UNIVERSIDADE
DE VIGO

ESCOLA SUPERIOR DE ENXEÑERÍA INFORMÁTICA

Memoria do Proxecto Fin de Carreira que presenta

D. Vanesa Vidal Castro

para a obtención do Título de **Enxeñeiro en Informática**

Construcción de un analizador morfosintáctico robusto.

Abril, 2009

Proxecto Fin de Carreira Nº: ENI-268

Director/a: Manuel Vilares Ferro
Codirector/a: Juan Otero Pombo
Área de coñecemento:
Ciencias da Computación e Intelixencia Artificial
Departamento: Informática

"Vive como si fueras a morir mañana. Aprende como si fueras a vivir para siempre."
Mahatma Gandhi

A mi familia

Agradecimientos

Este es el momento para mostrar mi más profundo agradecimiento a todas aquellas personas que me han ayudado y facilitado la realización de este proyecto, sea de la forma que sea.

En primer lugar, a mis dos directores, Manuel y Juan, ambos han desempeñado una excelente labor como guías y consejeros desde el principio hasta el final del proyecto. A Manuel, le quiero agradecer especialmente la oportunidad brindada y la confianza depositada en mi. Y a Juan, por su gran implicación y comprensión prestada a lo largo de todo el proyecto, especialmente en los inicios; sin su ayuda este trabajo no habría sido posible.

A los compañeros del laboratorio: Adrián, Nieves, Fran, Mila, Mario y Víctor. Por ofrecerme su ayuda prestada siempre que la he necesitado y por esos buenos momentos pasados que hacen más agradable el trabajo diario.

A mi familia, por estar siempre ahí, especialmente a mis padres, por el sacrificio que han realizado durante todos estos años, por buscar siempre la mejor formación para mí. Y a mi hermano Roberto, que siempre me ayudará en lo que esté en su mano.

Finalmente, a todas aquellas personas que han compartido conmigo buena parte de mi vida: Bea, Estela, Leti, María, Olalla, Pili, Patri, Rosana, ... y a las que quizás no agradezco demasiado a menudo su apoyo y cariño. A mis compañeras de piso: Diana, Lidia y Luz por su grata compañía y a todos aquellos que se alegran por mí.

A todos vosotros, G R A C I A S.

Índice general

I Memoria **11**

1. Introducción **13**
1.1. Identificación del proyecto 13
1.2. Organización de la documentación 14
1.2.1. Memoria 14
1.2.2. Manual técnico 14
1.2.3. Manual de usuario 15
1.3. Motivación 15
1.4. Contexto del proyecto 16
1.4.1. Etiquetación de palabras 16
1.4.2. Preprocesador morfológico 25
1.4.3. Modelización del diccionario 26
1.4.4. Corrector ortográfico 30
1.5. Objetivos del proyecto 36
1.5.1. Integración de varias herramientas 36
1.5.2. Recodificación del corrector 38
1.5.3. Corrección a errores multi-palabras 38

2. Tecnologías y herramientas empleadas **41**
2.1. Tecnologías empleadas 41
2.1.1. C 41
2.1.2. XML 42
2.1.3. SAX 42
2.1.4. LaTeX 43
2.2. Herramientas utilizadas 43
2.2.1. NetBeans 43
2.2.2. Visual Paradigm 44
2.2.3. Kile 44
2.2.4. Planner 44
2.2.5. Kubuntu 44

3. Planificación y presupuesto **47**
3.1. Planificación del proyecto 47
3.1.1. Planificación inicial 47
3.1.2. Planificación real 49
3.2. Presupuesto del proyecto 50
3.2.1. Presupuesto del proyecto 50

4. Conclusiones y posibles ampliaciones **53**
4.1. Conclusiones 53
4.2. Posibles ampliaciones 54

II Manual Técnico **55**

5. Análisis **57**
5.1. Sistema implementado 57
5.2. Estructuras de datos utilizadas por el *preprocesador* 58
5.3. Diagrama de casos de uso global 60
5.4. Caso de uso: Aplicar corrector 60
5.4.1. Aplicar corrección: Buscar corrección global 65
5.4.2. Aplicar corrección: Buscar corrección regional 68
5.4.3. Aplicar corrección: Enlazar correcciones 71
5.4.4. Aplicar corrección: Revisar nuevas contracciones 75
5.4.5. Aplicar corrección: Revisar nuevas expresiones 79
5.5. Caso de uso: Configurar corrector 82
5.6. Diagrama de clases 84

6. Diseño **85**
6.1. Diagrama de clases 85
6.2. Diccionario de clases 86
6.2.1. Sistema corrector (s_checker) 87
6.2.2. Parser corrector 96
6.2.3. Preprocesador 96
6.2.4. Parser preprocesador 97
6.2.5. Corrector regional (Bureg) 97
6.2.6. Corrector global (Savary) 98
6.2.7. MrTagoo 98
6.2.8. Contractions 99
6.2.9. Expressions 99
6.2.10. NADFA 100
6.2.11. TOKEN 100

6.2.12. SENTENCE 100
6.2.13. E/S 100
6.3. Diagramas de secuencia para los casos de uso 102
6.3.1. Caso de uso: Aplicar Corrección 102

7. Implementación y pruebas 109
7.1. Implementación 109
7.1.1. Diagrama de componentes 109
7.1.2. Fichero de configuración 109
7.1.3. Detalles de implementación 111
7.2. Pruebas 112
7.2.1. Pruebas de unidad 112
7.2.2. Pruebas de integración 113

III Manual de Usuario 115

8. Instalación y ejecución 117
8.1. Requisitos 117
8.1.1. Requisitos hardware 117
8.1.2. Requisitos software 117
8.2. Compilación 118
8.2.1. Compilación de la librería del etiquetador MrTagoo 118
8.2.2. Instalación del preprocesador 119
8.3. Ejecución del sistema 120
8.3.1. Directorio de configuración 120
8.3.2. Opciones de ejecución 122
8.3.3. Resultados obtenidos 127

Parte I

Memoria

Capítulo 1

Introducción

1.1. Identificación del proyecto

Código del PFC: **ENI - 268**

Título del PFC: **Construcción de un analizador morfosintáctico robusto**

Alumno: **Vanesa Vidal Castro**

Director del proyecto: **Manuel Vilares Ferro**

Co-director del proyecto: **Juan Otero Pombo**

Departamento: **Informática**.

Área: **Ciencias de la Computación e Inteligencia Artificial**.

Titulación: **Ingeniería Informática**.

1.2. Organización de la documentación

El CD-ROM, con toda la información del proyecto, se organiza en dos carpetas. Una con el *software* necesario para el funcionamiento de la aplicación, salvo el código del preprocesador, que por motivos de licencia no ha podido ser incluído.

En la otra carpeta, se incluye toda la documentación generada a lo largo del ciclo de vida del proyecto. Ésta se organizará en las siguientes partes: memoria, manual técnico y manual de usuario. Para cada una de ellas detallaremos sus contenidos.

1.2.1. Memoria

La memoria está formada por los siguientes capítulos:

- *Introducción*: Describe el origen del proyecto, el contexto en el que se sitúa y también los objetivos a alcanzar.
- *Tecnologías y herramientas empleadas en el proyecto*: Se describirán de forma breve las tecnologías y herramientas *software* que se han empleado para llevar a cabo el proyecto.
- *Planificación y presupuesto*: Incluye la planificación temporal inicial y la real, así como la explicación para los desajustes encontrados entre ambas. Además, se detalla el presupuesto necesario para la realización del proyecto. Se tendrán en cuenta los recursos físicos y los recursos humanos empleados.
- *Conclusiones y posibles ampliaciones*: Aquí, se reflexiona sobre el resultado final del proyecto, los problemas encontrados durante todo el desarrollo y, también, se incluyen una serie de reflexiones a nivel personal. Además, también se expondrán posibles ampliaciones a acometer para mejorar el proyecto.

1.2.2. Manual técnico

El manual técnico consta de los siguientes capítulos:

- *Análisis*: Contiene la especificación de los requisitos y se muestran los diagramas de secuencia, colaboración y actividades correspondientes a dichas especificaciones.
- *Diseño*: Se refinan los diagramas incluídos en el análisis para obtener un mayor nivel de especificación del sistema.
- *Implementación y pruebas*: Aquí se incluyen algunos detalles de implementación considerados relevantes y las pruebas para validar el correcto funcionamiento del sistema.

1.2.3. Manual de usuario

Este documento es una guía detallada de los pasos a seguir para realizar la instalación del sistema completo. También se decriben las diferentes opciones de ejecución del sistema y la interpretación de los resultados obtenidos.

1.3. Motivación

La era digital ha propiciado la existencia de grandes volúmenes de información. A finales del 2008, se han contabilizado 186 millones de páginas *web* en Internet. Esta información, se almacena generalmente con fin determinado, por tanto debería ser procesada o gestionada mediante algún tipo de herramienta. Para alcanzar estos objetivos se emplean diferentes técnicas y/o algoritmos centrados en las siguientes características: precisión y eficacia.

Existen diversos acercamientos posibles en orden a resolver este problema. En nuestro caso, hemos optado por un acercamiento basado en *procesamiento del lenguaje natural* (PLN).

El PLN es una subrama de la Inteligencia Artificial y de la Lingüística Computacional que se ocupa de diseñar mecanismos eficaces computacionalmente para la comunicación entre personas y máquinas por medio de lenguajes de comunicación humana. Esto es, se trata de crear herramientas que puedan analizar y generar estructuras de comunicación que los humanos utilizan habitualmente, de manera que el usuario pueda llegar a interaccionar con el ordenador de la misma forma que lo haría con un semejante.

El PLN se puede aplicar a campos tan diversos como son la traducción automática, la recuperación y organización de la información, la extracción de información y resúmenes, la corrección ortográfica de textos, el análisis sintáctico de las oraciones y el reconocimiento de voz entre otros.

El estudio de los lenguajes naturales se organiza, tradicionalmente, en cinco niveles de complejidad: fonética, morfología, sintaxis, semántica y pragmática. Este proyecto se centrará en los niveles morfológico y sintáctico del lenguaje:

- *Nivel Morfológico*: Trata de la composición de cada palabra a partir de unidades de significado más pequeñas, denominadas morfemas. Dentro de ellos incluimos prefijos, sufijos, raíces y flexiones.

- *Nivel Sintáctico*: Fija el papel que cada palabra juega dentro de una oración en base a una estructura gramatical.

Algunas de las tareas relacionadas con el PLN y que abordaremos en este proyecto son la corrección ortográfica, la segmentación y la etiquetación de texto.

La idea de este trabajo es combinar una serie de herramientas relacionadas con la etiquetación del lenguaje natural, con varias técnicas de corrección ortográfica para construír, finalemente, un corrector ortográfico contextual.

1.4. Contexto del proyecto

En este capítulo vamos a introducir el entorno en el que se va a desarrollar nuestro trabajo, antes de especificar los objetivos finales del proyecto.

1.4.1. Etiquetación de palabras

El objetivo de la *etiquetación* [1] del lenguaje natural es la asignación a cada palabra de una marca (etiqueta [2]) y que representa la información morfosintáctica asociada al término. Por ejemplo, su categoría (sustantivo, adjetivo, verbo, preposición...) o, cuando corresponda, su género; número o la forma verbal en la que se encuentra. Así, una etiqueta para el sustantivo *mesa* podría ser `Scfs`, que podría significar *sustantivo común femenino singular*.

Para realizar la etiquetación es necesario el uso de un diccionario que contenga todas las unidades léxicas del lenguaje y sus posibles etiquetas [3]. Los lexicones suelen alcanzar un tamaño considerable, por lo que la estructura de datos utilizada para su implementación y los algoritmos de búsqueda e inserción adquieren una gran relevancia. Con este fin, utilizaremos una estructura eficiente y compacta basada en autómatas para modelar el diccionario y que fue propuesta por Daciuk [2] y Graña [7].

Aunque *a priori* pudiese parecer una tarea sencilla, la etiquetación es un proceso sumamente complejo dado que los lenguajes naturales son inherentemente ambiguos. Es el caso, muy común, de las palabras homógrafas, dónde un mismo término posee varias etiquetas candidatas en función del contexto de la frase y de su posible función en la misma. Un ejemplo de palabra homógrafa sería *casa*, para la que existirían tres potenciales etiquetas diferentes:

1. Sustantivo común femenino singular.

2. Tercera persona del singular del presente de indicativo del verbo casar.

3. Segunda persona del imperativo del verbo casar.

Sin embargo, la base de la comunicación humana es la eliminación de ambigüedades en función del contexto del discurso, por eso nuestro objetivo final

[1] Se conoce en inglés como *parts of speech tagging*
[2] Denominada *tag* en inglés
[3] También conocido como lexicón.

es que cada palabra tenga una única etiqueta que indique, de forma inequívoca, su función léxica en el texto. Existen dos posibles fuentes de información para ayudar a tomar la decisión más acertada en cada caso:

- El contexto en el que aparece cada palabra. Para etiquetar cada palabra se tiene en cuenta las etiquetas de las palabras contiguas. En caso de que éstas sean, a su vez, ambiguas pueden considerarse otro tipo de técnicas. Por ejemplo, podemos elegir las secuencias más comunes para cada lengua. Así, en español la secuencia *artículo-sustantivo-adjetivo* es habitual, pero otras como *artículo-preposición-verbo* no lo son. Sin embargo, técnicas de este tipo, por sí solas, no ofrecen los resultados esperados, dada la gran cantidad de combinaciones posibles.

- La consideración de la frecuencia de aparición de cada etiqueta para una palabra determinada. Aunque pueda parecer un método simplista, se ha demostrado que la elección de la etiqueta más probable ha alcanzado un índice de acierto del 90 % [1].

Actualmente, la mayor parte de los etiquetadores obtienen un modelo de desambiguación a partir de un *corpus* [4] de entrenamiento. En consecuencia, el rendimiento de los etiquetadores se ve afectado por toda una serie de factores de entre los que podemos considerar,por ejemplo:

- El *tamaño del corpus* de entrenamiento disponible. En general, cuánto mayor sea el texto, mejores resultados.

- El *conjunto de etiquetas* [5]. Cuantas más etiquetas formen el conjunto, más complicado será el aprendizaje de patrones o secuencias de etiquetas.

- El *ámbito del texto* a etiquetar. El rendimiento será mejor para aquellos textos cuyo ámbito de conocimiento coincida con el del *corpus* de entrenamiento.

- La presencia de *palabras desconocidas*. Si el texto a etiquetar pertenece al mismo ámbito de conocimiento que áquellos que se han utilizado en la fase de entrenamiento, es de esperar que el número de palabras sin etiqueta sea menor.

Podemos diferenciar, asimismo, varios tipos de etiquetadores en función de la información que empleen para realizar la desambiguación morfosintáctica:

[4] Conjunto de textos científicos o literarios, etc., que pueden servir de base para una investigación.

[5] Se conoce como *tagset* en inglés.

- *Etiquetadores contextuales*: Utilizan el conocimiento lingüístico para la construcción de un modelo del lenguaje [8]. Se basan en el uso de reglas, que generalmente se construyen manualmente.

- *Etiquetadores estadísticos*: Los modelos del lenguaje se generan automáticamente mediante el uso de inferencia estadística, las probabilidades se calculan a partir de los datos [14]. Este tipo de etiquetador es independiente de la lengua utilizada.

Modelos de Markov

Los etiquetadores estadísticos están frecuentemente basados en *Modelos de Markov* [10] y pueden considerar técnicas adicionales, como el algoritmo de Viterbi [16] para encontrar la secuencia de etiquetas más probable. A continuación, se describirá de forma breve ambos ya que este es el modelo implementado por el etiquetador MrTagoo [7],[6] y [5], con el que vamos a trabajar.

Esta técnica fue desarrollada por A. Markov en 1913 para modelizar secuencias de palabras en ruso. En la actualidad, son ampliamente utilizados como herramientas estadísticas de propósito general. Formalizan la etiquetación como un proceso doblemente aleatorio y parametrizable. Los parámetros se pueden establecer de forma precisa durante el entrenamiento.

El sistema se puede modelar como un conjunto finito de estados. Transcurrido un intervalo discreto de tiempo el sistema cambia de un estado a otro de acuerdo con unas probabilidades de transición. Según la dependencia probabílistica de un estado con respecto al anterior, consideramos dos propiedades para las cadenas de Markov:

1. *Propiedad del horizonte limitado:* La probabilidad de un estado (etiqueta) no depende de todos los precedentes, sino sólo de un número finito de ellos. Se dice que una *cadena de Markov* es de *orden* n cuando utiliza n estados previos para predecir el siguiente estado.

2. *Propiedad del tiempo estacionario:* La probabilidad de transición entre dos estados i y j no depende del instante de tiempo en que nos encontremos, sólo dependerá de los propios estados.

Un proceso con las propiedades que acabamos de describir se denomina *Modelo de Markov Observable*, porque su salida es el conjunto de estados por los que pasa en cada instante de tiempo y cada estado se corresponde directamente con una observación.

Ejemplo 1.1. *Se puede utilizar un Modelo de Markov Observable para representar la evolución de la meteorología en una zona determinada. Para ello, asumimos que*

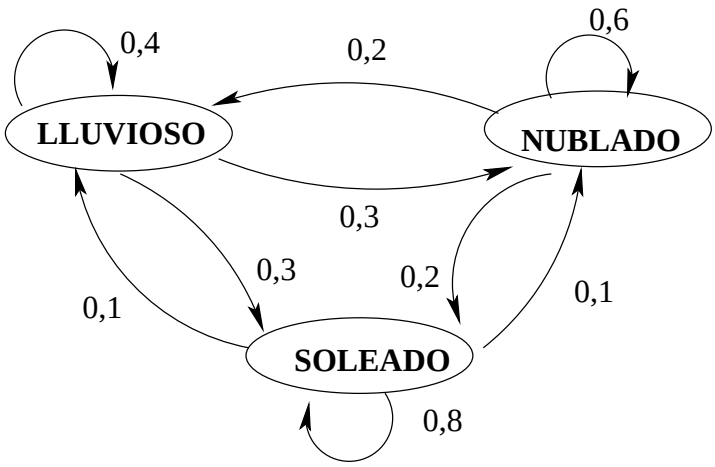

Figura 1.1: Modelo de Markov que representa la evolución del clima

se observa el clima una vez al día y que existen tres posibles observaciones: lluvioso, nublado y soleado. Además, supondremos que los tres estados tienen la misma probabilidad de ser estado inicial. La probabilidad de observar una determinada secuencia de estados se obtiene multiplicando las probabilidades que figuran en las aristas del grafo de la figura 1.1. Por ejemplo, si queremos saber la probabilidad de observar la secuencia `nublado-soleado-nublado-lluvioso`*, tendríamos que multiplicar la probabilidad de que* `nublado` *fuese estado inicial por la probabilidad de ir transitando de un estado a otro.*

$$P(nub, sol, nub, llu) = P(nub)P(sol|nub)P(nub|sol)P(llu|nub) =$$
$$= (\frac{1}{3}).(0,2).(0,1).(0,2) = 0,00133333$$

■

Sin embargo, estos modelos son demasiado restrictivos para la mayoría de problemas reales, ya que la salida de un estado no es aleatoria, sino que es siempre la misma. Por este motivo se hizo necesaria la extensión del concepto, para que la observación fuese una función probabilística del estado. Al modelo resultante se le denominó *Modelo de Markov Oculto* (MMO), éste es un modelo doblemente estocástico ya que uno de los procesos no es directamente observable, sino que

está oculto.

A continuación, se enumerarán los elementos que componen un MMO y se explicará la correspondencia de cada componente, aplicada al caso de la etiquetación:

- *Conjunto de estados del modelo ($\mathcal{Q}$):* se corresponde con el conjunto de las etiquetas.
- *Conjunto de sucesos observables (V):* está formado por las palabras del lexicón.
- *Distribución de probabilidad del estado inicial (π):* la probabilidad de que una etiqueta sea la primera en una frase.
- *Distribución de probabilidad de las transiciones entre estados (A):* la probabilidad de que después de una etiqueta se transite a otra.
- *Distribución de probabilidad de los sucesos observables para cada estado (B):* la probabilidad de que a partir de una determinada etiqueta se observe una palabra del lexicón. Por ejemplo, la probabilidad de que a partir de la etiqueta *preposición* se observe la palabra "*para*".

Los tres últimos parámetros representan probabilidades y por tanto deben ser estimados a partir de los datos observados. La estimación se puede realizar de forma supervisada, utilizando un corpus de entrenamiento etiquetado o no supervisada cuando el *corpus* no está etiquetado.

Ejemplo 1.2. *Para ilustrar el funcionamiento de un* MMO *se empleará un ejemplo con urnas y bolas. El modelo consta de un número N de urnas colocadas en una habitación aislada. Cada urna contiene un número grande de bolas y cada bola es de un solo color. Existe una persona dentro de la habitación encargada de sacar las bolas de la urna. Inicialmente, la persona escoge una urna aleatoriamente, saca una bola, grita su color en voz alta y devuelve la bola a la urna. A continuación, elige otra urna de acuerdo a un proceso aleatorio asociado a la primera y extrae otra bola ... y así un número determinado de veces. La secuencia de colores constituye la salida observable del modelo, quedando oculta para nosotros la secuencia de urnas que se ha seguido.*

El proceso se modeliza de forma que cada urna se corresponde con un estado del MMO*, la elección de la urna se realiza de acuerdo a una matriz de probabilidades de transición entre estados asociada al modelo, y para cada estado existe una función de probabilidad de los colores.*

Este ejemplo puede identificarse con el problema de la etiquetación de forma intuitiva. Las bolas representarían a las palabras y las urnas a las etiquetas que representan las palabras. Una misma palabra puede encontrarse en urnas distintas y puede estar varias veces en la misma urna.

■

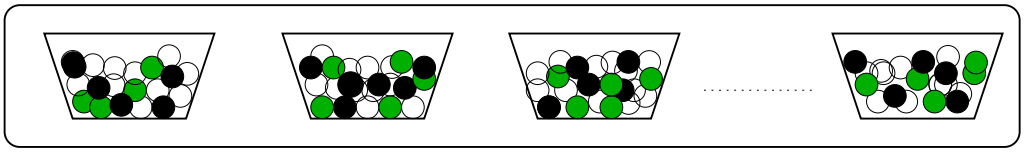

Figura 1.2: Representación de un MMO

El algoritmo de Viterbi

Uno de los problemas asociados a los MMOs es la elección de la secuencia de estados (etiquetas) más probable para las palabras de un texto. Para realizar esta elección se utiliza el *algoritmo de Viterbi* [16]. El objetivo es encontrar la secuencia de estados óptima que mejor explica la secuencia de observaciones. Formalmente, dado un modelo de Markov $\mu = (\mathcal{Q}, \mathcal{V}, \pi, \mathcal{A}, \mathcal{B})$, se trata de encontrar la secuencia $S = (q_1, q_2, \ldots, q_T)$ más probable a partir de una observación $O = (o_1, o_2, \ldots, o_T)$.

Consideremos la variable $\delta_t(i)$, que se define como

$$\delta_t(i) = \underset{q_1,q_2,\ldots,q_{t-1}}{max} P(q_1, q_2, \ldots, q_{t-1}, q_t = i, o_1, o_2, \ldots, o_t|\mu)$$

es decir, $\delta_t(i)$ almacena la probabilidad del mejor camino que termina en el estado i, teniendo en cuenta las t primeras observaciones. Se demuestra fácilmente que

$$\delta_{t+1}(j) = \left[\underset{1\leq i\leq N}{max}\ \delta_t(i)a_{ij}\right] b_j(o_{t+1})$$

Puesto que el objetivo es obtener la secuencia de estados más probable, a la vez que calculamos todos los $\delta_t(i)$ para todos los estados y todos los instantes de tiempo, debemos almacenar una traza que nos permita recordar cual fue el argumento que maximizó la ecuación. Para esto se utiliza la variable $\psi_t(j)$. A continuación se describen los cuatro pasos del algoritmo de Viterbi [16]:

1. Inicialización:
$$\delta_1(i) = \pi_i b_i(o_1), \qquad 1 \leq i \leq N$$

2. Recurrencia:
$$\delta_{t+1}(j) = \left[\underset{1\leq i\leq N}{max}\ \delta_t(i)a_{ij}\right] b_j(o_{t+1}), \qquad t = 1, 2, \ldots, T-1, \qquad 1 \leq j \leq N$$
$$\psi_{t+1}(j) = \underset{1\leq i\leq N}{\arg\max}\ \delta_t(i)a_{ij}, \qquad t = 1, 2, \ldots, T-1, \qquad 1 \leq j \leq N$$

3. Terminación:

$$q_T^* = \arg\max_{1 \leq i \leq N} \delta_T(i)$$

4. Construcción hacia atrás de la secuencia de estados:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \qquad t = T-1, T-2, \ldots, 1$$

En caso de producirse un empate durante el cálculo del mejor camino que termina en un estado ($\max_{1 \leq i \leq N} \delta_t(i) a_{ij}$), el algoritmo elige de forma aleatoria uno de entre los caminos con máxima probabilidad.

Para la etiquetación de palabras, los cálculos se hacen sobre enrejados simples como el de la figura 1.3. Para cada observación no se tendrán en cuenta todos los estados posibles (el conjunto de etiquetas completo), sino sólo las etiquetas candidatas que devuelve el lexicón para cada palabra. Por ejemplo, para la palabra *casa* sólo se van a considerar las etiquetas referentes a sus dos posibles orígenes, bien sea como sustantivo o como forma conjugada del verbo *casar*.

En la figura 1.3, se representa un enrejado en él que para la *palabra 1* se consideran dos estados o etiquetas posibles: preposición y sustantivo. Esto significa que la *palabra 1* es ambigua y aparece dos veces en el diccionario.

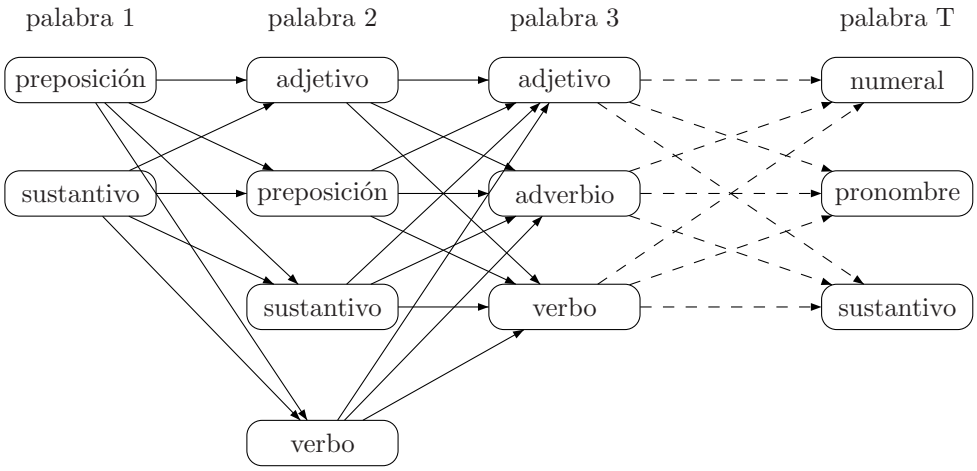

Figura 1.3: Enrejado simplificado para la etiquetación de una frase de T palabras

Sin embargo, la representación de las frases mediante enrejados simples[6] no es suficiente para reflejar la complejidad lingüística de un lenguaje. El principal problema se produce cuando un texto se puede segmentar de diferentes formas,

[6]También conocidos como *diagramas de Trellis*

lo que por ejemplo, resulta habitual con el uso de contracciones y formas verbales con pronombres enclíticos. En estos casos, una palabra está formada por varios componentes lingüísticos que deben ser separados en unidades lingüísticas diferenciadas. Por otra parte, existen locuciones en las que varias palabras actúan como una unidad lingüística compuesta. Así, por ejemplo, en gallego, la palabra *polo* puede ser segmentada en tres formas distintas:

- *Por* (preposición) + *o* (pronombre).
- *Pos* (verbo) + *o* (pronombre).
- *Polo* (sustantivo).

Las locuciones también son una fuente de ambigüedades, ya que en ciertos casos van a dar lugar al menos a dos formas de segmentación, la locución en sí y las palabras que la componen con sus etiquetas independientes. Por ejemplo, para la locución *sin embargo*, tenemos que podemos segmentarla:

- *Sin* (preposición) y *embargo* (sustantivo); como en la frase "*Resolvió sus deudas sin embargo*".
- *Sin embargo* (locución); como en la frase "*Sin embargo, resolvió sus deudas con el banco*".

La detección de estas alternativas de segmentación se debe realizar en una primera fase de preprocesado, y el algoritmo de Viterbi [16] será el encargado de realizar la desambiguación según el contexto en el que estas aparezcan. Para llevar a cabo nuestro proyecto, contamos con un preprocesador morfológico [4]. Dado que este es el marco de desarrollo donde se integra el código que se va a desarrollar, se explicará su funcionamiento pormenorizadamente en la siguiente sección.

La versión inicial del algoritmo de Viterbi sobre *enrejados* clásicos representa para cada palabra sus posibles etiquetas, no obstante esta estructura no permite reflejar correctamente las ambigüedades de segmentación. Los enrejados utilizan nodos que representan las unidades léxicas de la frase, por tanto al encontrarse un caso de segmentación los nodos del enrejado, tendrían que cambiar.

En la figura 1.4, se representa el enrejado correspondiente a la frase "*El sin embargo fue ...*", podemos observar como se generan transiciones no válidas (representadas como líneas discontinuas), desde la etiqueta C (locución) a la etiqueta S (sustantivo), pues la etiqueta C se refiere a la expresión completa. La transición válida se representa con líneas punteadas.

Para poder representar correctamente las ambigüedades de segmentación se necesita una nueva estructura de datos, los *retículos*[7]. Ahora las etiquetas se

[7]Conocidos como *lattices* en inglés

Figura 1.4: Los enrejados no permiten representar segmentaciones ambiguas

van a colocar en los arcos entre los estados, en lugar de formar los propios estados.Y los estados representan los huecos entre los componentes léxicos. Esta nueva representación resulta muy flexible y permite reflejar tanto alternativas de etiquetación como de segmentación.

La utilización de *retículos* implica modificar ligeramente el algoritmo de Viterbi, la nueva versión se conoce como *algoritmo de Viterbi-L* [6]. La principal diferencia es la incorporación de acumuladores para recoger las probabilidades de las palabras encerradas entre el arco de una etiqueta. Esta versión presentaba una anomalía porque las rutas del retículo con menor longitud tenían prioridad sobre las más largas, aún cuando no eran la solución más adecuada.

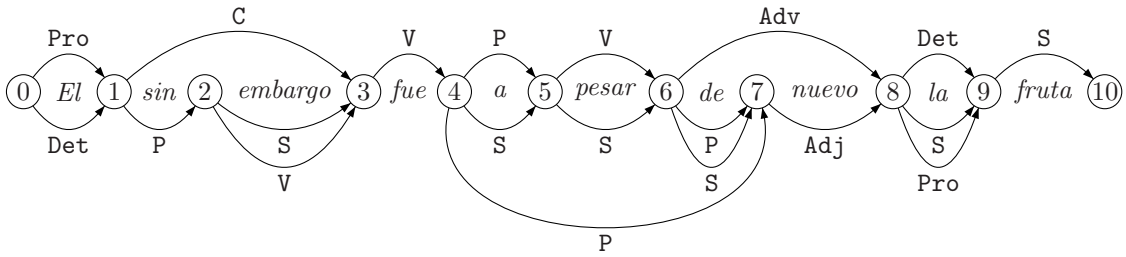

Figura 1.5: Segmentaciones ambiguas representadas en un retículo

Para solucionar esta anomalía se normalizan las probabilidades acumuladas. Gracias al uso de probabilidades logarítmicas los valores normalizados se obtienen dividiendo cada probabilidad entre el número de unidades léxicas, de esta forma los resultados obtenidos son directamente comparables. El algoritmo normalizado se denomina *Viterbi-N* [6].

1.4.2. Preprocesador morfológico

El etiquetador MrTagoo, al igual que la mayoría de etiquetadores actuales, asume que la información que recibe como entrada está separada en frases, y éstas a su vez en unidades de información de alto nivel de significado. Sin embargo, esto no es realista en absoluto ya que las lenguas presentan una serie de fenómenos que se deben tratar antes de la etiquetación. Para llevar a cabo esa labor usaremos el preprocesador morfológico [4] desarrollado para MrTagoo.

Durante el preprocesado, la primera tarea a realizar es la identificación de las frases y de las palabras. La fase de separación de oraciones no es tan simple como pudiese parecer, ya que los puntos no son indicadores exclusivamente del fin de una frase, sino que también pueden representar el final de una abreviatura o de un acrónimo. Además, el término ortográfico de una palabra no siempre coincide con el término lingüístico, como ocurre en el caso de las contracciones y de las formas verbales con pronombres enclíticos, que encierran en una palabra información de dos o más componentes lingüísticos. También nos encontramos el caso de las locuciones, en el que varias palabras actúan como una unidad léxica compuesta.

El preprocesador utilizado en este proyecto opta por separar o unir las palabras en unidades léxicas de alto nivel[8]. Así, separará un verbo de sus pronombres y creará una unidad léxica para cada componente. Las palabras que componen las locuciones se representan mediante una única unidad léxica.

Ejemplo 1.3. *La palabra "colléullelo" se descompone en tres unidades léxicas "colleu", "lles" y "lo", cada unidad se representa en una línea diferente.*

```
colléullelo--------------
colleu Vei30s coller Vei30s coller
lles Rad3ap lle Rad3fp lle Rad3mp lle Rad3ap lle Rad3fp lle ...
lo Raa3ms o Ddms o Scms lo
```

La locución preposicional "a pesar de" formada por tres palabras independientes se representa por una única unidad léxica.

```
a pesar de--------------
a pesar de Lp0 a pesar de
```

■

Además de las locuciones, existen otras unidades lingüísticas compuestas como pueden ser los nombres propios, las fechas y los números (cardinales u ordinales) que se deben tratar antes de ser enviados al etiquetador. Para ello el preprocesador está compuesto por una serie de módulos que se ejecutan de forma secuencial y en

[8] Comúnmente en inglés se denominan *tokens*

el que cada cual trata su propia problemática. En la figura 1.6 se muestra un gráfico con el orden en el que se aplican los módulos del preprocesador.

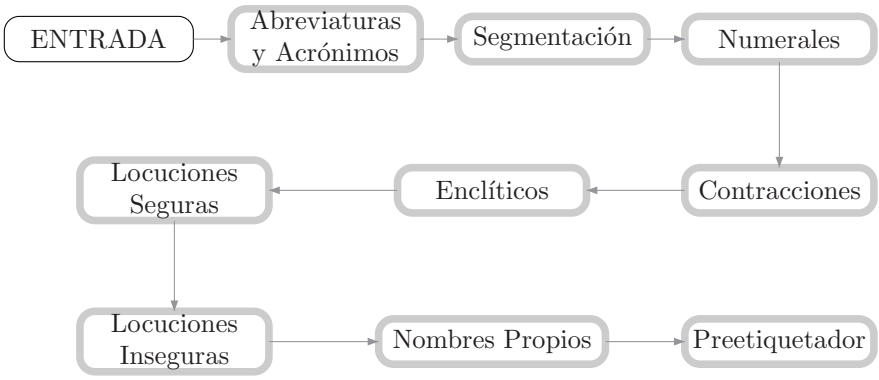

Figura 1.6: Estructura del preprocesador morfológico [4].

Este esquema de preprocesamiento está orientado a la desambiguación y etiquetación robusta del gallego. No obstante, se puede aplicar a otros idiomas con ligeras modificaciones, ya que la mayor parte de los módulos están abstraídos del idioma gracias al uso de los lexicones (que pueden ser contruidos para cada idioma). Los resultados obtenidos serán más exactos cuánta más información lingüística esté definida para un idioma. Esto significa que en los casos en qué se disponga de esta información se integrará en el sistema con el objetivo de obtener un comportamiento lo más veraz posible. Actualmente, se dispone de la información lingüística completa para la lengua gallega, es por ello que las mayoría de pruebas y ejemplos expuestos en el proyecto serán realizados en este idioma.

1.4.3. Modelización del diccionario

Como se ha mencionado anteriormente, el entorno de etiquetación utiliza una serie de *diccionarios* o *lexicones*. Estos lexicones son listas de palabras de un lenguaje asociadas a una serie de información. Aunque se pudiese pensar en las bases de datos relacionales como la herramienta más adecuada para gestionar este tipo de información, las características de los diccionarios que se enumeran a continuación ponen de manifiesto que no son el sistema de almacenamiento más adecuado[5]:

1. *Gran tamaño:* Un lexicón para el español puede llegar a contener millones de entradas.

2. *Accesos rápidos:* La recuperación de información debe ser lo más eficiente posible en términos de tiempo.

3. *Inserciones poco frecuentes:* Los lexicones apenas varían, por ello la ordenación se realiza una sola vez.

Por su parte, el diccionario empleado por las herramientas de las que partimos almacena la siguiente información por palabra:

- *Forma:* Es la propia palabra tal y como se escribe, esto es, su forma gráfica.
- *Etiqueta:* Es la categoría morfosintáctica que corresponde a la palabra.
- *Lema:* Es la forma canónica de la palabra, que quiere decir que todas las palabras derivadas de una misma, comparten un mismo lema. Los lemas suelen coincidir con las entradas de un diccionario convencional (en papel). Para los tiempos verbales, el lema es su infinitivo y para sustantivos lo será la forma en masculino y singular.
- *Probabilidad de emisión:* Es la probabilidad de que la etiqueta de la palabra aparezca asociada a ella. Esta probabilidad se calcula partir de un corpus de entrenamiento y se corresponde con la distribución de probabilidad de los sucesos observables para cada estado del MMO.

En el caso de las palabras ambiguas, es decir, aquellas que tienen más de una etiqueta posible, aparecen repetidas tantas veces como etiquetas tengan. Así por ejemplo, la palabra *casa* aparecería en el lexicón con al menos tres entradas distintas. A saber:

```
casa  Scfs    casa   ...
casa  V0m20s  casar  ...
casa  Vpi30s  casar  ...
```

El modelo que se utiliza para almacenar está información de forma compacta ha sido propuesto por Graña [5] y está basado en la utilización de Autómatas Finitos Deterministas Acíclicos Numerados (AFDAN). Estos autómatas son mínimos, para ello se aplica un algoritmo de minimización de autómatas descrito por Daciuk [2]. El tamaño de los lexicones es crucial ya que pueden llegar a contener desde cientos de miles de entradas hasta millones, por lo que el espacio utilizado en memoria es muy importante. Gracias a la minimización de los autómatas las palabras comparten tanto prefijos como sufijos, de esta forma el espacio ocupado en memoria se ve reducido en gran medida. En la figura 1.7 , se muestra un árbol de letras que representa las palabras *chispa*, *cohabitante* , *coherente* y *cooperarse*, antes de aplicar la minimización y en la figura 1.8 el resultado que se obtiene una vez aplicado.

Además, mediante la utilización de los AFDAN el tiempo de reconocimiento de una palabra es lineal con respecto a la longitud de la propia palabra, no depende

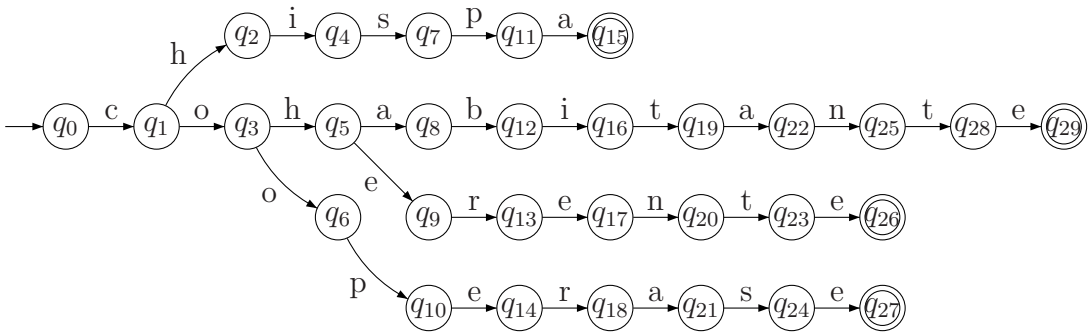

Figura 1.7: Antes de aplicar el algoritmo de minimización

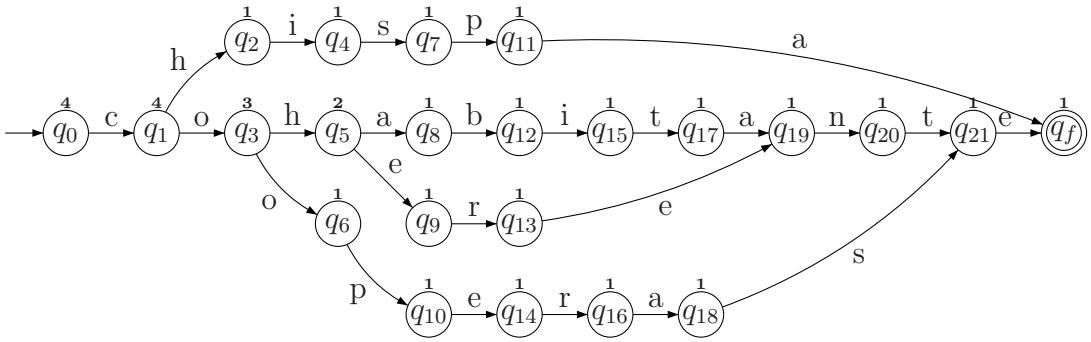

Figura 1.8: Después de aplicar el algoritmo de minimización

del tamaño del diccionario, ni del tamaño del autómata. Al tratarse de un autómata numerado se puede obtener para cada palabra un número que identifique de forma unívoca cada palabra mediante el uso de la función `palabra_a_índice` [5].

Ésta resulta ser la mejor función asociativa [9] que se podría utilizar para acceder de forma eficiente a la información ya que no permite colisiones y dispersa los datos en la tabla. Su funcionamiento es simple, se recorre la ruta de una palabra w, y se van sumando los pesos de los estados accesibles mediante letras lexicográficamente menores que w_i. Y, adicionalmente, a la suma final se añade uno.

Ejemplo 1.4. *A modo de ejemplo, se calculará el índice para la palabra coherente del ejemplo 1.8. La ruta o traza seguida por la palabra en el autómata es la siguiente:*

[9] Resulta frecuente el uso del término en inglés función *hash*.

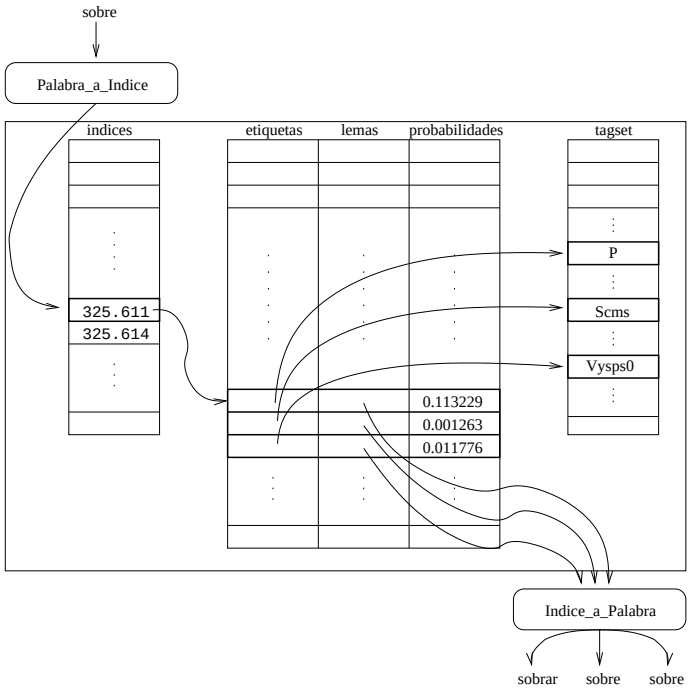

Figura 1.9: Modelización compacta de un diccionario [11].

$\{q_0, q_1, q_3, q_5, q_9, q_{13}, q_{19}, q_{20}, q_{21}, q_f\}$.

Los únicos estados que a su vez tienen estados accesibles mediante letras menores que la de la letra correspondiente a la propia palabra son q_1 y q_5. Para estos estados (q_2 y q_8, respectivamente) sus pesos son 1, y sumando la unidad adicional obtenemos como resultado índice de la palabra coherente el 3. Se puede comprobar fácilmente que los índices se asignan en orden alfabético.

■

En la figura 1.9 se representa el esquema de funcionamiento del diccionario. En primer lugar, se utiliza la función `palabra_a_indice`, mediante la que se obtiene una posición en la tabla de índices para una palabra dada. Esta posición, a su vez, contiene un número utilizado para indexar los tableros de *etiquetas*, *lemas* y *probabilidades*, es decir la posición absoluta que ocupa la palabra buscada dentro del lexicón. Estas tres tablas tienen la misma longitud.

En el tablero de etiquetas, tampoco se van a almacenar directamente las etiquetas, sino números que indexan la tabla del conjunto de etiquetas [10].

El tablero de *lemas* también almacena números. Dado que los lemas también forman parte del diccionario se almacena el número que devolvería la función

[10] Comúnmente conocido como tagset.

`Palabra_a_Indice` aplicada sobre el lema. Para recuperarlo se aplica la función inversa: `Indice_a_Palabra`.

En el tablero de *probabilidades* se almacenan directamente las probabilidades, pues son datos numéricos.

Como consecuencia, la estructura utilizada para almacenar el lexicón es lo más compacta posible, además permite una gran flexibilidad a la hora de incorporar o eliminar algún tablero.

Puesto que el diccionario está ordenado alfabéticamente, se garantiza que cuando una palabra tenga más de una etiqueta posible aparecerán contiguas. Para saber cuantas etiquetas tiene una palabra, basta con restar la posición absoluta de la palabra al valor de la siguiente casilla en el tablero de índices.

En el esquema se realiza el acceso para la palabra *sobre* cuya posición absoluta en el lexicón es la 325.611, se muestra como para cada posible etiqueta de la palabra se incluye una nueva fila. Para saber cuantas etiquetas posee la palabra restaremos al valor de la siguiente casilla (325.614) en la tabla de índices, la posición absoluta de *sobre* (325.611), obteniendo tres etiquetas diferentes.

1.4.4. Corrector ortográfico

Hasta ahora no se ha mencionado la forma en la que debe actuar un etiquetador al encontrar una palabra desconocida, es decir, que no forme parte del lexicón. En estos casos caben dos posibilidades:

1. Que se trate de palabras correctas que no formen parte del diccionario por ser extranjerismos, tecnicismos o palabras de creación reciente.

2. Que sean palabras incorrectas, que contengan errores ortográficos .

El entorno de etiquetación actual[11] en estos casos deja la palabra tal y como aparece y utiliza un *módulo de adivinación* [12] para deducir la etiqueta correspondiente a la palabra.

Una alternativa al *módulo de adivinación* es la utilización de una técnica de corrección ortográfica que obtenga alternativas de coste mínimo (cuya distancia de edición con la palabra a elegir sea la menor posible), que sí aparecen en el diccionario. El principal objetivo de este proyecto es la integración de una técnica de corrección ortográfica en el entorno de etiquetación del que partimos.

Para llevar a cabo la integración partimos de dos propuestas que implementan el diccionario como una estructura AFDAN. La primera se debe a Savary [12] y es una

[11] Cuando se mencione el entorno de etiquetación, nos estamos refiriendo al preprocesador junto al etiquetador morfológico.

[12] Conocido como *guesser* en inglés

técnica de *corrección global*, es decir, realiza una búsqueda exhaustiva a lo largo de todo el diccionario. Por este motivo, garantiza la localización de todas y cada una de las correcciones posibles, pero su coste computacional va a ser alto.

La otra técnica fue propuesta por Vilares et al. [15] e implementa una estrategia de *corrección regional*, que evita explorar todo el lexicón y suele ofrecer un menor número de alternativas de reparación.

Como resultado de aplicar una técnica de corrección, bien sea global o regional, sobre una palabra desconocida es frecuente obtener varias opciones de corrección. De esta forma se identifica un nuevo tipo de *ambigüedad*, denominada, *léxica*.

Antes de explicar detalladamente los algoritmos debemos introducir algunos conceptos con respecto a la corrección:

- Damerau [3] clasificó en 1964 los errores ortográficos generados por humanos en los siguientes:
  - Inserción de una letra (Ejemplo: "caasa" en vez de "casa").
  - Borrado de una letra (Ejemplo: "cas" en vez de "cas").
  - Sustitución de una letra por otra (Ejemplo: "cass" en vez de "casa").
  - Transposición o inversión de dos letras contiguas (Ejemplo: "caas" en vez de "casa").
- Wagner [17] define el concepto de *distancia de edición mínima* como el número mínimo de operaciones de edición, descritas por Damerau, necesarias para transformar una cadena de texto en otra.

Algoritmo de Savary

Savary [12] describe un método global de corrección ortográfica sobre AFs. Este algoritmo se basa en la búsqueda de los vecinos más cercanos para una palabra desconocida, es decir, aquellas palabras válidas cuya distancia de edición con la palabra de entrada sea mínima.

En el algoritmo se debe establecer un *umbral* que la distancia de edición mínima nunca debe superar. Para fijarlo se debe llegar a un compromiso entre tres factores:

- La eficiencia en tiempo de búsqueda, que se corresponde con el tamaño de la sección del AF que ha de ser explorarado.
- El tamaño de la lista de correcciones candidatas.
- La probabilidad de que la palabra esperada esté en la lista.

En el algoritmo 1 se muestra una implementación *recursiva* del algoritmo de Savary, donde:

Algorithm 1 Algoritmo de Savary[12]

```
1: procedimiento BUSQUEDA-TOLERANTE(w_{1..n}, q, t)
2:     (ed, S) ← (∞, ∅)
3:     si n = 0 y TERMINAL(q) entonces retornar (0, {ε})
4:     si n > 0 y ∃q′ / q.w_1 = q′ entonces
5:         (ed, S) ← BUSQUEDA-TOLERANTE(w_{2:n}, q′, t)
6:         t ← min(t, ed)
7:         para cada suf ∈ S hacer suf ← w_1 ⊕ suf
8:     fin si
9:     si t > 0 entonces
10:        si n > 0 entonces //1. Error de inserción de una letra
11:            (ed′, S′) ← BUSQUEDA-TOLERANTE(w_{2:n}, q, t − 1)
12:            (ed, S) ← AÑADIR-O-REEMPLAZAR((ed, S), (ed′ + 1, S′))
13:            t ← min(t, ed)
14:        fin si
15: //2. Error de inversión de una letra con la siguiente
16:        si n > 1 y w_1 ≠ w_2 y ∃q′ / q.w_2.w_1 = q′ entonces
17:            (ed′, S′) ← BUSQUEDA-TOLERANTE(w_{3:n}, q′, t − 1)
18:            para cada suf ∈ S′ suf ← w_2 ⊕ w_1 ⊕ suf
19:            (ed, S) ← AÑADIR-O-REEMPLAZAR((ed, S), (ed′ + 1, S′))
20:            t ← min(t, ed)
21:        fin si
22:        trans ← {(q′, c)} / q.c = q′
23:        para cada (q′, c) ∈ trans hacer
24: //3. Error de omisión de una letra
25:            (ed′, S′) ← BUSQUEDA-TOLERANTE(w_{1..n}, q′, t − 1)
26:            para cada suf ∈ S′ suf ← c ⊕ suf
27:            (ed, S) ← AÑADIR-O-REEMPLAZAR((ed, S), (ed′ + 1, S′))
28:            t ← min(t, ed)
29: //4. Error de reeemplazo de una letra por otra
30:            si n > 0 entonces
31:                (ed′, S′) ← BUSQUEDA-TOLERANTE(w_{2:n}, q′, t − 1)
32:                para cada suf ∈ S′ suf ← c ⊕ suf
33:                (ed, S) ← AÑADIR-O-REEMPLAZAR((ed, S), (ed′ + 1, S′))
34:                t ← min(t, ed)
35:            fin si
36:        fin para
37:    fin si
38:    retornar (ed, S)
39: fin procedimiento
```

- $w_{1..n}$ es la palabra de entrada,
- q es el estado en que se encuentra actualmente el AF y
- t es el umbral de error permitido.

La función BUSQUEDA-TOLERANTE trata de reconocer el sufijo $w_{1..n}$ partiendo del estado q y admitiendo t operaciones elementales de edición. La función devuelve un par (ed, S), donde S es el conjunto de sufijos, y ed es la distancia de edición entre el sufijo $w_{1..n}$ y cada uno de los sufijos contenidos en S. Si no hay sufijos en S, entonces $ed = \infty$, por ser la condición de inicialización en la línea 2.

La primera llamada a la función se hace con la cadena completa de entrada, el estado inicial del AF y el umbral de distancia de edición deseado. Inicialmente se seguirá el procedimiento estándar de reconocimiento que consiste en ir avanzando por los estados del AF con cada carácter de la cadena de entrada, tal y como puede verse entre las líneas 4 y 8. Si la palabra w pertenece al lexicón, entonces en algún momento t se hará igual a cero en la línea 6 de forma que al llegar a la línea 9 se evitará la ejecución del código que introduce las hipótesis de error entre las líneas 9 y 37.

Sin embargo, en el caso de que la palabra de entrada sea errónea, entonces el procedimiento estándar de reconocimiento acabará fallando por uno de los dos motivos siguientes: la cadena de entrada se ha agotado y no hemos alcanzado un estado final o bien se ha alcanzado un estado desde el que no es posible transitar con el carácter actual. En ambos casos alcanzaríamos la línea 9 con $t > 0$, lo que nos revelaría que existe un error en la posición actual de la cadena de entrada y trataríamos de aplicar en este punto alguna de las cuatro hipótesis elementales de error:

1. *Inserción de la letra actual* (líneas 10 a 14), en cuyo caso omitiríamos esa letra y trataríamos de reconocer el resto de la palabra de entrada ($w_{2..n}$) partiendo del estado actual.

2. *Inversión de la letra actual y la siguiente* (líneas 15 a 20). Para que esto ocurra es necesario que nos resten al menos 2 caracteres por reconocer y deberá existir un estado $q\prime$ alcanzable desde el estado actual, utilizando para ello los dos caracteres siguientes invertidos ($\exists q\prime \ / \ q.w_2.w_1 = q\prime$). En ese caso, trataríamos de reconocer el sufijo restante ($w_{3..n}$) partiendo del estado $q\prime$.

3. *Omisión de una letra en la posición actual* (líneas 25 a 28), en cuyo caso para cada transición que nos lleve desde el estado actual q a otro estado $q\prime$ por medio del carácter c (línea 23), trataremos de reconocer el sufijo actual, pero no desde el estado actual q sino desde cada uno de los $q\prime$, añadiendo luego

a cada una de las soluciones encontradas el carácter c correspondiente (línea 26).

4. *Reemplazamiento de la letra correcta por la letra actual* (líneas 30 a 35), en cuyo caso para cada transición que nos lleve desde el estado actual q a otro estado $q\prime$ por medio del carácter c, trataríamos de reconocer el resto de la palabra de entrada ($w_{2..n}$) partiendo de cada uno de los estados $q\prime$ y añadiendo luego a cada una de las soluciones encontradas el carácter c correspondiente (línea 32).

Cada vez que se encuentran nuevas soluciones el valor de *ed* y el contenido de S se actualizan por medio de la función Añadir-O-Reemplazar. Esta función comprueba si las nuevas soluciones son más cercanas a la palabra original que las soluciones que ya se encuentran en S, en cuyo caso S es reemplazado por el conjunto de nuevas soluciones, y el valor de *ed* pasa a ser la distancia de edición que existe entre la cadena de entrada y cada una de las nuevas soluciones. En caso contrario, se realiza la unión de los dos conjuntos y *ed* no se modifica. De este modo, sólo se retienen en S las soluciones con la menor distancia de edición respecto al sufijo original.

Algoritmo de corrección regional

El algoritmo de corrección ortográfica regional [15] aplica heurísticas para reducir el coste computacional de los algoritmos globales, además trata de mejorar la calidad de la respuesta obtenida, reduciendo el número de correcciones posibles para una palabra desconocida.

Los algoritmos globales, como el de Savary [12], dedican el mismo esfuerzo computacional en todos los puntos de la cadena que tratan de reconocer. Esto significa que no tienen en cuenta el punto en el que se ha producido el error, es decir, la transición exacta con la cual no se ha podido continuar el autómata. No obstante, parece lógico pensar que el error encontrado esté originado cerca del lugar en el que se ha detectado y no varios caracteres antes.

Cuando no sea posible transitar en el proceso de reconocimiento de una palabra en un AF, se pueden diferenciar dos situaciones distintas de error:

1. Se ha producido un error en el punto de la cadena en el que nos encontramos. En este caso la palabra se corregiría con una reparación local, es decir, en ese punto exacto.

2. Se ha producido un error en un punto anterior, lo que ha provocado que continuásemos por un camino erróneo, en el que no se ha podido seguir. Este camino erróneo tuvo que ser tomado en un punto del AF en el que había una bifurcación.

Para establecer el punto al que se va a retroceder cada vez que se encuentre un error se introduce un nuevo concepto, las *regiones*, que son una forma de delimitar la zona de búsqueda del error. El estado origen de una región será aquel para el que exista más de una transición de salida (en el que haya una bifurcación). Y el estado destino es aquel en el que convergen todos los caminos que parten desde el estado origen. Para garantizar que todos los caminos convergen en un estado, se debe asegurar la existencia de un único estado final.

En la figura 1.10 se muestra un AF que representa las palabras *descontar*, *despuntar*, *recontar* y *repuntar*. Este AF está formado por dos regiones, la primera con *estado origen* q_0 y la *estado destino* q_4, y la segunda empieza en q_4 y converge en q_7.

Para calcular los estados destino de las regiones en un AF, se utiliza un algoritmo basado en técnicas de programación dinámica [11], con el fin de evitar la repetición de cálculos comunes a regiones distintas.

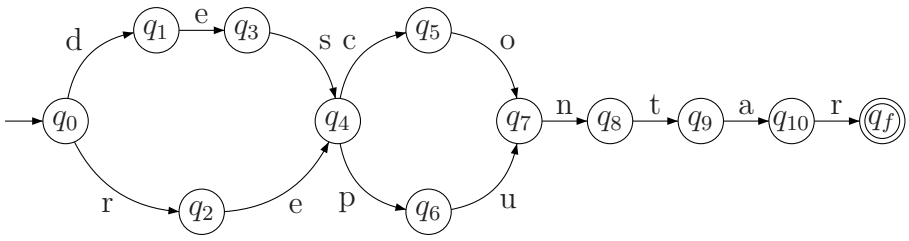

Figura 1.10: Cálculo de regiomes sobre un AF

En el momento en el que no se pueda continuar el reconocimiento de una palabra sobre un autómata, se retrocederá hasta el origen de la *región mínima*, es decir, el primer estado anterior a q en el que exista una bifurcación de caminos en el AF.

Una vez identificada la región sobre la que se debe actuar, se aplican las modificaciones necesarias para calcular el error, basándose en las operaciones de edición descritas por Damerau [3]. Al igual que Savary, se emplea como métrica de error la *distancia de edición* [17].

En este algoritmo también se contempla la posibilidad de *errores en cascada*, esto es, errores provocados por una reparación previa errónea. Una vez aplicada una reparación, si no podemos continuar por esa rama, se retrocederá hasta la región inmediatamente anterior a la región actual. Esto va a asegurar una calidad comparable a las estrategias globales, pero con los costes computacionales de una local.

1.5. Objetivos del proyecto

En el apartado anterior se ha intentado dar una visión global del complejo entorno donde se enmarca nuestro proyecto, para ahora explicar los objetivos generales a alcanzar en nuestro desarrollo.

1.5.1. Integración de varias herramientas

La finalidad principal de este proyecto es la construcción de un analizador morfosintáctico robusto que sea capaz de resolver tres posibles tipos de ambigüedades:

- *Morfosintáctica:* Cuando una unidad léxica es susceptible de recibir más de una etiqueta. Por ejemplo, "*para*" puede ser una preposición o un verbo.
- *De segmentación:* Cuando una frase pueda segmentarse de más de una manera posible. Por ejemplo, la locución *sin embargo*, se puede segmentar de dos formas:
  - *Sin* (preposición) y *embargo* (sustantivo); como en la frase "*Resolvió sus deudas sin embargo*".
  - *Sin embargo* (locución); como en la frase "*Sin embargo, resolvió sus deudas con el banco*".
- *Léxica:* uando existe más de una unidad léxica posible, este caso puede ocurrir cuando la palabra original fuese errónea [13].

Como se ha explicado, el algoritmo de Viterbi [16], implementado en MrTagoo [4], resuelve las ambigüedades de segmentación y también las morfosintácticas. Pero la herramienta existente no resuelve los casos en los que una palabra no forma parte del lexicón, simplemente trata de predecir o adivinar su etiqueta morfosintáctica, esto es, si se trata de un verbo, un sustantivo, un adjetivo..., así como su género, número o la forma verbal en la que se encuentra cuando corresponda.

Nuestro objetivo principal, se centrará en integrar las implementaciones de las técnicas de corrección ya existentes con el preprocesador morfológico, de forma que el algoritmo de Viterbi pueda resolver también las ambigüedades léxicas generadas una vez aplicadas las correcciones sobre las palabras erróneas. Esto significa que la aplicación resultante funcionará como un *corrector ortográfico contextual*, es decir, tendrá en cuenta el contexto en el que aparece la palabra para elegir la alternativa más correcta.

[13] Entendemos como errónea una palabra que no forma parte del diccionario.

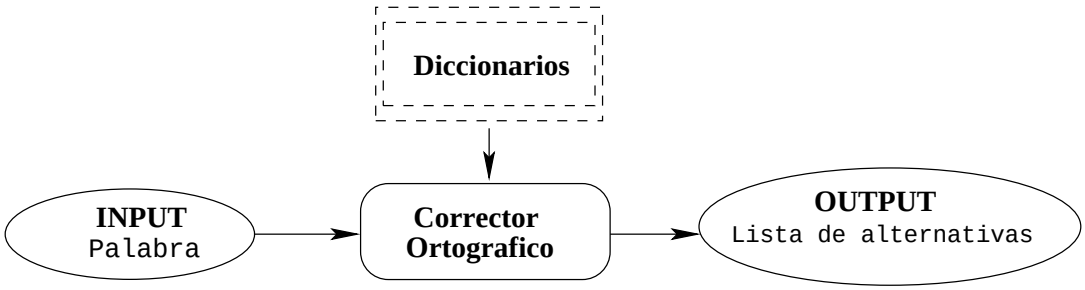

Figura 1.11: Funcionamiento general de las técnicas de corrección

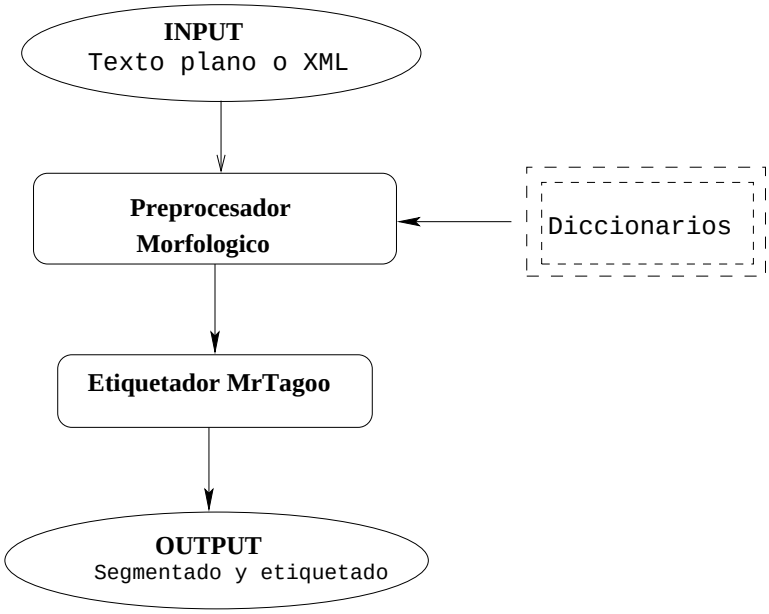

Figura 1.12: Entorno de etiquetación actual

La ambigüedad léxica se produce al obtenerse como resultado de la corrección una lista con las alternativas de coste mínimo para la palabra desconocida. En caso de que no hubiese correcciones posibles con el umbral establecido, se seguiría aplicando el módulo de adivinación. La figura 1.11 muestra una descripción gráfica del funcionamiento del corrector.

En la figura 1.12 se muestra de forma gráfica el funcionamiento del entorno de etiquetación del que se parte.

Se describe gráficamente en la figura 1.13 la integración de las herramientas descritas que se pretende obtener,

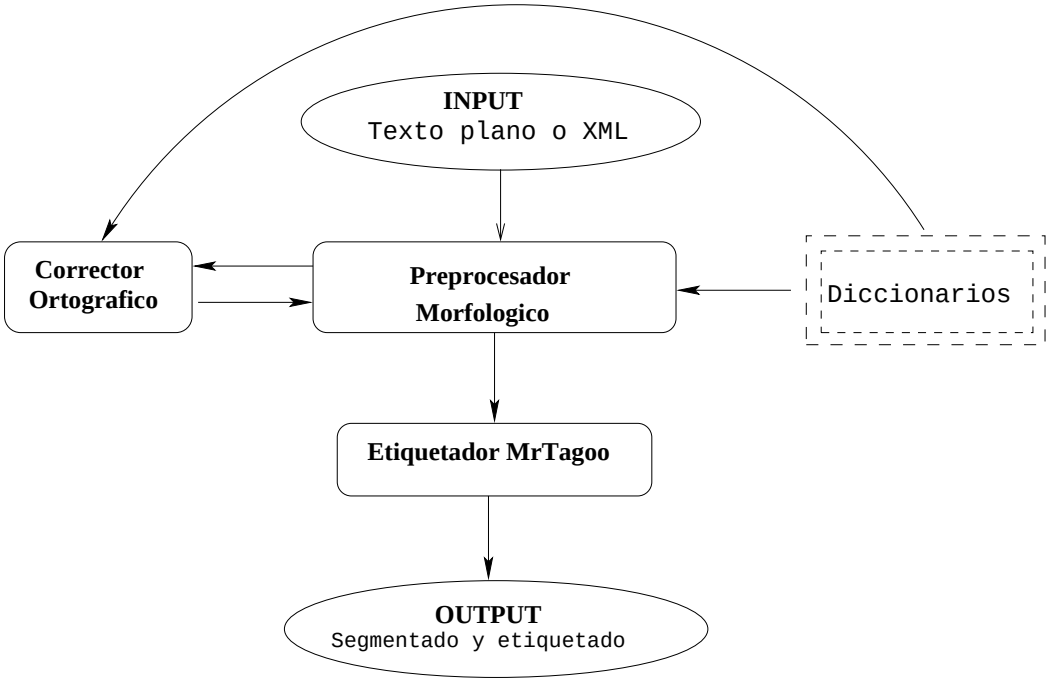

Figura 1.13: Esquema general del funcionamiento del proyecto.

1.5.2. Recodificación del corrector

Las dos implementaciones de técnicas de corrección ortográfica de las que disponemos son prototipos empleados para la realización de tests sobre listados de palabras. Por tanto, se ha de adaptar el código a las necesidades concretas de nuestro proyecto.

1.5.3. Corrección a errores multi-palabras

El último de los propósitos generales de nuestro proyecto es la ampliación de los tipos de errores a considerar por el corrector. La aplicación existente tiene en cuenta los cuatro tipos de errores de edición descritos por Damerau [3], que han sido descritos en la sección 1.4.4.

Se ampliará el corrector para que pueda resolver errores que afecten a varias palabras, como la unión de dos palabras o la división de una palabra en varias. Así, se tratara de solucionar errores como “sinembargo” o “ hier ba”.

El primer caso es la unión de dos palabras conocidas formando una palabra desconocida, es decir, omitir el espacio en blanco entre ambas palabras. En caso de que existan más alternativas a la solución, todas serán devueltas y será el etiquetador el encargado de seleccionar la más adecuada en función del contexto en el que se

hallen.

Otra posible funcionalidad a incorporar son los errores de división de palabras, puesto que es una falta común introducir un espacio en una posición incorrecta cuando escribimos. Cuando nos topemos con una palabra incorrecta se deberá comprobar si uniendóla con su predecesora o con su sucesora se consigue formar alguna palabra que conste como entrada en nuestro diccionario o lexicón.

Capítulo 2

Tecnologías y herramientas empleadas

En el presente capítulo se hará una breve descripción de las tecnologías y las herramientas empleadas para llevar a cabo este proyecto, destacando características representativas de las mismas, así como los motivos que han motivado su selección frente a otras herramientas y técnicas similares.

2.1. Tecnologías empleadas

2.1.1. C

El lenguaje de programación C fue creado en 1972 por Dennis Ritchie y Ken Thompson en los laboratorios BELL. C es la evolución de un lenguaje anterior denominado B, que, a su vez, está basado en el lenguaje BCPL [13].

C es un lenguaje de nivel medio porque combina elementos de lenguajes de alto nivel con la funcionalidad del lenguaje ensamblador. La utilización de este lenguaje permite la manipulación elementos de bajo nivel como los bits, bytes y direcciones, de esta forma el código generado es muy eficiente. Además, es portable a cualquier arquitectura.

Estas características convirtieron a C en el lenguaje idóneo para el desarrollo de las herramientas que emplearemos en este proyecto, en las que se manejan autómatas con requerimientos específicos de eficacia computacional. En consecuencia, nuestro sistema también utilizará dicho lenguaje.

2.1.2. XML

XML[1] es un metalenguaje extensible de etiquetas desarrollado por el *World Wide Web Consortium* (W3C). Este lenguaje permite estructurar, describir e intercambiar información de forma fiable, segura y simple. Actualmente, el lenguaje XML goza de una gran popularidad por ser un estándar abierto y libre. Además, es ampliamente utilizado para escribir los ficheros de configuración de un programa o para permitir la compatibilidad entre diferentes sistemas al servir de formato de intercambio de datos. A continuación se destacan algunas de las principales características XML:

- Es extensible, es decir, después de diseñado y puesto en producción, es posible extender XML con la adición de nuevas etiquetas, de modo que se pueda continuar utilizando sin complicación alguna.
- No es necesario crear un analizador específico para cada versión del lenguaje XML. Esto posibilita el empleo de cualquier analizador disponible. De esta manera se evitan errores y se acelera el desarrollo de aplicaciones.
- Si un tercero decide usar un documento creado en XML, es sencillo entender su estructura y procesarla. Mejora la compatibilidad entre aplicaciones.
- La codificación del contenido en XML consigue que la estructura de la información resulte más accesible. Esto lo convierte en un formato ideal para guardar datos de configuración de las aplicaciones.
- Presenta elementos como las DTD's o XMLSchema que permiten verificar de forma sencilla si la estructura del documento se corresponde con la deseada.
- Existen abundantes herramientas para trabajar con información representada en este formato. Estas herramientas facilitan el acceso, tratamiento y modificación de los datos. En concreto, en este proyecto se empleará la tecnología SAX combinada con el lenguaje de programación C.

En este proyecto, se utilizan documentos XML en los ficheros de configuración y, también, como formato para almacenar los resultados obtenidos.

2.1.3. SAX

SAX son las siglas de *simple* API *for* XML (API simple para XML), se creó originalmente como un API para *Java* y es un estándar *de facto*. Se han realizado versiones de SAX para muchos otro lenguajes de programación, tales como *Perl, Pyhton, Pascal, C, C++ y Visual Basic*.

[1] *http://www.w3.org/XML/*.

En nuestro proyecto, se emplea la interfaz de SAX en *libxml* para el lenguaje C. Aunque existen otras alternativas para el manejo de documentos XML como DOM (Document Object Model), utilizaremos SAX porque la cantidad de memoria que necesita es inferior a la ocupada por DOM, ya que no construye la representación de la estructura en memoria, sino que se limita a recorrer secuencialmente el documento para obtener la información deseada.

Dada la sencillez de los ficheros de configuración utilizados en nuestro proyecto se empleará la tecnología SAX para cargar su contenido.

2.1.4. LaTeX

LaTeX es un lenguaje compilado para el procesamiento de textos, formado por un gran conjunto de macros de TeX, escritas inicialmente por *Leslie Lamport* [9], con la intención de facilitar el uso del lenguaje de composición tipográfica creado por *Donald Knuth*. Es muy utilizado para la composición de artículos académicos, tesis y libros técnicos, dado que la calidad tipográfica de los documentos realizados con LaTeX es comparable a la de una editorial científica de primera línea. *LaTeX* es *software* libre bajo licencia *Licencia Pública del Proyecto LaTeX* (LPPL). Se ha utilizado *LaTeX* para realizar esta documentación.

2.2. Herramientas utilizadas

En esta sección se describen brevemente las diferentes herramientas utilizadas en la realización de las distintas tareas comprendidas en este proyecto. Para cada una de ellas se explicará con qué fin se ha empleado en este proyecto y porqué se ha seleccionado frente a otras alternativas similares.

2.2.1. NetBeans

NetBeans es una plataforma para el desarrollo de aplicaciones de escritorio, así como un *entorno integrado de desarrollo* (IDE).

El IDE de *Netbeans* es una herramienta para programadores pensada para escribir, compilar, depurar y ejecutar programas. Se trata de un producto libre gratuito y sin restricciones de uso. Está escrito en *Java* pero contiene un *plugin* para añadir soporte a los lenguajes C y C++ entre otros.

Se ha decidido emplear este IDE frente a otros, como por ejemplo *Eclipse*, que también incluye soporte para C dada la familiaridad previa con el entorno *Netbeans*, además de la inclusión de un mayor número de funcionalidades.

2.2.2. Visual Paradigm

Visual Paradigm es una herramienta CASE que utiliza UML como lenguaje de modelado (*www.visual-paradigm.com*). La herramienta está diseñada para una amplia gama de usuarios interesados en construir sistemas de *software* fiables con el uso del paradigma orientado a objetos, incluyendo actividades como ingeniería de *software*, análisis de sistemas y análisis de negocios. Se ha utilizado esta herramienta para la elaboración del análisis y del diseño en este proyecto. Aunque el lenguaje de programación empleado no es orientado a objetos se ha intentado seguir una adaptación lo más fiel posible en el momento de realizar la implementación.

2.2.3. Kile

Kile es un editor de Tex/LaTeX. Funciona conjuntamente con KDE en varios sistemas operativos. Las principales características por las que se ha seleccionado *Kile* frente a otros editores son las siguientes:

- Autocompletado de comandos LaTeX.
- Coloreado de sintaxis. *Kile* automáticamente marca los comandos LaTeX y resalta los parentesis.
- Permite crear proyectos y trabajar con múltiples ficheros simultaneamente.
- Proporciona plantillas y patrones para facilitar la creación de documentos.
- Plegado de código.
- Inserción fácil de citas y referencias cuando se usan proyectos.

2.2.4. Planner

Planner es una herramienta para la gestión de proyectos que permite realizar diagramas de Gantt, asignación de recursos, etc. Forma parte del proyecto *Gnome* y se puede utilizar sin problemas en KDE.

2.2.5. Kubuntu

Kubuntu[2] es una distribución GNU/Linux que ofrece un sistema operativo principalmente enfocado a computadores personales. Es una versión derivada de *Ubuntu* que usa el entorno de escritorio KDE más configurable que *Gnome*, entorno utilizado por *Ubuntu* Es una de las distribuciones de GNU/Linux más extendidas

[2] http://www.kubuntu.org/

actualmente. Se basa en *Debian* y concentra su objetivo en la facilidad y libertad de uso, la fluida instalación y los lanzamientos regulares (cada 6 meses: las versiones .04 en abril y las .10 en octubre). El principal patrocinador es Canonical Ltd., una empresa privada fundada y financiada por el empresario sudafricano *Mark Shuttleworth*. La versión utilizada en este proyecto es la 7.10, lanzada en octubre del 2007.

Capítulo 3

Planificación y presupuesto

3.1. Planificación del proyecto

3.1.1. Planificación inicial

La planificación inicial se divide en las siguientes fases:

- Estudios previos.y
- Análisis y diseño.
- Implementación y pruebas.
- Documentación.

Como consecuencia de la existencia de las herramientas que se han descrito en la sección 1.4 , la fase de estudios previos va a englobar la comprensión de dichas herramientas. Por este motivo, esta fase abarcara un período de tiempo significativo en el desarrollo de nuestro proyecto. La tabla de tiempos y el diagrama de *Gantt* para la planificación inicial se presentan en las figuras 3.1 y 3.2.

WBS	Name	Start	Finish	Work	Duration
1	**PROYECTO**	**oct 13**	**mar 30**	**120d**	**120d**
1.1	**Estudios Previos**	**oct 13**	**dic 8**	**29d**	**41d**
1.1.1	Estudio del problema	oct 13	oct 23	9d	9d
1.1.2	Estudio de las herramientas	nov 11	dic 1	15d	15d
1.1.3	Estudio de las tecnologías	dic 2	dic 8	5d	5d
1.2	**Fase de análisis y diseño**	**dic 9**	**ene 20**	**20d**	**30d**
1.2.1	Análisis del sistema	dic 9	dic 22	10d	10d
1.2.2	Diseño del sistema	ene 6	ene 20	10d	10d
1.3	Implementación	feb 4	mar 10	25d	25d
1.4	Pruebas finales	mar 11	mar 16	4d	4d
1.5	**Documentación**	**oct 24**	**mar 30**	**42d**	**111d**
1.5.1	Documentación de la investigación	oct 24	nov 10	12d	12d
1.5.2	Documentación del análisis	dic 23	ene 5	10d	10d
1.5.3	Documentación del diseño	ene 21	feb 3	10d	10d
1.5.4	Manual técnico y de usuario	mar 17	mar 30	10d	10d

Figura 3.1: Fases del proyecto en la planificación inicial.

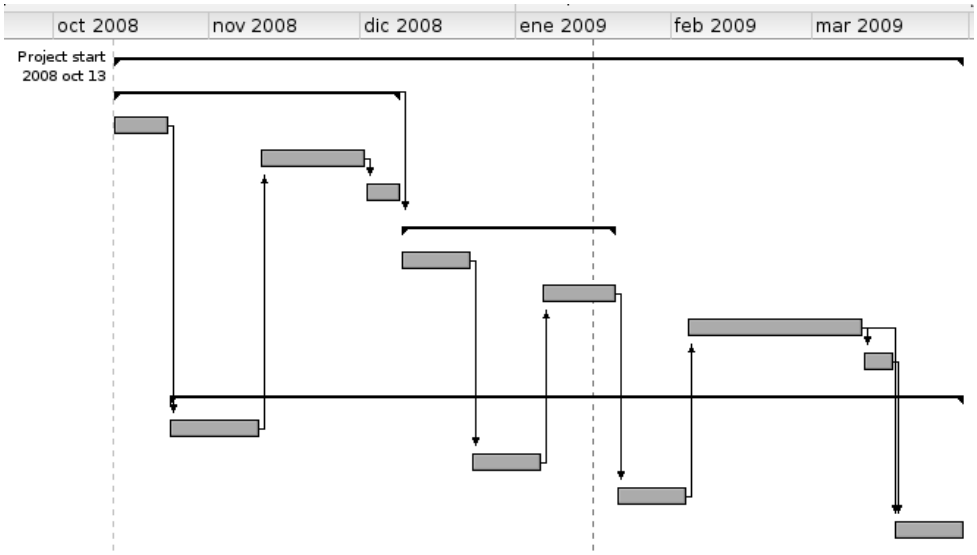

Figura 3.2: Diagrama de Gantt para la planificación inicial.

3.1.2. Planificación real

La planificación real del proyecto se presenta en las figuras 3.3 y 3.4. Las fases de desarrollo inicialmente previstas no han variado durante la realización del proyecto, en cambio, sí se han producido algunas desviaciones poco significativas en las diferentes etapas que no han afectado a la duración total.

La principal razón por la que se han producido las variaciones se debe a la mayor duración de las fases de estudio de las herramienta existentes y la documentación de la investigación llevada a cabo. El pequeño retraso provocado por este aumento se vió luego compensado al reducirse el tiempo empleado en la documentación del análisis y del diseño

WBS	Name	Start	Finish	Work	Duration
1	**PROYECTO**	**oct 13**	**mar 31**	**121d**	**121d**
1.1	**Estudios Previos**	**oct 13**	**dic 11**	**24d**	**44d**
1.1.1	Estudio del problema	oct 13	oct 17	5d	5d
1.1.2	Estudio de las herramientas	nov 17	dic 5	15d	15d
1.1.3	Estudio de las tecnologías	dic 8	dic 11	4d	4d
1.2	**Fase de análisis y diseño**	**dic 12**	**ene 23**	**20d**	**30d**
1.2.1	Análisis del sistema	dic 12	dic 25	10d	10d
1.2.2	Diseño del sistema	ene 9	ene 23	10d	10d
1.3	Implementación	feb 9	mar 13	25d	25d
1.4	Pruebas finales	mar 16	mar 17	2d	2d
1.5	**Documentación**	**oct 20**	**mar 31**	**50d**	**116d**
1.5.1	Documentación de la investigación	oct 20	nov 14	20d	20d
1.5.2	Documentación del análisis	dic 26	ene 8	10d	10d
1.5.3	Documentación del diseño	ene 26	feb 6	10d	10d
1.5.4	Manual técnico y de usuario	mar 18	mar 31	10d	10d

Figura 3.3: Fases del proyecto en la planificación real.

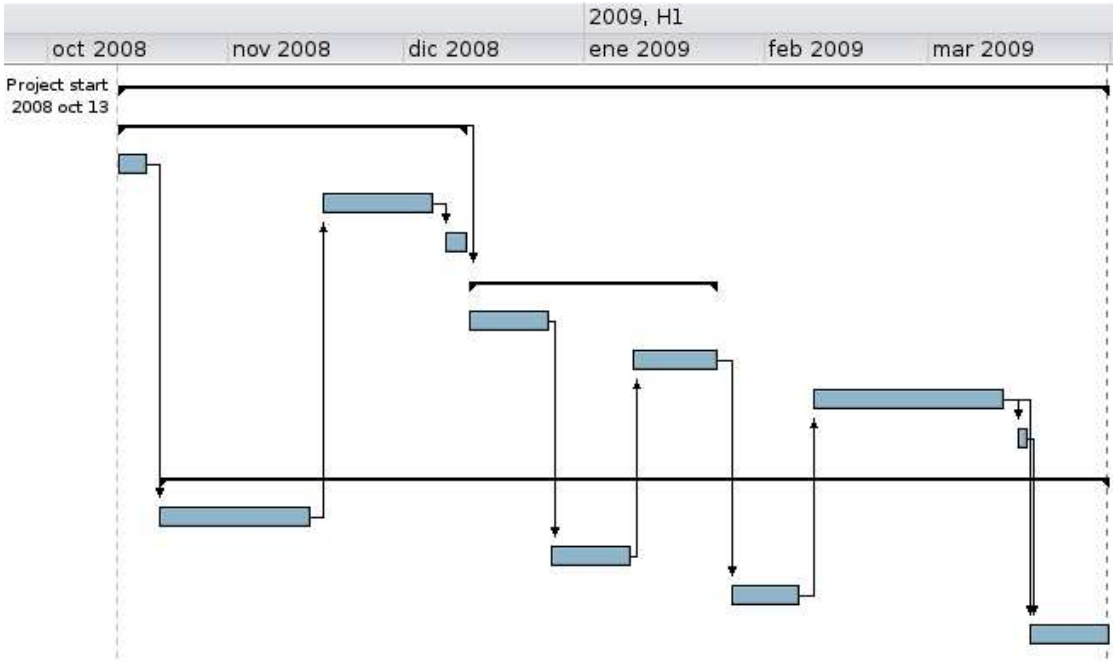

Figura 3.4: Diagrama de Gantt para la planificación real.

3.2. Presupuesto del proyecto

3.2.1. Presupuesto del proyecto

En esta sección, se detalla el presupuesto necesario para desarrollar el proyecto. Se diferencian los costes de los recursos físicos, es decir, a nivel de hardware y software,y los referentes a los recursos humanos.

Recursos físicos

Se debe tener en cuenta que como regla general en el desarrollo de aplicaciones, tanto los recursos software como los hardware están amortizados a diez proyectos.

- **Recursos hardware**

Hardware	Coste
Equipo de sobremesa con procesador Intel Core 2 Quad; 2,4 GHz; 2 Gb de RAM; HDD 500 Gb	1000 €
TOTAL (coste / 10)	100 €

- **Recursos software**
  Durante todo el desarrollo de este proyecto se ha utilizado exclusivamente *software* libre, por consiguiente el coste total de las herramientas enumeradas a continuación ha sido cero.

  - Sistema operativo GNU/Linux. Distribución *Kubuntu* 7.10.
  - Editor *NetBeans* versión 6.1 para Ubuntu.
  - Herramienta para el modelado: *Visual Paradigm community edition* para *Ubuntu*, versión libre disponible para alumnos de la *Escuela Superior de Ingeniería Informática* (ESEI) de la *Universidade de Vigo*.
  - Editor *Kile* para llevar a cabo la documentación, en LaTeX.
  - Gestor de proyectos Planner para la realización de los diagramas de *Gantt* necesarios.

Recursos humanos

Aunque nuestro sistema ha sido desarrollado por una única persona, dentro de las fases del proyecto se distingue la labor del analista, la del diseñador y la del programador que lleva a cabo las partes de implementación y pruebas.

Recursos Humanos	**Duración**	**Coste planificado**
Analista (40 €/hora)	20 días x 8 horas/día	6400 €
Diseñador (35 €/hora)	20 días x 8 horas/día	5600 €
Programador (30 €/hora)	37 días x 8 horas/día	8880 €
	Total	20880 €

Coste total del proyecto

El coste total del proyecto se calcula sumando los costes generados por los recursos humanos y por los recursos físicos.

Recursos Humanos	20880 €
Recursos Físicos	100 €
Coste Total	20980 €

Capítulo 4

Conclusiones y posibles ampliaciones

En este capítulo describiremos las conclusiones personales obtenidas una vez concluída la realización del proyecto y también las posibles ampliaciones que se pueden aplicar sobre el sistema final construido.

4.1. Conclusiones

Una vez finalizado el proyecto se extraen una serie de conclusiones muy positivas. En primer lugar, debo destacar la satisfacción por haber cumplido los objetivos establecidos en un principio, se ha conseguido crear un corrector ortográfico contextual partiendo de un etiquetador de palabras (*MrTagoo*), un preprocesador (*mrtgp* [1]) para ese etiquetador y dos técnicas de corrección ortográfica diferentes. Además, se han ampliado los tipos de errores corregidos a errores de unión de palabras diferentes y a errores de división de una palabra en varias.

El estudio de todas estas herramientas no ha sido fácil, ni, mucho menos, rápido dada la envergadura del código del que se partía. Además, la documentación era escasa y en extremos demasiado técnica, dado que se trata de proyectos en fase de desarrollo. Todas estas dificultades fueron superadas con paciencia y mucha dedicación.

La utilización del lenguaje de programación C, con el que apenas había trabajado anteriormente, me ha permitido comprobar su gran eficiencia en tiempo de ejecución y también el ahorro en memoria, necesario para las aplicaciones en las que se tratan una cantidad importante de datos. No obstante, su estudio fue duro, especialmente en lo que se refiere al manejo de punteros, y como suele ocurrir en estos casos se aprende a base de cometer muchas veces los mismos errores.

[1] Son las siglas para *MrTagoo preprocessor*.

Además, la utilización del lenguaje de marcado LaTeXpara la realización de esta documentación ha supuesto otro reto importante. Aunque conocía la existencia de esta herramienta en extremo útil para la creación de documentos técnicos y científicos, nunca lo había empleado. El resultado final ha sido muy positivo y los conocimientos adquiridos me han permitido desenvolverme con suficiencia en este ámbito.

Durante el desarrollo se han encontrado numerosos problemas, sobre todo referentes a la codificación de caracteres utilizada. También nos han surgido problemas a raíz de la incorporación, en ocasiones, de un número elevado de correcciones para una palabra desconocida, pues el etiquetador no estaba pensado para recibir esa cantidad de alternativas.

A nivel personal, el camino recorrido durante la realización del proyecto ha sido largo y en ocasiones tedioso, sobre todo debido la falta de experiencia previa en la utilización de las tecnologías empleadas, así como en el estudio del código existente del que se partía. Aunque en un principio se pueda pensar que el no tener que partir de cero facilita las cosas, en ciertos momentos las hace más complicadas, pues las decisiones que podamos tomar deben adaptarse a lo ya realizado. Sin embargo, una vez terminado el proyecto; la satisfacción de ver cumplido lo planificado es mayor.

4.2. Posibles ampliaciones

Se pueden incorporar una serie de ampliaciones en nuestro proyecto con el fin de obtener unos mejores resultados en la corrección:

- Aumentar el diccionario empleado por el corrector para que incluya también los verbos conjugados y combinados con todas los pronombres enclíticos que puedan acompañarlos. De este modo, se podría corregir un error ortográfico en una forma verbal con un pronombre enclítico asociado. En el sistema actual no se solucionan los problemas de este tipo.

- Reimplementar el preprocesador para eliminar la restricción que impide anidar listas de alternativas en los retículos, con el objetivo de reducir los costes en términos de espacio y tiempo en el momento de ejecutar el algoritmo de Viterbi sobre éstos.

- Ampliar los tipos de errores ortográficos a tratar. El sistema actual obtiene las correcciones de distancia mínima para una palabra de forma individual, uniéndola con su predecesora y con su sucesora. Podríamos ampliar el sistema para que comprobase las correcciones para la unión de todas ellas, es decir, la palabra precesora, la actual y la sucesora.

Parte II

Manual Técnico

Capítulo 5

Análisis

La fase de análisis de un proyecto *software* es la más importante en todo el proceso de desarrollo, en esta fase se capturan los requisitos del sistema y se modela el problema a resolver.

5.1. Sistema implementado

Nuestro sistema va a ser a su vez un subsistema dentro del *preprocesador morfológico* [4]. Constituirá un nuevo módulo dentro de éste y será el último en ejecutarse antes de llevar a cabo la etiquetación. Por tanto, no habrá interacción directa entre el sistema desarrollado y el usuario.

En lineas generales, el funcionamiento general del entorno de etiquetación existente es el siguiente: si las oraciones se pasan en un fichero XML se utiliza un *parser* SAX encargado de extraerlas una a una. Una vez obtenida la oración en texto plano, se construye en memoria una representación de la oración, que será manipulada para realizar las operaciones de fusión y segmentación necesarias. El resultado final de este proceso (en el que intervienen varios módulos especializados en contracciones, locuciones, etc.) es completado con las etiquetas y lemas de todas las palabras que no hayan sido etiquetadas por los módulos. Si hay alguna palabra desconocida, esto es, para la que no se conozcan sus posibles etiquetas y lemas, se le asignara "$*$" como única etiqueta y "$*$" como único lema posibles. Es aquí el momento justo donde se sitúa nuestro sistema que se aplicará sobre aquellas palabras que tengan como etiqueta y lema un "$*$", con el fin de obtener las alternativas de corrección para cada una de ellas, y siempre que sea posible se añaden las correcciones obtenidas a la estructura en memoria que representa la oración. Finalmente, se construye un nuevo fichero XML, que contiene la misma información que el fichero original, pero al que se le añade la información morfosintáctica desglosada dentro de cada oración.

El lenguaje de implementación será *C*, dado que partimos de dos herramientas ya existentes y desarrolladas en este lenguaje. No obstante, C es la opción más adecuada para este tipo de proyectos al permitir el control absoluto de la gestión de memoria y de la ejecución al programador.

Se empleará la librería *glib* que además de asegurar la portabilidad del código, nos proporciona numerosas estructuras de datos. La librería *libxml2* será, también, una herramienta fundamental en nuestra aplicación para manejar los archivos *xml* de configuración del sistema y de entrada y salida de datos.

Para realizar el análisis y diseño se seguirá la notación UML que si bien fue creada como complemento para la programación orientada a objetos, al tratarse de un objeto de modelado, es posible aplicarla al paradigma de programación estructurada, que es el seguido en este proyecto.

La arquitectura sobre la que se realizará el desarrollo es GNU/Linux para garantizar un entorno robusto y con una alta fiabilidad.

5.2. Estructuras de datos utilizadas por el *preprocesador*

Para el desarrollo de nuestro proyecto vamos a utilizar una serie de estructuras importantes en el entorno de etiquetación:

- NADFA(Numbered-Acyclic-Deterministic-Finite-Automaton)
  Esta es la estructura utilizada para almacenar los diccionarios del lenguaje. Es una implementación del modelo del diccionario compacto propuesto por Graña [5] basada en la utilización de los AFDAN y que se explicó en la sección 1.4.3.

- TOKEN
  Las unidades léxicas, generalmente una sola palabra aunque pueden ser más, se representan en memoria mediante el tipo *sentence_token*:

```
struct sentence_token {
guint8 type;
gchar *word;
GSList *tag;
GSList *lemma;
GSList *next;
GSList *prev;
gboolean tagged;
GSList *mate;
};
```

El elemento type especifica ante que tipo de token nos hayamos. Se diferencian cinco tipos de token:

- SENTENCE_INITIAL_TOKEN: Es el primer token de la oración.
- SENTENCE_FINAL_TOKEN: El último token de la oración.
- SENTENCE_ALTERNATIVE_INIT_TOKEN: Anuncia el comienzo de varias ramas con distintas alternativas de segmentación.
- SENTENCE_ALTERNATIVE_FINAL_TOKEN: Marca el fin de las alternativas de segmentación.
- SENTENCE_NORMAL_TOKEN: Es el que se usa para representar cada una de las unidades léxicas que forman la oración.

Los cuatro primeros tipos pueden considerarse meramente estructurales, ya que no contienen información real de la oración. En los tokens de los tipos estructurales las listas de etiquetas y lemas están vacías, y el elemento *word* contiene de nuevo el nombre del tipo de token.

Cada token de tipo SENTENCE_NORMAL_TOKEN representa una unidad de etiquetación. Como se puede ver, cada token contiene un elemento *word* en el que se guarda la palabra etiquetada. Además, incluye una lista de etiquetas y lemas posibles. Cabe destacar que estas dos listas siempre tienen la misma longitud, ya que para cada etiqueta siempre hay un lema. Las etiquetas y lemas en las mismas posiciones de sus listas van emparejadas (la primera etiqueta con el primer lema, etc.).

Además, existe una lista que referencia a los siguientes tokens de la oración y otra que referencia a los anteriores. Normalmente, estas listas contendrán un sólo elemento, salvo que el token sea un comienzo o final de alternativa. El elemento *tagged* es sólo un indicador para saber que el token está etiquetado. Por último, la lista *mate*, permite enlazar cada token con la oración original. Esto se hace mediante dos coordenadas (x,y) que continen las posiciones de comienzo y final de la posición de la oración original a la que respresenta este token.

- SENTENCE

  Una oración se define mediante el tipo *struct sentence* , que incluye la oración original en texto plano y una referencia al primer token de la oración, que es tipo SENTENCE_INITIAL_TOKEN.

```
struct sentence{
struct sentence_token *first;
```

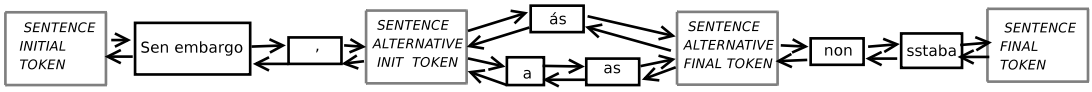

Figura 5.1: Ejemplo de una representación para una frase.

```
gchar *org_sentence;
};
```

Un ejemplo de representación de una frase, a modo ilustrativo, utilizando la estructura *token* sería el que se muestra en la figura 5.1.

El preprocesador impone como restricción que no existan una lista de alternativas en el interior de otra o lo que es lo mismo no permite anidamientos de alternativas.

5.3. Diagrama de casos de uso global

En esta sección se presentará el diagrama de casos de uso general del sistema y se irán detallando los casos de uso para especificar los objetivos y requisitos del sistema.

En el diagrama de casos de uso de la figura 5.2 se refleja el funcionamiento del sistema completo. La interacción con el usuario se realiza a través del preprocesador, y es éste el encargado de lanzar nuestro módulo o subsistema. Aunque el caso de uso *Analizar texto* ya exista previamente, lo incluímos aquí para ofrecer una visión global del funcionamiento de nuestro sistema. En la figura 1.6 se mostraba una visión global del funcionamiento del preprocesador, ahora nuestro módulo a desarrollar se aplicará en último lugar.

5.4. Caso de uso: Aplicar corrector

Este caso de uso es el encargado de aplicar la corrección sobre una frase previamente preprocesada. Es el principal caso de uso de nuestro sistema y por ello se ha considerado necesario dividirlo en otros más sencillos para explicarlo detalladamente, como se puede ver en el diagrama 5.3.

Aplicar corrección(escenario principal)

Descripción: Este escenario describe el proceso mediante el cual se aplica el corrector ortográfico sobre una frase recibida desde el *preprocesador*.

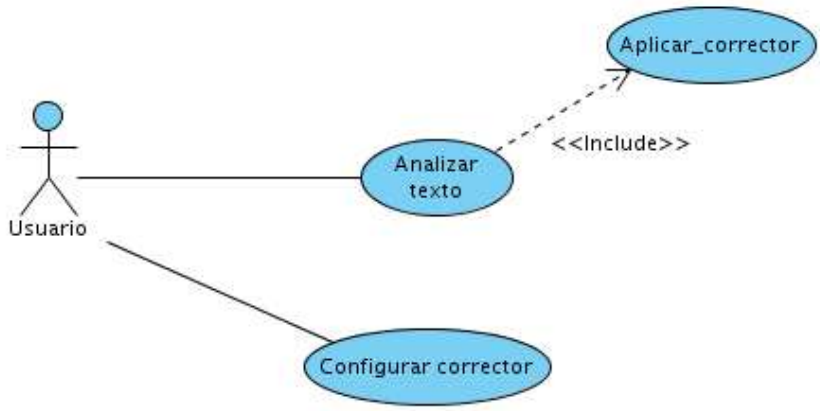

Figura 5.2: Diagrama de casos de uso global.

Precondiciones:

- La frase tiene que haber sido tratada previamente por el *preprocesador*

Flujo principal:

1. El *preprocesador* se encarga de lanzar la corrección, enviando la frase a corregir.

2. El sistema, para cada palabra que aún no tiene etiqueta aplica la búsqueda de correcciones global (se ejecuta aquí el caso de uso extendido *Buscar corección global*)*[Alternativa 1]*.

3. Una vez que se ha obtenido una lista con las correcciones (de tipo global o regional), el sistema debe enlazar esas correcciones en la estructura de la frase original(se ejecuta aquí el caso de uso incluído *Enlazar correcciones*).

4. El sistema comprueba si en las correcciones incluídas en la frase existe alguna posible contracción (se ejecuta aquí el caso de uso incluído *Revisar nuevas contracciones*).

5. El sistema vuelve al paso 2 mientras existan palabras sin etiqueta.

6. El sistema comprueba la existencia de nuevas locuciones en la frase con las correcciones (se ejecuta aquí el caso de uso incluído *Revisar nuevas expresiones.*)

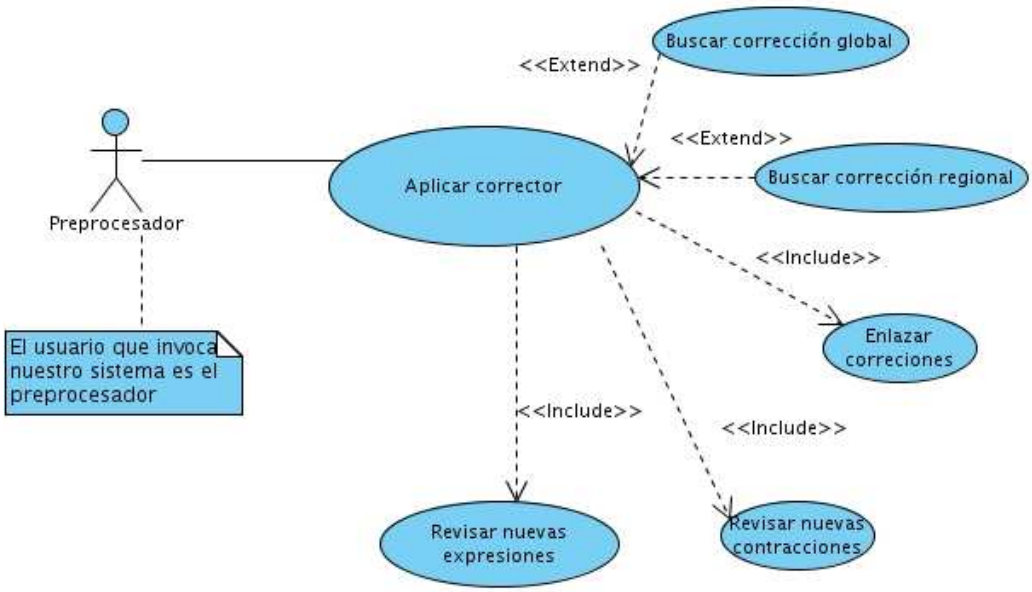

Figura 5.3: Diagrama de casos de uso para *Aplicar corrector*.

Flujo alternativo:

- *Alternativa 1:* Si el método de corrección a aplicar es el global entonces se ejecuta aquí el caso de uso extendido *Buscar corrección regional*.

Poscondiciones: Para cada palabra sin etiquetar y con posibles correcciones se han incorporado éstas a la frase original.

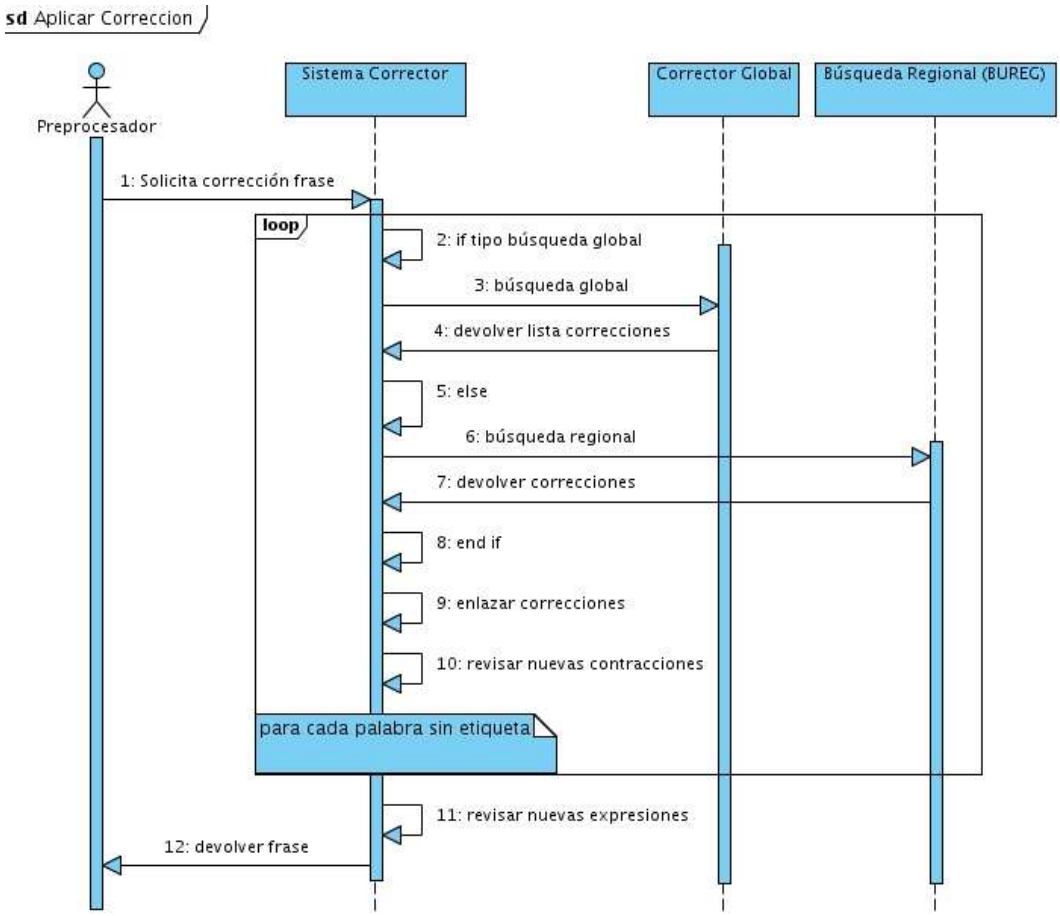

Figura 5.4: Diagrama de secuencia del análisis: Aplicar corrección

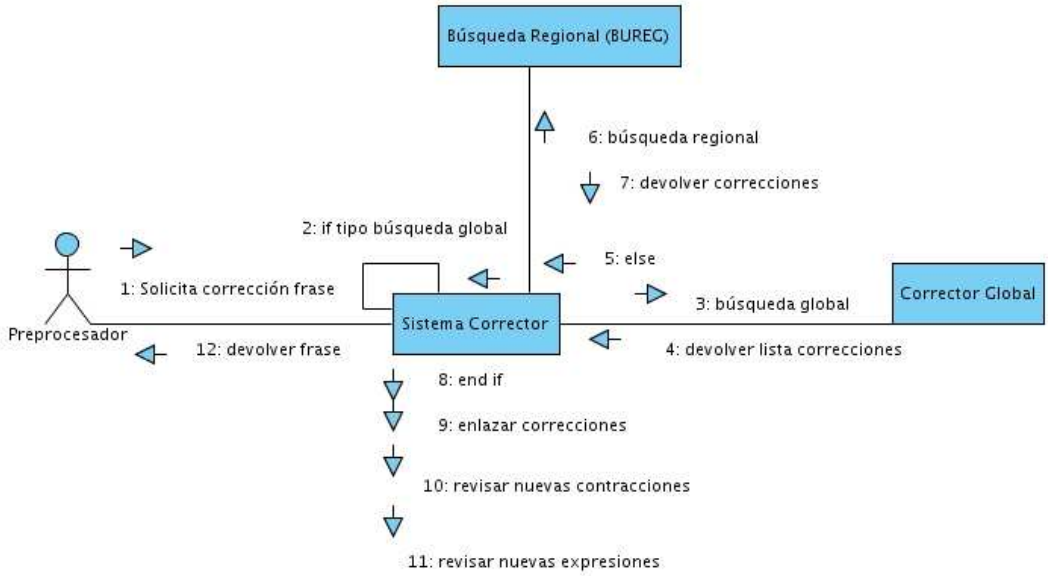

Figura 5.5: Diagrama de colaboración del análisis: Aplicar corrección

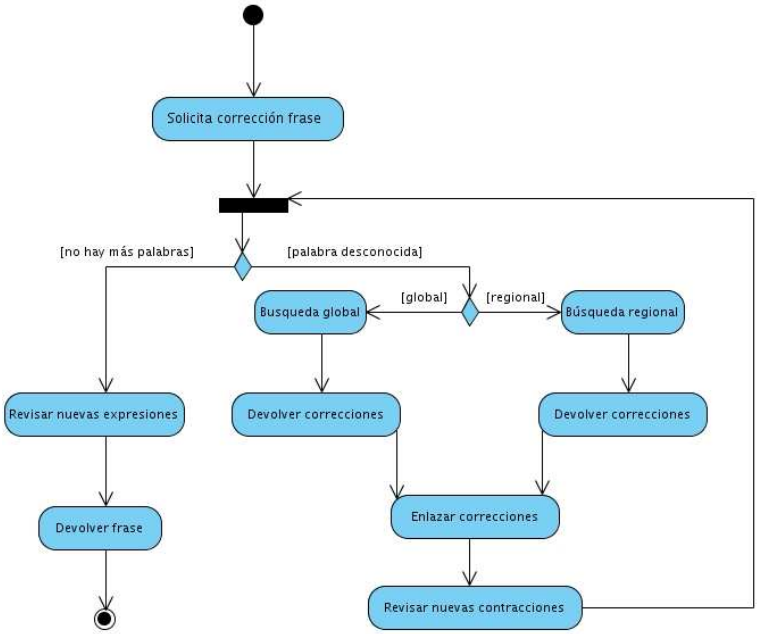

Figura 5.6: Diagrama de actividades del análisis: Aplicar corrección

5.4.1. Aplicar corrección: Buscar corrección global

Descripción: Este caso de uso obtiene las posibles correcciones para una palabra desconocida, aplicando para ello la técnica de corrección global desarrollada por Savary [12], que se describe en el algoritmo 1.

Precondiciones:

- Este caso de uso extiende siempre del caso de uso *Aplicar corrección*.
- El usuario ha configurado el corrector indicando como tipo de búsqueda el método global de Savary .
- La palabra no puede estar etiquetada.

Flujo principal:

1. El sistema se encarga de aplicar el método de búsqueda global sobre la palabra desconocida.
2. Se incorporan los resultados obtenidos a la lista de correcciones para esa palabra, guardando la distancia de edición mínima para los resultados.
3. Se aplica el método de búsqueda global sobre la unión de la palabra previa y la palabra desconocida.
4. Si la distancia de edición es menor que la previa se sobrescribe la lista con los nuevos resultados.*[Alternativa 1]*.
5. Se encarga aplica el método de búsqueda global sobre la unión de la palabra desconocida y la palabra posterior.
6. Si la distancia de edición es menor que la previa se sobrescribe la lista con los nuevos resultados.*[Alternativa 1]*.
7. Se devuelven la lista con las posibles correcciones cuya distancia de edición respecto a la palabra desconocida es mínima.

Flujo alternativo:

- *Alternativa 1:* Si la distancia de edición es igual se añaden los nuevos resultados a los ya existentes. En caso contrario no se hace nada.

Poscondiciones: Se devuelve una lista de correcciones candidatas para la palabra desconocida, que podría estar vacía.

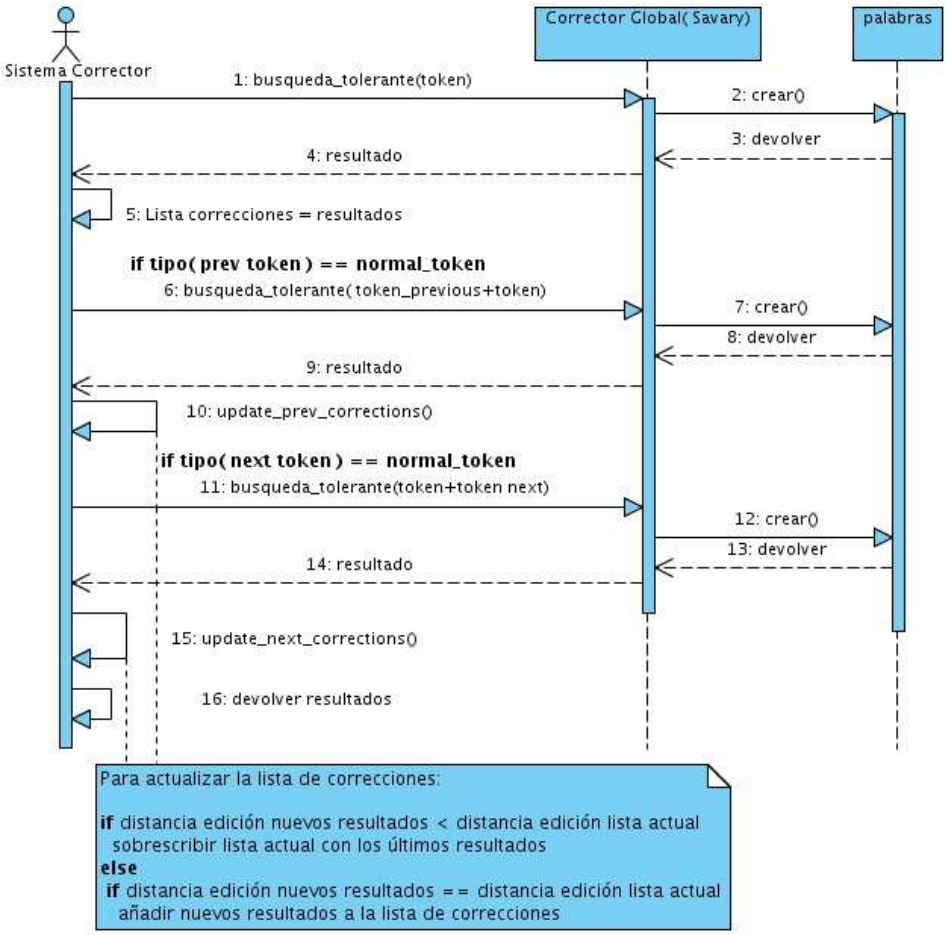

Figura 5.7: Diagrama de secuencia del análisis: Buscar corrección global

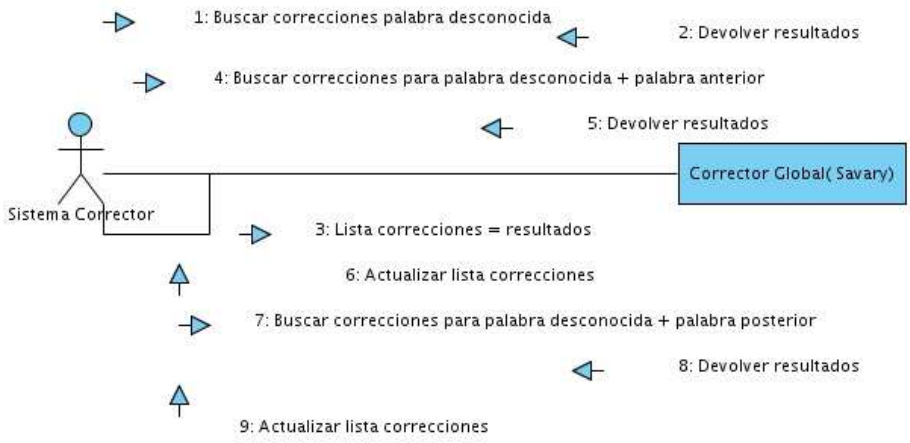

Figura 5.8: Diagrama de colaboración del análisis: Buscar corrección global

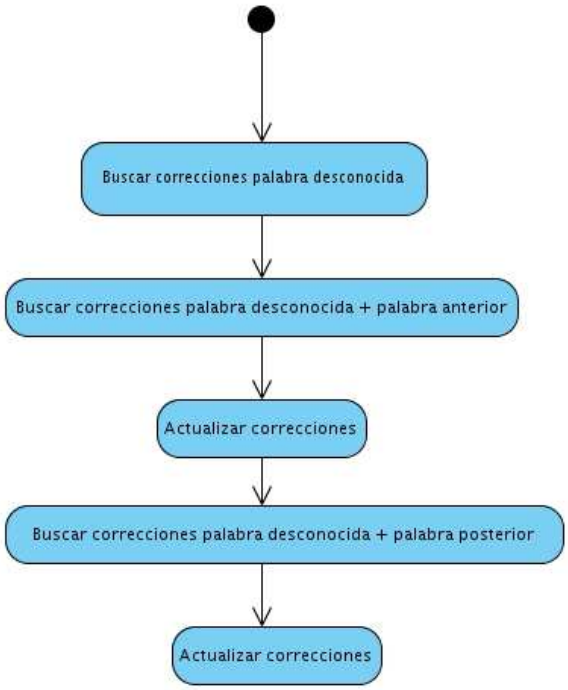

Figura 5.9: Diagrama de actividades del análisis: Buscar corrección global

5.4.2. Aplicar corrección: Buscar corrección regional

Descripción: Este caso de uso obtiene las posibles correcciones para una palabra desconocida, aplicando para ello la técnica de corrección regional [15].

Precondiciones:

- Este caso de uso extiende siempre del caso de uso *Aplicar corrección*.
- El usuario ha configurado el corrector indicando como tipo de búsqueda el método de corrección regional.
- La palabra no puede tener una etiqueta.

Flujo principal:

1. Se aplica el método de búsqueda regional sobre la palabra desconocida.
2. Se incorpora los resultados obtenidos a la lista de correcciones para esa palabra, guardando la distancia de edición mínima para los resultados.
3. Se aplica el método de búsqueda regional sobre la unión de la palabra previa y la palabra desconocida, buscando errores multipalabra.
4. Si la distancia de edición es menor que la previa se sobrescribe la lista con los nuevos resultados.*[Alternativa 1]*.
5. Se aplica el método de búsqueda regional sobre la unión de la palabra desconocida y la palabra posterior.
6. Si la distancia de edición es menor que la previa se sobrescribe la lista con los nuevos resultados.*[Alternativa 1*].
7. Se devuelven la lista con las posibles correcciones con distancia de edición mínima respecto a la palabra desconocida.

Flujo alternativo:

- *Alternativa 1:* Si la distancia de edición es igual se añaden los nuevos resultados a los ya existentes. Si la distancia es mayor no se hace nada.

Poscondiciones: Se devuelve una lista de correcciones candidatas para la palabra desconocida, que podría estar vacía.

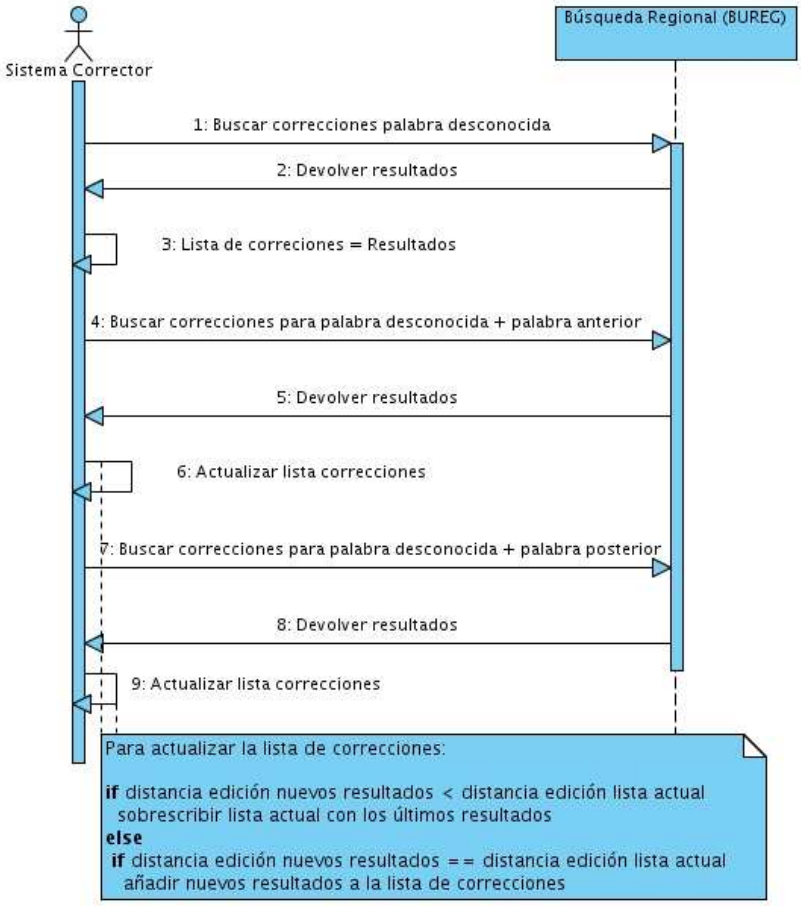

Figura 5.10: Diagrama de secuencia del análisis: Buscar corrección regional

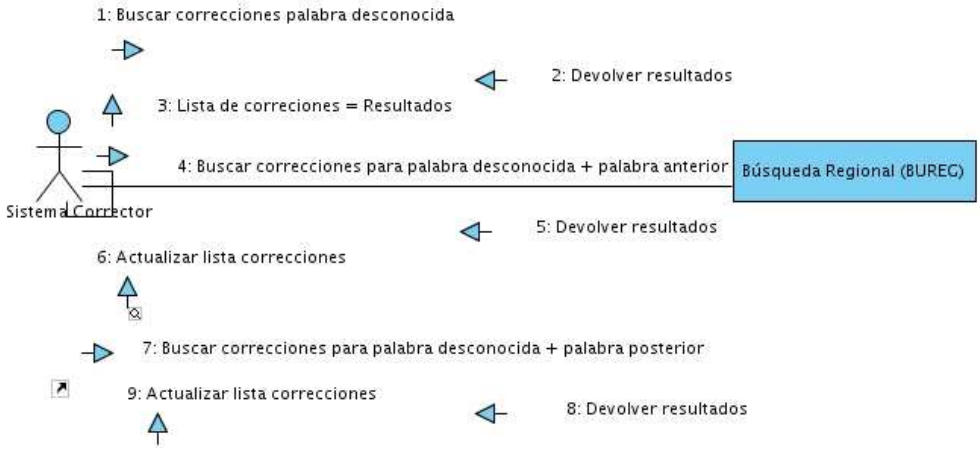

Figura 5.11: Diagrama de colaboración del análisis: Buscar corrección regional

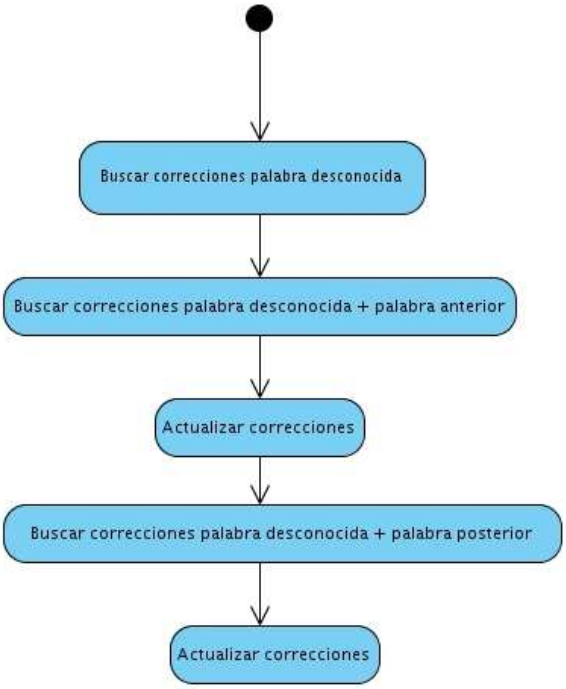

Figura 5.12: Diagrama de actividades del análisis: Buscar corrección regional

5.4.3. Aplicar corrección: Enlazar correcciones

Descripción: Este caso de uso se encarga de integrar una lista de correcciones para una palabra desconocida en la estructura original recibida desde el preprocesador. Este caso de uso se ejecuta siempre desde el caso de uso *Aplicar corrección*.

Precondiciones:

- Se ejecuta cada vez que se examina una palabra no etiquetada.

Flujo principal:

1. Si existen correcciones candidatas para la palabra, entonces se unirán las posibles alternativas siguiendo la misma estructura que utiliza el preprocesador.
2. Se comprueba si en la lista de correcciones no existen correcciones multipalabra *[Alternativa 1]*.
3. Si la palabra desconocida no se incluye como posible candidata y solo existe una corrección en la lista, se realiza la sustitución directamente en la frase original *[Alternativa 2]*.

Flujo alternativo:

- *Alternativa 1:* Si existen correciones multipalabra con la palabra anterior a la desconocida se invoca el caso de uso extendido *Enlazar correciones multipalabra previa*. Si existen correcciones multipalabra con la palabra posterior, se aplica el caso de uso extendido *Enlazar correcciones multipalabra posterior*.
- *Alternativa 2:* Si hay más de una corrección posible se debe crear una nueva alternativa dentro de la frase con todas las opciones.

Poscondiciones: Se devuelve un apuntador al lugar en donde se ha realizado el cambio en la frase.

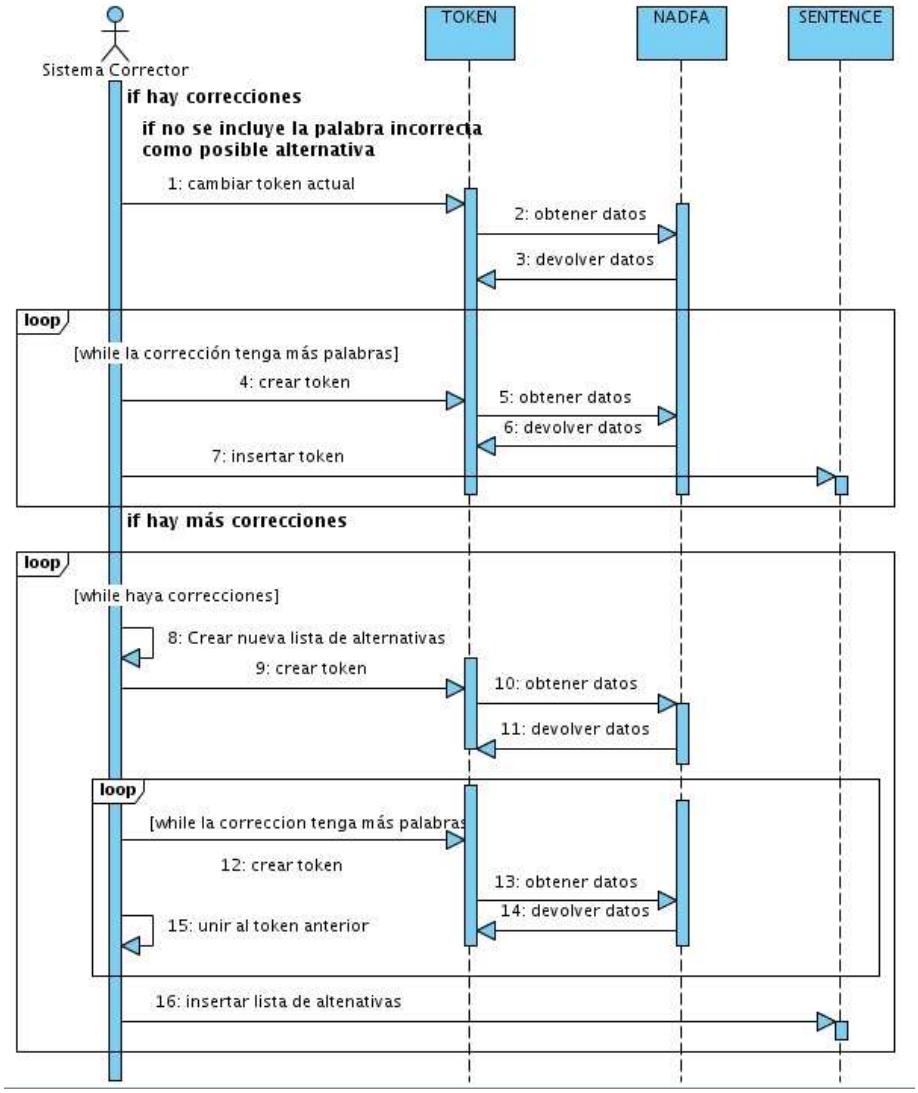

Figura 5.13: Diagrama de secuencia del análisis: Enlazar correcciones

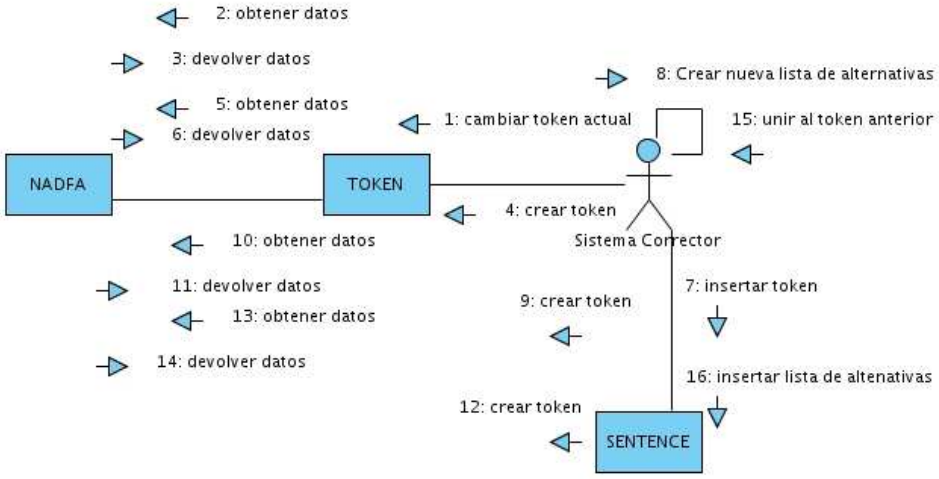

Figura 5.14: Diagrama de colaboración del análisis: Enlazar correcciones

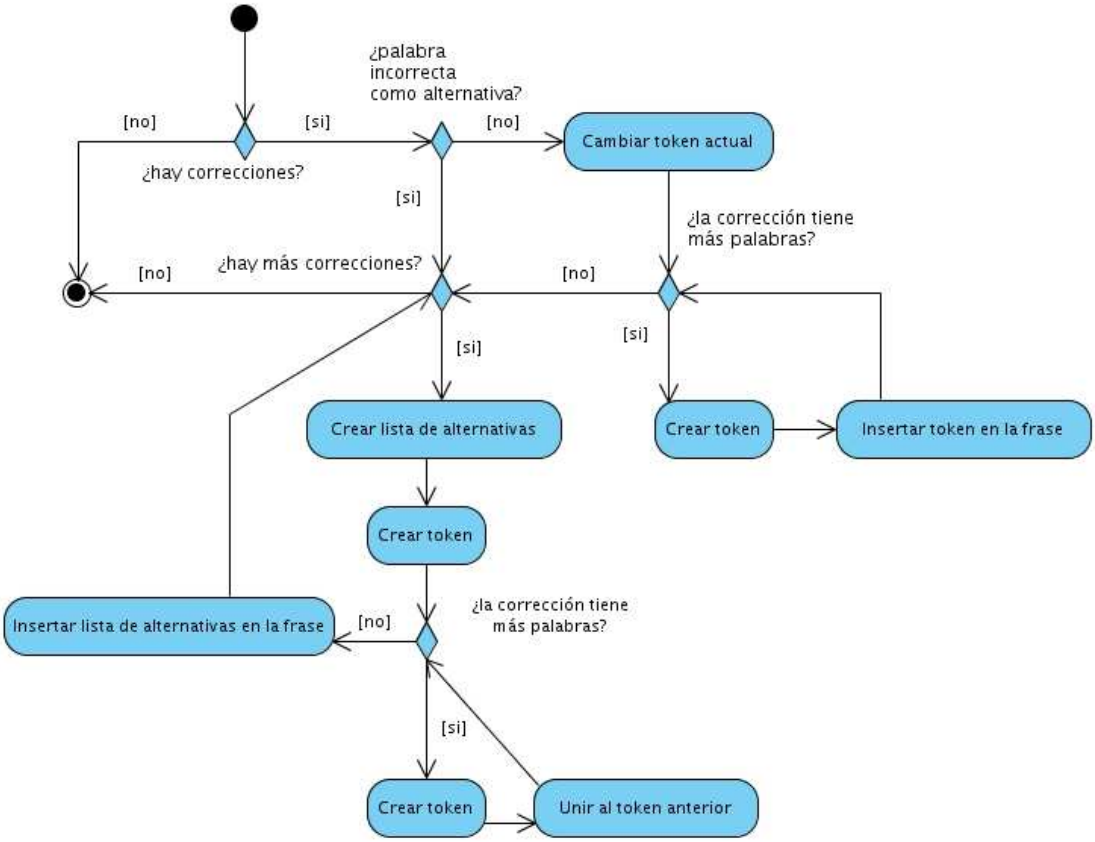

Figura 5.15: Diagrama de actividades del análisis: Enlazar correcciones

5.4.4. Aplicar corrección: Revisar nuevas contracciones

Descripción: Este caso de uso examina si alguna de las correcciones realizadas coincide con alguna de las contracciones del lenguaje.

Precondiciones:

- Este caso de uso se aplica una vez que las correcciones a una palabra se han integrado en la frase.

Flujo principal:

1. Si existen correciones a la palabra desconocida se examinan.
2. Se comprueba si la corrección que coincide con alguna contracción se encontraba dentro de una lista de alternativa o bifuración.*[Alternativa 1]*
3. Se crea la estructura necesaria para añadir la contracción como una nueva corrección.
4. Se enlaza la nueva corrección a la lista de alternativas.
5. Si hay más alternativas en la lista se vuelve al paso 2.

Flujo alternativo:

- *Alternativa 1:* Si la correción no forma parte de una alternativa significa que había una sola corrección posible y no se incluye la palabra desconocida como alternativa. En este caso, se aplica el módulo de detección de contracciones existente en el preprocesador.

Poscondiciones: Si existía alguna posible contracción en las nuevas contracciones se ha incorporado como tal.

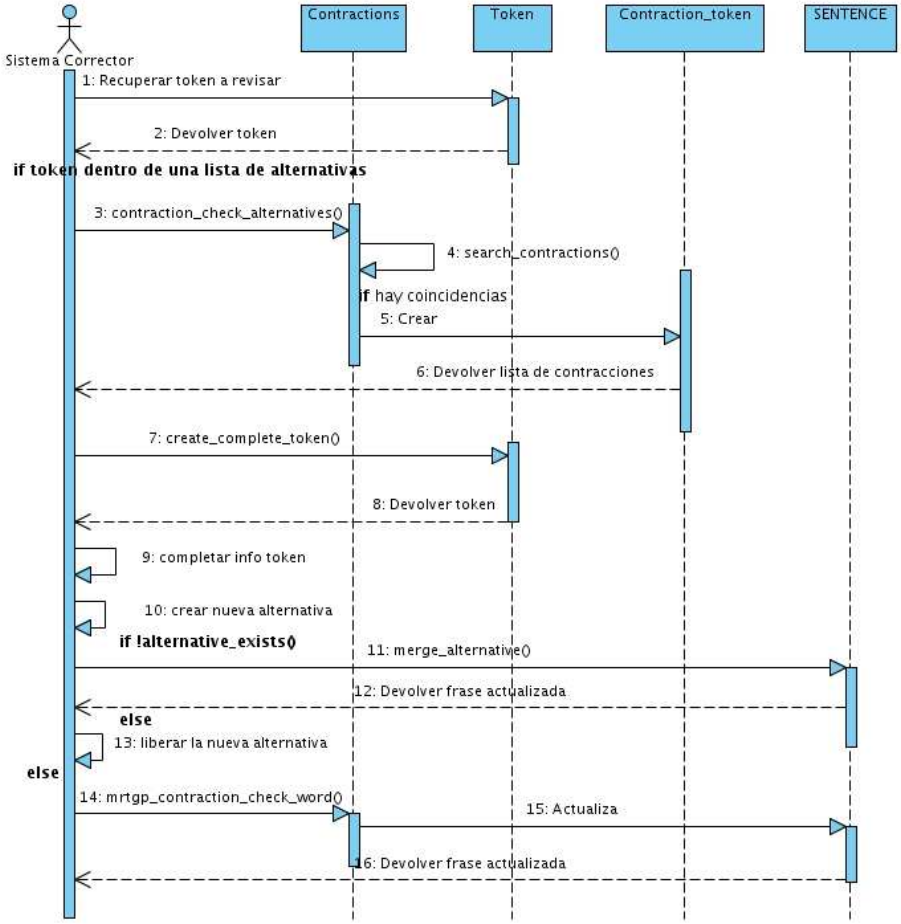

Figura 5.16: Diagrama de secuencia del análisis: Revisar nuevas contracciones

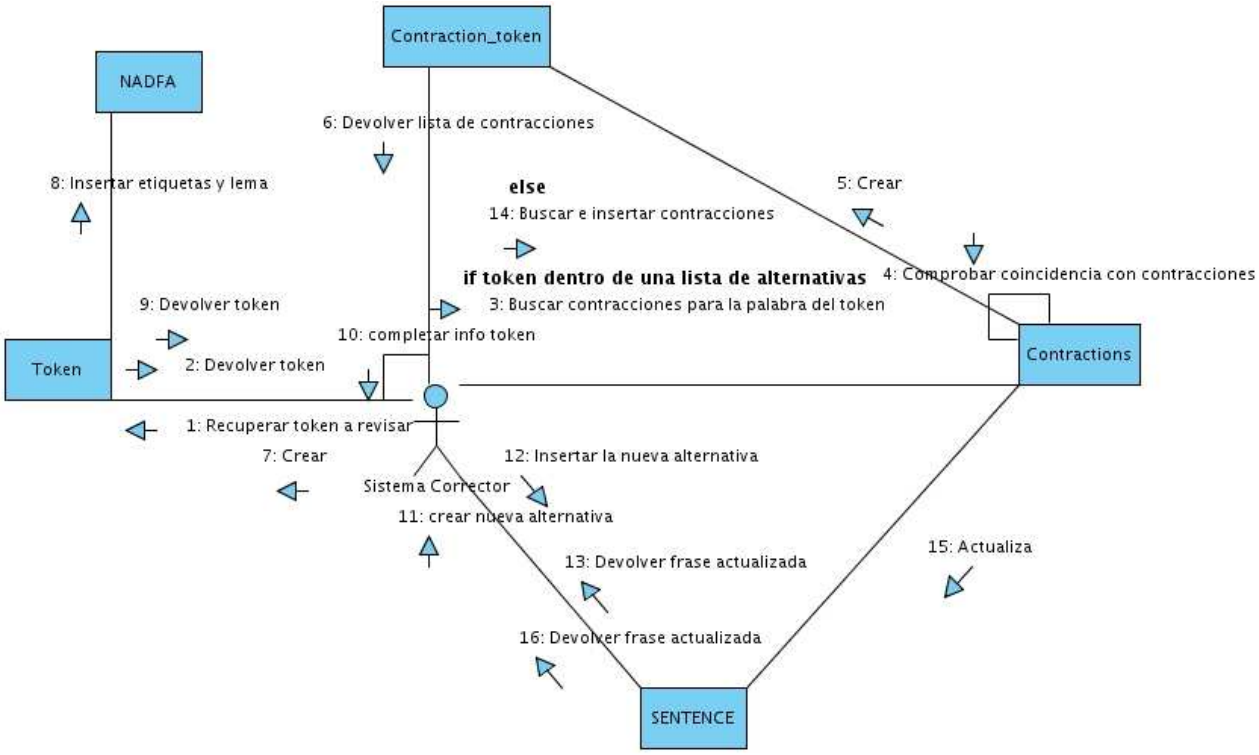

Figura 5.17: Diagrama de colaboración del análisis: Revisar nuevas contracciones

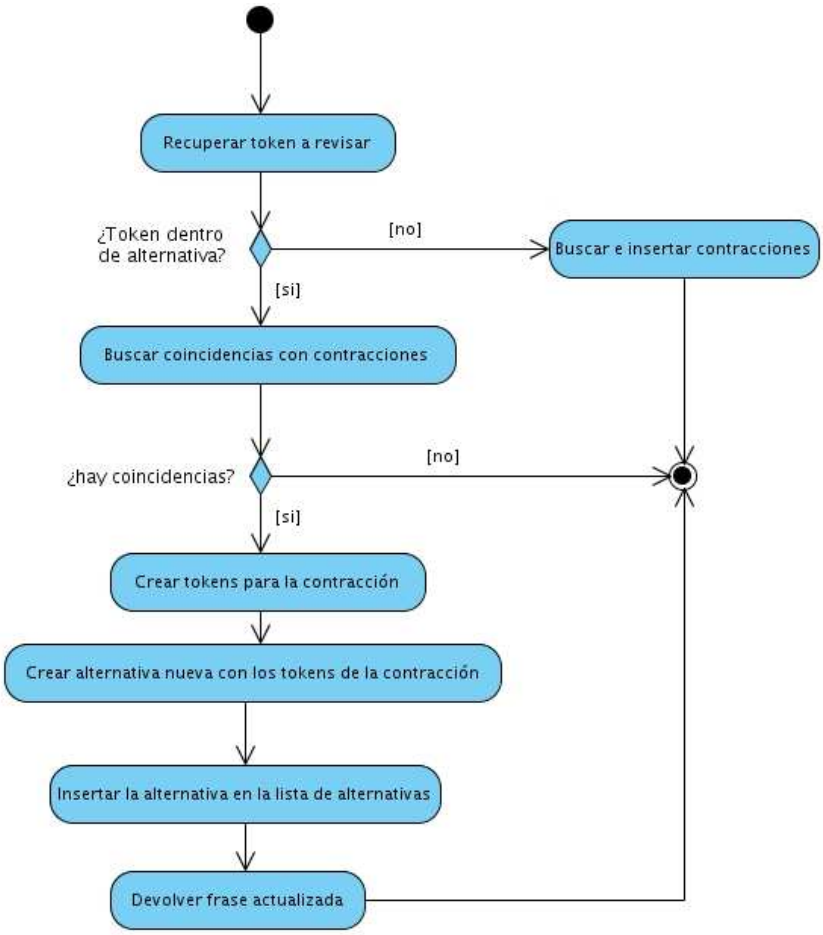

Figura 5.18: Diagrama de actividades del análisis: Revisar nuevas contracciones

5.4.5. Aplicar corrección: Revisar nuevas expresiones

Descripción: Este caso de uso se encarga de revisar la existencia de posibles locuciones como resultado de las nuevas correcciones.

Precondiciones:

- Existía alguna palabra sin etiqueta en la frase, es decir, ha habido cambios en la frase.

Flujo principal:

1. Se aplica el módulo de detección de locuciones seguras del preprocesador.
2. Se aplica el módulo de detección de locuciones no seguras del preprocesador.

Poscondiciones: Si las correcciones aplicadas coincidían con alguna locución, esta ha sido añadida a la lista de alternativas.

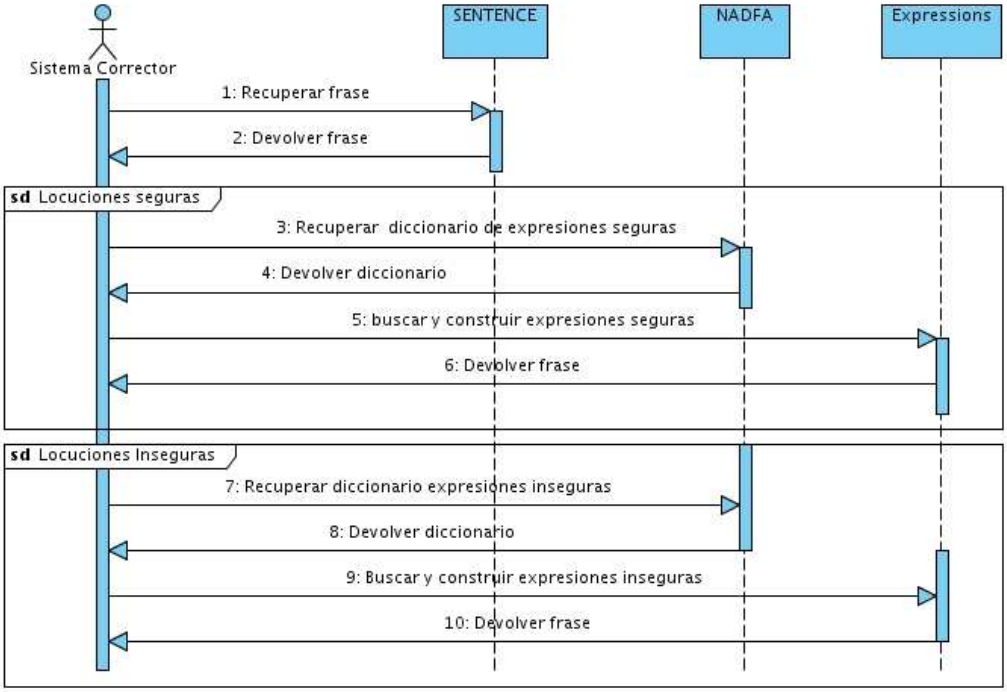

Figura 5.19: Diagrama de secuencia del análisis: Revisar nuevas expresiones

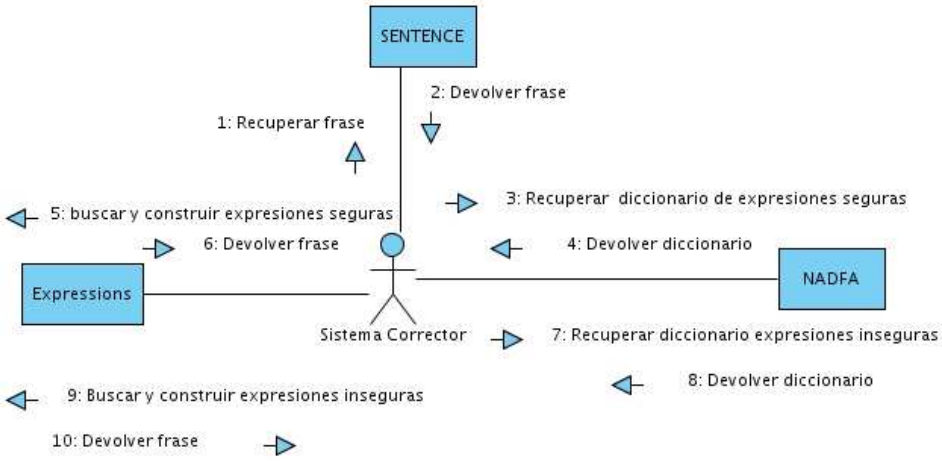

Figura 5.20: Diagrama de colaboración del análisis: Revisar nuevas expresiones

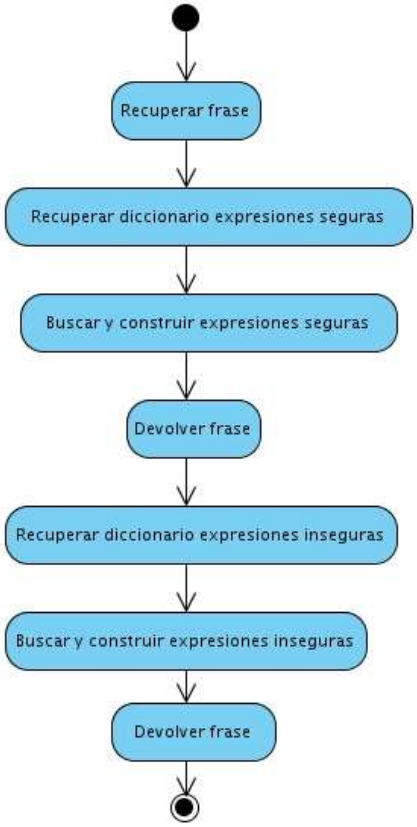

Figura 5.21: Diagrama de actividades del análisis: Revisar nuevas expresiones

5.5. Caso de uso: Configurar corrector

Este caso de uso es el encargado de cargar los parámetros de configuración del corrector ortográfico. Estos parámetros se encuentrán almacenados en un fichero en formato XML. El usuario es el encargado de introducir sus preferencias en el fichero mencionado. Una vez que se ejecute el preprocesador, si se ha activado la opción para utilizar el corrector ortográfico, es decir, nuestro sistema, entonces se cargará la información procedente del fichero. Se utilizará un *parser* SAX para leer su contenido y almacenarlo en memoria.

Descripción: Este caso de uso tiene como objetivo establecer los parámetros de configuración del corrector ortográfico.

Precondiciones:

- Durante el lanzamiento del preprocesador la opción para que se ejecute el módulo de corrección tiene que estar activada.
- La ruta donde se encuentra el archivo de configuración ha de ser correcta.

Flujo principal:

1. El preprocesador llama a un analizador de ficheros XML para cargar el contenido del archivo.
2. El analizador lee el contenido del archivo y lo almacena.
3. Mientras no se llegue al final del archivo se vuelve al paso anterior.
4. El analizador devuelve al preprocesador la información contenida en el archivo.

Poscondiciones: El preprocesador ahora conoce los parámetros para configurar la corrección.

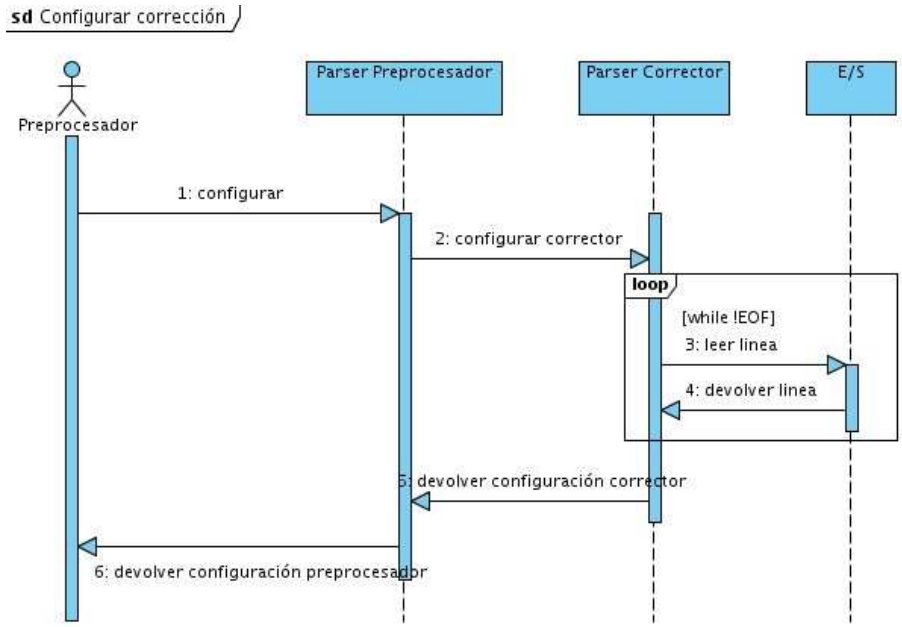

Figura 5.22: Diagrama de secuencia del análisis: Configurar corrección

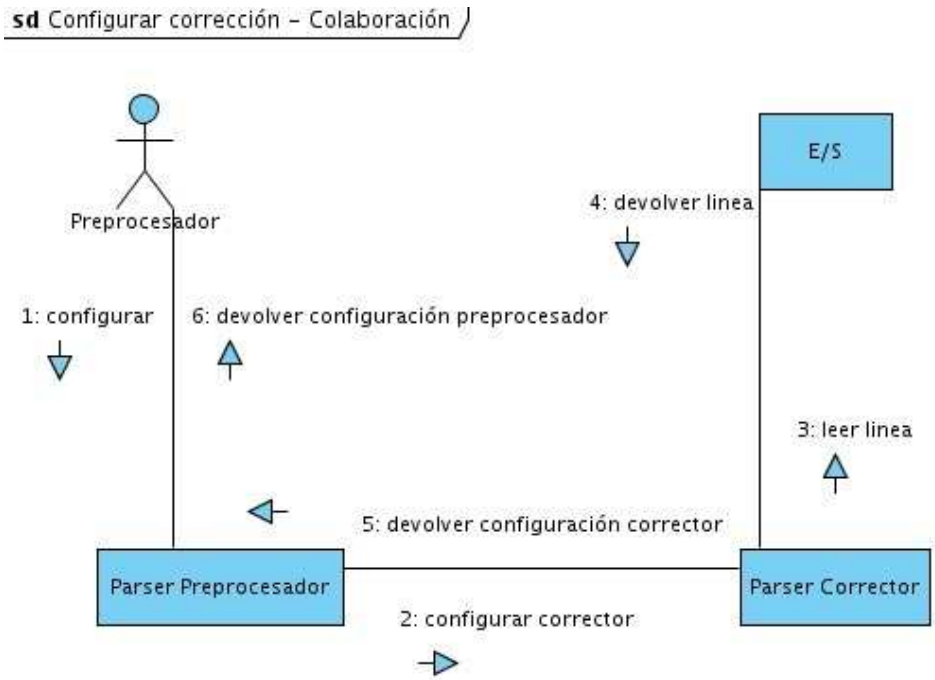

Figura 5.23: Diagrama de colaboración del análisis: Configurar corrección

5.6. Diagrama de clases

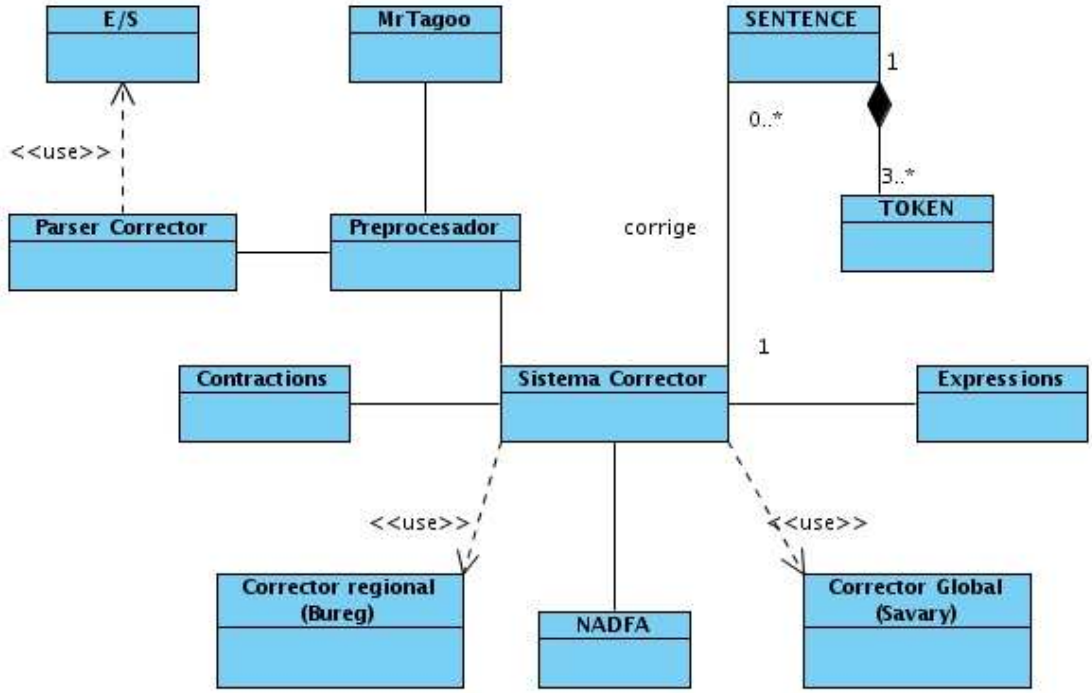

Figura 5.24: Diagrama de clases del análisis.

Capítulo 6

Diseño

En esta fase del desarrollo, se detalla como se van a construír los requisitos iniciales descritos durante la fase de análisis. Para ello, se realizará el diagrama de clases; se describirá con detalle cada una de esas clases, sus funciones y atributos; y se realizarán los diagramas de secuencia propios de esta etapa.

Es importante aclarar que dado que nuestro proyecto no ha sido implementado con un lenguaje de programación orientado a objetos, la correspondencia con las clases no va a ser directa. Por este motivo, asociaremos las clases con entidades lógicas empleadas por nuestro sistema, generalmente, se van a corresponder con módulos o estructuras de datos.

6.1. Diagrama de clases

El diagrama de clases representa el funcionamiento global del sistema, es decir, la forma en que éste se relaciona con las unidades lógicas existentes previamente.

Las clases desarrolladas en este proyecto son el *Sistema Corrector* y el *Parser Corrector*, sin embargo, para comprender el funcionamiento del sistema completo se hace necesario la representación de todas las clases utilizadas. A continuación, se hará un breve resumen de las clases empleadas con sus métodos más destacados y se explicará con detalle aquellas clases que debemos implementar en nuestro sistema. En la figura 6.1, se muestra el diagrama de clases general del diseño.

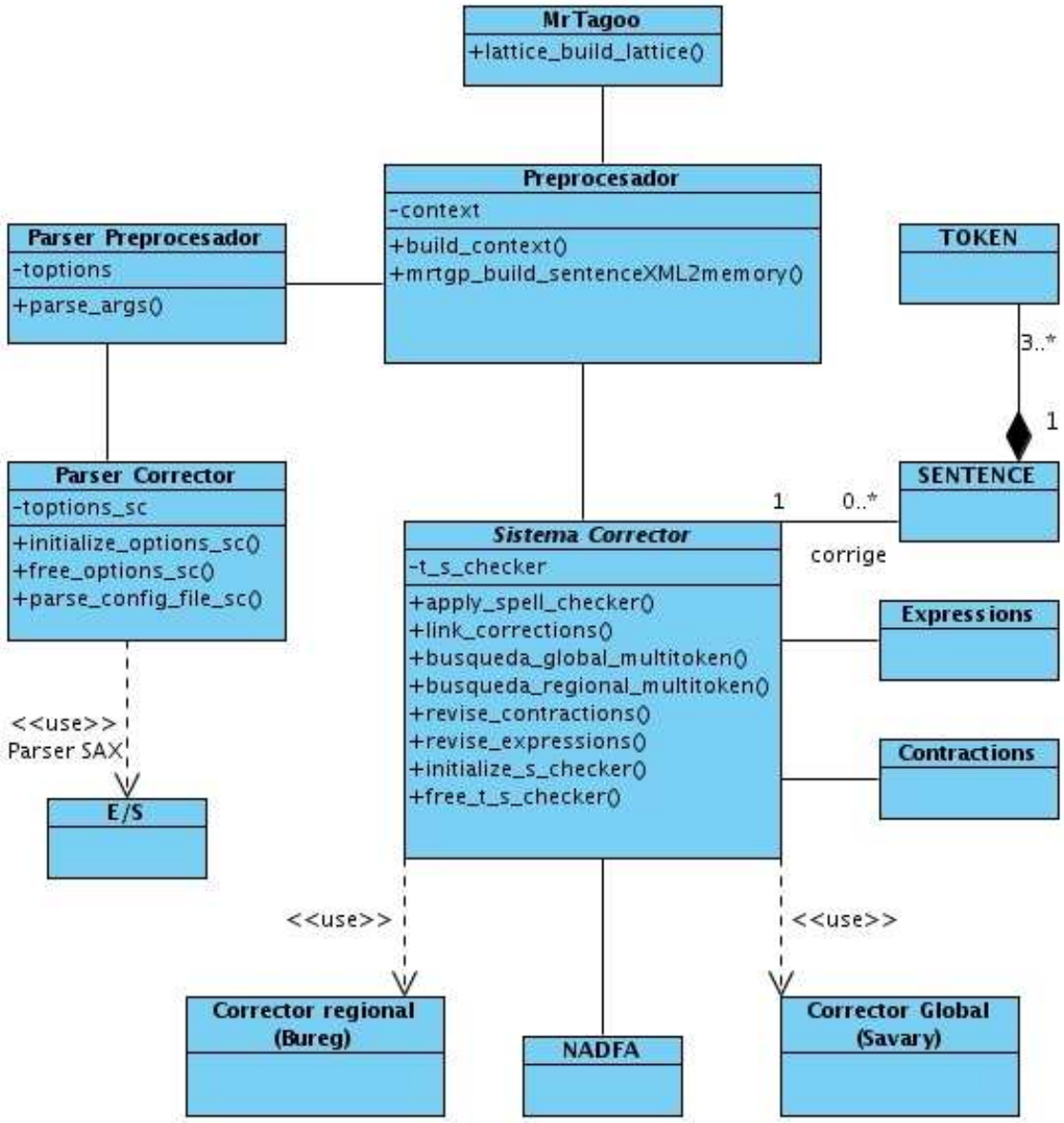

Figura 6.1: Diagrama de clases del diseño.

6.2. Diccionario de clases

En esta sección se detallará cada una de las clases que forman el diagrama de clases de la figura 6.1.

6.2.1. Sistema corrector (s_checker)

El Sistema corrector es el núcleo de nuestro proyecto. Su objetivo es aplicar la búsqueda de correcciones sobre las palabras que después de haber sido preprocesadas no se han podido etiquetar, esto es, son palabras desconocidas. Para ello recorre la frase a corregir. Si encuentra alguna palabra no etiquetada, aplicará sobre ella la búsqueda de correcciones, aplicando técnicas de corrección de tipo regional o global.

El hecho de que una palabra no aparezca en el lexicón es condición necesaria, pero no suficiente, para determinar que estamos ante un error ortográfico. Con el objetivo, de no descartar la posibilidad de que una palabra, aún no estando en el lexicón sea correcta, se puede incluír junto a las alternativas devueltas por el corrector la palabra original. Este opción es configurable por el usuario.

Figura 6.2: Diagrama de clases parcial.

Uno de los objetivos principales del proyecto, tal y como se describió en la sección 1.5, es la ampliación de tipos de errores a considerar por el corrector. Para corregir los errores ortográficos que resultan de unir dos palabra, por ejemplo, si escribimos *sinembargo* en lugar de *sin embargo*; es necesario realizar modificaciones dentro de las propias clases *Corrector global* y *Corrector regional*, que explicaremos más adelante. No obstante, la corrección de errores ortográficos que resultan de dividir una palabra en dos, por ejemplo, si se escribe *desapa rición* en lugar de *desaparición*, se llevará a cabo desde esta clase.

Para afrontar el manejo de los errores que resultan de dividir una palabra en dos, sabiendo que partimos de una estructura de tipo *token* no etiquetada (la palabra a

la que representa no se encontraba en el lexicón), se tendrán en cuenta tres casos posibles:

- TIPO 0: Se buscara correcciones para esa palabra de forma individual, a estas correcciones se le asignará de tipo 0.
- TIPO 1: Siempre que el *token* anterior al de la palabra desconocida represente a una palabra, su tipo tiene que ser SENTENCE_NORMAL_TOKEN, se buscarán correcciones para la unión o concatenación de la palabra anterior y la palabra desconocida a tratar. A las correcciones aquí obtenidas se le asignará el tipo 1.
- TIPO 2: Siempre que el *token* siguiente a la palabra desconocida represente a una palabra, su tipo tiene que ser SENTENCE_NORMAL_TOKEN, se buscarán correcciones para la unión o concatenación de la palabra desconocida y la palabra siguiente. Éstas correcciones serán de tipo 2.

A partir de ahora, las búsquedas en las que se apliquen estos tres casos se denominarán *búsqueda multitoken*.

En la sección 5.2 se describió la estructura *sentence* que representa una frase en el preprocesador. Se explicó también que esta estructura no admite anidamientos de alternativas. En consecuencia, no es viable aplicar de forma secuencial, sobre todas las palabras de una frase, los tres tipos de búsqueda que se acaban de describir; la existencia de varios errores consecutivos provocarían solapamientos que se deben evitar. La solución propuesta en este proyecto se describe a continuación.

El Sistema Corrector recorre la frase recibida desde el preprocesador, en el momento que se encuentre un *token* no etiquetado (se establecerá un contador a cero), se aplicará la búsqueda multitoken sobre esa palabra. Se seguirá recorriendo la frase y si el siguiente *token* no está etiquetado no se aplica la búsqueda sobre él, pues ya ha sido tratado junto al *token* anterior. Respecto al segundo *token* encontrado después de haberse aplicado la *búsqueda multitoken* sólo se trata si la palabra del *token* siguiente a este es conocida y sólo se aplica la búsqueda de correcciones para ese *token* de forma individual.

Para el tercer *token* a partir del último sobre el que se haya aplicado la *búsqueda multitoken* se vuelve a empezar (el contador se reinicia). En este punto, si el *token* no está etiquetado, se aplica sobre él la búsqueda de correcciones de los tres tipos. De esta forma, si el *token* anterior fuese incorrecto sería tratado aquí, tal como se ilustra en la figura 6.5.

Se ilustrará el funcionamiento que acabamos de explicar mediante tres situaciones posibles. En las figuras utilizadas un cuadrado representa un *token* y el asterisco representa que ese *token* no tiene etiquetas. Para la figura 6.3, se aplicará la *búsqueda multitoken*, representada por las tres flechas punteadas, sobre el TOKEN 2 y sobre

TOKEN 5. En la figura 6.4, se aplica la *búsqueda multitoken* sobre el TOKEN 2 y se realiza la búsqueda de correcciones individual sobre el TOKEN 4.

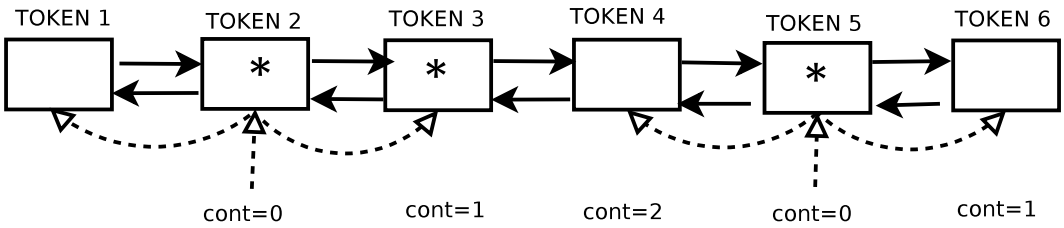

Figura 6.3: Representación de una frase con dos errores consecutivos.

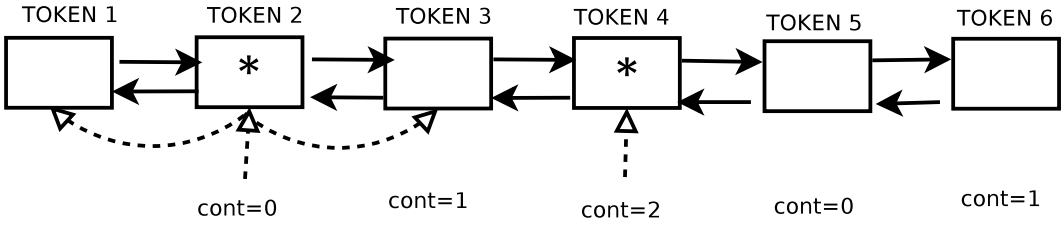

Figura 6.4: Representación de una frase con dos errores con una palabra correcta entre ellos.

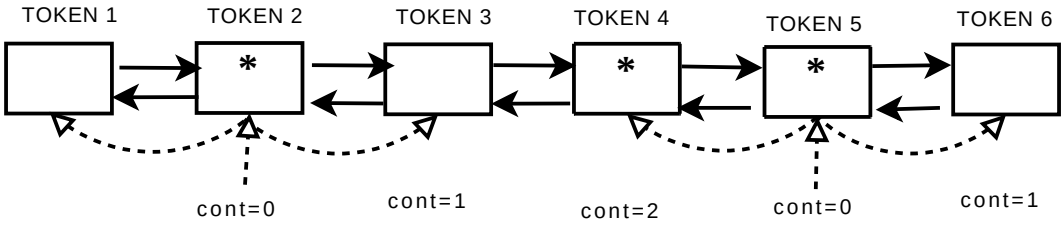

Figura 6.5: Representación de una frase con tres errores separados por una palabra correcta.

En la figura 6.6 se representa una situación más compleja. Para el TOKEN 2 se aplica la *búsqueda multitoken*, sin embargo como el siguiente *token* no es de tipo SENTENCE_NORMAL_TOKEN, no se aplica la búsqueda sobre la concatenación de la palabra del TOKEN 2 y la palabra del TOKEN 3. Además, se puede comprobar como varía el funcionamiento del contador (cont en las figuras), se reinicia cuando encontremos una palabra sin etiqueta, siempre que el contador sea mayor de 2.

Las correcciones que se obtienen para cada *token* se guardan en una lista y se deben enlazar en la estructura que representa la frase. Las correcciones tienen que

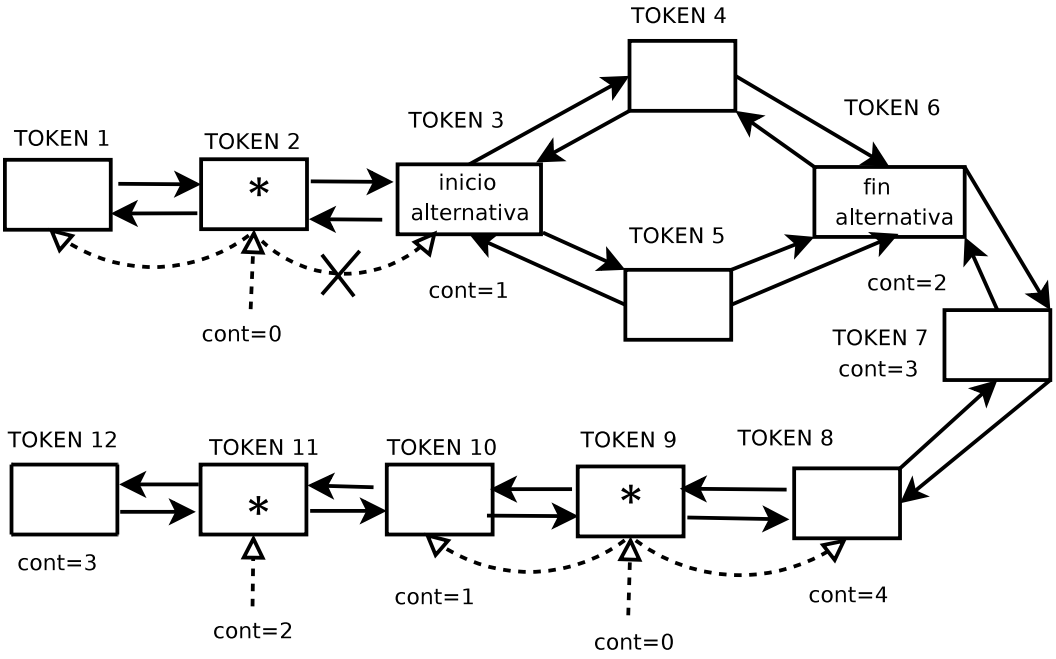

Figura 6.6: Representación de una frase con varios errores.

indicar el tipo (0, 1 ó 2) de búsqueda del que proceden, para poder realizar su integración de forma correcta en la frase.

Cada corrección sigue el formato:

```
tipo palabra1 [palabra2] [/] [palabra3] [palabra4]
```

Donde `tipo` indica el número que representa el tipo de la búsqueda del que proviene esa corrección. La "/" separa las correcciones de un *token* original de la frase de las correcciones de otro *token*, por tanto sólo puede aparecer la "/" en las correcciones de tipo 1 y 2.

- Si la corrección es de tipo 1 la palabra o palabras que preceden a la barra se corresponden con correcciones para el *token* anterior al que se trata y la palabra o palabras que aparecen después de la barra se corresponden con el *token* actual.

- Si la corrección es de tipo 2 la palabra o palabras que preceden a la barra se corresponden con correcciones para el *token* actual y la palabra o palabras que aparecen después de la barra se corresponden con el *token* siguiente al que se trata.

- Una corrección de tipo 1 en la que no aparezca la /"indica que la palabra es una corrección para la unión del *token* actual y el anterior a éste.
- Una corrección de tipo 2 en la que no aparezca la /"indica que la palabra es una corrección para la unión del *token* actual y el siguiente a éste.

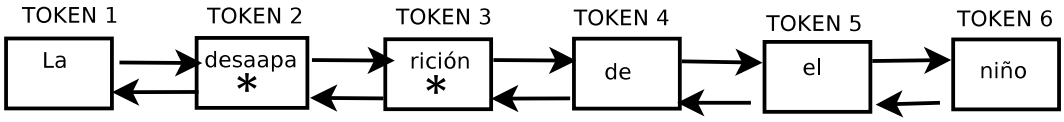

Figura 6.7: Estructura recibida por el Sistema Corrector.

Por ejemplo, para la frase: "*la desaapa rición del niño*", en la figura 6.7 se representa la estructura recibida por el Sistema corrector, se obtendrían dos correcciones de distancia de edición mínima a partir de la palabra "desaapa":

2	destapa	/	ración
2	desaparición		

La primera indica que "destapa" es una corrección para el *token* que representa a la palabra "desaapa" y "ración" es una corrección para el *token* siguiente, que representa a "rición". La segunda corrección "desaparición", es una corrección formada por una sola palabra pero representa a dos *tokens*, el actual y el siguiente, esto lo sabemos por el código 2 que indica que esta corrección se ha producido como resultado de unir la palabra del *token* actual y la del siguiente. Después de enlazar las correcciones en la estructura de la frase se debe obtener el resultado que se muestra en la figura 6.8.

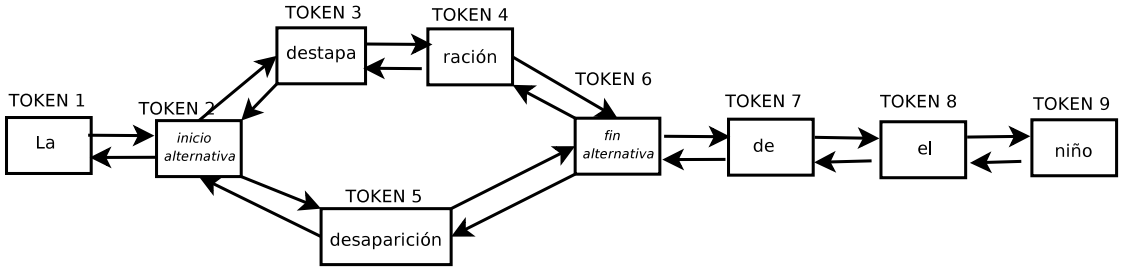

Figura 6.8: Estructura después de haber aplicado el Sistema Corrector.

Una vez que hemos integrado las correcciones obtenidas en la estructura de la frase, se vuelve a aplicar la detección de contracciones y de expresiones del lenguaje. Las contracciones se detectarán justo después de enlazar las correcciones

para cada *token* en la estructura que representa a la frase. De este modo, sólo se aplicará el módulo de detección de contracciones sobre los *tokens* afectados por las correcciones. El módulo de detección de contracciones existente en el preprocesador no estaba pensado para tratar listas de alternativas para una o varias palabras, pues se ejecutaba antes que estas pudiesen existir en la frase; por este motivo se ha modicado su funcionamiento para adaptarlo a nuestras necesidades.

Para controlar la aparición de nuevas locuciones, surgidas a raíz de los cambios realizados con las correcciones, se aplican los módulos de detección de locuciones (seguras o inseguras) del preprocesador sobre toda la frase. Esto significa que se aplican sólo una vez sobre la frase corregida completa y únicamente si ha tenido lugar alguna corrección. El motivo que nos ha llevado a realizar la detección sobre toda la frase, y no sólo sobre la sección modificada, se debe a que una corrección de una palabra puede coincidir con la última o la primera de una locución en las que el resto de las palabras eran correctas. Por tanto, se ha considerado más eficiente aplicar la detección para intentar encontrar nuevas locuciones solamente una vez que se haya finalizado la corrección sobre toda la frase. La figura 6.9 muestra la estructura de la frase para la secuencia original "ó cabbo" antes y después de ser revisada por el módulo de contracciones.

Un ejemplo representativo de la detección de nuevas contracciones y locuciones sería " ásveces", como se muestra en la figura 6.10. En ella se muestra la evolución del *token* desde que se recibe en el Sistema Corrector, suponiendo que la lista de correcciones está formada por una única corrección ("`0 ás veces`"), hasta que se devuelve al Sistema Preprocesador.

Después el preprocesador enviará esta información al etiquetador *MrTagoo* que se encargará de *desambiguar* la frase en función del contexto, es decir, eligirá la ruta más probable dentro de cada lista de alternativas.

A continuación, se describirán los atributos y métodos más importantes dentro de la clase Sistema Corrector.

Atributos:

- `t_s_checker`: Estructura que almacena los datos necesarios para configurar la corrección a aplicar. Está compuesta por los siguientes elementos:

`global_search`	Indica si la búsqueda es global o no(regional).
`word_incorrect`	Indica si la palabra desconocida se incluye como alternativa.
`regions`	Indica la ruta del fichero donde se encuentran las regiones.
`autom_ext`	Autómata extendido necesario para la búsqueda regional.
`umbrales`	Distancia máxima permitida con la palabra desconocida.

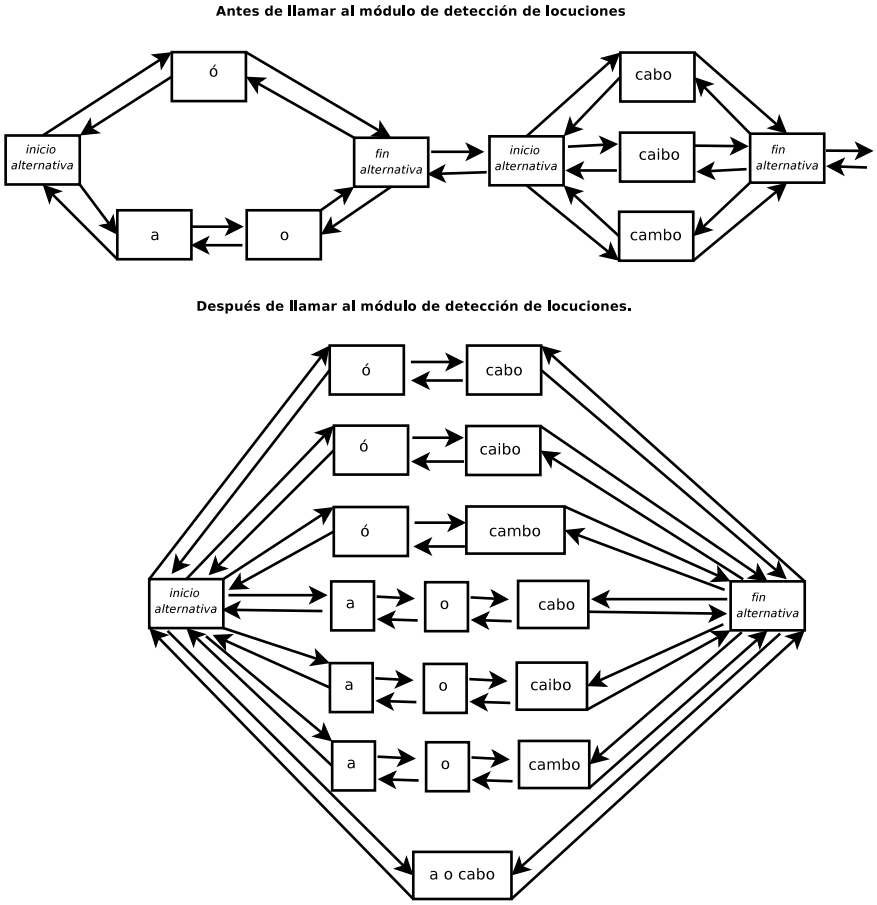

Figura 6.9: Representación de "ó cabbo" antes y después de llamar al módulo de detección de locuciones.

- `corrections`: Lista que almacena para las correcciones obtenidas para cada *token* con el formato explicado anteriormente.

Métodos:

- `apply_spell_checker()`: Recibe una frase desde el preprocesador y para cada palabra desconocida busca las posibles correcciones.

- `busqueda_global_multitoken()`: Aplica la búsqueda de correcciones individuales y multipalabra sobre la palabra desconocida, utilizando el método de corrección global.

- `busqueda_regional_multitoken()`: Aplica la búsqueda de correcciones sobre la palabra desconocida utilizando el método de corrección regional.

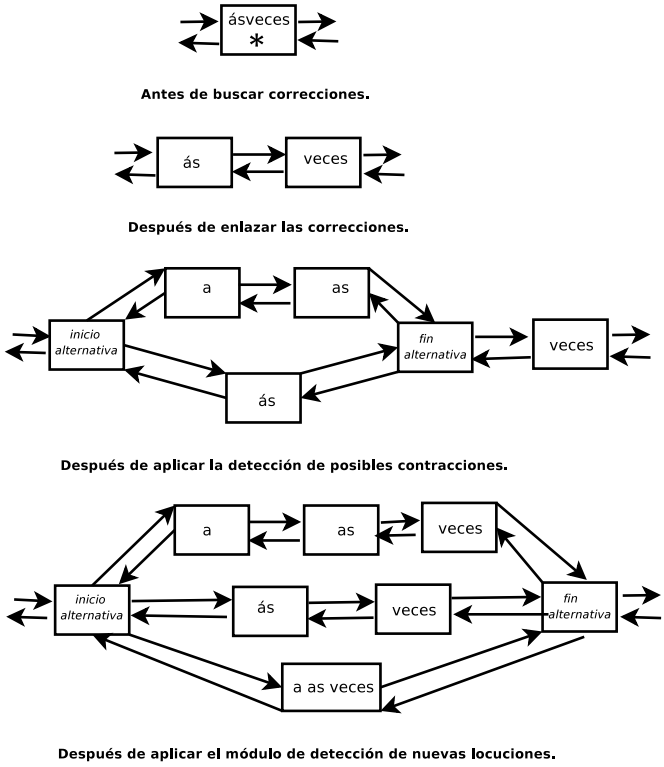

Figura 6.10: Representación de la evolución de la estructura de la frase en el Sistema corrector.

- `link_corrections()`: Se encarga de enlazar las correcciones obtenidas para la palabra desconocida en la estructura de datos original.
- `revise_contractions()`: Comprueba si alguna de las correcciones coincide con alguna contracción y, si es así, la incluye como posible corrección.
- `revise_expressions()`: Comprueba si la frase sobre la que se han hecho las correcciones contiene alguna locución y, si es así, la incluye como posible corrección.
- `update_prev_corrections()`: Actualiza la lista de correcciones con las obtenidas uniendo la palabra desconocida con la anterior. Para explicar el funcionamiento del método se distinguen tres posibles casos:

  1. Si la distancia de edición de las nuevas correcciones es menor a la distancia de las correcciones ya existentes en la lista (realizadas para la palabra desconocida individualmente), se reemplazan la lista de correcciones por las nuevas.

2. Si la distancia de edición de las nuevas correcciones es igual a la distancia de las correcciones ya existentes en la lista (realizadas para la palabra desconocida individualmente), entonces se debe comprobar si la palabra anterior es desconocida o no. Si no lo es entonces se añaden las nuevas correcciones a la lista de correcciones y se hacen las combinaciones de resultado necesarias. En caso contrario se reemplazan.
3. Si la distancia de edición de las nuevas correcciones es mayor a la distancia de las correcciones ya existentes en la lista (realizadas para la palabra desconocida individualmente) entonces, al igual que en el caso anterior, se debe comprobar si la palabra anterior es desconocida o no. Si es desconocida se debe buscar las correcciones para esa palabra y comparar la suma de la distancia de edición de las correcciones de las dos palabras individualmente frente a la distancia de edición de las correcciones para la unión de ambas. En función de los resultados obtenidos, se realizaran las combinaciones pertinentes en la lista de correcciones.

- `update_next_corrections()`: Actualiza la lista de correcciones con las obtenidas para la palabra desconocida unida a la siguiente. Es similar al método anterior, pero varía la forma en la que se hacen las modificaciones sobre la lista de correcciones.
- `create_token()`: Crea un nuevo *token* que represente a una palabra, debe buscarlo en los diccionarios
- `create_complete_token()`: Crea un *token* a partir de una forma, etiqueta y lema.
- `create_alternative()`: Crea una nueva estructura para reflejar una lista de alternativas dentro de la frase. Aunque existía un método similar en el preprocesador, no contemplaba la posibilidad de crear la alternativa con varios *tokens* dentro para una misma palabra.
- `clone_complete_token()`: Crea un *token* a partir de otro copiando su forma, etiqueta, lema y coordenadas que ocupa en la frase original.
- `delete_token()`: Elimina un *token* dentro de una frase, es decir, une el *token* anterior y el posterior y luego elimina el *token* a borrar.
- `merge_alternative()`: Crea una nueva alternativa y la une a una lista de alternativas ya existente.
- `alternative_exists()`: Comprueba que una secuencia de *tokens* no forme parte ya de una lista de alternativas.

- `contraction_check_alternatives()`: Revisa la existencia de palabras que coincidan con una contracción en el interior de una lista de alternativas. En caso de coincidencia, une la nueva o nuevas alternativas a la lista. La complejidad de este método radica en la prohibición de anidamiento de alternativas dentro de la estructura SENTENCE, que significa que no pueden existir una lista de alternativas dentro de otra.

6.2.2. Parser corrector

Esta clase representa el módulo encargado de cargar en memoria el contenido del fichero XML de configuración del *Sistema corrector*.

Atributos

- `toptions_sc`: Esta es la estructura utilizada para cargar en memoria los datos procedentes del fichero de configuración. Está formada por los elementos de la siguiente tabla:

`field_content`	Se utiliza para leer el contenido del fichero
`is_path`	Indica si se está tratando una ruta o no.
`config_dir`	Directorio en donde se encuentre el fichero.
`sc`	Estructura de tipo `t_s_checker` donde se guarda la configuración del corrector.

Métodos

- `parse_config_file_sc()`: Este método carga en memoria la información procedente del fichero de configuración del corrector mediante la utilización de un parser SAX.

6.2.3. Preprocesador

Esta clase representa al preprocesador morfológico. Se explicarán aquí aquellos atributos y métodos modificados para poder integrar el sistema corrector en el propio preprocesador.

Atributos

- CONTEXT: Esta es la estructura que contiene toda la información necesaria para el funcionamiento del preprocesador. A los datos que ya contiene se le añadirán los elementos necesarios para integrar nuestro sistema corrector.

`spell_checker`	Indica si el sistema corrector se debe aplicar o no.
`sc`	Son los parámetros de configuración del corrector, se corresponde con una estructura de tipo t_s_checker.

Métodos

- `build_context()`: Inicializa todos los parámetros necesarios para el preprocesador. Entre ellos, la estructura encargada de configurar el corrector.

- `mrtgp_build_sentenceXML2memory()`: Aplica los módulos de preprocesamiento de los datos. Desde aquí, y en último lugar, se invoca el sistema corrector que se va a desarrollar.

6.2.4. Parser preprocesador

Esta clase recibe los parámetros de entrada, es decir, aquellos que el usuario establece para llamar al programa, y los carga en una estructura en memoria. Se emplea un parser SAX para leer la información que se encuentra en el fichero XML de configuración. El usuario debe indicarle al preprocesador la ruta del directorio en la que se encuentra este fichero. Además, desde esta clase se llamará a la clase *Parser corrector*, que se encarga de leer el fichero de configuración del *Sistema corrector*.

Atributos

- `toptions`: Estructura en la que se almacena la información procedente del fichero de configuración. Se añade el elemento `spell_checker_file`, que guarda la ruta del fichero de configuración del *Sistema corrector*.

Métodos

- `parse_args()`: Toma los datos de entrada del usuario, así como la información procedente del fichero de configuración y los carga en la estructura toptions.

6.2.5. Corrector regional (Bureg)

Esta clase representa el módulo encargado de realizar la búsqueda de correcciones ortográficas de tipo regional. Se ampliará los tipos de error manejados (inserción, intercambio, transposición y borrado), de forma que se incluya también el error de unión de dos palabras en una sola.

Atributos

- `tbureg`: Contiene todos los datos necesarios para realizar la búsqueda. Además, también guardará los resultados obtenidos.

Métodos

- `busqueda_regional()`: Busca todas las posibles correcciones de coste regional mínimo para una palabra recibida.
- `obt_items_union()`: Se añade esté método al Corrector regional para obtener correcciones para una palabra en las que se haya saltado un espacio en blanco entre dos palabras correctas. Por ejemplo, para la palabra "casaverde" se obtiene la corrección "casa verde", en donde las palabras "casa" y "verde" forman parte del diccionario.
- `obt_items_error()`: Se modifica este método para que a las correcciones obtenidas para los errores de inserción, intercambio, transposición y borrado se añadan los obtenidos por el método `obt_items_union()`.

6.2.6. Corrector global (Savary)

Esta clase representa el módulo encargado de realizar la búsqueda de correcciones ortográficas de tipo global .

Atributos

- `palabras`: Estructura en la que se almacenan el número de correcciones a una palabra, las correcciones en sí y la distancia de edición con respecto a la palabra desconocida.

Métodos

- `busqueda_tolerante()`: Obtiene todas las posibles correcciones para una palabra desconocida con distancia de edición mínima. Esta distancia debe ser menor o igual a un umbral pasado como parámetro. Se modifica este método para que también detecte errores de unión de dos palabras, en los que se escriba dos palabras seguidas sin un espacio en blanco entre ellas.

6.2.7. MrTagoo

Esta clase representa al etiquetador que se encargará de eliminar las ambigüedades a nivel morfosintáctico y ,además, puede escoger entre flujos de *tokens*

con distinta longitud. Como salida se obtiene el texto segmentado y desambiguado. Para su implementación se utilizan los Modelos de Markov junto a una extensión del algoritmo de Viterbi explicados en la subsección 1.4.1.

Métodos

- `lattice_build_lattice()`: Como resultado de su aplicación se obtiene una única etiqueta para cada *token*.

6.2.8. Contractions

Esta clase representa el módulo de contracciones que se encarga de dividir una contracción en sus diferentes partes, etiquetando cada una de ellas. Para ello, utiliza un fichero de texto en el que se guarda una lista donde se especifica como se descomponen todas las contracciones del lenguaje. Por ejemplo, para el *token do* (del) su salida serían dos *tokens* distintos, uno de ellos con la preposición *de* y en otro el artículo *o*.

Atributos

- `contraction_token`: guarda una contracción y las formas en la que ésta se puede descomponer.

Métodos

- `mrtgp_contraction_search_contraction()`: Comprueba si una palabra dada coincide con alguna contracción y devuelve el atributo `contraction_token` correspondiente.

- `mrtgp_contraction_check_word()`: Engloba a la función anterior y, además, construye la estructura de *tokens* para representar una lista de alternativas dentro de la estructura SENTENCE. Entre estas alternativas se encuentra las posibles descomposiciones de la contracción y, si la palabra aparece en el lexicón, también se añade como alternativa.

6.2.9. Expressions

Esta clase se encarga de unir los *tokens* que componen una locución y de etiquetarlos como una unidad conjunta. Existen dos diccionarios de locuciones:

1. *Locuciones seguras*: Incluye aquellas locuciones que se sabe con seguridad (morfológicamente) que lo van a ser. *A priori* es un ejemplo de locución segura.

2. *Locuciones inseguras*: Incluye aquellas locuciones que pueden ser locuciones o no. *Sin embargo* es una expresión de este tipo porque no se sabe con seguridad si es una locución o si es por un lado *sin* (preposición) y por otro *embargo* (sustantivo o verbo).

Métodos

- `mrtgp_expression_build_all_certain_expressions`: Detecta si existe alguna locución segura en una frase y construye la estructura correcta para ésta.
- `mrtgp_expression_build_all_uncertain_expressions`: Detecta si existe alguna locución insegura en una frase y construye la estructura correcta para ésta.

6.2.10. NADFA

Esta clase representa el diccionario, es decir, a la estructura AFDAN (en inglés) explicada en la sección 1.4.3. Contiene los métodos necesarios para cargar un diccionario en memoria, acceder al índice asociado a una palabra dada y viveversa.

6.2.11. TOKEN

Esta clase representa una unidad léxica, originalmente se crea un *token* por cada palabra. Sin embargo, como se ha explicado anteriormente, un *token* puede contener más de una palabra [1] o una palabra puede generar más de un *token* [2].

Como se ha explicado en la sección 5.2 se diferencian cinco tipos de *tokens*.

6.2.12. SENTENCE

Esta clase representa una estructura en memoria de una frase a analizar. Está compuesta por la unión de los *tokens* que representan cada palabra de la frase para marcar el inicio y fin de cada frase se utiliza un tipo de *token* especial.

6.2.13. E/S

Esta clase incluye los ficheros de configuración que intervienen en la ejecución del sistema. Además del fichero de configuración del preprocesador, necesitaremos un fichero para configurar el sistema corrector.

[1] Como es el caso de las locuciones.

[2] Éste el caso de las contracciones o de los verbos con pronombres enclíticos

Este último permite indicar si se desea que una palabra incorrecta forme parte también de las alternativas de corrección. Esto significa que será el etiquetador el que decida cúal es la opción más probable para una palabra: que se trate de un error ortográfico, y escoja entre una de las posibles correcciones, o que la palabra realmente sea correcta pero no forme parte del lexicón.

Por otra parte, es en este fichero el lugar donde el usuario debe indicar el tipo de corrección ortográfica a aplicar. Si se desea que la corrección se realice mediante una técnica de tipo global, se indica mediante una 'g', o si lo que se quiere es que se aplique una técnica de tipo regional se indica mediante una 'r'.

Finalmente, se indican los umbrales máximos de error permitidos para encontrar correcciones para una palabra desconocida. Para la corrección global sólo se tiene en cuenta un umbral, pero para la corrección regional se pueden indicar umbrales de error diferentes, en función del tipo de error (borrado, inserción, reemplazo o transposición) que se esté tratando. Así, si establecemos como umbral 2, no se considerarán correcciones cuya distancia de edición con la palabra desconocida sea mayor que 2.

Se verá un ejemplo de la estructura del fichero de configuración en la sección de implementación (7.1).

6.3. Diagramas de secuencia para los casos de uso

En esta sección se describirán con mayor detalle los diagramas de secuencia de los casos de uso del análisis. Para la realización de los diagramas de secuencia del diseño se utilizarán algunas llamadas a funciones del preprocesador (empezarán con la siguiente secuencia *mrtgp_...*) para representar la interacción con el resto del sistema.

6.3.1. Caso de uso: Aplicar Corrección

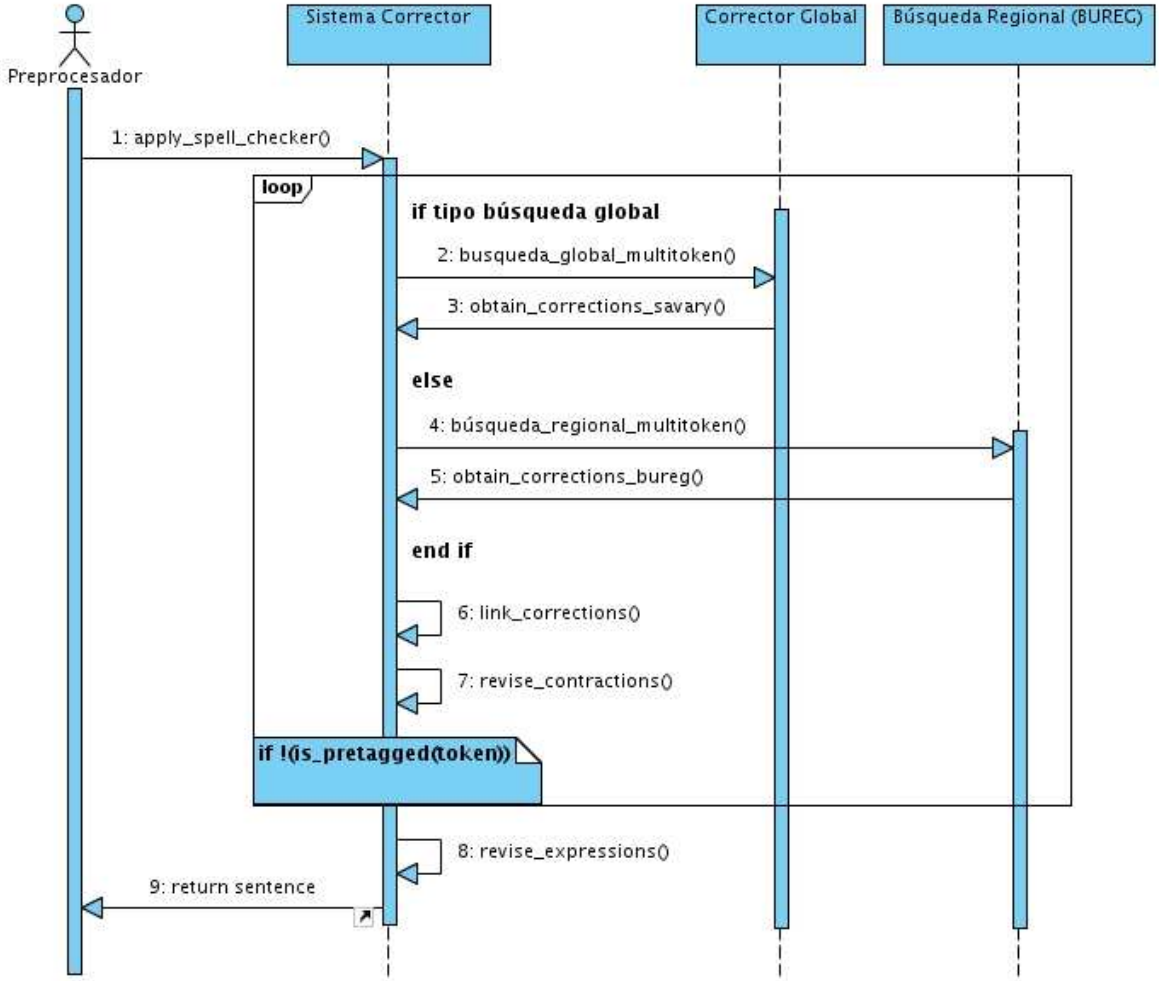

Figura 6.11: Diagrama de secuencia del diseño: Aplicar corrección

Aplicar Correccion: Buscar correccion global

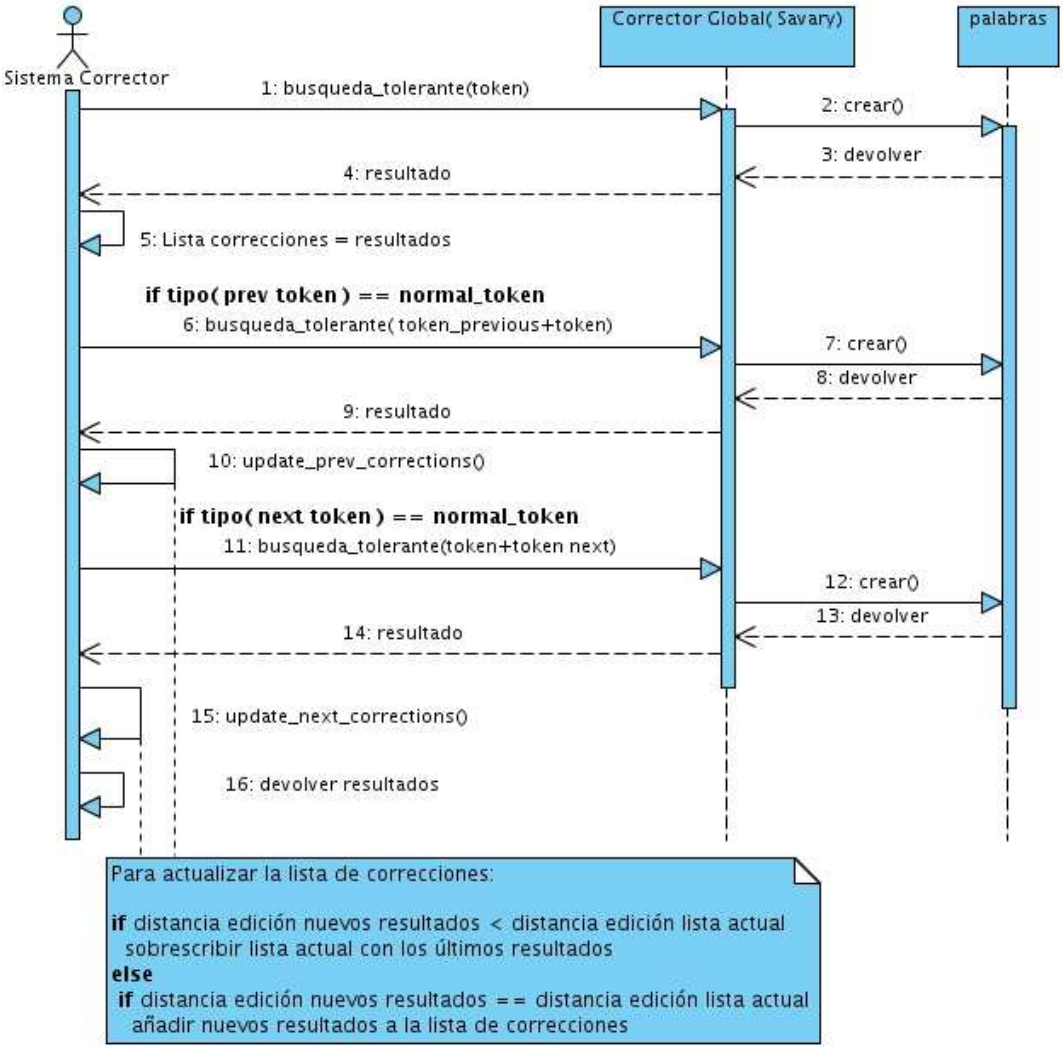

Figura 6.12: Diagrama de secuencia del diseño: Buscar corrección global

Aplicar Correccion: Buscar correccion regional

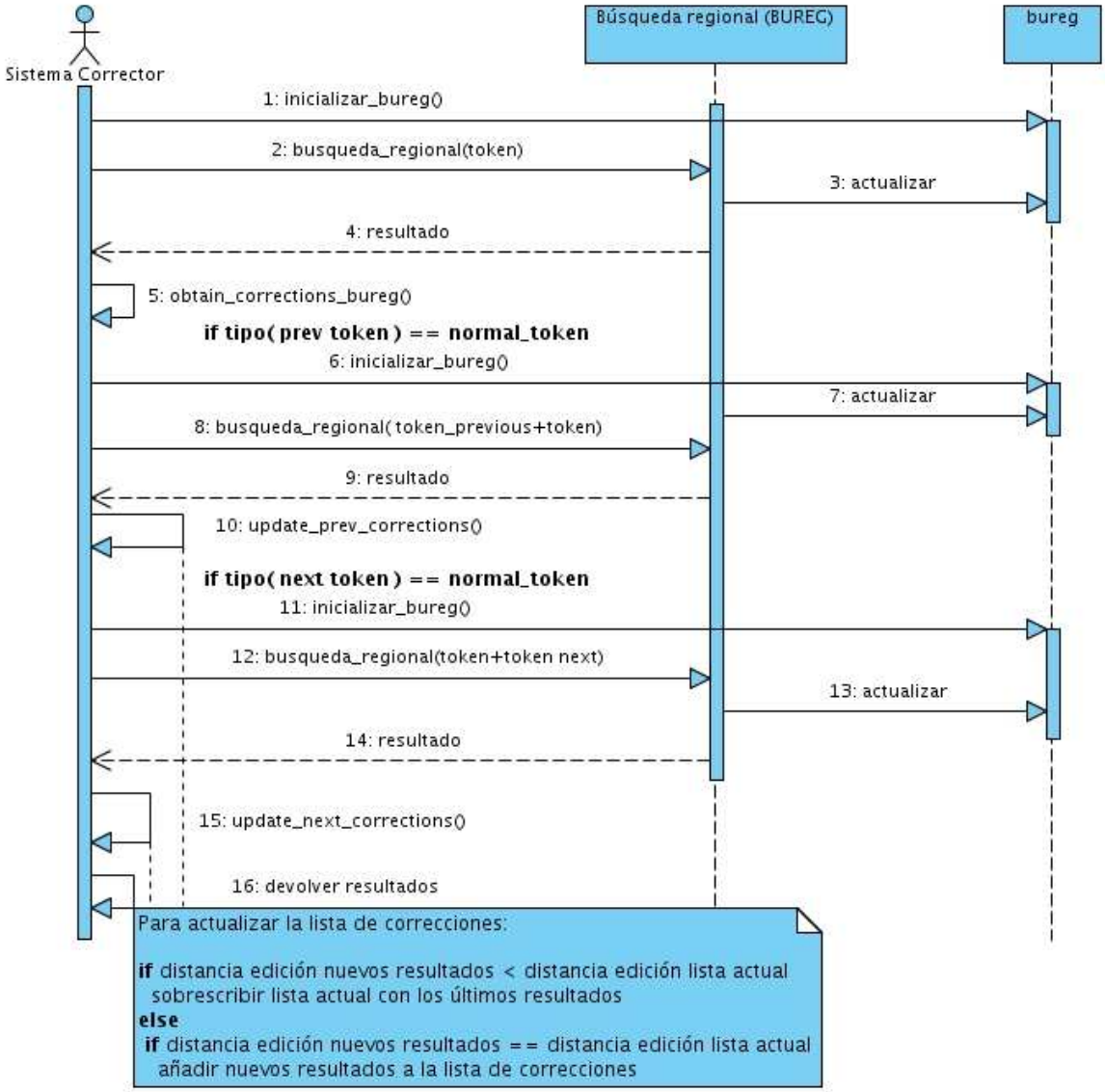

Figura 6.13: Diagrama de secuencia del diseño: Buscar corrección regional

Aplicar Correccion: Enlazar correcciones

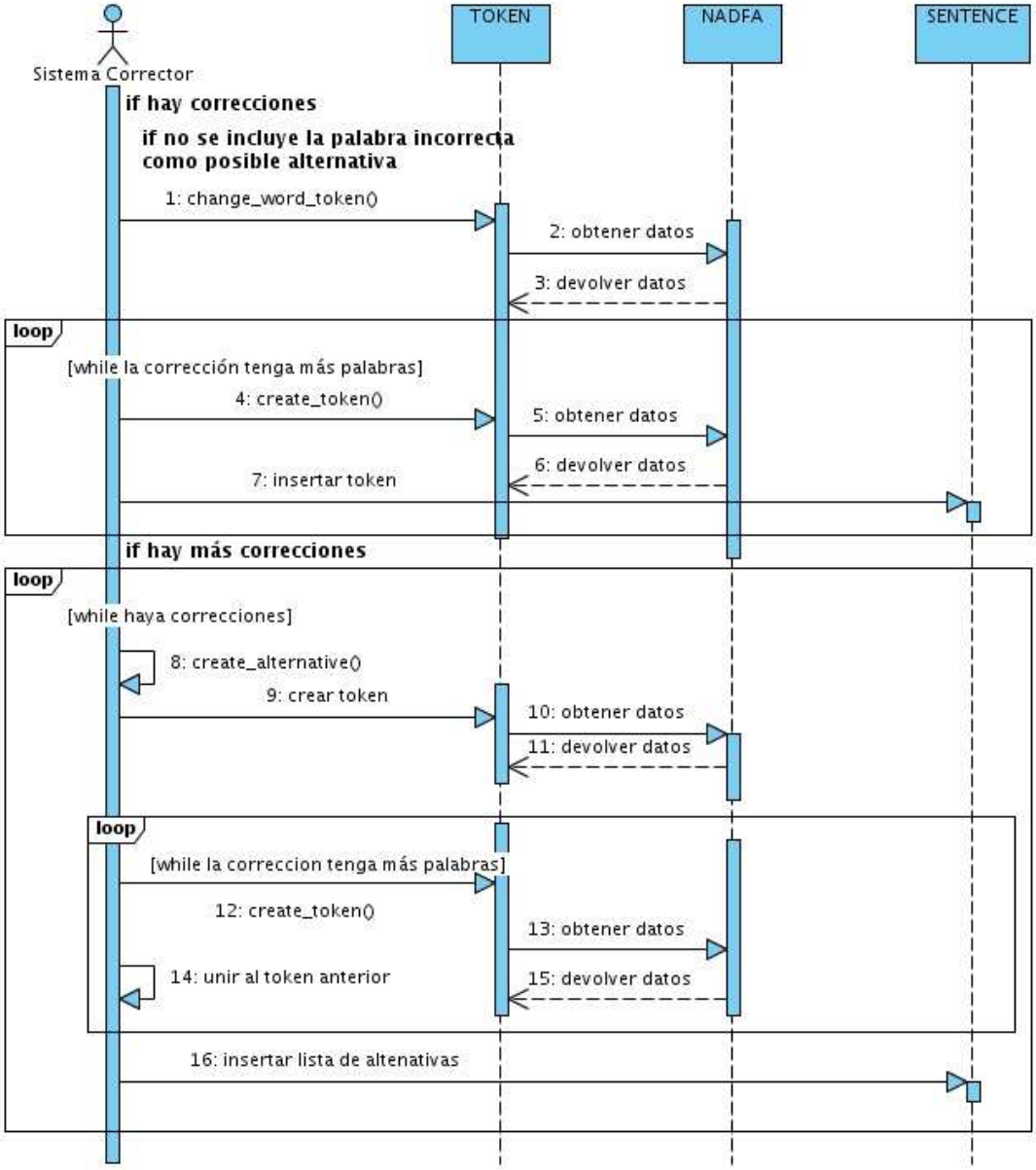

Figura 6.14: Diagrama de secuencia del diseño: Enlazar correcciones

Aplicar Correccion: Revisar nuevas contracciones

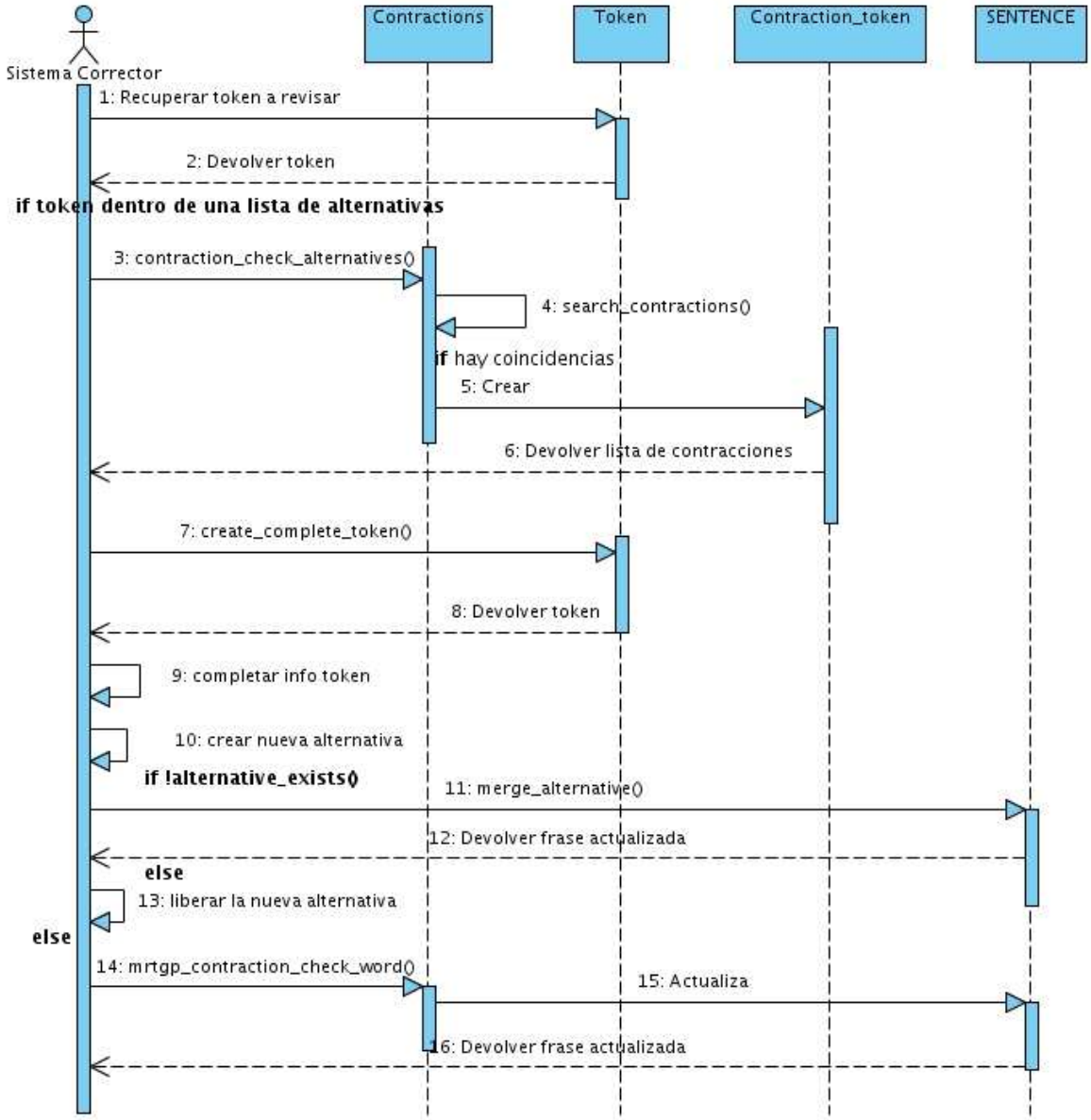

Figura 6.15: Diagrama de secuencia del diseño: Revisar nuevas contracciones

Aplicar Correccion: Revisar nuevas expresiones

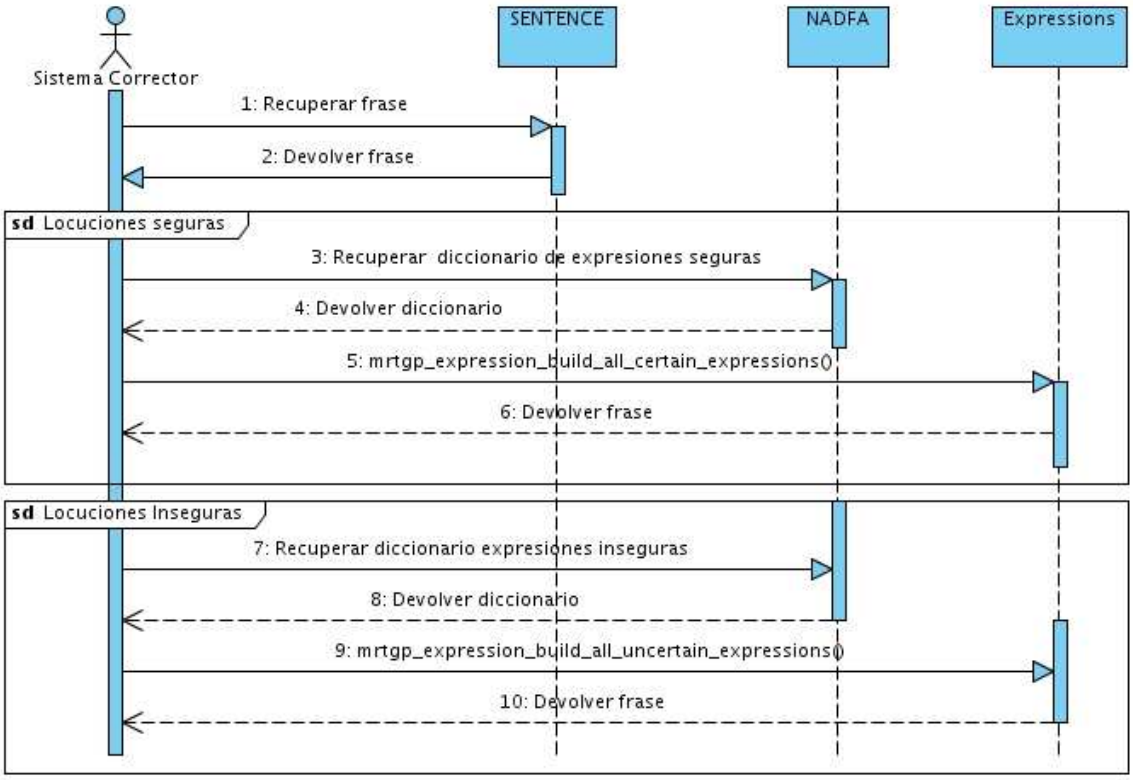

Figura 6.16: Diagrama de secuencia del diseño: Revisar nuevas expresiones

Capítulo 7

Implementación y pruebas

7.1. Implementación

En esta fase del desarrollo se traduce el modelo que se obtiene en el diseño en código fuente. En esta sección se reseñarán aspectos importantes en la implementación de nuestro sistema.

7.1.1. Diagrama de componentes

Mediante el diagrama de componentes se modela la estructura del software, además muestra la forma en la que se agrupan los componentes dentro de *paquetes* y la relación existente entre ellos. Este diagrama se muestra en la figura 7.1.

7.1.2. Fichero de configuración

El fichero de configuración está formado por el elemento raíz *configuration* que se compone por los siguientes elementos:

- word_incorrect: Indica si se incluye la palabra incorrecta como alternativa de corrección ("yes") o no se incluye ("no").
- regions: Fichero donde se almacenan las regiones. Se encuentra dentro del directorio de configuración que se pasa como parámetro para invocar al programa. Si la búsqueda es de tipo global no es necesario indicarlo.
- search_type: Indica si la técnica empleada para realizar la corrección ortográfica es de tipo global ("g") o de tipo regional ("r").
- tresholds: Establecen los umbrales de error máximos permitidos para encontrar correcciones, es decir, los caracteres que difieren de la palabra original.

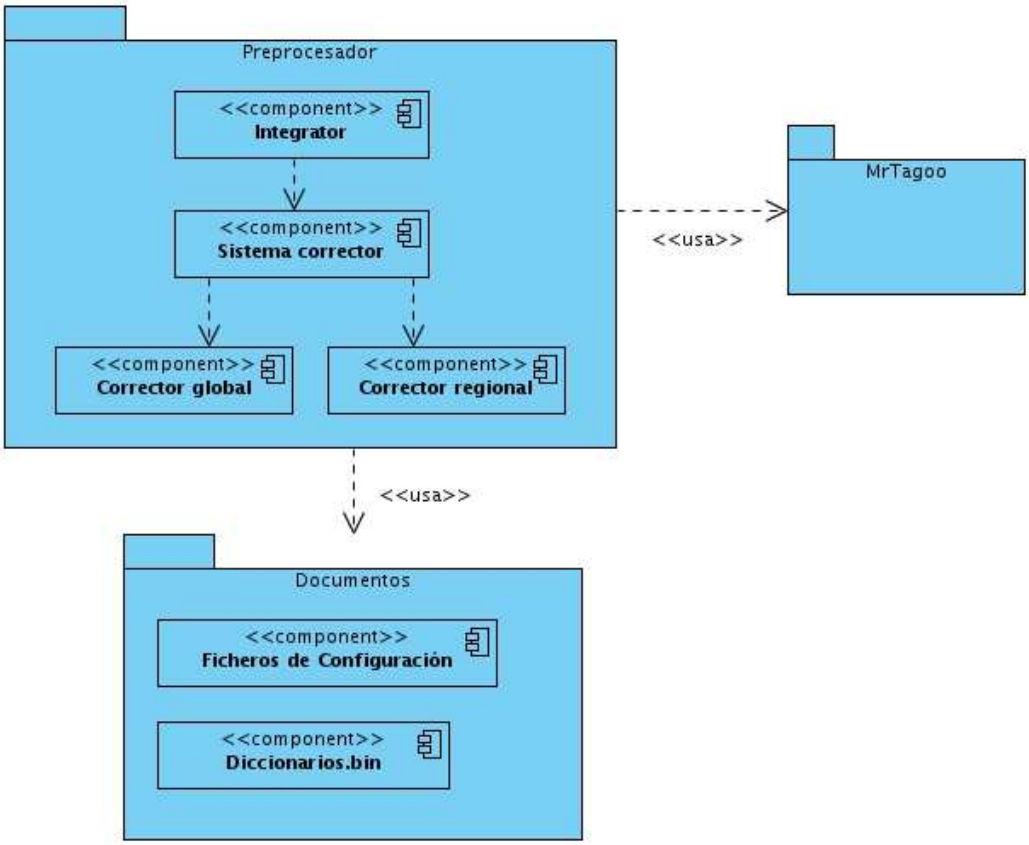

Figura 7.1: Diagrama de componentes

- consecutives: Número máximo de errores de edición consecutivos permitidos.
- global: Número de errores de edición máximo permitidos.
- local: Este elemento está formado por los siguientes componentes:
  - deletion: Número de errores de borrado máximo permitidos.
  - insertion: Número de errores de inserción permitidos
  - transposition: Número de errores de transposición permitidos.
  - replacement: Número de errores de reeemplazo permitidos.

La figura 7.2 muestra el contenido de un ejemplo para el archivo de configuración XML aquí descrito.

```
-<configuration>
   <word_incorrect>no</word_incorrect>
   <regions>lexiconPrincipal.reg</regions>
   <search_type>g</search_type>
  -<thresholds>
     <consecutives>2</consecutives>
     <global>2</global>
    -<local>
       <deletion>2</deletion>
       <insertion>2</insertion>
       <transposition>2</transposition>
       <replacement>2</replacement>
     </local>
   </thresholds>
  </configuration>
```

Figura 7.2: Ejemplo de utilización del fichero de configuración *s_checker.xml*.

7.1.3. Detalles de implementación

Un dato relevante para nuestro sistema es el juego de codificación de caracteres empleado, en este caso ha sido ISO-8859-1.

ISO 8859-1 es una norma de la ISO que define la codificación del alfabeto latino, incluyendo los diacríticos (como letras acentuadas, ñ, ç), y letras especiales (como ß, Ø), necesarios para la escritura de la mayor parte de las lenguas originarias de Europa occidental como por ejemplo: español, gallego, catalán, vasco, alemán, francés, inglés, italiano, portugués, etc. También se conoce como Alfabeto Latino n.º 1 o ISO Latín 1.

Esta norma pertenece al grupo de juegos de caracteres de la ISO conocidos como ISO/IEC 8859 que se caracterizan por poseer la codificación ASCII en su rango inicial (128 caracteres) y otros 128 caracteres para cada codificación, con lo que en total utilizan 8 bits.

Kubuntu usa UTF-8 como juego de caracteres predeterminado, así que debemos cambiar la configuración para establecer por defecto la codificación ISO-8859-1.

Para la codificación del sistema se ha utilizado de forma preferente los tipos de datos de la librería GLib [1] . Tiene especial importancia el tipo *GSList*, empleado para guardar la lista de correcciones en nuestro sistema.

[1] *http://library.gnome.org/devel/glib/2.20/*

7.2. Pruebas

En esta sección se detallan las pruebas llevadas a cabo para validar el correcto funcionamiento del software desarrollado. Estas pruebas han sido realizadas paralelamente con el proceso se implementación del software.

Una prueba se puede definir como una ejecución del programa con la intención de encontrar errores previos a la entrega al usuario. Se trata de buscar defectos en nuestro *software*.

7.2.1. Pruebas de unidad

Estas pruebas se centran en cada unidad de software o módulo individualmente, realizando pruebas de caja blanca y de caja negra. Se tiene que probar que la información que fluye entre los módulos del sistema se hace de forma adecuada hacia y desde la unidad de programa que está siendo probada. Es necesario, por lo tanto, probar cada componente de forma individual.

Pruebas de caja blanca

Las pruebas de caja blanca se encargan de validar la lógica interna de un módulo. Su realización debe garantizar que:

- Se recorren por lo menos todos los caminos de ejecución independientes de cada módulo.
- Se ejecutan todas las decisiones lógicas en su parte verdadera y en su parte falsa.
- Se recorren todos sus bucles.

Todas estas pruebas fueron realizadas durante la implementación del sistema, comprobando que los módulos implementados cumplen los requisitos que acabamos de describir

Pruebas de caja negra

Las pruebas de caja negra se centran en probar que se verifican los requisitos funcionales del software, es decir, que las funciones para las cuales se ha diseñado el sistema se cumplen sin errores. Los errores que se pretenden detectar mediante las técnicas de caja negra son:

- Funciones incorrectas o ausentes.
- Errores de rendimiento.

- Errores de inicialización o terminación.

Las pruebas de caja negra realizadas se han centrado en comprobar la integridad de los datos de entrada para el sistema, se han verificado los siguientes aspectos:

Título:	**Introducir un parámetro de ejecución incorrecto**
Resultado:	Se muestra un mensaje de error indicando el problema ocurrido.
Evaluación	Correcto

Título:	**Introducir un archivo de configuración que no está bien formado**
Resultado:	Se muestra un mensaje de error indicando que ha habido problemas en el análisis del fichero.
Evaluación	Correcto

Título:	**Introducir una ruta de un archivo incorrecta en el fichero de configuración**
Resultado:	Se muestra un mensaje de error indicando que no ha sido posible acceder al archivo.
Evaluación	Correcto

Título:	**Introducir un dato incorrecto en el fichero de configuración**
Resultado:	Se asigna el contenido establecido por defecto para esa opción.
Evaluación	Correcto

7.2.2. Pruebas de integración

Las pruebas de integración se llevan a cabo durante la construcción del sistema e involucran los diferentes módulos empleados y su cometido final es la verificación del sistema como conjunto.

Dado que uno de los objetivos de este proyecto es la integración de varias herramientas, se ha comprobado, de forma individual y paralela a la incorporación de cada módulo, que el funcionamiento era satisfactorio. Finalmente, se han hecho una serie de pruebas globales obteniendo los resultados esperados

Parte III

Manual de Usuario

Capítulo 8

Instalación y ejecución

Es este capítulo se detalla la información referente a los requisitos, la instalación y la puesta en funcionamiento del sistema. Además, se detalla la guía de uso para el usuario.

8.1. Requisitos

En esta sección se describen los requisitos, tanto *hardware* como *software*, necesarios para poner en marcha el sistema.

8.1.1. Requisitos hardware

Los requisitos *hardware* mínimos para el funcionamiento del sistema son los siguientes:

- CPU: La familia del preprocesador utilizada es indiferente, pero la frecuencia de reloj debe ser igual o superior a 1 GHz.
- Memoria RAM: 512 MB (recomendable 1 GB).
- Disco duro: 20 GB.

8.1.2. Requisitos software

Los requisitos *software* mínimos para el correcto funcionamiento del sistema son los siguientes:

- Sistema operativo GNU/Linux ya que la mayoría de herramientas empleadas han sido diseñadas para estos sistemas. Se recomienda el uso de una distribución basada en *Debian*, tal como *Ubuntu* o *Kubuntu*, que ha sido la

usada durante el desarrollo de nuestro sistema. Es importante instalar la versión de la distribución elegida para arquitecturas de 32 bits, ya que el sistema no funciona con un equipo que tenga instalado un sistema operativo con una versión de 64 bits.

- Compilador de C, se recomienda el uso de *gcc*.
- Librerías necesarias para que el sistema pueda funcionar:
  - *Glib-2.0* (2.14.1) que nos proporciona una gran cantidad de estructuras de datos y además mejora la portabilidad entre diferentes plataformas.
  - *libxml2* (2.6.30) para el manejo de los archivos de configuración XML.
  - *pkg-config* (0.22) que se utiliza como herramienta de ayuda para indicar las rutas a las cabeceras y/o librerías durante la fase de compilación.
  - *flex* (2.5.33) es un programa utilizado para el reconocimiento de patrones léxicos en un texto.
- *Automake/autoconf* tools.

8.2. Compilación

8.2.1. Compilación de la librería del etiquetador MrTagoo

Para compilar los ficheros del etiquetador el usuario debe abrir un terminal, situarse en el directorio raíz de la librería y ejecutar las siguientes ordenes:

```
aclocal
autoheader
autoconf
automake --add-missing
```

A continuación, se debe ejecutar el fichero configure. Se puede especificar una ruta de instalación de la librería mediante el uso del parámetro – prefix:

```
./configure [--prefix=<ruta_instalacion>]
```

Por ejemplo: `./configure --prefix=/home/vanesa/MrTagoo`. Por defecto, si no se incluye el parámetro prefix, la instalación se realiza en el directorio /usr/local.

Finalmente, se ejecuta el comando:

```
make install
```

Para poder utilizar esta librería, se debe exportar la ruta del fichero MrTagoo.pc que se encuentra en `<ruta_instalacion>`/share/confdata/pkgdata. Existen dos opciones diferentes para realizar este paso:

1. Copiar el fichero al directorio por defecto donde se encuentran los ficheros .pc para *pkg-config*, normalmente /usr/lib/pkgconfig.

2. Modificar la variable de entorno PKG_CONFIG_PATH para que incluya el directorio donde se encuentra el fichero MrTagoo.pc:

   ```
   export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:<ruta_instalacion>
   /share/confdata/pkgdata/
   ```

8.2.2. Instalación del preprocesador

Dado que el sistema desarrollado en este proyecto se integra dentro del preprocesador, la instalación se realiza de forma conjunta. Para llevar a cabo la instalación del programa se debe abrir un terminal o consola, situarse en el directorio raíz del preprocesador y ejecutar las siguientes ordenes:

```
aclocal
autoheader
autoconf
automake --add-missing
```

A continuación, se debe ejecutar el fichero configure. Se puede especificar una ruta de instalación de la librería mediante el uso del parámetro – prefix:

```
./configure [--prefix=<ruta_instalacion>]
```

Por ejemplo: `./configure --prefix=/home/vanesa/preprocesador`. Por defecto, si no se incluye el parámetro prefix, la instalación se realiza en el directorio /usr/local.

Para terminar la instalación se ejecuta el comando:

```
make install
```

```
-<noticia>
   <oración>A nena está cansad</oración>
   <oración>O neno xofa á pelota</oración>
   <oración> A árvore vai secar</oración>
   <oración>Ásveces vou a comer a cass</oración>
 </noticia>
```

Figura 8.1: Ejemplo del formato de un fichero XML a etiquetar.

8.3. Ejecución del sistema

La ejecución del sistema puede realizarse sobre un único fichero (bien sea en texto plano o bien en formato XML), sobre un directorio en el que se etiquetarán todos los ficheros con extensión .xml o introduciendo una frase por teclado desde la ventana del terminal.

Si los ficheros de entrada son de tipo XML deben estar *bien formados*, esto significa que deben cumplir las reglas sintácticas básicas de dicho formato. El proceso de etiquetación-corrección se aplicará al contenido de los elementos `<oración>` que forman parte del fichero de entrada. En la figura 8.1, se muestra un fichero XML de ejemplo con cuatro oraciones a etiquetar y corregir.

Para invocar el sistema se debe abrir un terminal y situarnos en el directorio `<ruta_instalacion>`/bin y escribir la siguiente orden:

```
integrator [opciones] <directorioConfiguración>
```

8.3.1. Directorio de configuración

En la sentencia dónde se invoca el sistema, `<directorioConfiguración>` es la ruta del directorio en la que se buscará el fichero de configuración "config.xml". En este archivo se indican las rutas donde se pueden encontrar los ficheros que necesita la aplicación. El usuario puede modificar la ubicación de cualquiera de ellos escribiendo la ruta absoluta (empezara por un /) o la ruta relativa al directorio de configuración. El contenido por defecto de este fichero se lista a continuación.

```
<configuration>
  <language>gal</language>
  <file_names>
     <tag_inference>tag_inference.txt</tag_inference>
     <info_numerals>info_numerals.txt</info_numerals>
     <enclitic_verb_rules>enclitic_verbs_rules.xml</enclitic_verb_rules>
     <enclitic_pronoun_rules>enclitic_pronouns_rules.xml</enclitic_pronoun_rules>
     <contractions>contraccions.txt</contractions>
     <main_lexicon>lexiconPrincipal.bin</main_lexicon>}
     <clitic_verb_lexicon>lexiconVerbosCliticos.bin</clitic_verb_lexicon>
     <clitic_lexicon>lexiconCliticos.bin</clitic_lexicon>
     <clitic_combination_lexicon>lexiconCombinacionsCliticos.bin</clitic_combination_lexicon>
     <uncertain_idiom_lexicon>lexiconLocucionsInseguras.bin</uncertain_idiom_lexicon>
     <certain_idiom_lexicon>lexiconLocucionsSeguras.bin</certain_idiom_lexicon>
     <proper_noun_lexicon>lexiconPropios.bin</proper_noun_lexicon>
     <trained_proper_noun_lexicon>lexiconPropiosEntrenados.bin</trained_proper_noun\_lexicon>
     <reversed_main_lexicon>lexiconPrincipalInvertido.bin</reversed_main_lexicon>
     <acronym_lexicon>lexiconSiglas.bin</acronym_lexicon>}
     <abbrev_lexicon>lexiconAbreviaturas.bin</abbrev_lexicon>}
     <hmm_model_name>training_corpus/corpus</hmm_model_name>}
  <file_names>
</configuration>
```

A continuación se explican cada uno de los elementos del fichero de configuración:

- `<language>`: Lenguaje empleado para la etiquetación/corrección.

- `<tag_inference>`: Fichero que contiene información relativa a las etiquetas recibidas por un nombre propio.

- `<info_numerals>`: Fichero que contiene información relativa a las etiquetas recibidas por un numeral.

- `<enclitic_verb_rules>`: Fichero de reglas de segmentación de formas enclíticas.

- `<enclitic_pronoun_rules>`: Fichero de reglas de filtrado de pronombres enclíticos.

- `<contractions>`: Fichero de contracciones que contiene cada una de las contracciones junto con los elementos que la componen, con su etiqueta y lema individual.

- `<main_lexicon>`: Lexicón principal, que incluye para cada forma sus posibles etiquetas y lemas.

- `<clitic_verb_lexicon>`: Lexicón de formas verbales con posibilidad de ir acompañadas de pronombres enclíticos.

- `<clitic_lexicon>`: Lexicón de pronombres enclíticos. Es una lista de los distintos pronombres enclíticos que pueden aparecer. Algunos de los que aquí se encuentran son en realidad contracciones de dos (por ejemplo llo = lle+o). En estos casos, la etiqueta que lo acompaña es la del pronombre de la izquierda de la contracción [1].

- `<clitic_combination_lexicon>`: Lexicón de combinaciones de pronombres enclíticos válidas.

- `<uncertain_idiom_lexicon>`: Lexicón de loeuciones inseguras. Estas locuciones representan ambigüedades segmentales, ya que puden fusionarse o no, dependiendo del contexto.

- `<certain_idiom_lexicon>`: Lexicón de locuciones seguras. Lista de aquellas locuciones que siempre se fusionan en una unidad léxica.

- `<proper_noun_lexicon>`: Lexicón de nombre propios. Una lista de nombres propios simples y compuestos (multipalabra) con sus correspondientes etiquetas y lemas.

- `<trained_proper_noun_lexicon`: Lexicón de nombres propios entrenados, que incluye nombres propios compuestos de varias palabras, pero sin etiquetas ni lemas. Este lexicón se usa para detectar nombres propios dentro del texto. Una vez detectado un nombre propio, se usa el lexicón de propios para etiquetarlo.

- `<reversed_main_lexicon>`: Lexicon principal invertido, que permite desde un lema y una etiqueta, obtener su forma/s asociada.

- `<acronym_lexicon>`: Lexicón de siglas.

- `<abbrev_lexicon>`: Lexicón de abreviaturas.

- `<hmm_model_name>`: Es el nombre del MMO entrenado.

8.3.2. Opciones de ejecución

En este apartado, se describirán las posibles opciones de ejecución del sistema completo. Si se invoca a *integrator*, sin opciones, el programa esperará a que se le introduzca una frase por teclado para procesarla. Para modificar el comportamiento por defecto del programa a las necesidades del usuario se pueden establecer las siguientes opciones:

[1] También aparecen algunos alomorfos en sus dos variantes (por ejemplo, -la y -a).

- `-d <directorio>`: Con ésta se indica al programa que procese todos los ficheros con extensión .xml que encuentre en el directorio indicado.

- `-f <fichero>`: Con ésta se indica al programa que procese el fichero que se indica.

- `-m <modelo>`: Con ésta se indica al programa la ruta de los ficheros que conforman el modelo MMO. Esta opción prevalece sobre lo que se indique en el fichero XML de configuración.

- `-i <format>`:Con esta opción se indica al programa el formato del fichero o ficheros de entrada. Los formatos permitidos son "plain" o "xml". Sólo tiene sentido acompañada de la opción -d o -f. Por defecto se toma el formato de entrada XML.

- `-a` : Si aparece esta opción (sólo tiene sentido cuando se etiqueta una o varios ficheros XML), en el fichero final etiquetado se incluyen todas las opciones de etiquetación, señalando cuales son las que ha seleccionado el etiquetador. En caso contrario, sólo aparece el resultado final del etiquetador.

- `-p`: Si aparece esta opción, sólo se aplica el preprocesador. Se utiliza cuando la oración a etiquetar se introduce directamente por la entrada estándar. Como resultado, produce la cadena con todas las opciones de segmentación y etiquetación en la salida estándar.

- `-b <archivo_configuracion_corrector>`: Con ésta se indica que además de la
etiquetación se añade al programa la funcionalidad de la corrección de palabras desconocidas. Mediante el parámetro `<archivo_configuracion_corrector>` se establece la ruta del archivo de configuración del sistema de corrección. El usuario puede modificar la ubicación de cualquiera de ellos escribiendo la ruta absoluta (empezará por un /) o la ruta relativa al directorio de configuración.

El contenido por defecto del archivo de configuración del corrector es el siguiente:

```
<configuration>
    <word_incorrect>no</word_incorrect>
    <regions>lexiconPrincipal.reg</regions>
    <search_type>g</search_type>
    <thresholds>
        <consecutives>2</consecutives>
        <global>2</global>
        <local>
            <deletion>2</deletion>
            <insertion>2</insertion>
            <transposition>2</transposition>
            <replacement>2</replacement>
        </local>
    </thresholds>
</configuration>
```

A continuación, se explicará el significado de cada uno de los elementos de configuración:

- `<word_incorrect>`: Indica si se incluye la palabra incorrecta como alternativa de corrección ("yes") o no se incluye ("no").
- `<regions>`: Fichero donde se almacenan las regiones. Se encuentra dentro del directorio de configuración que se pasa como parámetro para invocar al programa. Si la búsqueda es de tipo global no es necesario indicarlo.
- `search_type`: Indica si la técnica empleada para realizar la corrección ortográfica es de tipo global ("g") o de tipo regional ("r").
- `<tresholds>`: Establecen los umbrales de error máximos permitidos para encontrar correcciones, es decir, el número de caracteres que difieren de la palabra original.
    - `<consecutives>`: Número máximo de errores de edición consecutivos permitidos.
    - `<global>`: Número de errores de edición máximo permitidos.
    - `<local>`: Umbrales para la búsqueda regional.
        - `<deletion>`: Número de errores de borrado máximo permitidos.
        - `<insertion>`: Número de errores de inserción permitidos
        - `<transposition>`: Número de errores de transposición permitidos.
        - `<replacement>`: Número de errores de reeemplazo permitidos.

Ejemplos de ejecución:

- `$./integrator ../share/confdata/galician/`
  Pide una oración por teclado para etiquetarla y muestra el resultado por pantalla.

- `$./integrator -d /home/vanesa/pruebasXML/ ../share/confdata/galician/`
  Etiqueta todos los ficheros .xml que estén en el directorio /home/vanesa/pruebasXML.

- `$./integrator -a -d /home/vanesa/pruebasXML/ ../share/confdata/galician/`
  Igual que el anterior pero incluyendo todas las opciones de etiquetación.

- `$./integrator -f /home/vanesa/pruebasXML/prueba1.xml ../share/confdata/galician/`
  Etiqueta unicamente el fichero /home/vanesa/pruebasXML/prueba1.xml.

- `$./integrator -f /home/vanesa/pruebasXML/prueba2.txt ../share/confdata/galician/`
  Etiqueta unicamente el fichero /home/vanesa/pruebasXML/prueba1.txt y está en formato texto plano.

En la figura 8.2 se muestra la captura de una ejecución del programa en la que se activa la corrección para las palabras desconocidas (`$./integrator - b spellchecker.xml ../share/confdata/galician/`). Como no se han indicado más opciones el programa está esperando a que se introduzca una frase por teclado para procesarla.

```
vanesa@vanesa-desktop:~/PFC/Codigo_mrtgp_Original/Preprocesador/bin
Sesión  Editar  Vista  Marcadores  Preferencias  Ayuda
vanesa@vanesa-desktop:~/PFC/Codigo_mrtgp_Original/Preprocesador/bin$ ./integrator -b
spellchecker.xml ../share/confdata/galician/
Ola
Loading CONFIG_FILE ../share/confdata/galician/ ...
Config_file:spellchecker.xml
Ruta :../share/confdata/galician/spellchecker.xml
Loading ../share/confdata/galician/config.xml ...
Loading ../share/confdata/galician/tag_inference.txt ...
Loading tag inference rules...Loading ../share/confdata/galician/info_numerals.txt ..
.
Loading information about numerals: Loading ../share/confdata/galician/enclitic_verbs
_rules.xml ...
Loading ../share/confdata/galician/enclitic_pronouns_rules.xml ...
Loading ../share/confdata/galician/contraccions.txt ...
Loading the contractions file...
Loading ../share/confdata/galician/lexiconPrincipal.bin ...
Loading ../share/confdata/galician/lexiconVerbosCliticos.bin ...
Loading ../share/confdata/galician/lexiconCliticos.bin ...
Loading ../share/confdata/galician/lexiconCombinacionsCliticos.bin ...
Loading ../share/confdata/galician/lexiconLocucionsInseguras.bin ...
Loading ../share/confdata/galician/lexiconLocucionsSeguras.bin ...
Loading ../share/confdata/galician/lexiconPropios.bin ...
Loading ../share/confdata/galician/lexiconPropiosEntrenados.bin ...
Loading ../share/confdata/galician/lexiconPrincipalInvertido.bin ...
Loading ../share/confdata/galician/lexiconAbreviaturas.bin ...
Loading ../share/confdata/galician/lexiconSiglas.bin ...
Lexicons loaded successfully...
Loading model ../share/confdata/galician/training_corpus/corpus:
  Loading tag-set ../share/confdata/galician/training_corpus/corpus.tag
    (375 tags read)
  Loading n-grams ../share/confdata/galician/training_corpus/corpus.321
    (376 uni-grams, 6175 bi-grams, 30174 tri-grams read)
  Loading lexicon ../share/confdata/galician/training_corpus/corpus.lex
    (737716 words read)
  Loading guesser ../share/confdata/galician/training_corpus/corpus.suf
    (1123451 suffixes read)
Terminal
```

Figura 8.2: Captura de ejecución de un programa con el corrector activado.

8.3.3. Resultados obtenidos

En la figura 8.3 se introduce para corregir la frase “A pesarde todo non volveeu aparecer“ y en la figura 8.4 se muestran los resultados para dicha frase. Como se puede comprobar en este caso las correcciones y etiquetaciones realizadas son las esperadas, ”A pesar de todo non volveu aparecer.“.

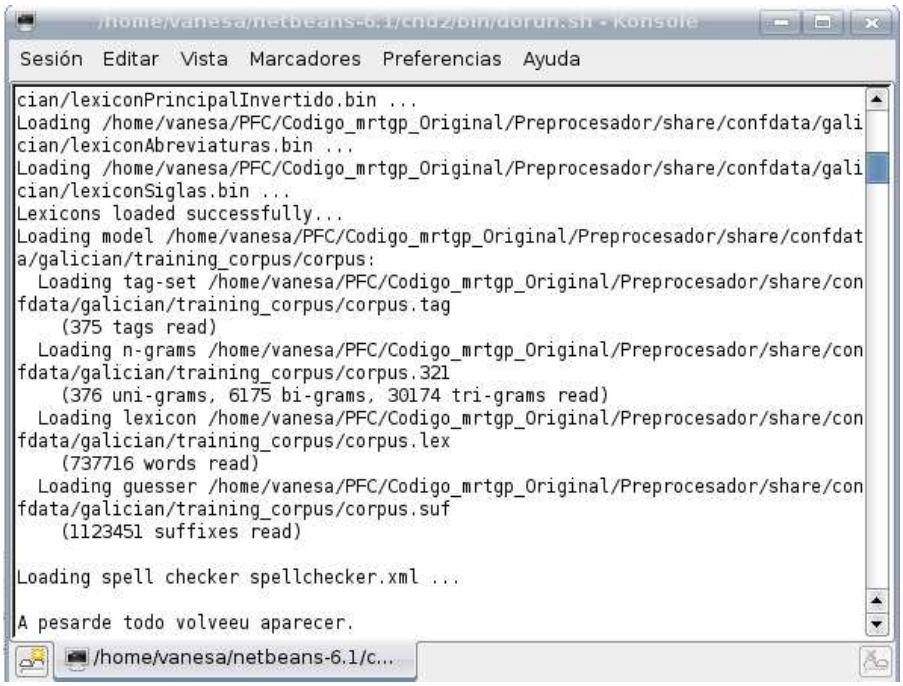

Figura 8.3: Captura de la ejecución de la frase “A pesarde todo non volveeu aparecer“.

Además de mostrar por pantalla el resultado final de la etiquetación, si el archivo de entrada es de tipo XML los resultados se almacenan en otro archivo XML. A continuación, se mostrarán dos ejemplos del formato utilizado por el archivo XML de salida. Estos archivos se guardarán en el mismo directorio que en el que se encuentre el fichero XML de origen, el nombre utilizado es el mismo que el del fichero de entrada al que de le añadirá la extensión .etq, por ejemplo para el archivo *prueba1.xml* el fichero de salida etiquetado se denominará *prueba1.xml.etq*.

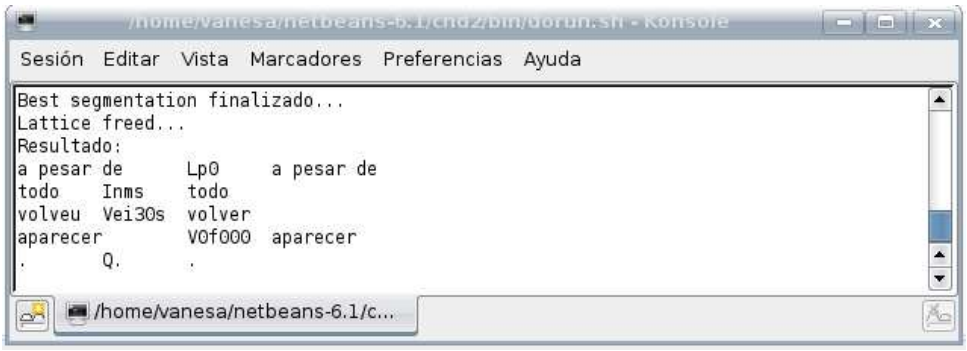

Figura 8.4: Resultado de la ejecución de la frase "A pesarde todo non volveeu aparecer".

```
<noticia>
    <oración>
        <expresión>A nena está cansad</expresión>
        <análise>
            <análise_unidade>
                 <unidade>A</unidade>
                 <constituínte>
                     <forma>a</forma>
                     <etiqueta>Ddfs</etiqueta>
                     <lema>o</lema>
                 </constituínte>
            </análise_unidade>
            <análise_unidade>
                <unidade>nena</unidade>
                <constituínte>
                    <forma>nena</forma>
                    <etiqueta>Scfs</etiqueta>
                    <lema>neno</lema>
                </constituínte>
            </análise_unidade>
            <análise_unidade>
                <unidade>está</unidade>
                <constituínte>
                    <forma>está</forma>
                    <etiqueta>Vpi30s</etiqueta>
                    <lema>estar</lema>
                </constituínte>
            </análise_unidade>
            <análise_unidade>
                <unidade>cansad</unidade>
                <constituínte>
                    <forma>cansan</forma>
                    <etiqueta>Vpi30p</etiqueta>
                    <lema>cansar</lema>
                </constituínte>
            </análise_unidade>
      </análise>
    </oración>
    .........
<\noticia>
```

Aunque pueda resultar intuitivo, se explicará el significado de las etiquetas empleadas para representar la salida . Las etiquetas `<oración>` y `</oración>` encierran el análisis completo de una frase. Para cada frase se guarda su forma original entre las etiquetas `<expresión>` y `</expresión>` y el análisis realizado de la frase entre las etiquetas `<análise>` y `</análise>`. Estas a su vez están formadas por elementos de tipo `<analise_unidade>`, cada uno de ello contiene el análisis para una unidad léxica.

El análisis de una frase, encerrado entre las etiquetas `<análise_unidade>` y `</análise_unidade>`, se compone por los elementos que se listan a continuación:

- `<unidade>`: Es la palabra o conjunto de palabras (por ejemplo en el caso de una locución) que se analiza, es decir, la forma en la que aparece en la frase original.

- `<constituinte>`: Existe un elemento de este tipo por cada unidad en la que se pueda descomponer la palabra analizada. Por ejemplo, las contracciones se representan con un elemento `<constituinte>` para cada componente de la contracción. A su vez, este elemento está formado por los siguientes:

  - `<forma>`: Es la palabra tal y como se escribe. Si se la palabra original (encerrada entre las etiquetas `<unidade>` y `</unidade>`) era desconocida, la forma será su corrección.
  - `<etiqueta>`: Es la categoría morfosintáctica a la que pertenece la palabra.
  - `<lema>`: Es la forma canónica de la palabra. Todas las palabras derivadas de una misma comparten el mismo lema, normalmente coinciden con las entradas en un diccionario convencional.

En caso de activar la opción *-a* en la ejecución del programa el archivo XML de salida mostrará todas las opciones de etiquetación, así como las opciones de corrección, indicando en ambos casos cúal ha sido la opción escogida. Para ello, se ampliará el formato arriba descrito de forma que cada elemento `<analise_unidade>` tenga la siguiente estructura:

```
<análise>
     <análise_unidade>
         <unidade>reaciones</unidade>
         <alternativas>
             <alternativa válido="si">
                 <constituínte>
                     <forma>reacciones</forma>
                     <etq_lema válido="si">
                         <etiqueta>Vps20s</etiqueta>
                         <lema>reaccionar</lema>
                         </etq_lema>
                 </constituínte>
             </alternativa>
             <alternativa>
                 <constituínte>
                     <forma>relaciones</forma>
                     <etq_lema>
                         <etiqueta>Vps20s</etiqueta>
                         <lema>relacionar</lema>
                         </etq_lema>
                 </constituínte>
             <alternativa>
                  <constituínte>
                      <forma>raciones</forma>
                     <etq_lema>
                         <etiqueta>Vps20s</etiqueta>
                         <lema>racionar</lema>
                         </etq_lema>
                 </constituínte>
            </alternativa>
        </alternativas>
    </análise_unidade>
</análise>
```

Índice de figuras

1.1. Modelo de Markov que representa la evolución del clima 19
1.2. Representación de un MMO 21
1.3. Enrejado simplificado para la etiquetación de una frase de T palabras 22
1.4. Los enrejados no permiten representar segmentaciones ambiguas . . . 24
1.5. Segmentaciones ambiguas representadas en un retículo 24
1.6. Estructura del preprocesador morfológico [4]. 26
1.7. Antes de aplicar el algoritmo de minimización 28
1.8. Después de aplicar el algoritmo de minimización 28
1.9. Modelización compacta de un diccionario [11]. 29
1.10. Cálculo de regiomes sobre un AF 35
1.11. Funcionamiento general de las técnicas de corrección 37
1.12. Entorno de etiquetación actual 37
1.13. Esquema general del funcionamiento del proyecto. 38

3.1. Fases del proyecto en la planificación inicial. 48
3.2. Diagrama de Gantt para la planificación inicial. 48
3.3. Fases del proyecto en la planificación real. 49
3.4. Diagrama de Gantt para la planificación real. 50

5.1. Ejemplo de una representación para una frase. 60
5.2. Diagrama de casos de uso global. 61
5.3. Diagrama de casos de uso para *Aplicar corrector*. 62
5.4. Diagrama de secuencia del análisis: Aplicar corrección 63
5.5. Diagrama de colaboración del análisis: Aplicar corrección 64
5.6. Diagrama de actividades del análisis: Aplicar corrección 64
5.7. Diagrama de secuencia del análisis: Buscar corrección global 66
5.8. Diagrama de colaboración del análisis: Buscar corrección global . . . 67
5.9. Diagrama de actividades del análisis: Buscar corrección global 67
5.10. Diagrama de secuencia del análisis: Buscar corrección regional 69
5.11. Diagrama de colaboración del análisis: Buscar corrección regional . . 70
5.12. Diagrama de actividades del análisis: Buscar corrección regional . . . 70

5.13. Diagrama de secuencia del análisis: Enlazar correcciones 72
5.14. Diagrama de colaboración del análisis: Enlazar correcciones 73
5.15. Diagrama de actividades del análisis: Enlazar correcciones 74
5.16. Diagrama de secuencia del análisis: Revisar nuevas contracciones . . . 76
5.17. Diagrama de colaboración del análisis: Revisar nuevas contracciones . 77
5.18. Diagrama de actividades del análisis: Revisar nuevas contracciones . . 78
5.19. Diagrama de secuencia del análisis: Revisar nuevas expresiones 80
5.20. Diagrama de colaboración del análisis: Revisar nuevas expresiones . . 80
5.21. Diagrama de actividades del análisis: Revisar nuevas expresiones . . . 81
5.22. Diagrama de secuencia del análisis: Configurar corrección 83
5.23. Diagrama de colaboración del análisis: Configurar corrección 83
5.24. Diagrama de clases del análisis. 84

6.1. Diagrama de clases del diseño. 86
6.2. Diagrama de clases parcial. 87
6.3. Representación de una frase con dos errores consecutivos. 89
6.4. Representación de una frase con dos errores con una palabra correcta entre ellos. 89
6.5. Representación de una frase con tres errores separados por una palabra correcta. 89
6.6. Representación de una frase con varios errores. 90
6.7. Estructura recibida por el Sistema Corrector. 91
6.8. Estructura después de haber aplicado el Sistema Corrector. 91
6.9. Representación de “ó cabbo“ antes y después de llamar al módulo de detección de locuciones. 93
6.10. Representación de la evolución de la estructura de la frase en el Sistema corrector. 94
6.11. Diagrama de secuencia del diseño: Aplicar corrección 102
6.12. Diagrama de secuencia del diseño: Buscar corrección global 103
6.13. Diagrama de secuencia del diseño: Buscar corrección regional 104
6.14. Diagrama de secuencia del diseño: Enlazar correcciones 105
6.15. Diagrama de secuencia del diseño: Revisar nuevas contracciones . . . 106
6.16. Diagrama de secuencia del diseño: Revisar nuevas expresiones 107

7.1. Diagrama de componentes 110
7.2. Ejemplo de utilización del fichero de configuración *s_checker.xml*. . . 111

8.1. Ejemplo del formato de un fichero XML a etiquetar. 120
8.2. Captura de ejecución de un programa con el corrector activado. . . . 126
8.3. Captura de la ejecución de la frase “A pesarde todo non volveeu aparecer“. 127

8.4. Resultado de la ejecución de la frase “A pesarde todo non volveeu aparecer“. 128

Bibliografía

[1] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 598–603, 1997.

[2] Daciuk, Mihov, Watson, and Watson. Incremental construction of minimal acyclic finite-state automata. In *Computational Linguistics, MIT Press for the Association for Computational Linguistics*. 2000.

[3] F. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3), March 1964.

[4] J. Graña Gil, F. M. Barcala Rodríguez, and J. Vilares Ferro. Etiquetación robusta del lenguaje natural: preprocesamiento y segmentación. *Procesamiento del Lenguaje Natural*, 27:173–180, 2001.

[5] Jorge Graña Gil. *Técnicas de Análisis Sintáctico Robusto para la Etiquetación del Lenguaje Natural*. PhD thesis, Universidad de A Coruña, September 2000.

[6] Jorge Graña Gil, Miguel Alonso Pardo, and Manuel Vilares Ferro. A Common Solution for Tokenization and Part-of-Speech Tagging: One-Pass Viterbi Algorithm vs. Iterative Approaches. *Lecture Notes in Artificial Intelligence*, 2448:3–10, 2002.

[7] Jorge Graña Gil, F. M. Barcala Rodríguez, and M. A. Alonso Pardo. Compilation methods of minimal acyclic finite-state automata for large dictionaries. *Lecture Notes of Computer Science*, 2494:135–148, 2001.

[8] S. Klein and R.F. Simmons. A computational approach to grammatical coding of English words. *Journal of the Association for Computing Machinery*, 10:334–347, 1963.

[9] Leslie Lamport. *LaTeX: A Document Preparation System*. 1986.

[10] A. A. Markov. An example of statistical investigation in text of Eugene Onyegin illustrating coupling of tests in chains. In *Proceedings of the Academy of Sciences*, volume 7, pages 153–162, St. Petersburg, 1913.

[11] Juan Otero Pombo. *Análisis Léxico Robusto*. PhD thesis, Universidade de Vigo.

[12] A. Savary. Typographical nearest-neighbor search in a finite-state lexicon and its application to spelling correction. In *CIAA: International Conference on Implementation and Application of Automata, LNCS*, 2001.

[13] Herbert Schildt. *Turbo C/C++. Manual de Referencia*. 1991.

[14] W.S. Stolz, P.H. Tannenbaum, and Carstensen. A stochastic approach to the grammatical coding of English. *Communications of the ACM*, 8(6):399–405, 1965.

[15] Manuel Vilares Ferro, Juan Otero Pombo, and Jorge Graña Gil. Regional finite-state error repair. *Lecture Notes in Computer Science (LNCS)*, pages 269–280, 2004.

[16] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Trans. Information Theory*, IT-13:260–269, April 1967.

[17] Robert A. Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5):265–268, 1974.

Índice alfabético

afdan, 27, 30, 58, 100

corpus, 17, 27

distancia de edición, 31, 33

enrejado, 22, 23
etiquetación, 16

lema, 27
lexicón, 16, 20

mmo, 19, 21, 27

pln, 15

retículo, 23

Savary, 30, 31, 35
savary, 65, 98
sax, 42, 82

uml, 44, 58

Viterbi, 18, 21, 24, 36, 54

xml, 42, 120