 1997.

[Becker, 1975] BECKER, J. D. 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.

[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.

[Boehm, 1981] BOEHM, B. W. *Software Engineering Economics*. 1st ed. Prentice Hall, Oct. 1981. 1317 p. ISBN: 0138221227.

[Busemann & Horacek, 1998] BUSEMANN S.; HORACEK H. A flexible shallow approach to text generation. In: INTERNATIONAL WORKSHOP ON NATURAL LANGUAGE GENERATION, 9., 1998, Niagara-on-the-Lake, Ontario, Canada. *Proceedings*... p. 238-247.

[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.

[Dale & Mellish, 1998] DALE, R.; MELLISH, C. Towards evaluation in natural language generation. In: INTERNATIONAL CONFERENCE ON LANGUAGE RESOURCES AND EVALUATION, 1., 1998, Granada, Spain, *Proceedings*..., p. 555-562.

[Dalianis & Hovy, 1999] DALIANIS, H.; HOVY, E. H. Aggregation in Natural Language Generation. *Computational Intelligence Journal*, v. 15, n. 4, p. 384-414, Nov. 1999.

[deRaeve *et al.*, 1995] DERAEVE, J.; MCCARRON S. P.; GOGESCH B.; HAYES R. Assertion Definition Language Project: Research Results and Future Plans. In: IPA INFORMATION TECHNOLOGY SYMPOSIUM, 14., 1995. Disponível em: http://adl.opengroup.org/ documents/Archive/95symposium-final.doc.pdf, Último acesso: 08/06/2006.

[Dustin *et al.*, 1999] DUSTIN, E.; RASHKA, J.; PAUL, J. *Automated Software Testing*: Introduction, Management, and Performance. 1st ed. Addison Wesley, Jun. 1999. 608 p. ISBN: 0201432870.

[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.

[Elhadad & Robin, 1996] ELHADAD, M.; ROBIN, J. An overview of SURGE: a reusable comprehensive syntactic realisation component. In: INTERNATIONAL WORKSHOP ON NATURAL LANGUAGE GENERATION, 8., 1996, Brighton, UK. *Proceedings*..., p. 1-4.

[Elhadad *et al.*, 1997] ELHADAD, M.; ROBIN, J.; MCKEOWN, K. Floating constraints in lexical choice. *Computational Linguistics*, v. 23, n. 2, p. 195-239, Nov. 1997. ISSN: 0891-2017

[Evens, 1989] EVENS, M. Introduction to relational models of the lexicon. *Cambridge Studies In Natural Language Processing Cambridge*. University Press, 1989. p. 1-37, ISBN: 0521363004.

[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.

[Fisher, 1991] FISHER, A. S. *Case*: Using Software Development Tools. 2nd ed. John Wiley & Sons, 1991. ISBN: 0471530425.

[Fowler, 1999] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. 1st ed. Addison-Wesley Professional, Jun. 1999. 464 p. ISBN: 0201485672.

[Goldberg *et al.*, 1994] GOLDBERG, E.; DRIEDGER, N.; KITTREDGE, R. Using natural language processing to produce weather forecasts. *IEEE Expert*, p. 45-53, 1994.

[Hartley & Paris, 2001] HARTLEY, A.; PARIS, C. Translation, controlled languages, generation. In: STEINER, Erich; YALLOP, Colin. *Exploring Translation and*

Multilingual Text Production: Beyond Content, New York: Mouton de Gruyter, 2001. p 307 - 326. ISBN: 3110167921.

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

[Hovy, 1988] HOVY, E. H. Planning Coherent Multisentential Text. In: ACL CONFERENCE, 26., 1988, Buffalo, NY. *Proceedings …*

[Iordanskaja *et al.*, 1992] IORDANSKAJA, L.; KIM, M.; KITTREDGE, R.; LAVOIE, B.; POLGUERE, A. Generation of extended bilingual statistical reports. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL LINGUISTICS, 14., 1992, Nantes, France. *Proceedings…* p. 1019-1023.

[Jacobs & Rosenbaum, 1968] JACOBS, R. A.; ROSENBAUM, P. S. *English transformational grammar*. John Wiley & Sons Inc, 1968. p. 299 ISBN: 0471002887.

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

[Kasper, 1989] KASPER, K. A Flexible Interface for Linking Applications to Penman's Sentence Generator. In: DARPA SPEECH AND NATURAL LANGUAGE WORKSHOP, 1989, Philadelphia, USA. *Proceedings…*p. 153-158.

[Kittredge *et al.*, 1991] KITTREDGE, R.; KORELSKY, T.; RAMBOW, O. On the need for communication knowledge. *Computational Intelligence 7*, 1991, p. 305-314.

[Knight & Chander, 1994] KNIGHT, K.; CHANDER, I. Automated postediting of documents. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 20., 1994, Seattle, United States. *Proceedings …* Menlo Park, CA, USA, American Association for Artificial Intelligence, 1994, p. 779-784. ISBN: 0-262-61102-3.

[Koresky & Ulysses-Staff, 1988] KORELSKY, T.; ULYSSES STAFF. Ulysses: a computer security modeling enviroment. In: NATIONAL CONFERENCE ON SECURITY AND PRIVACY, 14. 1988, Baltimore, USA. *Proceedings…*

[Lavoie & Rambow, 1997] LAVOIE, B.; RAMBOW, O. A Fast and Portable Realizer for Text Generation Systems. In: CONFERENCE ON APPLIED NATURAL LANGUAGE PROCESSING, 5., 1997, Washington DC, USA. *Proceedings*... San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. p. 265-268.

[Lavoie *et al.*, 1996] LAVOIE, B.; RAMBOW, O.; REITER, E. The ModelExplainer. In: INTERNATIONAL WORKSHOP ON NATURAL LANGUAGE GENERATION, 8. 1996, Brighton, UK. *Demos and Posters*... p. 9-12. Disponível em: http://www.csd.abdn.ac.uk/~ereiter/papers/nlgw96.pdf. Último Acesso: 11 de maio de 2005

[Lavoie *et al.*, 1997] LAVOIE, B.; RAMBOW, O.; REITER, E. Customizable descriptions of object-oriented models. In: CONFERENCE ON APPLIED NATURAL LANGUAGE PROCESSING, 5. 1997, Washington, DC, USA. *Proceedings*... San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. p. 253-256.

[Mann & Thompson, 1988] MANN, W.; THOMPSON, S. Rhetorical structure theory: toward a functional theory of text organization. 1988, Text 3, p. 243-281.

[Massoni *et al.*, 2001] MASSONI, T.; ALVES, V.; SOARES, S.; BORBA, P. PDC: Persistent Data Collections pattern. In: LATIN AMERICAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMMING, 1., 2001, Rio de Janeiro, Brazil, *University of São Paulo Magazine*. ICMC, São Paulo, Brazil, p. 311-326.

[Mathiessen, 1991] MATHIESSEN, C. Lexico(grammatical) choice in text generation. *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, p. 249-292, 1991.

[McKeown, 1985] MCKEOWN, K. R. Discourse strategies for generating natural-language text. *Artificial Intelligence Journal*, v. 27, n. 1, p. 1-41, 1985.

[McKeown *et al.*, 1990] MCKEOWN, K.; ELHADAD, M.; FUKUMOTO, Y.; LIM, J.; LOMBARDI, C.; ROBIN, J.; SMADJA, F. Natural language generation in COMET. In: DALE, R.; MELLISH, C.; ZOCK, M. *Current Research in Natural Language Generations*. Academic Press Professional Inc., San Diego, CA, USA. 1993. p. 103-139 ISBN: 0122007352.

[Melcuk, 1988] MELCUK, I. A. *Dependency Syntax: Theory and Practice*. State University of New York Press, 1988. 428 p. ISBN: 0887064507.

[Nogier & Zock, 1992] NOGIER, J. F.; ZOCK, M. Lexical Choice As Pattern Matching. *Conceptual structures: current research and practice*, Ellis Horwood, Upper Saddle River, NJ, USA, 1992. p. 413-435. ISBN: 0131758780.

[Paiva, 1998] PAIVA, DANIEL S. *A Survey of Applied Natural Language Generation Systems*. ITRI Technical Report Series. Information Technology Research Institute, University of Brighton, UK, 1998. Disponível em: http://www.itri.brighton.ac.uk/techreports/. Último acesso em: 20 de Abril de 2005.

[Paris *et al.*, 1995] PARIS, C.; VANDER LINDEN, K.; FISCHER, M.; HARTLEY, A.; PEMBERTON, L.; POWER, R.; SCOTT, D. A support tool for writing multilingual instructions. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 14., 1995, Montreal, Canada. *Proceedings*... p. 1398-1404.

[Pianta & Tovena, 1999] PIANTA, E.; TOVENA, L. M. Mixing representation levels: The hybrid approach to automatic text generation. In: CONVENTION OF ARTIFICIAL INTELLIGENCE AND SIMULATION OF BEHAVIOUR (AISB'99), 1999, Edinburgh, UK. *Proceedings* ... p. 8-13.

[Rambow, 1990] RAMBOW, O. Domain Communication Knowledge. In: WORKSHOP ON NATURAL LANGUAGE GENERATION, 5. 1990, Dawson, USA. *Proceedings*... p. 87-94.

[Rambow & Korelsky, 1992] RAMBOW, O.; KORELSKY, T. Applied Text Generation. In: CONFERENCE ON APPLIED NATURAL LANGUAGE PROCESSING, 3. 1992, Trento, Itália. *Proceedings*... Morristown, NJ, USA: Association for Computational Linguistics, 1992. p. 40-47.

[Reiter, 1995] Reiter, E. NLG vs. Templates. In: EUROPEAN WORKSHOP ON NATURAL-LANGUAGE GENERATION, 5, 1995, Leiden, The Netherlands, 1995, *Proceedings* ...

[Reiter & Dale., 1997] Reiter, E.; Dale, R. Building applied natural language generation systems. *Journal of Natural Language Engineering*, v. 3, n. 1, p. 57-87, 1997.

[Reiter & Dale, 2000] Reiter, E.; Dale, R. *Building Natural Language Generation Systems*. 1st ed. Cambridge University Press, 2000. 248 p. ISBN: 0-521-62036-8.

[Reiter *et al.*, 1995] Reiter, R.; Mellish, C.; Levine, J. Automatic generation of technical documentation. *Applied Artificial Intelligence*, p. 259-287, 1995.

[Reiter & Mellish, 1993] REITER E, MELLISH C. Optimizing the Costs and Benefits of Natural Language Generation. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (IJCAI-1993), 13, 1993, Chambery, France, *Proceedings* ..., p. 1164-1169.

[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.

[Salek *et al.*, 1994] SALEK A. K., SORENSON P. G., TREMBLAY J. P., AND PUNSHON J. M. The REVIEW system: From formal specifications to natural language. In: INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING, 1., 1994, Colorado Springs, *Proceedings*... p. 220–229.

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

[Sorenson *et al.*, 1988] SORENSON, P.G., TREMBLAY, J.P. AND MCALLISTER, A.J. The Metaview System for Many Specification Environments. *IEEE Software*, v. 5, n. 2, p. 30-38, Mar. 1988.

[Stone & Doran, 1997] STONE, M.; DORAN, C. Sentence planning as description using tree adjoining grammar. In: CONFERENCE ON ASSOCIATION FOR COMPUTATIONAL LINGUISTICS, 35., 1997, Madrid, Espanha. *Proceedings*... Morristown, NJ, USA: Association for Computational Linguistics, 1997. p. 198-205.

[W3C, 2006] World Wide Web Consortium (W3C). eXtensible Markup Language (XML). 2006. Disponível em: http://www.w3.org/XML/ . Último acesso: 04 de Julho de 2006.

[Wahlster *et al.*, 1997] WAHLSTER, W., ANDRÉ, E., FINKLER, W., PROFITLICH, H., AND RIST, T. Plan-based integration of natural language and graphics generation. *Artificial Intelligence*, vol. 63, n. 1-2, p. 387-427, 1993. ISSN: 0004-3702.

[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.

Apêndice A – Bases de Conhecimento

Neste apêndice, serão mostradas algumas entradas das bases de conhecimento do protótipo do SpecNL. A seção A.1 apresenta algumas classes da ontologia; a seção A.2 exibe algumas entradas do Léxico; e, por fim, a seção A.3 apresenta alguns dos *case frames* utilizados. As entradas são apresentadas no formato XML.

A.1. Ontologia

```
<ontology>
  <class>
    <description>Generic Class</description>
    <name>Object</name>
    <code>object</code>
    <subclasses>
      <class>
        <description>Represents a generic value</description>
        <name>Value</name>
        <code>value</code>
        <subclasses>
          <class>
            <description>Represents a state value, e. g., "enabled", "ON", "high".</description>
            <name>State Value</name>
            <code>state_value</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a value used to fill a field, e. g., "some chars", "uppercase
characters".</description>
            <name>Field Value</name>
            <code>field_value</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a location value, e. g., "valid destination".</description>
            <name>Location Value</name>
            <code>location_value</code>
```

```
            <subclasses />
          </class>
          <class>
            <description>Represents a color, e. g., "blue", "different color".</description>
            <name>Color Value</name>
            <code>color_value</code>
            <subclasses />
          </class>
        </subclasses>
      </class>
      <class>
        <description>Represents a generic hardware</description>
        <name>Hardware</name>
        <code>hardware</code>
        <subclasses>
          <class>
            <description>Represents a phone</description>
            <name>Phone</name>
            <code>phone</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a card, e. g., "SIM CARD", "MMC CARD".</description>
            <name>Card</name>
            <code>card</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a memory, e. g., "phone memory", "message
memory".</description>
            <name>Memory</name>
            <code>memory</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a flex bit, e. g., "flex bit
DB_DL_MESSAGE".</description>
            <name>Flex Bit</name>
            <code>flex_bit</code>
            <subclasses />
          </class>
        </subclasses>
      </class>
      <class>
        <description>Represents an application, e. g., "phonebook", "message
center".</description>
        <name>Application</name>
        <code>application</code>
        <subclasses />
      </class>
      <class>
        <description>Represents an operation, e. g., "Clean up", "Master Clear".</description>
        <name>Operation</name>
        <code>operation</code>
```

```
        <subclasses />
      </class>
      <class>
        <description>Represents the user who executes actions.</description>
        <name>User</name>
        <code>user</code>
        <subclasses />
      </class>
      <class>
        <description>Represents an event, e. g., "call", "alarm".</description>
        <name>Event</name>
        <code>event</code>
        <subclasses />
      </class>
      <class>
        <description>Represents a generic screen. It can be a Form, a Menu, a List or a
Dialog.</description>
        <name>Screen</name>
        <code>screen</code>
        <subclasses>
          <class>
            <description>Represents a form screen. It is a set of field items, e. g., "message
composer form".</description>
            <name>Form</name>
            <code>form</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a menu screen. It's like a list but it don't have its elements
set changed. Its elements generally points to an application, e. g., "main menu", "message
menu".</description>
            <name>Menu</name>
            <code>menu</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a list of elements of the type, e. g., "contact list", "dialed
calls".</description>
            <name>List</name>
            <code>list</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a dialog, e. g., "new delivery report dialog".</description>
            <name>Dialog</name>
            <code>dialog</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a transient, e. g., "new delivery transient".</description>
            <name>Transient</name>
            <code>transient</code>
            <subclasses />
          </class>
```

```
        </subclasses>
      </class>
      <class>
        <description>Represents a generic item.</description>
        <name>Item</name>
        <code>item</code>
        <subclasses>
          <class>
            <description>Represents a item of a Menu.</description>
            <name>Menu Item</name>
            <code>menu_item</code>
            <subclasses />
          </class>
          <class>
            <description>Represents an item whose value may vary, e. g., "status of sent
message".</description>
            <name>Variable Item</name>
            <code>variable_item</code>
            <subclasses />
          </class>
          <class>
            <description>Represents a field that can be filled.</description>
            <name>Field</name>
            <code>field</code>
            <subclasses />
          </class>
          <class>
            <description>Represents an item of a list.</description>
            <name>List Item</name>
            <code>list_item</code>
            <subclasses>
              <class>
                <description>Represents an item that can be sent.</description>
                <name>Sendable Item</name>
                <code>sendable_item</code>
                <subclasses />
              </class>
            </subclasses>
          </class>
          <class>
            <description>Represents a key. Soft keys, buttons, etc.</description>
            <name>Key</name>
            <code>key</code>
            <subclasses />
          </class>
        </subclasses>
      </class>
    </subclasses>
  </class>
</ontology>
```

A.2. Léxico

A seção A.2.1 apresenta alguns exemplos de nomes. A seção A.2.2 exibe os principais modificadores das descrições de casos de teste. Por fim, a seção A.2.3 exibe os verbos mais freqüentes nas descrições de saída..

A.2.1. Nomes

```
<lexicon>
    <noun>
        <term>full</term>
        <plural/>
        <model>FULL_STATE_VALUE</model>
        <class>state_value</class>
    </noun>
    <noun>
        <term>Clean Up request dialog</term>
        <plural/>
        <model>CLEAN_UP_REQUEST</model>
        <class>dialog</class>
    </noun>
    <noun>
        <term>clean up</term>
        <plural/>
        <model>CLEAN_UP</model>
        <class>operation</class>
    </noun>
    <noun>
        <term>call</term>
        <plural/>
        <model>CALL</model>
        <class>event</class>
    </noun>
    <noun>
        <term>read key</term>
        <plural/>
        <model>READ_KEY</model>
        <class>key</class>
    </noun>
    <noun>
        <term>exit key</term>
        <plural/>
        <model>EXIT_KEY</model>
        <class>key</class>
    </noun>
    <noun>
        <term>csm menu list</term>
        <plural/>
        <model>CSM_MENU_LIST</model>
```

```
        <class>menu</class>
    </noun>
    <noun>
        <term>inbox message</term>
        <plural/>
        <model>INBOX_MESSAGE</model>
        <class>sendable_item</class>
    </noun>
    <noun>
        <term>alarm</term>
        <plural/>
        <model>ALARM</model>
        <class>event</class>
    </noun>
    <noun>
        <term>phone</term>
        <plural/>
        <model>PHONE</model>
        <class>phone</class>
    </noun>
    <noun>
        <term>character</term>
        <plural/>
        <model>CHARACTER</model>
        <class>field_value</class>
    </noun>
    <noun>
        <term>menu key</term>
        <plural/>
        <class>key</class>
        <model>MENU_KEY</model>
    </noun>
    <noun>
        <term>insert option</term>
        <plural/>
        <class>menu_item</class>
        <model>INSERT_OPTION</model>
    </noun>
    <noun>
        <term>contact info option</term>
        <plural/>
        <class>menu_item</class>
        <model>CONTACT_INFO_OPTION</model>
    </noun>
    <noun>
        <term>phonebook option</term>
        <plural/>
        <class>menu_item</class>
        <model>PHONEBOOK_OPTION</model>
    </noun>
    <noun>
        <term>done softkey</term>
        <plural/>
        <class>key</class>
```

```
        <model>DONE_SOFTKEY</model>
    </noun>
    <noun>
        <term>screen</term>
        <plural/>
        <class>screen</class>
        <model>GENERAL_SCREEN</model>
    </noun>
    <noun>
        <term>word</term>
        <plural/>
        <class>field_value</class>
        <model>WORD_VALUE</model>
    </noun>
    <noun>
        <term>send to softkey</term>
        <plural/>
        <class>key</class>
        <model>SEND_TO_SOFTKEY</model>
    </noun>
    <noun>
        <term>new number option</term>
        <plural/>
        <class>menu_item</class>
        <model>NEW_NUMBER_OPTION</model>
    </noun>
    <noun>
        <term>cs key</term>
        <plural/>
        <class>key</class>
        <model>CENTER_SOFTKEY</model>
    </noun>
    <noun>
        <term>back softkey</term>
        <plural/>
        <class>key</class>
        <model>BACK_SOFTKEY</model>
    </noun>
    <noun>
        <term>ok softkey</term>
        <plural/>
        <class>key</class>
        <model>OK_SOFTKEY</model>
    </noun>
    <noun>
        <term>message list</term>
        <plural/>
        <class>list</class>
        <model>MESSAGE_LIST</model>
    </noun>

    <noun>
        <term>idle screen</term>
        <plural/>
```

```
        <class>screen</class>
        <model>IDLE_SCREEN</model>
    </noun>
    <noun>
        <term>send message option</term>
        <plural/>
        <class>menu_item</class>
        <model>SEND_MESSAGE_OPTION</model>
    </noun>
    <noun>
        <term>high</term>
        <plural/>
        <class>state_value</class>
        <model>HIGH_VALUE</model>
    </noun>
    <noun>
        <term>priority value</term>
        <plural/>
        <class>variable_item</class>
        <model>PRIORITY_VALUE_VARIABLE</model>
    </noun>
    <noun>
        <term>normal</term>
        <plural/>
        <class>state_value</class>
        <model>NORMAL_VALUE</model>
    </noun>
    <noun>
        <term>low</term>
        <plural/>
        <class>state_value</class>
        <model>LOW_VALUE</model>
    </noun>
    <noun>
        <term>urgent</term>
        <plural/>
        <class>state_value</class>
        <model>URGENT_VALUE</model>
    </noun>
    <noun>
        <term>priority field</term>
        <plural/>
        <class>field</class>
        <model>PRIORITY_FIELD</model>
    </noun>
    <noun>
        <term>inbox folder</term>
        <plural/>
        <class>list</class>
        <model>INBOX_FOLDER</model>
    </noun>
    <noun>
        <term>message inbox</term>
        <plural/>
```

```
            <class>list</class>
            <model>INBOX_FOLDER</model>
        </noun>
        <noun>
            <term>call back field</term>
            <plural/>
            <class>field</class>
            <model>CALL_BACK_FIELD</model>
        </noun>
        <noun>
            <term>contact</term>
            <plural/>
            <class>list_item</class>
            <model>CONTACT_ITEM</model>
        </noun>
        <noun>
            <term>drafts folder</term>
            <plural/>
            <class>list</class>
            <model>DRAFTS_FOLDER</model>
        </noun>
        <noun>
            <term>outbox folder</term>
            <plural/>
            <class>list</class>
            <model>OUTBOX_FOLDER</model>
        </noun>
        <noun>
            <term>contact list</term>
            <plural/>
            <class>list</class>
            <model>CONTACT_LIST</model>
        </noun>
        <noun>
            <term>empty</term>
            <plural/>
            <class>state_value</class>
            <model>EMPTY_STATE</model>
        </noun>
        <noun>
            <term>available</term>
            <plural/>
            <class>state_value</class>
            <model>AVAILABLE_VALUE</model>
        </noun>
        <noun>
            <term>drafts screen</term>
            <plural/>
            <class>list</class>
            <model>DRAFTS_FOLDER</model>
        </noun>
</lexicon>
```

A.2.2. Modificadores

```
<lexicon>
    <modifier>
        <term>some</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>SOME</model>
    </modifier>
    <modifier>
        <term>next</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>NEXT</model>
    </modifier>
    <modifier>
        <term>at most <int/></term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>AT_MOST.Int</model>
    </modifier>
    <modifier>
        <term>previous</term>
        <position>before</position>
        <precedence>1</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>PREVIOUS</model>
    </modifier>
    <modifier>
        <term>any</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>ANY</model>
    </modifier>
    <modifier>
        <term>right</term>
        <position>before</position>
        <precedence>1</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>RIGHT</model>
    </modifier>
    <modifier>
        <term>separated by space</term>
```

```
        <position>after</position>
        <precedence>1</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>SEPARATED_BY_SPACE</model>
    </modifier>
    <modifier>
        <term><int/></term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>INTEGER.Int</model>
    </modifier>
    <modifier>
        <term>valid</term>
        <position>before</position>
        <precedence>1</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>VALID</model>
    </modifier>
    <modifier>
        <term>blank</term>
        <position>before</position>
        <precedence>1</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>BLANK</model>
    </modifier>
    <modifier>
        <term>created</term>
        <position>both</position>
        <precedence>1</precedence>
        <numberinflection>plural</numberinflection>
        <article>yes</article>
        <model>CREATED</model>
    </modifier>
    <modifier>
        <term>all</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>ALL</model>
    </modifier>
    <modifier>
        <term>new</term>
        <position>before</position>
        <precedence>1</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>NEW</model>
    </modifier>
```

```
<modifier>
    <term>predefined</term>
    <position>before</position>
    <precedence>1</precedence>
    <numberinflection>singular</numberinflection>
    <article>yes</article>
    <model>PREDEFINED</model>
</modifier>
<modifier>
    <term>only <int/></term>
    <position>before</position>
    <precedence>0</precedence>
    <numberinflection>singular</numberinflection>
    <article>no</article>
    <model>ONLY_INT.Int</model>
</modifier>
<modifier>
    <term><int/> or more</term>
    <position>before</position>
    <precedence>0</precedence>
    <numberinflection>plural</numberinflection>
    <article>no</article>
    <model>NUM_OR_MORE.Int</model>
</modifier>
<modifier>
    <term>invalid</term>
    <position>before</position>
    <precedence>1</precedence>
    <numberinflection>singular</numberinflection>
    <article>yes</article>
    <model>INVALID</model>
</modifier>
<modifier>
    <term>original</term>
    <position>before</position>
    <precedence>0</precedence>
    <numberinflection>singular</numberinflection>
    <article>yes</article>
    <model>ORIGINAL</model>
</modifier>
<modifier>
    <term>saved</term>
    <position>both</position>
    <precedence>1</precedence>
    <numberinflection>singular</numberinflection>
    <article>yes</article>
    <model>SAVED</model>
</modifier>
<modifier>
    <term>stored</term>
    <position>both</position>
    <precedence>1</precedence>
    <numberinflection>singular</numberinflection>
    <article>yes</article>
```

```
        <model>STORED</model>
    </modifier>
    <modifier>
        <term>at least <int/></term>
        <position>before</position>
        <precedence>last</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>AT_LEAST.Int</model>
    </modifier>
    <modifier>
        <term>class 2</term>
        <position>before</position>
        <precedence>last</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>CLASS_2</model>
    </modifier>
    <modifier>
        <term>non class 2</term>
        <position>before</position>
        <precedence>last</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>NON_CLASS_2</model>
    </modifier>

    <modifier>
        <term>protected</term>
        <position>before</position>
        <precedence>2</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>PROTECTED_MODIFIER</model>
    </modifier>
    <modifier>
        <term>current</term>
        <position>before</position>
        <precedence>2</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>CURRENT_MODIFIER</model>
    </modifier>
    <modifier>
        <term>unread</term>
        <position>before</position>
        <precedence>2</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>UNREAD_MODIFIER</model>
    </modifier>
    <modifier>
        <term>unlocked</term>
        <position>before</position>
```

```
        <precedence>2</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>UNLOCKED_MODIFIER</model>
    </modifier>
    <modifier>
        <term>locked</term>
        <position>before</position>
        <precedence>2</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>LOCKED_MODIFIER</model>
    </modifier>
    <modifier>
        <term>related</term>
        <position>before</position>
        <precedence>2</precedence>
        <numberinflection>singular</numberinflection>
        <article>yes</article>
        <model>RELATED</model>
    </modifier>
    <modifier>
        <term>not enough</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>NOT_ENOUGH</model>
    </modifier>
    <modifier>
        <term>enough</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>ENOUGH</model>
    </modifier>
    <modifier>
        <term>old</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>OLD_MODIFIER</model>
    </modifier>
    <modifier>
        <term>several</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>SEVERAL_MODIFIER</model>
    </modifier>
    <modifier>
```

```
        <term>before <noun/></term>
        <position>after</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>BEFORE.Item</model>
    </modifier>
    <modifier>
        <term>after <noun/></term>
        <position>after</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>AFTER.Screen</model>
    </modifier>
    <modifier>
        <term>selected</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>SELECTED_MODIFIER</model>
    </modifier>
</lexicon>
```

A.2.3. Verbos

```
<lexicon>
    <verb>
        <term>contain</term>
        <past>contained</past>
        <participle>contained</participle>
        <gerund>containing</gerund>
        <thirdperson>contains</thirdperson>
    </verb>
    <verb>
        <term>set</term>
        <past>set</past>
        <participle>set</participle>
        <gerund>setting</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>have</term>
        <past>had</past>
        <participle>had</participle>
        <gerund>having</gerund>
      <thirdperson>has</thirdperson>
    </verb>
    <verb>
        <term>check</term>
        <past>checked</past>
```

```
        <participle>checked</participle>
        <gerund>checking</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>uncheck</term>
        <past>unchecked</past>
        <participle>unchecked</participle>
        <gerund>unchecking</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>change</term>
        <past>changed</past>
        <participle>changed</participle>
        <gerund>changing</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>edit</term>
        <past>edited</past>
        <participle>edited</participle>
        <gerund>editing</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>fill</term>
        <past>filled</past>
        <participle>filled</participle>
        <gerund>filling</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>select</term>
        <past>selected</past>
        <participle>selected</participle>
        <gerund>selecting</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>delete</term>
        <past>deleted</past>
        <participle>deleted</participle>
        <gerund>deleting</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>remove</term>
        <past>removed</past>
        <participle>removed</participle>
        <gerund>removing</gerund>
        <thirdperson/>
    </verb>
    <verb>
```

```
        <term>verify</term>
        <past>verified</past>
        <participle>verified</participle>
        <gerund>verifying</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>confirm</term>
        <past>confirmed</past>
        <participle>confirmed</participle>
        <gerund>confirming</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>start</term>
        <past>started</past>
        <participle>started</participle>
        <gerund>starting</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>create</term>
        <past>created</past>
        <participle>created</participle>
        <gerund>creating</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>send</term>
        <past>sent</past>
        <participle>sent</participle>
        <gerund>sending</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>go</term>
        <past>went</past>
        <participle>gone</participle>
        <gerund>going</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>press</term>
        <past>pressed</past>
        <participle>pressed</participle>
        <gerund>pressing</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>cancel</term>
        <past>cancelled</past>
        <participle>cancelled</participle>
        <gerund>cancelling</gerund>
        <thirdperson/>
```

```
    </verb>
    <verb>
        <term>display</term>
        <past>displayed</past>
        <participle>displayed</participle>
        <gerund>displaying</gerund>
        <thirdperson/>
    </verb>
    <verb>
        <term>contain</term>
        <past>contained</past>
        <participle>contained</participle>
        <gerund>containing</gerund>
        <thirdperson/>
    </verb>
</lexicon>
```

A.3. Case Frames

```
<grammar>
  <frame>
    <description>Set the value of an item. Example: Set the Fix Dialing to on</description>
    <name>SetItem</name>
    <verblist>
      <verb>set</verb>
      <verb>check</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">to-value</role>
    </roles>
  </frame>
  <frame>
    <description>Uncheck an item. Example: Uncheck the field</description>
    <name>UncheckItem</name>
    <verblist>
      <verb>uncheck</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
    </roles>
  </frame>
  <frame>
    <description>Change an item from value1 to value2. Example: Change the priority value
from low to normal</description>
    <name>ChangeItem</name>
    <verblist>
      <verb>change</verb>
    </verblist>
```

```
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">from-value</role>
      <role mandatory="false">to-value</role>
    </roles>
  </frame>
  <frame>
    <description>Edit an item with value. Example: Edit the message field with</description>
    <name>EditField</name>
    <verblist>
      <verb>edit</verb>
      <verb>fill</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">to-value</role>
    </roles>
  </frame>
  <frame>
    <description>Select an item from location. Example: Select the send message option from
menu</description>
    <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>
  <frame>
    <description>Unselect an item from location. Example: Unselect the send message option
from menu</description>
    <name>UnselectItem</name>
    <verblist>
      <verb>unselect</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">from-loc</role>
    </roles>
  </frame>
  <frame>
    <description>Highlight an item from location. Example: Highlight the Subject item from the
message</description>
    <name>HighlightItem</name>
    <verblist>
      <verb>highlight</verb>
    </verblist>
```

```
      <roles>
        <role mandatory="True">agent</role>
        <role mandatory="True">theme</role>
        <role mandatory="false">from-loc</role>
      </roles>
    </frame>
    <frame>
      <description>Delete/remove an item from location. Example: Delete the contents of the
Subject field</description>
      <name>DeleteItem</name>
      <verblist>
        <verb>delete</verb>
      </verblist>

      <roles>
        <role mandatory="True">agent</role>
        <role mandatory="True">theme</role>
        <role mandatory="false">from-loc</role>
      </roles>
    </frame>
    <frame>
      <description>Delete/remove an item from location.</description>
      <name>RemoveItem</name>
      <verblist>
        <verb>remove</verb>
      </verblist>
      <roles>
        <role mandatory="True">agent</role>
        <role mandatory="True">theme</role>
        <role mandatory="false">from-loc</role>
      </roles>
    </frame>
    <frame>
      <description>Check if an item value is at value. Example: Check the message
status</description>
      <name>VerifyItem</name>
      <verblist>
        <verb>verify</verb>
        <verb>check</verb>
      </verblist>
      <roles>
        <role mandatory="True">agent</role>
        <role mandatory="True">theme</role>
        <role mandatory="false">at-value</role>
      </roles>
    </frame>
    <frame>
      <description>Start an item. Example: Start Message Aplication</description>
      <name>StartItem</name>
      <verblist>
        <verb>start</verb>
        <verb>run</verb>
        <verb>execute</verb>
        <verb>perform</verb>
```

```
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
    </roles>
  </frame>
  <frame>
    <description>Save an item to location. Example: Save the message to Drafts
Folder</description>
    <name>SaveItem</name>
    <verblist>
      <verb>save</verb>
      <verb>store</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">to-loc</role>
    </roles>
  </frame>
  <frame>
    <description>Enter data in a field. Example: Enter any word into editor</description>
    <name>EnterData</name>
    <verblist>
      <verb>enter</verb>
      <verb>write</verb>
      <verb>type</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">to-loc</role>
    </roles>
  </frame>
  <frame>
    <description>Access something. Example: Access the phone number field</description>
    <name>GotoItem</name>
    <verblist>
      <verb>go</verb>
      <verb>access</verb>
      <verb>scroll</verb>
      <verb>navigate</verb>
      <verb>enter</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">to-loc</role>
    </roles>
  </frame>
  <frame>
    <description>Create an item. Example: Create a message</description>
    <name>CreateItem</name>
    <verblist>
      <verb>create</verb>
```

```
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">to-loc</role>
    </roles>
  </frame>
  <frame>
    <description>Unlock an item. Example: Unlock the message</description>
    <name>UnlockItem</name>
    <verblist>
      <verb>unlock</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
    </roles>
  </frame>
  <frame>
    <description>Send something to someplace. Example: Send the message to a valid
address</description>
    <name>SendItem</name>
    <verblist>
      <verb>send</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="false">to-loc</role>
    </roles>
  </frame>
  <frame>
    <description>Lock something. Example: Lock the voice mms msg.</description>

    <name>LockItem</name>

    <verblist>
      <verb>lock</verb>
    </verblist>

    <roles>
      <role mandatory="True">agent</role>

      <role mandatory="True">theme</role>
    </roles>
  </frame>

  <frame>
    <description>Specifies an item state. Example: Phone is in idle state</description>
    <name>IsState</name>
    <verblist>
      <verb>be</verb>
    </verblist>
    <roles>
```

```
        <role mandatory="True">theme</role>
        <role mandatory="True">at-value</role>
      </roles>
    </frame>
    <frame>
      <description>Specifies the phone screen (location). Example: Phone is in Message Inbox
Screen</description>
      <name>IsLocation</name>
      <verblist>
        <verb>be</verb>
      </verblist>
      <roles>
        <role mandatory="True">theme</role>
        <role mandatory="True">at-loc</role>
      </roles>
    </frame>
    <frame>
      <description>Something exists in some place. Example: Exists a message in the inbox
folder.</description>
      <name>ThereIsItem</name>
      <verblist>
        <verb>there is</verb>
        <verb>exist</verb>
      </verblist>
      <roles>
        <role mandatory="False">at-loc</role>
        <role mandatory="True">theme</role>
      </roles>
    </frame>
    <frame>
      <description>Read something. Example: Read the message.</description>
      <name>ReadItem</name>
      <verblist>
        <verb>read</verb>
        <verb>view</verb>
      </verblist>
      <roles>
        <role mandatory="True">agent</role>
        <role mandatory="True">theme</role>
        <role mandatory="False">from-loc</role>
      </roles>
    </frame>
    <frame>
      <description>Press something. Example: Press SELECT softkey.</description>
      <name>PressKey</name>
      <verblist>
        <verb>press</verb>
      </verblist>
      <roles>
        <role mandatory="True">agent</role>
        <role mandatory="True">theme</role>
      </roles>
    </frame>
    <frame>
```

```
    <description>An item contains/has something. Example: To: field contains only
numbers.</description>
    <name>HaveItem</name>
    <verblist>
      <verb>have</verb>
      <verb>contain</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
    </roles>
  </frame>
  <frame>
    <description>Display item</description>
    <name>DisplayItem</name>
    <verblist>
      <verb>display</verb>
      <verb>show</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
      <role mandatory="False">at-loc</role>
    </roles>
  </frame>
  <frame>
    <description>Cancel item</description>
    <name>CancelItem</name>
    <verblist>
      <verb>cancel</verb>
    </verblist>
    <roles>
      <role mandatory="True">agent</role>
      <role mandatory="True">theme</role>
    </roles>
  </frame>
</grammar>
```